Conception d'un programme de threading par double programmation dynamique

Projet de programmation avancée

Gloria BENOIT
M2 BI

Université Paris Cité
BQ4CY010 Programmation 3 et projet tuteuré
12/09/2024

Introduction

La séquence et la structure 3D d'une protéine représentent deux informations critiques pour la compréhension de sa fonction. Malgré le lien étroit entre ces derniers, on observe souvent que la structure d'une protéine est mieux conservée que sa séquence (Illergård et. al. 2009, [1]). Des mutations ont donc moins d'effets sur la séquence que sur la structure, preuve que l'information 3D d'une protéine est primordiale à sa fonctionnalité.

Il est maintenant relativement facile d'obtenir la séquence d'une protéine, notamment par les différentes méthodes de séquençage existantes, alors que sa structure est plus difficile à résoudre (Büyükköroğlu et. al. 2018, [2]). Il existe de nombreuses techniques de résolution de la structure d'une protéine, mais elles présentent des contraintes temporelles et qualitatives, notamment au niveau de la résolution.

Pour pallier cela, des approches comparatives ont été mises en place, basées sur l'utilisation de structures déjà résolues expérimentalement comme point de départ (Lam et. al. 2017, [3]). Il en existe deux types : les méthodes d'enfilage (ou *threading*), et les méthodes de modélisation par homologie. Nous nous concentrons ici sur le premier type, qui permet de tester plusieurs repliements structuraux pour une séquence, afin de déterminer celui qui conviendrait le mieux.

Le projet consiste à réimplémenter l'algorithme *THREADER* (Jones 1998, [4]), un algorithme d'enfilage basé sur la double programmation dynamique, sous python.

Matériels et méthodes

Fonctionnement de l'algorithme

L'algorithme repose sur le test de chaque résidu d'une séquence, appelée *query*, sur chaque position d'une structure, appelée *template*. Pour chacune de ses étapes, on construit une matrice dite *Low Level*, qu'on rempli par programmation dynamique selon l'algorithme classique d'alignement global (Needleman & Wunsch 1970, [5]). Dans cette matrice, on bloque le résidu à la position spécifiée. De ce fait, il n'est pas nécessaire de remplir l'entièreté de la matrice, étant donné que certaines paires résidu/position sont impossibles à observer.

Pour remplir les matrices *Low Level*, on utilise comme score les potentiels statistiques DOPE (*Discrete Optimized Protein Energy*, Shen & Sali 2006, [6]). Pour une position donnée, on va déterminer sa distance à la position figée et en déduire le score DOPE associé. Pour faciliter les calculs, on considère uniquement les carbones alpha du *template*. La manière de remplir une matrice *Low Level* est donc la suivante:

$$L_{(i,j)} = \begin{cases} L_{(i-1,j-1)} + DOPE_{(i,j)} \\ L_{(i-1,j)} + GAP \\ L_{(i,j-1)} + GAP \qquad \text{avec GAP} = 0 \end{cases} \tag{1}$$

On retient le score de la dernière case de chaque matrice *Low Level*, qu'on stocke dans une nouvelle matrice. Cette matrice est la matrice de score utilisée dans la construction de l'unique matrice *High Level*. Pour obtenir l'alignement final entre notre séquence *query* et notre structure *template*, on ajoute une colonne et une ligne supplémentaire en début de matrice. On la remplit ensuite de la manière suivante:

$$H_{(i,j)} = \begin{cases} H_{(i-1,j-1)} + Lscore_{(i,j)} \\ H_{(i-1,j)} + GAP \\ H_{(i,j-1)} + GAP \qquad \text{avec GAP} = 0 \end{cases} \tag{2}$$

Une fois la matrice *High Level* remplie, on retrace le chemin optimal pour déterminer l'enfilage de la séquence *query* sur la structure *template*. Etant donné que nous travaillons sur des énergies, nous cherchons à minimiser le score final de l'alignement. Ainsi, plus le score est bas et plus la séquence se place bien sur la structure. Les étapes majeurs de l'algorithme sont illustrées en figure 1.

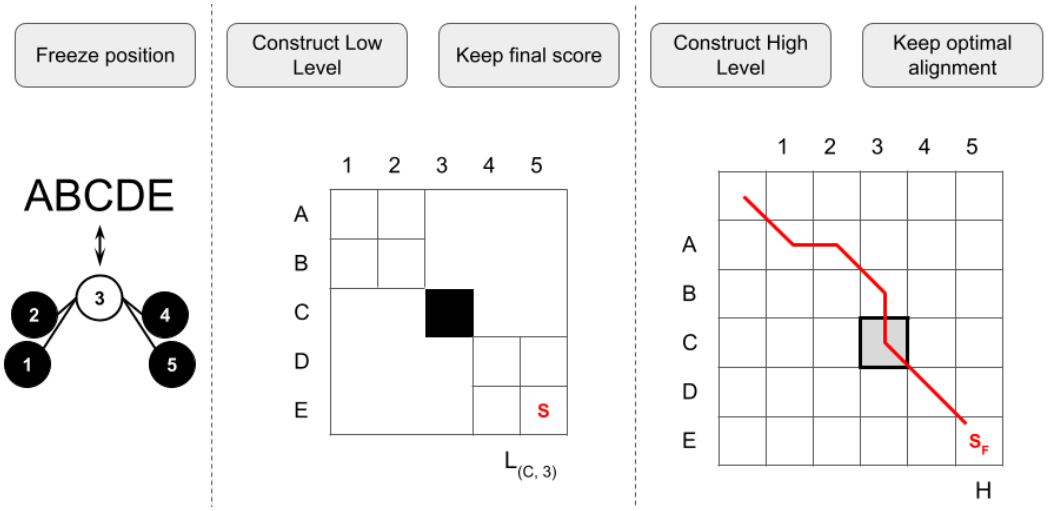

Figure 1: **Principe de l'algorithme.** $L_{(i,j)}$ fait référence à la matrice *Low Level* où le résidu i est figé en position j, et H fait référence à la matrice *High Level* unique.

Implémentation *Python*

L'algorithme a été entièrement conçu sous *Python* (version 3.12.5, Van Rossum & Drake Jr 1995, [7]), l'aide des librairies *sys*, *pathlib* et *multiprocessing*, innées à Python, ainsi que les librairies *numpy* (version 2.1.1, Harris et. al. 2020, [8]) et *pandas* (version 2.2.2, McKinney 2010, [9]).

Pour faciliter la généralisation du code, ainsi que sa robustesse, j'ai utilisé la programmation orientée objet. Ainsi, j'ai construit 5 classes, dont les relations sont présentées en figure 2.

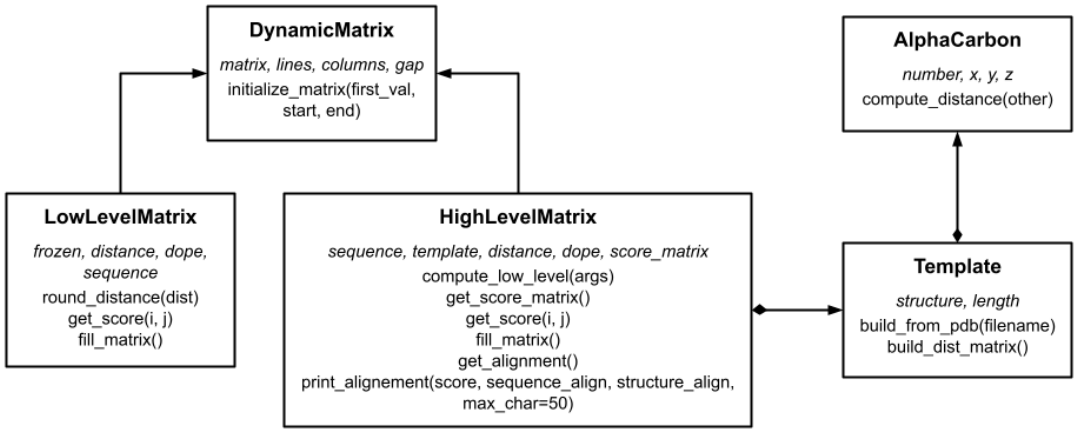

Figure 2: **Architecture des classes.** Les noms de classe sont en gras, suivi des attributs en italique puis des méthodes. Les flèches simples font référence à la notion d'héritage, et les flèches avec un losange à la base font référence à la notion de composition.

Pour augmenter la vitesse d'exécution, j'ai aussi parallélisé la construction des $n \times m$ matrices *Low Level*, avec n la taille de la séquence *query* et m la taille de la structure *template*.

L'algorithme prends en entrée une structure PDB, le *template*, ainsi qu'une ou plusieurs séquences *query*. Ce choix de tester un *template* pour plusieurs *query* est une méthode de repliement inverse, qui évalue la compatibilité d'une structure donnée avec un jeu de séquences. Il a été fait de manière à simplifier le temps

de calcul, puisque la matrice de distance du *template* n'a pas besoin d'être recalculée à chaque fois.

Une fois l'algorithme fini, il sauvegarde un fichier *ddt_[code du template].csv* de 4 colonnes, ordonné selon le score croissant :

- *QUERY*: Le code de la protéine utilisée
- *MAX_SCORE*: Le score optimal obtenu
- *ALIGN_SEQ*: La séquence alignée sous forme de liste
- *ALIGN_STRUCT*: La structure alignée sous forme de liste

Performances

Temps d'exécution

Pour enfiler une séquence *query* de taille n sur une structure *template* de taille m, il est nécessaire de construire $n \times m$ matrices *Low Level*, qu'il faut respectivement remplir sur $n \times m$ cases. On a donc une complexité de l'ordre de $O(n^2 \times m^2)$, en négligeant la construction de la matrice *High Level*. Ainsi la vitesse de l'algorithme est entièrement dépendante des tailles de la séquence *query* et de la structure *template*. Cela est illustré en figure 3.

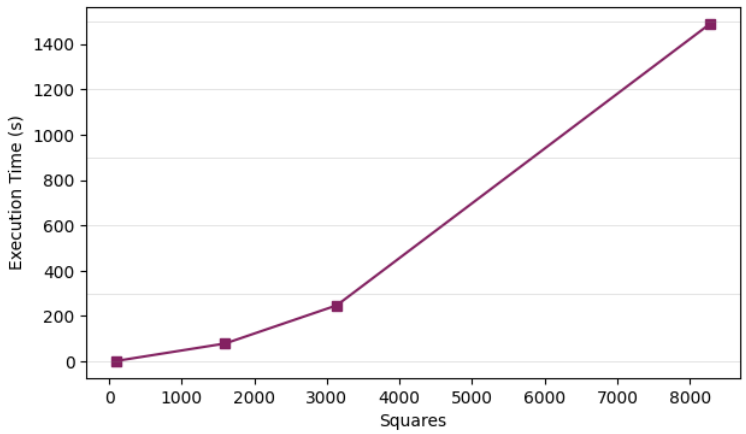

Figure 3: **Dépendance du temps d'exécution.** Le temps d'exécution est en fonction du nombre de cases à remplir pour une *query* et un *template* donnés. Un trait horizontal est marqué toutes les 5 minutes.

Petites protéines

La vitesse d'exécution dépendant fortement de la taille des protéines, j'ai d'abord testé mon algorithme sur des petites protéines d'environ 50 acides aminés:

- Hélices uniquement: 1BW6 (56 aa), 1BA4 (40 aa)
- Brins uniquement: 1AFP (51 aa), 1APF (49 aa)
- Hélices et brins: 1AYJ (51 aa), 1BK8 (50 aa)

Chacune de ces protéines a servi de *template* aux séquences de toutes les autres, y compris elle-même. On s'attends donc à ce que sa propre séquence ai le meilleur score, suivi de celle de structures secondaires similaires. Cependant, les résultats obtenus ont été très différents de ceux obtenus.

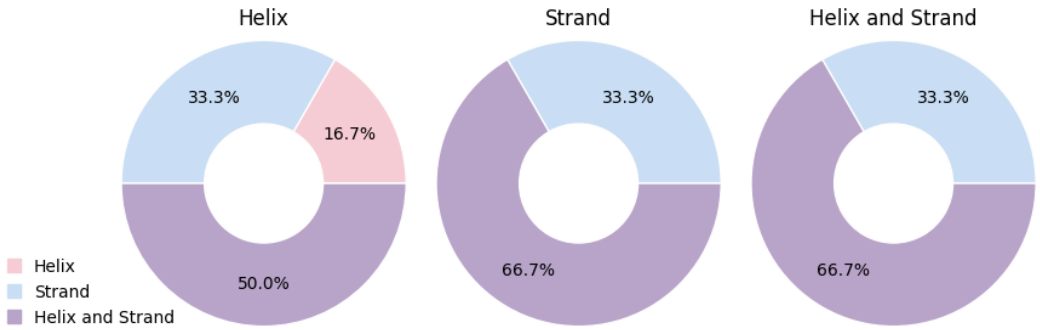

Figure 4: **Structures secondaires majoritaires observées chez les trois meilleures protéines pour chaque structuration du *template*.** Le rose représente les hélices, le bleu les brins et le violet la combinaison des deux.

Seulement deux protéines, 1AFP et 1AYJ, sur les six testées ont retrouvé leur propre séquence comme étant la meilleure. Pour résumer les résultats obtenus, on regarde pour chaque structure secondaire la structuration principale des trois meilleures protéines déterminées par l'algorithme, présenté en figure 4.

On observe ainsi que les trois premières protéines proposées par l'algorithme sont souvent des protéines structurées en hélices et en brins. On peut donc en déduire que l'algorithme n'a pas bien fonctionné sur ces protéines.

De plus, il est important de mentionner que la protéine 1BA4 a eu le pire score avec sa propre séquence. Etant la plus petite séquence, on peut supposer que cette terrible performance est due à la différence de taille. Notre algorithme possède donc un biais vis à vis de la taille des séquences comparée à celle de la structure *template*.

Grandes protéines

On peut aussi penser que notre algorithme fonctionne moins bien sur des petites protéines, car elles sont moins structurées que des grandes protéines. Cependant, comme explicité précemment, le temps d'exécution augmente fortement selon la taille des protéines utilisées. J'ai donc testé à nouveau mon algorithme sur des protéines plus grandes, d'environ 90 acides aminés:

- Hélices uniquement: 1A1W (91 aa), 1AB3 (88 aa)
- Brins uniquement: 1AE2 (87 aa)
- Hélices et brins: 1AB7 (89 aa)

Je n'ai utilisé que 1A1W comme *template*, et ai enfilé les séquences de toutes les autres protéines, y compris elle-même. On s'attend à retrouver sa propre séquence ayant le meilleur score, suivi de celle de 1AB3, puis 1AB7 et enfin 1AE2. Les résultats obtenus sont présentés en table 1.

QUERY	MAX_SCORE
1A1W	-861.42
1AB7	-858.92
1AB3	-841.84
1AE2	-829.80

Table 1: **Scores obtenus en sortie d'algorithme avec 1A1W comme *template*.**

Les résultats obtenus sont identiques à ceux attendus à l'exception du score de 1AB3, qu'on pensait être le second meilleur. Ce dernier étant plus grand que 1AE2, la protéine ayant obtenu un score inférieur, on peut

écarter la théorie que cela est entièrement du à la différence de taille. Les résultats sont donc meilleurs que sur des protéines plus petites, mais ne sont pas encore exactement au niveau attendu.

Pistes d'amélioration

L'algorithme implémenté est différent de l'algorithme original sur plusieurs points.
En effet, nous utilisons l'algorithme classique de Needleman & Wunsch, là où Jones ajoute le score à chaque case, avant d'y ajouter le minimum des trois cases alentours. C'est un premier axe d'amélioration de l'algorithme pour obtenir de meilleurs résultats.
Aussi, la pénalité de gap est spécifique à la structure secondaire observée, alors que nous avons fait le choix d'utiliser une pénalité de gap de 0. Il faudrait tester différentes valeurs de gap, selon les structures secondaires pour déterminer celles qui fonctionnent le mieux.
Enfin, après la construction d'une matrice *Low Level*, nous stockons uniquement la valeur finale, alors que Jones conserve la somme de toutes les valeurs du chemin optimal.

Outre ces différences avec *THREADER*, il existe aussi des points d'amélioration sur mon propre algorithme. Par exemple, il faudrait pouvoir attribuer un score DOPE à un acide aminé inconnu noté X. Il serait aussi intéressant de tester les différents modèles d'un fichier PDB, s'il en possède plusieurs. Mon algorithme ne récupère que les informations du premier, pour faciliter le temps de calcul.

Bibliographie

[1] K. Illergård, D. H. Ardell, and A. Elofsson, "Structure is three to ten times more conserved than sequence—a study of structural response in protein cores," *Proteins: Structure, Function, and Bioinformatics*, vol. 77, no. 3, pp. 499–508, 2009.

[2] G. Büyükköroğlu, D. D. Dora, F. Özdemir, and C. Hızel, "Chapter 15 - techniques for protein analysis," in *Omics Technologies and Bio-Engineering* (D. Barh and V. Azevedo, eds.), pp. 317–351, Academic Press, 2018.

[3] S. D. Lam, S. Das, I. Sillitoe, and C. Orengo, "An overview of comparative modelling and resources dedicated to large-scale modelling of genome sequences," *Acta Crystallogr D Struct Biol*, vol. 73, pp. 628–640, July 2017.

[4] D. T. Jones, "THREADER: Protein Sequence Threading by Double Dynamic Programming," in *Computational Methods in Molecular Biology* (S. Salzberg, D. Searls, and S. Kasif, eds.), ch. 13, Elsevier Science, 1998.

[5] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, 1970.

[6] M.-Y. Shen and A. Sali, "Statistical potential for assessment and prediction of protein structures," *Protein Sci*, vol. 15, pp. 2507–2524, Nov. 2006.

[7] G. Van Rossum and F. L. Drake Jr, *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[8] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, pp. 357–362, Sept. 2020.

[9] W. McKinney, "Data structures for statistical computing in python," in *Proceedings of the 9th Python in Science Conference* (S. van der Walt and J. Millman, eds.), pp. 51 – 56, 2010.