Département d'informatique

Faculté des Sciences

Université de Sherbrooke

Projet de session

par

Jean-François Nadeau

nadj2315

Remis à

Manuel Lafond

Dans le cadre de l'activité pédagogique

BIN702

BIN702 – Projet de session

Projet choisi :

Implémenter l'algorithme Fasta et déterminer s'il serait viable de l'utiliser sur un alphabet ASCII pour de la recherche successive de mots similaires dans une grande liste de mot. (Recherche de mots dans un dictionnaire, avec possibilité de modifications, insertions et délétions)

Code : https://github.com/jf-nadeau/Projet_BIN702

Introduction :

La recherche de mots dans un dictionnaire est un problème relativement simple. Cependant, lorsqu'on introduit la possibilité d'erreurs dans le mot recherché, trouver à quel mot l'utilisateur voulait faire référence est un peu plus complexe. Il existe plusieurs algorithmes de programmation dynamique pouvant résoudre ce problème, cependant, ceux-ci ne sont pas performants sur de très grands ensembles de données. C'est le genre de problème auquel les biologistes font face lors de l'alignement de séquence d'ADN, à l'exception que, plutôt que de chercher dans un dictionnaire, on cherche plutôt une région qui correspond le mieux à la séquence donnée dans une très grande séquence (souvent un génome). Puisque dans le vivant, l'ADN de chaque individu est unique et peut subir des mutations au fil du temps, lors de l'alignement, les biologistes doivent considérer qu'il y aura des variations par rapport au génome de référence. La taille du génome de certains organismes peut atteindre milliards de paires de bases. Il n'est donc pas envisageable d'utiliser des algorithmes de programmation dynamique naïvement pour effectuer l'alignement. Des heuristiques très performantes ont donc été développées pour résoudre ce problème. Le but de ce travail est de vérifier si « Fasta », l'un des algorithmes qui ont beaucoup été utilisé en bio-informatique, pourrait être utilisé pour de la recherche de mots dans un dictionnaire.

Revue de littérature :

La recherche de chaîne similaire est un problème courant en informatique. Une brève recherche permet d'identifier plusieurs algorithmes[1] pour effectuer cette tâche, par exemple, la distance de Hamming, de Levenshtein ou de Jaccard, MinHash et SimHash. Du côté du domaine de la biologie, un problème très similaire existe : Rechercher l'endroit dans un génome où une séquence donnée s'aligne le mieux. Des algorithmes comme Smith-Waterman peuvent être utilisé pour ce genre de tâche et donne de très bons résultats, mais le coût d'exécution est beaucoup trop élevé pour aligner une séquence sur un génome entier. Des heuristiques ont donc été développées afin d'approximer cette analyse. Le principe général est d'identifier les régions ayant le meilleur potentiel d'alignement avant d'exécuter un algorithme de programmation dynamique lent sur ces séquences, et ainsi éviter l'analyse couteuse de nombreuses régions non pertinentes. En 1985, une heuristique pour résoudre le problème de l'alignement de séquence a été développé dans l'algorithme Fasta[1]. Cet algorithme a été grandement utilisé dans le domaine de la biologie jusqu'à ce que Blast[2], que l'on pourrait voir comme son successeur, arrive. Blast suit le même principe que Fasta à quelques exceptions près. Par exemple, il ne considère pas seulement les sous-

séquences de la séquence en entrée, mais aussi toutes les variations possibles de ces sous séquence jusqu'à une distance d'édition « d » donnée. Ceci permet à Blast d'identifier de façon plus fiable la position où une séquence contenant plusieurs modifications (mutation / insertions / délétions) devrait s'aligner. J'ai choisi pour ce travail de faire une implémentation de l'algorithme Fasta et de le tester dans un contexte différent.

Dans le contexte dans lequel Fasta a été utilisé, on recherche généralement le meilleur alignement pour le pattern P dans une énorme séquence de référence T. Il est utilisé soit pour des séquences d'ADN (alphabet de quatre lettres) soit pour des séquences d'acides aminés (alphabet de 21 lettres). Pour ce travail, l'alignement se fera sur des mots plutôt que sur des séquences d'ADN ou des protéines. Nous aurons donc un alphabet comprenant tous les caractères ASCII, soit 128 caractères. Nous cherchons à identifier le meilleur alignement entre le pattern P et une liste de mot T_i. Afin d'utiliser Fasta de la même façon dont il a été prévu, il sera nécessaire de joindre les différentes séquences T_i en une grande séquence T. De plus, il sera nécessaire de se souvenir des positions de chaque séquence T_i dans T pour pouvoir retrouver et retourner le mot T_i correspondant au meilleur alignement trouvé dans T.

Méthodologie :

Afin de vérifier s'il serait efficace d'utiliser Fasta dans le contexte de recherche de mots dans un dictionnaire, mon implémentation sera comparée aux algorithmes suivants : la distance de Levenshtein, Smith-Waterman, et la distance de Jaccard combinée avec Smith-Waterman. Les indicateurs de performance vérifiés seront le temps d'exécution de l'algorithme ainsi que la fiabilité (Est-ce qu'il a réussi à trouver le mot recherché ou non).

Résumé des algorithmes choisis :

Distance de Levenshtein : Algorithme de programmation dynamique dans lequel on cherche le nombre minimal de modifications permettant de convertir un mot en l'autre. Une pénalité de -1 est attribuée à chaque modification, peu importe sa nature. Levenshtein effectue un alignement global entre les deux séquences.

Smith-Waterman : Algorithme de programmation dynamique similaire à la Distance de Levenshtein, à l'exception qu'une matrice de poids est utilisée pour définir le poids de chaque modification/match et que l'algorithme effectue un alignement local plutôt que global.

Fasta : Algorithme utilisant une heuristique qui permet de déterminer les régions ayant le plus de potentiel d'alignement avant d'exécuter Smith-Waterman entre la séquence P en entrée et ces régions seulement plutôt que sur toute la séquence T.

Distance de Jaccard : Algorithme visant à estimer la similarité de deux chaînes en se basant sur le nombre de sous chaîne de taille k commun entre les deux chaînes.

Jaccard + Smith-Waterman: Utiliser la distance de Jaccard pour identifier les meilleurs candidats à analyser puis exécuter Smith-Waterman seulement entre la séquence P en entrée et ces meilleurs candidats.

Implémentation de l'algorithme Fasta :

Entrées :

- P : la séquence à aligner
- words : La liste des mots T_i

Sortie :

- Mot correspondant au meilleur alignement trouvé

Étape 1 :

Créer T en concaténant tous les mots T_i de words et créer T_dict, un dictionnaire qui, possède comme clé chaque position possible dans T et comme valeur l'index du titre correspondant dans words.
Exemple : si un mot m débute à l'index 1000 et termine à l'index 10020, T[1000] = T[1001] = T[1003] = … = T[1020] = index de m dans words. Ce dictionnaire sera utilisé à la fin de l'algorithme pour retrouver le mot correspondant au meilleur alignement dans T.

Étape 2 :

Créer un dictionnaire contenant toutes les sous- chaînes de P et leurs positions dans P.

Étape 3 :

Créer un dictionnaire des sous- chaînes de T et leurs positions dans T.

Étape 4 :

Identifier les hotspots, soient les positions des sous- chaînes communes dans P et T.

Étape 5 :

Créer la matrice pour représenter les hotspots de la façon suivante : Créer une matrice de 0 de taille |P| par |T|. Pour chaque hotspot identifiés, à la position de départ du hotspot, remplacer le 0 par un 1 et l'étendre sur une distance k sur la diagonale (où k est la taille des kmers).

Étape 6 :

Identifier et scorer les diagonales contenant des hotspots de la façon suivante : Lire la matrice une diagonale à la fois, de gauche à droite. Pour chaque diagonale, si possible, identifier et conserver dans un dictionnaire la position du premier et du dernier 1. Ces positions correspondent au début et à la fin de la diagonale. Pour chaque diagonale, calculer le score d'alignement en utilisant la même matrice de poids qui sera utilisé plus tard dans Smith-Waterman. Ici, nous sommes sur une diagonale, il n'est donc pas nécessaire de rouler un algorithme de

programmation dynamique pour scorer la diagonale. Il suffit de sommer les scores d'alignement pour chaque position dans la diagonale. On obtient alors un dictionnaire contenant la position de début et de fin de la diagonale comme clé et le score d'alignement comme valeur.

Étape 7 :

Conserver seulement les meilleures diagonales (selon leur score d'alignement) pour la suite. Dans l'implémentation effectuée pour ce travail, les 100 meilleures diagonales ont été conservées.

Étape 8 :

Tenter d'identifier des super-diagonales :

Commentaires informels : Cette étape m'a donné du fil à retordre. Je n'ai pas été en mesure de penser à un algorithme très efficace et la seule documentation que j'ai trouvée indiquant comment cette étape était effectuée dans Fasta suggérait l'utilisation d'un graphe avec poids, mais après quelques tentatives, j'ai fini par opter par une simplification : Utiliser la distance euclidienne entre les diagonales pour déterminer lesquelles sont assez proches pour être jointes. Je suis conscient que cette étape est le point faible de mon algorithme, et que pour atteindre des performances qui ressemblent plus au véritable algorithme, c'est principalement cette section qui devrait être améliorée. Mais malgré cela, mon algorithme performe mieux que les algorithmes de programmation dynamiques standards.

Donc, pour chaque diagonale conservée à l'étape 8 (qui possède une position de début et une position de fin), vérifier si la diagonale possède un voisin à l'aide de la distance euclidienne entre sa fin et le début des potentiels voisin. Seuls les voisins dont le début est situé dans la région en bas à droite de celle-ci (>= diago.fin.i et >= diago.fin.j) sont considérées. Si un voisin est trouvé, ajouter les deux diagonales dans un dictionnaire contenant comme clé la diagonale et comme valeur le voisin. Finalement, après avoir tenté d'identifier les voisins de chaque diagonale, utiliser le dictionnaire diagos-voisins pour reconstruire et scorer les super-diagonales, puis les ajouter à la liste des diagonales conservées à l'étape 8.

Étape 9 :

Conserver seulement les meilleures diagonales/super-diagonales (selon leur score d'alignement) pour la suite. Dans l'implémentation effectué pour ce travail, les 10 meilleures diagonales/super-diagonales ont été conservées.

Étape 10 :

Pour chaque diagonale/super-diagonale conservée à l'étape 9, Identifier la sous-chaîne de T correspondante et calculer le score d'alignement entre P et cette sous-chaîne à l'aide de Smith-Waterman.

Étape 11 :

Prendre la diagonale ayant généré le meilleur score à l'étape 10 et à l'aide de la position de départ et du dictionnaire T_dict, trouver l'index du mot correspondant et le retrouver dans words.

Résultats :

Les analyses qui ont généré les résultats suivants ont été effectuées sur une liste de mots représentant les titres d'oeuvres musicales d'une société donnée. Cette liste se trouve dans le répertoire contenant les scripts des algorithmes. À noter que les séquences sont converties en lettres minuscules avant l'analyse.

Test 1 :

P = Abandonned_SewersOf York

Match attendu : Abandoned Sewers Of New York

	Temps d'exécution (secondes)	**Meilleur match**
Levenshtein	6.33	Abandoned Sewers Of New York
Smith-Waterman	8.99	Abandoned Sewers Of New York
Fasta	2.89	Abandoned Sewers Of New York
Jaccard	0.06	Abandoned Sewers Of New York

On remarque d'abord qu'il y a une bonne différence entre l'algorithme de Levenshtein et Smith-Waterman. Bien que ce sont tous les deux des algorithmes de programmation dynamique très similaires, Levenshtein effectue un alignement global, ce qui signifie qu'il retourne la valeur contenue à la position le plus en bas à droite de sa matrice. L'accès à cette valeur se fait en temps constant. Cependant, Smith-Waterman effectue un alignement local, ce qui veut dire qu'il doit chercher et retrouver la valeur maximale dans la matrice, donc la complexité est de O(n*m) où n et m correspondent aux dimensions de la matrice.

Ensuite, on remarque aussi une grande différence entre Smith-Waterman et Fasta. Bien qu'ils effectuent tous deux un alignement local, l'heuristique permettant d'identifier les meilleurs candidats utilisés dans l'implémentation de Fasta permet d'accélérer grandement le processus.

Finalement, la combinaison de l'algorithme de Jaccard et de Smith-Waterman performe extrêmement mieux que toutes les autres présentées. Ceci est dû au fait que la complexité du calcul de l'intersection de deux ensembles peut être effectué en temps linéaire O(n). De plus, une accélération a été utilisée pour réutiliser l'intersection plutôt que de calculer l'union entre les deux ensembles :

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

Nous obtenons donc une façon beaucoup plus rapide d'identifier les meilleurs candidats qu'avec Fasta dans ce contexte. De plus, nous n'avons pas à reconstruire T en concaténant tous les mots ni à construire un dictionnaire des index des différents T_i dans T comme nous le faisons dans Fasta. En somme, la distance de Jaccard semble simplement plus adapté e au contexte de recherche de mots dans un dictionnaire. Cependant, étant donné qu'elle ne prend pas en compte l'ordre de ses

sous-chaînes, celui-ci risque d'être moins performant dans certaines situations, par exemple, dans une situation où on retrouve beaucoup de sous-chaînes répétées. Mais puisque cet aspect s'éloigne du sujet de ce travail, il ne sera pas exploré.

Test 2 :

P = dreamnight10

Match attendu : Dream The Night 10

Note : la liste de titres contient les titres « Dream The Night » 1 à 11

	Temps d'exécution (secondes)	**Meilleur match**
Levenshtein	3.37	Dream The Night 10
Smith-Waterman	4.94	Dream The Night 10
Fasta	1.28	Dream The Night 10
Jaccard	0.05	Dream The Night 10

On remarque ici la même tendance côté performance relative. Cependant, globalement, le temps d'exécution a été plus court pour tous les algorithmes. Ceci s'explique par le fait que P était deux fois plus court que le P du test 1. Un P plus court signifie une matrice de programmation dynamique plus petite et une quantité de sous-chaînes plus petite. Il est donc logique que les performances soient meilleures dans ce cas.

Test 3 :

P = dream the night_10

Match attendu : Dream The Night 10

Note : la liste de titres contient les titres « Dream The Night » 1 à 11

	Temps d'exécution (secondes)	**Meilleur match**
Levenshtein	4.32	Dream The Night 10
Smith-Waterman	6.12	Dream The Night 10
Fasta	1.63	Dream The Night 11
Jaccard	0.08	Dream The Night 11

Dans ce test, un point faible de Fasta et de Jaccard a été identifié. Ces deux algorithmes effectuent la séparation des chaînes à comparer en sous-chaînes de taille k afin d'identifier les sous- chaînes communes. À cause de cela, si un non-match se situe à k ou moins caractères de la fin de P, ces derniers caractères ne feront pas partie des sous- chaînes communes. Dans notre cas, nous avons « Dream The Night 10 » qui termine par « \s10 » où \s est un espace. Puisque dans

l'implémentation utilisée, k =3, le dernier kmer commun avec «dream the night_10 » qu'il sera possible de former est « ght ». Il n'est pas possible de créer un kmer commun de 3 caractères entre « _10 » et « \s10 ». La fin du mot sera donc « ignorée ». Ce qui s'est produit ici est donc que les « Dream The Night 1 ;a 11» avaient tous en commun les sous- chaînes de « Dream the night » sans leur numéro. Ils ont donc toutes recu le même score. L'algorithme a alors retourné le dernier titre qu'il a rencontré possédant ce meilleur score.

Note : Pour ce qui est de Jaccard, puisqu'on roule Smith-Waterman contre le titre entier et non pas une sous- chaîne comme dans Fasta, on aurait quand même dû trouver le bon titre. Après vérification, il s'agit d'un cas d'exception où la liste des 10 meilleurs candidats considérés contenait tous les « Dream The Night» sauf « Dream The Night 10 ». En augmentant le nombre de candidats considérés, on trouve le bon match. Finalement, il serait aussi possible de modifier Fasta pour contrer ce genre de problème en passant dans Smith-Waterman les titres complets T_i correspondants aux super-diagonales trouvées plutôt que la sous-chaîne de T correspondant à la diagonale. De plus, il est intéressant de noter que l'algorithme Blast, le successeur de Fasta, n'aurais pas rencontré ce problème puisqu'il prend en compte toutes les variations possibles des kmers à une distance d. le kmer « _10 » aurait donc été pris en compte avec Blast.

Test 4 :

Hidden Treasures Of The Infinite (0 modifs)
Hidden_Treasures_Of_The_Infinite (4 modifs dispersées)
Hiden Tresurre_Of_de_Infinity (10 modifs dispersées)
Hidden The Infinite (13 modifs succéquentes)

Match attendu : Hidden Treasures Of The Infinite

(0 modifs)

	Temps d'exécution (secondes)	**Meilleur match**
Levenshtein	8.88	Hidden Treasures Of The Infinite
Smith-Waterman	12.94	Hidden Treasures Of The Infinite
Fasta	3.34	Hidden Treasures Of The Infinite
Jaccard	0.07	Hidden Treasures Of The Infinite

(4 modifs dispersées)

	Temps d'exécution (secondes)	**Meilleur match**
Levenshtein	8.87	Hidden Treasures Of The Infinite
Smith-Waterman	12.31	Hidden Treasures Of The Infinite
Fasta	3.34	Hidden Treasures Of The Infinite
Jaccard	0.08	Hidden Treasures Of The Infinite

(10 modifs dispersées)

	Temps d'exécution (secondes)	Meilleur match
Levenshtein	7,75	Hidden Treasures Of The Infinite
Smith-Waterman	11.23	Hidden Treasures Of The Infinite
Fasta	3.29	Hidden Treasures Of The Infinite
Jaccard	0.06	Hidden Treasures Of The Infinite

(10 modifs successifs)

	Temps d'exécution (secondes)	Meilleur match
Levenshtein	5,07	Dream The Night
Smith-Waterman	7,35	Hidden Treasures Of The Infinite
Fasta	2.25	Hidden Treasures Of The Infinite
Jaccard	0.07	Hidden Treasures Of The Infinite

On remarque ici que la tendance de performance relative entre les algorithmes se maintient dans les différents cas de figure présentés. Cependant, la fiabilité de Levenshtein est affectée par la suppression de caractères. L'explication est simple, la pénalité de modifier quelques caractères pour atteindre un mot de taille similaire était plus petite que la pénalité associée à corriger et insérer toutes les lettres manquantes.

Test 5 :

qwerty
qwertyqwerty
qwertyqwertyqwertyqwerty
qwertyqwertyqwertyqwertyqwertyqwertyqwertyqwerty

Match attendu : n/a. Ce test est pour vérifier l'impact de la taille de P sur le temps d'exécution. P n'est pas un mot contenu dans la liste, donc le meilleur match n'est pas une information pertinente dans ce test, cependant, il peut être intéressant de voir les résultats et de réfléchir à pourquoi les différents algorithmes ont retourné ce match.

qwerty

	Temps d'exécution (secondes)	Meilleur match
Levenshtein	1,85	redeye
Smith-Waterman	2,57	party until the walls fall 8
Fasta	0.54	Party Until The Walls Fall 8
Jaccard	0.04	Street Party

qwertyqwerty

	Temps d'exécution (secondes)	Meilleur match
Levenshtein	3.22	get bent
Smith-Waterman	5.18	dirty martinis 7
Fasta	0.99	Party Until The Walls Fall 8
Jaccard	0.05	Dirty Martinis

qwertyqwertyqwertyqwerty

	Temps d'exécution (secondes)	Meilleur match
Levenshtein	6.62	weather watchers
Smith-Waterman	10.01	dirty martinis 7
Fasta	1.98	Party Until The Walls Fall 8
Jaccard	0.07	Dirty Martinis

qwertyqwertyqwertyqwertyqwertyqwertyqwertyqwerty

	Temps d'exécution (secondes)	Meilleur match
Levenshtein	13.16	bittersweet september
Smith-Waterman	19.35	dirty martinis 7
Fasta	4.05	Party Until The Walls Fall 8
Jaccard	0.10	Dirty Martinis

qwerty

	Temps d'exécution (secondes)	Meilleur match
Levenshtein	27.41	bittersweet september 8
Smith-Waterman	39.29	dirty martinis 7
Fasta	8.23	Party Until The Walls Fall 8
Jaccard	0.17	Dirty Martinis

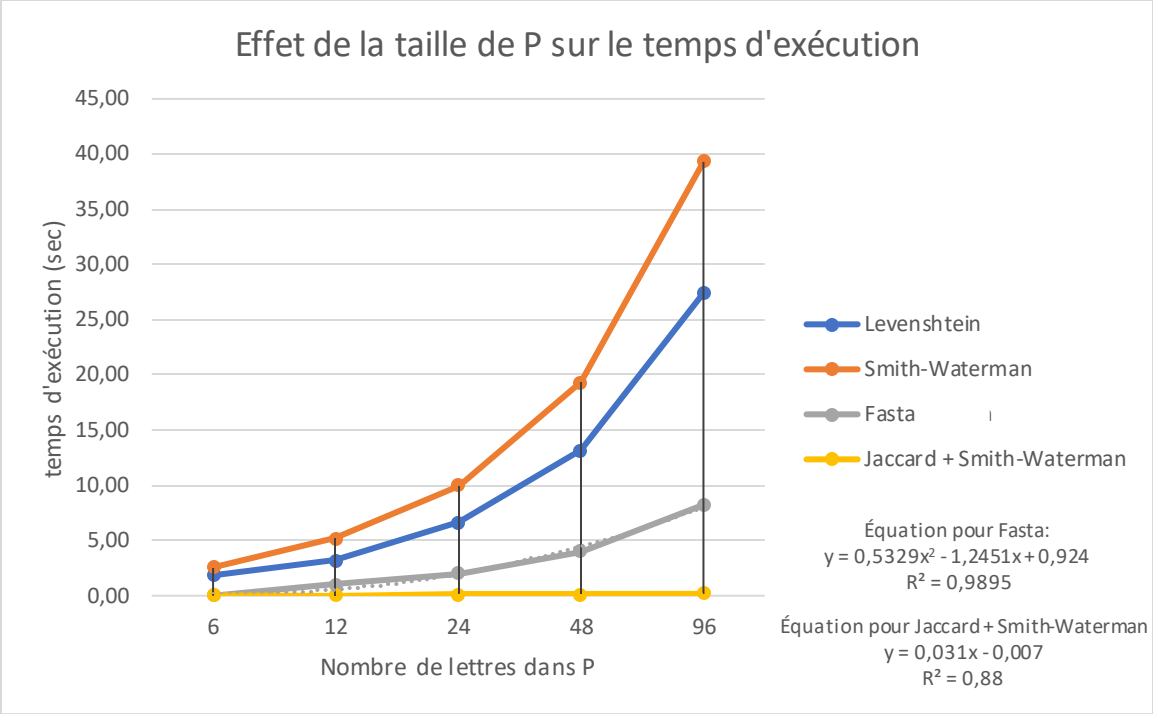

Figure 1: Effet de la taille de P sur le temps d'exécution de la recherche de P dans la liste de mots pour les algorithmes Levenshtein, Smith-Waterman, Fasta maison et Jaccard + Smith-Waterman. En x, le nombre de lettre dans le mot P en entrée et en y, le temps d'exécution en secondes.

Pour ce test, on remarque encore une fois que Levenshtein performe mieux que Smith-Waterman. La raison de cette différence a été expliqué dans le Test 1. On remarque aussi qu'ils ont une tendance exponentielle. Ce n'est pas une surprise puisqu'on sait que ces algorithmes de programmation dynamique ont une complexité de O(n*m). L'implémentation de Fasta effectuée performe beaucoup mieux que les algorithmes de programmation dynamique. Il s'agit encore d'une courbe exponentielle, mais de facteur beaucoup moins grand que Levenshtein et Smith-Waterman. Pour ce jeu de données, l'équation de la courbe est la suivante : $0,5329x^2 - 1,2451x + 0,924$. Si les chaînes à analyser sont très longues, On gagne beaucoup à utiliser cette implémentation de Fasta comparativement aux algorithmes de programmation dynamique classiques. Cependant, on voit encore clairement ici que l'algorithme de Jaccard combiné à Smith-Waterman performe beaucoup mieux que les autres et qui possède une complexité linéaire pour ce jeu de données. Il est tout de même important de noter que ce jeu de données était avantageux pour la distance de Jaccard, puisqu'on utilise un mot répété. Donc, dans les mots les plus longs, le terme répété « querty » génère des sous chaînes déjà présente dans l'ensemble des kmers. Donc, dans ce test, le nombre d'éléments dans l'ensemble des kmers de P n'augmentait pas de façon proportionnelle au nombre de caractères de P.

Conclusion :

Fiabilité :

En somme, les tests semblent suggérer que les 4 algorithmes testés donnent des résultats fiables et cohérents. Cependant, les tests ont permis d'identifier certains points faibles. L'algorithme de Levenshtein risque d'être peu efficace si P contient de nombreuses délétions, car même si on a plusieurs lettres qui correspondent dans une partie du mot, il peut s'avérer moins couteux de modifier les lettres restantes pour correspondre à un mot plus court que d'ajouter les lettres manquantes. Dans le contexte où il est utilisé, c'est le point faible d'un algorithme d'alignement global. Smith-Waterman a semblé fiable dans tous les cas. Dans ce contexte, l'alignement local est avantageux puisque même si on a de nombreuses délétion ou insertions, si une partie du mot suffisamment longue correspond, le score d'alignement permettra de le distinguer des autres. Fasta et Jaccard possèdent le même point faible puisqu'ils sont tous les deux basés sur l'analyse des sous chaînes communes. Si on a une modification dans P à k (ou moins) caractères du début ou de la fin, on ne générera aucune sous chaîne commune pour ces sections du mot. Dans le jeu de données utilisé, ceci pouvait avoir un grand impact puisqu'il existe plusieurs variants d'un même mot dont la seule différence est le # qui se trouve à la toute fin. Il est à noter que l'algorithme Blast, le successeur de Fasta, ne possèderait pas cette faiblesse puisqu'il prend en considération toutes les variations des kmers possible à une certaine distance d.

Temps d'exécution :

Le but de ce travail était de déterminer si une implémentation de Fasta sur un alphabet ASCII serait efficace sur la recherche de plusieurs mots dans une liste de mots. Au départ, il était déjà connu qu'utiliser un algorithme de programmation dynamique était fiable mais beaucoup trop lent. Les tests effectués dans ce travail on permis de déterminer que l'implémentation de Fasta dans ce contexte était plus avantageuse que l'utilisation naïve d'algorithmes de programmation dynamique. Cependant, elle ne semble pas suffisamment avantageuse comparativement à la distance de Jaccard combiné avec un algo de programmation dynamique. Je crois que bien que Fasta offre une amélioration dans ce contexte, il n'est pas tout à fait approprié à ce genre de d'analyse. Entre autres, il est nécessaire de créer T en concaténant les différents mots de la liste de mots et de créer un dictionnaire contenant les positions des différents mots dans T afin de pouvoir retrouver le mot correspondant au meilleur alignement trouvé. Tout ce calcul et cette utilisation de mémoire n'est pas nécessaire pour d'autres algorithmes plus appropriés pour la tâche, comme la distance de Jaccard. Cependant, puisque la distance de Jaccard considère les sous chaînes de façon non-ordonnées ne donnerait probablement pas des résultats fiables dans une recherche de séquence d'ADN sur un alphabet de 4 lettres et sur un génome de référence faisant des milliards de paires de base, dans lesquels les sous chaînes répétés sont plus fréquentes. Considérer l'ordre pour former des diagonales de séquences communes comme Fasta le fait donnera forcément des résultats plus fiables. Pour le contexte que nous avons présenté, d'autres algorithmes développés spécifiquement pour la recherche de mots similaires sont couramment utilisés. Nous avons présenté ici la distance de Jaccard, mais il en existe d'autres, tels que MinHash et SimHash, qui sont d'ailleurs utilisés par Google pour la recherche de pages

quasi-duplicats à l'aide de « web crawlers »[3]. Ce que je veux dire avec tout ça et que Fasta est un algorithme particulièrement performant et fiable dans le contexte pour lequel il a été créé, mais pour le contexte utilisé dans ce travail, il existe des solutions plus appropriées.

Références

1. Padole, Dr. Mamta. (2005). Search Algorithm Used in FASTA.
2. Myers G. (2013) What's Behind Blast. In: Chauve C., El-Mabrouk N., Tannier E. (eds) Models and Algorithms for Genome Evolution. Computational Biology, vol 19. Springer, London. https://doi.org/10.1007/978-1-4471-5298-9_1
3. Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. 2007. Detecting near-duplicates for web crawling. In Proceedings of the 16th international conference on World Wide Web (WWW '07). Association for Computing Machinery, New York, NY, USA, 141–150. DOI:https://doi.org/10.1145/1242572.1242592
4. *Globalement, les notes de cours très bien expliquées de Manuel Lafond sur ce sujet ont aussi été utilisées.*