ECOLE POLYTECHNIQUE DE BRUXELLES

BA3 IRCI - Informatique - Groupe 01

INFO-H304 : Compléments de programmation et d'algorithmique

Alignement de séquences de protéines avec l'algorithme de Smith-Waterman

Abdellaoui Sajid

Belokonskiy Alexandre

Najdi Louai

Jérémie Roland

2025

Table des matières

1 Introduction

Dans le cadre du cours de COMPLÉMENTS DE PROGRAMMATION ET D'ALGORITHMIQUE, il est demandé aux étudiants de développer un programme à l'aide du langage C++, permettant d'effectuer l'alignement local de séquences de protéines au moyen de l'algorithme de Smith–Waterman. Ce projet porte sur l'analyse d'une séquence de protéine fournie en la comparant à une large base de données existante. Cette comparaison est réalisée au moyen d'une mesure de similarité fondée sur l'utilisation d'une matrice de scores (BLOSUM 2.4), ainsi que sur l'application de pénalités d'ouverture et d'extension (gaps 3). Le programme est capable de manipuler des formats de fichiers courants en bioinformatique, tels que FASTA et BLAST.

La base de données de séquences protéiques fournie était initialement sous le format FASTA. Nous avons donc utilisé le programme NCBI BLAST+ pour la convertir et générer trois fichiers binaires optimisés (`.pin`, `.psq` et `.phr`).

2 Structure du programme

Étant donné que le projet est réalisé en `C++`, et vu la complexité de ce dernier, le code fut divisé en plusieurs classes. Celles-ci ont été implémentées à l'aide de fichiers `.h` et de fichiers compilables `.cpp`. Les fichiers `.h` ont été rangés dans un dossier `headers` et les fichiers `.cpp` dans un dossier `src`, assurant ainsi une structure de projet claire et modulable. Les fichiers `projetprelim.cpp`, `projet.cpp` et `projetopt.cpp` ont été mis à la racine du projet. Les sections ci-dessous abordent les différentes classes du projet.

2.1 dataPin

Cette classe, définie dans le fichier `blast.cpp`, stocke les informations importantes extraites du fichier binaire `.pin`.

Ce fichier agit comme un index et permet de récupérer trois informations clés :

— Le nombre total de protéines dans la base de données, stocké dans l'attribut `int numberofprot`.

— Les offsets indiquant les positions de départ et de fin pour la lecture des fichiers binaires `.psq` et `.phr`. Ces offsets sont stockés dans les attributs `vector<uint32_t> header_offsets` et `vector<uint32_t> sequence_offsets`.

Afin d'initialiser les attributs d'une instance de la classe, nous utilisons la fonction membre :

— `void dataPin::read_pin(const string& filepin)`

qui prend en argument le chemin du fichier `.pin`.

2.2 query

Cette classe, définie dans le fichier `fasta.cpp`, sert à représenter la protéine fournie en entrée avec 2 attributs `string id` et `string sequence`. Pour affecter ces attributs à une instance de la classe, nous utilisons la fonction membre :

— `void query::getIdandsequence(const string& filefasta)`

qui prend en argument le chemin d'un fichier FASTA ne contenant qu'une seule protéine.

2.3 Protein

Cette classe, définie dans le fichier `Protein.cpp`, sert à représenter une protéine de la base de données à travers trois attributs :

— `string id`

— `string sequence`

— `int sw_score` (initialisé à 0)

Pour définir l'identifiant (`id`) et la séquence (`sequence`), nous utilisons les fonctions :

— `string read_sequence(ifstream& file, const int a, const int b)`

— `string read_header(ifstream& file, const int a, const int b)`

Ces deux fonctions, définies dans le fichier `blast.h`, prennent respectivement en arguments :

— Les fichiers binaires `.psq` et `.phr` déjà ouverts.

— Les deux offsets (issus de la dataPin 2.1) pertinents qui permettent de localiser et de lire les informations.

L'attribut `int sw_score` est déterminé par la fonction :

— `int SWmatrix(const string& query_string, const string& prot_sequence, const Blosum& blosum, const int GOP, const int GEP)`

définie dans le fichier `SmithWaterman.cpp`, qui prend en argument la séquence de la protéine d'entrée, celle d'une protéine issue de la base de données dont on veut connaître le score, une instance de la classe `Blosum` 2.4, la pénalité d'ouverture de gap (`GOP`) ainsi que la pénalité d'extension de gap (`GEP`) et renvoie le score de la protéine cible sur base de l'algorithme de Smith-Waterman expliqué dans la section 3.

2.4 Blosum

La matrice de scores BLOSUM a été fournie sous forme de fichiers texte, il a donc fallu écrire des fonctions pour analyser le fichier texte et construire une matrice utilisable dans un code C++. La classe Blosum, définie dans le fichier `blosum.cpp`, a donc été créée pour rendre le tout plus simple. Cette classe possède les attributs suivants :

— `int size` représentant la taille de la matrice, elle n'est utile que pour déterminer les autres attributs.

— `vector<int> matrix` représentant la matrice de score formatée afin d'être réutilisable. Nous utilisons un vector 1D pour simuler une matrice 2D afin d'améliorer les performances.

— `unordered_map<char, int> indexMap` représentant une table de correspondance. Son rôle est de traduire chaque lettre d'acide aminé (comme 'A' ou 'E') en un indice numérique (comme 0 ou 7).

Ce dernier attribut est essentiel, car le fichier texte original utilise des lettres comme indices pour la matrice plutôt que des indices numériques, comme dans une matrice classique en C++. Il permet ainsi de trouver plus facilement le score correspondant à deux acides aminés grâce à la fonction membre `int Blosum::Score(char acide1, char acide2) const`, qui prend comme arguments deux acides aminés et renvoie leur score.
Le constructeur de la classe prend en argument le chemin du fichier texte de la matrice.

3 Implémentation de l'algorithme

Afin de comparer la similarité d'une séquence à une autre, il fallait implémenter l'algorithme de Smith-Waterman qui repose sur le principe de la programmation dynamique, consistant à calculer le score d'alignement optimal en décomposant le problème en sous-problèmes plus simples et en réutilisant les scores précédemment calculés. Cet algorithme permet ainsi de récupérer un score d'alignement local, à partir duquel il est possible de classer les 20 meilleures protéines issues de la base de données.
L'implémentation de l'algorithme s'est basée sur les références rendues avec le projet, en particulier [Rog11] où se trouvait l'explication de l'utilisation de l'algorithme. À l'aide de celui-ci, il est possible de comparer la séquence de la protein requête (classe `query`) avec celle d'une proteine issue de la base de données (classe `Protein`)
L'algorithme utilise trois matrices :

— $H_{i,j}$: score du meilleur alignement local se terminant par l'alignement des résidus q_i et p_j,

— $E_{i,j}$: score du meilleur alignement se terminant par un gap dans la séquence de q,

— $F_{i,j}$: score du meilleur alignement se terminant par un gap dans la séquence de p.

$P[q_i, p_j]$ représente le score d'alignement entre deux résidus q_i et p_j et est donné par la matrice des scores BLOSUM62 (voir section 2.4).
Les pénalités de gap sont définies comme suit :

$$Q = \text{GOP} + \text{GEP}, \qquad R = \text{GEP},$$

où GOP représente la pénalité d'ouverture de gap et GEP la pénalité d'extension.
Les relations de récurrence de l'algorithme reprises de [Rog11], page 4, sont alors :

$$H_{i,j} = \begin{cases} \max \begin{pmatrix} H_{i-1,j-1} + P[q_i, p_j] \\ E_{i,j} \\ F_{i,j} \\ 0 \end{pmatrix}, & i > 0 \text{ et } j > 0, \\ 0, & i = 0 \text{ ou } j = 0. \end{cases} \tag{1}$$

$$E_{i,j} = \begin{cases} \max \begin{pmatrix} H_{i,j-1} - Q \\ E_{i,j-1} - R \end{pmatrix}, & j > 0, \\ 0, & j = 0. \end{cases} \tag{2}$$

$$F_{i,j} = \begin{cases} \max \begin{pmatrix} H_{i-1,j} - Q \\ F_{i-1,j} - R \end{pmatrix}, & i > 0, \\ 0, & i = 0. \end{cases} \tag{3}$$

L'appel à la fonction suivante permet d'obtenir le score de comparaison entre la séquence requête et une protéine de la base de données :

— `int SMatrix(const query& query, const string& prot_sequence, const Blosum& blosum, const int GOP, const int GEP)`

Le code présente une boucle où i est incrémenté et à l'intérieur de celle-ci, une autre boucle où j est incrémenté. Le corps de cette double boucle correspond alors aux équations 1, 2 et 3.

Pour chaque nouvelle itération, `max_score` est remplacé par la valeur de $H[i, j]$ si celle-ci est plus grande que le score maximal local précédent. Finalement, on renvoie simplement cette valeur.

Optimisation mémoire

Puisque aucun backtracking[1] n'est nécessaire, il ne faut pas stocker l'intégralité des matrices H, E et F. Seules les données suivantes sont conservées en mémoire :

— une ligne précédente et une ligne courante de la matrice H,

— une ligne de la matrice E,

— une ligne pour la matrice F.

Cette approche permet de réduire la complexité mémoire de $\mathcal{O}(M.N)$ à $\mathcal{O}(N)$, tout en conservant la même complexité temporelle (M et N sont respectivement la taille de la séquence de protéine de requête et la séquence de protéine dont le score est demandé.

4 Structure de données

Le but du projet est d'appliquer l'algorithme de Smith-Waterman à chacune des protéines de la base de données, puis d'afficher les 20 meilleurs résultats dans le terminal.
Étant donné le nombre important de protéines à traiter (ici 573 661), nous devions choisir une structure de données capable de stocker nos instances de la classe `Protein` avec un coût d'insertion et de tri faible. Nous nous sommes naturellement tournés vers une file de priorité (priority queue), implémentée par un tas (heap).
Cette structure offre un coût d'insertion en $\mathcal{O}(\log n)$ et permet un tri final des n éléments en $\mathcal{O}(n \log n)$. De plus, le coût pour accéder au meilleur élément est en $\mathcal{O}(1)$ ([Rol25]).
Heureusement, la file de priorité est déjà implémentée dans la STL (`#include <queue>`), dont le cahier des charges autorise l'utilisation.
Pour utiliser cette structure de données de la STL, nous avons dû définir les opérateurs de comparaison (<, > et ==) dans la classe `Protein`. Ces opérateurs ont été basés sur l'attribut `int sw_score` (ou sur l'attribut `id` en cas de scores égaux) et définis dans le fichier `Protein.cpp`.

1. Le backtracking permet de reconstruire l'alignement local optimal en retraçant, depuis la case de score maximal, les cellules précédentes ayant contribué au calcul du score.

5 Optimisation

Lors de nos premiers tests, nous avons rapidement constaté que le temps de traitement augmentait rapidement avec la taille de la query et celle de la base de données. Malgré le tri efficace en $\mathcal{O}(n \log n)$ (où n est le nombre de protéines dans la base) l'algorithme de Smith-Waterman en lui-même possède une complexité en $\mathcal{O}(M \cdot N)$ (avec M la taille de la requête et N la taille d'une protéine cible). Appliquer l'algorithme à chaque protéine séquentiellement n'est donc pas une bonne stratégie, d'où la nécessité d'optimiser notre code.

5.1 Multithreading

La 1re optimisation possible est d'utiliser plusieurs threads. Un thread est un flot d'exécution indépendant au sein d'un même processus, partageant ses ressources mais avançant avec son propre état d'exécution ([BT19]). Nous avons donc écrit la fonction membre `static`[2] :

— `vector<Protein> Protein::initProtqueueMT(const string& phrfile, const string& psqfile, const dataPin& pin, const query& query, Blosum& blosum, int GEP,int GOP)`

Cette fonction possède un vector de file de priorité nommé all_thread_results. Cette fonction va créer le plus de thread possible (nombre donné par la fonction `std::thread::hardware_concurrency()`) et associé chacun de ces threads à la fonction membre `static` :

— `void Protein::computeSW(int start, int end, const query& query, const Blosum& blosum, const string& phrfile, const string& psqfile, const dataPin& pin, int GEP, int GOP, priority_queue<Protein>& thread_results)`

en leur associant à chacun un élément de all_thread_results par référence. Le rôle de `computeSW` est d'appliquer l'algorithme de Smith-Waterman à une partie de la base de données (les paramètres start et end définissent la plage de protéines que le thread doit analyser) et de mettre les 20 meilleurs protéines dans une file de priorité (chaque thread utilise la file de priorité qui lui a été associé par `initProtqueueMT`) . Une fois que tous les threads se sont terminés, `initProtqueueMT` fusionne toutes les files de priorité puis retourne la file résultante. De cette manière, on peut analyser plusieurs protéines en même temps, ce qui diminue grandement le temps de traitement de la base de données complète.

2. Une fonction membre statique en C++ est une fonction associée à une classe plutôt qu'à une instance, accessible sans créer d'objet

La fonction `initProtqueueMT` est utilisée dans le fichier `projetopt.cpp` alors que le fichier `projet.cpp` utililise la fonction membre, `static` aussi, `initProtqueue` qui parcourt la base de données séquentiellement.

5.2 SIMD et parallélisation

Une autre piste d'amélioration, et sans doute ce qui nous sépare d'un logiciel performant tel que SWIPE, est l'utilisation de la parallélisation et des registres SIMD. Tout ce qui est expliqué dans cette partie proviendra grandement de [Rog11], cette section introduit en partie ce qui est expliqué en détails dans cette source.

Parallélisation

Le programme actuel applique l'algorithme de Smith-Waterman à une séquence de protéine face à celle de la séquence de requête. Il est donc possible de paralléliser cette opération à plusieurs protéines d'un coup afin d'obtenir une optimisation du temps de recherche augmentée. Il serait idéal de comparer 16 séquences de protéines en même temps pour obtenir des performances optimales.

SIMD

Les SIMD, que chaque cœur du système dispose, (*Single Instruction, Multiple Data*) sont des unités vectorielles capables d'appliquer une même opération sur plusieurs données simultanément. L'implication de ces registres est donc presque indispensable afin d'obtenir des résultats optimaux lors d'une parallélisation.

5.3 Flags

L'option de compilation `-O3` active des optimisations inclues dans le compilateur permettant d'augmenter la vitesse d'exécution du programme. Ces optimisations correspondent entre autres à :

— **Inlining** : remplacement des appels de fonctions courtes par leur corps afin de supprimer le coût des appels et d'améliorer les performances.

— **Vectorisation automatique** : transformation des boucles afin d'exécuter plusieurs opérations en parallèle à l'aide d'instructions SIMD (à différencier avec l'utilisation des registres décrites 5.2).

— **Réordonnancement des instructions** : organisation des instructions afin d'optimiser l'utilisation du processeur.

(pour connaître toutes les optimisations, voir [GNU24]).

6 Utilisation de l'intelligence artificielle

L'intelligence artificielle a été partiellement utilisée lors de la conception du projet, notamment pour le débogage qui peut souvent s'avérer frustrant sans assistance. Elle a également été employée pour faciliter l'écriture du rapport en particulier lors de l'écriture des équations en LaTeX.

7 Conclusion

En conclusion, ce projet a permis de découvrir concrètement les différentes étapes nécessaires à la construction d'un projet informatique structuré. Il a mis en évidence l'importance de l'organisation du code, du choix de structures de données adaptées aux besoins, ainsi que du travail collaboratif sur une base de code commune via un dépôt git. Il a également permis de se familiariser avec la manipulation de fichiers binaires, une notion largement répandue en informatique, ainsi qu'avec les problématiques d'optimisation du temps d'exécution et de l'utilisation de la mémoire, encore perfectibles.
Les compétences acquises au cours de ce projet constituent une base utile pour la réalisation de projets futurs, tout en offrant un aperçu concret d'un domaine d'application lié à la bioinformatique.

Références

[BT19] BOUCHAREB, H. et J.-M. TORRES-MORENO (2019). *Introduction aux systèmes d'exploitation, cours et exercices en GNU/Linux*. Ellipses.

[Far10] FARRAR, Michael S. (2010). *NCBI BLAST Database Format*.

[GNU24] GNU PROJECT (2024). *GCC Optimization Options*. Free Software Foundation. URL : https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[Got82] GOTOH, Osamu (1982). « An improved algorithm for matching biological sequences ». In : *Journal of Molecular Biology* 162.3, p. 705-708.

[Kea25] KEA SIGMA DELTA (2025). *Endianness in Computing : What It Is, Why It Matters, and How to Handle It in C/C++*. YouTube video. URL : https://www.youtube.com/watch?v=Sh7hTquGybE.

[Rog11] ROGNES, Torbjørn (2011). « Faster Smith-Waterman database searches with inter-sequence SIMD parallelisation ». In : *BMC Bioinformatics* 12.1, p. 221.

[Rol25] ROLAND, Jérémie (2025). *Compléments de programmation et d'algorithmique – Cours 6 : Structures de données – Tas*.

[SW81] SMITH, Temple F. et Michael S. WATERMAN (1981). « Identification of common molecular subsequences ». In : *Journal of Molecular Biology* 147.1, p. 195-197.

[Wik25a] WIKIPEDIA (2025a). *Fasta format*. https://en.wikipedia.org/wiki/FASTA_format.

[Wik25b] — (2025b). *Proteinogenic amino acid*. http://en.wikipedia.org/wiki/Proteinogenic_amino_acid.