Construction d'arbres phylogénétiques de virus

Quentin Moreau

2021-2022

La pandémie actuelle, d'après IPBES [1], sera probablement suivie d'autres épidémies. Mieux connaître les virus devient alors un impératif scientifique. Les arbres phylogénétiques permettent justement aux biologistes de mieux connaître les espèces.

Problème. Comment construire un arbre phylogénétique optimisé à partir d'un ensemble de séquences ADN ?

1 Modélisation et notations

Définition 1.0.1. L'ADN est une macromolécule formée elle-même d'une chaîne de molécules appelées nucléotides qui sont l'adénine (A), la cytosine (C), la guanine (G) et la thymine (T). Chaque être vivant est caractérisé par son ADN.

Notations :
- Σ désigne l'alphabet des nucléotides $\{A, C, G, T\}$;
- Le langage de l'ADN, noté $\mathcal{L}$ est Σ^* ;
- Pour tout mot w de $\mathcal{L}$, $\mathcal{P}(w)$, $\mathcal{F}(w)$ et $\mathcal{S}(w)$ désignent respectivement les ensembles des préfixes, des facteurs et des suffixes de w ;
- Pour tout mot w de $\mathcal{L}$, $|w|$ désigne la longueur de w ;
- Pour tout mot w de $\mathcal{L}$, pour tout entier i de $[\![0, |w| - 1]\!]$, $w[i]$ désigne la $(i + 1)$ème lettre de w.
- Nous identifions les lettres A, C, G et T aux nombres 1, 2, 3 et 4. Nous écrivons par exemple, pour toute matrice M de $\mathcal{M}_4(\mathbf{R})$, $M[A, G]$ le coefficient de la première ligne et de la troisième colonne de M.

2 Recherche des régions codantes

2.1 Les régions codantes

Définition 2.1.1. Certaines régions de l'ADN évoluent plus lentement que d'autres (subissent moins de modifications génétiques d'une génération à l'autre). Elles sont appelées régions codantes, tandis que celles qui évoluent plus rapidement sont appelées régions non codantes par opposition.

Remarque. Les régions non codantes évoluent particulièrement rapidement chez les virus et peuvent être considérées comme aléatoires. Il est donc important de ne travailler que sur les régions codantes, plus représentatives des ressemblances entre espèces.

Problème. Comment déterminer les régions codantes ?

Soit w, une séquence ADN de $\mathcal{L}$. Les régions codantes de w sont des facteurs f de $\mathcal{F}(w)$ de la forme suivante :

$$\begin{gathered}(1)\ ATG \in \mathcal{P}(f)\,;\\ (2)\ f \text{ est suivi d'un élément de } \{TAA, TAG, TGA\}\,;\\ (3)\ \forall c \in \mathcal{F}(f), c \notin \{TAA, TAG, TGA\}\,;\\ (4)\ |f| \in 3\mathbf{Z}.\end{gathered}$$

Cependant, les conditions (1), (2), (3) et (4) ne sont pas suffisantes mais seulement nécessaires. Nous ajoutons donc, en accord avec [2] la condition :

$$(5)\ |f| \geq 300.$$

Remarque. La condition heuristique (5) n'est pas nécessaire. Nous sommes donc susceptibles de trouver des faux positifs et de faux négatifs dans nos tests sur les régions codantes.

Définition 2.1.2. Le mot ATG est nommé codon start et les éléments de $\{TAA, TAG, TGA\}$ sont nommés codons stop. L'ensemble des codons start et stop est noté $\mathcal{C}$.

2.2 Algorithmes de recherche de motif

Soit w, une séquence ADN de $\mathcal{L}$, et c, un codon de $\mathcal{C}$.

Définition 2.2.1. Une occurence de c dans w est un indice i de $[\![0, |w| - |c|]\!]$ tel que : $\forall j \in [\![0, |c| - 1]\!], w[i + j] = c[j]$.

Problème. Comment trouver toutes les occurrences de c dans w le plus efficacement possible ?

Définition 2.2.2. Posons $\mathcal{A} = (\mathcal{P}(c), \{\varepsilon\}, \{c\}, \delta)$ où δ est définie comme l'application de $\mathcal{P}(c) \times \Sigma$ dans $\mathcal{P}(c)$ qui à tout couple (q, a) de $\mathcal{P}(c) \times \Sigma$ associe le plus long mot de $\mathcal{P}(c) \cap \mathcal{S}(q.a)$. L'automate $\mathcal{A}$ est appelé automate des suffixes de c.

Proposition 2.2.1. *L'automate des suffixes de c reconnaît le langage $\mathcal{L}.c$.*

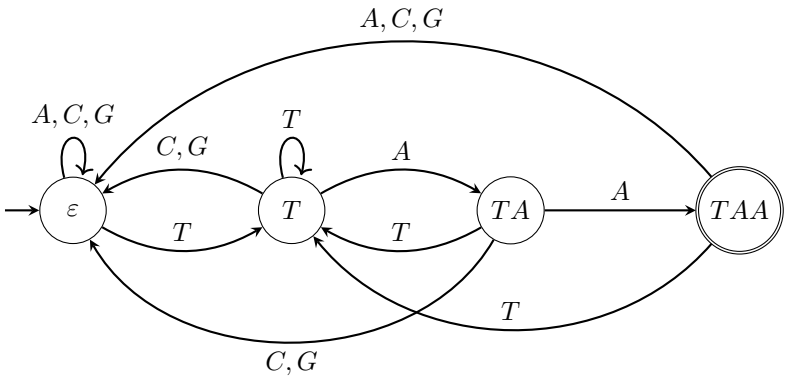


FIGURE 1 – Exemple : automate des suffixes de TAA

Algorithme 1 : Construction de l'automate des suffixes

Entrée : $\mathsf{c} \in \mathcal{C}$

$\mathsf{A}.\delta \leftarrow$ table de longueur $(|\mathsf{c}|+1) \times |\Sigma|$;

// les préfixes de c sont identifiés à leurs longueurs, donc les états de A sont des entiers

Pour $\mathsf{i} \leftarrow 0$ **à** $\mathsf{i} \leq |\mathsf{c}|$ **faire**
  Pour $\mathsf{j} \leftarrow 0$ **à** $\mathsf{i} < |\Sigma|$ **faire**
    $\mathsf{A}.\delta[\mathsf{i}][\mathsf{j}] \leftarrow 0$;
  Fin pour
Fin pour

Pour $\mathsf{i} \leftarrow 1$ **à** $\mathsf{i} \leq |\mathsf{c}|$ **faire**
  ancienne transition $\leftarrow \mathsf{A}.\delta[\mathsf{i}-1][\mathsf{c}[\mathsf{i}-1]]$;
  $\mathsf{A}.\delta[\mathsf{i}-1][\mathsf{c}[\mathsf{i}-1]] \leftarrow \mathsf{i}$;
  Pour $\mathsf{j} \leftarrow 0$ **à** $\mathsf{i} < |\Sigma|$ **faire**
    $\mathsf{A}.\delta[\mathsf{i}][\mathsf{j}] \leftarrow \mathsf{A}.\delta[$ancienne transition$][\mathsf{j}]$;
  Fin pour
Fin pour

Renvoyer A

Proposition 2.2.2. *L'algorithme 1 admet une complexité temporelle en* $O(|c| \times |\Sigma|)$.

Démonstration. L'algorithme présente, par deux fois, deux boucles pour imbriquées qui réalisent $|c|+1$ tours et $|\Sigma|$ tours. □

Proposition 2.2.3. *L'algorithme 2 admet une complexité temporelle en* $O(|w|+|c| \times |\Sigma|)$ *et une complexité spatiale en* $O(|c| \times |\Sigma|)$.

Algorithme 2 : Recherche avec l'automate des suffixes

Entrée : $\mathsf{w} \in \mathcal{L}$, $\mathsf{c} \in \mathcal{C}$

```
file ← file vide ;
A ← automate des suffixes de c ;
etat ← 0;
Pour i ← 0 à i < |w| faire
    etat ← A.δ[etat][w[i]];
    Si etat = |c| alors
        file.enfiler (i);
    Fin si
Fin pour
Renvoyer file
```

Démonstration. Le calcul de l'automate des suffixes de c nécessite un $O(|c| \times |\Sigma|)$ d'opérations élémentaires d'après la proposition 2.2.2. De plus, l'algorithme 2 parcourt ensuite la chaîne w à travers une boucle pour. □

Durées d'éxecutions des algorithmes sur 200000 recherches d'un motif de 3 lettres dans une séquence de 10000 lettres

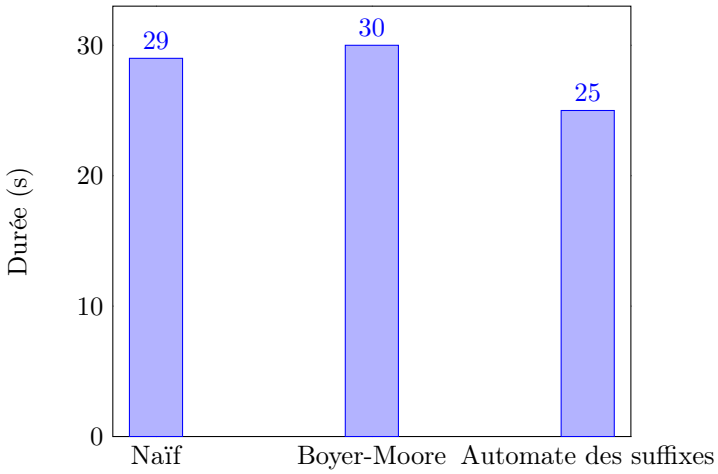


FIGURE 2 – Comparaison des différents algorithmes de recherche

Dans la suite, nous travaillerons exclusivement sur les régions codantes. Ainsi, lorsque l'on parlera de la séquence ADN w, on entendra la concaténée de ses régions codantes.

3 Comparaison des espèces

3.1 Distance de Levenshtein

Problème. Comment quantifier la distance entre les espèces ?

Définition 3.1.1. La distance de Levenshtein pondérée d est définie de $\mathcal{L}^2$ dans $\mathbf{R}$ récursivement par :

$$\forall (S_1, S_2) \in \mathcal{L}^2, \forall (X, Y) \in \Sigma^2,$$
$$\mathrm{d}(\mathrm{S}_1.X, S_2.Y) = \min(\mathrm{d}(S_1, S_2) + M[X, Y], \mathrm{d}(S_1.X, S_2) + \alpha, \mathrm{d}(S_1, S_2.Y) + \alpha),$$
$$\text{et } \mathrm{d}(\varepsilon, S_1) = \mathrm{d}(S_1, \varepsilon) = \alpha |S_1|$$

où $M = \begin{array}{c} \\ A \\ C \\ G \\ T \end{array} \begin{array}{c} \begin{array}{cccc} A & C & G & T \end{array} \\ \begin{pmatrix} 0 & \gamma & \beta & \gamma \\ \gamma & 0 & \gamma & \beta \\ \beta & \gamma & 0 & \gamma \\ \gamma & \beta & \gamma & 0 \end{pmatrix} \end{array}$ avec α, β et γ des réels.

Remarque. Dans cette modélisation, α représente le coût d'insertion ou de délétion, et β et γ représentent des coûts de deux types de substitutions appelés respectivement transition et transvection.

3.2 Algorithme de Needleman-Wunsch

Pour calculer la distance de Levensthein, on utilise un algorithme de programmation dynamique dû à Needleman et Wunsch [3] :

Algorithme 3 : Algorithme de Needleman-Wunsch

Entrée : $(\mathsf{w}_1, \mathsf{w}_2) \in \mathcal{L}^2$, $\alpha \in \mathbf{R}$, $M \in \mathcal{M}_{|\Sigma|}(\mathbf{R})$

distances $\leftarrow$ table de longueur $(|\mathsf{w}_1| + 1) \times (|\mathsf{w}_2| + 1)$;

Pour j $\leftarrow 0$ **à** j $\leq |\mathsf{w}_2|$ **faire**
  Pour i $\leftarrow 0$ **à** i $\leq |\mathsf{w}_1|$ **faire**
    Si i $= 0$ **alors**
      distances[i][j] $\leftarrow$ j $\times \alpha$;
    Fin si
    Sinon si j $= 0$ **alors**
      distances[i][j] $\leftarrow$ i $\times \alpha$;
    Fin si
    Sinon
      distances[i][j] $\leftarrow \min($distances[i $-$ 1][j $-$ 1] $+ M[\mathsf{w}_1[$i $-$ 1]][$\mathsf{w}_2[$j $-$ 1]], distances[i][j $-$ 1] $+ \alpha$, distances[i $-$ 1][j] $+ \alpha)$;
    Fin
  Fin pour
Fin pour

Renvoyer distances$[|w_1|][|w_2|]$

Proposition 3.2.1. *L'algorithme 3 admet une complexité spatiale et temporelle en* $O(|w_1| \times |w_2|)$.

Démonstration. L'algorithme 3 introduit un tableau bidimensionnel de taille $|w_1| \times |w_2|$ et admet deux boucles pour imbriquées réalisant $|w_2|$ et $|w_1|$ tours. □

Remarque. L'algorithme a été linéarisé dans le programma réel en n'enregistrant que deux colonnes.

4 Création de l'arbre phylogénétique

L'algorithme UPGMA permet enfin de construire l'arbre phylogénétique à proprement parler :

Algorithme 4 : UPGMA

Entrée : distances, arbres phylogénétiques, `// arbres phylogénétiques est un tableau d'arbres phylogénétiques (chacun est représenté par un couple d'une chaîne de caractère et d'un pointeur) déjà créés initialisé par les noms des différentes espèces`
n `// n désigne le nombre d'espèces à traiter`

Si n ≤ 1 **alors**
  Renvoyer arbres phylogénétiques[0]
Fin si

`// recherche du minimum des coefficients non diagonaux`
minimum $\leftarrow$ distances[1][0];
$i_1 \leftarrow 1$;
$i_2 \leftarrow 0$;
Pour i $\leftarrow 1$ **à** i $<$ n **faire**
  Pour j $\leftarrow 0$ **à** j $<$ i **faire**
    Si distances[i][j] $<$ minimum **alors**
      $i_1 \leftarrow i$;
      $i_2 \leftarrow j$;
      minimum $\leftarrow$ distances[i][j];
    Fin si
  Fin pour
Fin pour

`// calcul des nouvelles distances`
Pour i $\leftarrow 0$ **à** i $<$ n **faire**
  distances[i][i_1] $\leftarrow$ (distances[i][i_2] + distances[i][i_1])/2;
  distances[i_1][i] $\leftarrow$ distances[i][i_1];
  distances[i][i_2] $\leftarrow$ distances[i][i_1];
  distances[i_2][i] $\leftarrow$ distances[i][i_1];
Fin pour

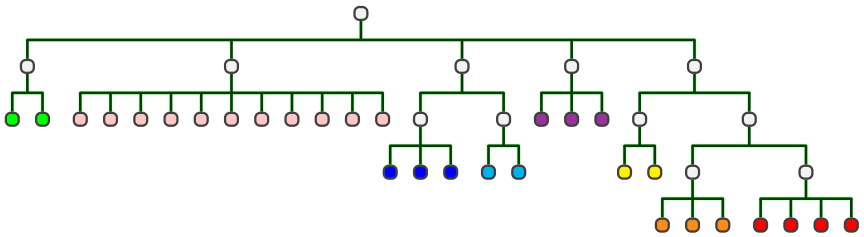
`// échange des lignes` i_1 `et n-1`
Pour i $\leftarrow 0$ **à** i $<$ n **faire**
  distances[i_1][i] $\leftarrow$ distances[n $-$ 1][i];
  distances[i][i_1] $\leftarrow$ distances[i_1][i];
Fin pour

nouvel arbre phylogénétique $\leftarrow$ ("ancêtre commun",[arbres phylogénétiques[i_1], arbres phylogénétiques[i_2]]);
arbres phylogénétiques[i_2] $\leftarrow$ nouvel arbre phylogénétique;
arbres phylogénétiques[i_1] $\leftarrow$ arbres phylogénétiques[n $-$ 1];

Renvoyer UPGMA(distances, arbres phylogénétiques, n $-$ 1)

5 Arbres créés

5.1 Arbre de référence



Familles : Alphasatellitidae, Anelloviridae, Betalipothrixvirus, Alphalipothrixvirus, Sphaerolipoviridae, Roniviridae, Betacoronavirus , Alphacoronaviridae.

FIGURE 3 – Arbre phylogénétique du NCBI [4]

5.2 Arbres obtenus à partir des algorithmes

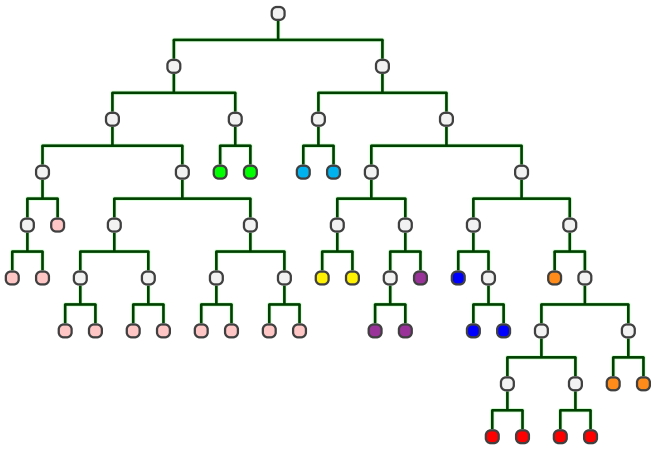


FIGURE 4 – Arbre pour $\alpha = \beta = \gamma = 1$

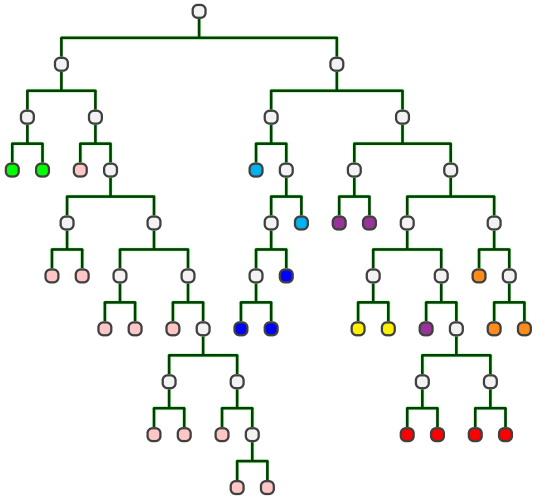


FIGURE 5 – Arbre pour $\alpha \in \{10, 11, 100\}$, $\beta = 1$ et $\gamma = 3$

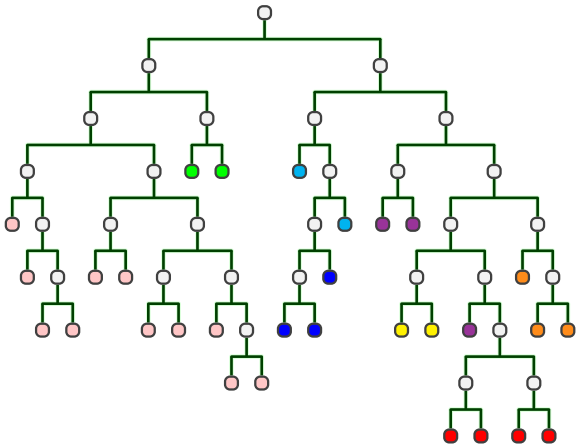


FIGURE 6 – Arbre pour $\alpha = 10$, $\beta = 1$ et $\gamma = 2$

6 Annexe

```
1  /* Quentin MOREAU
2  * TIPE Bioinformatique
3  * Reconstruction d'arbres phylogenetiques
4  *
5  * fichier d'entete main
6  *
7  * 2021-2022
8  */
9
```

```
10
11
12  #ifndef MAIN_H_INCLUDED
13  #define MAIN_H_INCLUDED
14
15  #include "construction_arbre_phylogenetiques.h"
16
17
18
19  typedef enum Nucleotide Nucleotide;
20  enum Nucleotide
21  {
22      A, C, G, T
23  };
24
25
26
27  typedef struct Element Element;
28  struct Element
29  {
30      int element;
31      Element* precedent;
32      Element* suivant;
33  };
34
35
36  typedef struct File File;
37  struct File
38  {
39      Element* premierElement;
40      Element* dernierElement;
41  };
42
43
44
45  void enfiler(File* file, int element);
46  int defiler(File* file);
47  void supprimer(File* file);
48  Arbre* arbrePhylogenetique(char* nomsFichiers[], char* noms[], int
        nombreSequence);
49
50
51
52  #endif // MAIN_H_INCLUDED
```

main.h

```
1  /* Quentin MOREAU
2  * TIPE Bioinformatique
3  * Reconstruction d'arbres phylogenetiques
4  *
5  * fichier source main
6  *
7  * 2021-2022
8  */
9
10
```

```
11
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <time.h>
15
16 #include "main.h"
17 #include "fichiers.h"
18 #include "recherche_motif.h"
19 #include "prediction_regions_codantes.h"
20 #include "alignement.h"
21 #include "construction_arbre_phylogenetique.h"
22
23
24
25
26 int main()
27 {
28
29
30     char* virusFichiers[30] = { "sequences/Coronaviridae/Middle
       East respiratory syndrome-related coronavirus.fasta",
31         "sequences/Coronaviridae/Human coronavirus 229E.fasta",
32         "sequences/Coronaviridae/Bat coronavirus HKU10.fasta",
33         "sequences/Coronaviridae/Bat coronavirus CDPHE15.fasta",
34         "sequences/Coronaviridae/Human coronavirus NL63.fasta",
35         "sequences/Coronaviridae/Human coronavirus HKU1.fasta",
36         "sequences/Coronaviridae/Severe acute respiratory syndrome-
       related coronavirus.fasta",
37         "sequences/Roniviridae/Gill-associated virus.fasta",
38         "sequences/Roniviridae/Yellow head virus.fasta",
39         "sequences/Sphaerolipoviridae/Haloarcula hispanica
       icosahedral virus 2.fasta",
40         "sequences/Sphaerolipoviridae/Haloarcula hispanica virus
       PH1.fasta",
41         "sequences/Sphaerolipoviridae/Haloarcula virus HCIV1.fasta"
       ,
42         "sequences/Adnaviria/Acidianus filamentous virus 3.fasta",
43         "sequences/Adnaviria/Acidianus filamentous virus 6.fasta",
44         "sequences/Adnaviria/Acidianus filamentous virus 8.fasta",
45         "sequences/Adnaviria/Sulfolobales Beppu filamentous virus
       2.fasta",
46         "sequences/Adnaviria/Sulfolobus filamentous virus 1.fasta",
47         "sequences/Alphasatellitidae/Ageratum yellow vein Singapore
        alphasatellite.fasta",
48         "sequences/Alphasatellitidae/Coconut foliar decay
       alphasatellite.fasta",
49         "sequences/Anelloviridae/Simian torque teno virus 30.fasta"
       ,
50         "sequences/Anelloviridae/Simian torque teno virus 31.fasta"
       ,
51         "sequences/Anelloviridae/Simian torque teno virus 32.fasta"
       ,
52         "sequences/Anelloviridae/Simian torque teno virus 34.fasta"
       ,
53         "sequences/Anelloviridae/Torque teno virus 10.fasta",
54         "sequences/Anelloviridae/Torque teno virus 11.fasta",
55         "sequences/Anelloviridae/Torque teno virus 12.fasta",
```

```
56          "sequences/Anelloviridae/Torque teno virus 13.fasta",
57          "sequences/Anelloviridae/Torque teno virus 14.fasta",
58          "sequences/Anelloviridae/Torque teno virus 15.fasta",
59          "sequences/Anelloviridae/Torque teno virus 16.fasta"
60      };
61
62      char* virusNoms[30] = {"Middle East respiratory syndrome-
        related coronavirus",
63          "Human coronavirus 229E",
64          "Bat coronavirus HKU10",
65          "Bat coronavirus CDPHE15",
66          "Human coronavirus NL63",
67          "Human coronavirus HKU1",
68          "Severe acute respiratory syndrome-related coronavirus",
69          "Gill-associated virus",
70          "Yellow head virus",
71          "Haloarcula hispanica icosahedral virus 2",
72          "Haloarcula hispanica virus PH1",
73          "Haloarcula virus HCIV1",
74          "Acidianus filamentous virus 3",
75          "Acidianus filamentous virus 6",
76          "Acidianus filamentous virus 8",
77          "Sulfolobales Beppu filamentous virus 2",
78          "Sulfolobus filamentous virus 1",
79          "Ageratum yellow vein Singapore alphasatellite",
80          "Coconut foliar decay alphasatellite",
81          "Simian torque teno virus 30",
82          "Simian torque teno virus 31",
83          "Simian torque teno virus 32",
84          "Simian torque teno virus 34",
85          "Torque teno virus 10",
86          "Torque teno virus 11",
87          "Torque teno virus 12",
88          "Torque teno virus 13",
89          "Torque teno virus 14",
90          "Torque teno virus 15",
91          "Torque teno virus 16"
92      };
93
94      arbrePhylogenetique(virusFichiers, virusNoms, 30);
95
96
97
98
99
100
101
102     return 0;
103 }
104
105
106
107
108
109
110
111
```

```
112 Arbre* arbrePhylogenetique(char* nomsFichiers[], char* noms[], int
        nombreSequence)
113 {
114     Nucleotide** sequencesLues = malloc(sizeof(Nucleotide*) *
        nombreSequence);
115     Nucleotide** sequences = malloc(sizeof(Nucleotide*) *
        nombreSequence);
116     int* tailles = malloc(sizeof(int) * nombreSequence);
117 
118     if (sequencesLues == NULL || sequences == NULL || tailles ==
        NULL)
119     {
120         printf("echec (arbrePhylogenetique)");
121         exit(EXIT_FAILURE);
122     }
123 
124     Nucleotide codonStop0[] = { T, A, A };
125     Nucleotide codonStop1[] = { T, A, G };
126     Nucleotide codonStop2[] = { T, G, A };
127     Nucleotide codonStart[] = { A, T, G };
128 
129 
130     AutomateSuffixes* automates[4] = { constructionAutomateSuffixes
        (codonStop0, 3),
131                                        constructionAutomateSuffixes(
        codonStop1, 3),
132                                        constructionAutomateSuffixes(
        codonStop2, 3),
133                                        constructionAutomateSuffixes(
        codonStart, 3)
134     };
135 
136 
137     for (int i = 0; i < nombreSequence; i++)
138     {
139         sequencesLues[i] = lireSequence(nomsFichiers[i]);
140 
141         Nucleotide* sequenceCodanteLue = extraireSequencesCodantes(
        sequencesLues[i] + 1, sequencesLues[i][0], automates);
142         sequences[i] = sequenceCodanteLue + 1;
143         tailles[i] = sequenceCodanteLue[0];
144 
145         if (tailles[i] == 0)
146         {
147             printf("pas de region codante trouvee");
148             exit(EXIT_FAILURE);
149         }
150     }
151 
152     int** distances = distancesLevenshtein(sequences, tailles,
        nombreSequence);
153 
154     Arbre* arbre = constructionArbreUPGMA(distances, noms,
        nombreSequence);
155 
156     afficherArbre(arbre);
157 
```

```
158         return NULL;
159
160     }
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176     ////////////////////////////////////////// FILES
            //////////////////////////////////////////
177
178
179
180
181
182     void enfiler(File* file, int element)
183     {
184
185         Element* dernierElement = file->dernierElement;
186         Element* nouvelElement = malloc(sizeof(Element));
187
188         if (nouvelElement == NULL)
189         {
190             printf("echec (enfiler)");
191             exit(EXIT_FAILURE);
192         }
193
194         nouvelElement->element = element;
195         nouvelElement->precedent = NULL;
196         nouvelElement->suivant = NULL;
197
198         if (dernierElement != NULL)
199         {
200             dernierElement->suivant = nouvelElement;
201             nouvelElement->precedent = dernierElement;
202         }
203         else
204         {
205             file->premierElement = nouvelElement;
206         }
207
208         file->dernierElement = nouvelElement;
209     }
210
211
212     int defiler(File* file)
213     {
```

```
214     Element* premierElement = file->premierElement;
215     if (premierElement == NULL)
216     {
217         return -1;
218     }
219     int element = premierElement->element;
220     file->premierElement = premierElement->suivant;
221     if (premierElement->suivant != NULL)
222     {
223         premierElement->suivant->precedent = NULL;
224     }
225     free(premierElement);
226     return element;
227 }
228
229
230
231 void supprimer(File* file)
232 {
233     while (file->premierElement != NULL)
234     {
235         defiler(file);
236     }
237 }
```

main.c

```
1  /* Quentin MOREAU
2  * TIPE Bioinformatique
3  * Reconstruction d'arbres phylogenetiques
4  *
5  * fichier d'entete fichiers
6  *
7  * 2021-2022
8  */
9
10
11
12 #ifndef FICHIERS_H_INCLUDED
13 #define FICHIERS_H_INCLUDED
14
15
16
17 Nucleotide* lireSequence(char* nom);
18 void afficherSequence(Nucleotide* sequence, int taille);
19
20
21
22
23
24
25 #endif // FICHIERS_H_INCLUDED
```

fichiers.h

```
1  /* Quentin MOREAU
2  * TIPE Bioinformatique
```

```c
 * Reconstruction d'arbres phylogenetiques
 *
 * fichier source fichier
 *
 * 2021-2022
 */


#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "main.h"
#include "fichiers.h"



Nucleotide* lireSequence(char* nom)
{
    FILE* fichier = NULL;
    fopen_s(&fichier, nom, "r");

    if (fichier == NULL)
    {
        printf("impossible d'ouvrir %s", nom);
        exit(EXIT_FAILURE);
    }

    fseek(fichier, 0, SEEK_END);
    int tailleFichier = ftell(fichier);
    rewind(fichier);

    char test = 'a';
    int curseur = 0;
    while (test != '\n')
    {
        curseur++;
        test = fgetc(fichier);
    }

    int lettre = 0;

    int taille = (tailleFichier-1 - curseur) - (tailleFichier-1 -
    curseur) / 61; // 60 carac par ligne + 1 \n
    Nucleotide* sequence = malloc(sizeof(Nucleotide) * (taille + 1)
    ); // on enregistre aussi la taille


    if (sequence == NULL)
    {
        printf("echec (lireSequence 1)");
        exit(EXIT_FAILURE);
    }


    sequence[0] = taille;
    int i = 1;
```

```
58
59      while (lettre != EOF)
60      {
61          lettre = getc(fichier);
62          if (lettre != '\n' && lettre != EOF)
63          {
64              switch (lettre)
65              {
66              case 'A':
67                  sequence[i] = A;
68                  break;
69              case 'C':
70                  sequence[i] = C;
71                  break;
72              case 'G':
73                  sequence[i] = G;
74                  break;
75              case 'T':
76                  sequence[i] = T;
77                  break;
78              default:
79                  printf("%c",lettre);
80                  printf("\n\nERROR\n\n");
81              }
82              i += 1;
83          }
84      }
85
86
87      fclose(fichier);
88
89
90      return sequence;
91  }
92
93
94
95  void afficherSequence(Nucleotide* sequence, int taille)
96  {
97      for (int i = 0; i < taille; i++)
98      {
99          switch (sequence[i])
100         {
101         case A:
102             printf("A");
103             break;
104         case C:
105             printf("C");
106             break;
107         case G:
108             printf("G");
109             break;
110         case T:
111             printf("T");
112             break;
113         default:
114             printf(" erreur " );
```

```
                break;
            }
        }
    }
```

fichiers.c

```
/* Quentin MOREAU
* TIPE Bioinformatique
* Reconstruction d'arbres phylogenetiques
*
* fichier entete recherche_motif
*
* 2021-2022
*/



#ifndef RECHERCHE_MOTIF_H_INCLUDED
#define RECHERCHE_MOTIF_H_INCLUDED

#include "main.h"







typedef struct AutomateSuffixes AutomateSuffixes;
struct AutomateSuffixes
{
    int etatFinal;
    int** transitions;
};




File* rechercheNaive(Nucleotide sequence[], Nucleotide motif[], int tailleSequence, int tailleMotif);
File* boyerMoore(Nucleotide sequence[], Nucleotide motif[], int tailleSequence, int tailleMotif);
AutomateSuffixes* constructionAutomateSuffixes(Nucleotide motif[], int taille);
File* rechercheAutomateSuffixes(AutomateSuffixes* automate, Nucleotide sequence[], int tailleSequence);



#endif // RECHERCHE_MOTIF_H_INCLUDED
```

recherche_motif.h

```
/* Quentin MOREAU
* TIPE Bioinformatique
* Reconstruction d'arbres phylogenetiques
```

```
4  *
5  * fichier source recherche_motif
6  *
7  * 2021-2022
8  */
9
10
11
12 #include <stdlib.h>
13 #include <stdio.h>
14
15 #include "recherche_motif.h"
16
17
18
19
20
21
22
23
24
25 ////////////////////////////////////////// RECHERCHE NAIVE
       ////////////////////////////////////////////
26
27
28
29
30
31 File* rechercheNaive(Nucleotide sequence[], Nucleotide motif[], int
       tailleSequence, int tailleMotif)
32 {
33     File* indices = malloc(sizeof(File));
34
35     if (indices == NULL)
36     {
37         exit(EXIT_FAILURE);
38     }
39
40     indices->dernierElement = NULL;
41     indices->premierElement = NULL;
42
43     for (int i = 0; i < tailleSequence - tailleMotif; i++)
44     {
45         int indiceMotif = 0;
46         int test = 1;
47
48         while (test)
49         {
50             if (indiceMotif<tailleMotif)
51             {
52                 test = sequence[i + indiceMotif] == motif[
       indiceMotif];
53                 indiceMotif++;
54             }
55             else
56             {
57                 test = 0;
```

```
58              }
59          }
60
61          if (indiceMotif >= tailleMotif)
62          {
63              enfiler(indices, i);
64          }
65      }
66
67
68      return indices;
69  }
70
71
72
73
74
75
76
77
78
79  ////////////////////////////////////////// BOYER MOORE
        ////////////////////////////////////////////
80
81
82
83
84
85
86  File* boyerMoore(Nucleotide sequence[], Nucleotide motif[], int
        tailleSequence, int tailleMotif)
87  {
88      File* indices = malloc(sizeof(File));
89
90      if (indices == NULL)
91      {
92          exit(EXIT_FAILURE);
93      }
94
95      indices->dernierElement = NULL;
96      indices->premierElement = NULL;
97
98      int table[4] = { -1, -1, -1, -1 };
99
100     for (int i = 0; i < tailleMotif; i++)
101     {
102         table[motif[i]] = i;
103     }
104
105
106     for (int i = 0; i < tailleSequence + 1 - tailleMotif; i++)
107     {
108         int indiceMotif = tailleMotif - 1;
109         int test = 1;
110
111         while (test)
112         {
```

```
113             if (indiceMotif >= 0)
114             {
115                 test = sequence[i + indiceMotif] == motif[
    indiceMotif];
116                 indiceMotif--;
117             }
118             else
119             {
120                 test = 0;
121             }
122         }
123 
124         if (indiceMotif < 0) // le motif est apparu
125         {
126             enfiler(indices, i);
127             if (i + tailleMotif < tailleSequence)
128             {
129                 i += tailleMotif - table[sequence[i + tailleMotif]]
     - 1;
130             }
131         }
132 
133         else // le motif n'est pas apparu
134         {
135             i += max(0, indiceMotif-table[sequence[i + indiceMotif
    ]] - 1);
136         }
137 
138 
139     }
140 
141 
142     return indices;
143 }
144 
145 
146 
147 
148 
149 
150 
151 
152 
153 
154 
155 
156 ////////////////////////////////////////// AUTOMATE DES SUFFIXES
    //////////////////////////////////////////
157 
158 
159 
160 
161 
162 
163 
164 
165 AutomateSuffixes* constructionAutomateSuffixes(Nucleotide motif[],
```

```
        int taille)
166 {
167
168     AutomateSuffixes* automate = malloc(sizeof(AutomateSuffixes));
169     int** transitions = malloc(sizeof(int*) * (taille+1));
170
171     if (automate == NULL || transitions == NULL)
172     {
173         exit(EXIT_FAILURE);
174     }
175
176     automate->etatFinal = taille;
177     automate->transitions = transitions;
178
179     for (int i = 0; i <= taille; i++)
180     {
181         transitions[i] = malloc(sizeof(int) * 4);
182         if (transitions[i] == NULL)
183         {
184             exit(EXIT_FAILURE);
185         }
186     }
187
188
189
190     for (int i = 0; i <= taille; i++)
191     {
192         for (int j = 0; j < 4; j++)
193         {
194             transitions[i][j] = 0;
195         }
196     }
197
198
199
200     for (int i = 1; i <= taille; i++)
201     {
202
203         int ancienneTransition = transitions[i - 1][motif[i - 1]];
        // enregistrement de l'ancienne transition
204
205
206         transitions[i - 1][motif[i - 1]] = i;
207
208         for (int j = 0; j < 4; j++)
209         {
210             transitions[i][j] = transitions[ancienneTransition][j];
211         }
212
213     }
214
215
216
217     return automate;
218 }
219
220
```

```
221
222
223
224
225
226
227 File* rechercheAutomateSuffixes(AutomateSuffixes* automate,
        Nucleotide sequence[], int tailleSequence)
228 {
229
230     File* indices = malloc(sizeof(File));
231
232     if (indices == NULL)
233     {
234         exit(EXIT_FAILURE);
235     }
236
237     indices->dernierElement = NULL;
238     indices->premierElement = NULL;
239
240
241
242
243
244     int etat = 0;
245
246     for (int i = 0; i < tailleSequence; i++)
247     {
248         etat = automate->transitions[etat][sequence[i]];
249
250         if (etat == automate->etatFinal)
251         {
252             enfiler(indices, i + 1 - automate->etatFinal); //
        automate->etatFinal == taille du motif
253         }
254     }
255
256     return indices;
257 }
```

recherche_motif.c

```
1  /* Quentin MOREAU
2  * TIPE Bioinformatique
3  * Reconstruction d'arbres phylogenetiques
4  *
5  * fichier d'entete prediction_regions_codantes
6  *
7  * 2021-2022
8  */
9
10
11
12 #ifndef PREDICTION_REGIONS_CODANTES_H_INCLUDED
13 #define PREDICTION_REGIONS_CODANTES_H_INCLUDED
14
15 #define TAILLE 300
```

```
16
17 #define STOP0 0
18 #define STOP1 1
19 #define STOP2 2
20 #define START 3
21
22
23 File* rechercheCodonsStart(Nucleotide sequence[], int taille,
       AutomateSuffixes** automates);
24 File* rechercheCodonsStop(Nucleotide sequence[], int taille,
       AutomateSuffixes** automates);
25 File* predictionORF(Nucleotide sequence[], int taille,
       AutomateSuffixes** automates);
26 Nucleotide* extraireSequencesCodantes(Nucleotide sequence[], int
       taille, AutomateSuffixes** automates);
27
28
29 #endif // PREDICTION_REGIONS_CODANTES_H_INCLUDED
```

prediction_regions_codantes.h

```
1  /* Quentin MOREAU
2  * TIPE Bioinformatique
3  * Reconstruction d'arbres phylogenetiques
4  *
5  * fichier source prediction_regions_codantes
6  *
7  * 2021-2022
8  */
9
10
11
12 #include <stdio.h>
13 #include <stdlib.h>
14
15 #include "main.h"
16 #include "recherche_motif.h"
17 #include "prediction_regions_codantes.h"
18
19
20
21
22 File* rechercheCodonsStart(Nucleotide sequence[], int taille,
       AutomateSuffixes** automates)
23 {
24
25     if (automates != NULL)
26     {
27         return rechercheAutomateSuffixes(automates[3], sequence,
       taille);
28     }
29     else
30     {
31         Nucleotide codonStart[] = { A,T,G };
32         return boyerMoore(sequence, codonStart, taille, 3);
33     }
34 }
```

```
35
36
37
38  File* rechercheCodonsStop(Nucleotide sequence[], int taille,
        AutomateSuffixes** automates)
39  {
40
41      File* codonsStop0 = NULL;
42      File* codonsStop1 = NULL;
43      File* codonsStop2 = NULL;
44
45      if (automates != NULL)
46      {
47          codonsStop0 = rechercheAutomateSuffixes(automates[STOP0],
        sequence, taille);
48          codonsStop1 = rechercheAutomateSuffixes(automates[STOP1],
        sequence, taille);
49          codonsStop2 = rechercheAutomateSuffixes(automates[STOP2],
        sequence, taille);
50      }
51      else
52      {
53          Nucleotide codonStop0[] = { T, A, A };
54          Nucleotide codonStop1[] = { T, A, G };
55          Nucleotide codonStop2[] = { T, G, A };
56          codonsStop0 = boyerMoore(sequence, codonStop0, taille, 3);
57          codonsStop1 = boyerMoore(sequence, codonStop1, taille, 3);
58          codonsStop2 = boyerMoore(sequence, codonStop2, taille, 3);
59      }
60
61
62      File* codonsStop = malloc(sizeof(File));
63
64      if (codonsStop == NULL)
65      {
66          exit(EXIT_FAILURE);
67      }
68
69      codonsStop->dernierElement = NULL;
70      codonsStop->premierElement = NULL;
71
72
73      while (codonsStop0->premierElement != NULL)
74      {
75          if (codonsStop1->premierElement != NULL)
76          {
77              if (codonsStop2->premierElement != NULL)
78              {
79                  if (codonsStop0->premierElement->element <
        codonsStop1->premierElement->element)
80                  {
81                      if (codonsStop0->premierElement->element <
        codonsStop2->premierElement->element)
82                      {
83                          enfiler(codonsStop, defiler(codonsStop0));
84                      }
85                      else
```

```
86                      {
87                          enfiler(codonsStop, defiler(codonsStop2));
88                      }
89                  }
90                  else
91                  {
92                      if (codonsStop1->premierElement->element <
    codonsStop2->premierElement->element)
93                      {
94                          enfiler(codonsStop, defiler(codonsStop1));
95                      }
96                      else
97                      {
98                          enfiler(codonsStop, defiler(codonsStop2));
99                      }
100                 }
101             } // codonStopActuel2 = NULL
102
103             else
104             {
105                 if (codonsStop0->premierElement->element <
    codonsStop1->premierElement->element)
106                 {
107                     enfiler(codonsStop, defiler(codonsStop0));
108                 }
109                 else
110                 {
111                     enfiler(codonsStop, defiler(codonsStop1));
112                 }
113             }
114         } // codonStopActuel1 = NULL
115
116         else
117         {
118             if (codonsStop2->premierElement != NULL)
119             {
120                 if (codonsStop0->premierElement->element <
    codonsStop2->premierElement->element)
121                 {
122                     enfiler(codonsStop, defiler(codonsStop0));
123                 }
124                 else
125                 {
126                     enfiler(codonsStop, defiler(codonsStop2));
127                 }
128             } // codonStopActuel2 = NULL
129             else
130             {
131                 enfiler(codonsStop, defiler(codonsStop0));
132             }
133         }
134     } // codonStopActuel0 = NULL
135
136     while (codonsStop1->premierElement != NULL)
137     {
138         if (codonsStop2->premierElement != NULL)
139         {
```

```
140             if (codonsStop1->premierElement->element < codonsStop2
    ->premierElement->element)
141             {
142                 enfiler(codonsStop, defiler(codonsStop1));
143             }
144             else
145             {
146                 enfiler(codonsStop, defiler(codonsStop2));
147             }
148         }
149 
150         else
151         {
152             enfiler(codonsStop, defiler(codonsStop1));
153         }
154     }
155 
156     while (codonsStop2->premierElement != NULL)
157     {
158         enfiler(codonsStop, defiler(codonsStop2));
159     }
160 
161     return codonsStop;
162 }
163 
164 
165 
166 
167 
168 
169 File* predictionORF(Nucleotide sequence[], int taille,
    AutomateSuffixes** automates)
170 {
171     File* codonsStart = rechercheCodonsStart(sequence, taille,
    automates);
172     File* codonsStop = rechercheCodonsStop(sequence, taille,
    automates);
173 
174 
175     File* orfs = malloc(sizeof(File));
176     orfs->dernierElement = NULL;
177     orfs->premierElement = NULL;
178 
179 
180     int tailleSequenceCodante = 0;
181 
182 
183 
184     while (codonsStart->premierElement != NULL && codonsStop->
    premierElement != NULL)
185     {
186         // recherche de deux codons stops en phase successifs
    distants d'au moins TAILLE
187         int indiceCodonStop0 = defiler(codonsStop); // premier
    codon stop
188         int indiceCodonStop1 = defiler(codonsStop); // deuxieme
    codon stop
```

```
189         int invalide = (indiceCodonStop1 - indiceCodonStop0) % 3 !=
        0 || (indiceCodonStop1 - indiceCodonStop0) <= TAILLE; // = 1 !
190         int fileVide = 0;
191
192
193         while (invalide)
194         {
195             if (codonsStop->premierElement == NULL)
196             {
197                 invalide = 0;
198                 fileVide = 1;
199             }
200             else if ((indiceCodonStop1 - indiceCodonStop0) % 3 == 0
        && (indiceCodonStop1 - indiceCodonStop0) < TAILLE)
201             {
202                 int indiceCodonStopSuivant = defiler(codonsStop);
203                 indiceCodonStop0 = indiceCodonStop1;
204                 indiceCodonStop1 = indiceCodonStopSuivant; // il ne
        doit pas y avoir de codon stop entre les deux !
205             }
206             else if ((indiceCodonStop1 - indiceCodonStop0) % 3 !=
        0)
207             {
208                 indiceCodonStop1 = defiler(codonsStop);
209             }
210             else
211             {
212                 invalide = 0;
213             }
214         }
215
216
217         if (!fileVide) // on a trouve deux codons stops ideaux
218         {
219
220             if (codonsStart->premierElement != NULL)
221             {
222                 int indiceCodonStart = defiler(codonsStart);
223
224                 invalide = indiceCodonStart > indiceCodonStop0 &&
        indiceCodonStart < indiceCodonStop1 && (indiceCodonStart -
        indiceCodonStop0) % 3 == 0 && indiceCodonStop1 -
        indiceCodonStart >= TAILLE;
225                 fileVide = 0;
226
227                 while (invalide)
228                 {
229                     if (codonsStart->premierElement == NULL)
230                     {
231                         invalide = 0;
232                         fileVide = 1;
233                     }
234                     else if (indiceCodonStart > indiceCodonStop1)
235                     {
236                         invalide = 0; // le codon start doit etre
        entre les deux codons stop donc on sort
237                     }
```

```
238                     else if ((indiceCodonStart - indiceCodonStop0)
        % 3 != 0 || (indiceCodonStop1 - indiceCodonStart) < TAILLE ||
        indiceCodonStart < indiceCodonStop0)
239                     {
240                         indiceCodonStart = defiler(codonsStart);
241                     }
242                     else
243                     {
244                         invalide = 0;
245                     }
246                 }
247 
248 
249                 // ajout
250                 if (!fileVide)
251                 {
252 
253                     enfiler(orfs, indiceCodonStart);
254                     enfiler(orfs, indiceCodonStop1);
255 
256                     tailleSequenceCodante += indiceCodonStop1 - 3 -
         indiceCodonStart;
257 
258 
259 
260                     // decalage de codonStartActuel pour que l'
        indice du codon stop soit superieur a celui du codon start
261 
262                     int codonStartInvalide = codonsStart->
        premierElement != NULL;
263 
264                     while (codonStartInvalide)
265                     {
266                         if (codonsStart->premierElement->element >
        indiceCodonStop1) // le codon start suivant fonctionne
267                         {
268                             codonStartInvalide = 0;
269                         }
270                         else if (codonsStart->premierElement->
        suivant != NULL)
271                         {
272                             if (codonsStart->premierElement->
        suivant->element > indiceCodonStop1)
273                             {
274                                 defiler(codonsStart);
275                                 codonStartInvalide = 0;
276                             }
277                             else
278                             {
279                                 defiler(codonsStart);
280                             }
281                         }
282                         else
283                         {
284                             defiler(codonsStart);
285                             codonStartInvalide = 0;
286                         }
```

```
287                         }
288                     }
289                 }
290             }
291 
292         }
293 
294         // ajout de la taille en tete de file
295 
296         Element* premierElement = orfs->premierElement;
297         Element* nouvelElement = malloc(sizeof(Element));
298         nouvelElement->element = tailleSequenceCodante;
299         nouvelElement->precedent = NULL;
300         nouvelElement->suivant = NULL;
301 
302         if (premierElement != NULL)
303         {
304             premierElement->precedent = nouvelElement;
305             nouvelElement->suivant = premierElement;
306         }
307         else
308         {
309             orfs->dernierElement = nouvelElement;
310         }
311 
312         orfs->premierElement = nouvelElement;
313 
314         return orfs;
315     }
316 
317 
318 
319     Nucleotide* extraireSequencesCodantes(Nucleotide sequence[], int
        taille, AutomateSuffixes** automates)
320     {
321         File* orfsFile = predictionORF(sequence, taille, automates);
322 
323         int tailleSequenceCodante = defiler(orfsFile);
324 
325         Nucleotide* orfs = malloc(sizeof(int) * (tailleSequenceCodante
            + 1));
326         int indice = 1;
327 
328         if (orfs == NULL)
329         {
330             exit(EXIT_FAILURE);
331         }
332 
333         orfs[0] = tailleSequenceCodante;
334 
335 
336 
337         while (orfsFile->premierElement != NULL)
338         {
339             int indiceStartCodon = defiler(orfsFile);
340             int indiceStopCodon = defiler(orfsFile);
341 
```

```
342             for (int i = indiceStartCodon; i < indiceStopCodon - 3; i
        ++)
343             {
344                 orfs[indice] = sequence[i];
345                 indice++;
346             }
347         }
348 
349 
350 
351 
352         return orfs;
353     }
```

prediction_regions_codantes.c

```
1   /* Quentin MOREAU
2   * TIPE Bioinformatique
3   * Reconstruction d'arbres phylogenetiques
4   *
5   * fichier d'entete alignement
6   *
7   * 2021-2022
8   */
9   
10  
11  #ifndef ALIGNEMENT_H_INCLUDED
12  #define ALIGNEMENT_H_INCLUDED
13  
14  #define TRANSITION 1
15  #define TRANSVERSION 3
16  #define COUT_INSERTION_DELETION 10
17  
18  
19  int alignementRecursif(Nucleotide sequence1[], Nucleotide sequence2
        [], int tailleSequence1, int tailleSequence2);
20  int coutRecursif(Nucleotide sequence1[], Nucleotide sequence2[],
        int indice1, int indice2);
21  int needlemanWunsch(Nucleotide sequence1[], Nucleotide sequence2[],
         int tailleSequence1, int tailleSequence2);
22  int** distancesLevenshtein(Nucleotide** sequences, int* tailles,
        int nombreSequence);
23  
24  
25  #endif // ALIGNEMENT_H_INCLUDED
```

alignement.h

```
1   /* Quentin MOREAU
2   * TIPE Bioinformatique
3   * Reconstruction d'arbres phylogenetiques
4   *
5   * fichier source alignement
6   *
7   * 2021-2022
8   */
9   
```

```c

#include <stdlib.h>
#include <stdio.h>

#include "main.h"
#include "alignement.h"



int alignementRecursif(Nucleotide sequence1[], Nucleotide sequence2[], int tailleSequence1, int tailleSequence2)
{
    return coutRecursif(sequence1, sequence2, tailleSequence1 - 1, tailleSequence2 - 1);
}


int coutRecursif(Nucleotide sequence1[], Nucleotide sequence2[], int indice1, int indice2)
{

    int const COUTS_SUBSTITUTIONS[4][4] = { {0,TRANSVERSION, TRANSITION,TRANSVERSION},
                                            {TRANSVERSION,0, TRANSVERSION,TRANSITION},
                                            {TRANSITION, TRANSVERSION,0,TRANSVERSION},
                                            {TRANSVERSION, TRANSITION,TRANSVERSION,0} };

    if (indice1 == 0 && indice2 == 0) return 0;
    if (indice1 == 0) return indice2 * COUT_INSERTION_DELETION;
    if (indice2 == 0) return indice1 * COUT_INSERTION_DELETION;

    return min(coutRecursif(sequence1, sequence2, indice1 - 1, indice2 - 1) + COUTS_SUBSTITUTIONS[sequence1[indice1]][sequence2[indice2]],
        min(coutRecursif(sequence1, sequence2, indice1, indice2 - 1) + COUT_INSERTION_DELETION,
            coutRecursif(sequence1, sequence2, indice1 - 1, indice2) + COUT_INSERTION_DELETION));
}




int** distancesLevenshtein(Nucleotide** sequences, int* tailles, int nombreSequence)
{
    int** distances = malloc(sizeof(int*) * nombreSequence);

    if (distances == NULL)
    {
        exit(EXIT_FAILURE);
    }
```

```
55
56      for (int i = 0; i < nombreSequence; i++)
57      {
58          distances[i] = malloc(sizeof(int) * nombreSequence);
59          if (distances[i] == NULL)
60          {
61              exit(EXIT_FAILURE);
62          }
63
64          for (int j = 0; j < nombreSequence; j++)
65          {
66              distances[i][j] = 0;
67          }
68      }
69
70      for (int i = 1; i < nombreSequence; i++)
71      {
72          for (int j = 0; j < i; j++)
73          {
74              //distances[i][j] =  alignementRecursif(sequences[i],
    sequences[j], tailles[i], tailles[j]);
75              distances[i][j] = needlemanWunsch(sequences[i],
    sequences[j], tailles[i], tailles[j]);
76              distances[j][i] = distances[i][j];
77          }
78      }
79
80      return distances;;
81  }
82
83
84
85
86
87
88
89
90
91  int needlemanWunsch(Nucleotide sequence1[], Nucleotide sequence2[],
     int tailleSequence1, int tailleSequence2)
92  {
93
94      if (tailleSequence2 < tailleSequence1)
95      {
96          return needlemanWunsch(sequence2, sequence1,
    tailleSequence2, tailleSequence1);
97      }
98
99
100     int* colonnes[2] = { NULL,NULL };
101
102     colonnes[0] = malloc(sizeof(int) * (tailleSequence1 + 1));
103     colonnes[1] = malloc(sizeof(int) * (tailleSequence1 + 1));
104
105     int const COUTS_SUBSTITUTIONS[4][4] = { {0,TRANSVERSION,
    TRANSITION,TRANSVERSION},
106                                             {TRANSVERSION,0,
```

```
        TRANSVERSION, TRANSITION},
107                                                 {TRANSITION,
        TRANSVERSION, 0 , TRANSVERSION},
108                                                 {TRANSVERSION,
        TRANSITION, TRANSVERSION, 0}  };
109
110
111     if (colonnes[0] == NULL || colonnes[1] == NULL)
112     {
113         exit(EXIT_FAILURE);
114     }
115
116     for (int i = 0; i <= tailleSequence1; i++)
117     {
118         colonnes[0][i] = i * COUT_INSERTION_DELETION;
119     }
120     for (int i = 0; i <= tailleSequence1; i++)
121     {
122         colonnes[1][i] = i * COUT_INSERTION_DELETION;
123     }
124
125
126
127     int k = 0;
128
129     while (k < tailleSequence2)
130     {
131
132         colonnes[k % 2][0] = (k+1) * COUT_INSERTION_DELETION;
133
134         for (int i = 0; i < tailleSequence1; i++)
135         {
136             colonnes[k % 2][i + 1] = min(colonnes[(k + 1) % 2][i] +
        COUTS_SUBSTITUTIONS[sequence1[i]][sequence2[k]], /* \ */
137                 min(colonnes[k % 2][i] + COUT_INSERTION_DELETION,
        /* | */
138                     colonnes[(k + 1) % 2][i + 1] +
        COUT_INSERTION_DELETION)); /* - */
139         }
140
141         k++;
142     }
143
144
145
146
147     int cout = colonnes[(k + 1) % 2][tailleSequence1];
148
149
150
151     free(colonnes[0]);
152     free(colonnes[1]);
153
154
155     return cout;
156
157 }
```

alignement.c

```c
/* Quentin MOREAU
* TIPE Bioinformatique
* Reconstruction d'arbres phylogenetiques
*
* fichier d'entete construction_arbre_phylogenetique
*
* 2021-2022
*/



#ifndef CONSTRUCTION_ARBRE_PHYLOGENETIQUE_H_INCLUDED
#define CONSTRUCTION_ARBRE_PHYLOGENETIQUE_H_INCLUDED


#include "main.h"


typedef struct Arbre Arbre;
struct Arbre
{
    char* nom;
    Arbre** arbres;
};


void afficherArbre(Arbre* arbre);
void afficherArbreRecursif(Arbre* arbre, int espace);
Arbre* constructionArbreUPGMA(int** distances, char** noms, int
    taille);
Arbre* constructionArbreUPGMARecursif(int** distances, Arbre**
    arbres, int taille);



#endif // CONSTRUCTION_ARBRE_PHYLOGENETIQUE_H_INCLUDED
```

construction_arbre_phylogenetique.h

```c
/* Quentin MOREAU
* TIPE Bioinformatique
* Reconstruction d'arbres phylogenetiques
*
* fichier source construction_arbre_phylogenetique
*
* 2021-2022
*/


#include <stdio.h>
#include <stdlib.h>
#include <string.h>

```

```c
#include "main.h"
#include "construction_arbre_phylogenetique.h"









////////////////////////////////////////// AFFICHAGE ARBRE
    //////////////////////////////////////////


void afficherArbre(Arbre* arbre)
{
    afficherArbreRecursif(arbre, 0);
}

void afficherArbreRecursif(Arbre* arbre, int espace)
{
    if (arbre != NULL)
    {
        for (int i = 0; i < espace; i++) printf(" ");
        printf(" -> ");
        printf(arbre->nom);

        if (arbre->arbres != NULL)
        {
            printf("\n");
            afficherArbreRecursif(arbre->arbres[0], espace + 4 +
    strlen(arbre->nom));
            printf("\n");
            afficherArbreRecursif(arbre->arbres[1], espace + 4 +
    strlen(arbre->nom));
            printf("\n");
        }

    }
}









////////////////////////////////////////// UPGMA
    //////////////////////////////////////////


Arbre* constructionArbreUPGMA(int** distances, char** noms, int
    taille)
{
```

```
67      Arbre** arbres = malloc(sizeof(Arbre*) * taille);
68
69      if (arbres == NULL)
70      {
71          exit(EXIT_FAILURE);
72      }
73
74      for (int i = 0; i < taille; i++)
75      {
76          arbres[i] = malloc(sizeof(Arbre));
77          arbres[i]->arbres = NULL;
78          arbres[i]->nom = noms[i];
79      }
80
81      return constructionArbreUPGMARecursif(distances, arbres, taille
        );
82  }
83
84
85  Arbre* constructionArbreUPGMARecursif(int** distances, Arbre**
        arbres, int taille)
86  {
87      if (taille <= 1)
88      {
89          return arbres[0];
90      }
91
92      int indice1 = 1, indice2 = 0;
93      int minimum = distances[1][0];
94
95      // recherche de minimum
96      for (int i = 1; i < taille; i++)
97      {
98          for (int j = 0; j < i; j++)
99          {
100             if (distances[i][j] < minimum) // on a indice2<indice1
101             {
102                 indice1 = i;
103                 indice2 = j;
104                 minimum = distances[i][j];
105             }
106         }
107     }
108
109     // distances recalculees
110
111     for (int i = 0; i < taille; i++)
112     {
113         distances[i][indice1] = (distances[i][indice2] + distances[
        i][indice1]) / 2;
114         distances[indice1][i] = distances[i][indice1];
115         distances[i][indice2] = distances[i][indice1];
116         distances[indice2][i] = distances[i][indice1];
117     }
118
119     // insertion de la ligne taille-1 dans la ligne indice1
120     for (int i = 0; i < taille; i++)
```

```
121     {
122         distances[indice1][i] = distances[taille - 1][i];
123         distances[i][indice1] = distances[indice1][i];
124     }
125
126
127
128     if (arbres[indice1] == NULL || arbres[indice2] == NULL)
129     {
130         exit(EXIT_FAILURE);
131     }
132
133     Arbre** sousArbres = malloc(2 * sizeof(Arbre*));
134     Arbre* nouvelArbre = malloc(sizeof(Arbre));
135
136     if (sousArbres == NULL || nouvelArbre == NULL)
137     {
138         exit(EXIT_FAILURE);
139     }
140
141     sousArbres[0] = arbres[indice1];
142     sousArbres[1] = arbres[indice2];
143
144     nouvelArbre->nom = "ancetre commun";
145     nouvelArbre->arbres = sousArbres;
146
147     arbres[indice2] = nouvelArbre;
148     arbres[indice1] = arbres[taille - 1];
149
150
151     return constructionArbreUPGMARecursif(distances, arbres, taille
        - 1);
152
153
154
155 }
```

construction_arbre_phylogenetique.c

Références

[1] IPBES. *Échapper à l'«ère des pandémies» : Les experts mettent en garde contre de pires crises à venir ; Options proposées pour réduire les risques.* URL : `https://ipbes.net/sites/default/files/2020-11/20201029%20Media%20Release%20IPBES%20Pandemics%20Workshop%20Report%20FR_Final_0.pdf`.

[2] François RECHENMANN. *À la recherche des régions codantes.* URL : `https://interstices.info/a-la-recherche-de-regions-codantes/`.

[3] Christian D. Wunsch SAUL B. NEEDLEMAN. "A general method applicable to the search for similarities in the amino acid sequence of two proteins". In : *Journal of Molecular Biology* 48 (1970), p. 443-453. ISSN : 0022-2836.

[4] NCBI. *Taxonomy browser (Viruses)*. URL : https://www.ncbi.nlm.nih.gov/Taxonomy/Browser/wwwtax.cgi?mode=Undef&name=Viruses&lvl=3&srchmode=1&keep=1&unlock.