



PROJET LU3IN003

Alignment de séquences

Dao Quoc Hiep , Elias Bendjaballah

RAPPORT

4 décembre 2019

Table des matières

1	Le problème d'alignement des séquences	2
1.1	Alignement de deux mots	2
2	Algorithmes pour l'alignement de séquences	2
2.1	Méthode naïve par énumération	2
3	Programmation dynamique	5
3.1	Calcul de la distance d'édition par programmation dynamique	5
3.2	Calcul d'un alignement optimal par programmation dynamique	7
3.3	Amélioration de la complexité spatiale du calcul de la distance	10
4	Amélioration de la complexité spatiale du calcul d'un alignement optimal par la méthode "diviser pour régner"	13
4.1	Question 29	18

1 Le problème d'alignement des séquences

1.1 Alignement de deux mots

Question 1

On montre que si (\bar{x}, \bar{y}) est un alignement de (x, y) et (\bar{u}, \bar{v}) un alignement de (u, v) alors $(\bar{x} \cdot \bar{u}, \bar{y} \cdot \bar{v})$ est un alignement de $(x \cdot u, y \cdot v)$.

Par l'absurde, supposons que (\bar{x}, \bar{y}) soit un alignement de (x, y) et (\bar{u}, \bar{v}) un alignement de (u, v) et que $(\bar{x} \cdot \bar{u}, \bar{y} \cdot \bar{v})$ ne soit pas un alignement de $(x \cdot u, y \cdot v)$. Alors, il existerait au moins un déséquencement dans $(\bar{x} \cdot \bar{u}, \bar{y} \cdot \bar{v})$. Ce déséquencement peut alors se trouver dans l'un des sous-mots \bar{x} , \bar{y} , \bar{u} ou \bar{v} . S'il se trouve dans \bar{x} ou \bar{y} alors (par définition d'un alignement et selon la condition (iii) $|\bar{x}| = |\bar{y}|$ ou n'importe quelle autre condition provoquant un déséquencement), (\bar{x}, \bar{y}) n'est pas un alignement de (x, y) ce qui est en contradiction avec l'hypothèse de départ. De manière similaire, si le déséquencement se trouve dans \bar{u} ou \bar{v} alors (\bar{u}, \bar{v}) n'est pas un alignement de (u, v) ce qui contredit l'hypothèse de départ.

On en déduit que si (\bar{x}, \bar{y}) est un alignement de (x, y) et (\bar{u}, \bar{v}) un alignement de (u, v) alors $(\bar{x} \cdot \bar{u}, \bar{y} \cdot \bar{v})$ est un alignement de $(x \cdot u, y \cdot v)$.

Question 2

La longueur maximale d'un alignement (x, y) est de : $|x| + |y|$ soit une valeur de $n + m$.

Cette valeur correspond au cas où les déséquencements ne sont corrigés que par des insertions et des délétions. Un exemple possible est :

Si l'on considère $x=ATTAGCGTTA$ et $y=AGCTCGATGA$, on peut obtenir :

$\bar{x} :$	A	T	T	A	G	C	G	T	T	A	-	-	-	-	-	-	-	-	-	-	-	-	-
$\bar{y} :$	-	-	-	-	-	-	-	-	-	-	A	G	C	T	C	G	A	T	G	A			

2 Algorithmes pour l'alignement de séquences

2.1 Méthode naïve par énumération

Question 3

Le nombre de mots de longueur n pouvant être obtenus en ajoutant exactement k gaps est donné par :

$$\binom{n+k}{k} = C_{n+k}^k = \frac{(n+k)!}{k!(n+k-k)!} = \frac{(n+k)!}{k!n!}$$

Il s'agit du nombre de manières possibles d'arranger les k gaps du mot \bar{x} parmi les $n+k$ possibilités engendrées par ces insertions.

Question 4

Sachant que $n \geq m$ et que k gaps sont ajoutés à x , on aura $k' = n - m + k$ gaps ajoutés à y . Avec $n - m$ les gaps à ajouter en raison de la possible différence de taille entre x et y et k les gaps devant être ajoutés pour satisfaire le (iii) de la définition d'un alignement ($|\bar{x}| = |\bar{y}|$). Le nombre de manières possibles d'insérer ces gaps dans y , sachant qu'un gap de \bar{y} ne doit pas être à la même position qu'un gap de \bar{x} est donné par :

$$C_n^{k'} = \binom{n}{n-m+k} = \frac{n!}{(n-m+k)!(m-k)!}$$

On en déduit le nombre d'alignements possibles de (x, y) :

Pour chaque k gap(s) ajoutés, il est possible d'obtenir C_{n+k}^k versions de x et pour chaque x , on peut obtenir $C_n^{k'}$ versions de y donnant un alignement correct. Ce qui permet d'écrire :

$$A = \sum_{k=0}^m C_{n+k}^k \times C_n^{n-m+k}$$

$$A = \sum_{k=0}^m \frac{(n+k)!}{k!n!} \times \frac{n!}{(n-m+k)!(m-k)!} = \frac{(n+k)!}{k!(n-m+k)!(m-k)!}$$

Pour $|x| = 15$ et $|y| = 10$, on obtient un résultat de :

$$A = \sum_{k=0}^{10} C_{15+k}^k \times C_{15}^{5+k} = 298\,199\,265 \text{ alignements possibles.}$$

Question 5

Un algorithme naïf qui consisterait à énumérer tous les alignements de deux mots en vue de trouver la distance d'édition aurait une complexité exponentielle comme calculé à la question précédente. Il faudrait lister la totalité des alignements possibles pour ensuite retourner celui dont la distance d'édition est minimale.

$$\Theta\left(\sum_{k=0}^m \frac{(n+k)!}{k!(n-m+k)!(m-k)!}\right) \subset \mathcal{O}(n!)$$

Il en est de même pour un algorithme qui procéderait similairement pour retourner un alignement de coût minimal. Sa complexité serait aussi en $\mathcal{O}(n!)$.

Question 6

La complexité spatiale d'un algorithme naïf qui énumérerait tous les alignements de deux mots pour retourner la distance d'édition serait de $\mathcal{O}(n!)$ si on garde en mémoire à chaque étape et pour chaque possibilité, le coût total des modifications réalisées.

Un algorithme qui retournerait un alignement de coût minimal serait dans le pire cas (en considérant les alignements les plus longs possibles, soit ceux de taille $n+m$) :

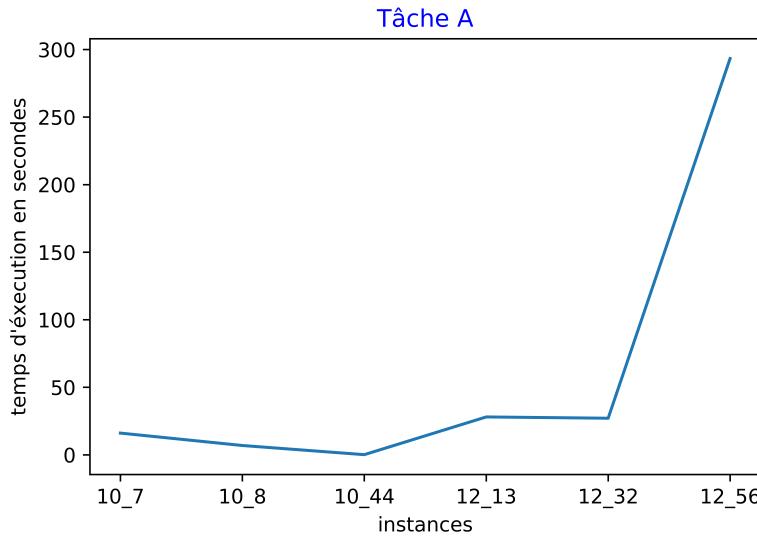
$$\mathcal{O}(n!) \times 2(n+m) \text{ soit } \mathcal{O}(n!2(n+m)) \subset \mathcal{O}(n!)$$

On passe en \mathcal{O} en raison du fait que ce cas n'est pas atteint.

Tâche A

Représentation du temps d'exécution de DIST_NAIF en fonction des instances testées :

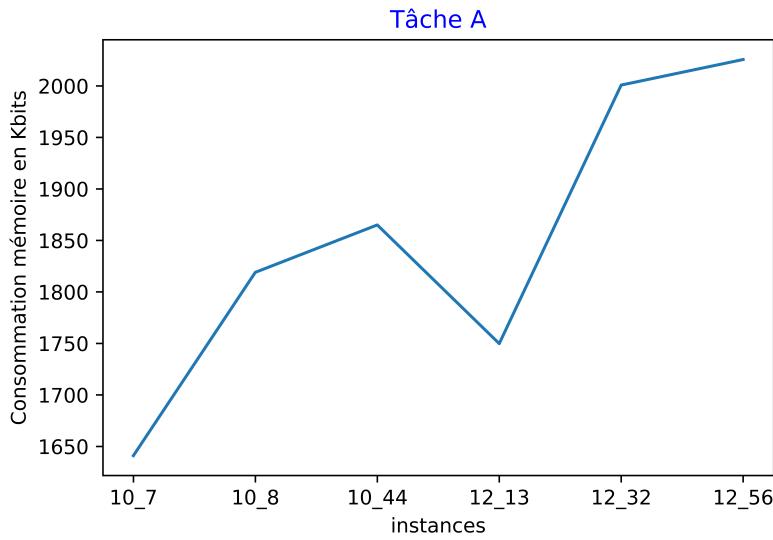
La première instance dont la résolution prend plus d'une minute est *Inst_0000012_56.adn* avec une durée d'exécution d'environ 293 secondes.



L'évolution de la courbe avec une augmentation importante pour *Inst_0000012_56.adn* où l'on a $|x| = 12$ et $|y| = 11$ suggère qu'il s'agit d'une fonction dont la complexité temporelle est exponentielle, ce qui est en accord avec le résultat de la question 5.

Cette augmentation brutale peut s'expliquer par le nombre important de possibilités d'alignements de cette instance. En effet, pour $|x| = 12$ et $|y| = 11$, on peut avoir d'après la formule de la question 4, $A = 103\,274\,625$ alignements possibles. Tandis que pour l'instance précédente *Inst_0000012_32.adn*, on a $|x| = 12$ et $|y| = 9$, ce qui engendre $A = 14\,218\,905$ alignements possibles. Soit bien moins que *Inst_0000012_56.adn*, d'où la différence des temps d'exécution des deux instances.

Représentation de la consommation mémoire de DIST_NAIF en fonction des instances testées : Nous avons essayé de mesurer la consommation mémoire de la tâche A mais avons rencontré quelques difficultés.



Il ne nous a été possible de tester que peu d'instances en raison de la complexité trop importante de la fonction. Les données recueillies ayant permis la réalisation du graphe précédent ne sont donc pas suffisantes pour avoir une estimation correcte. Au delà de *Inst_0000013_89.adn*, le système stoppe le programme en raison d'une consommation mémoire trop importante.

Cependant, sachant qu'en python un *int* est codé sur 16 bits, et si l'on ne considère que les paramètres (i, j, c et $dist$) dont la valeur évolue pour les fonctions étudiées, on peut écrire que l'on a : $4 \times 16 \times \mathcal{O}(n!)$ bits consommés, soit une complexité spatiale exponentielle.

3 Programmation dynamique

3.1 Calcul de la distance d'édition par programmation dynamique

Question 7

Soit (\bar{u}, \bar{v}) un alignement de $(x_{[1\dots i]}, y_{[1\dots j]})$ de longueur l .

Si $\bar{u}_l = -$ alors \bar{v}_l est un élément de l'alphabet Σ tel que $\bar{v}_l \in \{A, T, C, G\}$.

Si $\bar{v}_l = -$ alors \bar{u}_l est un élément de l'alphabet Σ .

Si $\bar{u}_l \neq -$ et $\bar{v}_l \neq -$, alors ils peuvent soit :

- correspondre au même caractère.
- être deux caractères appartenants à une paire concordante.
- être deux caractères appartenants à une paire non concordante.

Question 8

Si $\bar{u}_l = -$ alors : $C(\bar{u}, \bar{v}) = C(\bar{u}_{[1\dots l-1]}, \bar{v}_{[1\dots l-1]}) + c_{ins}$

Si $\bar{v}_l = -$ alors : $C(\bar{u}, \bar{v}) = C(\bar{u}_{[1\dots l-1]}, \bar{v}_{[1\dots l-1]}) + c_{del}$

Si $\bar{v}_l \neq -$ et $\bar{u}_l \neq -$ alors : $C(\bar{u}, \bar{v}) = C(\bar{u}_{[1\dots l-1]}, \bar{v}_{[1\dots l-1]}) + c_{sub}(\bar{u}_l, \bar{v}_l)$

Question 9

Pour $i \in [1..n]$ et $j \in [1..m]$, on peut en déduire l'expression de $D(i, j)$ à partir des valeurs de D à des rangs plus petits :

$$D(i, j) = \min\{D(i', j') + c_{sub}(i, j), D(i, j') + c_{del}, D(i', j) + c_{ins}\}$$

On divise le problème en sous-problèmes dont les résolutions successives permettent d'obtenir en fin d'exécution la solution au problème de départ. En exprimant $D(i, j)$ en fonction de trois sous-problèmes plus petits qui sont $D(i', j')$, $D(i, j')$ et $D(i', j)$ (réalisation d'une substitution, d'une insertion ou d'une délétion en vue d'arriver à un alignement optimal) dont nous calculons les valeurs puis choisissons celui dont la distance d'édition est minimale, nous remplissons un tableau à n lignes et m colonnes dont chaque case contient la solution au sous-problème considéré.

La structure de cette méthode de résolution peut aussi être vue comme un graphe orienté acyclique où chaque noeud représente un sous-problème (soit une case du tableau) et chaque arc, une contrainte de précédence sur l'ordre dans lequel les problèmes sont résolus. Si l'on passe de la case $(i - 1, j - 1)$ à la case (i, j) on réalise une substitution ou un match. Si l'on passe de la case $(i - 1, j)$ à la case (i, j) , on réalise une délétion et si l'on va de la case $(i, j - 1)$ à la case (i, j) , on procède à une insertion.

Question 10

Pour le cas de base $i = 0$ et $j = 0$, on aura $D(0, 0) = 0$, car considère deux mots vides ϵ . La distance d'édition est donc nulle et il n'y a aucun coût à prendre en compte.

Question 11

Pour les deux autres cas de base :

- Pour $i \in [1..n]$, on a donc : $D(i, 0) = i \times 2$

Cela correspond au cas où l'on calcule la distance d'édition entre le préfixe de taille 0 de y et le préfixe x , de taille i . Ceci nécessite l'ajout de i gaps en y , ce qui correspond à réaliser i délétions soit un coût de $i \times c_{del}$.

- Pour $j \in [1..m]$, on a donc : $D(0, j) = j \times 2$

Cela correspond au cas où l'on calcule la distance d'édition entre le préfixe de taille 0 de x et le préfixe de y , de taille j . Ceci se fait en ajoutant j gaps en x , ce qui correspond à j insertions soit un coût de : $j \times c_{ins}$.

Question 12

Algorithm 1: DIST_1

Data: x et y deux mots de tailles n et m tels que $(x, y) \in \Sigma^* \times \Sigma^*$

Résultat: La distance d'édition entre ces deux mots tout en remplissant le tableau T

Initialisation du tableau T de taille $(|x| + 1) \times (|y| + 1)$;

```

for  $i \in [0 \dots n + 1]$  do
    |  $T[i][0] = i \times 2;$ 
end
for  $j \in [1 \dots m + 1]$  do
    |  $T[0][j] = j \times 2;$ 
end
for  $i \in [1 \dots n + 1]$  do
    | for  $j \in [1 \dots m + 1]$  do
        | |  $T[i][j] = \min(T[i - 1][j] + c_{del}, T[i][j - 1] + c_{ins}, T[i - 1][j - 1] + c_{sub}(x[i], y[j]));$ 
    | end
end
Return  $T[n][m]$ 

```

Question 13

L'exécution de cet algorithme consiste à remplir un tableau de n lignes et m colonnes, ce qui revient donc à une complexité spatiale de $\Theta(nm)$. ($\Theta(nm) \times 16$ bits étant donné que chaque case contient un entier).

Question 14

Sachant que le même nombre d'opérations est réalisé à chaque calcul et en supposant que leur exécution se fait en temps constant, on peut en déduire que la complexité temporelle est elle aussi en $\Theta(nm)$ car DIST_1 calcule nm valeurs pour résoudre les sous-problèmes considérés.

3.2 Calcul d'un alignement optimal par programmation dynamique

Question 15

Soit $(i, j) \in [1 \dots n] \times [1 \dots m]$, on montre que : si $j > 0$ et $D(i, j) = D(i, j - 1) + c_{ins}$, alors $\forall (\bar{s}, \bar{t}) \in Al^*(i, j - 1)$, $(\bar{s} \cdot -, \bar{t} \cdot y_j) \in Al^*(i, j)$. Par l'absurde, si $j > 0$ et $D(i, j) = D(i, j - 1) + c_{ins}$, alors $\exists (\bar{s}, \bar{t}) \in Al^*(i, j - 1)$, $(\bar{s} \cdot -, \bar{t} \cdot y_j) \notin Al^*(i, j)$. Dans ce cas, la distance d'édition de $(\bar{s} \cdot -, \bar{t} \cdot y_j)$ est supérieure à n'importe quel alignement de $Al^*(i, j)$. Réaliser une insertion n'est donc pas le meilleur choix pour résoudre le sous-problème (i, j) et une autre opération aurait dû être effectuée. Contradiction avec le fait que $D(i, j) = D(i, j - 1) + c_{ins}$. Les trois cas présentés permettent de retrouver un alignement optimal possible en remontant de la dernière case du tableau à la première case en respectant les contraintes de précédence à chaque saut de case. Ces contraintes sont déduites des valeurs des cases du tableau T calculées par DIST_1.

Question 16

Algorithm 2: SOL_1

Data: x et y deux mots tels que $(x, y) \in \Sigma^* \times \Sigma^*$ et T un tableau indexé par $[1...|x|] \times [1...|y|]$ contenant les valeurs de D .

Réultat: Un alignement minimal de (x, y)

$u = [];$
 $v = [];$
 $i = |x|;$
 $j = |y|;$

while $i > 0$ and $j > 0$ **do**

if $T[i][j] = T[i][j - 1] + c_{ins}$ **then**
 insérer – en tête de u ;
 insérer y_{j-1} en tête de v ;
 $j = j - 1$;
 passer à l'itération suivante (par un continue);
 end
 if $T[i][j] = T[i - 1][j] + c_{del}$ **then**
 insérer x_{i-1} en tête de u ;
 insérer – en tête de v ;
 $i = i - 1$;
 passer à l'itération suivante (par un continue);
 end
 if $T[i][j] = T[i - 1][j - 1] + c_{sub}(x_{i-1}, y_{j-1})$ **then**
 insérer x_{i-1} en tête de u ;
 insérer substitution(y_{i-1}, x_{i-1}) soit x_{i-1} en tête de v ;
 $i = i - 1$;
 $j = j - 1$;
 passer à l'itération suivante (par un continue);
 end

end

while $i > 0$ **do**

 insérer x_{i-1} en tête de u ;
 insérer – en tête de v ;
 $i = i - 1$;

end

while $j > 0$ **do**

 insérer – en tête de u ;
 insérer y_{j-1} en tête de v ;
 $j = j - 1$;

end

Return u and v ;

Question 17

En combinant SOL_1 et $DIST_1$, on résout le problème ALI avec une complexité temporelle de : $\Theta(nm) + \Theta(n + m) \subset \Theta(nm)$. Soit une complexité totale de $\Theta(nm)$.

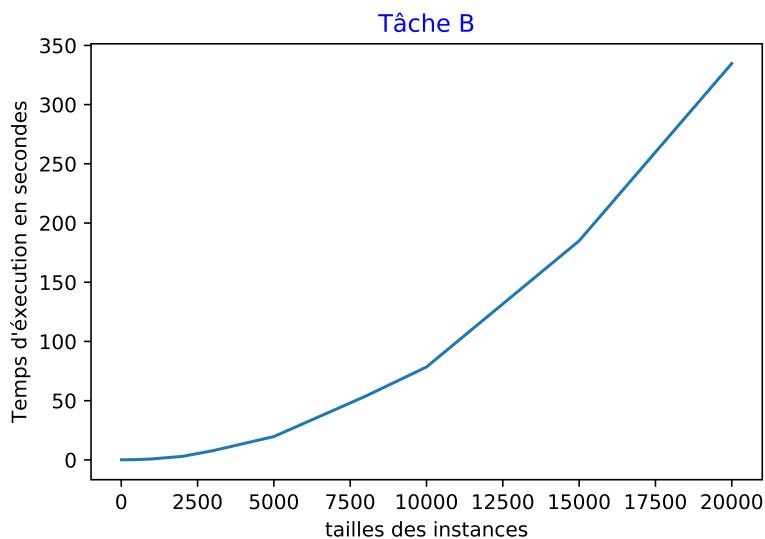
Avec $\Theta(nm)$ la complexité requise pour remplir le tableau T et $\Theta(n + m)$ la complexité pour effectuer le parcours permettant de retrouver un alignement optimal.

Question 18

La complexité spatiale de SOL_1 et $DIST_1$ combinés est en $\Theta(nm)$ car il s'agit toujours d'un tableau à remplir pour la fonction $DIST_1$ et du parcours de ce même tableau, de la dernière case jusqu'à la première en mettant à jour 2 listes dans lesquelles on stocke les résultats temporaires, ce qui ne modifie pas la complexité spatiale induite par $DIST_1$.

Tâche B

Représentation du temps CPU de l'exécution de PROG_DYN en fonction des instances testées (pour des instances nécessitant moins de 10 minutes chacune) :



A partir des instances de taille 50 000, l'exécution se suspend en raison d'une surconsommation de la mémoire utilisée. Il ne nous a donc pas été possible de tester des instances d'une taille supérieure à 20000.

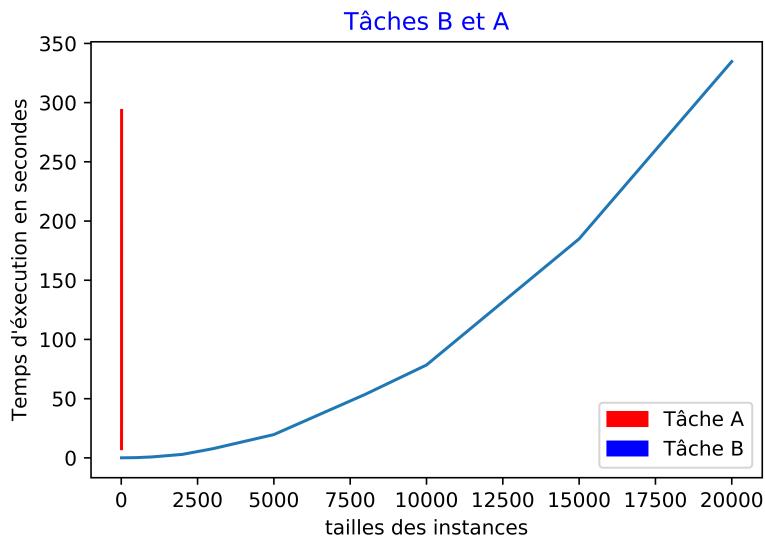
La courbe obtenue est en accord avec la complexité théorique. En effet, la courbe montre une évolution similaire à celle d'une fonction polynomiale. Sachant que $n \geq m$, et en supposant que $m \approx \frac{n}{2}$ pour les instances testées, on peut estimer que la courbe se rapproche d'une fonction du type $f(x) = \frac{x^2}{2}$.

Estimation de la consommation mémoire pour une instance de très grande taille :

Pour l'instance $Inst0010000_50.adn$ pour laquelle $|x| = 10000$ et $|y| = 8882$ et en considérant le fait que chaque caractère est codé sur 8 bits, on peut calculer que l'on aura : $10000 \times 8882 \times 8 =$

710 560 kbits utilisés. Ce que nous avons vérifié à l'aide d'une mesure expérimentale ayant donné un résultat proche ($\approx 715 000$ kbits).

On remarque une différence importante entre cet algorithme de programmation dynamique et l'algorithme naïf. Les temps d'exécution pour les instances de petites tailles sont quasiment nuls et rendent la distance d'édition ainsi qu'un alignement optimal. Il est ainsi possible de tester plus d'instances grâce à une meilleure consommation mémoire ainsi qu'un temps de calcul plus rapide. On remarque bien sur le graphe suivant la différence entre la complexité exponentielle du premier algorithme et la complexité polynomiale de *PROG_DYN*.



3.3 Amélioration de la complexité spatiale du calcul de la distance

Question 19

Lors du calcul de chacune des cases du tableau T (lors de la résolution de chacun des sous-problèmes formant le problème étudié), seules les cases du tableau de la ligne supérieure et de la ligne courante sont nécessaires. A chaque étape, les indices utiles au calcul sont i , $i - 1$, j ainsi que $j - 1$. Seuls les sous-problèmes les plus proches (l'alignement optimal des préfixes considérés) sont nécessaires. Il est donc possible d'arriver à calculer la distance d'édition d'un alignement optimal en ayant recours à beaucoup moins de mémoire que précédemment.

Question 20

Algorithm 3: DIST_2

Data: x et y deux mots tels que $(x, y) \in \Sigma^* \times \Sigma^*$

Réultat: La distance d'édition entre ces deux mots tout en remplissant le tableau T de dimension $2 \times m$

Initialisation du tableau T de taille $2 \times m$;

$k = |x|$;

$cpt = 0$;

$i = 1$;

for $j \in [1 \dots m]$ **do**

| $T[0][j] = j \times 2$;

end

$T[1][0] = 2$;

while $cpt < k$ **do**

for $j \in [1 \dots m]$ **do**

| $T[i][j] =$

$\min(T[(i-1)\%2][j] + c_{del}, T[i][j-1] + c_{ins}, T[(i-1)\%2][j-1] + c_{sub}(x_{cpt}, y_{j-1}))$;

end

$i = (i+1)\%2$;

$T[i][0] = T[(i-1)\%2][0] + 2$;

$cpt = cpt + 1$;

end

if $k \% 2 = 0$ **then**

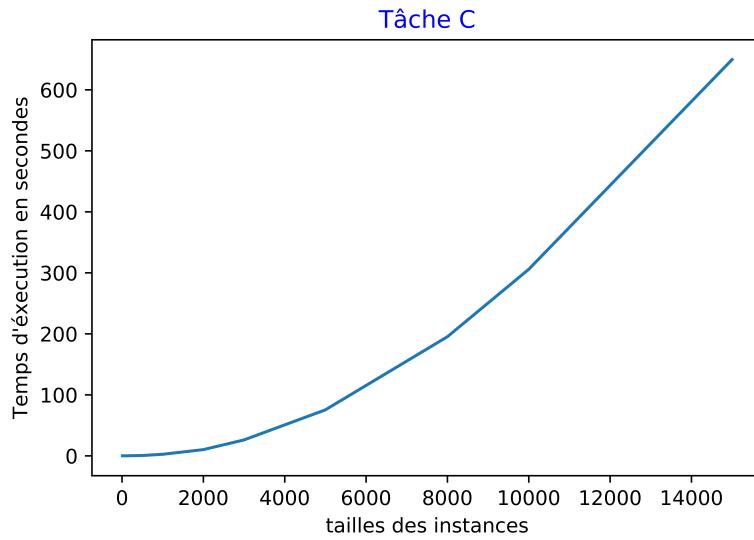
| Return $T[0][m]$

end

Return $T[1][m]$

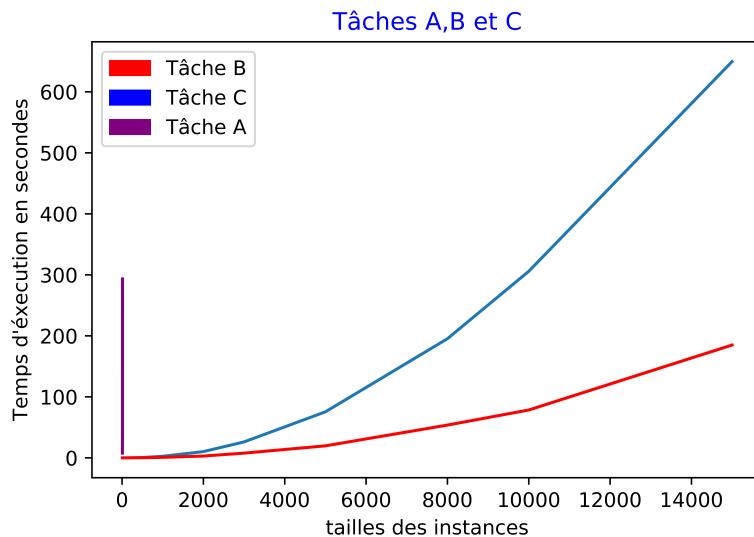
Tâche C

Représentation du temps CPU de l'exécution de DIST_2 en fonction des instances testées (pour des instances nécessitant moins de 10 minutes chacune) :



On observe que la courbe correspond bien à la complexité théorique. En effet, la complexité temporelle ne change pas par rapport à l'ancien algorithme car nous faisons toujours nm calculs. On a donc bien une courbe correspondant à une complexité polynomiale. Cependant, les calculs supplémentaires induits par les calculs d'indices et les modulus, augmentent de manière sensible le temps d'exécution sans toutefois changer le type de complexité.

On peut comparer les résultats de *DIST_2* avec ceux de *DIST_1* :



On voit bien l'augmentation du temps de calcul induite par les calculs d'indices entre la tâche B et la tâche C. Une instance de taille 15000 est exécutée en environs 200 secondes pour la tâche B tandis qu'il faut un peu plus de 600 secondes pour la tâche C. L'évolution en fonction de la taille d'instance reste cependant similaire et est en accord avec celle d'une fonction polynomiale pour les deux algorithmes. On peut alors estimer la quantité de mémoire utilisée par *DIST_2* pour une instance de très grande taille :

Pour l'instance *Inst0010000_50.adn*, sachant qu'un caractère est codé sur 8 bits et que $|x| =$

10000 et $|y| = 8882$, on peut calculer que l'on a :

$2 \times 8882 \times 8 = 142\,112$ bits consommés. Ce qui représente une différence importante avec les 715 000 kbits de la tâche B. *DIST_2* a bien une complexité spatiale linéaire en $\Theta(m)$ bien meilleure que la complexité spatiale polynomiale en $\Theta(nm)$ de *DIST_1* mais on a en contre partie, une complexité temporelle plus importante.

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

Question 21

Algorithm 4: mot_gaps

Data: $k \in \mathbb{N}$

Result: Le mot constitué de k gaps

Return $k \times -$;

Question 22

Algorithm 5: align_lettres_mot

Data: x un mot de longueur 1 et y un mot non vide de longueur quelconque

Result: un meilleur alignement parmi ceux possibles pour (x, y)

$i = 0$;

while $i < |y|$ **do**

if $y_i = x$ **then**
 | Return (mot_gaps(i) + x + mot_gaps($|y| - i - 1$), y);

end

 | $i = i + 1$;

end

$i = 0$;

while $i < |y|$ **do**

if $c_{sub}(x, y_i) = 3$ **then**
 | Return (mot_gaps(i) + substitution2(x, y_i) + mot_gaps($|y| - i - 1$), y);

end

 | $i = i + 1$;

end

if $i = |y|$ **then**

 | Return (mot_gaps($i - 1$) + substitution2(x, y_{i-1}), y);

end

substitution2(a, b) étant une fonction qui renvoie le caractère a à chaque appel pour réaliser la substitution de b par a .

Question 23

Un alignement optimal (\bar{s}, \bar{t}) de (x^1, y^1) est :

$$x^1 = BAL \text{ et } y^1 = RO$$

$\bar{s} :$	B	A	L	
	$\bar{t} :$	B	A	-

Avec un coût de 13.

Un alignement optimal (\bar{u}, \bar{v}) de (x^2, y^2) est :

$$x^2 = LON \text{ et } y^2 = ND$$

$\bar{u} :$	L	O	N	-	
	$\bar{v} :$	-	-	N	D

Avec un coût de 9.

$(\bar{s} \cdot \bar{u}, \bar{t} \cdot \bar{v})$ n'est pas un alignement optimal de (x, y) car on obtiendrait :

$$x = BALLON \text{ et } y = ROND$$

$\bar{u} :$	B	A	L	L	O	N	-	
	$\bar{v} :$	B	A	-	-	-	N	D

Avec un coût total de 22.

Alors qu'il est possible d'obtenir :

$\bar{u} :$	B	A	L	L	O	N	-	
	$\bar{v} :$	B	-	-	-	O	N	D

Avec un coût total de $17 < 20$. L'alignement précédent n'est donc pas optimal.

Question 24

Algorithm 6: SOL_2

Data: x et y deux mots.

u et v deux paramètres servant à stocker les alignements optimaux temporaires et qui contiendront en fin d'exécution les alignements optimaux de x et y

Result: (u, v) l'alignement optimal calculé pour (x, y)

$n = |x|$;

$m = |y|$;

$i = \lfloor \frac{n}{2} \rfloor$;

if $m = 0$ et $n \neq 0$ **then**

 ajouter x à u ;

 ajouter n gaps à v ;

else if $n = 0$ et $m \neq 0$ **then**

 ajouter m gaps à u ;

 ajouter y à v ;

else if $n = 1$ et $m \geq 1$ **then**

 ajouter à u le premier élément du tuple résultat de $\text{align_lettre_mot}(x, y)$;

 ajouter à v le second élément du tuple résultat de $\text{align_lettre_mot}(x, y)$;

else if $n > 1$ et $m \geq 1$ **then**

 j=coupure(x, y) ;

 SOL_2($x[:i], y[:j], u, v$) ;

 SOL_2($x[i:], y[j:], u, v$) ;

Return (u, v) ;

Les sous-mots sont indexés comme des sous-tableaux en python au sein des appels récursifs à SOL_2 .

Question 25

Algorithm 7: Coupure

Data: x et y deux mots.

Result: l'indice j^* permettant de réaliser la coupure de (x, y) sachant que $i^* = \frac{|x|}{2}$

Initialisation de deux tableaux T et I de dimensions $2 \times m$;

```

 $k = |x|$  ;
 $coup = \lfloor \frac{k}{2} \rfloor$  ;
 $cpt = 0$  ;
 $i = 1$  ;
 $i2 = 1$  ;
for  $j \in [1 \dots m + 1]$  do
     $| T[0][j] = j \times 2$  ;
end
 $T[i][0] = 2$  ;
for  $j \in [0 \dots m + 1]$  do
     $| I[0][j] = j$  ;
     $| I[1][j] = j$  ;
end
while  $cpt < k$  do
    for  $j \in [1 \dots m + 1]$  do
         $| T[i][j] =$ 
             $\min(T[(i - 1)\%2][j] + c_{del}, T[i][j - 1] + c_{ins}, T[(i - 1)\%2][j - 1] + c_{sub}(x_{cpt}, y_j))$  ;
        if  $cpt \geq coup$  then
             $| \text{if } T[i][j] = T[(i - 1)\%2][j] + c_{del} \text{ then}$ 
                 $| | I[i2][j] = I[(i2 - 1)\%2][j]$  ;
             $| \text{else if } T[i][j] = T[i][j - 1] + c_{ins} \text{ then}$ 
                 $| | I[i2][j] = I[i2][j - 1]$  ;
             $| \text{else if } T[i][j] = T[(i - 1)\%2][j - 1] + c_{sub}(x_{cpt}, y_{j-1}) \text{ then}$ 
                 $| | I[i2][j] = I[(i2 - 1)\%2][j - 1]$  ;
        end
        if  $cpt \geq coup$  then
             $| i2 = (i2 + 1)\%2$  ;
         $i = (i + 1)\%2$  ;
         $T[i][0] = T[(i - 1)\%2][0] + 2$  ;
         $cpt = cpt + 1$  ;
end
if  $(k - coup)\%2 = 0$  then
     $| \text{Return } I[0][|y|]$  ;
Return  $I[1][|y|]$  ;

```

Question 26

Lors de l'exécution de l'algorithme, on a recours aux lignes i et $i - 1$ des Tableaux T et I (le tableau I n'étant rempli qu'à partir de la moitié du tableau T) , on peut affirmer que la complexité spatiale de *coupure* est donc en :

$$\Theta(4m) \text{ soit une complexité de } \Theta(m)$$

Question 27

Dans le pire des cas, si $n = m$ et si l'on a $n - 1$ coupures pour le couple de mots considéré, on sera dans le cas où l'arbre des appels récursifs est complet. Sachant qu'il s'agit d'un arbre binaire, il sera donc de hauteur $\log_2(n)$ et le nombre total d'appels récursifs sera de $2^{\log_2(n)+1} - 1 = 2n - 1$. A chaque appel récursif engendrant d'autres appels récursifs (donc non résolus par un cas de base), la fonction *coupure* utilisée a une complexité spatiale en $\Theta(m)$. Une fois l'indice de la coupure j^* calculé, la mémoire utilisée par *coupure* est libérée. Ainsi, *SOL_2* a une consommation mémoire en $\Theta(\max(n + m))$, avec toutefois une consommation inférieure à *DIST_2* car seule les listes servant à retourner les résultats sont gardées en mémoire du début à la fin de l'exécution.

Question 28

coupure réalise toujours $n \times m$ calculs pour déterminer la distance d'édition ainsi que le tableau I donnant les coupures du couple de mot étudié. A partir de $i^* = \frac{|x|}{2}$, le tableau I est aussi mis à jour, ce qui implique des comparaisons supplémentaires. Cependant on peut considérer que ces calculs et ces mises à jour, se font en temps constant et *coupure* a donc une complexité temporelle en $\Theta(nm)$.

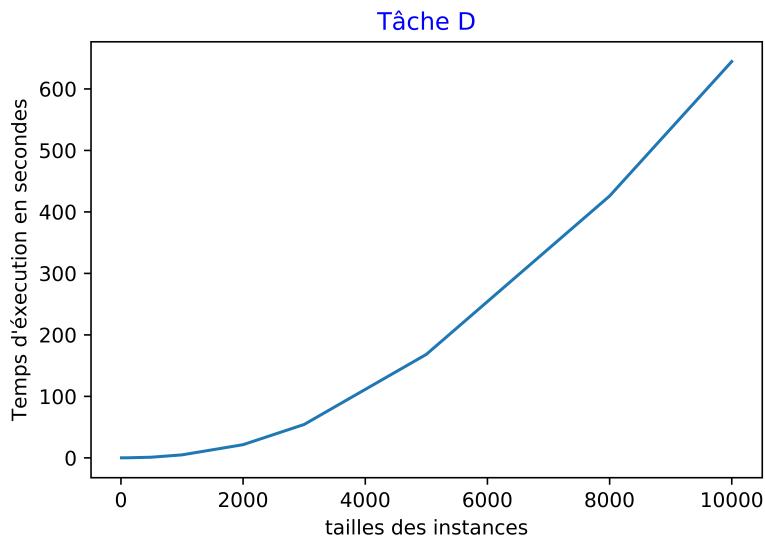
Tâche D

Représentation du temps CPU de l'exécution de *SOL_2* en fonction des instances testées (pour des instances nécessitant moins de 10 minutes chacune) :

On observe sur cette courbe, une évolution comparable à celle d'une fonction polynomiale. La complexité théorique en $\Theta(nm)$ est donc en accord avec la complexité mesurée. En effet, pour $n > 1$ et $m \geq 1$, *SOL_2* fait appel à *coupure* qui est en $\Theta(nm)$ et entraîne deux appels récursifs. Si l'on suppose que *coupure* coupe chaque sous-mot en deux sous-mots de longueurs approximativement égales (ce qui peut amener à une perte de généralité, mais qui est vrai pour certaines instances que nous avons testé) et sachant que *align_letters_mots* a une complexité en $\Theta(n + m) \sim \Theta(n)$ et que $m \approx \frac{n}{2}$, on peut essayer d'écrire une équation de récurrence :

$$T(n) = 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \Theta(nm) \sim 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \Theta(n^2)$$

En appliquant le théorème maître, il est possible de déduire que *SOL_2* a une complexité en $\Theta(n^2)$.

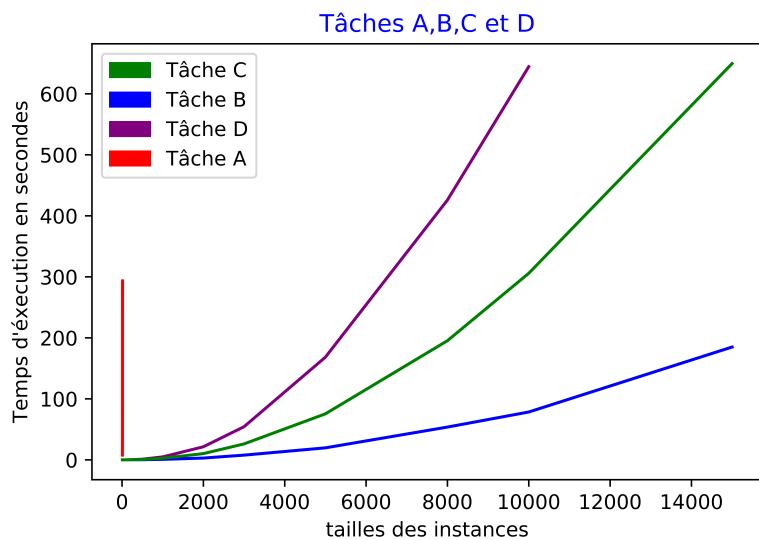


On peut estimer la consommation mémoire de *SOL_2*, si l'on considère le fait que *coupure* libère la mémoire à chaque fois qu'elle est appelée, que chaque caractère est codé sur 8 bits et que l'on stocke les résultats dans deux listes dont la taille maximale est de $n + m$, on peut affirmer que l'on a : $8 \times 2 \times (n + m)$ bits consommés.

Pour l'instance *Inst_0001000_23.adn* on peut estimer que l'on aura :

$8 \times 2 \times (1000 + 887) = 30\,196$ bits consommés. Notre mesure expérimentale a donné 30.9 Kbits (pour chacune des mesures, un espace mémoire supplémentaire doit être considéré en raison du fait que l'exécution a lieu au sein d'un environnement d'exécution consommant lui aussi de la mémoire d'où les bits en surplus dans les mesures). On en déduit que la complexité spatiale est bien en $\Theta(n + m)$ ce qui est meilleur que les algorithmes des tâches B et C.

4.1 Question 29



Lors de la réalisation des tâches B, C et D, nous avons à chaque fois réduit la consommation mémoire nécessaire à la résolution du problème d'alignement des séquences. Nous avons remarqué que l'algorithme de la tâche C consommait effectivement moins de mémoire que celui de la tâche B mais entraînait un temps de calcul plus long. Il était cependant possible d'atteindre la même taille d'instance pour les 2 algorithmes .

Pour la tâche D, on remarque clairement sur le graphe comparatif ci-dessus que la complexité temporelle de D (bien qu'étant de même type) est bien supérieure à celle de B et C. Pour les tâches B et C, il nous a été possible de tester des instances ayant une taille maximale de 15 000 tandis que pour la tâche D, on dépasse les 10 minutes de temps d'exécution dès une instance de taille 10 000. Il est donc clair qu'en passant à une complexité spatiale moins importante pour la tâche D, nous avons perdu de manière importante en complexité temporelle avec un nombre maximal de 10 000 nucléotides pouvant être testés en un temps raisonnable. Cela peut s'expliquer par le fait que l'utilisation de la fonction coupure calcule plusieurs fois le tableau T donnant les distances d'éditions pour les sous-mots considérés. A chaque coupure, on calcule le tableau T pour le sous-mot que l'on veut couper de manière à obtenir un alignement optimal et il apparaît clairement que la tâche D est plus lente par rapport à la tâche C qui elle même est plus lente que la tâche B. Il est donc possible de résoudre ce problème avec une complexité spatiale linéaire mais au détriment de la complexité temporelle.