Projet de LU3IN003
Alignement de séquences

Zein Sakkour
Philemon Wehbe

Table des matières

1 Choix d'implémentation

Structures

1. La structure `duo_chaine` : on l'utilise pour stocker une instance génome avec 2 chaînes x, y et leurs tailles respectives n et m.
2. La structure Align : on l'utilise pour stocker un alignement de 2 chaînes x, y, la taille (`size`) de l'alignement et un itérateur (`iter`) qui pointe vers la fin de la chaîne x et y pour pouvoir les remplir de la fin au début.
3. la structure txtFile (liste chaînée) : on l'utilise pour pouvoir stocker les noms de fichiers dans genom par la chaîne `Fname` (nom du fichier), `len` : la taille maximale entre x et y dans le fichier `Fname` ainsi qu'un pointeur vers le prochain fichier (`next`)

Tests

En se plaçant dans un terminal dans le présent fichier :

— La commande `make` compilera l'ensemble du programme et retournera un ensemble d'exécutables, notamment un nommé `myProg`, qui permet de tester plusieurs fonctions. La commande `make clean` supprimera tous les exécutables afin de pouvoir recompiler si besoin.

— Pour tester les fonctions sur toutes les instances de génome il suffit de passer au répertoire Testing (`cd Testing`) et lancer la commande `./testingDP_ALL_INST` qui calculera grâce aux deux fonctions `SOL_1` et `SOL_2` (et affichera) le coût de l'alignement optimal ainsi que la distance d'édition pour toutes les instances de génome disponibles, tout en vérifiant que ces deux valeurs sont bien égales pour toutes les instances, grâce à un appel à `assert`.
Un "sanity check" sera aussi effectué : si, en supprimant les $-$ du $\overline{x}$ ou $\overline{y}$ obtenu, on obtient les chaînes de départ x ou y, la fonction `sanity_check()` renvoie un message positif.
De même, un exécutable `./testingNM_ALL_INST` effectue les mêmes tests mais avec les fonctions "naïves".

— Pour vérifier qu'il n'y a pas de fuite mémoire (*spoiler :* il n'y en a pas...), un fichier `valgrind.rpt` est mis à disposition et contiendra les messages obtenus lors d'un appel à valgrind. Pour vérifier, lancer cette commande sur le terminal :

```
valgrind -leak-check=yes -log-file=valgrind.rpt -s -leak-check=full
-show-leak-kinds=all ./testingDP_ALL_INST
```

Outils

— gnuplot : nous a aidé à obtenir les graphes présents dans ce rapport. Les fichiers `.txt` utilisés pour appeler gnuplot sont disponibles dans le répertoire DATA.
— valgrind : gestion de mémoire. De plus, valgrind nous a permis de calculer la mémoire consommée par nos fonctions en testant sur plusieurs instances individuellement.
— makefile : pour automatiser la compilation.
— langage utilisé : C.

N.B.

Nos tests ont été réalisés sur une puce M1 Pro, donc pourraient être plus rapides que sur d'autres machines, d'où des temps CPU légèrement plus faibles.
Sinon, petite erreur au niveau de la table des matières (elle n'est pas mise à jour automatiquement contrairement à l'habitude, bug qu'on n'a tenté de résoudre en vain).

2 Le problème d'alignement de séquences

2.1 Alignement de deux mots

Question 1

Soient $(\overline{x}, \overline{y})$ et $(\overline{u}, \overline{v})$ des alignements de (x, y) et (u, v) respectivement. Montrons que $(\overline{x}\cdot\overline{u}, \overline{y}\cdot\overline{v})$ est un alignement de $(x\cdot u, y\cdot v)$.

1. Montrons que $\pi(\overline{x}\cdot\overline{u}) = x\cdot u$
   En effet, $\pi(x\cdot u)$ est le mot obtenu en retirant les – de $\overline{x}\cdot\overline{u}$. Il contient donc tous les caractères de $\overline{x}$ différents de –, suivis des caractères de $\overline{u}$ différents de –. Alors $\pi(\overline{x}\cdot\overline{u})$ contient les caractères de x suivis des caractères de u, comme $\pi(\overline{x}) = x$ et $\pi(\overline{u}) = u$. D'où $\pi(\overline{x}\cdot\overline{u}) = x\cdot u$.
2. De même, $\pi(\overline{y}\cdot\overline{v}) = y\cdot v$.

3. Montrons que $|\overline{x}\cdot\overline{u}| = |\overline{y}\cdot\overline{v}|$
   On a
   $$\begin{aligned} |\overline{x}\cdot\overline{u}| &= |\overline{x}| + |\overline{u}| && \textit{par concaténation} \\ &= |\overline{y}| + |\overline{v}| && \textit{car } (\overline{x},\overline{y}) \textit{ et } (\overline{u},\overline{v}) \textit{ alignements} \\ &= |\overline{y}\cdot\overline{v}| && \textit{par concaténation} \end{aligned}$$
   CQFD.
4. Montrons que $\forall i \in \{1,\dots,|\overline{x}\cdot\overline{u}|\}, (\overline{x}\cdot\overline{u})_i \neq -$ ou $(\overline{y}\cdot\overline{v})_i \neq -$
   Soit $i \in \{1,\dots,|\overline{x}\cdot\overline{u}|\}$. Deux cas se présentent :

   — Si $i > |(\overline{x},\overline{y})|, (\overline{x}\cdot\overline{u})_i \in \overline{x}$, donc $(\overline{y}\cdot\overline{v})_i \in \overline{y}$ car $|\overline{x}| = |\overline{y}|$. Comme $(\overline{x},\overline{y})$ est un alignement de (x,y), alors $(\overline{x}\cdot\overline{u})_i \neq -$ ou $(\overline{y}\cdot\overline{v})_i \neq -$.
   — Sinon, $i \geq |(\overline{x},\overline{y})|$, donc $(\overline{x}\cdot\overline{u})_i \in \overline{u}$, dans ce cas $(\overline{y}\cdot\overline{v})_i \in \overline{v}$ car $|\overline{x}| = |\overline{y}|$ (un caractère de $\overline{u}$ ne peut pas avoir le même indice qu'un caractère de $\overline{y}$). D'où $(\overline{x}\cdot\overline{u})_i \neq -$ ou $(\overline{y}\cdot\overline{v})_i \neq -$, $(\overline{u},\overline{v})$ étant un alignement de (u,v).

   Par suite, $\forall i \in \{1,\dots,|\overline{x}\cdot\overline{u}|\}, (\overline{x}\cdot\overline{u})_i \neq -$ ou $(\overline{y}\cdot\overline{v})_i \neq -$

Les 4 propriétés précédentes étant vérifiées, on a bien que $(\overline{x}\cdot\overline{u},\overline{y}\cdot\overline{v})$ est un alignement de $(x\cdot u, y\cdot v)$.

Question 2

Soit x un mot de longueur n, et soit y un mot de longueur m. Soit $(\overline{x},\overline{y})$ un alignement de (x,y). Montrons que sa longueur maximale est de $n+m$.
D'abord, montrons que cet alignement existe. En effet, ce dernier peut être construit de telle sorte que pour tout i, si $\overline{x}_i \neq -$, *i.e.* $\overline{x}_i \in x$, alors $\overline{y}_i = -$, et inversement.
Par exemple, pour $\Sigma = \{A,T,G,C\}$, si $x = ATTGTAC$ $(n = 7)$ et $y = ATCTTA$ $(m = 6)$, un alignement de longueur $n + m = 7 + 6 = 13$ est le suivant :

$$\begin{aligned} \overline{x} &= A - TT - G - T - A - C- \\ \overline{y} &= -A - -T - C - T - T - A \end{aligned}$$

Ensuite, montrons que cette longueur est bien maximale. Par l'absurde, supposons qu'il existe un alignement de (x,y) tel que $|(\overline{x},\overline{y})| = n + m + 1$
$\overline{x}$ est donc composé de $n+m+1$ éléments, dont n caractères de x, et $m+1$ –; de même pour $\overline{y}$, m caractères de y et $n+1$ –.
Comme $\overline{x}$ n'a pas autant de – que de caractères de $\overline{y}$ et inversement, il

existe donc un $i \in \{1, \ldots, n+m+1\}$ pour lequel $\overline{x}_i = \overline{y}_i = -$, donnant une contradiction. En effet, cet alignement ne respecte pas la 4^e propriété des alignements. De même, pour tout alignement de longueur supérieure à $n+m+1$, on trouve donc au moins un indice i pour lequel la 4^e propriété des alignements n'est pas vérifiée.
Ainsi, il ne peut y avoir d'alignement de (x, y) de longueur strictement supérieure à $n+m$.

3 Algorithmes pour l'alignement de séquences

3.1 Méthode naïve par énumération

Question 3

Soit x un mot de taille n auquel on rajoute k gaps. On a donc un total de $n+k$ positions parmi lesquelles on choisit k pour placer des gaps. Le nombre possible de mots de n caractères et k gaps est le nombre de façons d'insérer k gaps pour obtenir un mot $\overline{x}$ tel que $|\overline{x}| = n+k$:

$$\binom{n+k}{k} = \binom{n+k}{n} = \frac{(n+k)!}{k!\,n!}$$

Question 4

Soient x un mot de taille n et y un mot de taille $m \leq n$. On cherche à trouver le nombre de gaps à ajouter à y si on en ajoute k à x, de sorte que les deux mots obtenus $\overline{x}$ et $\overline{y}$ ne contiennent pas de gap à une même position.
Si $k \geq m$, c'est impossible.
Sinon, si on insère k gaps à x de taille n, il faut en placer $n+k-m$ dans y de taille m. Il y a alors $\binom{n+k}{n+k-m}$ possibilités. Or, pour éviter d'avoir deux gaps à la même position, les k positions contenant un $-$ dans $\overline{x}$ sont interdites, et on doit donc placer les $n+k-m$ gaps dans $\overline{y}$ sur $n+k-k=n$ positions valables. Ainsi, pour tout $0 \leq k \leq m$, il existe $\binom{n+k}{k}\binom{n}{n+k-m}$ alignements possibles de longueur $n+k$.
Le nombre total d'alignements $(\overline{x}, \overline{y})$ est donc de :

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

Par exemple, pour $n = |\overline{x}| = 15$ et $m = |\overline{y}| = 10$, le nombre total d'alignements $(\overline{x}, \overline{y})$ est :

$$\sum_{k=0}^{10} \binom{15+k}{k} \binom{15}{5+k} = 298199265$$

Question 5

Un algorithme naïf listerait tous les alignements possibles avant de retourner celui de distance d'édition minimale. Mesurer le coût de chaque alignement se fait en $O(n)$.
En supposant que le minimum est mis à jour au fur et à mesure du parcours (donc en $O(1)$), la complexité est donc en

$$O\left(n \times \sum_{k=0}^{m} \binom{n+k}{k} \binom{n}{n+k-m}\right) \equiv O\left(n \times \sum_{k=0}^{m} \frac{(n+k)!}{k!\,(n+k-m)!\,(m-k)!}\right) \subset O(n(n+m)!)$$

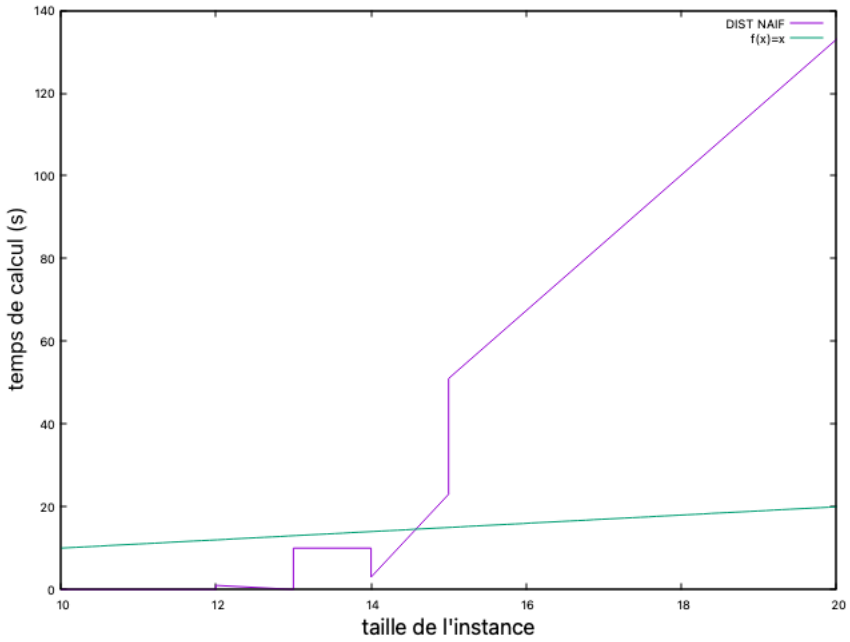
En effet, $(n+m)!$ majore le produit des coefficients binomiaux. C'est donc une complexité factorielle, donc exponentielle.

Question 6

Soit x et y de tailles respectives n et m. Un algorithme naïf va énumérer les alignements possibles de (x, y). À chaque alignement obtenu, il comparera son coût à celui de coût minimal sauvegardé et remplacer ce dernier s'il est de coût inférieur. Comme le coût est stocké en $O(1)$, et que l'on ne stocke que l'alignement de coût minimal (de longueur maximale $n+m$), la complexité spatiale de l'algorithme naïf est donc en $O(n+m)$.

3.1.1 Tâche A

En moins d'une minute, on peut tester jusqu'à l'instance de taille 15 en utilisant la méthode naïve. L'instance de taille 20 dépasse la minute de temps de calcul voulue.



Tâche A

Pour la méthode naïve, pour une instance de taille 10, on a besoin de 6.5048828125 octets, donc environ 0.006 Ko. On remarque que pour les instances durant moins d'une minute, la consommation est presque la même (pour une instance de taille 6, 6.654 octets, de même 6.5068359375 et 6.5107421875 octets pour des tailles de 12 et 13 respectivement).

4 Programmation dynamique

4.1 Calcul de la distance d'édition par programmation dynamique

Question 7

Soit $(\overline{u}, \overline{v})$ un alignement de (u, v) de longueur l.
— Si $\overline{u}_l = -$, alors $\overline{v}_l = y_j$ comme $\overline{v}_l \neq -$.
— Si $\overline{v}_l = -$, alors $\overline{u}_l = x_i$ comme $\overline{u}_l \neq -$.
— 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

$$C(\overline{u}, \overline{v}) = C(\overline{u}_{[1...l]}, \overline{v}_{[1...l]}) + c(\overline{u}_l, \overline{v}_l)$$

Reprenons les 3 cas précédents :

— Si $\overline{u}_l = -$, alors $C(\overline{u}, \overline{v}) = C(\overline{u}_{[1...l]}, \overline{v}_{[1...l]}) + c_{ins}$
— Si $\overline{v}_l = -$, alors $C(\overline{u}, \overline{v}) = C(\overline{u}_{[1...l]}, \overline{v}_{[1...l]}) + c_{del}$
— Si $\overline{u}_l \neq -$ et $\overline{v}_l \neq -$, alors $C(\overline{u}, \overline{v}) = C(\overline{u}_{[1...l]}, \overline{v}_{[1...l]}) + c_{sub}(x_i, y_j)$

Question 9

$D(i, j)$ s'obtient récursivement à partir des états précédents $D(i, j-1)$, $D(i-1, j)$ et $D(i-1, j-1)$ par une insertion, suppression et substitution respectivement. Comme $D(i, j)$ est minimal par définition, on a alors :

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

Question 10

$D(0, 0)$ est la distance d'édition entre les deux sous-mots vides (donc identiques). Aucune opération n'est nécessaire et le coût est donc nul par définition, et la distance d'édition aussi : $D(0, 0) = 0$.

Question 11

Pour $i = 0$, on ne réalise que des insertions de caractères de y :

$$D(0, j) = j \times c_{ins}$$

Pour $j = 0$, on ne réalise que des suppressions d'éléments de x :

$$D(i, 0) = i \times c_{del}$$

Question 12

```
Dist_1(x,y,T){
//INPUT x,y,T[0..|x |]*[0...| y |]
n=|x| ;
m=|y|;
T[0][0] = 0;
for(i=1...n){
    T[i][0] = T[i-1][0] + C_DEL;
}
for(j=1...m){
    T[0][j] = T[0][j-1] + C_INS;
}
for(i=1...n){
    for(j=1...m){
        INS = T[i][j-1] + C_INS;
        DEL = T[i-1][j] + C_DEL;
        SUB = T[i-1][j-1] + cout_substitution(x[i-1] , y[j-1]);
        T[i][j] = min3(INS , DEL , SUB);
    }
}
return T[n][m] ;
}

```

Question 13

Nous devons stocker deux mots x et y de taille n et m respectivement. Nous avons donc besoin d'un tableau T de $n \times m$ entiers, d'où une complexité spatiale de $\Theta(nm)$.

Question 14

On suppose que les opérations (élémentaires) se font en $\mathcal{O}(1)$. Nous avons deux boucles de n et m itérations respectivement, ainsi que deux boucles imbriquées de $n \times m$ itérations. La complexité cumulée est donc de $\mathcal{O}(n + m + nm) \equiv \mathcal{O}(nm)$.

4.2 Calcul d'un alignement optimal par programmation dynamique

Question 15

Montrons que :

— Si $j > 0$ et $D(i,j) = D(i,j-1) + c_{ins}$, alors $\forall(\overline{s},\overline{t}) \in Al^*(i,j-1)$, $(\overline{s}\cdot -, \overline{t}\cdot y_j) \in Al^*(i,j)$.
En effet, pour passer de $D(i,j-1)$ à $D(i,j)$, on s'intéresse à un alignement de coût minimal, donc de distance d'édition $D(i,j) = D(i,j-1) + c_{ins}$ par hypothèse. Cet alignement est fait en insérant un élément y_j supplémentaire à un alignement $(\overline{s},\overline{t}) \in Al^*(i,j-1)$. On obtient alors un alignement $(\overline{s}\cdot -, \overline{t}\cdot y_j) \in Al^*(i,j)$ de coût minimal $D(i,j)$.

— Si $i > 0$ et $D(i,j) = D(i-1,j) + c_{del}$, alors $\forall(\overline{s},\overline{t}) \in Al^*(i-1,j)$, $(\overline{s}\cdot x_i, \overline{t}\cdot -) \in Al^*(i,j)$.
En effet, pour passer de $D(i,j-1)$ à $D(i,j)$, on s'intéresse à un alignement de coût minimal, donc de distance d'édition $D(i,j) = D(i-1,j) + c_{del}$ par hypothèse. Cet alignement est fait en insérant un élément x_i supplémentaire à un alignement $(\overline{s},\overline{t}) \in Al^*(i-1,j)$. On obtient alors un alignement $(\overline{s}\cdot x_i, \overline{t}\cdot -) \in Al^*(i,j)$ de coût minimal $D(i,j)$.

— Si $D(i,j) = D(i-1,j-1) + c_{sub}(x_i,y_j)$, alors $\forall(\overline{s},\overline{t}) \in Al^*(i-1,j-1)$, $(\overline{s}\cdot x_i, \overline{t}\cdot y_j) \in Al^*(i,j)$.
En effet, pour passer de $D(i-1,j-1)$ à $D(i,j)$, on s'intéresse à un alignement de coût minimal, donc de distance d'édition $D(i,j) = D(i-1,j-1) + c_{sub}(x_i,y_j)$ par hypothèse. Cet alignement est fait en échangeant une lettre x_i de x et une lettre y_j de y dans un alignement $(\overline{s},\overline{t}) \in Al^*(i-1,j-1)$. On obtient alors un alignement $(\overline{s}\cdot x_i, \overline{t}\cdot y_j) \in Al^*(i,j)$ de coût minimal $D(i,j)$.

Question 16

```
SOL_1(x,y,T){
//INPUT :x,y,T[0..|x |]*[0...| y |]
    i=|x|
    j=|y|
    Algn res // Alignement
    while ((i > 0) && (j > 0)) {
        if ( (T[i][j] == T[i][j-1] + C_INS)){
            res.x= '-'+res.x
            res.y= y[j-1]+res.y
            j--;
        }
        else if ( (T[i][j] == T[i-1][j] + C_DEL)) {
            res.x= x[i-1]+res.x
            res.y= '-'+res.y
            i--;
        }
        cout_sub=cout_substitution(x[i-1] , y[j-1])
        else if ( (T[i][j] == T[i-1][j-1] + cout_sub)){
            res.x= x[i-1]+res.x
            res.y= y[j-1]+res.y
            i--;
            j--;
        }
    }
    while (i > 0) {
        res.x= x[i-1]+res.x
            res.y= '-'+res.y
            i--;
    }
    while (j > 0) {
        res.x= '-'+res.x
            res.y= y[j-1]+res.y
            j--;
    }
    return res;
}

```

Question 17

`SOL_1` est de complexité temporelle $\mathcal{O}(n+m)$. Les opérations à l'intérieur de la boucle sont en $\mathcal{O}(1)$ donc en combinant les deux algorithmes, la

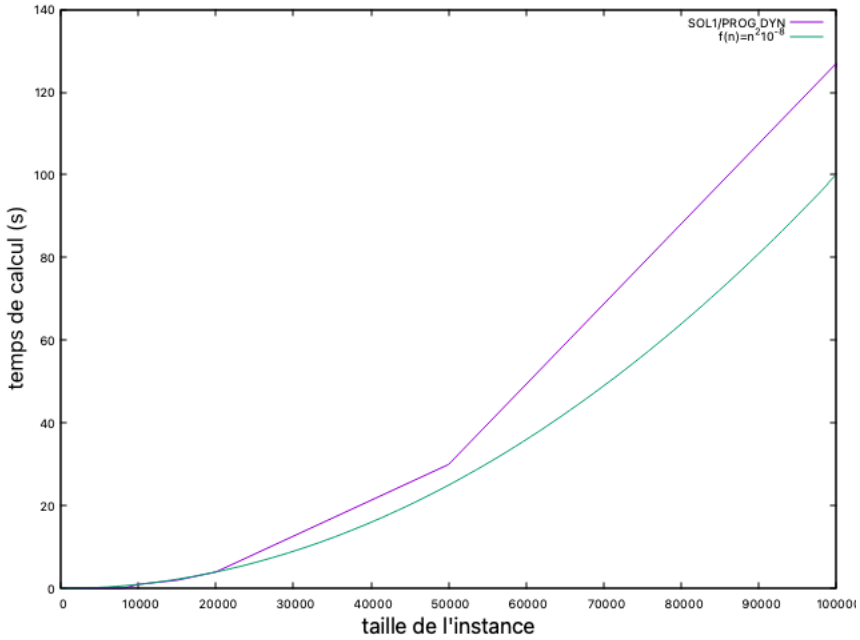
complexité est de $\mathcal{O}(nm + n + m) \equiv \mathcal{O}(nm)$.

Question 18

La complexité spatiale de `SOL_1` est en $\mathcal{O}(n + m)$ car l'alignement est de longueur maximale $n + m$. Le reste des variables dans `SOL_1` sont de taille constante. D'après la question 13, la complexité spatiale de `DIST_1` est en $\mathcal{O}(nm)$. La complexité spatiale totale est donc en $\mathcal{O}(n + m + nm) \equiv \mathcal{O}(nm)$.

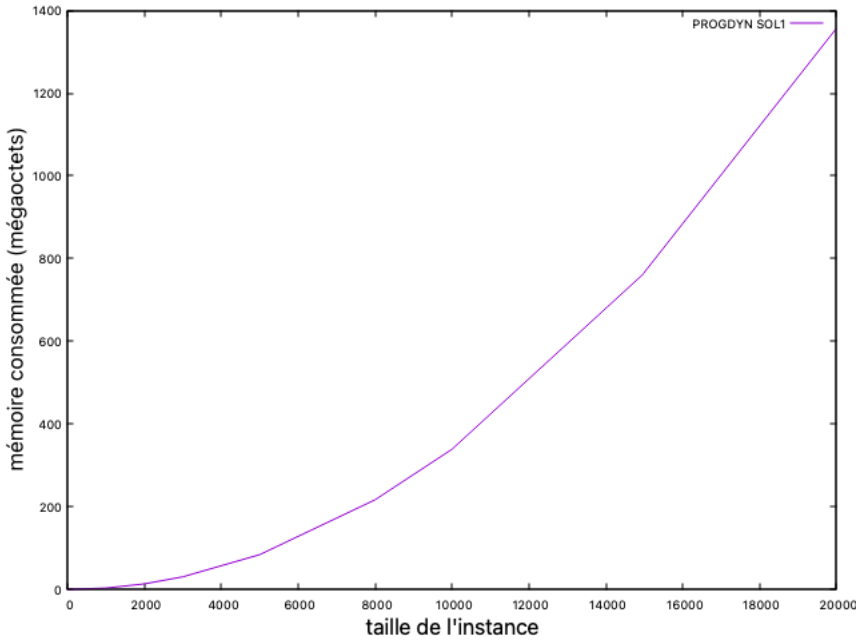
4.2.1 Tâche B

Nous avons testé `PROG_DYN` qui appelle `SOL_1` et `DIST_1` sur toutes les instances de génome.



Tâche B : temps CPU

Dans ce graphe, on compare le temps CPU, en fonction de la taille de l'instance, à un multiple de n^2 (ici 10^{-8}) qui correspond bien à une complexité temporelle de $\mathcal{O}(nm)$.



Tâche B : mémoire consommée

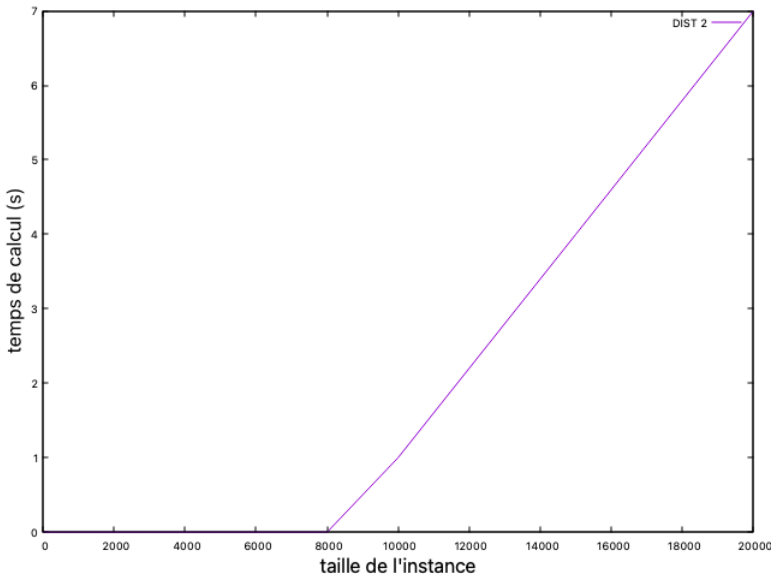
Le second graphe modélise la mémoire consommée en fonction de la taille de l'instance. Pour une instance de grande taille, on estime 1400 Mo de mémoire utilisée pour 20000 caractères.

4.3 Amélioration de la complexité spatiale du calcul de la distance

Question 19

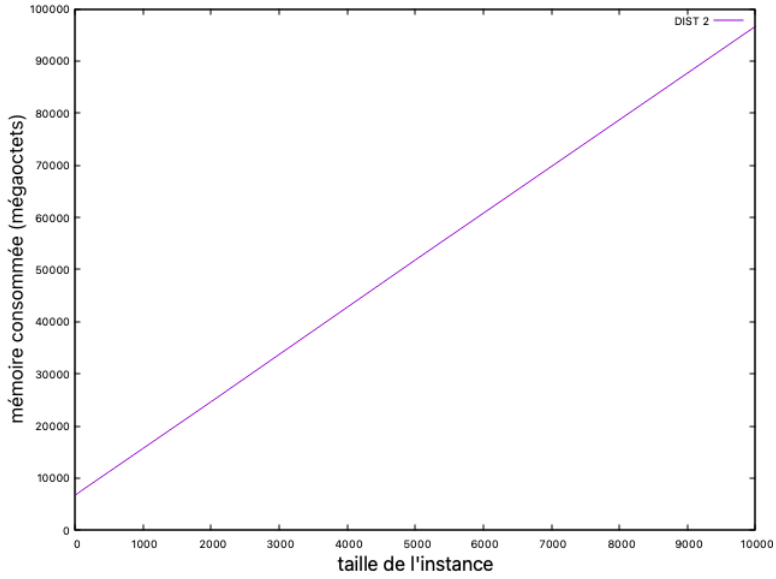
Pour remplir $T[i][j]$ dans `DIST_1`, on se sert uniquement des trois cases $T[i][j-1]$, $T[i-1][j]$ et $T[i-1][j-1]$ (d'après l'équation de la question 9) qui sont bien sur les lignes i ou $i-1$.

4.3.1 Tâche C

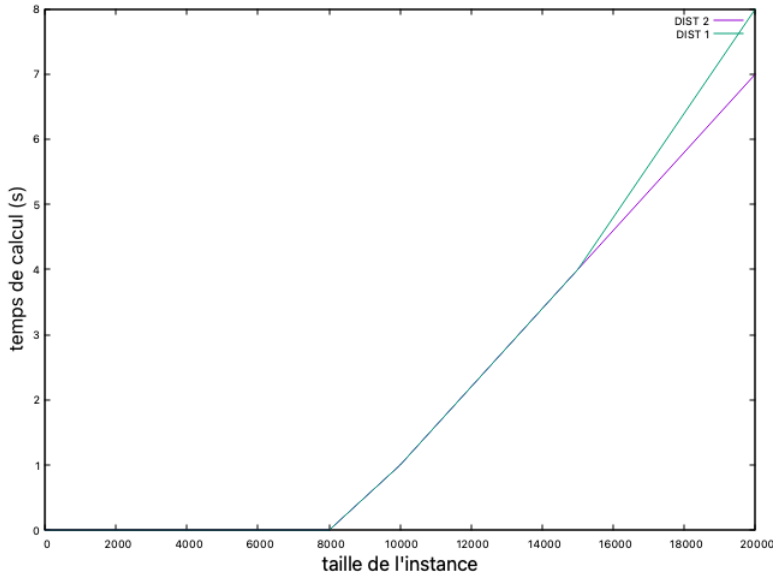


Temps CPU de `DIST_2`

Comme `DIST_2` est en $O(nm)$, on remarque que la courbe obtenue ne correspond pas à la complexité temporelle théorique ; elle est meilleure en pratique (à peu près $O(n)$ comme on voit une droite).



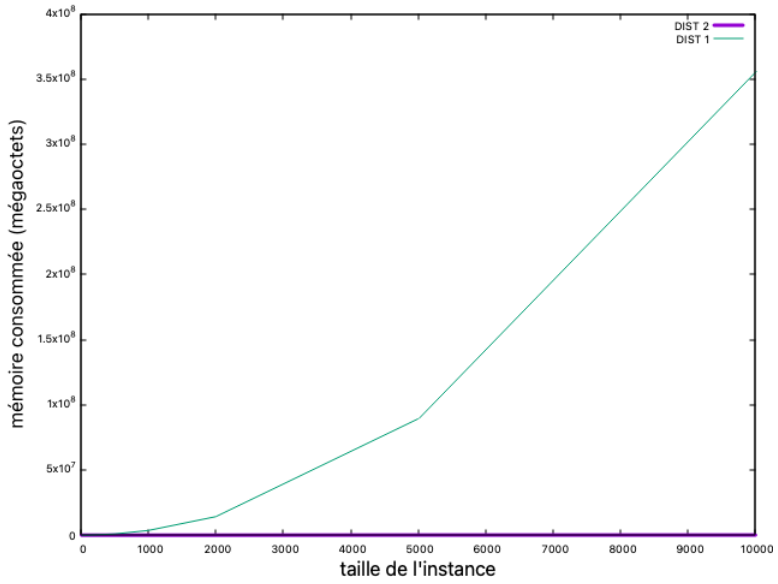
Mémoire utilisée par `DIST_2`



Comparaison des temps CPU de `DIST_1` et `DIST_2`

On observe que les courbes obtenues pour `DIST_1` et `DIST_2` coïncident, donc leurs complexités temporelles sont bien les mêmes, comme on peut le

voir théoriquement aussi.



Comparaison de la mémoire utilisée par `DIST_1` et `DIST_2`

Par contre, la consommation mémoire de `DIST_2` est négligeable par rapport à celle de `DIST_1` donc `DIST_2` est de meilleure complexité spatiale que `DIST_1`.

Question 20

```
Dist_2(x,y,n,m,DP){
//INPUT : x,y,n,m,DP[0..|x|]*[0...|y|]
    n = |x|
    m = |y|
    for(j=1...m){
        DP[0][j] = j* C_INS
    }
    for(i=1...n){
        DP[1][0] = DP[0][0] + C_DEL
        for(j=1...m){
            INS = DP[1][j-1] + C_INS
            DEL = DP[0][j] + C_DEL
            cout_sub=cout_substitution(x[i-1] , y[j-1])
            SUB = DP[0][j-1] +cout_sub
            DP[1][j] = min3(INS , DEL , SUB)
        }
        swap(D[0],D[1])
    }
    return DP[0][m]
}

```

Question 21

```
mot_gaps(k){
//INPUT : k
    res =[]
    for(i=0...k)
        res=res+'-'
    return res;
}


```

Question 22

```
align_lettre_mot(x,y,m){
//INPUT : x,y,m
//m=|y|
    i=0
    cout_i=INF
    cout_curr=0
    Algn res //Alignement final
    for (t=0...m){
        cout_curr=cout_substitution(x,y[t]);
        if(cout_curr<cout_i){
            i=t;
            cout_i=cout_substitution(x,y[i]);
        }
        if(cout_curr==0){
            i=t;
            cout_i=cout_substitution(x,y[i]);
            break;
        }
    }
    if(cout_i<C_DEL+C_INS){
        res.x=mot_gaps(m);
        res.x[i]=x;
        res.y=y

    }else{
        res.x=x+mot_gaps(m)
        res.y='-'+y

    }
    return res;

}

```

Question 23

1. $(\overline{s}, \overline{t}) = (BAL,\ RO-)$ est un alignement optimal de (x^1, y^1) de distance d'édition $d = 13$.
2. $(\overline{u}, \overline{v}) = (LON-,\ --ND)$ est un alignement optimal de (x^2, y^2) de distance d'édition $d = 9$.
3. $(\overline{s} \cdot \overline{u},\ \overline{t} \cdot \overline{v}) = (BALLON-,\ RO---ND)$ est de coût 22, mais

l'alignement $(BALLON-, R---OND)$ est de coût $17 < 22$. Alors, $(\overline{s}\cdot\overline{u},\ \overline{t}\cdot\overline{v})$ n'est pas un alignement optimal de (x, y).

Question 24

```
SOL_2(x,y,n,m,T,I){
// INPUT :x,y, n=|x|, m=|y| , T[0..|x ||*[0...| y|] ,I [0..| x ||*[0...| y|]
//on a besoin de T et I pour coupure pour eviter
// de reallouer a chaque appel recursif
    Algn res // Alignement final
    if (n == 0){
        res.x = mot_gaps(m)
        res.y = y
        return res
    }
    if (m == 0){
        res.x = x
        res.y=mot_gaps(n)
        return res;
    }
    if(n==1){
        res=align_lettre_mot(x,y,m);
        return res;
    }
    j = coupure(x,y,n,m,T,I);
    i = n/2;
    Algn algn_1 = SOL_2(x[0..i[,y[0...j[, i, j, T,I);
    Algn algn_2 = SOL_2(x[i..n[,y[j..m[,n-i,m-j,T,I);
    res = algn_1.algn_2 //concatenation
    return res
}

```

Question 25

```
coupure(x,y,n,m,T,I){
// INPUT :x,y, n=|x|, m=|y| , T[0..|x ||*[0...| y|] ,I [0..| x ||*[0...| y|]
    i* = n/2;
    for(j=1...m) {
        I[0][j]=j;
        T[0][j] = T[0][j-1] + C_INS;
    }
    for(i=1...n){
        T[1][0] = T[0][0] + C_DEL;
        I[1][0] = 0;
        if (i=iE+1){
            for(j=1...m) I[0][j]=j;
        }
        for(j=1...m){
            INS = T[1][j-1] + C_INS;
            DEL = T[0][j] + C_DEL;
            SUB = T[0][j-1] + cout_substitution(x[i-1] , y[j-1]);
            T[1][j] = min(INS , DEL , SUB);
            if (T[1][j]==SUB) I[1][j]=I[0][j-1];
            if (T[1][j]==DEL) I[1][j]=I[0][j];
            if (T[1][j]==INS) I[1][j]=I[1][j-1];
        }
        swap(T[0],T[1]);
        swap(I[0],I[1]) ;
    }
    return I[0][m];
}

```

Question 26

Dans `coupure`, deux tableaux T, I de taille $2m$ sont alloués. Le reste des variables est de taille constante. Donc la complexité spatiale de `coupure` est en $\mathcal{O}(m)$.

Question 27

`SOL_2` va allouer deux sous-alignements de taille en $\mathcal{O}(n+m)$ en mémoire chacun. La fonction `coupure` est en $\mathcal{O}(m)$. On major avec une relation de

récurrence sur T, la complexité spatiale totale :

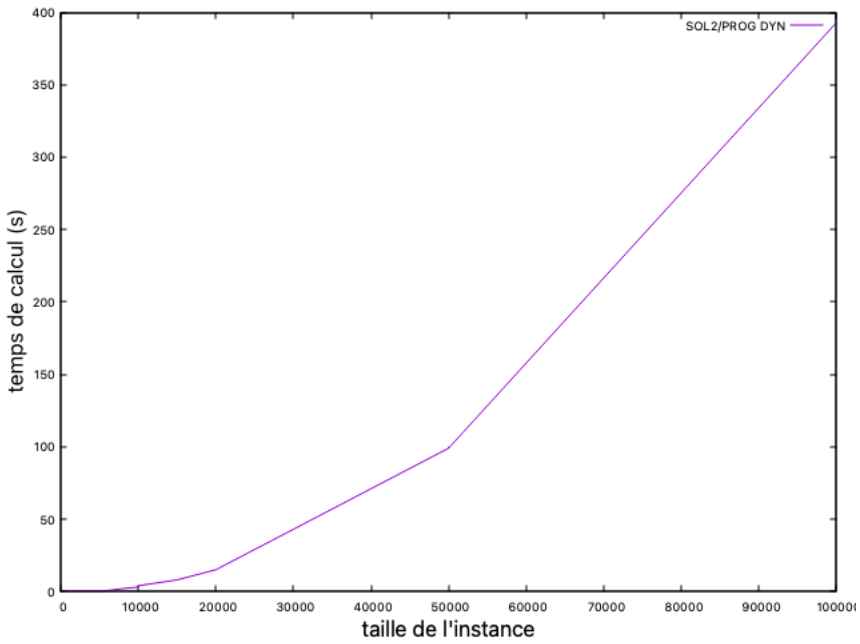
$$T(n+m) = 2T\left(\frac{n+m}{2}\right) + \mathcal{O}(n+m)$$

`SOL_2` réalise deux appels récursifs et l'appel courant est en $\mathcal{O}(n+2m) \equiv \mathcal{O}(n+m)$.
Le théorème maître ($2 = 2^1$, 2^e cas) donne alors une complexité spatiale totale en $\mathcal{O}((n+m)\log_2(n+m)) \equiv \mathcal{O}(n \times \log_2(n))$ (car $n \geq m$).

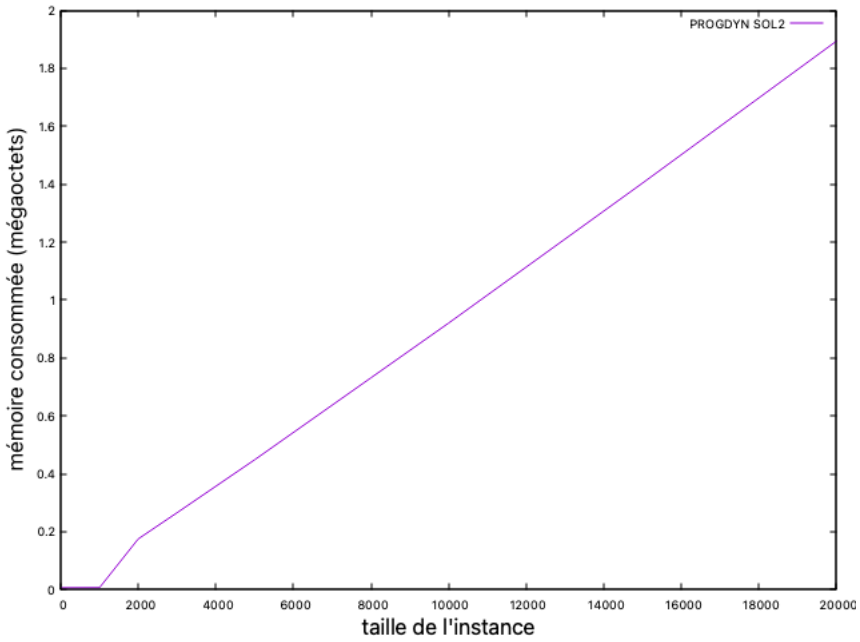
Question 28

`coupure` contient une première boucle en de m itérations où les opérations effectuées sont en $\mathcal{O}(1)$. Ensuite, deux boucles imbriquées en $\mathcal{O}(n \times m)$. La complexité totale est donc en $\mathcal{O}(nm+m) \equiv \mathcal{O}(nm)$.

4.3.2 Tâche D



Tâche D : temps CPU

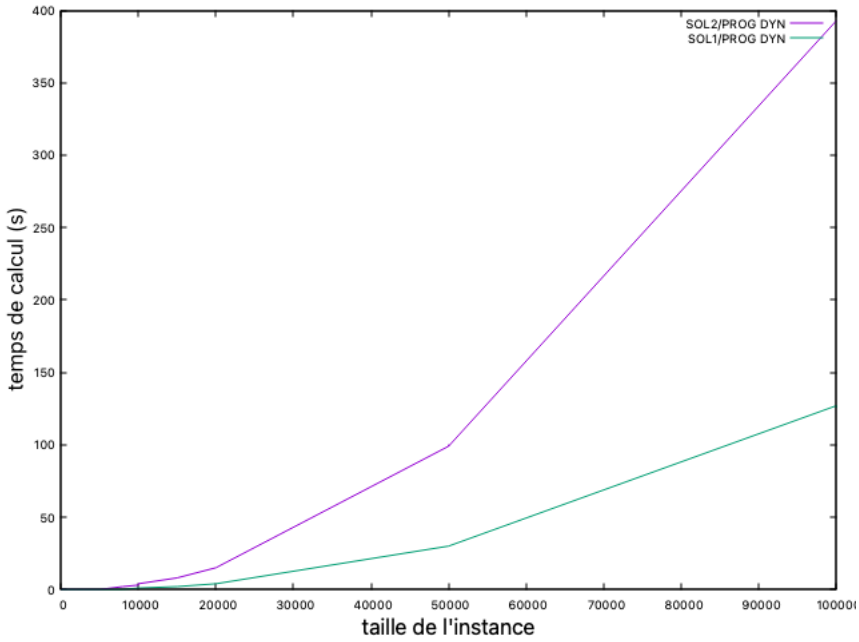


Tâche D : mémoire utilisée

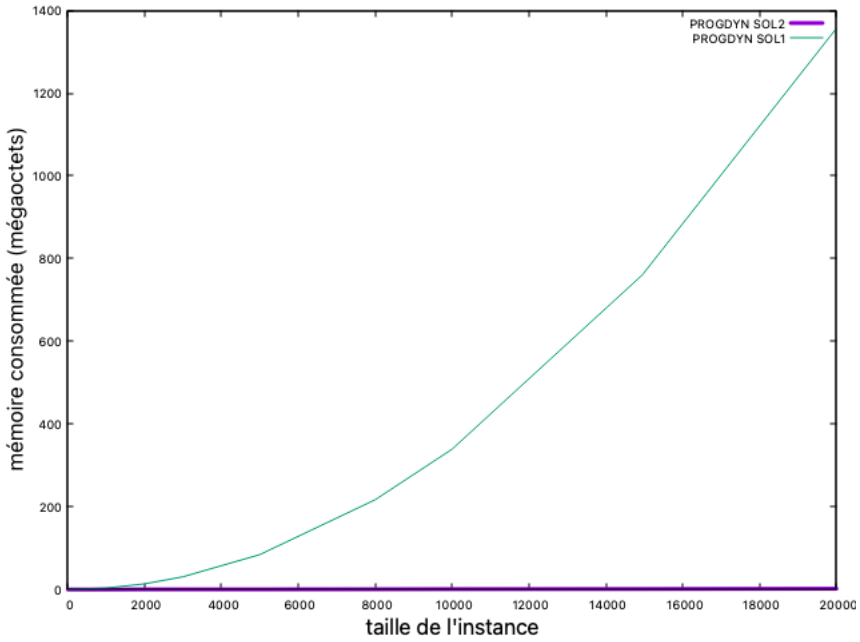
On observe que la mémoire utilisée par cette méthode est presque linéaire par rapport à la taille de l'instance testée. Pour une instance de 20000 caractères, la mémoire consommée est d'environ 1.9 mégaoctets. On peut calculer la mémoire consommée pour des instances plus grandes pourvu que la progression reste linéaire, grâce au coefficient directeur de la droite obtenue ci-dessus, qui vaut environ 9.5×10^{-5}.

Question 29

Comparons les temps CPU ainsi que la mémoire consommée respective de `SOL_1` et `SOL_2`.



Comparaison des temps CPU de `SOL_1` et `SOL_2`



Comparaison de la mémoire consommée par `SOL_1` et `SOL_2`

On remarque que la méthode "diviser pour régner" appelant `SOL_2` est environ 3 fois plus lente que la méthode appelant `SOL_1`. Par contre, la mémoire utilisée par la méthode appelant `SOL_2` est négligeable comparée à celle appelant `SOL_1`.
On gagne donc énormément en complexité spatiale en triplant le temps CPU, donc en perdant en complexité temporelle.

5 L'alignement local des séquences

Question 30

On vérifie en expérimentant avec 3 instances `Inst_0000125_BO.adn`, `Inst_0000200_BO.adn` et `Inst_0000250_BO.adn` qu'on a construites dans le fichier `Instances_genome`.
Les tests se trouvent dans le choix 7 de `myProg`. Ces tests confirment bien que le coût de l'alignement est égal à celui de la formule donnée pour $|x| << |y|$.

Question 31

En appliquant cette formule l'alignement optimal de la question 2 sera de coût 0. Pourtant, il n'existe aucun alignement de cout 0 pour cet exemple.

Question 32

On peut facilement adapter `SOL_1` qui est en $\mathcal{O}(n * m)$ en termes de complexité spatiale et `DIST_1`, `DIST_2` et `PROG_DYN` en modifiant les coûts, comme proposé dans l'enoncé. Par contre, on ne pourra pas adapter la fonction `SOL_2` pour ameliorer la complexité spatiale (méthode "diviser pour régner") car le découpage (fonction `coupure`) en sous-problèmes ne prendra pas en compte que le coût au début et à la fin doit être nul (ce n'est pas le cas quand on coupe avec `coupure`), mais une méthode avec des indicateurs (flags) pourra régler cela en passant quelques paramètres à la fonction pour remettre le début et la fin à un coût nul.