BIN702 - Série d'exercices sur BLAST

Manuel Lafond

Exercice 1: Rappelez-vous que dans l'algorithme BLAST, une des étapes "fusionne" des matches, ce qui correspond à joindre deux diagonales. Rappelons les définitions. On a deux séquences S et T et deux paramètres k et d. Ici, k est la longueur des matches voulus et d est l'espace maximum permis entre deux matches à fusionner.
Un **match** est une paire (i, j) telles que les sous-chaînes $S[i..i+k-1], T[j..j+k-1]$ qui sont égales. Un **match étendu** est une paire de matches (i, j), (i', j') tels que les conditions suivantes sont satisfaites:

- $i < i', j < j'$;
- $i' - i = j' - j \leq d$;

donc le "trou" est de la même longueur et n'est pas supérieur à d.
Étant donné S, T, k et d, donnez un algorithme qui trouve tous les matches, puis tous les matches étendus. Quel est la complexité de votre algorithme?

Solution. J'ai donné en classe l'algorithme qui trouve les matches (en stockant les k-mers de T dans une table de hachage).
Soit $H = [(i_1, j_1), \ldots, (i_n, j_n)]$ la liste des matches de longueur k trouvés. L'algorithme naïf va comparer chaque (i_a, j_a) avec chaque (i_b, j_b), ce qui prend un temps $O(|H|^2)$. Avec un peu plus d'effort mental, on peut faire mieux.
Imaginons chaque (i_a, j_a) comme un point dans l'espace 2D. Imaginons ensuite la droite de pente 1 qui passe par (i_a, j_a). Cette droite peut être exprimée par la fonction linéaire $y = x + j_a - i_a$. De plus, tous les (i_b, j_b) qui sont sur cette droite sont les matches étendus potentiel de (i_a, j_a). En d'autres termes, on place chaque (i_a, j_a) sur la droite à laquelle il appartient, puis on compare les matches qui ont été placés sur la même droite.
Ceci n'est qu'une intuition qui mène à l'approche suivante. Cette histoire de droite montre qu'en fait, $i' - i = j' - j$ si et seulement si $j' - i' = j - i$. On peut donc faire comme suit:

- initialiser $D = dict()$ comme un dictionnaire dont la clé est un entier et la valeur est une liste;
- pour chaque (i, j) de H, ajouter (i, j) à la liste $D[j - i]$;

  pour chaque ℓ tel que $D[\ell]$ est non-vide, trier $D[\ell]$ en fonction des coordonnées en i. Puis, pour c de 1 à $|D[\ell]|$, comparer le c-ième élément de $D[\ell]$ avec les d prochains pour voir s'ils forment un match étendu (on note qu'il est inutile d'aller au-delà de d, et qu'on peut arrêter après avec rencontré un match dont la différence est $> d$).

Le temps pour construire D est $O(|H|)$. Ensuite, le temps total passé pour trier tous les $D[\ell]$ est $O(|H| \log |H|)$, et chaque élément demande un temps $O(d)$. La complexité est donc $O(|H| \log |H| + |H|d)$.

□

Exercice 2: Rappelons que BLAST cherche, pour tout k-mer C de S, toutes les occurrences de mots C' qui ne sont pas trop distants de C.

a. Prenons la distance de Hamming pour trouver ces C'. C'est-à-dire, la distance de Hamming entre deux chaînes de même longueur est le nombre de positions où les chaînes ont un caractère différent. Donnez un algorithme qui, étant donné une séquence C de longueur k et un entier d retourne tous les mots C' à distance Hamming au plus d de C. Quelle est la complexité?

   Solution. Il y a plusieurs façons, mais selon moi le plus simple est de choisir d ou moins positions à affecter, puis de les changer selon les options possibles.

```
fonction getHammingStrings(C, d)
    L = [ ] //liste à retourner
    Soit P ⊆ {1, 2, ..., d} tel que |P| ≤ d    // Implémentable avec,
     par exemple en python, itertools.combinations
    pour chaque {p_1, ..., p_k} ∈ P faire
        liste_prev = [C]
        pour i = 1..k faire
            //on génère toutes les façons de modifier C[p_i]
            liste_tmp = [ ]
            pour chaque symbole b tel que C[p_i] ≠ b faire
                pour C' ∈ liste_prev faire
                    C'' = C'.copy()
                    C''[p_i] = b
                    liste_tmp.append(C'')
                fin
            fin
            liste_prev = liste_tmp;
        fin
        L.append_all(liste_prev)
    fin
```

L'idée est que pour chaque combinaison $\{p_1, \ldots, p_k\}$ de $k \leq d$ positions, on génère toutes les combinaisons de façons de modifier $C[p_1], C[p_2], \ldots, C[p_k]$. Si $n = |C|$, le nombre de combinaisons de positions dans P sera

$$\sum_{i=0}^{d} \binom{n}{i} \leq d\binom{n}{d}$$

où $\binom{n}{i}$ est le coefficient binomial. Il est bien connu que $d\binom{n}{d} \in O(dn^d)$.

Ensuite, soit Σ l'alphabet. On voit que pour chaque combinaison dans P, on applique $|\Sigma| - 1$ modifications de la position p_1, fois $|\Sigma| - 1$ modifications de la position p_2, fois ..., et ce pour au plus d positions. En d'autres termes, on va énumérer $O(|\Sigma|^d)$ chaînes de caractères modifiées. Chaque copie à énumérer prend un temps $O(n)$.

Au total, le temps passé est donc $O(dn^d \cdot |\Sigma|^d \cdot n)$.

□

b. Pour compliquer un peu les choses, donnez un algorithme qui, étant donné une séquence C de longueur k et un entier d retourne tous les

mots C' à distance **Levenshtein** au plus d de C.

Soit b le nombre de chaînes à énumérer. Quelle est la complexité de votre algorithme par rapport à b? Peut-on le faire en temps $O(b)$? (note: je n'ai pas la réponse à cette dernière question, mais c'est instructif d'y réfléchir)

Solution. Le plus simple pour moi est un algorithme récursif. On commence avec C et on essaie toutes les façons d'appliquer une modif. Pour chaque façon, on recommence récursivement mais avec d réduit de 1. On prend note de chaque chaîne rencontrée.

```
global L = set() //ensemble, pour éviter les répétitions
getLevStrings(C, 1, d) //appel initial
fonction getLevStrings(C, d)
    si d < 0 alors
        return
    pour chaque position i de C faire
        //tester chaque deletion C' = obtenu de C en supprimant C[i]
        L.insert(C')
        getLevStrings(C', d − 1)

        //tester chaque mutation pour chaque symbole b différent de
         C[i] faire
            C' = obtenu de C en posant C[i] = b
            L.insert(C')
            getLevStrings(C', d − 1)
        fin

        //tester chaque mutation pour chaque symbole b faire
            C' = obtenu de C en insérant b avant la position i
            L.insert(C')
            getLevStrings(C', d − 1)
        fin
    fin
    pour chaque symbole b faire
        C' = obtenu de C en insérant b après la dernière position i
        L.insert(C')
        getLevStrings(C', d − 1)
    fin
```

Déterminer la quantité de chaînes ajoutées est un problème ouvert. On peut tout de même borner le temps de cet algorithme. On note que la longueur de C ne dépasse jamais $n+d$ à travers l'exécution. L'algorithme créé un arbre de récursion dans lequel chaque noeud fait $|C|(1+2|\Sigma|)+|\Sigma|$ appels récursifs. Pour simplifier, on peut borner ceci par $|C|(3|\Sigma|) \leq (n+d)(3|\Sigma|)$. La profondeur de l'arbre de récursion est au plus d, et donc l'algorithme prend un temps $O((n+d)^d(3|\Sigma|)^d)$.

□

Exercice 3: Montrez que si P et T sont deux chaînes aléatoires sur alphabet Σ, le nombre espéré de sous-chaînes égales entre P et T de longueur exactement k est $(m-k+1)(n-k+1)\cdot|\Sigma|^{-k}$.
(note : ceci demande un tout petit peu de connaissances en prob. et stats. et je ne demanderais pas ça en examen. Je recommande d'utiliser la linéarité de l'espérance.)

Solution. Prenons deux position i de P et j de T, avec $i \leq n-k+1$ et $j \leq m-k+1$. La probabilité que $P[i..i+k-1] = T[j..j+k-1]$ est $\frac{1}{|\Sigma|^k} = |\Sigma|^{-k}$, car vous avec $1/|\Sigma|$ chance que $P[i] = T[j]$, fois $1/|\Sigma|$ chance que $P[i+1] = T[j+1]$, et ainsi de suite. Ceci est pour deux positions spécifiques i et j.
Soit $\mathbb{I}_{i,j}$ une variable indicatrice qui vaut 1 si $P[i..i+k-1] = T[j..j+k-1]$, et qui vaut 0 sinon. Soit X le nombre de paires de sous-chaînes identiques (qui est une variable aléatoire). On a

$$X = \sum_{i=1}^{n-k+1} \sum_{j=1}^{m-k+1} \mathbb{I}_{i,j}$$

On veut connaître l'espérance $\mathbb{E}[X]$ de X. On a

$$
\begin{aligned}
\mathbb{E}[X] &= \mathbb{E}\left[\sum_{i=1}^{n-k+1}\sum_{j=1}^{m-k+1} \mathbb{I}_{i,j}\right] \\
&= \sum_{i=1}^{n-k+1}\sum_{j=1}^{m-k+1} \mathbb{E}[\mathbb{I}_{i,j}] \quad \text{par linéarité de l'espérance} \\
&= \sum_{i=1}^{n-k+1}\sum_{j=1}^{m-k+1} |\Sigma|^{-k} \\
&= (n-k+1)(m-k+1)|\Sigma|^{-k}
\end{aligned}
$$

□