Sorbonne Université

Rapport de projet

Alignement de séquences

Adham Noureldin
Enzo Roaldes
DM-IM 2022 L3

Encadrant : O. Spanjaard
Responsable : F. Pascual

19 Septembre 2022 — 30 Novembre 2022

Table des matières

1 Le problème d'alignement de séquences

1.1 Alignement de deux mots

Question 1 :
Tout d'abord, montrons que $\pi(a.b) = \pi(a)\pi(b)$, pour a, b deux mots quelconques.
Soit a de longueur n contenant k_1 gaps et b de longueur m contenant k_2 gaps.
Alors $a.b$ est de longueur $n + m$ contenant $k_1 + k_2$ gaps.
Ainsi, enlever les $k_1 + k_2$ à $a.b$ revient à enlever les k_1 gaps à a puis les k_2 gaps à b puis concaténer : nous aurons le même mot, de longueur $n + m - k_1 - k_2$ avec les mêmes lettres dans le même ordre et sans gaps.
Vérifions maintenant que $(\overline{x}.\overline{u}, \overline{y}.\overline{v})$ est un alignement de $(x.u, y.v)$.

1. Nous avons $\pi(\overline{x}.\overline{u}) = \pi(\overline{x}).\pi(\overline{u}) = x.u$, en utilisant pour la dernière égalité le fait que $(\overline{x}, \overline{y})$ et $(\overline{u}, \overline{v})$ sont des alignement de (x, y) et (u, v) respectivement.
2. Un raisonnement similaire permet de conclure que $\pi(\overline{y}.\overline{v}) = y.v$
3. Comme la longueur de la concaténation de deux mots est la somme des longueurs des deux mots, nous avons $|\overline{x}.\overline{u}| = |\overline{x}| + |\overline{u}| = |\overline{y}| + |\overline{v}| = |\overline{y}.\overline{v}|$
4. Vérifions enfin que $\forall i \in [1, \ldots, |\overline{x}.\overline{u}|], (\overline{x}.\overline{u})_i \neq -$ ou $(\overline{y}.\overline{v})_i \neq -$.
   Pour cela, faisons une distinction de cas :
   - Si $i \leq |\overline{x}|$, alors $(\overline{x}.\overline{u})_i = \overline{x}_i \neq -$ ou $(\overline{y}.\overline{v})_i = \overline{y}_i \neq -$.
   - Si $i > |\overline{x}|$, alors $(\overline{x}.\overline{u})_i = \overline{u}_i \neq -$ ou $(\overline{y}.\overline{v})_i = \overline{v}_i \neq -$.

On a donc montré que $(\overline{x}.\overline{u}, \overline{y}.\overline{v})$ est un alignement de $(x.u, y.v)$.

Question 2 : Soit $x, y \in \Sigma^*$, tels que $|x| = n$ et $|y| = m$.
Prenons $\overline{x}, \overline{y}$ tels que chaque caractère de Σ dans $\overline{x}$ se trouve en face d'un gap $-$ dans $\overline{y}$ et que chaque caractère de Σ dans $\overline{y}$ se trouve en face d'un gap $-$ dans $\overline{x}$.
De cette façon nous avons bien que $(\overline{x}, \overline{y})$ est un alignement de (x, y) avec $|\overline{x}| = |x|+|y| = n+m$ (car nous avons $|x|$ caractères de Σ et $|y|$ gaps $-$ dans $\overline{x}$).

Montrons par l'absurde qu'il ne peut exister de recouvrement de (x, y) tel que $|\overline{x}| > n + m$.
Supposons $|\overline{x}| = n + m + i$ avec $i > 0$. Alors $\overline{x} = \overline{x_1}\ \overline{x_2} \ldots \overline{x_{n+m+i}}$ et comme $|\overline{x}| = |\overline{y}|$ nous avons donc $\overline{y} = \overline{y_1}\ \overline{y_2} \ldots \overline{y_{n+m+i}}$.
On sait que $|x| = n$ donc parmis les $n + m + i$ caractères de $\overline{\Sigma}$ dans $\overline{x}$, n sont des caractères de Σ. Ainsi nous avons $m + i$ gaps $-$.
De ce fait, pour n'importe quelle répartition des gaps dans $\overline{x}$, nous aurons $m+i$ gaps. Or chaque gap de $\overline{x}$ doit se trouver en face d'un caractère de Σ dans $\overline{y}$. Mais il n'y a que m caractères de Σ dans $\overline{y}$.
De ce fait il y aura toujours i gaps de $\overline{x}$ qui se retrouveront en face d'un gap dans $\overline{y}$. Ce qui

contredit le fait que $(\overline{x}, \overline{y})$ soit un recouvrement de (x, y).

Nous pouvons en conclure que la taille maximale d'un recouvrement de (x, y) est de taille $m+n$.

Illustrons ceci par un exemple.
Reprenons l'exemple $\Sigma = \{A, T, G, C\}$, $x = ATTGTA$, $y = ATCTTA$. L'alignement de (x, y) de longueur maximal est donc de la forme

$$\overline{x} = -A - TT - -G - TA-$$
$$\overline{y} = A - T - -CT - T - -A$$

Comme expliqué précédemment, chaque lettre de $\overline{x}$ (resp. $\overline{y}$) se situe en face d'un gap de $\overline{y}$ (resp. $\overline{x}$). Les deux mots sont de taille $n + m$.

2 Algorithmes pour l'alignement de séquences

2.1 Méthode naïve par énumération

Question 3 : Nous avons $|x| = n$ et $|\overline{x}| = n + k$.
On place tout d'abord les k gaps dans $n+k$ espaces (car $|\overline{x}| = n+k$), et ensuite, nous plaçons les lettres comme il faut : comme nous devons respecter l'ordre des lettres de x, il ne reste qu'une seule façon de placer les lettres dans les n espaces qui restent.
Nous avons donc $\binom{n+k}{k} \times 1 = \binom{n+k}{k} = C_{n+k}^{k}$ alignements possibles.

Question 4 : Soit x et y deux mots de taille n et m avec $n \geq m$. Supposons que nous ajoutons k gaps à x pour créer le mot $\overline{x}$ de taille $n + k$.
D'après la question 2, nous savons qu'un alignement de (x, y) et toujours de taille $\leq n + m$. Ainsi, en ajoutant k gaps au mot x, nous obtenons un mot de taille $n + k$. Toutefois ce mot ne peux excéder la taille $n + m$. Nous en déduisons donc que $k \leq m$.
Partie 1
On sait que $|\overline{y}| = |\overline{x}| = n + k$ Donc il faut ajouter à y un total de $(n + k - m)$ gaps pour que $\overline{y}$ ait une longueur $n + k$. En effet $|\overline{y}| = |y| + (n + k - m) = n + k = |\overline{x}|$.
Partie 2
On suppose $(\overline{x}, \overline{y})$ est un alignement de (x, y) ie. $n \geq m \geq k$.
On a $|\overline{x}| = n + k$ avec k gaps. Il y a donc $\binom{n+k}{k}$ arrangements possibles des k gaps parmi les $n + k$ caractères de $\overline{x}$.
De même, $|\overline{y}| = n + k$ avec $n + k - m$ gaps. Il y a donc $\binom{n+k}{n+k-m}$ arrangements possibles des

$n+k-m$ gaps parmi les $n+k$ caractères de $\overline{y}$. Mais chaque gap dans $\overline{y}$ doit se trouver en face d'une lettre de $\overline{x}$.
Ainsi parmi les $n+k$ caractères de $\overline{y}$, k caractères sont obligés d'être des lettres. Il ne reste donc plus que n endroits où mettre les gaps. C'est-à-dire $\binom{n}{n+k-m} = \binom{n}{m-k}$ façons de créer $\overline{y}$.
Partie 3
Le nombre d'alignements possibles, pour k fixé, de (x, y) est le nombre d'arrangements possibles de $\overline{x}$ multiplié par le nombre d'arrangements possibles de $\overline{y}$ sachant l'arrangement de $\overline{x}$ choisit. Ce qui nous donne $\binom{n+k}{k} \times \binom{n}{m-k}$ arrangements.
Donc le nombre de façons possibles de construire $(\overline{x}, \overline{y})$ est $\sum_{k=0}^{m} \binom{n+k}{k} \times \binom{n}{m-k}$.
Pour $|x| = 15$ et $|y| = 10$ nous trouvons $\sum_{k=0}^{10} \binom{15+k}{k} \times \binom{15}{10-k}$ = 298 199 265.

Question 5 : L'algorithme naïf énumère tous les alignements possibles.
Aussi d'après la question 4, nous savons que pour un couple de mots (x, y) de taille n et m, il y a $\sum_{k=0}^{m} \binom{n+k}{k} \times \binom{n}{m-k}$ alignements possibles. Nous devons donc tous les énumérer, ce qui est l'opération la plus coûteuse de l'algorithme en termes de temps.
Il est possible de majorer grossièrement le nombre d'alignements total de la façon suivante :

$$\begin{aligned}
\sum_{k=0}^{m} \binom{n+k}{k} \times \binom{n}{m-k} &= \sum_{k=0}^{m} \frac{(n+k)!}{k!n!} \times \frac{n!}{(m-k)!} \\
&= \sum_{k=0}^{m} \frac{(n+k)!}{k!(m-k)!} \\
&\leq \sum_{k=0}^{m} (n+k)! \\
&\leq \sum_{k=0}^{m} (n+m)! \\
&\approx O(m \times (n+m)!)
\end{aligned} \tag{1}$$

On obtient donc de l'ordre de $O(m \times (n+m)!)$ alignements. Nous pouvons donc en conclure une complexité temporelle factorielle.
Chercher un alignement de coût minimal revient à calculer la distance d'édition et inversement. Ainsi, la complexité temporelle pour calculer la distance d'édition entre deux mots ne varie pas de celle pour trouver un alignement de coût minimal.

Question 6 : Soient $x, y \in \Sigma$ tels que $|x| = n$ et $|y| = m$.
Un raisonnement naïf consisterait à énumérer tous les alignements possibles de (x, y) et à les comparer à l'alignement minimal trouvé jusqu'à présent.

Comme pour la question précédente, l'objectif est de trouver la distance d'édition ou l'alignement de coût minimal.
La complexité spatiale va dépendre de la construction de l'algorithme, mais en général, il faudra garder en mémoire à tout instant :

1. L'alignement de coût minimal trouvé jusqu'à présent dans le cas de la recherche d'alignement ($O(n+m)$) ou simplement la distance d'édition dans le second cas ($O(1)$).
2. L'alignement qu'on est en train d'examiner pour voir s'il est de coût inférieur à celui trouvé jusqu'à présent ($O(n+m)$).

Ainsi, la complexité spatiale de l'algorithme naïf est en $O(n+m)$ ce qui est une complexité linéaire.

Tâche A :

- La première instance s'exécutant en plus de 60 secondes est "Inst_0000012_56.adn".

```
Inst_0000010_7 0.0363619327545166
Inst_0000010_8 4.815987825393677
Inst_0000010_44 1.913606882095337
Inst_0000012_13 8.716949939727783
Inst_0000012_32 8.83556079864502
Inst_0000012_56 66.54521012306213
```

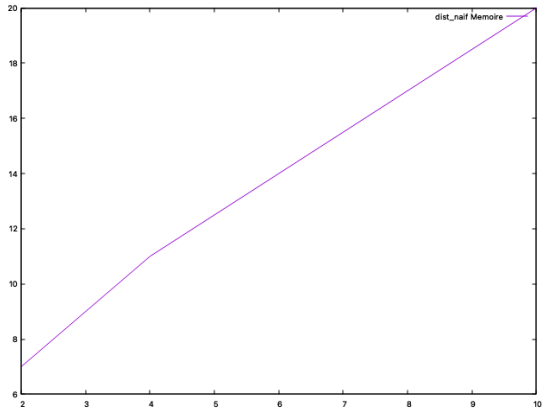
- Consommation mémoire : Cet algorithme est de complexité spatiale en $O(n+m)$. Ceci est dû au fait que nous ne considérons qu'un unique alignement à la fois. En effet lors de l'appel récursif nous entrons dans l'un des trois cas possibles, jusqu'à arriver au cas de base, où $i = |x|$ et $j = |y|$, ce qui correspond à un alignement.

  Ainsi les appels récursifs s'empilent un à un, en incrémentant i et/ou j de 1 à chaque appel, jusqu'à atteindre le cas de base. C'est à ce moment que l'appel se termine et que nous entrons dans un nouvel appel c'est-à-dire que nous passons d'un alignement au prochain.

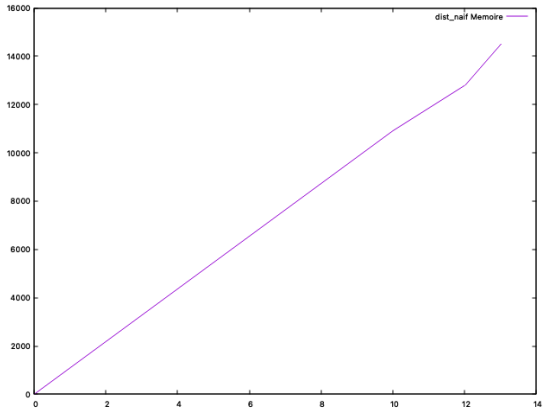
  En suivant ce raisonnement, le pire cas serait d'incrémenter i ou j à chaque appel, et ainsi d'empiler au plus $n+m$ appels dans la pile.
  L'étape suivante dépend du langage de programmation utilisé pour l'implémentation. Dans notre cas, en python, le passage des paramètres se fait par référence/ment ?, et donc le coût spatial d'un appel récursif est en $O(1)$.

  Comme nous avons au plus $n+m$ appels récursifs sur la pile, la complexité spatiale de l'algorithme est en $O(n+m)$.

Pour bien voir cela, le premier graphique ci-dessous nous permet de voir l'évolution linéaire de la hauteur de l'arbre des appels récursifs, en fonction de la taille de nos instances. Le deuxième nous montre que la complexité spatiale est linéaire.



(a) *Hauteur de l'arbre des appels récursifs*



(b) *Complexité spatiale de dist_naif*

Toutefois, les résultats expérimentaux peuvent être insuffisants car nous ne testons que des instances de longueur 12 au maximum, à cause de la complexité temporelle élevée.

2.2 Programmation dynamique

2.2.1 Calcul de la distance d'édition par programmation dynamique

Question 7 : Soit $(\overline{u}, \overline{v})$ un alignement de $(x_{[1..i]}, y_{[1..j]})$ de longueur l.
Nous avons $\overline{u}_l = -$, c'est-à-dire que le dernier caractère de $\overline{u}$ est un gap. Par propriétés de l'alignement, cela implique que $\overline{v}_l \neq -$. De ce fait, $\overline{v}_l$ est le dernier caractère de $\overline{v}$ qui n'est pas un gap ie. c'est le dernier caractère de $y_{[1..j]}$ ie. c'est y_j.
Un raisonnement similaire permet de conclure que, si $\overline{v}_l = -$, alors $\overline{u}_l \neq -$, alors $\overline{u}_l = x_i$.
De même, si $\overline{u}_l \neq -$ et $\overline{v}_l \neq -$ alors les deux caractères de fin seront les derniers caractères de $x_{[1..i]}$ et $y_{[1..j]}$. Ainsi $\overline{u}_l = x_i$ et $\overline{v}_l = y_j$.

Question 8 : Par définition et en utilisant la question précédente nous obtenons

$$C(\overline{u}, \overline{v}) = C(\overline{u}_{[1..l-1]}, \overline{v}_{[1..l-1]}) + c(\overline{u}_l, \overline{v}_l) \text{ avec } c(\overline{u}_l, \overline{v}_l) = \begin{cases} c_{\text{ins}} & \text{si } \overline{u}_l = - \\ c_{\text{del}} & \text{si } \overline{v}_l = - \\ c_{\text{sub}}(x_i, y_j) & \text{si } \overline{u}_l \neq - \text{ et } \overline{v}_l \neq - \end{cases}$$

Question 9 : Soit $i \in [1..n]$ et $j \in [1..m]$.
$D(i,j) = d(x_{[1..i]}, y_{[1..j]}) = min\{C(\overline{u},\overline{v}) \mid (\overline{u},\overline{v}) \text{ alignement de } (x_{[1..i]}, y_{[1..j]})\}$.
De plus nous savons que $C(\overline{u},\overline{v}) = C(\overline{u}_{[1..l-1]}, \overline{v}_{[1..l-1]}) + c(\overline{u}_l, \overline{v}_l)$ (cf. question 8).

Trois cas sont possibles :
$$\begin{cases} c(\overline{u}_l, \overline{v}_l) = c_{ins} \text{ ie } \overline{u}_l = - \text{ et } \overline{v}_l = y_j \\ c(\overline{u}_l, \overline{v}_l) = c_{del} \text{ ie } \overline{u}_l = x_i \text{ et } \overline{v}_l = - \\ c(\overline{u}_l, \overline{v}_l) = c_{sub} \text{ ie } \overline{u}_l = x_i \text{ et } \overline{v}_l = y_j \end{cases}$$

Ainsi nous en déduisons
$$\begin{cases} C(\overline{u}_{[1..l-1]}, \overline{v}_{[1..l-1]}) = d(x_{[1..i]}, y_{[1..j-1]}) \\ C(\overline{u}_{[1..l-1]}, \overline{v}_{[1..l-1]}) = d(x_{[1..i-1]}, y_{[1..j]}) \\ C(\overline{u}_{[1..l-1]}, \overline{v}_{[1..l-1]}) = d(x_{[1..i-1]}, y_{[1..j-1]}) \end{cases}$$

Alors
$$\begin{cases} D(i,j) = D(i,j-1) + c_{ins} & \text{si } c(\overline{u}_l, \overline{v}_l) = c_{ins} \\ D(i,j) = D(i-1,j) + c_{del} & \text{si } c(\overline{u}_l, \overline{v}_l) = c_{del} \\ D(i,j) = D(i-1,j-1) + c_{sub}(x_i, y_j) & \text{si } c(\overline{u}_l, \overline{v}_l) = c_{sub} \end{cases}$$

Aussi, nous savons par définition que $D(i,j) = min\{C(\overline{u},\overline{v}) \mid (\overline{u},\overline{v}) \text{ alignement de } (x_{[1..i]}, y_{[1..j]})\}$.
Et nous avons montré, dans l'étape précédente, qu'il n'y a que trois façons de construire l'alignement de $(x_{[1..i]}, y_{[1..j]})$, d'où :
$D(i,j) = min\{D(i,j-1) + c_{ins}, D(i-1,j) + c_{del}, D(i-1,j-1) + c_{sub}(x_i, y_j)\}$.

Question 10 : Par définition : $D(0,0) = d(x_{[1..0]}, y_{[1..0]}) = d(\epsilon,\epsilon) = 0$ car les deux mots sont identiques.

Question 11 : Soit $j \in [1..m]$.
On peut procéder par récurrence sur j, en s'aidant par les questions 9 et 10.
Soit $P(j) : D(0,j) = \sum_{k=1}^{j} c_{ins} = j \times c_{ins}$.
Pour $j = 0$, nous avons $D(0,0) = 0 \times c_{ins} = 0$, ce qui est vrai par la question 10.
Soit $P(j-1) : D(0,j-1) = (j-1) \times c_{ins}$ vraie. Montrons $P(j)$.
Selon la question 9, comme $i = 0$, nous ne pouvons pas avoir $i - 1$ et il ne nous reste qu'une seule option des trois pour construire $D(i,j) : D(i,j) = D(i,j-1) + c_{ins}$.
Par hypothèse de récurrence, $D(i,j) = (j-1) \times c_{ins} \times c_{ins} = j \times c_{ins}$.
On a donc bien montré que $D(0,j) = j \times c_{ins}$.

Autre justification :
On a $D(0,j) = d(x_{[1..0]}, y_{[1..j]}) = d(\epsilon, y_{[1..j]}) = C(\overline{x},\overline{y})$ avec

$$\overline{x} = ---...-$$
$$\overline{y} = y_1 y_2 y_3 ... y_j$$

C'est d'ailleurs le seul alignement possible, puisqu'on est contraint par les inégalités $n+k \geq m$, et $k \leq m$ (cf. question 4) (ici m est la longueur du mot $y_{[1..j]}$). Comme ici $n = 0$, nous avons donc forcément $k = m$. Ainsi pour avoir un alignement de $(\epsilon, y_{[1..j]})$ il faut insérer dans $\overline{x}_{[1..0]}$ chaque caractère de $y_{[1..j]}$, représenté chacun par un gap.
C'est-à-dire $D(0, j) = \sum_{k=1}^{j} c_{ins} = j \times c_{ins}$.

Question 12 :

Algorithm 1 : DIST_1

Entrées : x et y deux mots
Sortie : $d(x, y)$ la distance d'édition entre x et y
Algorithme :
mat : matrice de taille $|x| \times |y|$
$mat_{0,0} \leftarrow 0$
for i allant de 1 à $|x|$ inclus **do**
    $mat_{i,0} \leftarrow i \times c_{del}$
end for
for j allant de 1 à $|y|$ inclus **do**
    $mat_{0,j} \leftarrow j \times c_{ins}$
end for
for i allant de 1 à $|x|$ inclus **do**
    for j allant de 1 à $|y|$ inclus **do**

$$mat_{i,j} \leftarrow \text{minimum} \begin{cases} mat_{i,j-1} + c_{ins} \\ mat_{i-1,j} + c_{del} \\ mat_{i-1,j-1} + c_{sub}(x_i, y_j) \end{cases}$$

    end for
end for
return $mat_{|x|,|y|}$

Question 13 : Pour exécuter DIST_1, nous avons besoin d'une matrice d'entiers de taille $n \times m$ pendant l'exécution de l'algorithme. La complexité spatiale est donc en $O(n \times m)$.

Question 14 : Les deux premières boucles s'exécutent respectivement n et m fois avec un corps de boucle en $O(1)$. La complexité temporelle des deux boucles est donc en $O(n + m)$.

La troisième boucle est une double boucle allant de 1 à n et de 1 à m ne contenant que des opérations élémentaires.
La complexité temporelle de DIST_1 est donc en $O(n + m + n \times m) = O(n \times m)$.

2.2.2 Calcul d'un alignement optimal par programmation dynamique

Question 15 : Soit $j > 0$ et $D(i,j) = D(i,j-1) + c_{ins}$ et soit $(\overline{s}, \overline{t}) \in Al^*(i, j-1)$.
Ainsi nous avons $C(\overline{s}, \overline{t}) = d(x_{[1..i]}, y_{[1..j-1]})$.
On pose $\overline{s}' = \overline{s}.-$ et $\overline{t}' = \overline{t}.y_j$ montrons que $(\overline{s}', \overline{t}') = d(x_{[1..i]}, y_{[1..j]})$.
D'après la question 8, nous savons que

$$\begin{aligned} C(\overline{s}', \overline{t}') &= C(\overline{s}'_{[1..l-1]}, \overline{t}'_{[1..l-1]}) + c(\overline{s}'_l, \overline{t}'_l) \\ &= C(\overline{s}, \overline{t}) + c(-, y_j) = (x_{[1..i]}, y_{[1..j-1]}) + c_{ins} \\ &= D(i, j-1) + c_{ins} \\ &= D(i,j) \text{ par hypothèse.} \end{aligned} \tag{2}$$

Alors $C(\overline{s}', \overline{t}') = d(x_{[1..i]}, y_{[1..j]})$, et donc $(\overline{s}.-, \overline{t}.y_j) \in Al^*(i,j)$.

De même, en supposant $i > 0$ et $D(i,j) = D(i-1,j) + c_{del}$ et soit $(\overline{s}, \overline{t}) \in Al^*(i-1, j)$.
Ainsi nous avons $C(\overline{s}, \overline{t}) = d(x_{[1..i-1]}, y_{[1..j]})$.
On pose $\overline{s}' = \overline{s}.x_i$ et $\overline{t}' = \overline{t}.-$ montrons que $(\overline{s}', \overline{t}') = d(x_{[1..i]}, y_{[1..j]})$.
D'après la question 8, nous savons que

$$\begin{aligned} C(\overline{s}', \overline{t}') &= C(\overline{s}'_{[1..l-1]}, \overline{t}'_{[1..l-1]}) + c(\overline{s}'_l, \overline{t}'_l) \\ &= C(\overline{s}, \overline{t}) + c(x_i, -) \\ &= (x_{[1..i-1]}, y_{[1..j]}) + c_{del} \\ &= D(i-1, j) + c_{del} = D(i,j) \text{ par hypothèse.} \end{aligned} \tag{3}$$

Alors $C(\overline{s}', \overline{t}') = d(x_{[1..i]}, y_{[1..j]})$, et donc $(\overline{s}.-, \overline{t}.y_j) \in Al^*(i,j)$.

Enfin, supposons $D(i,j) = D(i-1,j-1) + c_{sub}(x_i, y_j)$ et soit $(\overline{s}, \overline{t}) \in Al^*(i-1, j-1)$.
Ainsi nous avons $C(\overline{s}, \overline{t}) = d(x_{[1..i-1]}, y_{[1..j-1]})$.
On pose $\overline{s}' = \overline{s}.x_i$ et $\overline{t}' = \overline{t}.y_j$ montrons que $(\overline{s}', \overline{t}') = d(x_{[1..i]}, y_{[1..j]})$.
D'après la question 8, nous savons que

$$\begin{aligned} C(\overline{s}', \overline{t}') &= C(\overline{s}'_{[1..l-1]}, \overline{t}'_{[1..l-1]}) + c(\overline{s}'_l, \overline{t}'_l) \\ &= C(\overline{s}, \overline{t}) + c(x_i, y_j) \\ &= (x_{[1..i-1]}, y_{[1..j-1]}) + c_{sub}(x_i, y_j) \\ &= D(i-1, j-1) + c_{sub}(x_i, y_j) \\ &= D(i,j) \text{ par hypothèse;} \end{aligned} \tag{4}$$

Alors $C(\overline{s}', \overline{t}') = d(x_{[1..i]}, y_{[1..j]})$, et donc $(\overline{s}.x_i, \overline{t}.y_j) \in Al^*(i,j)$.

Question 16 :

Algorithm 2 : SOL_1

Entrées : x et y deux mots et une matrice T de taille $|x| \times |y|$ contenant les valeurs de D
Sortie : alignement minimal de (x, y)
Algorithme :
$(s, t) \leftarrow (\text{""}, \text{""})$
$i \leftarrow |x|$
$j \leftarrow |y|$
while $T_{i,j} > 0$ **do**
  if $j > 0$ et $T_{i,j} = T_{i,j-1} + c_{ins}$ **then**
    $(s, t) \leftarrow (\text{"}-\text{"} + s, y_j + t)$
    $j \leftarrow j - 1$
  else if $i > 0$ et $T_{i,j} = T_{i-1,j} + c_{del}$ **then**
    $(s, t) \leftarrow (x_i + s, \text{"}-\text{"} + t)$
    $i \leftarrow i - 1$
  else if $T_{i,j} = T_{i-1,j-1} + c_{sub}(x_i, y_j)$ **then**
    $(s, t) \leftarrow (x_i + s, y_j + t)$
    $j \leftarrow j - 1$
    $i \leftarrow i - 1$
  end if
end while
while $i > 0$ **do**
  $(s, t) \leftarrow (x_i + s, y_i + t)$
  $i \leftarrow i - 1$
end while
return (s, t)

Question 17 : Soit $|x| = n$ et $|y| = m$.

Complexité de DIST_1 :
DIST_1 est composée de 3 boucles.
Boucle 1 : n tours de boucle, chacun contenant une opération élémentaire. La complexité de l'exécution de cette boucle est donc en $O(n)$.
Boucle 2 : m tours de boucle, chacun contenant une opération élémentaire. La complexité de l'exécution de cette boucle est donc en $O(m)$.
Boucle 3 : n tours de boucle. Cette boucle contient une boucle imbriquée qui s'exécute m fois et dont l'unique opération s'effectue en $O(1)$. Cela nous donc une complexité en $O(m)$ pour la boucle imbriquée et une complexité en $O(n \times m)$ pour la boucle 3.

Complexité de SOL_1 :
Par le même procédé de raisonnement, SOL_1 est constitué de 2 boucles.
Boucle 1 : au plus $n + m$ tours de boucle, car i et/ou j décroît de 1 à chaque tour. Aussi le corps de la boucle s'effectue en $O(1)$ donc le coût de la boucle est en $O(n + m)$.
Boucle 2 : au plus n tours de boucle si nous supposons que la boucle 1 n'a fait que décroître j. De plus le corps de la boucle s'effectue en $O(1)$ donc le coût de la boucle est en $O(n)$.
La complexité temporelle de SOL_1 est donc en $O(n + m + n) = O(n + m)$.

Complexité de PROG_DYN :
On reprend le raisonnement de l'algorithme DIST_1 pour cacluler la matrice des distances d'éditions, en $O(n \times m)$. Une fois la matrice créée il suffit d'appliquer l'algorithme SOL_1 pour obtenir l'alignement minimal, en $O(n + m)$. Enfin il faut renvoyer le chemin obtenu ce qui se fait en $0(1)$.
La complexité temporelle de l'algorithme PROG_DYN qui résout le problème ALI est donc en $O(n \times m + (n + m) + 1) = O(n \times m)$.

Question 18 : Soit $|x| = n$ et $|y| = m$.

Complexité spatiale de DIST_1 :
Pendant l'exécution de DIST_1, à tout moment nous gardons la matrice de taille $n \times m$ en mémoire, plus quelques opérations élémentaires sur les entiers nécessitant une complexité spatiale en $O(1)$. De plus nous gardons en mémoire les deux mots x et y, ce qui nécessite une complexité spatiale de $O(n + m)$.
DIST_1 a donc une complexité spatiale en $O(n \times m + (n + m) + 1) = O(n \times m)$.

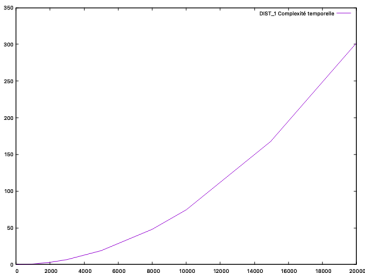
Complexité spatiale de SOL_1 :
Tout d'abord, nous gardons les mots x et y en mémoire et la matrice, ce qui nécessite une complexité de $O(n \times m + (n + m)) = O(n \times m)$.
Ensuite, nous construisons l'alignement (s, t) de (x, y), ce qui nécessite une complexité spatiale

d'au plus $O(2 \times (n+m)) = O(n+m)$.
Enfin, nous utilisons quelques manipulations d'entiers et de caractères en $O(1)$.
La complexité spatiale de SOL_1 est donc de l'ordre de $O(n \times m + (n+m) + 1) = O(n \times m)$.

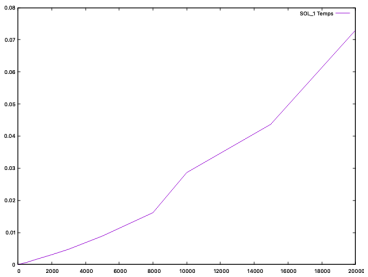
Complexité spatiale de PROG_DYN :
Tout d'abord nous exécutons DIST_1 pour calculer la matrice, ce qui nécessite une complexité spatiale en $O(n \times m)$.
Ensuite nous utilisons cette matrice pour exécuter SOL_1, ce qui nécessite aussi une complexité spatiale en $O(n \times m)$.
Enfin nous retournons la distance d'édition calculée par DIST_1 et l'alignement calculé par SOL_1, ce qui a une complexité spatiale de $O(1 + 2 \times (n+m)) = O(n+m)$.
La complexité spatiale de l'algorithme PROG_DYN qui résout le problème ALI est donc en $O(n \times m)$.

Tâche B :

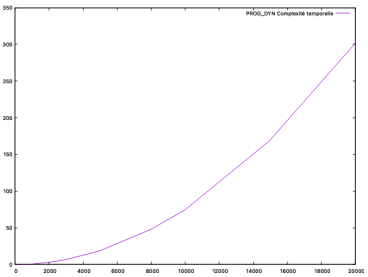
- Courbes de consommation de temps :



(a) *DIST_1*



(b) *SOL_1*



(c) *PROG_DYN*

Ces courbes correspondent bien aux complexités théoriques trouvées à la question 17. DIST_1 et PROG_DYN sont quadratiques, alors que SOL_1 est presque linéaire.
Les instances qui prennent plus que 10 minutes sont celles de longueurs supérieures à 20000.

- Courbe de consommation mémoire de PROG_DYN :

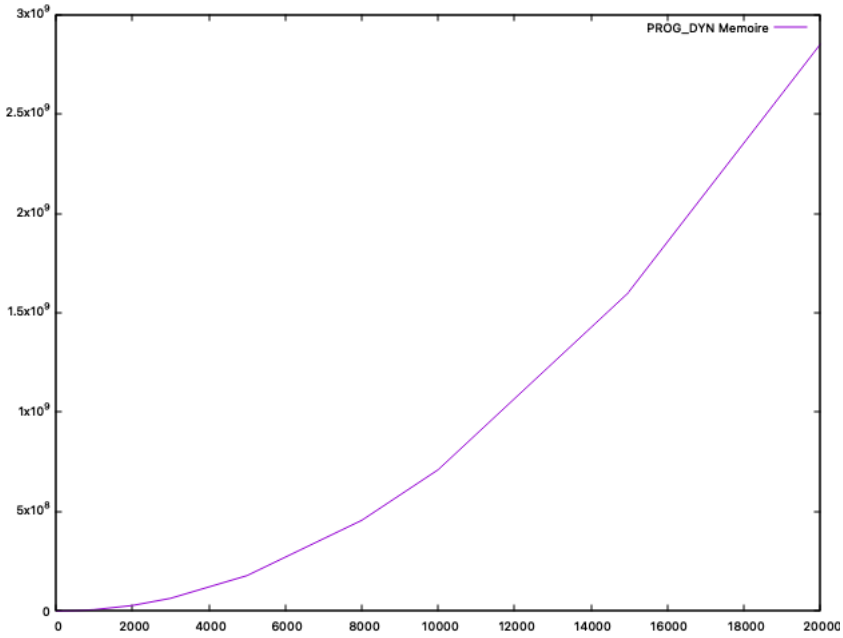


Figure 3 – *Sur l'axe des ordonnées : la mémoire prise, en octets.*

2.3 Amélioration de la complexité spatiale du calcul de la distance

Question 19 : Pour remplir la case $D_{i,j}$, seules les trois cases de la matrice D : $D_{i,j-1}$, $D_{i-1,j}$ et $D_{i-1,j-1}$ sont nécessaires d'après la question 9.
L'accès à la lignes $i-1$ nous permet de trouver $D_{i-1,j-1}$ et $D_{i-1,j}$, quand à la ligne i, celle-ci nous donne la case $D_{i,j-1}$.
Nous pouvons donc calculer facilement $D_{i,j}$.

Question 20 :

Algorithm 3 : DIST_2

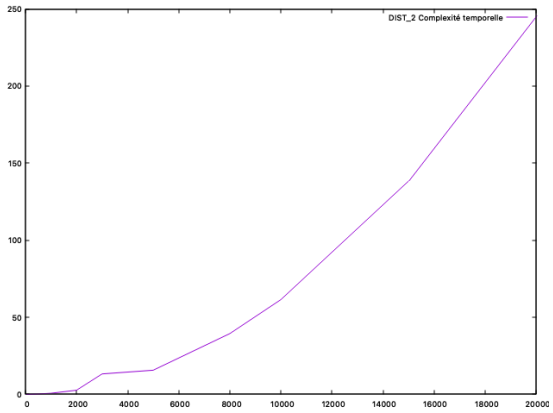
Entrées : x et y deux mots
Sortie : $d(x, y)$ la distance d'édition entre x et y
Algorithme :
$l1$: liste de taille $|y|$.
$l2$: liste de taille $|y|$.
for j allant de 0 à $|x|$ inclus **do**
    $l1[j] \leftarrow j \times c_{ins}$
end for
for i allant de 1 à $|x|$ inclus **do**
    $l2[0] = i \times c_{del}$
    for j allant de 1 à $|y|$ inclus **do**

$$l2[j] \leftarrow \text{minimum} \begin{cases} l2[j-1] + c_{ins} \\ l1[j] + c_{del} \\ l1[j-1] + c_{sub}(x_i, y_j) \end{cases}$$

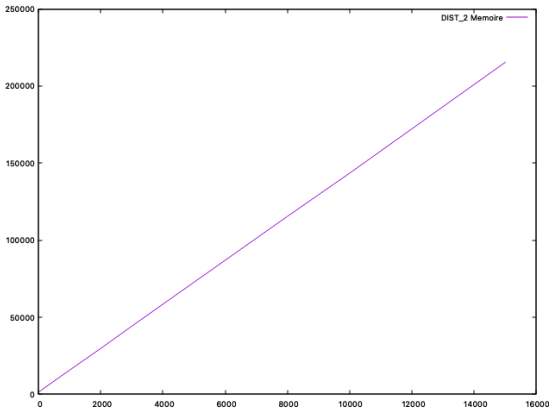
    end for
    $l1 \leftarrow l2$
end for
return $l2[|y|]$

Tâche C :

- Complexité temporelle expérimentale de DIST_2 :
  La complexité temporelle de DIST_2 est censée être la même que pour DIST_1. En effet, les deux courbes obtenues expérimentalement ont la même allure, ce qui correspond à la complexité théorique de DIST_2.
- Complexité spatiale expérimentale de DIST_2 :
  Le résultat obtenu est linéaire, ainsi pour une grande instance d'ordre $O(n)$, l'espace mémoire occupé sera en $O(n)$.



(a) *Complexité temporelle de DIST_2*



(b) *Complexité spatiale de DIST_2*

2.4 Amélioration de la complexité spatiale du calcul d'un alignement optimal par la méthode "diviser pour régner"

Question 21 :

Algorithm 4 : mots_gaps

Entrées : k un entier
Sortie : le mot constitué de k gaps.
return $k \times$ " $-$ "

Question 22 : On défini $(x)^c$ comme le symbole complémentaire de la paire concordante de x.

Algorithm 5 : align_lettre_mot

Entrées : x mot de longueur 1 et y mot non vide longueur quelconque.
Sortie : alignement optimal de (x, y).
Algorithme :
$ali\_x \leftarrow$ mot_gaps($|y|$)
$tmp \leftarrow +\infty$
for i in range $|y|$ **do**
  if $y[i] = x$ **then**
    $ali\_x[i] \leftarrow x$
    return ali_x, y
  end if
  if $y[i] = (x)^c$ **then**
    $tmp \leftarrow i$
  end if
end for
if $tmp = +\infty$ **then**
  return $x \,.\, ali\_x, \text{ " } - \text{ " } \,.\, y$
end if
$ali\_x[tmp] \leftarrow x$
return aly_x, y

Question 23 : Pour trouver un alignement optimal de (x^1, y^1) et (x^2, y^2) nous avons souhaité utiliser notre fonction $PROG\_DYN()$. Pour se faire nous avons du modifier $c_{sub}(a, b)$, c_{del} et c_{ins} afin que les coûts correspondent à l'énoncé.

L'alignement optimal $(\overline{s}, \overline{t})$ de (x^1, y^1) est $(\text{"}BAL\text{"}, \text{"}RO-\text{"})$.
L'alignement optimal $(\overline{u}, \overline{v})$ de (x^2, y^2) est $(\text{"}LON-\text{"}, \text{"}--ND\text{"})$.
La concaténation donne $(\overline{s}.\overline{u}, \overline{t}.\overline{v})$ = $(\text{"}BALLON-\text{"}, \text{"}RO---ND\text{"})$.

Enfin, nous avons utilisé notre fonction pour trouver l'alignement optimal de (x, y), et nous avons obtenu $(\text{"}BALLON-\text{"}, \text{"}R---OND\text{"})$. Il est clair que ce n'est pas le même alignement.

Nous pouvons aussi vérifier que le coût de $(\text{"}BALLON-\text{"}, \text{"}R---OND\text{"}) = 17$, alors que le coût de $(\text{"}BALLON-\text{"}, \text{"}RO---ND\text{"}) = 22$.

Question 24 :

Algorithm 6 : SOL_2

Entrées : x et y deux mots.
Sortie : alignement optimal de (x, y).
Algorithme :
if $|y| = 0$ **then**
    return (x, mot_gaps($|x|$))
end if
if $|x| = 0$ **then**
    return (mot_gaps($|y|$, y))
end if
if $|x| = 1$ **then**
    return align_lettre_mot(x, y)
end if
$i_1 = \lfloor \frac{|x|}{2} \rfloor$ et $i_2 = \text{coupure}(x, y)$
$(s_1, t_1) \leftarrow \text{SOL\_2}(x_{[0..i_1]}, y_{[0..i_2]})$
$(s_2, t_2) \leftarrow \text{SOL\_2}(x_{[i_1+1..|x|]}, y_{[i_2+1..|y|]})$
return $(s_1.s_2, t_1.t_2)$

Question 25 :

Algorithm 7 : coupure

Entrées : x et y deux mots.
Sortie : La coupure j^* associé à l'indice $\lfloor \frac{|x|}{2} \rfloor$ de (x, y).
Algorithme :
$l1$, $l2$, $p1$ et $p2$: listes de taille $|y|$.
for j allant de 0 à $|x|$ inclus **do**
$l1[j] \leftarrow j \times c_{ins}$
end for
for i allant de 1 à $\lfloor \frac{|x|}{2} \rfloor$ inclus **do**
$l2[0] = i \times c_{del}$
for j allant de 1 à $|y|$ inclus **do**

$$l2[j] \leftarrow \text{minimum} \begin{cases} l2[j-1] + c_{ins} \\ l1[j] + c_{del} \\ l1[j-1] + c_{sub}(x_i, y_j) \end{cases}$$

end for
$l1 \leftarrow l2$
end for
for i allant de $\lfloor \frac{|x|}{2} \rfloor + 1$ à $|x|$ inclus **do**
$l2[0] = i \times c_{del}$
$p2[0] = p1[0]$
for j allant de 1 à $|y|$ inclus **do**

$$l2[j] \leftarrow \text{minimum} \begin{cases} l2[j-1] + c_{ins} \\ l1[j] + c_{del} \\ l1[j-1] + c_{sub}(x_i, y_j) \end{cases}$$

$$p2[j] \leftarrow \begin{cases} p2[j-1] & \text{si } l2[j] = l2[j-1] + c_{ins} \\ p1[j] & \text{si } l2[j] = l1[j] + c_{del} \\ p1[j-1] & \text{si } l2[j] = l1[j-1] + c_{sub}(x_i, y_j) \end{cases}$$

end for
$p1 \leftarrow p2$
$l1 \leftarrow l2$
end for
return $p2[|y|]$

Question 26 : Lors de l'appel de $coupure(x, y)$ nous devons garder en mémoire les mots x et y ce qui représente un coût en $O(n + m)$.
Aussi, durant l'exécution de la fonction, il est nécessaire de garder en mémoire les 4 tableaux $l1$, $l2$, $p1$ et $p2$ de taille m contenant des entiers, ce qui engendre pour chaque tableau un coût mémoire en $O(m \times 1) = O(m)$. Tout au long de l'algorithme, nous ne faisons que modifier les valeurs stockées dans ces listes, toujours par des entiers en $O(1)$.
Ainsi la complexité spatiale de l'algorithme est en $O((n + m) + 4 \times m) = O(n + m)$.

Question 27 : Soit $|x| = n$ et $|y| = m$.
Complexité spatiale de SOL_2 :

- cas $|y| = 0$ en $O(n)$
- cas $|x| = 0$ en $O(m)$
- cas $|x| = 1$ en $O(2m) = O(m)$
- cas $|x| \geq 2$ et $|y| \geq 1$

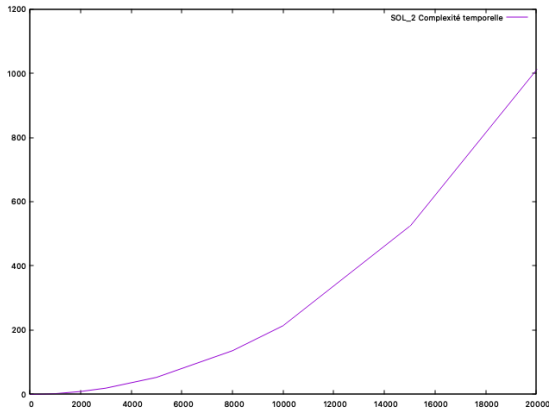
Coupure est de complexité spatiale $O(n + m)$.

$$\begin{aligned} A(n) &= \text{complexité appel sur}(x^1, y^1) + \text{complexité appel sur}(x^2, y^2) + \text{complexité } coupure \\ &= A(n/2) + A(m) + O(n) \\ &\leq 2 \times A(n) + O(n) \end{aligned} \tag{5}$$

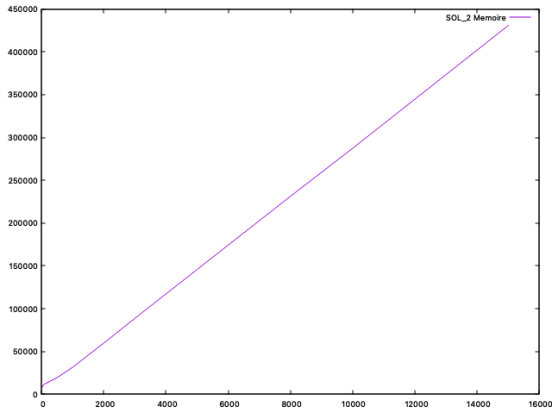
Où la dernière égalité vient du fait que x et coupé en deux parties égales mais pas y que l'on majore par m et coupure majoré par $O(n)$. On trouve $a = 2$, $b = 1$ et $d = 1$, ainsi on a $a = 2 \geq 1 = b^d$.
Donc une complexité spatiale en $O(n)$ Aussi les deux appels récursifs sur (x^1, y^1) et (x^2, y^2)

Question 28 : L'algorithme $coupure()$ est composé de 3 boucles ainsi que d'opérations élémentaires s'effectuant en $O(1)$.
Boucle 1 : n tours de boucle et le corps de boucle s'effectue en $O(1)$. La complexité temporelle de cette boucle est donc en $O(n)$.
Boucle 2 : $\lfloor \frac{n}{2} \rfloor$ tours de boucle. Cette boucle contient une boucle imbriquée qui s'exécute m fois et dont les opérations s'effectuent en $O(1)$. Cela nous donc une complexité en $O(m)$ pour la boucle imbriquée et une complexité en $O(n \times m)$ pour la boucle 2.
Boucle 3 : $\lfloor \frac{n}{2} \rfloor$ tours de boucle. Cette boucle contient une boucle imbriquée qui s'exécute m fois et dont les opérations s'effectuent en $O(1)$. Cela nous donc une complexité en $O(m)$ pour la boucle imbriquée et une complexité en $O(n \times m)$ pour la boucle 3.
Ainsi la complexité temporelle totale de $coupure()$ est en $O(n+(n\times m)+(n\times m)) = O(n\times m)$.

Tâche D :



(a) *Complexité temporelle de SOL_2*



(b) *Complexité spatiale de SOL_2*

SOL_2 a une complexité temporelle quadratique, et une complexité spatiale linéaire. Pour une instance de taille 15000, on est sur 430Mo d'espace mémoire.

Question 29 : Nous comparons SOL_2 avec PROG_DYN au lieu de SOL_1 car SOL_2 calcul la matrice de distance tout comme PROG_DYN, contrairement à SOL_1 qui prend la matrice en paramètre.
La complexité expérimentale de SOL_2 est quadratique, tout comme PROG_DYN, à un facteur de 3 près, comme le montre les données ci-dessous.
La première colonne représente la taille des instances, la deuxième le temps pris pour les calculer, en secondes.

```
10 9.449323018391927e-05
12 0.0001900196075439453
13 0.00016736984252929688
14 0.00017261505126953125
15 0.00020734469095865885
20 0.0003388722737630208
50 0.0027222633361816406
100 0.0088653564453125
500 0.2519436677296956
1000 0.8052153587341309
2000 3.0269216696421304
3000 6.817572295665741
5000 18.858280181884766
8000 48.10878396034241
10000 74.87125595410664
15000 169.71952136357626
20000 301.56670602162677
```

(a) *PROG_DYN*

```
10 0.00027108192443847656
12 0.00037900606791178387
13 0.0005354086558024088
14 0.0007491906483968099
15 0.000587622324625651
20 0.0022521018981933594
50 0.007923762003580729
100 0.0352013905843099
500 0.6017513275146484
1000 2.183717966079712
2000 8.96295158068339
3000 19.20134437084198
5000 52.884756565093994
8000 135.91277726491293
10000 213.70985134442648
15000 523.6690533955892
20000 1010.8613258997599
```

(b) *SOL_2*

A Commentaires sur le code

A.1 Précisions sur l'implémentation

Nous avons décidé d'implémenter notre code en Python. Au début, nous avons implémenté nos fonctions en utilisant les instructions basiques de programmation en Python. Évidemment, nous avons rapidement été confrontés à des problèmes de mémoire, principalement liés à la création de nos matrices, qui nous empêchaient même de tester nos fonctions sur les instances de qui prennent plus de 10 minutes pour s'exécuter, à cause de :

1. Un entier en python prend 28 bytes, ce qui est énorme est beaucoup plus de ce qu'on aurait besoin au cours du projet.
2. L'idée derrière nos fonctions en termes de mémoire était d'allouer la matrice au début et ne faire que changer les valeurs à l'intérieur. Ceci marche très bien avec une implémentation en C par exemple, mais comme python est non typé, nous avons trouvé que ces calculs d'entiers nous coûtaient beaucoup de mémoire, alors que ce n'est pas du tout nécessaire pour notre implémentation.

On a donc opté pour une implémentation en numpy, qui n'est pas seulement beaucoup plus efficace, mais c'est aussi le standard pour des calculs de matrices et de calculs scientifiques aussi grands en général.
Par exemple, pour le calcul de la matrice de DIST_1 pour une instance de 20000, on a passé de 12 Go de RAM nécessaires au calcul sous l'ancienne implémentation $(28 \text{ bytes} \times 20000 \times 20000)$, à 2.5 Go sous l'implémentation numpy, ce qui nous a permis aussi de tester nos fonctions au-delà des instances qui prennent 10 minutes à s'exécuter.

A.2 Organisation du code

Vous trouverez dans le répertoire les fonctions de chaque tâche dans un fichier python séparé. Nous avons aussi fourni un fichier python contenant les jeux de tests que nous avons utilisé pour tester nos fonctions.