Université de Sherbrooke - Département d'informatique
BIN702 - Algorithmes pour la bioinformatique (Automne 2023)

EXAMEN INTRA

Enseignant : Manuel Lafond Date : le 20 octobre 2023

- Cet examen d'une durée de 3 h 30 est individuel. **Je porterai une attention particulière au plagiat.**
- Vous devez remettre votre examen via turnin: `https://turnin.dinf.usherbrooke.ca/`.

  Vous devez remettre un fichier pdf contenant vos réponses (aucune forme spécifique n'est exigée).
- Toute forme de documentation est permise.
- Vous devez répondre à 7 questions totalisant 107 points. Si votre note dépasse 100, elle sera ramenée à 100.
- Vous pouvez **utiliser les résultats démontrés** en classe, dans les notes de cours et dans les exercices sans justification.
- Pour décrire vos algorithmes, vous pouvez soit fournir un pseudo-code, ou bien des phrases qui décrivent les éléments clé de votre algorithme. Les étapes importantes doivent être claires et leur complexité justifiée de façon appropriée. Sauf indication contraire, le **maximum de points** sera atteint si le code atteint une **complexité en temps optimale**.

QUESTION 1 : quelques questions d'échauffement (15 points)

Répondez aux questions de compréhension suivantes. Chaque question compte pour 3 points.

a. Soient S et T deux séquences et, selon une matrice de scores donnée, soient G leur score d'alignement global et L leur score d'alignement local. Vrai ou faux : G est nécessairement plus petit ou égal à L. Justifiez votre réponse.

Solution.
Vrai. Un alignement local cherche des sous-chaînes de S et T qui s'alignent bien. Un cas particulier de sous-chaînes est S et T en entier. Un alignement global est donc un cas spécial d'un alignement local, et alors G ne peut pas être plus grand que L. □

b. Pour trouver des régions similaires entre deux séquences, on peut effectuer un alignement local ou utiliser l'algorithme BLAST. Donnez un avantage et un inconvénient d'utiliser BLAST par rapport à un alignement local. Justifiez brièvement vos réponses.

Solution.
BLAST est plus rapide puisqu'il utilise une heuristique qui cherche des régions autour desquelles chercher. Par contre, il peut manquer des occurrences. □

c. Soit S_1 et S_2 deux séquences de la même longueur n. Soit A_1 l'arbre de suffixes pour S_1 et A_2 l'arbre de suffixes pour S_2. Est-ce que A_1 et A_2 ont nécessairement le même nombre de noeuds? Si oui, dites pourquoi. Sinon, décrivez comment obtenir une séquence de longueur n dont l'arbre de suffixes a le nombre maximum possible de noeuds, parmi toutes les séquences de longueur n.

Solution.
Non. Une séquence dans laquelle tous les caractères sont identiques maximise le nombre de noeuds, par exemple $AAAAAA$. □

d. Dans l'assemblage de séquences, le graphe de chevauchements (aussi appelé le graphe d'*overlap*) contient une arête de X vers Y avec comme poids le plus long suffixe de X qui est un préfixe de Y. Expliquez l'intérêt de chercher un chemin de poids maximum qui passe par chaque sommet exactement une fois.

Solution.
Un tel chemin permet de reconstruire une séquence de longueur minimum qui contient chacun des reads en entrée. □

e. Lorsque nous discutions de profils HMM, nos modèles de markov cachés incluaient des états pour des *deletions* (D_i) et pour des *insertions* (I_i). Expliquez pourquoi chaque état I_i a une boucle vers lui-même, alors que ce n'est pas le cas pour les états D_i.

Solution.
Une deletion correspond à "sauter" une colonne de l'alignement multiple du groupe de séquences avec lequel on se compare, et une colonne ne peut être sautée qu'une fois. Une insertion ajoute un caractère entre deux colonnes, ce qu'on peut répéter indéfinimment. □

QUESTION 2 : alignement global et local (14 points)

Soient les deux séquences suivantes :

$$S = ACGC$$
$$T = GATTAG$$

a. (7 points) On veut calculer la **distance de Levenshtein** entre S et T. Donnez la table de programmation dynamique qui permet de trouver cette distance. Donnez ensuite un alignement global qui atteint cette distance.

Note : Vous devriez ajouter les flèches de trace arrière dans votre table. De cette façon, même si vous faites une erreur de calcul, je pourrai voir si vous avez appliqué les bonnes règles ou non.

Solution.
Vérifiez votre réponse via `https://rna.informatik.uni-freiburg.de/Teaching/index.jsp?toolName=Needleman-Wunsch`. Choisissez "Distance" puis Match=0, autre=1. □

b. (7 points) On veut calculer un alignement **local** de *score maximum*, en utilisant les scores +2 pour un *match*, -1 pour un *gap* et -1 pour une *mutation*. Donnez la table de programmation dynamique permettant de calculer l'alignement **local** de score maximum entre S et T. Donnez ensuite **tous** les alignements locaux de score maximum.

Solution.
Vérifiez votre réponse via `https://rna.informatik.uni-freiburg.de/Teaching/index.jsp?toolName=Smith-Waterman`. □

QUESTION 3 : arbres de suffixe (16 points)

a. (8 points) Construisez l'arbre de suffixes généralisé pour les trois séquences $balboa, alban$ et b (oui, le dernier mot est seulement la lettre b).

Solution.
Vérifiez votre réponse via `http://guanine.evolbio.mpg.de/cgi-bin/drawStrees/drawStrees.cgi.pl`. □

b. (8 points) Rappelons que pour deux séquences S et T, on a définit $overlap(S,T)$ comme la longueur du plus long suffixe de S qui est aussi un préfixe de T. Ces valeurs servaient à reconstruire le graphe de chevauchements. Il est possible de calculer $overlap(S,T)$ en temps $O(|S|+|T|)$. Donnez un algorithme qui atteint cette complexité.

Indice: on peut tester chaque position i de S et voir si le suffixe démarrant à la position i est un préfixe de T. Pour atteindre un temps linéaire, il faudrait gérer chaque position en temps $O(1)$. Ma solution utilise des idées qui servaient à trouver les palindromes.

Solution.
J'acceptais une réponse courte du style:
pour $i = 1..|S|$
– si $PLCP(i,1) = |S| - i + 1$
— retourner $|S| - i + 1$
retourner 0

Pour fins pédagogique, je donne une réponse plus détaillée. On peut faire comme suit:

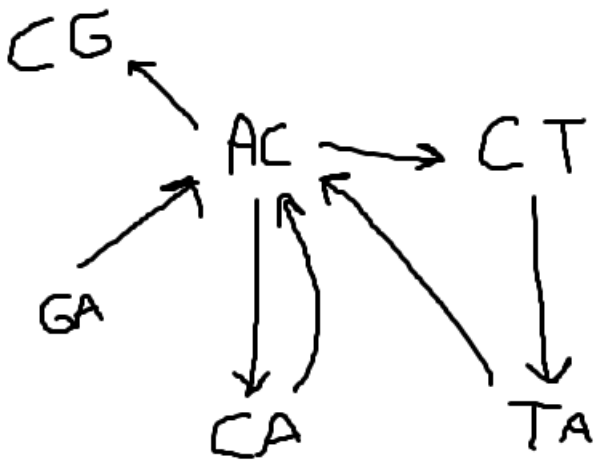
- Construire l'arbre de suffixes pour S et T. (temps $O(|S|+|T|)$)
- Faire un prétraitement pour requêtes PLPC en temps $O(1)$ (avec lca). (temps $O(|S|+|T|)$)
- pour chaque $i = 1..|S|$, vérifier si $PLPC(i,1) = |S| - i + 1$. Si oui, retourner $|S| - i + 1$. Ici, $PLPC(i,1)$ signifie le plus long préfixe commun entre $S[i..|S|]$ et $T[1..|T|]$. Chaque appel $PLPC$ prend un temps $O(1)$ et donc l'étape prend un temps $O(|S|+|T|)$.
- Si on n'a rien retourné, alors retourner 0.

□

QUESTION 4 : séquençage et assemblage (14 points)

a. (9 points) Soit l'ensemble de 3-mers {GAC, ACT, ACA, CTA, ACG, CAC, TAC}. Construisez le graphe de De Bruijn correspondant à cet ensemble. Donnez ensuite une séquence contenant tous ces 3-mers exactement une fois, ou si ce n'est pas possible, dites pourquoi.

Solution.
Acceptez cet humble graphe fait en paint.



Il y a un chemin Eulérien $GA - AC - CA - AC - CT - TA - AC - CG$. Il correspond à la séquence $GACACTACG$.

□

b. (5 points) Donnez la transformée de Burrows-Wheeler de la séquence *amalgama*. Je n'ai pas besoin de la matrice de rotations entière — seulement de la transformée finale.

Solution.
Vérifiez via `http://guanine.evolbio.mpg.de/cgi-bin/bwt/bwt.cgi.pl`. □

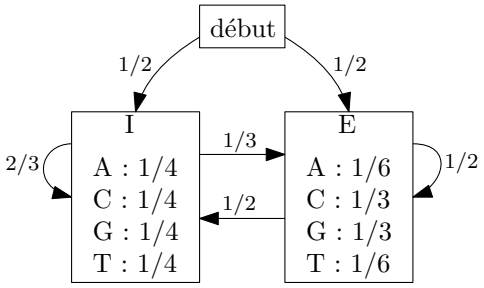
QUESTION 5 : modèles de Markov (12 points)

a. (6 points) Un *intron* est un segment de gène qui est coupé de la molécule d'ARN après traduction, alors qu'un *exon* est un segment de gène qui y reste. Le modèle de Markov caché suivant classifie chaque position d'une séquence comme faisant partie d'un intron (état I) ou d'un exon (état E). Tout chemin commence à l'état début, qui n'émet aucun symbole.

Donnez la probabilité d'observer la séquence ACT en passant par la séquence d'états IEE. Vous devriez laisser une trace de vos calculs.

b. (6 points) Soit un HMM spécifié par (Q, α, Σ, e) tel que vu en classe. Soit S une séquence.

Nous avons vu que l'algorithme de Viterbi calcule une table V dans laquelle, pour tout $i \in \{1, 2, \ldots, n\}$ et tout état $q \in Q$, $V[i, q]$ est la probabilité maximum qu'un chemin d'états



$q_1 q_2 \dots q_i$ satisfaisant $q_i = q$ génère S. Dans le cas où $i > 1$, on a vu la récurrence

$$V[i, q] = \max_{h \in Q}(V[i-1, h] \cdot \alpha(h, q) \cdot e(q, S[i])).$$

Dans vos mots, expliquez pourquoi cette récurrence calcule correctement $V[i, q]$. Soyez concis(e).

Solution.
Un chemin de longueur i qui termine à q doit d'abord faire $i-1$ étapes, finir à un certain état h, puis aller vers l'état q. Pour chaque état h, $V[i-1, h]$ contient la probabilité maximum d'arriver à h en $i-1$ étapes. La récurrence considère tous les états possibles ayant précédé q et prend celui qui donne le maximum. □

Vous devez choisir **deux (2) questions** parmi les suivantes. Si vous me remettez plus de deux réponses, je corrigerai seulement les deux premières. Indiquez clairement vos choix.

QUESTION 6 : tableaux de suffixes et BWT (18 points)

Soit S une séquence de longueur n sur alphabet $\{A, C, G, T\}$, mis à part le dernier symbole qui est le caractère de terminaison \$. Le symbole \$ est lexicographiquement plus petit que les autres et il vient donc avant A, C, G et T dans l'ordre alphabétique.
On définit par $suff(S)$ la liste de tous les suffixes de S, *triée* en ordre alphabétique (croissant). Par exemple, si $S = ACGA\$$, alors $suff(S) = [\$, A\$, ACGA\$, CGA\$, GA\$]$.

a. (6 points) Supposons que vous avez accès à l'arbre de suffixes A pour la séquence S. Montrez que vous pouvez construire $suff(S)$ en temps $O(n^2)$.

Suggestion: on parcourt A récursivement en visitant les enfants dans un ordre approprié.

Solution.
Il suffit de démarrer à la racine et récursivement visiter ses enfants en ordre alphabétique. Lorsqu'on arrive à une feuille, on ajoute son suffixe correspondant à la liste. Ceci visitera les suffixes en ordre alphabétique en temps $O(n)$. À chaque feuille, on doit lire et écrire un suffixe de longueur $O(n)$, ce qui prend un temps total de $O(n^2)$ car il y a n feuilles à visiter.

J'acceptais une réponse brève comme celle ci-haut. Pour plus de détail, on pouvait donner l'algo comme suit:

```
suff(S) = [ ] //variable globale à remplir;
fonction getSuffixes(v, S, A)
    //v est le noeud courant;
    si v est une feuille alors
        Soit X le suffixe correspndant à la feuille v;
        suff(S).append(X);
    sinon
        Trier les enfants de v en ordre alphabétique, selon le premier caractère de la
         branche menant v à l'enfant;
        pour chaque enfant w de v en ordre alphabétique faire
            getSuffixes(w, S, A);
        fin
```

L'algorithme visite chaque noeud une fois et fera donc $O(n)$ appels. Dans un appel avec un noeud interne, on note que le tri se fait en temps $O(1)$ car l'alphabet est $\{A, C, G, T\}$. À chaque feuille, on doit lire et écrire un suffixe de longueur $O(n)$, ce qui prend un temps total de $O(n^2)$ car il y a n feuilles à visiter.

□

b. (6 points) Le *tableau de suffixes* de S est dénoté $SA(S)$, où SA est pour *suffix array*. $SA(S)$ est une liste de n entiers. La i-ème position de $SA(S)$ contient la position de départ du i-ème suffixe en ordre lexicographique. En d'autres termes, $SA(S)[i]$ est la position dans S du suffixe contenu dans $suff(S)[i]$. Par exemple, si $S = ACGA\$$, alors $SA(S) = [5, 4, 1, 2, 3]$. Montrez que si vous avez accès à l'arbre de suffixes A pour S, vous pouvez obtenir $SA(S)$ en temps $O(n)$.

Suggestion: c'est comme à la question précédente, sauf qu'à la sortie on a des entiers.

Solution.
On prend le même algorithme qu'à la question précédente. Par contre, au lieu d'avoir un tableau de séquences, on a un tableau d'entiers. Lorsqu'on arrive à une feuille, au lieu d'écrire le suffixe correspondant à la feuille, on écrit la position du suffixe correspondant.

Ceci demande seulement de parcourir l'arbre et d'écrire un tableau de taille n, ce qui prend un temps $O(n)$. □

c. (6 points) Montrez qu'on peut calculer la transformée Burrows-Wheeler de S en temps $O(n)$.

Suggestion: si vous arrivez à trouver le lien entre $SA(S)$ et $BWT(S)$, ceci devient trivial.

Solution.
Il suffit de mettre $BWT(S)$ comme une séquence de longueur n, dans laquelle le i-ème symbole est $BWT(S)[i] = S[SA[i]] - 1$ (où ici, on suppose que $0 - 1 = |S|$).

Cette réponse était suffisante pour que je l'accepte (et quelques personnes l'ont trouvée!). Pour comprendre que ça fonctionne, $BWT(S)[i]$ est le dernier caractère de la i-ème rotation en ordre alphabétique. De plus, $SA[i]$ est la position du i-ème suffixe en ordre alphabétique. On peut remarquer que la i-ème rotation débute à la position $SA[i]$ (ceci fonctionne grâce au \$), et donc $SA[i] - 1$ est la dernière position de la i-ème rotation. □

QUESTION 7 : alignement non-symétrique (18 points)

Soient S et T deux séquences, avec $|S| = n$ et $|T| = m$.

a. (9 points) Supposons que vous voulez calculer un alignement global de score maximum de S vers T, mais qu'il n'y a eu aucune insertion[1] dans T. Ceci demande de calculer l'alignement global de S vers T, mais en considérant qu'aucun *gap* n'est permis dans S (des gaps sont permis dans T). Supposez qu'une matrice M vous donne tous les scores nécessaires.

Donnez les récurrences et conditions initiales de programmation dynamique permettant le calcul de cet alignement global en temps $O(nm)$.

Solution.
Les conditions initiales sont:
$V[i, 0] = \sum_{k=1}^{i} M[S[i], \text{“}-\text{”}]$
$V[0, j] = -\infty$ car on ne peut pas aligner $T[1..j]$ avec des gaps dans S.

Pour la récurrence avec $i > 0, j > 0$, il suffit de reprendre ce qu'on a vu en cours et d'éliminer le cas qui correspond à mettre un gap dans S à la dernière colonne.

$$V[i, j] = \max \begin{cases} V[i-1, j-1] + M[S[i], T[j]] \\ V[i-1, j] + M[S[i], \text{“}-\text{”}] \end{cases}$$

□

b. (9 points) En plus des conditions de la question précédente, on suppose maintenant que les suppressions ont une longueur minimum c. Si vous décidez d'ajouter un *gap* dans T, il doit nécessairement être suivi de $c - 1$ autres gaps ou plus. Montrez comment calculer l'alignement global de S vers T sous ces conditions, en donnant les récurrences et conditions initiales. La complexité sous-jacente devrait être $O(n^2 m)$.

Solution.
Une façon est de reprendre l'approche avec coût de gap générique présentée en classe. Quand on considère d'aligner $S[i]$ avec un gap, on peut simplement consulter les valeurs qui sont au moins c rangées au-dessus de j, ce qui garantit que l'alignement produit a au moins c gaps conscutifs.

Solution.
Soit $V[i, j]$ le score maximum d'un alignement entre $S[1..i]$ et $T[1..j]$, avec la condition de ne jamais avoir moins de c gaps consécutifs.

Les conditions initiales sont:
$V[i, 0] = -\infty$ si $i < c$ car on ne peut pas aligner i symbols avec moins de c gaps.
$V[i, 0] = \sum_{k=1}^{i} M[S[i], \text{“}-\text{”}]$ si $i \geq c$
$V[0, j] = -\infty$ car on ne peut pas aligner $T[1..j]$ avec des gaps dans S.

Pour la récurrence avec $i > 0, j > 0$:

$$V[i, j] = \max \begin{cases} V[i-1, j-1] + M[S[i], T[j]] \\ \max_{k=0..i-c} \left(V[i-k, j] + \sum_{h=k+1}^{i} M[S[h], \text{“}-\text{”}] \right) \end{cases}$$

[1] Une application est l'alignement d'ARN sur un gène — l'ARN est issu du gène, a pu subir des suppressions, mais pas d'insertion.

Votre réponse pouvait s'arrêter ici et je l'acceptais.

Je donne quand même les détails d'analyse de complexité pour fins pédagogiques. Pour le calcul d'une entrée $V[i,j]$, on peut pré-calculer et stocker toutes les valeurs $\sum_{h=k+1}^{i} M[S[h], \text{“}-\text{”}]$ en temps $O(n)$ en calculant d'abord pour $k=0$, puis pour $k=1$ en soustrayant le premier terme de la sommation, puis pour $k=2$ en soustrayant le second, et ainsi de suite (je ne demandais pas cette optimisation). Une fois ces entrées précalculées, on peut calculer l'entrée $V[i,j]$ en temps $O(n)$ en parcourant $k=0$ à $i-c$ et en utilisant les sommes précalculées. Puisqu'il y a nm entrées à calculer, ceci prend un temps $O(n^2m)$. □

□

QUESTION 8 : recherche de motifs répétés avec erreurs (18 points)

Soit S une séquence et ℓ, k et d, des entiers. On dit qu'un k-mer X est un (k,d)-*motif* de S si S contient ℓ k-mers à des positions distinctes, qui sont chacun à distance de Hamming[2] au plus d de X. En d'autres termes, X est un (k,d)-motif s'il existe des positions $p_1, \ldots, p_\ell$ distinctes telles que $S[p_i \,..\, p_i + k - 1]$ est à distance de Hamming au plus d de X, pour tout $i \in \{1, \ldots, \ell\}$. Notez que X n'est pas nécessairement une sous-chaîne de S.
Par exemple, si $S = ACGCCTACCGACC$ et $\ell = 4$, $X = ACC$ est un $(3,1)$-motif de S car il y a les k-mers ACG, GCC, ACC et ACC, respectivement aux positions $1, 3, 7, 11$ et qui sont tous à distance de Hamming 1 ou moins de ACC.

a. (6 points) Pour un ℓ donné, décrivez un algorithme permettant de trouver un $(k,0)$-motif dans S, s'il y en a un, en temps $O(n)$.

Solution.
Un $(k,0)$-motif n'est qu'un k-mer qui a au moins ℓ occurrences. Pour trouver un tel k-mer, on peut construire l'arbre de suffixes pour S et étiquetter chaque noeud par sa profondeur en caractères et le nombre de feuilles qui descendent du noeud. Tout ceci peut se faire en temps $O(n)$.

Ensuite, on parcourt chaque noeud de profondeur en caractères au moins k. Si on en trouve un qui a au moins ℓ feuilles descendantes, on retourne les k premiers caractères correspondant à la chaîne menant au noeud trouvé. Cette deuxième étape prend aussi un temps $O(n)$. □

b. (12 points) Pour un ℓ donné, décrivez un algorithme permettant de trouver un $(k,1)$-motif dans S, s'il y en a un. Justifiez la complexité de votre approche par rapport à n et k, en considérant $|\Sigma|$ comme une constante. On suppose que n est beaucoup plus grand que k.

Vous devriez viser une complexité $O(nk^3)$ ou mieux, mais vous aurez la majorité de vos points pour une complexité $O(n^2k^2)$.

Solution.
Un $(k,1)$-motif doit être soit un k-mer de S, ou bien une chaîne à distance de Hamming 1 d'un k-mer de S. On peut donc tester toutes ces telles chaînes et, pour chacune d'entre elle, vérifier si c'est un $(k,1)$-motif.

[2]Rappelons que la distance de Hamming entre S et T est le nombre de positions où les séquences diffèrent. Par exemple, la distance de Hamming entre $S = ACCGG$ et $T = TCCTG$ est 2.

```
fonction get-(k,1)-motif(S)
    Construire l'arbre de suffixes A de S;
    pour chaque k-mer P de S faire
        pour chaque k-mer P' à distance 0 ou 1 de P faire
            //vérifier si P' est un (k,1)-motif;
            //en regardant si S contient ℓ k-mers à distance 0 ou 1 de P';
            cpt = 0;
            pour chaque k-mer R à distance 0 ou 1 de P' faire
                occ = nombre d'occurrences de R dans S (utiliser A);
                cpt = cpt + occ;
            fin
            si cpt ≥ ℓ alors
                retourner P';
        fin
    fin
    retourner "Aucun motif trouvé";
```

Le nombre de k-mers P de S est $O(n)$. Le nombre de k-mers P' à distance 0 ou 1 d'un tel P est $O(|\Sigma|k) = O(k)$ (si vous ne voyez pas pourquoi, demandez-moi). Le nombre de k-mers à distance 0 ou 1 de P' est aussi $O(k)$. Finalement, compter le nombre d'occurrences de R peut se faire en temps $O(|R|) = O(k)$ si les noeuds de A sont étiquettés par leur nombre de feuilles descendantes. La complexité est donc $O(n \cdot k \cdot k \cdot k) = O(nk^3)$.

J'acceptais aussi qu'on compte $O(n)$ pour la dernière étape, qui donnerait une complexité $O(n^2k^2)$.

□

QUESTION 9 : quelques problèmes d'assemblage (18 points)

a. (9 points) Soit $r_1, r_2, \ldots, r_m$ un ensemble de séquences représentant des *lectures* (aussi appelé *reads*) obtenues de données de séquençage. Étant donné un paramètre k, donnez le pseudo-code d'un algorithme qui reconstruit le graphe de Bruijn des k-mer présents dans ces lectures. Donnez ensuite la complexité de votre algorithme, avec justification.

Vous n'avez pas à viser un algorithme optimal. Si votre algorithme est correct et que votre complexité est en temps polynomial par rapport à la longueur des r_i, vous aurez tous vos points.

Solution.

```
fonction getGrapheBruijn(r_1, ..., r_m, k)
    G = un graphe vide;
    pour i = 1 .. m faire
        pour p = 1 ..|r_i| − k + 1 faire
            K_1 = r_i[p .. p + k − 2];
            K_2 = r_i[p + 1 .. p + 1 + k − 1];
            Ajouter un sommet K_1 à G s'il n'est pas déjà présent;
            Ajouter un sommet K_2 à G s'il n'est pas déjà présent;
            Ajouter l'arête (K_1, K_2);
        fin
    fin
```

Soit $\ell = \max_{i=1..m} |r_i|$. L'algorithme fait $O(m \cdot \ell)$ iteérations et chacune peut être eimplémentée en temps $O(k)$ pour lire le k-mer de r_i à la position p, et les autres opérations en temps $O(1)$. La complexité est donc $O(nmk)$.

J'acceptais aussi que l'on définisse ℓ comme une sorte de moyenne des $|r_i|$. En réalité, la complexité de cet algorithme est $O(k \cdot \sum_{i=1}^{m} |r_i|)$, mais ce n'était pas nécessaire d'aller à ce niveau de détail.

□

b. (9 points) Supposons que G est un graphe de De Bruijn qui n'a pas de chemin Eulérien (un chemin orienté qui passe une fois par arête). Une explication possible est que certains k-mers ne sont pas présents dans l'entrée. Avec un peu de chance, il se peut qu'il manque un seul k-mer, ce qui correspondrait à une arête manquante dans le graphe.

Donnez un algorithme qui, étant donné un graphe de De Bruijn G, détermine s'il est possible d'ajouter une arête à G pour que le graphe résultant ait un chemin Eulérien. Donnez ensuite la complexité de votre algorithme, avec justification optionnelle.

Vous aurez un maximum de points pour une complexité optimale (à vous de la déterminer).

Vous pouvez supposer que pour chaque noeud v, vous pouvez obtenir le nombre de voisins entrants et sortants de v en temps constant.

Solution.
Il suffit d'observer qu'en ajoutant une arête, on augmente le degré sortant d'un sommet de 1, et le degré entrant d'un sommet de 1.

Il faut chercher à satisfaire les conditions mentionnées en classe: chaque sommet doit avoir $in = out$, sauf peut-être un sommet avec $in = out - 1$ et un avec $in = out + 1$.

- On cherche l'arête (u, v) à ajouter. Dans ce qui suit, u est le départ de l'arête et v la fin. Initialement, u et v sont à *null*.
- S'il y a un sommet w avec $in \geq out + 2$, mettre $u = w$.
  Sinon, s'il y a deux sommets ou plus avec $in = out + 1$, en choisir un et le désigner comme u.
- S'il y a un sommet w avec $in \leq out - 2$, mettre $v = w$.
  Sinon, s'il y a deux sommets ou plus avec $in = out - 1$, en choisir un et le désigner comme v.

- Si u et v sont définis, ajouter l'arête (u, v). Si seulement u est défini, choisir v comme un sommet avec $in = our - 1$ et ajouter (u, v). Si un tel v n'existe pas, choisir v arbitrairement. Si seulement v est défini, appliquer la même logique pour choisir u.
- Repasser chaque sommet pour compter in et out, et vérifier que les conditions nécessaires sont satisfaites.

Chacune de ces étapes se fait en temps $O(n)$.

Une réponse dans le style ci-haut donnait tous les points, même si certains cas n'avaient pas été couverts (par exemple, si seulement u est choisi mais pas v). Il n'était pas nécessaire de justifier votre algorithme dans votre réponse, mais je le fais pour fins pédagogiques. S'il y a un w avec $in \geq out + 2$, il faut absolument rebalancer ce sommet et mettre $u = w$, sinon on n'a aucune chance de réparer le graphe. Sinon, s'il y a deux sommets avec $in = out + 1$, il ne peut y en n'avoir qu'un. Puisque les conditions n'imposent pas de choix sur le sommet avec $in = out + 1$, on peut choisir de rebalancer n'importe lequel et donc on prend u arbitrairement. La logique est la même pour le choix de v.

Si u et v ont été choisis, c'est qu'ils sont "forcés" et si ajouter (u, v) ne répare pas le graphe, alors il n'y a rien à faire. Si u a été choisi mais pas v, alors il y a 0 ou 1 sommet avec $in = out - 1$. Dans les deux cas, augmenter le degré entrant de v ne créé pas de problème. □