Université Libre de Bruxelles

ECOLE
POLYTECHNIQUE
DE BRUXELLES

Complément de programmation et d'algorithmique
INFO-H304

Alignement de Protéines avec l'algorithme de Smith-Waterman

Etudiants :

Achten Alexandre	494484
Guilbaud Nicolas	493487
Humblet Raphaël	514085

Professeur :

Roland Jérémie

Décembre 2022

Année académique 2022-2023

Table des matières

1 Introduction

L'objectif final de ce projet est de pouvoir déterminer, le plus efficacement possible, les 20 séquences de protéines se rapprochant le plus d'une séquence de protéine de recherche fournie à l'aide de l'algorithme de Smith-Waterman. Les protéines sont stockées dans une base de donnée sous le format NCI BLAST. Elle est donc répartie en plusieurs fichiers, le .pin, .phr et .psq qui contiennent respectivement l'index, les headers et les séquences de chacune des protéines.

Dans la Section 2 de ce rapport, le fonctionnement global du code est détaillé. De la lecture des différents fichiers, à l'initialisation des différentes structures de données nécessaires en passant par le calcul du score.

La Section 3 explique de façon détaillée l'algorithme de comparaison des séquences, son implémentation dans le code et la complexité de ce dernier.

La Section 4 décrit le fonctionnement et les améliorations apportées dans la version `projetopti` du projet, notamment avec l'utilisation de threads.

2 Fonctionnement global

Projet

Il a été décidé d'utiliser l'orienté objet afin d'améliorer la clarté et l'efficacité du code. Il existe 4 types d'objet différents : `Database, Index, Protein` et `Tas_min`. Ces dernières sont détaillées dans la suite de cette section.

Une fois les différents paramètres entrés par l'utilisateur, ils récupérés (chemin vers la protéine de recherche, chemin vers la base de donnée, chemin vers la matrice BLOSUM, gap open penalty et gap extension penalty) et l'objet Database est initialisé (voir Sous-section 2.3).

Il comporte un Index et le chemin vers la base de donnée. L'Index consiste en une "traduction" du fichier .pin. En effet, lorsque la base de donnée (Database) est initialisée, le fichier .pin est lu et les différentes informations concernant la base de données sont récupérées et stockées dans l'Index (voir Sous-section 2.2).

Une fois que les objets Database et Index sont initialisés, le programme lit la séquence de la protéine de requête à partir du chemin reçu en paramètre et la stocke dans un objet Protein (voir Sous-section 2.1).

Ensuite, le programme exécute la fonction `search_top_20()` qui parcourt le fichier .psq. Il récupère chaque séquence grâce aux offsets stockés dans l'Index et calcule son score de compatibilité avec la protéine de recherche avec la fonction `Smith_Waterman()`. Ce calcul de score est détaillé dans la Section 3.

Le score obtenu est ensuite comparé avec le score le plus petit se trouvant dans la liste top_20. Si le nouveau score est plus grand, la protéine actuelle va remplacer la protéine avec le score le plus bas dans la liste. L'objet top_20 est donc une liste

de 20 objets `Protein` contenant les séquences, score et numéro de l'offset dans le tableau d'offset stocké dans l'index.

Étant donné qu'il faut supprimer le minimum à chaque fois qu'une séquence possède un score supérieur au minimum de la liste. Il a été décidé que la liste top_20 serait structuré comme un Tas min, les motivations de ce choix et son implémentation sont détaillés dans la Sous-section 2.4.

Une fois tout le fichier .psq parcouru, la liste top_20 contient les 20 protéines ayant les meilleurs scores. Cependant, pour faciliter la lecture des headers dans l'ordre décroissant, il a été décidé de transférer les protéines du Tas min dans une nouvelle liste `proteins`. Cette liste contient les 20 protéines dans un ordre croissant (selon le score). Si le score est le même, les protéines sont alors triées de façon décroissante selon la place de l'offset dans l'index.

Lorsque la liste `proteins` est complète, la fonction `get_headers()` est appelée. Cette fonction cherche dans le fichier .psq les headers de chacune des 20 protéines. Elle écrit ensuite dans le terminal les headers et le score de chaque protéine.

Projet préliminaire
L'objectif du projet préliminaire était de récupérer le header de la séquence reçue. Le code est exactement le même mais à la place de calculer le score de compatibilité entre les 2 protéines. Les 2 protéines sont comparées et si les protéines sont les mêmes. Le header est récupéré et imprimé. Le code s'arrête dès que la séquence a été trouvée dans la base de donnée.

2.1 Classe Protein

L'implémentation de la classe `Protein` permet de faciliter la manipulation des protéines dans le code. Chaque protéine est caractérisée par un nom (`header`) et une séquence (`sequence`). Lors des comparaisons de protéines, un score leur sera attribué et sera stocké dans l'attribut `storedScore`. Le `offset_number` permet de repérer la protéine dans la base de donnée. La Table 1 récapitule les attributs et leurs types.

`Protein`

Attributes
`header, sequence :string`
`storedScore :int`
`offset_number :int32_t`

Functions Getters and setters for each attribute

TABLE 1 : Attributs et fonctions de la classe `Protein`

2.2 Classe Index

La classe `Index` contient les informations sur la base de donnée utilisée. Chaque attribut correspond à une caractéristique de la base de donnée. Les plus importants

et utilisés sont les `headerOffset` et `seqOffset`. Ils permettent de retrouver les header des protéines ainsi que leur séquence ce qui permet de facilement parcourir toute la base de donnée. Un schéma de la classe `Index` est présenté à la Table 2. Les attributs sont initialisés dans le constructeur de `Index` qui lit le fichier .pin dont le chemin est récupéré en paramètre. L'index est conforme au format NCBI BLAST [Farrar 2010].

`Index`

Attributes
`version, dbType, titleLength,`
`timeLength, seqNber, maxSeq :int32_t`
`residueCount :int64_t`
`title, time :string`
`headerOffset, seqOffset :int32_t*`

Functions Getters for each attributes

TABLE 2 : Attributs et fonctions de la classe `Index`

2.3 Classe Database

La classe `Database` permet de stocker l'index, les gaps_penaltys (open et extension) et les chemins vers la base de donnée et la matrice BLOSUM. Le constructeur de `Database` appelle le constructeur de `Index` et l'assigne directement à l'attribut `index`. La Table 3 illustre la structure de la classe `Database`.

`Database`

Attributes
`index :Index`
`db_path, blosum_path :string`
`gap_open_penality, gap_extension_penality :int`

Functions Getters and setters for each attribute

TABLE 3 : Attributs et fonctions de la classe `Database`

2.4 Classe Tas_min

La classe `Tas_min` a été créée afin d'accélérer la recherche des meilleurs scores. L'avantage de l'utilisation du Tas min est que l'obtention du minimum est en complexité de $\mathcal{O}(1)$. Son utilisation est justifiée par le fait que lors de la recherche des 20 scores maximum, il n'est pas nécessaire de garder tous les scores en mémoire. Lorsqu'un score est plus grand que le minimum du Tas min, le minimum est retiré et la nouvelle protéine est insérée. [Roland 2022]

Le seul attribut de cette classe est un vector de Protein nommé `data`, contenant des protéines (de la classe `Protein`). La fonction `insert_sort()` permet d'insérer un nouvel élément dans le tas et ensuite de réarranger le vector pour qu'il garde ses

propriétés de tas min. L'insertion a une complexité de $\mathcal{O}(\log n)$ pour un tas équilibré. La fonction `popMin()` permet de retirer le minimum du tas min. Sa complexité est en $\mathcal{O}(\log n)$. Il est donc intéressant d'utiliser un Tas min étant donné la grande taille de la base de donnée (564 000 séquences) puisqu'à chaque fois qu'il faut remplacer le minimum cela possède un coût de $\mathcal{O}(2 \log n)$.

La fonction `size()` permet simplement de renvoyer la taille du vector `data`. La Table 4 reprend la structure de la classe `Tas_min`.

`Tas_min`
Attributes `data :vector<Protein>`

Functions
`insert_sort(), getMin(), popMin()`
`size()`

TABLE 4 : Attributs et fonctions de la classe `Tas_min`

3 Algorithme d'alignement

3.1 Fonctionnement

L'algorithme utilisé pour l'alignement se base sur celui de Smith-Waterman. Ce dernier compare une séquence de requête de taille m avec une autre de taille n. Il fonctionne de la manière suivante.

Une comparaison est faite entre la sous-séquence (de taille i) de la protéine de requête avec celle (de taille j) de la seconde protéine.

Lors de cette comparaison, l'algorithme calcule le sous-score maximal en appliquant d'abord une pénalité sur la taille des sous-chaînes, avant d'y rajouter un score de similarité entre les résidus i et j de la protéine de requête et de la seconde respectivement.

Une fois le sous-score calculé, l'algorithme le compare avec le meilleur sous-score gardé en mémoire et mémorise le maximum.

L'expression mathématique du sous-score est la suivante :

$$H_{i,j} = \max[H_{i-1,j-1} + P[i,j], E_{i,j}, F_{i,j}] \tag{1}$$

où P[i,j] est la matrice donnant le score de similarité entre les résidus i et j,

$$F_{i,j} = \max_{1 \leq k \leq i}[H_{i-k,j} - w_k] \tag{2}$$

et

$$E_{i,j} = \max_{1 \leq k \leq j}[H_{i,j-k} - w_k] \tag{3}$$

En supposant que w_k soit un poids linéaire, [Gotoh 1982] les équations 2 et 3 peuvent être réécrites de sorte à obtenir l'expression suivante :

$$H_{i,j} = \begin{cases} \max \begin{cases} H_{i-1,j-1} + P[q_i, d_j] \\ E_{i,j} \\ F_{i,j} \\ 0 \end{cases} & \begin{vmatrix} i > 0 \\ \cap \\ j > 0 \end{vmatrix} \\ 0 & \begin{vmatrix} i = 0 \\ \cup \\ j = 0 \end{vmatrix} \end{cases}$$

$$E_{i,j} = \begin{cases} \max \begin{cases} H_{i,j-1} - Q \\ E_{i,j-1} - R \end{cases} & |j > 0 \\ 0 & |j = 0 \end{cases}$$

$$F_{i,j} = \begin{cases} \max \begin{cases} H_{i-1,j} - Q \\ F_{i-1,j} - R \end{cases} & |i > 0 \\ 0 & |i = 0 \end{cases}$$

$$S = \max_{1 \leqslant i \leqslant m \cap 1 \leqslant j \leqslant n} H_{i,j}$$

FIGURE 1 : Algorithme Smith-Waterman modifié [Rognes 2011]

où R et Q représentent respectivement le "gap extension penality" et la somme du "gap extension penality" et du "gap open penality". S représente le meilleur des sous-scores, et donc le score total une fois l'algorithme complètement exécuté. Enfin, $P[q_i, d_j]$ correspond au score de similarité entre les deux résidus comparés lors d'une itération quelconque.

3.2 Implémentation

L'implémentation de l'algorithme de Smith-Watermann peut se faire directement à partir de l'expression mathématique (Figure 1). Cependant, il existe une implémentation plus optimisée de l'algorithme [Rognes 2011], donnée à la Figure 2 où N correspond au stockage de H[i+1,j+1] pour un traitement ultérieur et P[q] au score de similarité.

```
paddsb P[q], H // H = H + P[q]
pmaxub F, H    // H = max(H, F)
pmaxub E, H    // H = max(H, E)
pmaxub H, S    // S = max(S, H)
psubsb R, F    // F = F - R
psubsb R, E    // E = E - R
movdqa H, N    // N = H
psubsb Q, H    // H = H - Q
pmaxub H, E    // E = max(H, E)
pmaxub H, F    // F = max(H, F)
```

FIGURE 2 : Implémentation améliorée pour une itération [Rognes 2011]

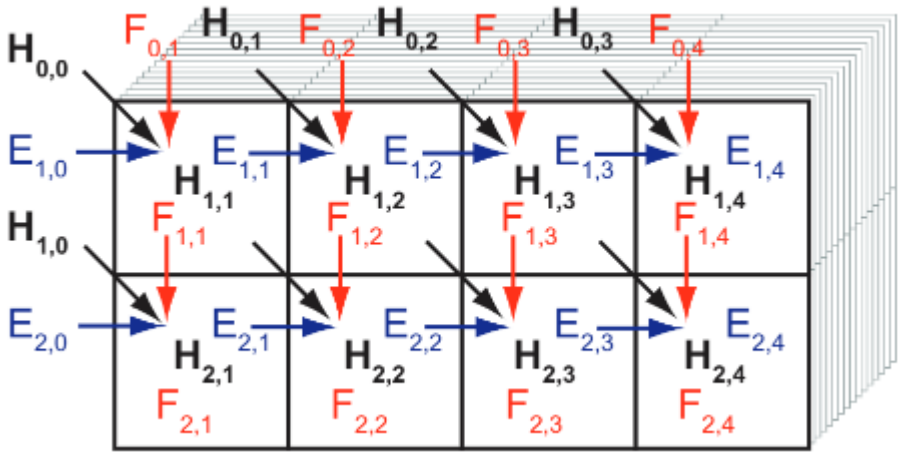

FIGURE 3 : Graphe de dépendance [Rognes 2011]

Cette version parcourt la même récurrence colonne par colonne, de sorte que moins de mémoire soit nécessaire pour le fonctionnement de l'algorithme (voir Sous-section 3.3).

Les 2 méthodes, algorithme de base et optimisé, ont été implémentées au code. La version optimisée est la fonction `Smith_Waterman()` et celle de base dans la fonction `Smith_Waterman2()`.

3.3 Coût de l'algorithme

L'algorithme de Smith-Waterman opère en $\mathcal{O}(N^3)$ en temps et $\mathcal{O}(N^2)$ en mémoire, avec $N = \max(n, m)$. Avec la réécriture de la récurrence, on peut descendre à une complexité en temps de $\mathcal{O}(N^2)$, étant donné qu'il est nécessaire d'exécuter m fois n itérations (voir Figure 1).

De plus, l'implémentation proposée permet de réduire la complexité mémoire en $\mathcal{O}(N)$. En effet, il est nécessaire de ne retenir qu'une valeur de F, 2m valeurs de H (les m valeurs de N incluses) et n valeurs de E pour chaque colonne parcourue (cf Figure 3).

Après avoir testé les deux implémentations, il y a en effet une énorme différence en terme d'utilisation de mémoire, cela permet de passer de 314 MB pour la version classique à 4 MB pour une très longue séquence.

4 Optimisation du programme

Afin d'optimiser le programme, dans la version `projetopt`, des threads ont été utilisés. D'après [Rognes 2011], le nombre de threads utilisés doit être égal ou inférieur au nombre de cœurs logiques de la machine. De nos jours, la plupart des ordinateurs possèdent 8 cœurs logiques, donc 8 threads sont utilisés.

Dans la version non optimisée, la fonction `search_top_20()` parcourt le fichier .psq avec un seul thread. Pour accélérer l'algorithme, une autre fonction `search_top_20opt()`

utilise 7 threads qui lisent la base de donnée à partir de 7 endroits différents (car il reste un thread principal). Pour trouver ces endroits, la taille de la base de donnée a simplement été divisée par 7. Chaque thread stocke les protéines au plus haut score dans la même variable `tas_min` qui est de la classe `Tas_min` et qui est commune à tous les threads.

Afin d'éviter les problèmes de concurrence, des mutex ont été créées pour réguler l'accès en lecture au fichier ainsi que l'accès en modification de la variable `tas_min`. Les mutex permettent de bloquer l'accès aux autres threads et de les mettre en attente lorsqu'un thread effectue déjà une opération sur l'objet (ici la variable `tas_min` ou le fichier .psq).

Le code du `projetopt` est dans les fichiers researchopt.cpp et projetopt.cpp.

5 Conclusion

Pour conclure, tous les objectifs du cahier des charges ont été atteints. Le makefile permet à l'utilisateur de compiler `projetprelim`, `projet` et `projetopt`. Le projetprelim ne renvoie que le header de la séquence de protéine reçue. Le projet renvoie les 20 séquences de la base de données qui sont les plus similaires à la séquence de requête et les scores de ces 20 séquences sont corrects.

L'algorithme de Smith-Waterman est donc fonctionnel et optimisé à l'aide de la méthode de Rognez.

Le code a été pensé de façon à limiter le coût en temps et en mémoire de l'exécution du programme. En effet, il utilise le moins de mémoire possible et le nombre d'opérations a été minimisé afin d'accélérer le programme.

NB : Le test automatique `./testfinal` ne fonctionne pas pour les protéines de recherche 2 et 4 bien que les scores obtenus soient les mêmes que ceux attendus. En effet, dans le résultat obtenu, les séquences possédant les mêmes scores ne sont pas forcément dans le même ordre que dans le résultat espéré. Cependant, il s'agit bien des 20 mêmes séquences et des 20 mêmes scores.

Références

FARRAR, Michael S., 2010. *NCBI BLAST Database Format.*

GOTOH, Osamu, 1982. *An Improved Algorithm for Matching Biological Sequences.*

ROGNES, Torbjorn, 2011. *Faster Smith-Waterman database searches with inter-sequence SIMD parallelisation.*

ROLAND, Jérémie, 2022. *Complément de programmation et d'algorithmique.*