Levenshtein Edit Distance Embedding and De Bruijn Graph

Mohamed ELKHATRI

1 Introduction

DNA is a molecule that contains genetic information determined by a sequence of nucleotides (A,T,G,C). Determining this sequence is useful for research aimed at understanding how organisms live, as well as for identifying, diagnosing, and potentially finding treatments for genetic diseases and virology. However, DNA sequencing techniques only allow for sequencing limited-sized fragments, with a certain error rate, which varies depending on the method. Since these errors are either an insertion, deletion, or substitution, the Levenshtein distance is used to talk about similarity between sequences. We then work on the De Bruijn graph, which, from several readings of fragments, seeks to reconstruct the entire DNA sequence. In addition, several algorithms that take into account sequencing errors require finding similar sequences among a large number of words. This problem has been widely explored and has efficient solutions for the Euclidean distance, so I plan to embed the word space equipped with the Levenshtein distance in a Euclidean space such that the distances are roughly preserved. This will be done using a recurrent neural network that I propose.

2 Notations

- We denote by $\|\cdot\|_2$ or $\|\cdot\|$ the usual Euclidean norm.
- We denote by $t[i]$ the i^{th} letter of t, starting indexing at 0.
- We denote by $t[a:b]$ the subword between positions a and b inclusive.
- We denote by $|x|$ the cardinality of a set or the length of a sequence.

3 Levenshtein distance

Definition 1 (Levenshtein distance)**.** *The Levenshtein distance (denoted by Δ) between two words a and b is the minimal number of characters that need to be deleted, inserted, or replaced in a to obtain b. Formally, the distance between two words is obtained by the recurrence relation:*

$$\Delta(a,b) = \begin{cases} \max(|a|,|b|) & si\ |a|=0\ ou\ |b|=0 \\ \Delta(a^-,b^-) & si\ a[0]=b[0] \\ 1+\min\begin{cases}\Delta(a^-,b)\\ \Delta(a,b^-)\\ \Delta(a^-,b^-)\end{cases} & sinon \end{cases}$$

where m^- is the word m without its first letter.

3.1 Calculation of the Levenshtein distance

3.1.1 Exact calculation

We simply use the recurrence relation in a dynamic programming algorithm. We can then compute the distance between two words with a time complexity of $\mathcal{O}(|a||b|)$, which is rather slow.

		A	C	C	G	A	T
	0	1	2	3	4	5	6
T	1	1	2	3	4	5	5
A	2	1	2	3	4	4	5
C	3	2	1	2	3	4	5
C	4	3	2	1	2	3	4
A	5	4	3	2	2	2	3
T	6	5	4	3	3	3	2

Table 1: Example of Levenshtein distance calculation using dynamic programming

3.2 Problem

This is the problem of the K nearest neighbors under the Levenshtein distance. We have a set of N words. For each word, we seek the K words that are most similar to it under the Levenshtein distance.

3.3 Embedding of (Σ^n, Δ) in $\mathbb{R}^d$

The problem of the K nearest neighbors is already quite complex for the Euclidean distance, so one might imagine that it is even more so for the Levenshtein distance, which is less simple. However, there are several algorithms that are quite effective for the usual Euclidean distance, including the kd-tree structure. I propose to embed the space of words into the Euclidean space such that the distances are more or less preserved.

Definition 2 (Distortion C embedding)**.** *We call a distortion C embedding of a metric space (E, N) into $(\mathbb{R}^d, \|\cdot\|_2)$ an application f:* $\begin{cases} (E,N) \longrightarrow (\mathbb{R}^d, \|\cdot\|_2) \\ m \longmapsto v_m \end{cases}$ *such that:*

$\exists C_1, C_2 > 0$, $\forall (x,y) \in E \times E$, $C_1 C_2 = C$ *and*

$$\tfrac{1}{C_1} N(x,y) \leq \|f(x) - f(y)\| \leq C_2 N(x,y)$$

Remark 1. *An embedding is called isometric if it has a distortion $C = 1$. In this case, the distances are exactly preserved.*

Remark 2. *The existence of an isometric embedding is not guaranteed.*

For the rest of the document, I will work on the metric space of DNA sequences of length n equipped with the Levenshtein distance, i.e., (Σ^n, Δ) where $\Sigma = A, T, G, C$

Theorem 1 (Bourgain's Theorem)**.** *Any metric space of cardinality N can be embedded in $\mathbb{R}^{\mathcal{O}(\log^2 N)}$ with distortion $\mathcal{O}(\log N)$*

We are then guaranteed the existence of an embedding of (Σ^n, Δ) into $\mathbb{R}^{\mathcal{O}(\log^2(4^n))} = \mathbb{R}^{\mathcal{O}(n^2)}$ with distortion $\mathcal{O}(\log(4^n)) = \mathcal{O}(n)$, which is not very interesting. But we will see that the results are much more promising in practice. In order to look for a reasonable embedding, I have decided to design a neural network for this purpose.

4 Neural Network

Most of the articles working on this embedding problem have used a convolutional neural network (CNN). Therefore, I propose a recurrent neural network (RNN) architecture that I try to justify in the following part.

4.1 Data Processing

Definition 3 (Component according to a letter of a word)**.** *For a word $s \in \Sigma^n$, and a letter $c \in \Sigma$, we call the component of s according to c the column vector $I_c(s)$ such that: $I_c(s)_{i,1} = \mathbb{1}[s[i] = c]$ for $0 \leq i < n$*

We generate, for a sequence s, the set of components $I_c(s)$ according to each letter. For example, for:

$$s = \text{ACATC} \longrightarrow \begin{array}{l} I_A(s) = [10100]^\top \\ I_C(s) = [01001]^\top \\ I_G(s) = [00000]^\top \\ I_T(s) = [00010]^\top \end{array}$$

Remark 3. *Any sequence of Σ^n is uniquely defined by 3 of the 4 components, as the 4^{th} can be deduced from the other 3: $I_T(s) = [11...11]^\top - (I_A(s) + I_C(s) + I_G(s))$*

4.2 Network Structure

The structure of the recurrent neural network (RNN) is shown in Figure 1. The network works in 3 steps, where at each step a new component of the sequence is considered (according to A, then C, then G), and then a vector E_i of $\mathbb{R}^d$ is generated which considers only the first few letters of the sequence (for example, E_1 only takes into account the positions of the letters A and C). Furthermore, I omitted the component according to the last letter T due to **Remark 3** in order to speed up the training and computation of image vectors. Thus, E_2 is the final image $f(s)$ of the sequence s.

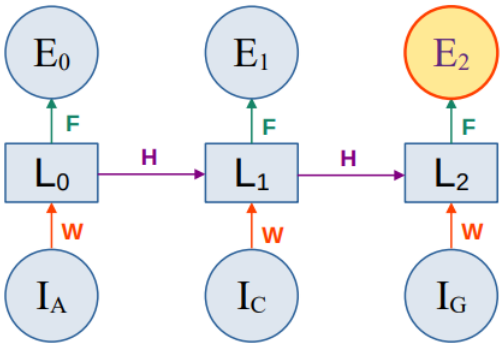

Figure 1: Architecture of RNN

More formally, our network is as follows:

$$\begin{cases} L_i = \tanh(WI_i + HL_{i-1} + B) \\ E_i = \tanh(FL_i) \times \alpha \\ \alpha = \dfrac{3n}{4\sqrt{d}} \end{cases}$$

where:

- We identify I_0, I_1, I_2 with I_A, I_C, I_G respectively
- tanh is chosen as the activation function
- α is the enlargement factor

α was chosen somewhat roughly but gave good results. For an enlargement factor β, the maximum distance between two vectors of the form $\tanh(X) \times \beta$ with $X \in \mathbb{R}^d$ is $2\beta\sqrt{d}$, while the maximum distance between two sequences of Σ^n is n, so for $2\beta\sqrt{d} = n$, we need $\beta = \frac{n}{2\sqrt{d}}$. To allow for a little more space, we replace $\frac{1}{2}$ with $\frac{3}{4}$ and so we set $\alpha = \frac{3n}{4\sqrt{d}}$.

4.3 Learning Algorithm

I_0, I_1, I_2 are still confused with I_A, I_C, I_G. For a sequence s, let s_i be the sequence considering only the first $i+1$ letters:

$$s_i = \sum_{j=0}^{i} (j+1) I_j(s)$$

.

For example, for $s = AGGT$, we have $s_2 = [1\ 3\ 3\ 0]^\top$, which in this case (where $i = 2$) amounts to replacing the alphabet $\Sigma = \{A, C, G, T\}$ with $\Sigma' = \{1, 2, 3, 0\}$, which does not change the Levenshtein distance, consistent with Remark 3. To train the neural network, we generate two sequences $s^{(1)}$ and $s^{(2)}$ each time. Then we pass both through the current network, generating for each 3 vectors $E_0^{(i)}, E_1^{(i)}, E_2^{(i)}$ where $i \in \{1, 2\}$.

Remark 4. *Generating two independent random sequences is to be avoided because the distance between them often takes values close to $\frac{n}{2}$, so similar sequences will not have nearby images in Euclidean space. We then generate the second sequence from the first by modifying a random number of letters.*

We use the function σ as the error function: $\sigma = \sum_{i=0}^{2} w_i(\Delta(s_i^{(1)}, s_i^{(2)}) - \|E_i^{(1)} - E_i^{(2)}\|)^2$ where w_i is the weight of the error at the i^{th} step of the recurrent network. Clearly, the most important error is at the end since it corresponds to the final images of the sequences, so we choose $w_0 = w_1 = 1$ and $w_2 = 3$. Finally, we apply gradient back-propagation with this error function.

$$\boxed{s^{(1)}} \longrightarrow \boxed{\textbf{RNN}} \longrightarrow \boxed{E_0^{(1)}, E_1^{(1)}, E_2^{(1)}}$$

$$\Longrightarrow \sigma = \sum_{i=0}^{2} w_i(\Delta(s_i^{(1)}, s_i^{(2)}) - \|E_i^{(1)} - E_i^{(2)}\|)^2$$

$$\boxed{s^{(2)}} \longrightarrow \boxed{\textbf{RNN}} \longrightarrow \boxed{E_0^{(2)}, E_1^{(2)}, E_2^{(2)}}$$

4.4 Results

I trained the network to embed sequences of lengths $n = 10, 25, 100$ into $\mathbb{R}^{10}, \mathbb{R}^{25}, \mathbb{R}^{100}$, respectively. To test the performance, we generated 50000 pairs of sequences and calculated the average absolute difference between the Levenshtein distance of the two sequences and the Euclidean distance of the two vectors generated by the RNN for each sequence. The following results were obtained:

Length of sequences n	Dimension of Euclidean space d	Average absolute error
10	10	0.99
25	25	2.16
100	100	5.95

Table 1: Average error after training

To examine in more detail the error of our model for the case $n = 100$, $d = 100$, I generated 500 pairs of sequences and graphically represented the Euclidean distance between their images as a function of their Levenshtein distance below:

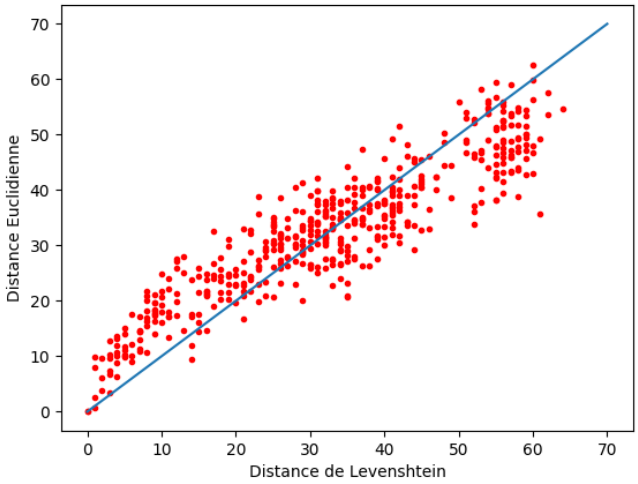

Figure 2: Euclidean distance of generated images vs Levenshtein distance of sequences

4.5 Justification for the choice of network structure

The network generates the image vector of a sequence by considering the positions of a new letter each time. I then showed the following result, which indicates that the Levenshtein distance for an entire sequence is indeed related to the distances for its components.

Theorem 2. *Let* $s^{(1)}$ *and* $s^{(2)}$ *be two sequences of* Σ^n. *Then* $\sum_{i=0}^{3} \Delta(s_i^{(1)}, s_i^{(2)}) \leq 4\Delta(s^{(1)}, s^{(2)})$.

The proof is mainly based on the following lemma:

Lemma 3. *Let* $s^{(1)}$ *and* $s^{(2)}$ *be two sequences of* Σ^n. *Then there exists* $m \in \mathbb{N}$, $(a_i)1 \leq i \leq m \in (\Sigma^+)^m$, *and* $(y_i)1 \leq i \leq m+1$ *and* $(z_i)1 \leq i \leq m+1$ *in* $(\Sigma^*)^{(m+1)}$ *such that:*

- $s^{(1)} = y_1 a_1 y_2 a_2 y_3 ... a_m y m + 1$
- $s^{(2)} = z_1 a_1 z_2 a_2 z_3 ... a_m z_{m+1}$
- $\sum_{i=1}^{m+1} \max(|y_i|, |z_i|) = \Delta(s^{(1)}, s^{(2)})$

A	**C**	**G**	**C**	**C**	**A**	**A**	**T**	**G**	**G**
T	**G**	**C**	**C**	**G**	**T**	**T**	**A**	**T**	**T**

Example illustrating Lemma 3

Lemma 3 also allows for a better visualization of the optimal substitution, insertion, or deletion operations to transform $s^{(1)}$ into $s^{(2)}$. It suffices to keep the a_i, and transform the y_i into z_i by substituting the first $\min(|y_i|, |z_i|)$ letters and inserting or deleting $||y_i| - |z_i||$ letters in the suffix depending on whether $|y_i| \leq |z_i|$ or $|y_i| > |z_i|$. Thus Theorem 2 quickly follows from Lemma 3 since applying these operations also allows each of the $s_i^{(1)}$ to be transformed to $s_i^{(2)}$.

Then, I attempted to show that $\sum_{i=0}^{3} \Delta(s_i^{(1)}, s_i^{(2)}) \geq \Delta(s^{(1)}, s^{(2)})$ by a rather long induction that almost always worked, except for a very specific case that made the proposition false. However, experimentally, I found that it is almost always true.

5 De Bruijn Graph

DNA sequencing techniques only allow for sequencing limited-sized fragments. The goal is to reconstruct the entire sequence from several readings of its fragments. In this section, we work under the following assumptions:

- The readings are error-free.
- The readings are uniformly distributed over the DNA sequence.
- The readings are all of size $L = 100$.

Definition 4 (k-mer)**.** *A k-mer of a certain sequence is a contiguous subsequence of size k of that sequence (example: GCT is a 3-mer in Figure 3).*

Definition 5 (De Bruijn graph of a sequence)**.** *The De Bruijn graph of a sequence s is a directed graph (V, E) whose vertices V are the set of k-mers of s for a fixed k, and edges E are created between every two consecutive k-mers (for example, the 3-mers TAG and AGC are consecutive in the sequence of Figure 3, so there is an edge from TAG to AGC).*

Remark 5. *E is a multiset, meaning that we allow an edge to appear multiple times.*

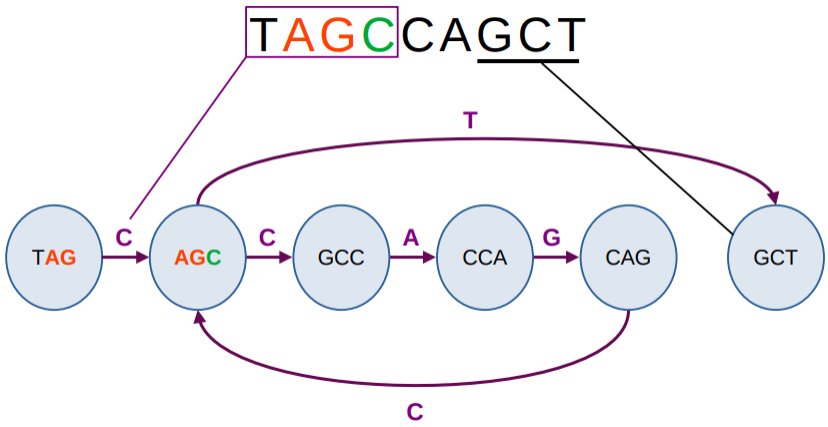

Figure 3: An example of a De Bruijn graph for $k = 3$

Remark 6. *An edge represents the fact of moving to the next k-mer and thus adding a letter to the end (and removing one from the beginning). A subsequence (of length $\geq k$) of a sequence is represented by a path in its De Bruijn graph.*

For example, reading the subword CCAGC in Figure 3 involves following the path: CCA$\longrightarrow$CAG$\longrightarrow$AGC in the graph.

Definition 6 (Eulerian path)**.** *An Eulerian path of a directed graph is a path that passes through all the arcs exactly once per arc.*

Remark 7. *Thus, the entire sequence is represented in the De Bruijn graph by an Eulerian path.*

Definition 7 (De Bruijn graph of a set of reads)**.** *The De Bruijn graph of a set of reads is the union of the De Bruijn graphs of those reads, i.e., the graph $(\bigcup_i V_i, \bigcup_i E_i)$.*

For a large number of reads that cover the sequence well, the De Bruijn graph of those reads contains all the arcs of the De Bruijn graph of the sequence being determined, repeated a certain number of times. For example, for the same sequence as in Figure 3, the two reads in Figure 4 give a De Bruijn graph almost identical to that of the sequence, but with the arc $CCA \longrightarrow CAG$ repeated twice.

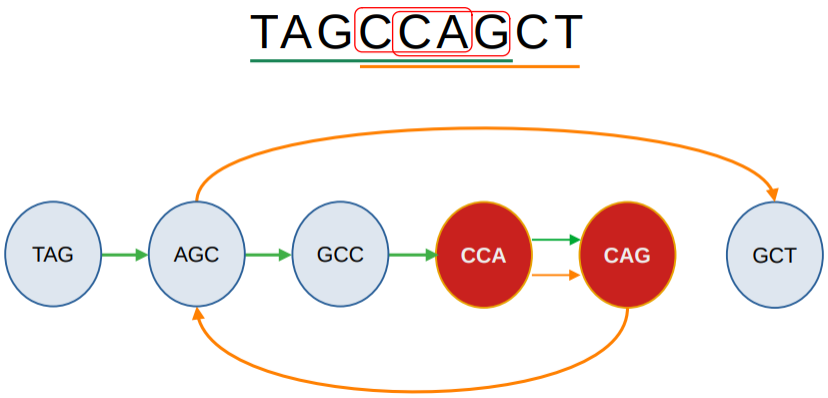

Figure 4: De Bruijn Graph for 2 reads (underlined)

My idea was then to estimate this number of arc repetitions in order to eliminate them and keep only one copy.

Theorem 4. *For a sequence $s \in \Sigma^m$, if N reads of length L are uniformly distributed over the sequence, G is the De Bruijn graph of s for k-mers of length k, and C_i is the expected number of copies of the arc $s[i : i+k-1] \longrightarrow s[i+1 : i+k]$, then:*

$$\forall i \in [L\text{–}1\text{–}k,\ m\text{–}L] :\ \mathbb{E}(C_i) = N\frac{L-k}{m-L+1} = \mathbb{E}(C)$$

Remark 8. *An arc corresponds to two successive k-mers, so the arcs correspond exactly to (k+1)-mers. Therefore, we will say that a read contains an arc if it contains the corresponding (k+1)-mer.*

Proof. We denote E_i the arc s[i:i+k-1] → s[i+1:i+k] and $A_i = \mathbb{1}$the i^{th} read contains E_i.

Then: $C_j = \sum i = 1^N A_i$, but $\mathbb{E}(A_i) = \mathbb{P}(A_i) = \frac{\text{number of possible reads containing } E_i}{\text{number of possible reads}} = \frac{L-k}{m-L+1}$ thus

$$\mathbb{E}(C_j) = \sum_{i=1}^{N} \mathbb{E}(A_i) = N\frac{L-k}{m-L+1}$$

□

The approach I apply is to transform the De Bruijn graph of the reads into a weighted graph where the weight w of an arc is the number of times it is repeated, and thus to try to obtain a unique copy for each arc of the De Bruijn graph of the sequence, we change each weight w to $f(w) = \max([\frac{w}{\mathbb{E}(C)}], 1)$. The idea is basically to divide by rounding by the number of repetitions, and to ensure that no arc disappears, we keep at least one copy if the rounded division is zero.

At this point, the obtained graph should be quite similar to the De Bruijn graph of the sequence we are looking for, and therefore, given the remark, we search for an Eulerian path using the Hierholzer algorithm, which has linear complexity $\mathcal{O}(|E|)$.

Theorem 5. *A directed graph has an Eulerian path if and only if it is connected and all its nodes have the same incoming and outgoing degree, except possibly two nodes that differ by 1.*

All of this seems very rough, yet the results obtained are very satisfying. For a fixed read length $L = 100$, and for a certain sequence length m ranging from 100 to 20000, for 100 different sequences, we generate $N = 2m$ reads, and attempt to recover the sequence by the proposed method, then the error is the average of the Levenshtein distances between each of the 100 initial sequences and the recovered sequence. The results indicate that for a well-chosen k-mer size, the sequence is almost certainly recovered.

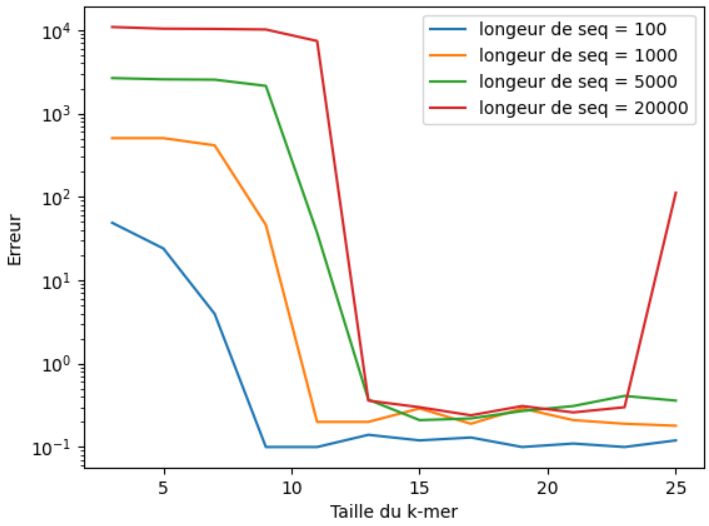

Figure 5: Error (Levenshtein distance) between initial and recovered sequences as a function of the chosen k-mer size

References

[1] Pierre Pericard: Algorithms for the reconstruction of conserved marker sequences in metagenomic data: https://tel.archives-ouvertes.fr/tel-01738687v1/document

[2] Phillip E. C. Compeau, Pavel A. Pevzner, and Glenn Tesler: Why are de Bruijn graphs useful for genome assembly?: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5531759/

[3] Xinyan Dai, Xiao Yan, Kaiwen Zhou, Yuxuan Wang, Han Yang, James Cheng: Convolutional Embedding for Edit Distance: Xinyan DAI, Xiao Yan, Kaiwen Zhou, Yuxuan Wang, Han Yang, and James Cheng. 2020. Convolutional Embedding for Edit Distance. In Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '20), July 25-30, 2020, Virtual Event, China. ACM, New York, NY, USA, 10 pages.

[4] Periklis Papakonstantino: Bourgain's theorem for metric embedding:
http://www.cs.toronto.edu/ avner/teaching/S6-2414/LN2.pdf