HLIN608 - Algorithmique du texte

-

Optimisations d'algorithmes d'assemblages de séquences ADN générées par modèle de Markov caché

-

Antoine AFFLATET, Hayaat HEBIRET et Jérémie ROUX

2019 - 2020

Table des matières

I Introduction 3

II Génération des séquences d'ADN de taille définie 4

1 Matrice de transition des nucléotides dans l'ADN 4

2 Probabilité et séquençage d'un fragment virtuel d'ADN 4

3 Détermination d'un enchaînement des nucléotides par un modèle de Markov caché 5

4 Déroulement de l'algorithme de génération de séquences 6
4.1 Sélection probabiliste du nucléotide suivant 6
4.2 Génération d'une séquence de nucléotides avec le modèle étudié 7

III Algorithme d'assemblage de séquences et optimisations 8

1 Algorithme d'assemblage naïf exact 8

2 Algorithme d'assemblage naïf non-exact 8

3 Optimisation en supprimant les sous-mots 9

4 Optimisation en ne recalculant pas totalement la matrice d'assemblage 9

IV Résultats obtenus et comparaison des optimisations de l'algorithme d'assemblage 11

1 Nombre de nucléotides gagnés par l'assemblage 11

2 Pourcentage de nucléotides gagnés par l'assemblage 12

3 Temps d'exécution des optimisations de l'algorithme d'assemblage 14

V Conclusion 16

Références 16

Première partie

Introduction

Nous sommes 3 étudiants de 3ème année de licence informatique de la Faculté des Sciences (Université de Montpellier) :

- Antoine AFFLATET
- Hayaat HEBIRET
- Jérémie ROUX (CMI)

Dans le cadre de l'UE HLIN608 (Algorithmique du texte), nous avons choisi le sujet « Optimisations d'algorithmes d'assemblages de séquences ADN générées par modèle de Markov caché » que nous avons proposé à Mme Annie Chateau.

Fig. 1 – *Vue d'artiste d'une double hélice d'ADN codant de l'information génétique*

Le but de ce travail est de générer des séquences de nucléotides pour former des fragments virtuels d'acide désoxyribonucléique (ADN) (voir Fig. 1) de taille définie t (Partie 2). Ensuite, nous allons en assembler 10 (Partie 3), c'est à dire faire en sorte qu'une super-séquence de taille minimale puisse contenir l'ensemble de l'information des 10 fragments de taille t précédemment générés. Nous étudierons plusieurs versions plus ou moins optimisées de cet algorithme d'assemblage.

Enfin, dans les Parties 4 et 5, nous analyserons les résultats obtenus en mettant en évidence la meilleure utilisation de ces algorithmes d'assemblages en fonction des contraintes liées à la complexité en temps. Dans ce but nous ferrons donc varier la taille t des séquences initiales à assembler et comparerons les différentes implémentations de l'algorithme d'assemblage.

Deuxième partie

Génération des séquences d'ADN de taille définie

Afin de mettre en application nos futurs algorithmes, on travaille sur l'alphabet des 4 lettres qui représentent les abréviations de chacun des 4 nucléotides qui composent un brin d'ADN :

- A (Adénine)
- C (Cytosine)
- G (Guanine)
- T (Thymine)

1 Matrice de transition des nucléotides dans l'ADN

On concatène les lettres afin de créer des mots (séquences) qui codent l'information génétique. Pour rendre aléatoire la génération des séquences d'ADN tout en gardant un semblant de vérité biologique, on concatène ces quatre lettres selon la matrice de transition probabiliste présentée en Fig. 2.

$p_{x\ !\ y}$	A	C	G	T
A	0:300	0:205	0:285	0:210
C	0:322	0:298	0:078	0:302
G	0:248	0:246	0:298	0:208
T	0:177	0:239	0:292	0:292

Fig. 2 – *Matrice de transition probabiliste entre les nucléotides dans une séquence d'ADN [3]*

Cette matrice présente la probabilité générale pour chaque nucléotide y d'apparaître après un nucléotide x. On note cette probabilité $p_{x\ !\ y}$ Par exemple, la probabilité pour un nucléotide C donné que s'en suive un A est de $p_{C\ !\ A} = 0{:}322$.

2 Probabilité et séquençage d'un fragment virtuel d'ADN

Le but de cette génération est de créer un séquence. On suppose l'équiprobabilité du nucléotide en tête de la séquence, soit une probabilité de 0:25 pour chacun des 4 nucléotides (1).

$$p(A) = p(T) = p(G) = p(C) = p_{debut} = 0{:}25 \tag{1}$$

Pour calculer la probabilité d'une séquence, on multiplie les probabilités des transitions indiquées dans la matrice de transition. Voici quelques exemple pour calculer la probabilité de la séquence AA (2) et CTGAA (3).

$$p(AA) = p_{debut}\quad p_{A\ !\ A} = 0{:}250\quad 0{:}300 = 0{:}075 \tag{2}$$

$$p(CTGAA) = p_{debut}\quad p_{C\ !\ T}\quad p_{T\ !\ G}\quad p_{G\ !\ A}\quad p_{A\ !\ A} = 0{:}250\quad 0{:}302\quad 0{:}292\quad 0{:}248\quad 0{:}300\quad 0{:}00164 \tag{3}$$

On utilisera par la suite la probabilité de cette séquence afin d'évaluer le nombre minimal d'exécution de l'algorithme qui assemble 10 de ces séquences de même taille pour couvrir la majorité des cas. On suppose que l'algorithme d'assemblage devra d'autant plus être répété que les séquences sont longues, la probabilité associée à une séquence est d'autant plus faible que la longueur de cette séquence est importante.

3 Détermination d'un enchaînement des nucléotides par un modèle de Markov caché

Le modèle de Markov caché (ou *hidden Markov model* en anglais, abrégé HMM) est dans notre cas un automate qui permet de représenter sous forme graphique la matrice de transition probabiliste vue précédemment. On obtient donc l'automate de la Fig. 3 qui rassemble les données de la Fig. 2 en intégrant l'équiprobabilité du nucléotide en tête de séquence.

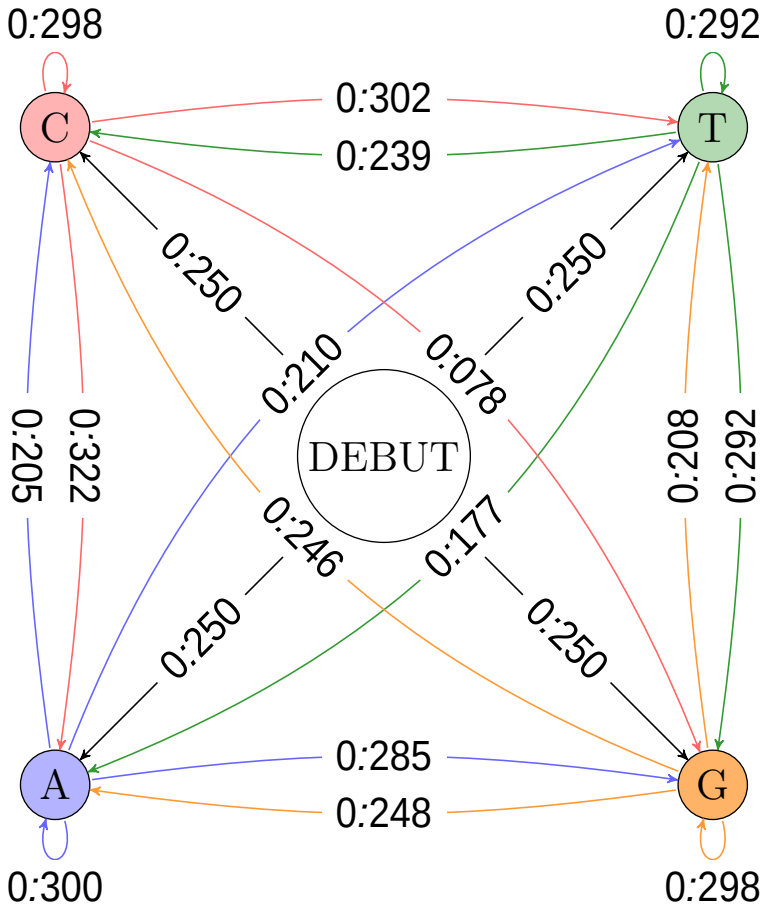

Fig. 3 – *Modèle de Markov caché associé à la* Fig. *2*

La taille t de la séquence étant définie, on limite le nombre de transitions t pour créer une séquence de cette taille. On ajoute un à un les nucléotides en respectant le modèle de Markov caché évoqué ci-dessus. Une fois la séquence de longueur t (et donc après avoir effectué t transitions), le dernier nucléotide ajouté marque la fin de la séquence. On remarque que la somme des probabilités des transitions sortantes (celles qui sont de la même couleur) pour un état donné fait toujours 1.

4 Déroulement de l'algorithme de génération de séquences

4.1 Sélection probabiliste du nucléotide suivant

Pour générer des séquences, il faut définir un sous-programme (Algorithme 1) qui lit la matrice de transition et donne le nucléotide suivant. Ce que l'on définit comme matrice de transition dans ce programme est en réalité un tableau représentant la probabilité à ce que les nucléotides A,C,G,T (dans cet ordre là) apparaissent après un nucléotide n, le dernier de la séquence en cours d'assemblage. Celui-ci n'est pas donné, la matrice en entrée de l'algorithme est adaptée au besoin pour correspondre aux probabilités à ce que A,C,G ou T succèdent à n.

Algorithme 1 : *selectionNucleotideSelonMatrice*(M : matrice de transition) : caractère (nucléotide)

```
Variable : p : entier;
p ← entierAleatoireEntre(0,1000);
si p ≤ M[0] × 1000 alors
    renvoyer 'A';
sinon
    si p ≤ (M[0] + M[1]) × 1000 alors
        renvoyer 'C';
    sinon
        si p ≤ (M[0] + M[1] + M[2]) × 1000 alors
            renvoyer 'G';
        sinon
            renvoyer 'T';
        fin
    fin
fin
renvoyer max;
```

On note $M_{x\,!\,y}$ la matrice à une dimension qui représente pour un x donné le tableau des probabilités de x vers tout y (toutes les possibilités de l'alphabet étudié). On génère un entier e compris entre 0 et 1000. Voici un exemple étant donné la matrice $M_{A\,!\,y} = M_{A\,!\,fA;C;G;Tg}$ qui contient l'ensemble des probabilités des transitions de A vers A, C, G et T (dans cet ordre). Voici quel va être le nucléotide choisi en fonction l'entier généré aléatoirement e :

$$M_{A\,!\,y} = M_{A\,!\,fA;C;G;Tg} = [p_{A\,!\,A};p_{A\,!\,C};p_{A\,!\,G};p_{A\,!\,T}] = [0{:}300;0{:}205;0{:}285;0{:}210]$$

Si $p_{A\,!\,y}\ 2\ [0;p_{A\,!\,A}] = [0;0{:}300]$ alors $y = A$,
autrement dit, si $e\ 2\ [0;300]$ alors on obtient le caractère A

Sinon si $p_{A\,!\,y}\ 2]p_{A\,!\,A};p_{A\,!\,A} + p_{A\,!\,C}] =]0{:}300;0{:}300 + 0{:}205] =]0{:}300;0{:}505]$ alors $y = C$,
autrement dit, si $e\ 2]300;505]$ alors on obtient le caractère C

Sinon si $p_{A\,!\,y}\ 2]p_{A\,!\,A} + p_{A\,!\,C};p_{A\,!\,A} + p_{A\,!\,C} + p_{A\,!\,G}] =]0{:}300 + 0{:}205;0{:}300 + 0{:}205 + 0{:}285] =]0{:}505;0{:}790]$
alors $y = G$, autrement dit, si $e\ 2]505;790]$ alors on obtient le caractère G

Sinon si $p_{A\,!\,y}\ 2]p_{A\,!\,A} + p_{A\,!\,C} + p_{A\,!\,G};p_{A\,!\,A} + p_{A\,!\,C} + p_{A\,!\,G} + p_{A\,!\,T}] =$
$]0{:}300 + 0{:}205 + 0{:}285;0{:}300 + 0{:}205 + 0{:}285 + 0{:}210] =]0{:}790;1]$ alors $y = T$,
autrement dit, si $e\ 2]790;1000]$ alors on obtient le caractère T

4.2 Génération d'une séquence de nucléotides avec le modèle étudié

Pour générer cette séquence de nucléotides, on a implémenté l'Algorithme 2. La génération de la séquence dépend essentiellement de la variable t donnée en entrée qui définit la longueur de cette séquence. Ainsi, si la taille de la séquence attendue est supérieure à 1, le choix du nucléotide en tête est A, C, G ou T en équiprobabilité de 0.25. On incrémente une variable x qui contient la longueur de la séquence en cour de construction.

Algorithme 2 : *generation*(t : entier (taille de la séquence à générer)) : chaîne de caractères (séquence)

```
Variables : s : chaîne de caractères, x : entier, dernierNucleotide : caractère;
s ← "";
x ← 0;
si t > 1 alors
    s ← s + selectionNucleotideSelonMatrice([0.250;0.250;0.250;0.250]);
    x ← x + 1;
fin
tant que t > x faire
    dernierNucleotide ← s[ 1];
    si dernierNucleotide = 'A' alors
        s ← s + selectionNucleotideSelonMatrice([0.300;0.205;0.285;0.210]);
    sinon
        si dernierNucleotide = 'C' alors
            s ← s + selectionNucleotideSelonMatrice([0.322;0.298;0.078;0.302]);
        sinon
            si dernierNucleotide = 'G' alors
                s ← s + selectionNucleotideSelonMatrice([0.248;0.246;0.298;0.208]);
            sinon
                s ← s + selectionNucleotideSelonMatrice([0.177;0.239;0.292;0.292]);
            fin
        fin
    fin
    x ← x + 1;
fin
renvoyer s;
```

Tant que $t > x$, c'est à dire que la longueur t de la séquence attendue en sortie de l'algorithme est strictement supérieure à la longueur x de la séquence en cours de construction, on regarde quel est le *dernierNucleotide* de la séquence en construction. Ce nucléotide va permettre de choisir la ligne correspondante dans la matrice de transition probabiliste présentée en Fig. 2. On envoi donc cette ligne (matrice à une dimension) qui n'est pas la même selon le *dernierNucleotide* dans la fonction *selectionNucleotideSelonMatrice*(M) présentée dans l'Algorithme 1. Enfin, on concatène la chaîne en cours de génération avec le caractère obtenu après exécution de la fonction *selectionNucleotideSelonMatrice*(M). Cette chaîne ayant grandit d'un caractère, on incrémente x et on recommence les instructions à partir du test $t > x$ jusqu'à ce que ce test ne soit plus vrai et qu'on renvoi en sortie de l'algorithme la chaîne de longueur $t = x$ ainsi générée.

Troisième partie

Algorithme d'assemblage de séquences et optimisations

L'objectif de la génération de ces chaînes de taille t est de pouvoir en construire 10 en respectant le modèle étudié dans la Partie précédente et faire tourner un algorithme d'assemblage qui va faire en sorte de créer une chaîne de longueur minimale contenant l'ensemble des séquences des 10 chaînes. On réduit la longueur de cette super-chaîne en créant des chevauchements (ou *overlaps* en anglais) de manière à ce que la taille d'une super-séquence/super-chaîne soit le plus possible inférieure à la somme des tailles des chaînes que l'on assemble.

On cherche à créer un algorithme naïf pour résoudre ce problème d'assemblage de 10 séquences en une superséquence de taille minimale, garantissant le résultat.

1 Algorithme d'assemblage naïf exact

Cet algorithme produit la plus petite super-séquence cependant il nécessite l'exploration de l'ensemble des super-séquences possibles afin de toutes les comparer ; il est donc exponentiel. Nous nous baserons donc par la suite sur un algorithme d'assemblage approché grâce à une méthode gloutonne (naïf non-exact dans ce cas là).

2 Algorithme d'assemblage naïf non-exact

Un algorithme naïf est le premier algorithme qui nous vient à l'esprit pour résoudre un problème posé. Dans cet Algorithme 3, on fusionne en priorité les séquences qui maximisent le chevauchement. Tant que l'ensemble des séquences à fusionner contient plus d'un élément, on créé une matrice contenant l'ensemble des valeurs de chevauchement. On choisi dans la matrice un chevauchement maximal et on fusionne les deux séquences dont est issu le chevauchement maximal. On supprime de l'ensemble des séquences les deux que nous venons de fusionner et y ajoutons la nouvelle formée.

Algorithme 3 : *assemblageNaifNonExact*(L : liste de mots) : mot (super-séquence)

```
Variables : nb, i, j, overlapMax : entiers, motFusion : chaîne de caractères, T : tableau, M : matrice;
nb ← longueur(L);
motFusion ← "";
T ← recupererMots(L);
si nb = 1 alors
    motFusion ← T[0];
fin
tant que nb > 1 faire
    M ← calculerOverlap(T);
    overlapMax ← max(M);
    (i;j) ← position(M;overlapMax);
    motFusion ← fusionOverlap(i;j;overlapMax;T);
    T:effacer(i);
    T:effacer(j);
    T:ajouterFin(motFusion);
    nb ← nb − 1;
fin
renvoyer motFusion;
```

Cet algorithme génère la super-séquence qui fusionne à chaque tour de boucle les deux séquences qui ont le plus grand chevauchement. L'algorithme a une complexité en $O(n^3 \quad m)$ (avec n le nombre de séquences et m la taille de la plus grande séquence).

3 Optimisation en supprimant les sous-mots

Dans cette optimisation (Algorithme 4), on cherche à se rapprocher d'une solution optimale. Ainsi, nous vérifions avant la fusion des séquences s'il n'existe pas de fusions de séquences qui ne seraient pas détectées par les chevauchements. En effet, si un mot est strictement inclus dans un autre, il n'existe pas de chevauchements entre les deux. Nous éliminons donc toutes les séquences pouvant être inclues dans d'autres. Pour la suite du programme, nous utilisons le même fonctionnement que celui de l'Algorithme 3.

Algorithme 4 : *assemblageSuppressionSousMots*(L : liste de mots) : mot (super-séquence)

```
Variables : nb, i, j, overlapMax : entiers, motFusion : chaîne de caractères, T : tableau, M : matrice;
nb    longueur(L);
motFusion    "";
T    recupererMots(L);
T    triMotsParLongueur(T);
T    suppressionMotsInclus(T);
si nb = 1 alors
    motFusion    T[0];
fin
tant que nb > 1 faire
    M    calculerOverlap(T);
    overlapMax    max(M);
    (i;j)    position(M;overlapMax);
    motFusion    fusionOverlap(i;j;overlapMax;T);
    T:effacer(i);
    T:effacer(j);
    T:ajouterFin(motFusion);
    nb    nb    1;
fin
renvoyer motFusion;
```

Cette amélioration n'affecte pas beaucoup la complexité puisqu'elle rajoute un tri fusion en complexité $o(n \quad log \quad n)$ ainsi que la suppression des mots inclus en $o(n^2 \quad m)$ (avec n le nombre de séquences et m la taille de la plus grande séquence). Ces ajouts sont inférieurs en ordre de grandeur par rapport au recalcul des chevauchements. Cet Algorithme 4 a donc une complexité en $O(n^3 \quad m)$ (avec n le nombre de séquences et m la taille de la plus grande de ces séquences).

4 Optimisation en ne recalculant pas totalement la matrice d'assemblage

Nous avons modifié l'algorithme d'assemblage précédent afin de ne calculer qu'une seule fois la matrice des chevauchements (ou *overlaps* en anglais). Dans cette optimisation (Algorithme 5), on cherche à ne construire qu'une matrice durant toute l'exécution et à la conserver.

Nous suivons le même schéma que lors de l'optimisation précédente jusqu'au moment où on fusionne les mots. À cet instant, au lieu de créer une nouvelle matrice, on modifie la matrice existante afin de ne garder que les informations concernant les séquences qui n'ont pas encore été fusionnées.

Algorithme 5 : *assemblageNonRecalculMatrice*(*L* : liste de mots) : mot (super-séquence)

```
Variables : nb, i, j, overlapMax : entiers, motFusion : chaîne de caractères, T : tableau, M : matrice;
nb ← longueur(L);
motFusion ← "";
T ← recupererMots(L);
T ← triMotsParLongueur(T);
T ← suppressionMotsInclus(T);
si nb = 1 alors
    motFusion ← T[0];
fin
tant que nb > 1 faire
    M ← calculerOverlap(T);
    overlapMax ← max(M);
    (i;j) ← position(M;overlapMax);
    motFusion ← fusionOverlap(i;j;overlapMax;T);
    M ← rangeMatrice(M;i;j);
    T:effacer(i);
    T:effacer(j);
    M:effacerDerniereColonne();
    M:effacerDerniereLigne();
    T:ajouterFin(motFusion);
    nb ← nb − 1;
fin
renvoyer motFusion;
```

Nous allons donc ordonner la matrice de manière à ce que la dernière ligne et la dernière colonne comportent les informations obsolètes du fait de la fusion des séquences. L'avant dernière ligne et colonne représentent quant à elle les informations liées à la nouvelle séquence formée. Il suffit donc de supprimer la dernière ligne et colonne de la matrice (ainsi que les deux mots fusionnés) et d'ajouter la nouvelle séquences formée en dernière position (avant dernière ligne et colonne dans la matrice avant rangement). Le reste du fonctionnement est similaire à l'optimisation précédente.

Cette amélioration permet de diminuer la complexité de l'algorithme, il rajoute en effet le rangement de la matrice en $O(n^3)$ mais diminue le calcul des chevauchements. L'algorithme a une complexité en $O(n^3)$ (avec n le nombre initial de séquences).

Quatrième partie

Résultats obtenus et comparaison des optimisations de l'algorithme d'assemblage

Afin d'observer et quantifier la différence entre toutes les implémentations de l'algorithme d'assemblage effectuées, on fait varier la taille des séquences à assembler. Les résultats obtenus sont les suivants.

1 Nombre de nucléotides gagnés par l'assemblage

On cherche tout d'abord à étudier la longueur des super-séquences formées. Ce que l'on appelle le gain moyen g_{moyen} (4) par l'assemblage est en fait la différence entre la somme maximale possible pour la super-séquence (sommes des tailles des séquences assemblées) et la longueur de la super-séquence obtenue. On constate d'après les données rassemblées dans la Fig. 4 que pour chaque taille t de séquence initiale d'assemblage, l'algorithme naïf, celui avec optimisation des sous-mots et celui avec optimisation du non recalcul de la matrice donnent tout 3 des résultats similaires. On constate que le nombre de nucléotides gagné par l'assemblage est d'environ 11 lorsque l'on assemble 10 séquence de taille 2. Ce gain est d'environ 15 nucléotides pour les autres tailles t de séquences étudiées : 5, 10, 50 et 100.

$$g_{moyen} = t \quad n_{motsAssembler} \quad \left(\sum_{i=1}^{precision} g_i \right) = precision = t \quad 10 \quad \left(\sum_{i=1}^{100} g_i \right) = 100 \tag{4}$$

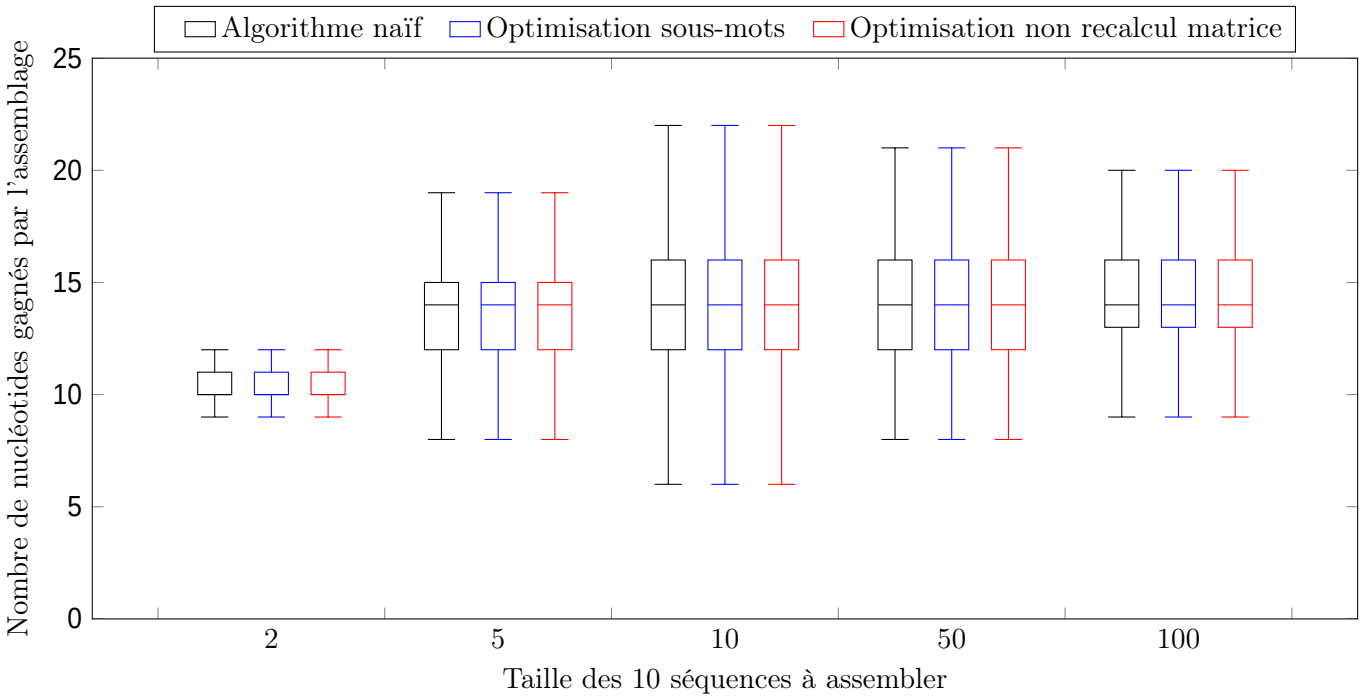

Fig. 4 – *Nombre de nucléotides gagnés par l'assemblage (par rapport à la super-séquence de taille maximale) en fonction de la taille des séquences à assembler (avec 100 super-séquences générées pour chaque taille de séquence initiale) pour chacun des 3 algorithmes d'assemblage*

Ces résultats étant équivalents peu importe l'algorithme choisi, on présente l'évolution de gain pour l'algorithme naïf sur des tailles de séquences initiales allant de 2 à 100 (voir Fig. 5). On constate que pour des tailles de séquences initiales allant de 2 à 4, le nombre de nucléotides gagnés par l'assemblage augmente, il est à peu près constant pour des tailles de mots à assembler allant de 4 à 100. On peut conclure de ce résultat que le gain de nucléotides n'est pas proportionnel à la taille des séquences à assembler. Cependant, ce résultat n'a pas vraiment de sens et il conviendrait de comparer ces valeurs par rapport au pourcentage que ce gain représente (voir section suivante).

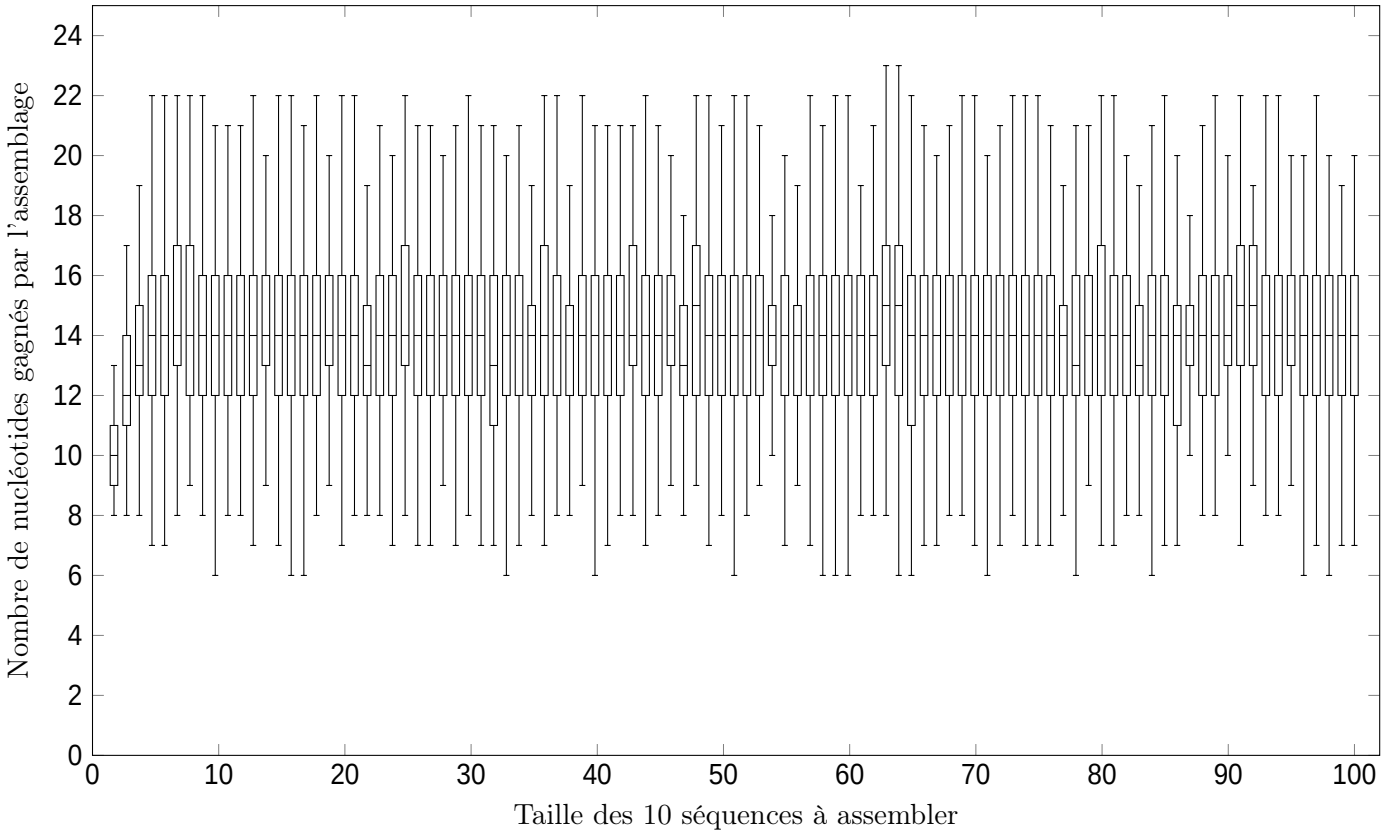

Fig. 5 – *Évolution du nombre de nucléotides gagnés par l'assemblage de l'algorithme naïf (par rapport à la super-séquence de taille maximale) en fonction de la taille des séquences à assembler (avec 100 super-séquences générées pour chaque taille de séquence initiale)*

2 Pourcentage de nucléotides gagnés par l'assemblage

On cherche maintenant à comparer des gains moyens en pourcentages de nucléotides $g_{moyenPourcentage}$ (5) car il est plus logique de comparer ce que ce gain représente par rapport à la taille de la super-séquence maximale. D'après la Fig. 6, on constate que le pourcentage de nucléotides gagnés par l'assemblage est d'autant plus important que les séquences à assembler sont courtes.

$$g_{moyenPourcentage} = 100 \quad (((\sum_{i=1}^{precision} g_i)=(t \quad n_{motsAssembler})) \quad precision) = 100 \quad (((\sum_{i=1}^{100} g_i)=(t \quad 10)) \quad 100) \quad (5)$$

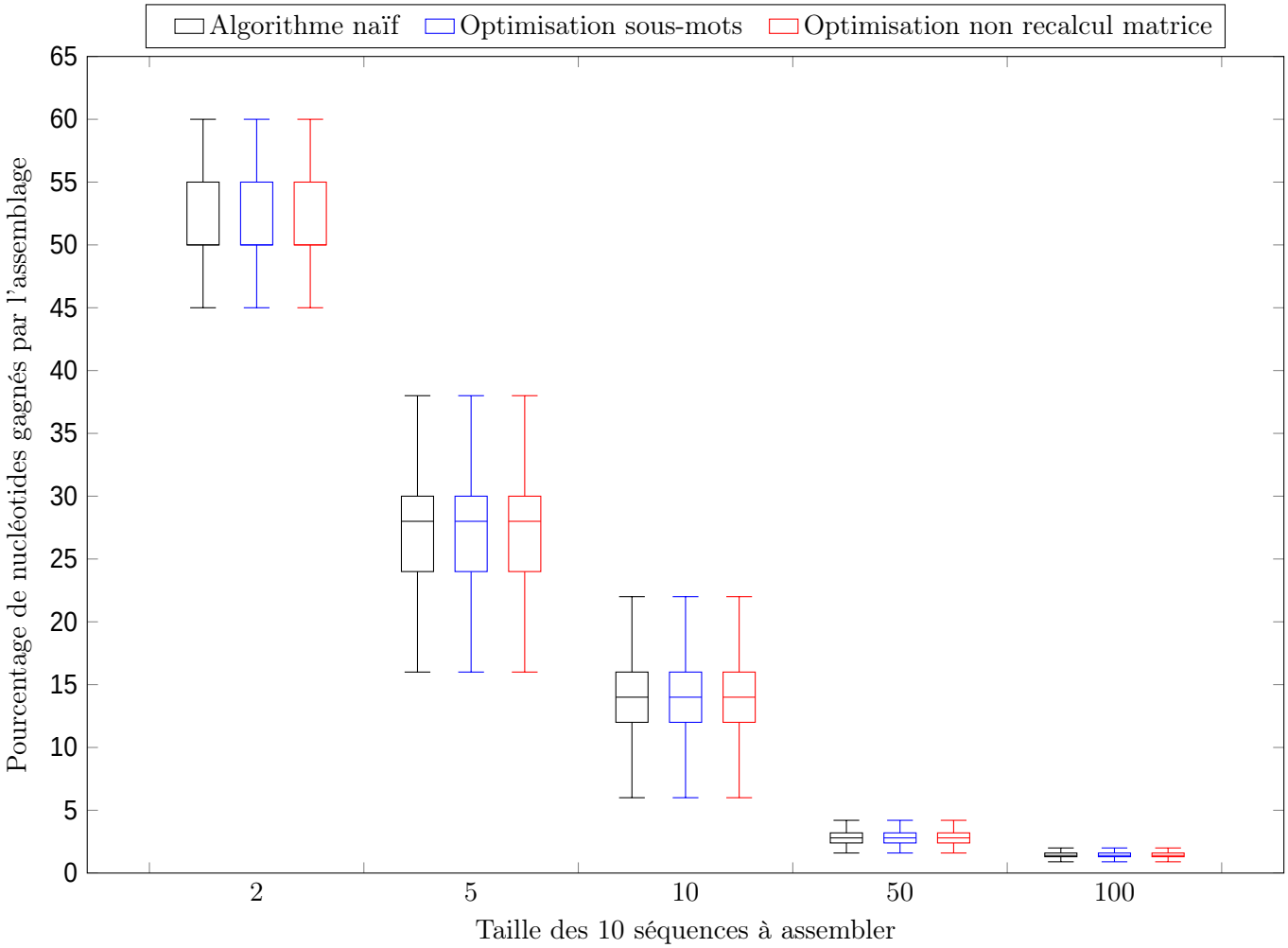

Fig. 6 – *Pourcentage de nucléotides gagnés par l'assemblage (par rapport à la super-séquence de taille maximale) en fonction de la taille des séquences à assembler (avec 100 super-séquences générées pour chaque taille de séquence initiale) pour chacun des 3 algorithmes d'assemblage*

Le pourcentage de nucléotides gagnés est autour de 50% pour des séquences de taille 2 à assembler, 27% pour des séquences de taille 5, 15% pour celles de taille 10, 4% pour celles de taille 50 et 2% pour celles de taille 100. Comme pour la Fig. 4, les 3 algorithmes étudiés ont des résultats similaires.

On s'intéresse donc au graphe de la Fig. 7 pour observer l'évolution du pourcentage de nucléotides gagnés par l'assemblage en fonction de la taille initiale des 10 séquences à assembler (ici dans le cas de l'algorithme naïf). On constate que ce pourcentage de gain est de 50% pour des séquences initiales de taille 2 et décroît jusqu'à 15% pour celles de taille 10. On observe ensuite une asymptote horizontale autour de 2% laissant suggérer que c'est un pourcentage minimum pour des tailles infinies de séquence.

Chose intéressante, on observe que la répartition des pourcentages diminue au fur et à mesure que le pourcentage diminue, la boîte à moustache associée étant de plus en plus petite. Ceci s'explique principalement par le fait que la taille des séquences augmentant, il devient de plus en plus difficile de faire de gros chevauchements par rapport à la taille de ces mots. La probabilité d'obtenir telle ou telle séquence diminuant quand la taille augmente, ceci explique également cette baisse progressive d'amplitude des pourcentages.

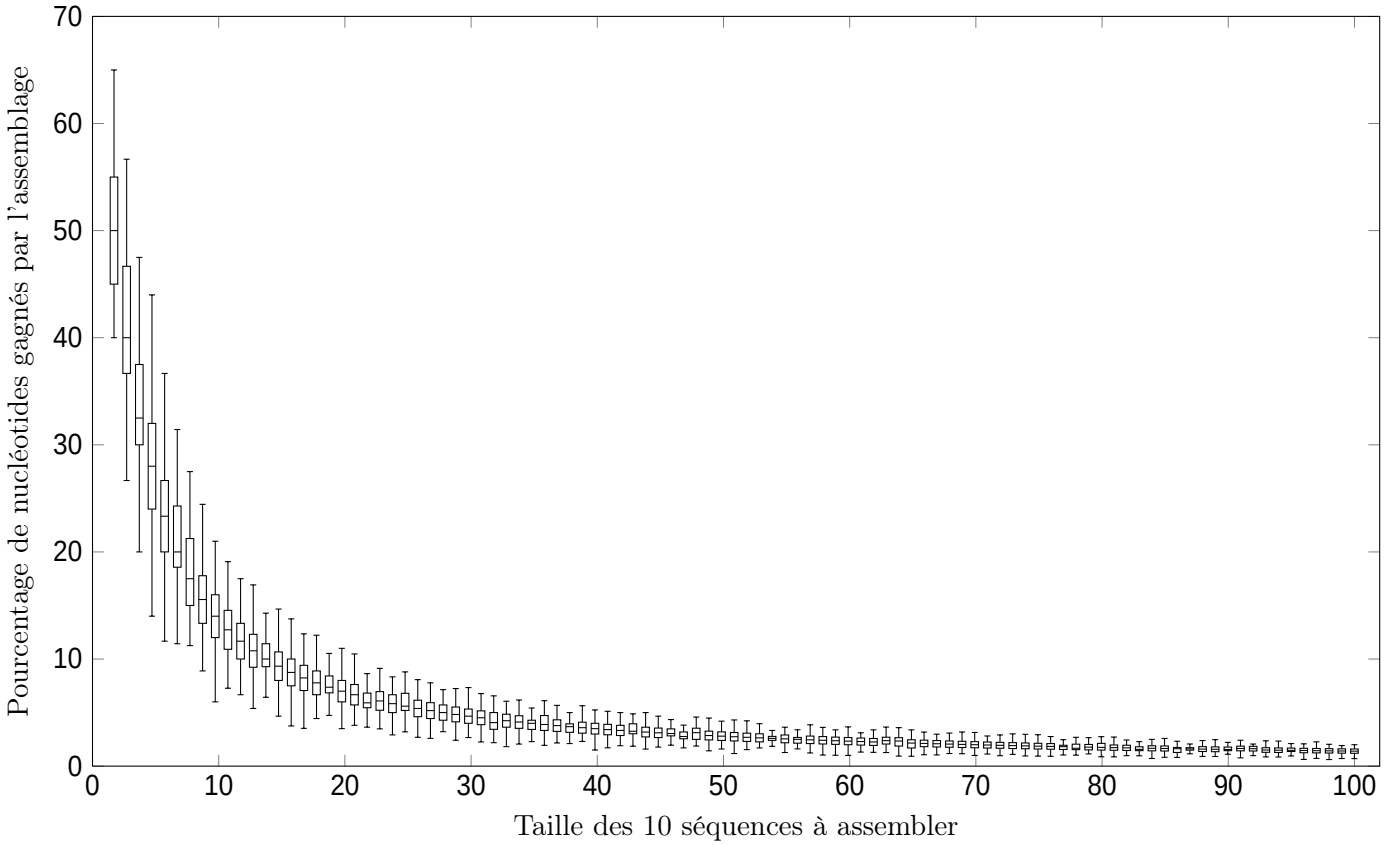

Fig. 7 – *Évolution du pourcentage de nucléotides gagnés par l'assemblage (par rapport à la super-séquence de taille maximale) en fonction de la taille des séquences à assembler (avec 100 super-séquences générées pour chaque taille de séquence initiale)*

3 Temps d'exécution des optimisations de l'algorithme d'assemblage

On observe maintenant l'évolution du temps d'exécution des algorithmes étudiés en fonction de la taille initiale des séquences à assembler. Les Figs. 8 et 9 rassemblent les données collectées, la première étant en échelle normale, la deuxième avec l'axe des ordonnées (temps en secondes) en échelle logarithmique.

On constate que l'algorithme naïf est assez gourmand en temps de calcul. L'optimisation avec la suppression des sous-mots en début de programme permet de gagner quelques millisecondes de temps de calcul mais l'évolution du temps d'exécution du programme reste assez similaire à celle de l'algorithme naïf. On peut dire que ces deux algorithmes là ont un temps d'exécution proportionnel à la taille des séquences à assembler.

Concernant l'optimisation qui consiste à ne pas recalculer la matrice à chaque tour de la boucle d'assemblage, on observe un gain de temps conséquent (de l'ordre de 0.012 secondes) pour des assemblages de taille 100 comparativement aux 2 algorithmes précédemment étudiés. On constate une brusque augmentation (de l'ordre du dixième de milliseconde) du temps de calcul avec le graphe de la Fig. 9 entre la taille 22 et 23 des séquences initiales pour cette optimisation. Nous ne savons pas exactement ce qui est à l'origine de cette soudaine augmentation qui visiblement n'est pas une erreur de code puisque les données avant et après cette augmentation sont quasiment constantes. Une allocation mémoire pourrait en être à l'origine car on constate également une légère hausse au même endroit pour les autres implémentions (courbes bleue et rouge).

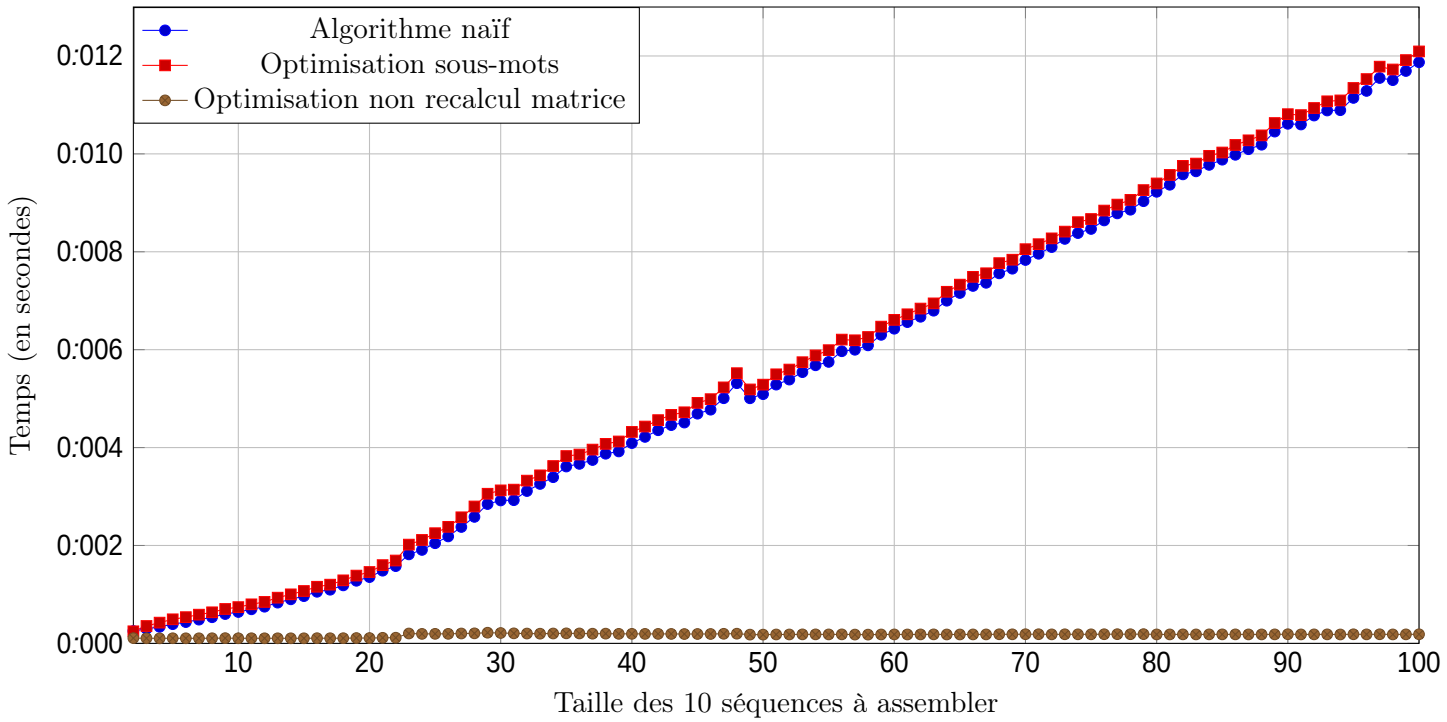

Fig. 8 – *Temps (en secondes) mis pour e ectuer un assemblage complet en fonction de la taille des séquences à assembler (avec 1000 super-séquences générées pour chaque taille de séquence initiale)*

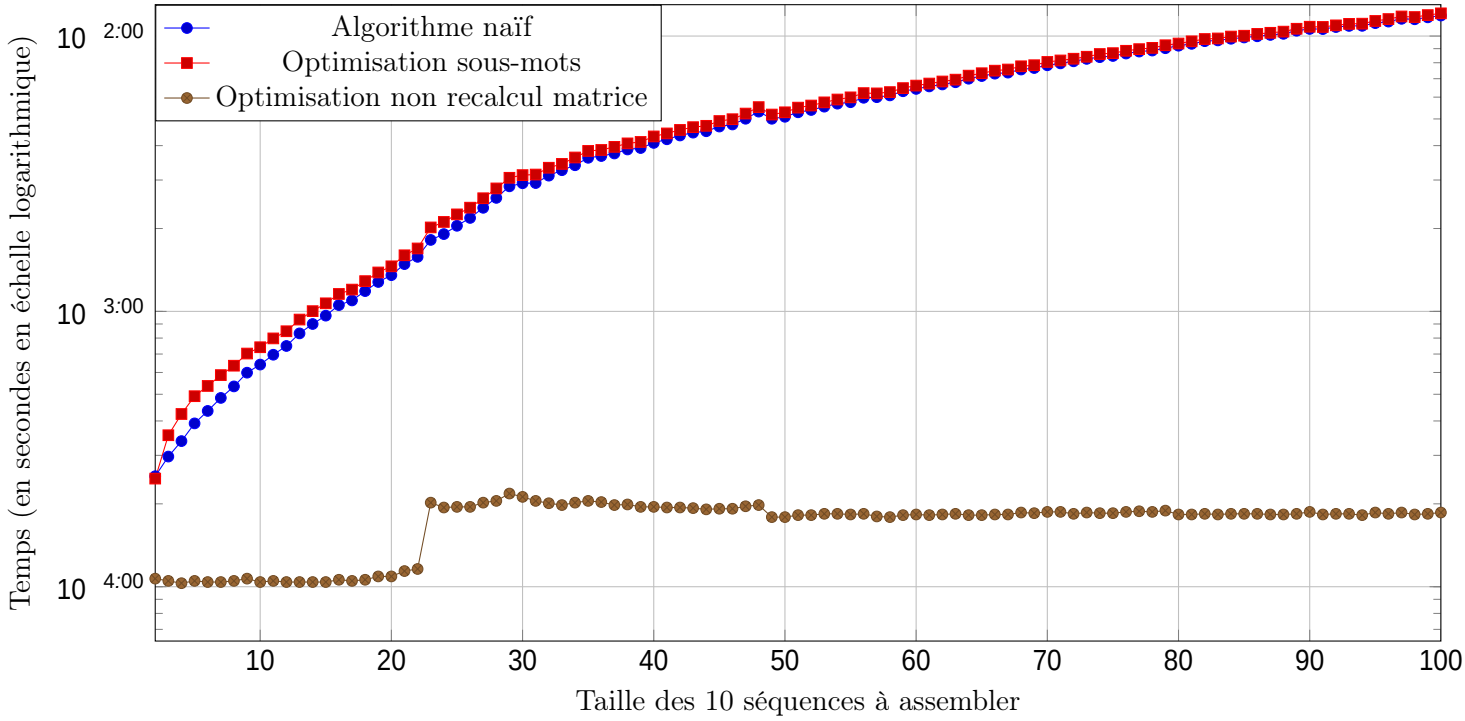

Fig. 9 – *Temps (en secondes en échelle logarithmique) mis pour e ectuer un assemblage complet en fonction de la taille des séquences à assembler (avec 1000 super-séquences générées pour chaque taille de séquence initiale)*