Rapport du mini-projet

Ryan Lahfa

Table des matières

Table des figures

Question 1

Soient $(\overline{x}, \overline{y}), (\overline{u}, \overline{v})$ des alignements de $(x, y), (u, v)$ respectivement.

Alors, par propriété d'alignement, on a : $|\overline{xu}| = |\overline{x}| + |\overline{u}| = |\overline{y}| + |\overline{v}| = |\overline{yv}|$, d'où (iii).

Puis, il est immédiat que π est un morphisme de mots, d'où, par morphisme, on en tire la propriété (i) et (ii) par les hypothèses d'alignement.

Enfin, soit $i \in [[1, |\overline{xu}|]]$, si $i \leq |\overline{x}|$, alors par (iv) sur $(\overline{x}, \overline{y})$, on a : $(\overline{xu})_i = \overline{x}_i \neq -$ ou $(\overline{yv})_i = \overline{y}_i \neq -$.

Sinon si $i \in [[|\overline{x}| + 1, |\overline{xu}|]]$, alors par (iv) sur $(\overline{u}, \overline{v})$, $(\overline{xu})_i = \overline{u}_i \neq -$ ou $(\overline{yv})_i = \overline{v}_i \neq -$.

D'où, la condition (iv).

Conclusion : $(\overline{x} \cdot \overline{u}, \overline{y} \cdot \overline{v})$ est un alignement de $(x \cdot u, y \cdot v)$.

Question 2

Notons $\mathcal{A}(x, y)$ l'ensemble des alignements de (x, y), pour $a \in \mathcal{A}(x, y)$, on note $|a|$ sa longueur.

Posons $\overline{x} = (-)^m x$ et $\overline{y} = y\,(-)^n$.

Alors : $\pi(\overline{x}) = x$ et $\pi(\overline{y}) = y$, puis $|\overline{x}| = m + n = n + m = |\overline{y}|$.

Enfin, pour tout $i \in [[1, m]]$, on a : $\overline{y}_i \neq -$, pour tout $i \in [[m+1, n+m]]$, on a : $\overline{x}_i \neq -$ par construction.

Donc, $(\overline{x}, \overline{y})$ est un alignement de (x, y) de longueur $n + m$.

Supposons qu'il existe un alignement $U = (u, v)$ de longueur $K \geq n + m + 1$.

Notons $G(m) = \{i \in [[1, |m|]] \mid m_i = -\}$ les positions des gaps du mot m et $T(m)$ son complémentaire.

Puisque U est un alignement et par (i), on a : $\operatorname{card} G(u) = |u| - |x| \geq m + 1$.

De même, par (iv) et (iii), on a : $T(v) = G(u)$.

Or : $\operatorname{card} T(v) = |y| = m$ par (ii).

Donc : $m = \operatorname{card} T(v) = \operatorname{card} G(u) \geq m + 1$, ce qui est absurde.

Conclusion : La longueur maximale d'un alignement de (x, y) est $n+m$, atteinte pour le mot construit plus haut.

Question 3

La question se ramène à dénombrer l'ensemble des parties à k éléments de $[[1, n+k]]$ (ensemble des indices du mot final).

Une fois celui-ci fixé, le mot final est entièrement déterminé en remplissant les indices vides par les lettres de x.

Conclusion : Il y a $\binom{n+k}{k}$ mots possibles obtenus à partir de x en ajoutant exactement k gaps.

Question 4

Une fois qu'on a un mot $\overline{x}$ à partir de x en ajoutant exactement k gaps, ce mot devient de longueur $n+k$, donc il faut ajouter $n+k-m \geq 0$ gaps à y, afin que le résultat final soit aussi de longueur $n+k$ et respecte la condition (iii).

La question se ramène à présent à dénombrer l'ensemble des parties à $n+k-m$ éléments de $[[1, n+k]] \quad G(\overline{x})$ de cardinal $n+k-k = n$, de la même façon qu'à la question 3.

D'où, il existe $\binom{n}{n+k-m}$ façons d'insérer les gaps à y sans violer la condition (iv).

On note $\mathcal{A}^{(k)}(x,y) = \{U = (u,v) \in \mathcal{A}(x,y) \mid \operatorname{card} G(u) = k\}$, alors :

$$\mathcal{A}(x,y) = \bigsqcup_{k=0}^{m} \mathcal{A}^{(k)}(x,y)$$

Ainsi, notons $G_k = \operatorname{card} \mathcal{A}^{(k)}(x,y) = \binom{n+k}{k}\binom{n}{n+k-m}$.

Alors : $\operatorname{card} \mathcal{A}(x,y) = \sum_{k=0}^{m} G_k$ est le nombre d'alignements de (x,y).

Application numérique : Pour $|x| = 15$ et $|y| = 10$, on vérifie bien $|x| \geq |y|$ et on en tire que $\operatorname{card} \mathcal{A}(x,y) = 298199265$ dans ce cas.

Question 5

Fixons deux mots x, y tel que $|x| = n$ et $|y| = m$ et $n \geq m$.

Par la question précédente, soit $k \in [[0, m]]$ tel que $G_k = \max_{j \in [[0,m]]} G_j$.

$$\operatorname{card} \mathcal{A}(x,y) \leq m G_k$$

Or : $G_k \leq \dfrac{(n+k)^k}{k!} \dfrac{n^{n+k-m}}{(n+k-m)!} \leq \dfrac{(2n)^n n^{2n}}{k!(n+k-m)!} \leq (2n)^n n^{2n}$

Donc : $\operatorname{card} \mathcal{A}(x,y) = O(m(2n)^n n^{2n}) = O(m \exp(3n \log n))$ lorsque $n \to +\infty$.

Or, calculer la distance d'édition revient à énumérer les alignements de (x, y) et pour chacun d'eux, de calculer le coût qui se fait en itérant sur un mot de longueur $n + m$ au plus, d'après la question 3.

Donc, on en tire une complexité temporelle aux environs de $O((n + m)m \exp(3n \log n))$ lorsque $n \to +\infty$, donc du type exponentielle afin de calculer la distance d'édition entre x et y, ce qui revient aussi à trouver l'alignement de coût minimal, lorsqu'on procède par énumération exhaustive.

Question 6

Afin de trouver la distance d'édition entre deux mots, on peut avoir une variable qui garde en mémoire les distances qu'on voit et on énumère tous les alignements.

Stocker une variable de distance ne consomme qu'un entier, en revanche, énumérer tous les alignements peut se faire récursivement, ce qui consommera de la mémoire au niveau de la pile d'appel et de la fonction elle-même.

Afin de suivre les alignements qu'on construit, on peut se contenter d'allouer deux chaînes de taille $n + m$ et de les passer le long des appels récursifs et de réécrire sur la case qui nous intéresse, c'est toujours possible, puisque `DIST_NAIF_REC` ne regarde que les indices $i' \geq i$ et $j' \geq j$.

Enfin, on peut évaluer la profondeur de l'arbre des appels récursifs en basant sur `DIST_NAIF_REC` qui est de profondeur $n + m$ dans le pire cas.

Ainsi, la complexité spatiale selon l'implémentation exacte sera en $\Theta(n + m)$.

Question 7

Si $\overline{u}_l = -$, alors : $\overline{v}_l = y_j$ par (iv) et (ii).

Si $\overline{v}_l = -$, alors : $\overline{u}_l = x_i$ par (iv) et (i).

Si $\overline{u}_l \neq -$ et $\overline{v}_l \neq -$, alors : $\overline{u}_l = x_i$ et $\overline{v}_l = y_j$.

Question 8

On a, en notant $R = C(\overline{u}_{[1,\ldots,l-1]}, \overline{v}_{[1,\ldots,l-1]})$.

$$C(\overline{u}, \overline{v}) = c(\overline{u}_l, \overline{v}_l) + R$$

Donc, d'après la question 7,

1er cas : Si $\overline{u}_l = -$, alors : $C(\overline{u}, \overline{v}) = c_{\text{ins}} + R$.

2ème cas : Si $\overline{v}_l = -$, alors : $C(\overline{u}, \overline{v}) = c_{\text{del}} + R$.

3ème cas : Si $\overline{u}_l \neq -$ et $\overline{v}_l \neq -$, alors : $C(\overline{u}, \overline{v}) = c_{\text{sub}}(x_i, y_j) + R$.

Question 9

Au préalable, on notera $x' = x_{[1...i]}$ et $y' = x_{[1...j]}$, notons aussi :

$$U(i,j) = \min\{D(i-1,j-1) + c_{\text{sub}}(x_i, y_j), D(i-1,j) + c_{\text{del}}, D(i,j-1) + c_{\text{ins}}\}$$

Montrons que : $D(i,j) = U(i,j)$.

Soit (u,v) un alignement optimal de (x', y') de longueur l.

Distinguons les cas en utilisant la question 7.

1er cas : Si $u_l = -$.

Alors : $v_l = y_j$, donc : $Z = (u_{[1...l-1]}, v_{[1...l-1]})$ est un alignement de $(x_{[1...i]}, y_{[1...j-1]})$.

Or s'il existe un alignement de coût strictement inférieur à celui de Z, notons le W, on pourrait l'utiliser pour construire un alignement de coût strictement inférieur à celui de (u,v).

Donc, cela est absurde, ce qui entraîne que $D(i,j-1) = C(Z) = D(i,j) - c_{\text{ins}}$.

Donc : $D(i,j-1) + c_{\text{ins}} = D(i,j)$.

2ème cas : Si $v_l = -$

De façon symétrique avec le 1er cas.

On a : $D(i-1,j) + c_{\text{del}} = D(i,j)$.

3ème cas : Si $u_l \neq -$ et $v_l \neq -$.

Cette fois-ci, en posant $Z = (u_{[1...l-1]}, v_{[1...l-1]})$ est un alignement de $(x_{[1...i-1]}, y_{[1...j-1]})$.

Le même argument convient à prouver que Z est un alignement optimal.

D'où : $D(i,j) = D(i-1,j-1) + c_{\text{sub}}(x_i, y_j)$.

Conclusion : $\boxed{D(i,j) = U(i,j).}$

Question 10

On a : $D(0,0) = d(x_\emptyset, y_\emptyset) = d(\varepsilon, \varepsilon) = 0$.

Puisque l'unique alignement de $(\varepsilon, \varepsilon)$ est $(\varepsilon, \varepsilon)$ de longueur 0.

En effet, s'il existait un alignement de longueur $k \geq 1$, alors, en le notant (a,b), on aurait : $\pi(a) = \varepsilon$ et $\pi(b) = \varepsilon$, donc : a, b seraient intégralement constitués de gaps, or, la condition (iv) impose que deux gaps ne peuvent pas se produire à la même position.

Ce qui contredit le fait que a, b soient constitués de gaps intégralement, donc, il n'existe pas de tel alignement.

Conclusion : $D(0, 0) = 0$.

Question 11

Fixons $j \in [[1, m]]$.

On a : $D(0, j) = d(x_\emptyset, y_{[1,\dots,j]}) = d(\varepsilon, y_{[1,\dots,j]})$.

Alors, $D(0, j) = j$, en effet, soit (a, b) un alignement de $(\varepsilon, y_{[1,\dots,j]})$.

Alors, puisque $\pi(a) = \varepsilon$, on a : $a = (-)^{|a|}$.

Or : $\pi(b) = y_{[1,\dots,j]}$ entraîne $|b| \geq j$.

De plus, $|a| = |b|$ fournit $|a| \geq j$.

Or, la condition (iv) force b à ne comporter aucun gap, donc : $|b| \leq j$.

D'où : $|a| = |b| = |y|$ et il existe un unique alignement $((-)^j, y_{[1\dots j]})$ de longueur j et de coût jc_{ins}, donc c'est le minimum.

Par symétrie des rôles joués par x, y, on prouve que $D(i, 0) = i$ pour tout $i \in [[1, n]]$.

Conclusion : $\forall i \in [[1, n]], D(i, 0) = ic_{\text{del}}$ et $\forall j \in [[1, m]] = D(0, j) = jc_{\text{ins}}$.

Question 12

Algorithme 1 : DIST_1

Entrées : x, y deux mots de longueur n, m
Données : Tableau T à deux dimensions de taille $(n+1)(m+1)$
Sorties : $T[n, m]$
début
  $T(0,0) \longleftarrow 0$
  pour *i allant de 1 à n* **faire**
    $T(i,0) \leftarrow ic_{\text{del}}$
  fin
  pour *j allant de 1 à m* **faire**
    $T(0,j) \leftarrow jc_{\text{ins}}$
  fin
  pour *i allant de 1 à n* **faire**
    pour *j allant de 1 à m* **faire**
      $A \leftarrow T(i-1,j) + c_{\text{ins}}$
      $S \leftarrow T(i,j-1) + c_{\text{del}}$
      $C \leftarrow T(i-1,j-1) + c_{\text{sub}}(x_i, y_j)$
      $T(i,j) \leftarrow \min\{A, S, C\}$
    fin
  fin
fin

Question 13

On stocke un tableau de taille nm d'entiers d'au plus k bits.

Cela fournit une complexité spatiale en $\Theta(nmk)$.

Question 14

Le calcul de $U(i,j)$ en supposant que les valeurs précédentes sont stockées dans un tableau réutilisable ne revient qu'à effectuer 3 comparaisons, 3 sommes, un appel à c_{sub}.

Or, si on suppose que toutes ces opérations s'effectuent en temps constant, ce que l'on peut faire, selon l'architecture du processeur, si les entiers prennent au plus quatre mots par exemple (x86-64).

Alors, on en tire une complexité en $\Theta(ij)$ pour le calcul complet de $U(i,j)$ car on effectue ij itérations des opérations précédentes.

Conclusion : Le calcul de `DIST_1` recourant à l'appel de $U(n,m)$, il se fait donc en $\Theta(nm)$.

Question 15

Traitons le cas où $j > 0$ et $D(i,j) = D(i,j-1) + c_{\text{ins}}$.

Soit $(s,t) \in \text{Al}^*(i,j-1)$.

Or : $(-, y_j)$ est alignement de (ε, y_j) et (s,t) est alignement de $(x_{[1\ldots i]}, y_{[1\ldots j-1]})$.

Par la question 1, $(s \cdot -, t \cdot y_j)$ est donc alignement de $(x_{[1\ldots i]}, y_{[1\ldots j]})$.

Ensuite, $C(s \cdot -, t \cdot y_j) = c_{\text{ins}} + C(s,t) = c_{\text{ins}} + D(i,j-1) = D(i,j)$.

Conclusion : $(s \cdot -, t \cdot y_j) \in \text{Al}^*(i,j)$.

Question 16

Algorithme 2 : SOL_1

Entrées : x, y deux mots de longueur n, m et un tableau T indexé par $[[0, n-1]] \times [[0, m-1]]$ contenant les valeurs de D

Sorties : Un couple (u, v) alignement optimal de (x, y)

début

- $u \longleftarrow \varepsilon$
- $v \longleftarrow \varepsilon$
- $i \longleftarrow n-1$
- $j \longleftarrow m-1$
- **tant que** $i > 0$ *ou* $j > 0$ **faire**
  - **si** $i > 0$ *et* $j > 0$ *et* $T(i,j) = T(i-1,j-1) + c_{ins}$ **alors**
    - $u \leftarrow x_i \cdot u$
    - $v \leftarrow y_j \cdot v$
    - $i \leftarrow i-1$
    - $j \leftarrow j-1$
  - **fin**
  - **sinon si** $i > 0$ *et* $T(i,j) = T(i-1,j) + c_{del}$ **alors**
    - $u \leftarrow x_i \cdot u$
    - $v \leftarrow - \cdot v$
    - $i \leftarrow i-1$
  - **fin**
  - **sinon**
    - $u \leftarrow - \cdot u$
    - $v \leftarrow y_j \cdot v$
    - $j \leftarrow j-1$
  - **fin**
- **fin**

fin

Question 17

Dans le pire cas, `SOL_1` effectue $n + m$ itérations (d'abord, il décremente i entièrement puis j par exemple, ou vice-versa).

Or : $n + m = O(nm)$.

Conclusion : Le problème `ALI` est donc résolu en $\Theta(nm)$, puisque `DIST_1` se calcule en $\Theta(nm)$.

Question 18

D'après la question 13, `DIST_1` est de complexité spatiale $\Theta(nmk)$ où k est une majoration du nombre de bits des entiers utilisés.

Puisque `SOL_1` n'alloue que deux chaînes de caractère de taille au plus n et respectivement m et deux entiers bornés par n et m respectivement, i.e. est donc de complexité spatiale $O(n + m)$.

Or : $n + m = O(nmk)$.

Conclusion : `ALI` est résolu en complexité spatiale $\Theta(nmk)$.

Question 19

À toute itération de l'algorithme `DIST_1`, la ligne i et la ligne $i - 1$ est employée.

Question 20

Il suffit donc de garder que deux lignes à la fois, ce qui fournira une complexité spatiale linéaire.

Algorithme 3 : DIST_2

Entrées : x, y deux mots de longueur n, m
Données : LigneCourante et DerniereLigne représentant les deux dernières lignes du tableau D à tout instant
Sorties : LigneCourante$(m) = d(x, y)$
début
  LigneCourante(0) $\longleftarrow$ 0
  pour *j allant de 0 à m* **faire**
    DerniereLigne$(j) \leftarrow jc_{\text{ins}}$
  fin
  pour *i allant de 1 à n* **faire**
    LigneCourante(0) $\leftarrow ic_{\text{del}}$
    pour *j allant de 1 à m* **faire**
      $A \leftarrow$ DerniereLigne$(j) + c_{\text{ins}}$
      $S \leftarrow$ LigneCourante$(j-1) + c_{\text{del}}$
      $C \leftarrow$ DerniereLigne$(j-1) + c_{\text{sub}}(x_i, y_j)$
      LigneCourante$(j) \leftarrow \min\{A, S, C\}$
    fin
    DerniereLigne $\leftarrow$ LigneCourante
  fin
fin

Question 21

Algorithme 4 : mot_gaps

Entrées : Un entier naturel k
Sorties : m le mot constitués de k gaps
début
  si $k = 0$ **alors**
    retourner ε
  fin
  sinon
    retourner $-$mot_gaps$(k-1)$
  fin
fin

Question 22

Fixons x, y deux mots, x de longueur 1 et y de longueur $m \geq 1$.

Alors, il faut examiner s'il vaut mieux faire une suppression ou une substitution, cela revient à calculer :

$$k_0 = \underset{k \in [[1,m]]}{\operatorname{argmin}} c_{\text{sub}}(x_1, y_k)$$

Ensuite, on compare : $c_{\text{sub}}(x_1, y_{k_0})$ et $c_{\text{del}} + c_{\text{ins}}$.

On sait par ailleurs que l'alignement optimal vérifie :

$$\begin{aligned} C(u, v) &= \min\{(m-1)c_{\text{ins}} + c_{\text{sub}}(x_1, y_{k_0}), mc_{\text{ins}} + c_{\text{del}}\} \\ &= (m-1)c_{\text{ins}} + \min\{c_{\text{sub}}(x_1, y_{k_0}), c_{\text{ins}} + c_{\text{del}}\} \end{aligned}$$

En effet, si on décide d'aligner une lettre, on peut construire un alignement de longueur m ($v = y$). Si on décide de procéder à une suppression, on doit construire un alignement de longueur $m+1$ (par exemple, $v = y-$ et $u = (-)^m x$). Toute autre décision ajoute un coût strictement supérieur aux coûts précédents.

Algorithme 5 : align_lettre_mot

Entrées : Un mot x de longueur 1 et y un mot de longueur $m \geq 1$
Sorties : (u, v) un alignement optimal de (x, y)
début
  $k_0 \longleftarrow \operatorname{argmin}_{k \in [[1,m]]} c_{\text{sub}}(x_1, y_k)$
  si $c_{del} + c_{ins} \geq c_{sub}(x_0, y_{k_0})$ **alors**
    si $k_0 = 1$ **alors**
      retourner $(x_1 \cdot \texttt{mot\_gaps}(m-1), y)$
    fin
    sinon si $k_0 = m$ **alors**
      retourner $(\texttt{mot\_gaps}(m-1) \cdot x_1, y)$
    fin
    sinon
      retourner $(\texttt{mot\_gaps}(k_0-1) \cdot x_1 \cdot \texttt{mot\_gaps}(m-k_0), y)$
    fin
  fin
  sinon
    retourner $(\texttt{mot\_gaps}(m) \cdot x_1, y \cdot -)$
  fin
fin

Question 23

On se donne $(\overline{s}, \overline{t}) = (BAL\text{—}, \text{——}RO)$, qui est un alignement de x^1 de longueur 5 et de coût : $3c_{\text{del}} + 2c_{\text{ins}} = 5c_{\text{del}} = 15$.

Il est optimal car toute substitution (aucune lettre n'est en commun) est plus coûteuse qu'une insertion ou une suppression. Et il ne peut pas être raccourci puisque sa longueur est minorée par $\left|x^1\right| + \left|y^1\right| = 5$.

On se donne $(\overline{u}, \overline{v}) = (LON\text{–}, \text{—}ND)$, qui est un alignement de (x^2, y^2) de longueur 4 et de coût : $c_{\text{del}} + 2c_{\text{ins}} = 3c_{\text{ins}} = 9$.

Il est optimal car toute autre substitution que celle effectuée avec N sera de coût plus grand qu'une insertion ou une suppression, et une substitution supplémentaire aura un coût supérieur puisqu'il n'y a que N qui est en commun. Et il ne peut pas être raccourci puisque sa longueur est minorée par $\left|x^2\right| + \left|y^2\right| - 1 = 4$ (on compte la substitution qu'une fois).

On remarque que $(BALLON\text{–}, \text{——}ROND)$ est un alignement de (x, y) de longueur 7 et de coût : $3c_{\text{ins}} + c_{\text{sub}}(L, R) + c_{\text{del}} = 12 + 5 = 17$ puisque L et R sont des consonnes distinctes.

À présent, si on regarde l'alignement obtenu par la concaténation des alignements précédents : $(BAL\text{—}LON\text{–}, \text{——}RO\text{—}ND)$ est de coût : $3c_{\text{ins}} + 2c_{\text{del}} + 2c_{\text{ins}} + c_{\text{del}} = 8c_{\text{ins}} = 24 > 17$.

Conclusion : L'alignement obtenu ne peut pas être optimal.

Question 24

Algorithme 6 : SOL_2

Entrées : (x, y) un couple de mots de longueur n, m
Sorties : (u, v) un alignement minimal de (x, y)
début
  si $n = 0$ *ou* $m = 0$ **alors**
    retourner $((-)^m x, (-)^n y))$
  fin
  sinon si $n = 1$ **alors**
    retourner align_lettre_mot(x, y)
  fin
  sinon si $m = 1$ **alors**
    retourner la permutation des couples de align_lettre_mot(y, x)
  fin
  sinon
    $i^* \leftarrow \left\lfloor \frac{n}{2} \right\rfloor$
    $j^* \leftarrow \text{coupure}(x, y)$
    $(s, t) \leftarrow \text{SOL\_2}(x_{[0 \dots i^*]}, y_{[0 \dots j^*]})$
    $(u, v) \leftarrow \text{SOL\_2}(x_{[i^*+1 \dots n]}, y_{[j^*+1 \dots m]})$
    retourner $(s \cdot u, t \cdot v)$
  fin
fin

Question 25

Algorithme 7 : coupure

Données : DDerniereLigne, DLigneCourante représentent la dernière ligne et la ligne courante du tableau D, et pour IDerniereLigne, ILigneCourante la dernière ligne et la ligne courante du tableau I

Entrées : Un couple (x, y) de mots de longueur n, m respectivement vérifiant $n, m \geq 2$

Sorties : L'indice j^* associée à i^* comme défini dans le projet

```
début
  DLigneCourante(0) ← 0
  ILigneCourante(0) ← 0
  pour j allant de 0 à m faire
    DDerniereLigne(j) ← j c_ins
    IDerniereLigne(j) ← j
  fin
  pour i allant de 1 à n faire
    DLigneCourante(0) ← i c_ins
    pour j allant de 1 à m faire
      A ← DDerniereLigne(j) + c_ins
      S ← DLigneCourante(j − 1) + c_del
      C ← DDerniereLigne(j − 1) + c_sub(x_i, y_j)
      DLigneCourante(j) ← min{A, S, C}
      si i > i* alors
        si DLigneCourante(j) = A alors
          ILigneCourante(j) ← IDerniereLigne(j)
        fin
        sinon si DLigneCourante(j) = S alors
          ILigneCourante(j) ← ILigneCourante(j − 1)
        fin
        sinon
          ILigneCourante(j) ← IDerniereLigne(j − 1)
        fin
      fin
    fin
    si i > i* alors
      IDerniereLigne ← ILigneCourante
    fin
    DDerniereLigne ← DLigneCourante
  fin
  retourner ILigneCourante(m)
fin
```

Question 26

À chaque tour de boucle, on a trois entiers de k bits au plus.

On ne retient que quatre lignes de longueur m d'entiers de k bits durant tout le long des boucles.

D'où une complexité spatiale en $\Theta(mk)$, linéaire en m donc.

Remarque : En pratique, on pourrait obtenir une complexité en $\Theta(\min\{n, m\}k)$ si on place toujours le mot le plus court en second argument, quitte à permuter les alignements obtenus à la fin.

Question 27

Les opérations de `SOL_2` se limitent à allouer la solution de taille $n+m$ au plus, allouer deux entiers, faire appel à `coupure`, appeler récursivement `SOL_2` sur des instances où x' est de taille $n' \leq n/2$, puis concaténer les résultats.

Supposons qu'on puisse couper les chaînes de caractère sans qu'elle prenne plus de place en mémoire.

Alors, on a deux entiers majorées par n et m respectivement (i^* et j^*).

Ainsi, les seuls coûts mémoires deviennent la longueur des alignements dont la somme des longueurs est majorée par $n + m$ et la pile d'appel.

Or, on peut majorer la profondeur de la pile d'appel par la profondeur de l'arbre des appels récursifs qui est en $O(\log_2 n)$ en raison de i^* (qui va entraîner la fin des appels aussitôt qu'il sera vide ou de longueur 1).

Conclusion : On en tire une complexité spatiale en $O(mk + (n + m) + \log n)$, ce que l'on peut réécrire en : $\boxed{O(mk + n).}$

Remarque : Dans le cas où on peut majorer le nombre de bits par une constante, on obtient une complexité spatiale en $O(n + m)$, linéaire.

Question 28

Supposons qu'on dispose de deux mots de longueur n, m respectivement que l'on passe à `coupure`.

On effectue $\Theta(nm)$ itérations, chaque itération peut se faire en $O(1)$ (trois sommes, un appel à , un minimum, trois comparaisons, six indexations, trois assignations) et on peut implémenter les copies du tableau en $O(m)$.

Conclusion : `coupure` est de complexité temporelle en $\Theta(nm)$.

Question 29

D'un point de vue théorique, on remarque qu'on recalcule `coupure` trop souvent, donc expérimentalement, on s'attend à constater une perte de vitesse.

En pratique, on trace les temps CPU de `SOL_1` et `SOL_2` qu'on peut retrouver figure 5 avec des échelles logarithmiques en abscisses et en ordonnées.

On constate que `SOL_2` est plus lent au départ puis beaucoup plus rapide vers la fin que `SOL_1`, on peut avancer plusieurs raisons :

(1) `SOL_2` est écrit en utilisant `Data.Vector.Unboxed.Mutable` dans un contexte monadique `ST`, ce qui n'est pas le cas de `SOL_1`, ainsi il bénéficie d'optimisations importantes (notamment car `Int` est un type primitif et il y a des spécialisations faites en ce sens-là).

(2) GHC est un compilateur très agressif qui fonctionne mieux sur un style récursif plutôt qu'impératif : `SOL_2` est récursif tandis que `SOL_1` recourt à un appel de calcul du tableau D qui lui est itératif.

(3) La localité durant le parcours du tableau D n'est pas nécessairement assurée dans `SOL_1` tandis que dans `SOL_2`, on s'en assure.

On peut facilement imaginer que lorsque n est petit, les optimisations ne sont pas si intéressantes que ça, cependant lorsque n est grand, GHC montre son efficacité.

Cela choque notre intuition concernant le recalcul des `coupure` néanmoins.

Afin de vérifier cette assertion, j'ai décidé de réécrire `SOL_1` en `SOL_1'` avec un style `ST` pour le calcul du tableau D de façon mutable, je présenterai les résultats durant la soutenance.

Tâche A

On observe que l'implémentation est valide sur les instances fournies.

On constate pour $n = 10$ avec `time` :

```
1.08user 0.10system 0:01.17elapsed 101%CPU (0avgtext+0avgdata 161452maxresident)k
0inputs+8outputs (0major+24557minor)pagefaults 0swaps
```

Puis pour $n = 13$ avec `time` :

```
41.33user 0.34system 0:42.07elapsed 99%CPU (0avgtext+0avgdata 161484maxresident)k
0inputs+0outputs (0major+24562minor)pagefaults 0swaps
```

On vérifie pour $n = 15$ avec `time` :

```
540.91user 1.61system 9:04.83elapsed 99%CPU (0avgtext+0avgdata 161824maxresident)k
0inputs+8outputs (0major+24791minor)pagefaults 0swaps
```

On conclut donc que $n = 13$ est la limite pour calculer sous moins d'une minute.

Quant à la consommation RAM, on donne ici les calculs effectués par l'instrumentation d'Haskell durant l'exécution :

```
  7,012,881,760 bytes allocated in the heap
  3,816,895,368 bytes copied during GC
  1,173,899,880 bytes maximum residency (12 sample(s))
      3,364,248 bytes maximum slop
           1119 MB total memory in use (0 MB lost due to fragmentation)

                                    Tot time (elapsed)  Avg pause  Max pause
 Gen  0      6035 colls,     0 par    3.014s   3.189s     0.0005s    0.0019s
 Gen  1        12 colls,     0 par    1.671s   2.400s     0.2000s    1.1812s

 INIT    time    0.000s  (  0.000s elapsed)
 MUT     time    3.782s  (  3.956s elapsed)
 GC      time    4.685s  (  5.589s elapsed)
 RP      time    0.000s  (  0.000s elapsed)
 PROF    time    0.000s  (  0.000s elapsed)
 EXIT    time    0.000s  (  0.000s elapsed)
 Total   time    8.468s  (  9.546s elapsed)

 %GC     time       0.0%  (0.0% elapsed)

 Alloc rate    1,854,169,786 bytes per MUT second

 Productivity  44.7% of total user, 41.4% of total elapsed
```

Pour une instance $n = 12$, on constate donc qu'1 GiB de RAM ont été utilisé pour le calcul, ce qui est attendu compte tenu de l'absence d'optimisation de la façon naïve de calculer.

Tâche B

On vérifie de la même façon qu'avec la tâche A les instances connues.

De plus, on vérifie que les sorties de `PROG_DYN` vérifient les conditions d'un alignement sur toutes les instances faisables en moins de dix minutes.

On trace la courbe de temps CPU de `DIST_1` et `SOL_1` dans la figure 1.

Tâche C

On vérifie de la même façon qu'avec la tâche A les instances connues.

On trace la courbe de temps CPU de `DIST_1` et `DIST_2` dans la figure 3.

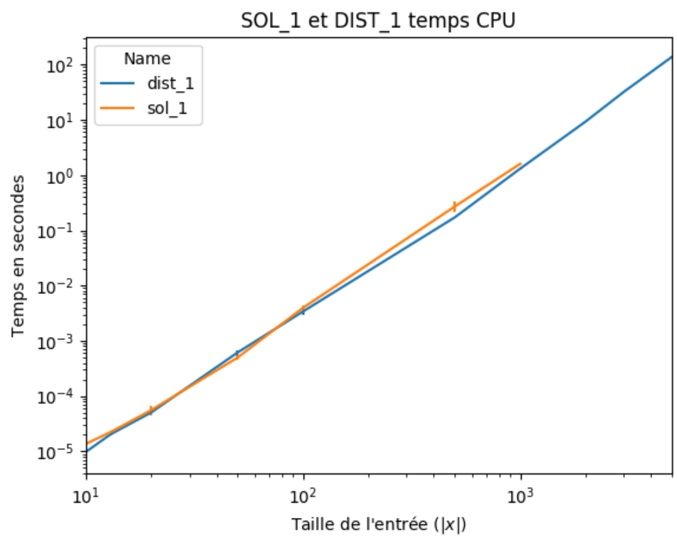

FIG. 1 : Moyenne du temps CPU (statistiquement signifiant, $R^2 > 0.9$) en échelle logarithmique sur les deux axes pour `DIST_1`, `SOL_1`

Tâche D

On vérifie comme dans la tâche B que `SOL_2` retourne des alignements correctement produits.

On trace la courbe de temps CPU de `SOL_2` dans la figure 3.

Comparatif

Un mot sur la méthodologie de la mesure temps CPU et RAM

Tant que c'est possible, les comparatifs sont issus de calculs effectués assez de fois pour que la variance soit minimale et que l'écart-type reste acceptable.

D'ailleurs, les tracés sont effectués avec les erreurs, mais la plupart des erreurs sont de l'ordre de la milliseconde donc ne sont pas visible.

Cependant, certaines fonctions ne peuvent pas tourner assez de fois pour minimiser leur variance, par exemple `DIST_NAIF` est trop lente pour qu'on puisse faire des statistiques sérieuses.

Contrairement à la consommation RAM qui est facilement reproducible car les allocateurs (de GHC) n'ont pas un comportement indéterministe dans des conditions idéales.

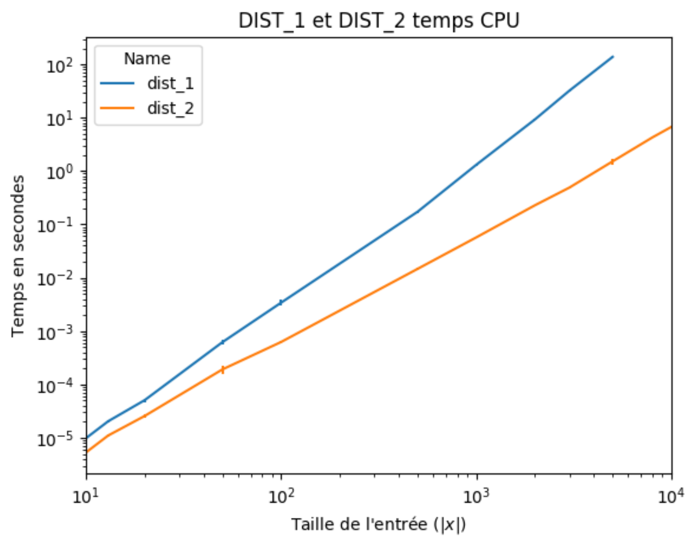

FIG. 2 : Moyenne du temps CPU (statistiquement signifiant, $R^2 > 0.9$) en échelle logarithmique sur les deux axes pour `DIST_1`, `DIST_2`

C'est pour cela qu'on verra dans certains tracés l'absence de points pour certaines fonctions puisque cela prendrait trop de temps à calculer de façon statistiquement signifiant. Ce n'est pas grave puisqu'une fonction qui prend trop de temps à être calculé statistiquement est une fonction dont les points seront trop haut sur l'hypothétique courbe complète.

Enfin, on a bien veillé à employer les formes normales (ou la weak head normal form si cela suffisait) des fonctions en Haskell lors des mesures pour éviter de fausser le calcul en raison du comportement d'évaluation paresseux.

Les courbes de consommation RAM seront mis dans les slides de la soutenance, en attendant, les tableaux de consommation RAM des expériences sont joints à la fin.

Distances d'édition

On trace la courbe de temps CPU de `DIST_1`, `DIST_2`, `DIST_NAIF` sur les entrées faisables par les fonctions dans la figure4

Calcul d'un alignement optimal

On trace la courbe de temps CPU de `PROG_DYN` et `SOL_2` tant qu'ils prennent pas plus de dix minutes (par instance) qu'on peut retrouver à la figure 5.

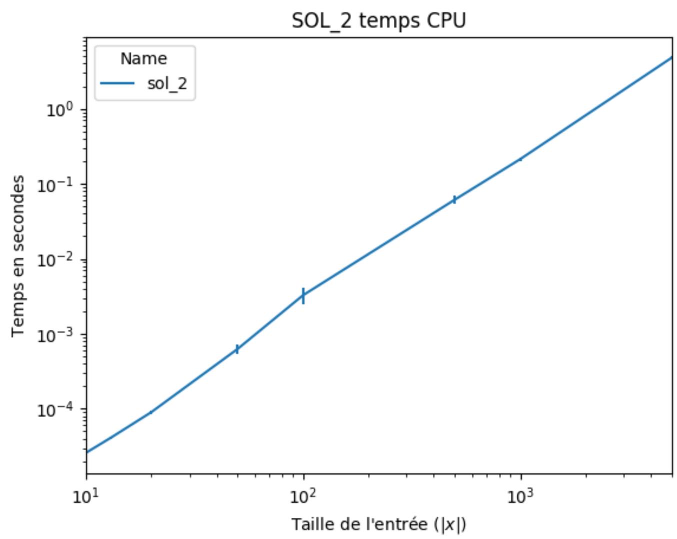

FIG. 3 : Moyenne du temps CPU (statistiquement signifiant, $R^2 > 0.9$) en échelle logarithmique sur les deux axes pour `SOL_2`

Consommation RAM (tableaux)

Cas	Allocations (en octets)	GCs (ramasse-miette, en octets)
dist_2(10)	0	0
dist_1(10)	0	0
dist_2(12)	0	0
dist_1(12)	0	0
dist_2(13)	0	0
dist_1(13)	0	0
dist_2(14)	0	0
dist_1(14)	0	0
dist_2(15)	0	0
dist_1(15)	0	0
dist_2(20)	0	0
dist_1(20)	0	0
dist_2(50)	0	0
dist_1(50)	0	0
dist_2(100)	0	2
dist_1(100)	0	3
dist_2(500)	0	59
dist_1(500)	10,315,168	91
dist_2(1000)	0	232
dist_1(1000)	40,442,336	359

Cas	Allocations (en octets)	GCs (ramasse-miette, en octets)
dist_2(2000)	0	929
dist_1(2000)	240,446,144	1,441
dist_2(3000)	0	2,085
dist_1(3000)	551,022,480	3,234
dist_2(5000)	8,584	5,811
dist_1(5000)	1,470,465,344	9,066
dist_2(10000)	381,520	23,137

Cas	Allocations (en octets)	GCs (ramasse-miette, en octets)
prog_dyn(10)	0	0
sol_2(10)	0	0
prog_dyn(12)	0	0
sol_2(12)	0	0
prog_dyn(13)	0	0
sol_2(13)	0	0
prog_dyn(14)	0	0
sol_2(14)	0	0
prog_dyn(15)	0	0
sol_2(15)	0	0
prog_dyn(20)	0	0
sol_2(20)	0	0
prog_dyn(50)	0	0
sol_2(50)	0	1
prog_dyn(100)	0	3
sol_2(100)	2,792	7
prog_dyn(500)	10,315,112	91
sol_2(500)	0	175
prog_dyn(1000)	40,434,408	359
sol_2(1000)	0	704
prog_dyn(2000)	239,912,392	1,442
sol_2(2000)	0	2,770
prog_dyn(3000)	551,520,008	3,261
sol_2(3000)	86,008	6,322
prog_dyn(5000)	1,445,826,880	8,935
sol_2(5000)	690,808	17,480
sol_2(10000)	2,418,336	70,119

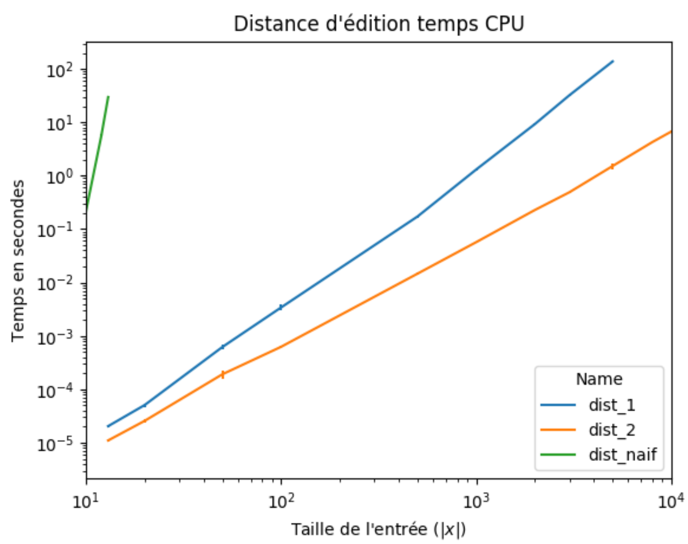

FIG. 4 : Moyenne du temps CPU (statistiquement signifiant, $R^2 > 0.9$) en échelle logarithmique sur les deux axes pour `DIST_1`, `DIST_2`, `DIST_NAIF`

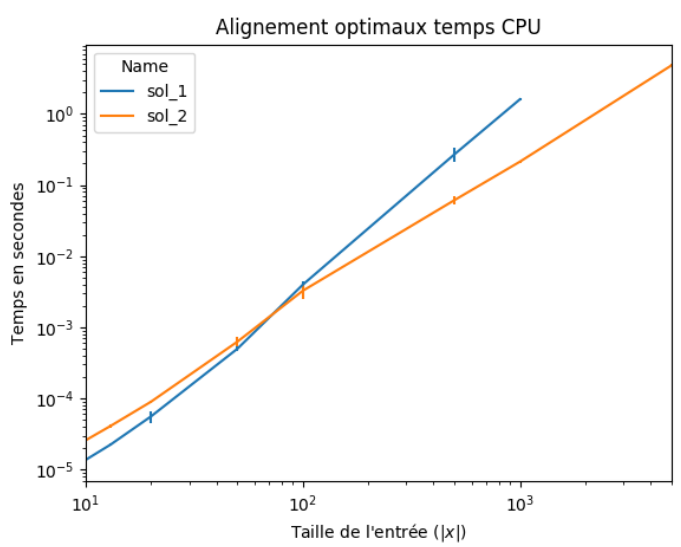

FIG. 5 : Moyenne du temps CPU (statistiquement signifiant, $R^2 > 0.9$) en échelle logarithmique sur les deux axes pour `SOL_1`, `SOL_2`