Levenshtein Distance Closest Pair

Nick Kulaga, Or Broder

February 26, 2025

Abstract

This study is an edition to a previous study "Efficient Closest Pair Identification in Binary Datasets via Multi-Hash LSH". This current study tackles a similar problem under a different distance metric known as Levenshtein distance.

1 Introduction

This paper is an extension of our previous work, *"Efficient Closest Pair Identification in Binary Datasets via Multi-Hash LSH"*, where we addressed the problem of finding the closest pair of binary strings in regard to Hamming distance. In that work, we presented both brute-force and multi-hash Locality-Sensitive Hashing (LSH) methods and compared their performance. In this paper we want to explore a similar question of finding the closest pair in regard to Levenshtein distance. The Levenshtein distance between two strings is defined as the minimum number of operations required to edit one string into the other using deletions, substitutions and insertions. Algorithms trying to calculate this distance efficiently have a wide range of applications from correcting spelling errors in text editing software's to correcting obstructed code words through obstructed mediums.

2 Problem Definitions

Given a set of m binary strings, each of length n, the objective is to determine the pair with the minimum Levenshtein distance, the number of edit operations (insertions, deletions and substitutions) needed to transform one of the words to the other.

3 Levenshtein distance discussion

Our previous paper discussed Hamming distance as the criterion to the closest pair in a set. We did not elaborate on the characteristics of Hamming distance, which can be

interpreted as a special case of metric space, and the distance being measured under L_1-norm. The points in space are restricted s.t for $x \in \mathbb{R}^n$: $\forall i, s.t\ 1 \leq i \leq n \quad x_i \in \{0,1\}$. This is a well-established and widely known distance metric. In contrast, the Levenshtein distance requires a more complex method of calculation, which increases the overall complexity of the problem. To illustrate this further, if we had restricted the editing operations to substitutions only, the Hamming and Levenshtein distances would have been identical, as any discrepancy between bits would be equivalent to a substitution.

4 Calculating Levenshtein distance

4.1 Naïve approach

The straightforward approach to finding the minimum edit distance involves recursively reducing the prefixes of both strings, gradually decreasing their length:

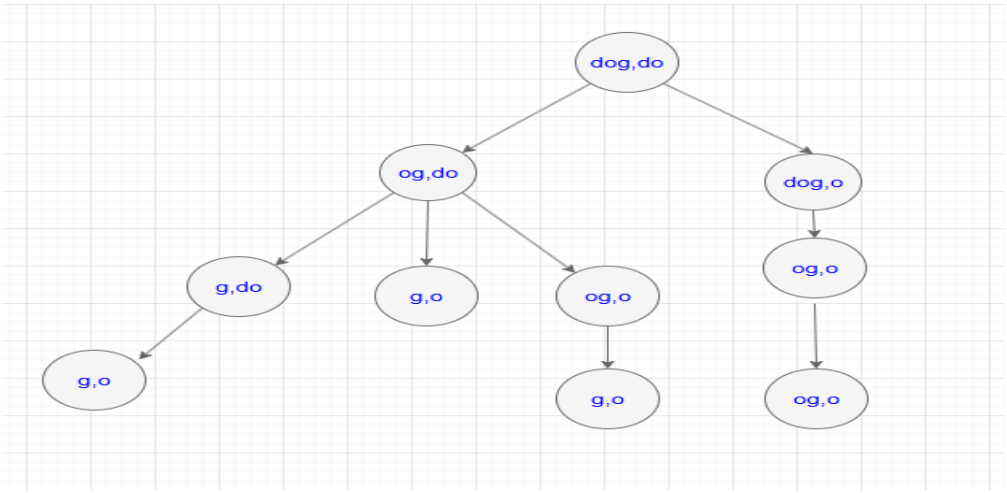
```
int recursiveLevenshteinDistance(string A, string B) {
    if (A.empty()) return B.size();
    if (B.empty()) return A.size();

    string cut_A = A.substr(1);
    string cut_B = B.substr(1);

    if (A[0] == B[0]) {
        return recursiveLevenshteinDistance(cut_A, cut_B);
    } else {
        int insertOp = recursiveLevenshteinDistance(A, cut_B);
        int deleteOp = recursiveLevenshteinDistance(cut_A, B);
        int replaceOp = recursiveLevenshteinDistance(cut_A, cut_B);

        return 1 + min({insertOp, deleteOp, replaceOp});
    }
}
```

The base cases work under the occurrence where turning an empty string to its counterpart requires n insertions where n is the size of the non-empty string.
As this approach accounts for any possible action performed on the given strings, we get a ternary tree-based behaviour, which results in $O(3^n)$ complexity.

4.2 Dynamic programming matrix-based calculation

The following method was introduced in “The String-to-String Correction Problem” by Robert A. Wagner and Michael J. Fischer, 1974.
A major downside to the naïve approach is recalculating the overlapping states. For example (we consider the non-binary case as it is more intuitive) consider the two strings

"dog" and "do". Using the naïve method we get the following calling scheme(excluding pair consisting of at least one empty string):



The simple observation indicates repetition of same function calls with same pair of suffixes. This unnecessary recalculation is very inefficient considering the complexity of finding the ternary tree $O(3^n)$. The method that was mentioned in the paper suggests storing intermediate results to avoid recalculation of already attained values. For the case of strings X, Y s.t $X = x_1x_2x_3, \dots, x_n$, $Y = y_1y_2y_3 \dots y_m$, we build a matrix with $m + 1$ columns and $n + 1$ rows. For i, j s.t $1 \leq i \leq n$+1 and $1 \leq j \leq m + 1$the cell matrix cell $M_{i,j}$ represents the Levenshtein distance between the truncated words $X_i = x_1x_2 \dots x_{i+1}$ and $Y_j = y_1, y_2, \dots, y_j$. For $i = 1$ we get the empty word for X(the same for $j = 1$ regarding Y). By this definition we would get the intended results at $M_{n+1,m+1}$. The implementation is such:

```
int DPlevenshteinDistance(const string &A, const string &B) {
    int m = A.size(), n = B.size();
    vector<vector<int>> dp(m + 1, vector<int>(n + 1));

    for (int i = 0; i <= m; i++) dp[i][0] = i;

    for (int j = 0; j <= n; j++) dp[0][j] = j;

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            int cost = (A[i - 1] == B[j - 1]) ? 0 : 1;
            dp[i][j] = min({ dp[i - 1][j] + 1, dp[i][j - 1] + 1,dp[i - 1][j - 1] + cost
            });
        }
    }
    return dp[m][n];
}
```

Each block in our matrix is filled once and we achieve time complexity of $O(n * m)$. For our intended purpose for binary same length code, we simply get $O(n^2)$.

4.3 Ukkonen's improvement

As mentioned before, one of the common uses of calculating the edit distance is in spelling correction in text editing software. We will get to managing a set of strings under the Levenshtein distance, which in the context of spelling correction, is simply a dictionary. A common assumption done for such software is that the user knows what word he is intending to write and so the edit distance from the words he is typing is relatively low. This assumption can be implied to other code transferring buffers if we can prove or show heuristically that the edit distance between the source and the receiver has a limit and then use this limit to efficiently correct errors. The algorithm was introduced by Esko Ukkonen in the paper "Approximate Matching of Run-Length Compressed Strings by Veli Makinen, Gonzalo Navarro and Esko Ukkonen, 1985.
If we assume that the edit distance of two words is at most k, we can modify the DP approach to skip some of the calculations, essentially leaving some of the matrix cells empty. The algorithm is as follows:

```
int ukkonenLevenshteinDistance(const string& A, const string& B, int k) {
    int m = A.size();
    int n = B.size();

    vector<vector<int>> dp(2, vector<int>(n + 1, 0));
    for (int j = 0; j <= n; j++) {
        dp[0][j] = j;
    }
    for (int i = 1; i <= m; i++) {
        int curr = i % 2;
        int prev = (i - 1) % 2;

        for (int j = max(0, i - k); j <= min(n, i + k); j++) {
            if (A[i - 1] == B[j - 1]) {
                dp[curr][j] = dp[prev][j - 1];
            } else {
                dp[curr][j] = min({dp[prev][j] + 1,    dp[curr][j - 1] + 1,   dp[prev][j - 1] + 1});
            }
        }
    }
    return dp[m % 2][n];
}
```

We simply modified the DP algorithm to check cells which are distant from the diagonal by k. This algorithm achieves time complexity $O(k * \max(m, n))$ and again, for our focused problem $O(k * n)$. Under the assumption that $k \ll n$ we get a great solution of $O(n)$.

5 Finding The Closest Pair Out of a Set

In the previous paper we discussed finding the closest pair out of a set of binary strings regarding Hamming distance. The methodology we introduced as the most efficient was Multi-Hash LSH, which applied hashing on bits entries. The nature of Levenshtein

distance as we discussed, can't be applied on this method out of the "shifting" behaviour of the deletions and insertions operations.

5.1 Brute force

The obvious and brute method we can use to find our closest pair deserves mentioning as an example of a very costly method. This method would include calculating the Levenshtein distance across all strings in the set and finding the minimum distance. Such method would require m^2 calculations for a data set of m strings. Applying the DP method for the distance we would get $O(n^2m^2)$ time complexity.

5.2 Divide and Conquer method

The "divide and conquer" method is a fundamental algorithm introduced by Donald Knuth in his book "The Art of Computer Programming" (1986). This method is commonly used in KNN algorithms to find close sets. As the name suggests, the algorithm splits the sets into two parts, using a middle plane, and looks for the closest in each pair. The algorithm doesn't ignore the possibility of the closest pair being separated by the division, furthermore it utilizes the fact that because we are looking for the minimum distanced pair, an so it limits is cross-group search to data points only in distance smaller than the minimum found in both groups. We would like to apply this method to the Levenshtein distance and sort of modify the time complexity to account for calculating the distance of each pair in $O(n^2)$ instead of the Euclidean of $O(1)$. Unfortunately, we didn't manage to apply this method as finding a plane that would split the data equally in the "editing space" turned out to be very costly.

5.3 BK Tree

The BK tree introduced by W.A Burkhard and R.M Keller in their 1972 paper "Some Approaches to Best-Match File Searching", is a data structure tree that describes a set of unique strings. Each node represents a string in the set, and each arc to another node, has a value which represents the edit distance between them. Using this data structure is a good method for searching a close match of a given word out of an already defined set, which as mentioned before, is a great method for finding close word for a given dictionary, or in the world of biochemistry, finding an original protein sequence out of a damaged one. The time complexity of building the tree from m strings with length n is $O(m^2n)$ and searching a match takes $O(mlogm)$.

6 Conclusion and Discussion

Unfortunately, the unique properties of the Levenshtein distance as opposed to some other distance metrics, which can be calculated using simpler norms (we mentioned the relation between L_1 norm to hamming distance, and L_2 to Euclidean), poses a computational challenge. Furthermore, the cost of finding the direct Levenshtein distance between two strings was shown in the paper "Edit Distance Cannot Be Computed in Strongly Subquadratic Time (unless SETH is false)" by Simon Weber, as the name suggests that under the SETH assumption, there won't be a calculation more efficient than the $O(n^2)$ we showed. The complexity of this distance itself, shows its challenge in our problem of finding the closest pair; even when using data structures that strongly relates to Levenshtein distance such as the BK tree we discussed and others(trie, Aho-Corasick, etc.) their construction and later comparisons, doesn't improve on the straight forward approach at all. However, a strong assumption, as we mentioned, and is commonly used is the max distance assumption for some k.

The second assumption we mentioned was $k \ll n$. Through most of the papers that span on the topic of sub sequences of strings which we read to try find a better solution, the motive for most of them was the study of protein chains and more specifically DNA sequencing. The $k \ll n$ assumption is very practical in this field as the human genome has an approximate of $6 * 10^9$ bits of data. With this large assumption we can also achieve in relation to m a $m \ll n$ relation which will give us

$O(k * n * m^2) = O(n * m^2) = O(n)$. This result indeed seems good but as we shown in such cases we would prefer to rely on perhaps k or m. Unfortunately we couldn't manage to construct such an algorithm or find literature sources that suggest otherwise.

References

[1] V. I. Levenshtein, *Binary Codes Capable of Correcting Deletions, Insertions, and Reversals*, *Soviet Physics Doklady*, vol. 10, pp. 707–710, 1966.

[2] D. H. Wagner and M. J. Fischer, "The String-to-String Correction Problem," *Journal of the ACM (JACM)*, vol. 21, no. 1, pp. 168–173, 1974.

[3] A. V. Aho and M. J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[4] B. Burkhard and R. Keller, "Some Approaches to Best-Match File Searching," *Information Systems*, vol. 4, no. 1, pp. 1–8, 1979.

[5] E. Ukkonen, "On-line Construction of Suffix Trees," *Algorithmica*, vol. 14, no. 3, pp. 249–260, 1995.

[6] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd ed., Addison-Wesley, 1998.