Ensimag 2A
2020-2021

Applications de la science des données et de l'intelligence artificielle :

Algorithmes de programmation dynamique pour l'alignement de séquences

Compte-rendu et archive tgz du projet à transmettre avant le 30 mars 2021 via TEIDE.

1. Présentation

En bio-informatique, l'alignement par paires est une méthode permettant de représenter deux séquences de macromolécules biologiques (ADN, ARN ou protéines) l'une sous l'autre de manière à identifier les régions homologues ou similaires, et de quantifier leur similarité. Un alignement peut être global s'il aligne obligatoirement les deux séquences dans leur intégralité, ou local s'il aligne des sous-chaînes des séquences d'intérêt. Des insertions ou des délétions (indels) peuvent être introduites dans les deux séquences pour améliorer l'alignement. Par exemple, pour les deux courtes séquences d'ADN GATTACA et ATAC:

global	local
GATTACA || || -AT-AC-	(gat) TAC (a) ||| (a) TAC

Selon l'origine des séquences, un alignement global peut être très pénalisant alors que des sous-séquences s'alignent localement très bien. C'est le cas de l'exemple ci-dessous : l'alignement global (haut) est mauvais alors que l'alignement local (bas) est excellent. Pour la plupart des applications on réalisera donc plutôt un alignement local.

```
--T--CC-C-AGT--TATGT-CAGGGGACACG--A-GCATGCAGA-GAC
  |  || |  ||  | | | |||    || |  | |  | ||||   |
AATTGCCGCC-GTCGT-T-TTCAG----CA-GTTATG--T-CAGAT--C

                  tccCAGTTATGTCAGgggacacgagcatgcagagac
                     ||||||||||||
aattgccgccgtcgttttcagCAGTTATGTCAGatc
```

L'algorithme de Needleman-Wunsch (1970), basé sur la programmation dynamique, permet de

réaliser des alignements globaux de manière optimale. L'algorithme de Smith-Waterman (1981) en est fortement inspiré et permet quant à lui d'obtenir des alignements locaux optimaux. Ses performances ne permettent pas forcément de chercher des séquences similaires à une séquence d'intérêt dans un génome complet – pour ce type de tâche on se tourne vers des algorithmes heuristiques comme BLAST, qui ne garantissent pas de trouver les alignements optimaux mais sont très rapides. En revanche Smith-Waterman reste très utilisé dans des scénarios où les séquences à aligner sont en nombre et taille raisonnables, ou simplement pour peaufiner des alignements à partir de régions similaires identifiées de manière heuristique.
L'objectif de ce mini-projet est d'implémenter en C l'algorithme de Smith-Waterman, puis une extension de cet algorithme (algorithme de Altschul & Erickson 1986, amélioration de l'algorithme de Gotoh 1982) qui permet d'utiliser une fonction de pénalité affine plutôt que linéaire pour les insertions-délétions.

2. Algorithme de Smith-Waterman

L'algorithme de Smith-Waterman a été vu en cours, vous pouvez vous référer aux diapos pour illustration. Pour aligner deux séquences *s1* et *s1* de longueurs *m* et *n*, on construit une matrice de *m*+1 lignes et *n*+1 colonnes. La case (i,j) contiendra le score S(i,j) du meilleur alignement se terminant au $i^{ème}$ symbole de *s1* et au $j^{ème}$ symbole de *s2*, ainsi que l'identité de la ou des case(s) "prédécesseur" qui ont permis d'obtenir ce meilleur alignement (si son score est positif). Cette ou ces cases sont forcément parmi les trois prédécesseurs possibles de (i,j): (i,j-1), (i-1,j), (i-1,j-1) selon qu'on a inséré un symbole de *s1*, inseré un symbole de *s2*, ou aligné un symbole de *s1* avec un symbole de *s2* (match ou mismatch).
Pour remplir cette matrice, on l'initialise avec des zéros dans la première ligne et dans la première colonne. On parcourt ensuite la matrice de gauche à droite et de haut en bas, et on remplit chaque case en considérant les quatre possibilités : prolonger un alignement en provenance d'un des trois prédécesseurs possibles de la case, ou repartir à zéro si les trois scores obtenus sont négatifs. Plus formellement, on calcule le score S(i,j) ainsi :

```
S(i,j) = max{ S(i-1,j-1) + subst(s1[i-1],s2[j-1]) ,
              S(i-1,j) + indel ,
              S(i,j-1) + indel ,
              0 }
```

Une fois la matrice remplie, on identifie les cases de score maximal : il s'agit des positions terminales dans *s1* et *s2* des alignements optimaux. Pour construire les alignements optimaux, il ne reste qu'à remonter dans la matrice depuis chacune de ces cases (vers le coin haut-gauche) en

suivant les liens vers les prédécesseurs retenus, jusqu'à arriver à une case sans prédécesseur.
Dans un premier temps vous vous contenterez de construire et afficher un des alignements optimaux (plutôt que tous) : les afficher tous est délicat et nécessite un algorithme récursif (pour quand une case a plusieurs prédécesseurs), cela constitue l'extension 2 (objectif 4E). Je recommande d'implémenter Altschul-Erickson (objectif 4B) avant.
Un paramètre important est la *fonction de coût*. En effet, quand on parle d'alignement optimal, il s'agit d'optimalité selon la fonction de coût choisie. Une telle fonction doit définir les coûts (ou scores) :

1. des substitutions, quand on aligne ensemble un symbole de chaque séquence, qu'il s'agisse d'un match (même symbole) ou d'un mismatch (symboles différents). Le score d'un match sera positif alors que celui d'un mismatch sera généralement négatif. Il est donc nécessaire de définir le coût de chaque paire de symboles. Pour l'ADN on peut employer une fonction simple, par exemple +5 pour un match et -4 pour un mismatch comme proposé dans *aliCost.c*.
2. des indels, quand un symbole d'une séquence est aligné avec un trou dans l'autre séquence (représenté par un '-'). Le coût d'une indel est forcément négatif, mais plusieurs modèles peuvent être considérés. Le plus simple est le modèle à coût linéaire: chaque indel a le même coût (par exemple -10 pour l'ADN). Le coût d'une indel de longueur N est donc -10N, d'où le nom « coût linéaire ». C'est ce modèle qui est considéré dans l'algorithme de Smith-Waterman, et que vous implémentez dans un premier temps (objectif 4A). Malheureusement il n'est pas toujours pertinent... Par exemple si on aligne un ARN épissé sur un fragment génomique, on s'attend à une longue insertion-délétion (insertion dans le fragment génomique, ou délétion dans l'ARN, ce qui est identique) pour chaque intron. Il est donc préférable que les indels soient regroupées en blocs dans les alignements. Il faut alors utiliser plutôt un modèle affine de coût, où l'ouverture d'une indel coute cher mais où l'extension d'une indel existante est peu pénalisée. Par exemple dans *aliDnaAffineLongGaps.c* les coûts d'ouverture et d'extension sont respectivement -100 et -0.05. Pour généraliser Smith-Waterman au modèle affine, il est nécessaire d'utiliser trois matrices plutôt qu'une seule. L'algorithme correspondant (algorithme de Altschul & Erickson) est décrit ci-dessous et sera implémenté dans un second temps (objectif 4B).

3. Organisation du travail

Vu les contraintes de temps et pour ceux qui sont un peu rouillés en C, je vous propose un

découpage en modules ainsi qu'une définition des principales structures de données. Je recommande fortement de les utiliser. Les principaux modules sont :

- *aliCalcSW* : gère et remplit la matrice de scores.
- *aliOutSW* : gère l'affichage. Ce module effectue en particulier le parcours dans la matrice remplie afin d'afficher les alignements optimaux : il s'agit de partir de chaque position terminale d'un alignement optimal, et de remonter vers le coin haut-gauche en suivant les traces laissées lors du remplissage, jusqu'à arriver à une case de score nul (c'est le début de cet alignement optimal). Attention, ce module n'est pas forcément plus simple que *aliCalcSW*...

De plus, les modules suivants sont fournis:

- *aliCost* : gère les pénalités de substitutions et d'indels, via un struct unique qui peut servir de manière homogène pour ADN et protéines, et pour un modèle linèaire ou affine pour les indels. L'implémentation est un peu lourde (saisie de BLOSUM62 pour les protéines), je vous en fais cadeau.
- *aliGetSeq* : saisie des sequences (ADN ou protéines) sur STDIN.
- *mem* : petite fonction pour l'allocation de mémoire.

Enfin, plusieurs fichiers avec un *main* sont proposés à titre d'exemples : *aliDnaLinear.c*, *aliDnaAffine.c*, *aliDnaAffineLongGaps.c*, *aliProtAffine.c*, et les exécutables correspondants obtenus avec mon implémentation sont fournis (**_ntm*) pour vous servir de point de comparaison.

Par ailleurs, une remarque concernant *aliOutSW* : vous avez compris qu'on part d'une case de score max dans la matrice et qu'on remonte jusqu'à trouver une case sans prédécesseur. On "remonte" donc dans nos strings d'entrée, mais au final le print doit évidemment afficher les séquences dans le bon sens. Plutôt que de chercher des solutions un peu complexes (construire des listes chainées, mémoriser les chemins, inverser les strings…), je propose la piste suivante : une string c'est un tableau de char terminé par le char '\0', autrement dit une string est un pointeur vers un char. On peut donc allouer un gros tableau de char, le remplir à l'envers en partant de la fin, s'arrêter quelque part dans le tableau, et le pointeur vers ce "quelque part" est lui-même une string. L'archive contient le petit code *printStrings.c* qui illustre cette approche, vous pouvez vous en inspirer.

L'archive est disponible ici :
http://membres-timc.imag.fr/Nicolas.Thierry-Mieg/teaching_ASDIA_2021.html

Par ailleurs vous maîtrisez sûrement **git** pour travailler efficacement à plusieurs. Et je recommande toujours d'utiliser **valgrind** (« pour nous les fainéants qui préfèrent trouver leurs bugs en 3s ») avant de plonger dans **gdb**!

4. Objectifs et progression

4A. Algorithme de Smith-Waterman.

Comme indiqué, je vous demande initialement de traiter les séquences d'ADN, avec un modèle de coût linéaire pour les indels, et de construire et afficher l'un des alignements optimaux.

4B. Algorithme de Altschul & Erickson.

Extension au modèle de coût affine des indels (cout d'ouverture o, cout d'extension e, o <= e < 0 ; si o=e on retrouve le modèle à coût linéaire).
Etendez votre programme pour utiliser un modèle affine de coût. Vous devrez créer de nouveaux modules *aliCalcAE* et *aliOutAE* , analogues aux modules **SW*. C'est un bon moment pour prendre le temps de réfléchir en équipe...

Pour étendre Smith-Waterman au modèle de coût affine, la solution proposée (algorithme de Altschul & Erickson) consiste à définir trois matrices analogues à la matrice utilisée précédemment:
- la première, D, contient les meilleurs scores ainsi que les pointeurs de backtrack, sachant que le dernier déplacement était diagonal (substitution == match ou mismatch);
- la seconde, V, sachant que le dernier déplacement était vertical (insertion d'un gap dans la séquence *s2*);
- la troisième H sachant que le dernier déplacement était horizontal (gap dans *s1*).

Le remplissage s'effectue ainsi:
D(i,j) = max[0, D(i-1,j-1)+s, V(i-1,j-1)+s, H(i-1,j-1)+s] (avec s le score de la substitution)
V(i,j) = max[0, D(i-1,j)+o, V(i-1,j)+e, H(i-1,j)+o]
H(i,j) = max[0, D(i,j-1)+o, V(i,j-1)+o, H(i,j-1)+e]
Les alignements optimaux commencent et terminent forcément par un match (ou un mismatch de score positif), on les trouvera donc dans la matrice D. De plus on peut remplacer les 0 par -1 dans V et H pour éviter d'afficher trois fois le même alignement.
Les structures de données proposées (*struct cell* dans *aliCalcSW.h*) devront donc être modifiées dans la nouvelle version *aliCalcAE.h*. Une solution consiste à stocker 3 scores plutôt qu'un seul

dans un *struct cell*, et utiliser 9 bits plutôt que 3 dans *prevs*. Trois uint8_t (un par matrice) ou bien un uint16_t global feront donc l'affaire. Notez que dans chaque matrice, les prédécesseurs possibles d'une case donnée ont toujours les mêmes coordonnées (par exemple (i-1,j-1) pour D). La sémantique de *prevs* change donc complètement ! Ce qu'il faut stocker pour le backtrack c'est maintenant: de quelle(s) matrice(s) provient-on : D, V, ou H?

4C. Application : épissage.

Quand votre implémentation du modèle affine fonctionne: voir *exemple_splicing*. Effectuez les alignements demandés et répondez aux questions. Vous pouvez effectuer les alignements avec les exécutables fournis *aliDnaAffineLongGaps_ntm* et *aliDnaLinear_ntm* , et comparer avec votre programme pour la mise au point (ou si vous n'arrivez pas à implémenter Altschul & Erickson).

4D. Extension 1 : séquences peptidiques (protéines).

Adaptez votre programme pour traiter les séquences peptidiques.

On travaille maintenant sur un alphabet à 20 symboles (les acides aminés). Le score d'un mismatch peut être positif si les deux symboles sont jugés similaires, par exemple s'il s'agit de deux acides aminés aux propriétés physico-chimiques très proches. On utilisera donc une matrice de coûts plus sophistiquée que pour l'ADN, qui prendra en compte les similarités entre acides aminés ainsi que leurs fréquences respectives dans le protéome considéré. De telles matrices ont été produites par des chercheurs et sont largement utilisées, en particulier les matrices des familles PAM et BLOSUM. Vous utiliserez la matrice BLOSUM62. Celle-ci est déjà codée dans *aliCost.c* .

Si votre code est propre, implémenter cette extension devrait être trivial (*make aliProtAffine* pourrait suffire).

Vous pourrez alors aligner les sous-unités alpha, beta et zeta de l'hémoglobine humaine, voir *exemple_Hemoglobin* et répondre aux questions de ce fichier.

4E. Extension 2.

Adaptez votre programme pour afficher **tous** les alignements optimaux, plutôt que juste l'un d'entre eux. Vous pourrez alors répondre à la troisième question du fichier *exemple_splicing* avec votre propre implémentation, plutot que de vous appuyer sur mon *aliDnaAffineLongGaps_ntm* .