Séquençage d'ADN et recherche des séquences similaires

Mohamed ELKHATRI

Numéro de dossier: 53616

1 Introduction

L'ADN est une molécule contenant l'information génétique qui est déterminée par une séquence de nucléotides (A,T,G,C). Déterminer cette séquence est donc utile aussi bien pour les recherches visant à savoir comment vivent les organismes que pour identifier, diagnostiquer et potentiellement trouver des traitements à des maladies génétiques et à la virologie. Or, les techniques de séquençage d'ADN ne permettent que de séquencer des fragments de tailles limitées, avec un certain taux d'erreur en plus, qui varient selon la méthode. Ces erreurs étant soit une insertion, suppression ou substitution, on s'intéresse à la distance de Levenshtein pour parler de similitude entre séquences. On travaille alors sur le graphe de De Bruijn, qui à partir de plusieurs lectures de fragments, cherche à reconstruire la séquence entière d'ADN. En plus, plusieurs algorithmes qui tiennent en compte des erreurs de séquençage nécessitent de trouver les séquences similaires parmi un grand nombre de mots. Ce problème est largement exploré et possède des solutions efficaces pour la distance euclidienne, j'envisage donc de plonger l'espace des mots muni de la distance de Levenshtein dans un espace euclidien tel que les distances sont à peu près conservées. Ceci à l'aide d'un réseau de neuronnes récurrent que je propose.

2 Notations

— On note $\|\cdot\|_2$ ou $\|\cdot\|$ la norme euclidienne usuelle
— On note $t[i]$ la $i^{\text{ème}}$ lettre de t, en commençant l'indexation par 0
— On note $t[a:b]$ le sous-mot entre les positions a et b inclues
— On note $|x|$ le cardinal d'un ensemble ou la taille d'une séquence

3 La distance de Levenshtein

Définition 1 (Distance de Levenshtein)**.** *La distance de Levenshtein (notée Δ) entre deux mots a et b est le nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer dans a pour obtenir b. Formellement, la distance entre deux mots est obtenue par la relation de récurrence :*

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

où m^- est le mot m sans sa première lettre.

3.1 Calcul de la distance de Levenshtein

3.1.1 Calcul exact

On utilise simplement la relation de récurrence dans un algorithme de programmation dynamique. On arrive alors à calculer la distance entre deux mots avec une complexité temporelle de $\mathcal{O}(|a||b|)$, ce qui est plutôt lent.

		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

Tableau 1 : Exemple de calcul de distance de Levenshtein par programmation dynamique

3.2 Problème

C'est le problème des K plus proches voisins sous la distance de Levenshtein. On dispose d'un ensemble de N mots. Pour chaque mot, on cherche les K mots qui lui sont le plus similaires par la distance de Levenshtein.

3.3 Plongement de (Σ^n, Δ) dans $\mathbb{R}^d$

Le problème des K plus proches voisins étant déjà assez compliqué pour la distance euclidienne, on pourrait donc imaginer qu'il l'est encore plus pour la distance de Levenshtein qui est moins simple. Pourtant, il existe plusieurs algorithmes assez efficaces autour de la distance euclidienne usuelle, notamment la structure des kd-trees.
Je propose alors de plonger l'espace des mots dans l'espace euclidien tel que les distances sont plus ou moins conservées.

Définition 2 (Plongement de distortion C)**.** *On appelle plongement de distortion C d'un espace métrique (E, N) dans $(\mathbb{R}^d, \|\cdot\|_2)$ une application f :* $\begin{cases} (E,N) \longrightarrow (\mathbb{R}^d, \|\cdot\|_2) \\ m \longmapsto v_m \end{cases}$ *telle que :*

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

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

Remarque 1. *Un plongement est dit isométrique s'il est de distortion $C = 1$. Dans ce cas, les distances sont exactement conservées.*

Remarque 2. *L'existence d'un plongement isométrique n'est pas guarantie.*

Pour le reste du document, je vais travailler sur l'espace métrique des séquences d'ADN de taille n muni de la distance de Levenshtein, c'est-à-dire (Σ^n, Δ) où $\Sigma = \{A, T, G, C\}$

Théorème 1 (Théorème de Bourgain)**.** *Tout espace métrique de cardinal N est plongeable dans $\mathbb{R}^{\mathcal{O}(\log^2 N)}$ avec une distortion $\mathcal{O}(\log N)$*

On est alors guaranti l'existence d'un plongement de (Σ^n, Δ) dans $\mathbb{R}^{\mathcal{O}(\log^2(4^n))} = \mathbb{R}^{\mathcal{O}(n^2)}$ avec une distortion $\mathcal{O}(\log(4^n)) = \mathcal{O}(n)$, ce qui n'est pas très intéressant. Mais on verra que les résultats sont beaucoup plus prometteurs en pratique.
Afin de chercher un plongement raisonnable, j'ai décidé de concevoir un réseau de neurones pour cela.

4 Réseau de neuronnes

La plupart des articles travaillant sur ce problème de plongement sont partis sur un réseau convolutionnel (CNN). J'ai décidé alors de faire autrement, ainsi je propose une architecture de réseau de neuronnes récurrent (RNN) que je tente de justifier dans la partie qui suit.

4.1 Traitement de données

Définition 3 (Composante selon une lettre d'un mot)**.** *Pour un mot $s \in \Sigma^n$, et une lettre $c \in \Sigma$, on appelle composante de s selon c le vecteur colonne $I_c(s)$ tel que : $I_c(s)_{i,1} = \mathbb{1}[s[i] = c]$ pour $0 \leq i < n$*

On génère alors pour une séquence s l'ensemble des composantes $I_c(s)$ selon chacune des lettres.

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

Remarque 3. *Toute séquence de Σ^n est uniquement définie par 3 des 4 composantes, car la $4^{ème}$ peut être déduit des 3 autres : $I_T(s) = [11...11]^\top - (I_A(s) + I_C(s) + I_G(s))$*

4.2 Structure du réseau

La structure du réseau de neuronnes récurrent (RNN) est montrée dans la Figure 1. Le réseau marche sous 3 étapes, où à chaque étape on prend en considération une nouvelle composante de la séquence (selon A, puis C, puis G), et génère alors un vecteur E_i de $\mathbb{R}^d$ qui ne considère que les premières quelques lettres de la séquence uniquement (par exemple, E_1 ne prend en compte que les positions des lettres A et C). De plus, j'ai omis la composante selon la dernière lettre T vu la **Remarque 3** afin d'accélérer l'entrainement et le calcul des vecteurs images. Et donc, E_2 est l'image finale $f(s)$ de la séquence s.

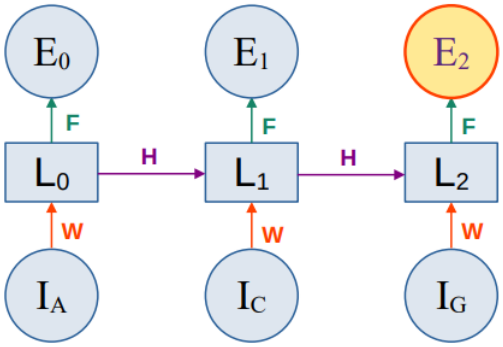

FIGURE 1 – Architécture du RNN

Plus formellement, notre réseau est le suivant :

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

où :

- On confond I_0, I_1, I_2 avec I_A, I_C, I_G respectivement
- tanh est choisie comme fonction d'activation
- α est le facteur d'agrandissement

α a été choisi un peu grossièrement mais a donné de bons résultats. Pour un facteur d'agrandissement β, la distance maximale entre deux vecteurs de la forme $\tanh(X) \times \beta$ avec $X \in \mathbb{R}^d$ est $2\beta\sqrt{d}$, or la distance maximale entre deux séquences de Σ^n est n, donc pour que $2\beta\sqrt{d} = n$ il faut que $\beta = \frac{n}{2\sqrt{d}}$, or pour donner un peu plus d'espace on remplace le $\frac{1}{2}$ par $\frac{3}{4}$ et donc j'ai posé $\alpha = \frac{3n}{4\sqrt{d}}$

4.3 Algorithme d'apprentissage

On confond encore I_0, I_1, I_2 avec I_A, I_C, I_G. Et on note pour une séquence s, s_i la séquence ne considérant que les $i+1$ premières lettres :

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

.

Par exemple pour $s = AGGT$, on a que $s_2 = [1\ 3\ 3\ 0]^\top$, ce qui revient dans ce cas (où $i = 2$) à remplacer l'alphabet $\Sigma = \{A, C, G, T\}$ par $\Sigma' = \{1, 2, 3, 0\}$, ce qui ne change pas la distance de Levenshtein, ce qui est consistent avec la Remarque 3. Pour entrainer le réseau de neuronnes, on génère à chaque fois deux séquences $s^{(1)}$ et $s^{(2)}$. Puis on fait passer les deux par le réseau actuel, générant ainsi pour chacune 3 vecteurs $E_0^{(i)}, E_1^{(i)}, E_2^{(i)}$ où $i \in \{1, 2\}$.

Remarque 4. *Le fait de générer deux séquences aléatoires indépendemment l'une de l'autre est à éviter car la distance entre les deux dans ce cas prend très souvent des valeurs proches de $\frac{n}{2}$ et donc les séquence similaires n'auront pas d'images proches l'une de l'autre dans l'espace euclidien. On génère alors la deuxième séquence à partir de la première en lui modifiant un nombre aléatoire de lettres.*

On utilise la fonction σ comme fonction d'erreur : $\sigma = \sum_{i=0}^{2} w_i (\Delta(s_i^{(1)}, s_i^{(2)}) - \|E_i^{(1)} - E_i^{(2)}\|)^2$

où w_i est le poids de l'erreur à la $i^{\text{ème}}$ étape du réseau récurrent. Clairement, l'erreur la plus importante est celle à fin puisque ça correspond aux images finales des séquences, donc on choisit $w_0 = w_1 = 1$ et $w_2 = 3$. Finalement, on applique avec cette fonction d'erreur la méthode de rétropropagation du gradient.

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

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

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

4.4 Résultats

J'ai entrainé le réseau pour plonger les séquences de tailles $n = 10, 25, 100$ dans $\mathbb{R}^{10}, \mathbb{R}^{25}, \mathbb{R}^{100}$ respectivement. Pour tester la performance, on génère 50000 paires de séquences et calcule la différence absolue moyenne entre la distance de Levenshtein des deux séquences et la distance euclidienne des deux vecteurs générés par le RNN pour chacune des séquences, et on obtient les résultats suivant :

Taille des séquences n	Dimension de l'espace euclidien d	Erreur absolue moyenne
10	10	0.99
25	25	2.16
100	100	5.95

Tableau 1 : Erreur moyenne après entrainement

Pour voir plus en détail l'erreur de notre modèle pour le cas $n = 100$, $d = 100$, j'ai généré 500 paires de séquences, et représenté graphiquement la distance euclidienne entre leurs images, en fonction de leur distance de Levenshtein ci-dessous :

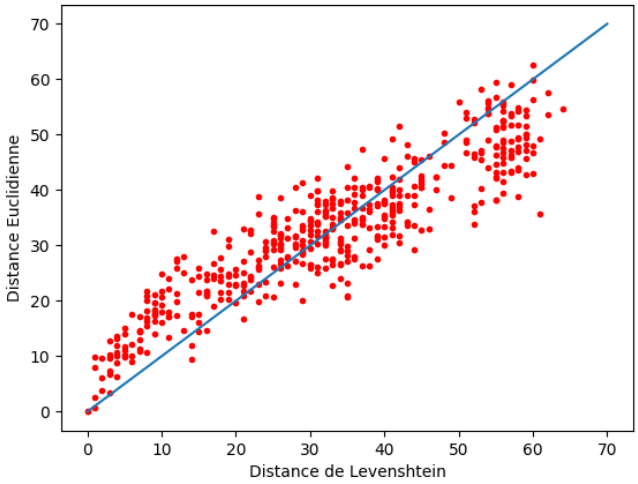

FIGURE 2 – Distance euclidienne des images générées vs distance de Levenshtein des séquences

4.5 Justification du choix de structure du réseau

Le réseau génère le vecteur image d'une séquence en considérant les positions d'une nouvelle lettre à chaque fois. J'ai montré alors le résultat suivant qui indique que la distance de levenshtein pour une séquence entière est bien reliée aux distances pour ses composantes.

Théorème 2. *Soient* $s^{(1)}$ *et* $s^{(2)}$ *deux séquences de* Σ^n*, alors* $\sum_{i=0}^{3} \Delta(s_i^{(1)}, s_i^{(2)}) \leq 4\Delta(s^{(1)}, s^{(2)})$

La démonstration se base principalement sur le lemme suivant :

Lemme 3. *Soient* $s^{(1)}$ *et* $s^{(2)}$ *deux séquences de* Σ^n*, alors il existe* $m \in \mathbb{N}$ *tel qu'il existe* $(a_i)_{1\leq i\leq m} \in (\Sigma^+)^m$*, et* $(y_i)_{1\leq i\leq m+1}$ *et* $(z_i)_{1\leq i\leq m+1}$ *dans* $(\Sigma^*)^{(m+1)}$ *tel que :*

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

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

Exemple illustrant le Lemme 3

Le lemme 3 permet aussi de mieux visualiser les opérations optimales de substitution, insertion ou suppression pour transformer $s^{(1)}$ en $s^{(2)}$, puisqu'il suffit de garder les a_i, et de transformer les y_i en z_i en substituant les $\min(|y_i|, |z_i|)$ premières lettres, et insérant ou supprimant $||y_i| - |z_i||$ lettres dans le suffixe selon les cas $|y_i| \leq |z_i|$ ou $|y_i| > |z_i|$. Et donc le théorème 2 découle rapidement du lemme 3 puisqu'on voit qu'appliquer ces opérations permet aussi de transformer chacune des $s_i^{(1)}$ vers $s_i^{(2)}$.

Ensuite, j'ai tenté de montrer que $\sum_{i=0}^{3} \Delta(s_i^{(1)}, s_i^{(2)}) \geq \Delta(s^{(1)}, s^{(2)})$ par une induction assez longue qui marchait quasiment toujours, sauf pour un cas très spécifique qui fait que la proposition soit fausse, mais expérimentalement j'ai trouvé qu'elle est vérifié quasiment toujours.

5 Graphe de Bruijn

Les techniques de séquençage d'ADN ne permettent que de séquencer des fragments de tailles limitées, le but est de reconstruire la séquence entière à partir de plusieurs lectures de ses fragments. Dans cette partie on travaille sous les hypothèses suivantes :

— Les lectures se font sans erreur
— Les lectures sont uniformément réparties sur la séquence d'ADN
— Les lectures sont toutes de taille $L = 100$

Définition 4 (k-mer)**.** *Un k-mer d'une certaine séquence est un sous-mot contigu de taille k de cette séquence (exemple : GCT est un 3-mer dans Figure 3)*

Définition 5 (Graphe de De Bruijn d'une séquence)**.** *Le graphe de De Bruijn d'une séquence s, est un graphe orienté (V, E) dont les sommets V est l'ensemble des k-mers de s pour un k fixe, et les arcs E sont créées entre chaque 2 k-mers consécutifs (par exemple les 3-mers TAG et AGC sont consécutifs dans la séquence de Figure 3, alors on a un arc de TAG vers AGC)*

Remarque 5. *E est un multiensemble, c'est à dire qu'on permet qu'un arc y apparaisse plusieurs fois.*

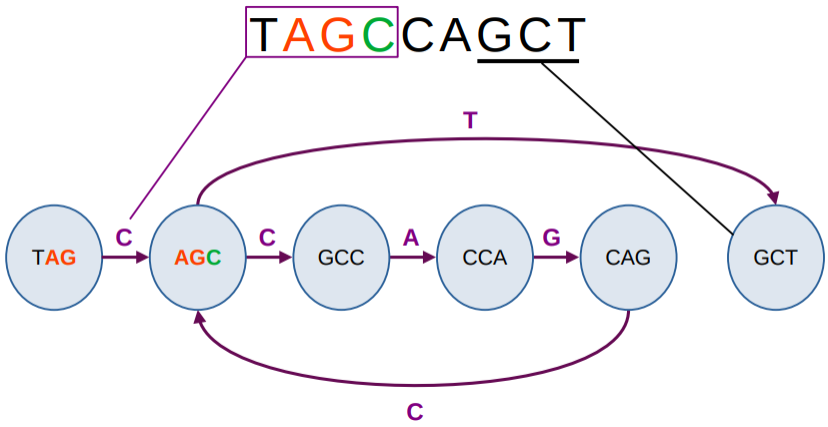

FIGURE 3 – Un exemple de graphe de De Bruijn pour $k = 3$

Remarque 6. *Un arc représentant le fait de passer au k-mer suivant et donc d'ajouter une lettre à la fin (et retirer celle au début), un sous-mot (de taille $\geq k$) d'une séquence est représenté par un chemin dans son graphe de De bruijn.*

Par exemple, lire le sous-mot CCAGC dans Figure 3 revient à suivre le chemin : CCA$\longrightarrow$CAG$\longrightarrow$AGC dans le graphe.

Définition 6 (Chemin eulérien)**.** *Un chemin eulérien d'un graphe orienté est un chemin qui passe par tous les arcs, une seule fois par arc.*

Remarque 7. *Ainsi la séquence en entier est représentée dans le graphe de De Bruijn par un chemin eulérien.*

Définition 7 (Graphe de De Bruijn d'un ensemble de lectures)**.** *Le graphe de De Bruijn d'un ensemble de lectures est l'union des graphes de De Bruijn de ces lectures, c'est à dire le graphe* $(\bigcup_i V_i, \bigcup_i E_i)$.

Pour un grand nombre de lectures qui recouvrent assez bien la séquence en sa totalité, le graphe de De Bruijn de ces lectures contient tous les arcs du graphe de Bruijn de la séquence qu'on cherche à déterminer, répétés un certain nombre de fois.
Par exemple pour la même séquence que dans **Figure 3**, les deux lectures dans **Figure 4** donnent

un graphe de De Bruijn quasi identique à celui de la séquence, mais avec l'arc $CCA \longrightarrow CAG$ répété deux fois.

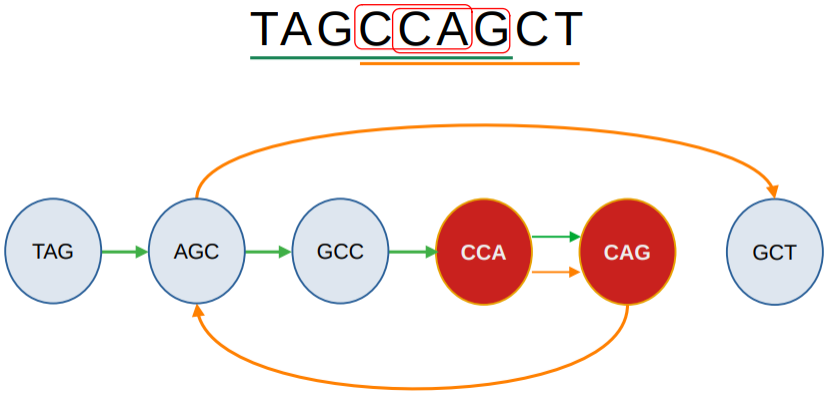

FIGURE 4 – Graphe de De Bruijn de 2 lectures (soulignées)

Mon idée était alors d'estimer ce nombre de répétitions d'arcs afin de les éliminer et n'en garder qu'une seule copie.

Théorème 4. *Pour une séquence $s \in \Sigma^m$, si on effectue N lectures de taille L uniformément répartis sur la séquence, si G le graphe de De Bruijn de s pour des k-mers de taille k, et C_i est la vad du nombre de copies de l'arc $s[i : i+k-1] \longrightarrow s[i+1 : i+k]$, alors :*

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

Remarque 8. *Un arc correspond à deux k-mers successifs, donc les arcs correspondent exactement aux $(k+1)$-mers. On dira donc qu'une lecture contient un arc si elle contient le $(k+1)$-mer correspondant.*

Démonstration. On note E_i l'arc s[i :i+k-1] → s[i+1 :i+k] et $A_i = \mathbb{1}_{\text{la } i^{\text{ème}} \text{ lecture contient } E_i}$.

Alors $C_j = \sum_{i=1}^{N} A_i$, or $\mathbb{E}(A_i) = \mathbb{P}(A_i) = \frac{\text{nombre de lectures possibles contenant } E_i}{\text{nombre de lectures possibles}} = \frac{L-k}{m-L+1}$ donc

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

□

L'approche que j'applique est alors de transformer le graphe de De Bruijn des lectures en un graphe pondéré où le poid w d'un arc est le nombre de nombre de fois où c'est répété, et donc pour essayer d'obtenir une unique copie pour chaque arc du graphe de De Bruijn de la séquence, on change chaque poids w par $f(w) = \max([\frac{w}{\mathbb{E}(C)}], 1)$. L'idée étant en gros qu'on divise en arrondissant par le nombre de répétitions, et pour qu'aucun arc ne disparaisse, on garde au moins une copie si la division arrondie est nulle. A ce point, le graphe obtenu devrait être assez similaire au graphe de De Bruijn de la séquence qu'on cherche, et donc vu la remarque, on cherche un chemin eulérien par l'algorithme de Hierholzer qui est de complexité linéaire $\mathcal{O}(|E|)$.

Théorème 5. *Un graphe orienté admet un chemin eulérien si et seulement si il est connexe et tous ses noeuds sont de même degré entrant que degré sortant, sauf éventuellement deux pour lesquels ils diffèrent de 1.*

Tout cela parait très grossier, pourtant les résultats obtenus sont très satisfaisant. Pour une taille de lecture fixe $L = 100$, et pour une certaine longeur de séquence m allant de 100 à 20000, pour 100 séquences différentes, on génère $N = 2m$ lectures, et tente de récupérer la séquence par la méthode proposée, puis l'erreur est la moyenne des distances de Levenshtein entre chacune des 100 séquences

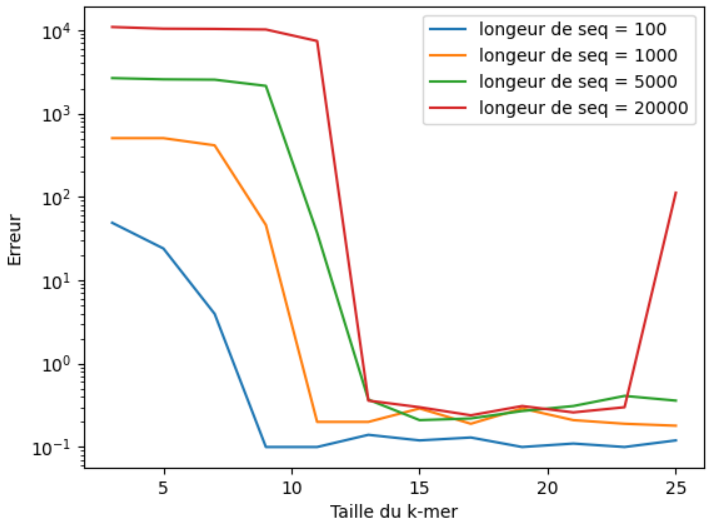

FIGURE 5 – Erreur (distance de Levenshtein) entre les séquences initiales et séquences récupérées selon la taille de k-mer choisie

initiales et la séquence retrouvée. Les résultats indiquent que pour une taille de k-mer bien choisie on récupère quasi sûrement la séquence.

Références

[1] Pierre Pericard : Algorithmes pour la reconstruction de séquences de marqueurs conservés dans des données de métagénomique : 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

6 Annexe

```cpp
#include <bits/stdc++.h>

#define ll long long
#define pb push_back
#define x first
#define y second
#define sz(u) (int)(u.size())

using namespace std;


int sequence_length;
int num_reads = sequence_length;
int read_length;
int kmer_length;
double expected_coverage;

struct Graph{
    vector<vector<int>> adj;
    vector<string> noeuds;
    Graph(vector<vector<int>> G, vector<string> nodes): adj(G), noeuds(nodes){}
};

int gen_rand(int l, int r){
    return l + rand()%(r-l+1);
}

int alignement_lent(string a, string b){
    int n = a.length(), m = b.length();
    vector<vector<int>> dp(n+1,vector<int>(m+1));
    dp[0][0]=0;
    for(int i=0;i<=n;i++){
        for(int j=0;j<=m;j++){
            if(i==0 && j==0) continue;
            dp[i][j] = n+m;
            int non_egaux = a[i]!=b[j];
            if(i>0) dp[i][j] = dp[i-1][j]+1;
            if(j>0) dp[i][j] = min(dp[i][j], dp[i][j-1]+1);
            if(i>0 && j>0) dp[i][j] = min(dp[i][j], dp[i-1][j-1]+non_egaux);
        }
    }
    return dp[n][m];
}

Graph  de_bruijn_graph(vector<string> lectures, int k){ //k : taille du kmer
    unordered_map<string,int> indice_du_kmer;
    vector<string> noeud;
    vector<vector<int>> graph_db;
    map<pair<int,int>,int> edges;
    for(string t : lectures){
        int indice_precedent = -1;
        for(int i=0;i+k<=t.length();i++){
            string kmer="";
            for(int j=0;j<k;j++){
                kmer += t[i+j];
```

```
            }
            int indice = -1;
            if(indice_du_kmer.count(kmer) == 0){
                indice = (int)(noeud.size());
                indice_du_kmer[kmer] = indice;
                graph_db.push_back(vector<int>{});
                noeud.pb(kmer);
            }
            else{
                indice = indice_du_kmer[kmer];
            }
            if(indice_precedent!=-1){
                edges[{indice_precedent,indice}]++;
                 }
            indice_precedent = indice;
        }
    }
    for(auto &p : edges){
        p.y = max(1,(int)(round(double(p.y)/expected_coverage)+0.5));
        for(int k=0;k<p.y;k++){
            graph_db[p.x.x].pb(p.x.y);
        }
    }
    return Graph(graph_db, noeud);
}


vector<int> parcours_eulerien(vector<vector<int>> graph){
    vector<int> parcours_temp;
    vector<int> ret;
    int init = 0, n = graph.size();
    vector<int> in_deg(n);
    for(int i=0;i<n;i++){
        for(int j=0;j<sz(graph[i]);j++){
            in_deg[graph[i][j]]++;
        }
    }
    for(int i=0;i<n;i++){
        if(sz(graph[i])==in_deg[i]+1){
            init = i;
            break;
        }
    }
    parcours_temp.pb(init);
    while(!parcours_temp.empty()){
        int cur = parcours_temp.back();
        bool sort = false;
        while(graph[cur].empty()){
            ret.pb(cur);
            parcours_temp.pop_back();
            if(parcours_temp.empty()){
                sort = true;
                break;
            }
            cur = parcours_temp.back();
        }
```

```
        if(sort) break;
        if(!graph[cur].empty()){
            parcours_temp.pb(graph[cur].back());
            graph[cur].pop_back();
        }
    }
    reverse(ret.begin(),ret.end());
    return ret;
}

void calc_cvg(){
    expected_coverage = num_reads * double(read_length - kmer_length) / double(sequence_length-read_l
}

//File: matrix.cpp

#include <bits/stdc++.h>
#include <eigen3/Eigen/Dense>
#include <random>

using namespace Eigen;
using namespace std;

#define double long double
#define pb push_back
#define Matrix MatrixXd

Matrix mulDiag(Matrix M, vector<double> D){
    for(int i=0;i<M.rows();i++)
        for(int j=0;j<M.cols();j++)
            M(i,j)*=D[j];
    return M;
}

Matrix Diagmul(vector<double> D, Matrix M){
    for(int i=0;i<M.rows();i++)
        for(int j=0;j<M.cols();j++)
            M(i,j)*=D[i];
    return M;
}

double norm(Matrix vec){
    return sqrt(vec.squaredNorm());
}

//File: neural_network.cpp
#include "matrix.cpp"

struct network{
    int len;
    int dim;
    int layer_dim;
    double learning_rate;
    double momentum;
    Matrix W,H,F;
    Matrix input[2][3], layer[2][3], ac_layer[2][3];
```

```
    Matrix layer_bias;
    Matrix embedding[2][3], ac_embedding[2][3]; //vecteur image
    Matrix grad_errW, grad_errH, grad_errF, grad_bias; //gradients
    string input_str[2];
    int edit_dist[3];
    double embed_dist[3];
    string data_path;
    double scale;
    int batch_size=2;
    network(int string_len, int final_dim, int layerdim=-1, double speed, double mom){
        if(layerdim==-1) layer_dim = string_len;
        else layer_dim = layerdim;
        len = string_len;
        dim = final_dim;
        W = Matrix::Random(layer_dim, len);
        H = Matrix::Random(layer_dim,layer_dim);
        F = Matrix::Random(dim, layer_dim);
        grad_errW = Matrix::Zero(layer_dim, len);
        grad_errH = Matrix::Zero(layer_dim,layer_dim);
        grad_errF = Matrix::Zero(dim, layer_dim);
        learning_rate = speed;
        layer_bias = Matrix::Random(layer_dim,1);
        grad_bias = Matrix::Zero(layer_dim,1);
        scale = ((double)(len)*3.)/(4.*sqrt(dim));
        momentum = mom;
        data_path = "data"+to_string(len)+"_"+to_string(dim)+".txt";
    }

    void clear_input(){
        for(int i=0;i<2;i++){
            for(int j=0;j<3;j++){
                input[i][j] = Matrix::Zero(len,1);
            }
        }
    }

    double levenshtein(string a, string b){
        int n = a.length(), m = b.length();
        int dp[n+1][m+1];
        for(int i=0;i<=n;i++) dp[i][0]=i;
        for(int i=1;i<=m;i++) dp[0][i]=i;
        for(int i=1;i<=n;i++){
            for(int j=1;j<=m;j++){
                dp[i][j] = min({dp[i-1][j]+1, dp[i][j-1]+1, dp[i-1][j-1]+int(a[i-1]!=b[j-1])});
            }
        }
        return (double)(dp[n][m]);
    }

    double deriv(double x){
        return (1-tanh(x)*tanh(x))/(double)(2.5*len);
    }

    Matrix tanhM(Matrix M){
        for(int i=0;i<M.rows();i++)
            for(int j=0;j<M.cols();j++)
```

```
                M(i,j) = tanh(M(i,j)/(2.5*len));
        return M;
    }

    Matrix derivM(Matrix M){
        for(int i=0;i<M.rows();i++)
            for(int j=0;j<M.cols();j++)
                M(i,j) = deriv(M(i,j));
        return M;
    }

    void add_input(string sq, int ind=0){
        int n = sq.length();
        for(int i=0;i<n;i++){
            if(sq[i]=='A') input[ind][0](i,0)=1;
            else if(sq[i]=='G') input[ind][1](i,0)=1;
            else if(sq[i]=='C') input[ind][2](i,0)=1;
        }
        input_str[ind] = sq;
    }

    void execute(int ind){
        layer[ind][0] = (W * input[ind][0]) + layer_bias;
        ac_layer[ind][0] = tanhM(layer[ind][0]);
        embedding[ind][0] = F * layer[ind][0];
        ac_embedding[ind][0] = tanhM(embedding[ind][0]);
        for(int i=1;i<=2;i++){
            layer[ind][i] = (W * input[ind][i]) + (H * layer[ind][i-1]) + layer_bias;
            ac_layer[ind][i] = tanhM(layer[ind][i]);
            embedding[ind][i] = F*layer[ind][i];
            ac_embedding[ind][i] = tanhM(embedding[ind][i]);
        }
    }

    void calcError(){
        for(int i=0;i<3;i++) embed_dist[i] = norm(ac_embedding[0][i]-ac_embedding[1][i]);
        for(int i=0;i<3;i++){
            string a[2] = {string(len,'0'), string(len,'0')};
            for(int j=0;j<=i;j++){
                for(int k=0;k<len;k++){
                    if(abs(input[0][j](k,0) - 1.)<1e-2) a[0][k] = '1'+j;
                    if(abs(input[1][j](k,0) - 1.)<1e-2) a[1][k] = '1'+j;
                }
            }
            edit_dist[i] = levenshtein(a[0],a[1]);
        }
    }

    void clear_gradients(){
        grad_errF *= momentum;
        grad_errH *= momentum;
        grad_errW *= momentum;
        grad_bias *= momentum;
    }

    void set_learning_rate(){
```

```
280         default_random_engine gen;
281         uniform_real_distribution<double> distrib(0.04, 0.06);
282         learning_rate = distrib(gen);
283     }
284 
285     void backPropagate(){
286         calcError();
287 
288         Matrix grad_LiLi_1[2];
289         for(int i=0;i<2;i++){
290             vector<double> dg;
291             for(int j=0;j<layer_dim;j++){
292                 dg.pb(deriv(layer[0][i+1](j,0)));
293             }
294             grad_LiLi_1[i] = mulDiag(H.transpose(), dg);
295         }
296 
297         for(int k=2;k<3;k++){
298             if(abs(embed_dist[k])<1e-2) return;
299             Matrix E1_E2 = ac_embedding[0][k]-ac_embedding[1][k];
300             double diff = embed_dist[k]*scale-edit_dist[k];
301             Matrix grad_err_lay_bias[k+1];
302             Matrix grad_errL[k+1];
303             vector<double> diagE;
304             for(int i=0;i<dim;i++) diagE.pb(deriv(embedding[0][k](i,0)));
305             grad_errL[k] = (mulDiag(F.transpose(),diagE) * E1_E2)*(2.*diff*scale/embed_dist[k]);
306             for(int i=k-1;i>=0;i--){
307                 grad_errL[i] = grad_LiLi_1[i] * grad_errL[i+1];
308             }
309             for(int i=0;i<k+1;i++){
310                 vector<double> dg;
311                 for(int j=0;j<layer_dim;j++) dg.pb(deriv(layer[0][i](j,0)));
312                 Matrix tmp = Diagmul(dg, grad_errL[i]);
313                 grad_bias += (1. - momentum) * (tmp*learning_rate);
314                 grad_errW += (1. - momentum) * (tmp * input[0][i].transpose());
315                 if(i>0) grad_errH +=  (1. - momentum) * (tmp * ac_layer[0][i-1].transpose());
316             }
317             grad_errF += (1. - momentum) * (Diagmul(diagE,E1_E2 * (2.*diff*scale/embed_dist[k])) *ac_
318         }
319 
320         for(int i=0;i<2;i++){
321             vector<double> dg;
322             for(int j=0;j<layer_dim;j++){
323                 dg.pb(deriv(layer[1][i+1](j,0)));
324             }
325             grad_LiLi_1[i] = mulDiag(H.transpose(), dg);
326         }
327 
328         for(int k=2;k<3;k++){
329             Matrix E1_E2 = ac_embedding[0][k]-ac_embedding[1][k];
330             double diff = embed_dist[k]*scale-edit_dist[k];
331             Matrix grad_err_lay_bias[k+1];
332             Matrix grad_errL[k+1];
333             vector<double> diagE;
334             for(int i=0;i<dim;i++) diagE.pb(deriv(embedding[1][k](i,0)));
335             grad_errL[k] = (mulDiag(F.transpose(),diagE) * E1_E2)*(2.*diff*scale/embed_dist[k]);
```

```
            for(int i=k-1;i>=0;i--){
                grad_errL[i] = grad_LiLi_1[i] * grad_errL[i+1];
            }
            for(int i=0;i<k+1;i++){
                vector<double> dg;
                for(int j=0;j<layer_dim;j++) dg.pb(deriv(layer[1][i](j,0)));
                Matrix tmp = Diagmul(dg, grad_errL[i]);
                grad_bias -= (1. - momentum) * (tmp*learning_rate);
                grad_errW -= (1. - momentum) * (tmp * input[1][i].transpose());
                if(i>0) grad_errH -= (1. - momentum) * (tmp * ac_layer[1][i-1].transpose());
            }
            grad_errF -= (1. - momentum) * (Diagmul(diagE,E1_E2 * (2.*diff*scale/embed_dist[k])) *ac_
        }
    }

    void apply_gradients(){
        /*grad_errH += Matrix::Random(H.rows(),H.cols())*0.0005;
        grad_errW += Matrix::Random(W.rows(),W.cols())*0.0005;
        grad_errF += Matrix::Random(F.rows(),F.cols())*0.0005;
        grad_bias += Matrix::Random(layer_dim,1)*0.0005;*/
        H -= grad_errH * learning_rate;
        W -= grad_errW * learning_rate;
        F -= grad_errF * learning_rate;
        layer_bias -= grad_bias * learning_rate;
    }

    void save(){
        fstream file;
        file.open(data_path, ios::out);
        file<<len<<' '<<dim<<' '<<layer_dim<<'\n';
        for(int i=0;i<W.rows();i++){
            for (int j=0;j<W.cols();j++)
                file<<W(i,j)<<' ';
            file<<'\n';
        }
        file<<'\n';
        for(int i=0;i<H.rows();i++){
            for(int j=0;j<H.cols();j++)
                file<<H(i,j)<<' ';
            file<<'\n';
        }
        file<<'\n';
        for(int i=0;i<F.rows();i++){
            for(int j=0;j<F.cols();j++)
                file<<F(i,j)<<' ';
            file<<'\n';
        }
        file<<'\n';
        for(int j=0;j<layer_dim;j++)
                file<<layer_bias(j,0)<<' ';
        file<<'\n';
        file.close();
    }

    void load_data(){
        fstream file;
```

```
392         file.open(data_path, ios::in);
393         if(file){
394         file>>len;
395         file>>dim;
396         file>>layer_dim;
397         W = Matrix::Zero(layer_dim, len);
398         H = Matrix::Zero(layer_dim,layer_dim);
399         F = Matrix::Zero(dim, layer_dim);
400         for(int i=0;i<W.rows();i++)
401             for(int j=0;j<W.cols();j++)
402                 file>>W(i,j);
403         for(int i=0;i<H.rows();i++)
404             for(int j=0;j<H.cols();j++)
405                 file>>H(i,j);
406         for(int i=0;i<F.rows();i++)
407             for(int j=0;j<F.cols();j++)
408                 file>>F(i,j);
409         for(int i=0;i<layer_dim;i++)
410             file>>layer_bias(i,0);
411         file.close();}
412     }
413
414     string gen_rand_seq(){
415         string alphabet[4] = {"A", "C", "G", "T"};
416         string ret="";
417         for(int i = 0; i<len; i++){
418             ret += alphabet[rand()%4];
419         }
420         return ret;
421     }
422
423     string rand_with_dist(string a){
424         int half = rand()%5;
425         if(half==0) return gen_rand_seq();
426         int dst = (rand()%((int)(len*0.95)))+1;
427         string ret=a;
428         char c[4]={'A','G','C','T'};
429         for(int i=0;i<dst;i++){
430             int tmp = rand()%4;
431             int pos = rand()%len;
432             ret[pos] = c[tmp];
433         }
434         return ret;
435     }
436
437     void train(){
438         load_data();
439         double errormean = 0;
440         for(int i=1;i<=iterations;i++){
441             for(int j=0;j<20000;j++){
442                 clear_gradients();
443                 set_learning_rate();
444                 learning_rate = learning_rate / (double)(batch_size);
445                 for(int k=0;k<batch_size;k++){
446                     clear_input();
447                     add_input(gen_rand_seq(),0);
```

```
                    add_input(rand_with_dist(input_str[0]),1);
                    execute(0);
                    execute(1);
                    backPropagate();
                }
                apply_gradients();
            }
            save();
            errormean=0;
        }
    }

    void gen_data(){
        load_data();
        freopen("data.txt","w",stdout);
        for(int i=0;i<iter;i++){
            clear_input();
            add_input(gen_rand_seq(),0);
            add_input(rand_with_dist(input_str[0]),1);
            execute(0);
            execute(1);
            calcError();
            cout<<edit_dist[2]<<' '<<embed_dist[2]*scale<<'\n';
        }
    }
};
```