Université libre de Bruxelles
École Polytechnique de Bruxelles
B-IRCI3

INFO-H-304 - Compléments de programmation et d'algorithmique - 2022/2023

Projet

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

Groupe 1 :
Younes El Mokhtari
Brenno Ferreira
Ismail Jeq

Professeur :
Jérémy Roland

23 décembre 2022

Table des matières

1 Description de la structure du projet

Ce projet utilise trois différentes classes (Query, Database, Protein) et le fichier main.

1.1 Query

La classe Query permet d'extraire la protéine de requête du fichier de requête FASTA et la matrice de substitution du fichier BLOSUM62.

1.2 Database

La classe Database permet d'extraire les protéines se trouvant dans les fichiers BLAST (.pjs, .pin, .phr et .psq), d'aligner chaque protéine avec la protéine de requête et de sélectionner les 20 protéines les plus alignées.

1.3 Protein

La classe Protein permet de stocker chaque protéine et de calculer le score de sa protéine par rapport à la protéine de requête en utilisant l'algorithme de Smith-Waterman.

1.4 Main

Le fichier main fait appel à toutes les classes précédemment citées pour au final, afficher sur le terminal le score des 20 protéines les plus alignées avec la protéine de requête.

FIGURE 1 – Scores des 20 protéines les plus alignées affichés sur le terminal

2 Choix d'implémentation

Concernant les choix d'implémentation, le programme fonctionne de la façon suivante :

Dans le main, on récupère dans un premier temps les différents arguments donnés au programme : le fichier de la protéine de requête, la ficher de la base de données FASTA, le fichier BLOSUM62, le gap open penalty et le gap extension penalty.

La classe Query traite les fichiers FASTA et BLOSUM62 pour en faire respectivement un objet Protein et une matrice blosum. La classe Database quant à elle récupère la base de données FASTA et extrait à partir des fichiers BLAST toutes les protéines de la base de données. La fonction `alignProteins()` de Database permet ensuite d'aligner chaque protéine de la base de données avec la protéine de requête crée précedemment via la classe Query.

L'alignement des protéines utilise la méthode `calculate_score()` de Protein pour calculer le score de chaque protéine en fonction de la protéine de requête en utilisant l'algorithme de Smith-Waterman. Enfin, le ficher main affiche les 20 protéines ayant le plus gros score, après un tri effectué par la méthode `getMostAlignedProteins()` de Database.

```cpp
int main(int argc, char *argv[])
{
    // Arguments
    Query query_protein(argv[1]);
    Database database(argv[2]);
    Query query_blosum(argv[3]);
    int gap_open_penalty = stoi(argv[4]);
    int gap_extension_penalty = stoi(argv[5]);

    // Création de la matrice BLOSUM et de la protéine de requête
    vector<vector<int>> blosum = query_blosum.makeMatrixBlosum();
    Protein protein_query = query_protein.makeProteinFromQuery();

    // Alignement de la protéine de requête avec toutes les protéines de la base de données
    database.alignProteins(protein_query, blosum, gap_open_penalty, gap_extension_penalty);

    // Affichage des 20 protéines les plus alignées avec la protéine de requête
    vector<Protein> most_aligned_proteins = database.getMostAlignedProteins();
    for (int i = 0; i < most_aligned_proteins.size(); i++)
    {
        cout << most_aligned_proteins[i].getId() << " " << most_aligned_proteins[i].getScore() << endl;
    }
    return 0;
}
```

FIGURE 2 – Main

3 Algorithme de Smith-Waterman

Ce programme utilise l'algorithme de Smith-Waterman amélioré par Gotoh pour l'alignement de protéines présenté dans l'article discutant les méthodes pour accélerer cet algorithme. [Rog11].

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

$$E_{i,j} = \begin{cases} \max \begin{cases} H_{i,j-1} - Q \\ E_{i,j-1} - R \end{cases} & |j > 0 \\ 0 & |j = 0 \end{cases} \quad (2)$$

$$F_{i,j} = \begin{cases} \max \begin{cases} H_{i-1,j} - Q \\ F_{i-1,j} - R \end{cases} & |i > 0 \\ 0 & |i = 0 \end{cases} \quad (3)$$

$$S = \max_{1 \leqslant i \leqslant m \cap 1 \leqslant j \leqslant n} H_{i,j} \quad (4)$$

FIGURE 3 – Formulation mathématique de l'algorithme d'alignement de protéines de Smith-Waterman amélioré par Gotoh. La matrice $H_{i,j}$ est la matrice de scores, $E_{i,j}$ est la matrice de score en alignant le même acide aminé dans une position précédente de la séquence de requête et $F_{i,j}$ est celle respective à la séquence de la base de donnée. $P[q_i, d_i]$ est le score d'alignement donné par la matrice de substitution. Le paramètre Q est la somme des paramètres gap open penalty et gap extension penalty et R est le gap extension penalty.[Rog11]

Dans le projet, la fonction qui permet de générer le score de chaque protéine s'appelle `calculate_score()`. Cette fonction prends pour paramètre la séquence de la protéine de requête, la matrice de substitution BLOSUM62 et les valeurs de gap extension penalty et gap open penalty. Elle commence par déclarer :

- `matrix` : correspond à la matrice de score H dont les dimensions sont $M \times N$ où :

  M = longueur de la séquence de requête

  N = longueur de la séquence de la base de donnée
- `max_values_line` : contient le numéro de la colonne du plus grand score de la ligne actuelle
- `max_values_column` : contient le numéro de la ligne du plus grand score sur la colonne actuelle

Alors, la fonction itère deux fois via des for imbriqués en parcourant pour chacun élément de la séquence de requête tous les éléments de la séquence candidate afin de construire une ligne de la matrice de scores.

```cpp
/**
 * Cette fonction calcule le score d'une protéine aligné avec une protéine de requête via l'algorithme de Smith-Waterman
 * @param sequence La séquence de la protéine de requête
 * @param blosum La matrice BLOSUM
 * @param gap_open_penalty Le gap open penalty
 * @param gap_extension_penalty Le gap extension penalty
 */
void Protein::calculate_score(string sequence, vector<vector<int>> &blosum, int gap_open_penalty, int gap_extension_penalty)
{
    vector<vector<int>> matrix(sequence.size() + 1, vector<int>(this->sequence.size() + 1, 0));
    vector<int> max_values_lines(sequence.size() + 1, 0);
    vector<int> max_values_columns(this->sequence.size() + 1, 0);
    for (int line = 0; line < sequence.size() + 1; line++)
    {
        for (int column = 0; column < this->sequence.size() + 1; column++)
        {
            if (line == 0 | column == 0)
            {
                matrix[line][column] = 0;
            }
            else
            {
                // Algorithme de Smith-Waterman
                int h1 = matrix[line - 1][column - 1] + substitution_matrix(blosum, sequence[line - 1], this->sequence[column - 1]);
                int h2 = matrix[line][max_values_lines[line]] - gap_penalty(gap_open_penalty, gap_extension_penalty, column - max_values_lines[line]);
                int h3 = matrix[max_values_columns[column]][column] - gap_penalty(gap_open_penalty, gap_extension_penalty, line - max_values_columns[column]);
                int h4 = 0;
                matrix[line][column] = max(max(h1, h2), max(h3, h4));

                // On garde le plus gros score
                if (matrix[line][column] > score)
                {
                    score = matrix[line][column];
                }
```

FIGURE 4 – Implémentation de l'algorithme de Smith-Watermann en C++.

D'abord, le première condition à tester dans la boucle sert à tester si on se trouve dans la première ligne ou la première colonne de la matrice car il faut garantir qu'elles soient nulles pour respecter l'équation (1) de l'algorithme où :

$$H_{i,j} = 0 \text{ si } i = 0 \cup j = 0 \tag{1}$$

Sinon, on peut estimer les quatre candidates de score pour l'élément $H_{i,j}$ $\forall i, j > 0$.

h_1 : Premier candidat

Le premier candidat h_1 correspond à :

$$h_1 = H_{i-1,j-1} + P[q_i, d_j] \tag{2}$$

L'élément $H_{i-1,j-1}$ est facilement accessible via ses indices mais l'élément $P[q_i, d_j]$ s'obtient via la fonction `substitution_matrix()` qui parse le fichier texte de la matrice

BLOSUM62 et donc, pour chaque combinaison d'acide aminé qui est comparé, donne un score de similarité correspondant.

h_2 et h_3 : Second et troisième candidat

Le deuxième et le troisième candidats correspondent à un score si on alignait l'acide aminé avec un acide aminé précédent de la séquence de la protéine candidate ou de la séquence de protéine de requête, respectivement. Afin de déterminer h_2, il faut chercher le plus grand score sur la ligne en parcourant les colonnes et le soustraire le gap penalty, ici généré par la fonction `gap_penalty()` à l'aide de la formule $W_k = a \cdot k + b$ où W_k est le gap penalty total, a est la valeur du gap extension penalty, k est le décalage entre le plus grand score et l'élément actuelle et b est la valeur du gap open penalty. Pour trouver h_3, le problème est symétrique et donc, au lieu de chercher sur la ligne, il faut chercher sur les colonnes.

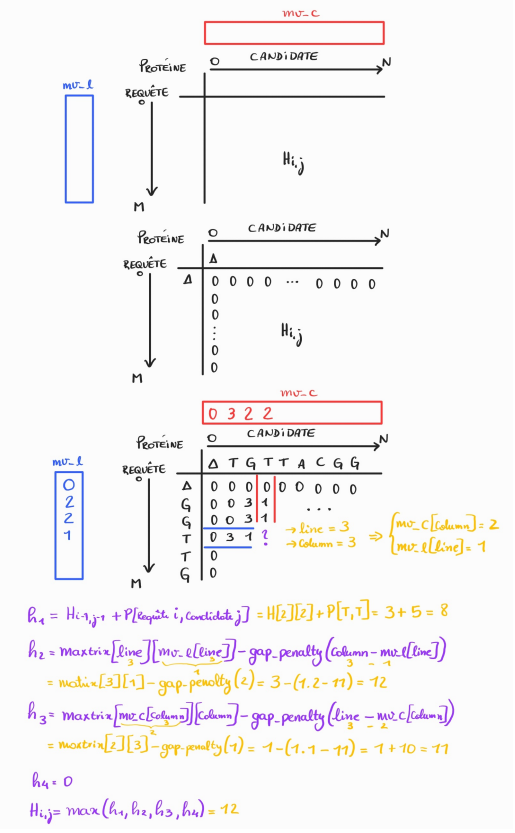


FIGURE 5 – Construction de la matrice H des scores d'alignement pas à pas pour un exemple donné.

L'équation devient donc :

$$h_2 = H_{i,j-1} - W_k | j > 0 \tag{3}$$

$$h_3 = H_{i-1,j} - W_k | i > 0 \tag{4}$$

On remarque que ceci diffère de l'algorithme présenté dans la figure 3 mais ce n'est qu'une façon plus concise d'écrire la même chose. En effet, la fonction *max* dans l'algorithme correspond à notre recherche de maximum sur la ligne ou la colonne et les deux paramètres Q et R à son intérieur sont rassemblée dans le paramètre W_k, qui est la pénalité linéaire introduite par Gotoh.

Cet algorithme est lent si pour chaque élément $H_{i,j}$ on vient parcourir toute la ligne et ensuite toute la colonne actuelle pour déterminer le score maximale pour effectuer le calcul. Afin de l'accélérer, les matrices `max_values_line` et `max_values_column` gardent l'indice colonne et l'indice ligne, respectivement, de l'élément maximal sur la ligne et colonne actuelle. Donc, pour générer le candidat h_2 de l'élément $H_{3,3}$, par exemple, on peut directement regarder l'élément d'indice 3 de la matrice `max_values_column` et cet élément sera la colonne où se trouve l'élément maximale sur la ligne 3 de H. Alors, on sait que l'élément $H_{3,1}$ est le plus grand score sur cette ligne. Cet exemple est donné et détaillé danss le schéma de la figure 5.

h_4 : Quatrième candidat et obtention du score

Le dernier candidat h_4 est nul afin d'éviter des valeurs négatives qui n'ont pas de sens dans un matrice de score. Alors, l'élément $H_{i,j}$ est la valeur maximale entre h_1, h_2, h_3 et h_4 donc, le plus grand score d'alignement possible pour un acide aminé donné. Comme le score d'alignement de toute la protéine est le plus grand score de toute la matrice, l'attribut score de l'objet protéine est modifié à la ligne 87 que si le nouveau score qui vient d'être généré est plus grand que le score lui-même.

Mise à jour des matrices max_values_lines et max_values_columns

À la fin de la génération d'un élément de la matrice $H_{i,j}$, on vient mettre à jour les matrices `max_values_line` et `max_values_column`. Pour mettre à jour `max_values_line`, on vient vérifier si le nouvel élément diminué de la plus petite pénalité, de décalage 1, est

supérieur au candidat h_2, qui le plus grand élément sur la ligne diminué de sa pénalité correspondante. Si c'est le cas, l'indice du nouvel vient remplacer l'indice de l'ancien pour cette ligne. La mise à jour de `max_values_column` se fait de façon symétrique en tenant compte des colonnes et en comparant avec h_3.

```
89
90              // On garde la colonne de l'élement maximal dans la ligne dans max_values_lines[line]
91              if (matrix[line][column] - gap_penalty(gap_open_penalty, gap_extension_penalty, 1) >= h2)
92              {
93                  max_values_lines[line] = column;
94              }
95
96              // On garde la ligne de l'élement maximal dans la colonne dans max_values_columns[column]
97              if (matrix[line][column] - gap_penalty(gap_open_penalty, gap_extension_penalty, 1) >= h3)
98              {
99                  max_values_columns[column] = line;
100             }
```

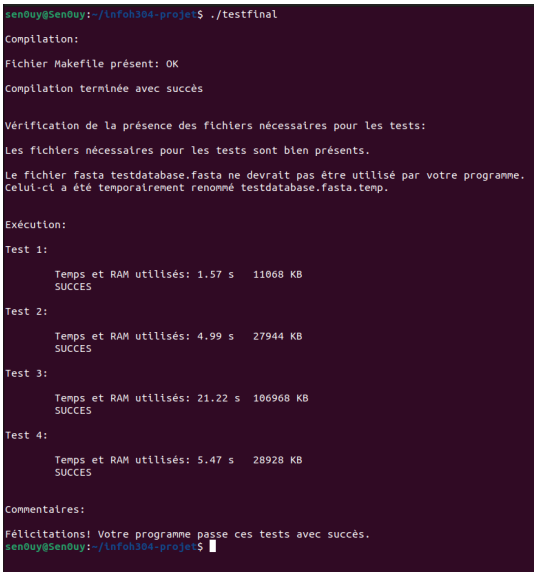
FIGURE 6 – Mise à jour des éléments de la matrice max_values_line et max_values_column.

4 Optimisation du projet

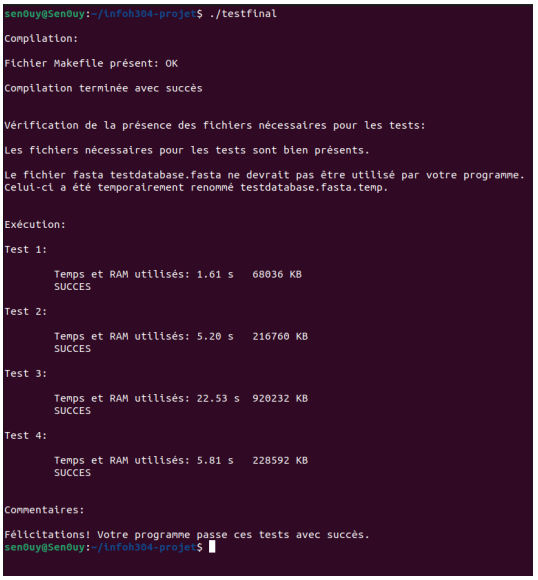
Pour la partie optimisation du projet, il est possible d'optimiser le code de différentes manières, en allant de la simple parallélisation (multi-threading) à des méthodes d'accélération plus sophistiquées en utilisant des instructions vectorielles sur le CPU ou d'autres processeurs basés sur le GPU. Dans notre cas, il a été opté de faire de la simple parallélisation en créant un thread par protéine pour le calcul de son score :

```
void Database::alignProteins(Protein protein_query, vector<vector<int>> &blosum, int gap_open_penalty, int gap_extension_penalty)
{
    searchNumberProteins();
    thread threads[NUMBER_PROTEIN];
    for (int index = 0; index < NUMBER_PROTEIN; index++)
    {
        int header_offset = getOffset(index, "header");
        int sequence_offset = getOffset(index, "sequence");
        string id = getProteinId(header_offset);
        string sequence = getProteinSequence(sequence_offset);
        Protein protein = Protein(id, sequence);
        protein_vector.push_back(protein);
    }

    for (int i = 0; i < NUMBER_PROTEIN; i++)
    {
        threads[i] = thread(align_thread, ref(protein_vector[i]), ref(protein_query), ref(blosum), gap_open_penalty, gap_extension_penalty);
    }

    for (int i = 0; i < NUMBER_PROTEIN ; i++) {
        if (threads[i].joinable()) {
            threads[i].join();
        }
    }
}

void Database::align_thread(Protein &protein, Protein &protein_query, vector<vector<int>> &blosum, int gap_open_penalty, int gap_extension_penalty) {
    protein.calculate_score(protein_query.getSequence(), blosum, gap_open_penalty, gap_extension_penalty);
}
```

FIGURE 7 – Utilisation de la parallélisation pour le calcul du score



(a) Résultats des tests (testfinal) pour projet



(b) Résultats des tests (testfinal) pour projetopt

FIGURE 8 – Résultats des tests

En comparant les vitesses d'exécution pour les tests entre projet et projetopt, on obtient des résultats assez similaire bien que l'on utilise un seul thread pour projet et plusieurs threads en parallèle pour projetopt. Une façon d'améliorer la vitesse d'exécution totale serait d'adopter la "striped approach" par Farrar pour le calcul du score qui semble beaucoup plus rapide que celle implémenté actuellement (colonne par colonne) dans le code[Rog11].

Références

[Rog11] Torbjørn ROGNES. "Faster Smith-Waterman database searches with inter-sequence SIMD parallelisation". en. In : *BMC Bioinformatics* 12.1 (déc. 2011), p. 221. ISSN : 1471-2105. DOI : 10.1186/1471-2105-12-221. URL : https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-12-221 (visité le 05/12/2022).