UNIVERSITÉ DE SHERBROOKE
Faculté de génie
Département de génie électrique et génie informatique

RAPPORT APP 5 – GÉNIE INFORMATIQUE

Structures de données et complexité

GIF270

Présenté à
Jean-Samuel Lauzon, Frédéric Mailhot

Présenté par

Marie-Eve Castonguay – casm1907 – 19025939 – T2

Sherbrooke - 11 mars 2020

TABLE DES MATIÈRES

1. STRUCTURES DE DONNÉES UTILISÉES

Pour cet résoudre la problématique de cet APP, j'ai principalement utilisé des tableaux de hachage. En Python, on réfère aux tableaux de hachage lorsque l'on parle de dictionnaires. Afin de rendre notre programme le plus efficace possible, nous devions nous baser sur la complexité des algorithmes pour choisir le type de structures de données à utiliser. J'ai choisi d'organisé mes n-grammes dans un dictionnaire Python parce que le Big-O pour accéder aux éléments du tableau ainsi que pour en ajouter est de O(1). En fait, puisqu'on accède directement aux éléments du tableau par leur clé/index, on effectue un accès direct d'ordre O(1). Pour ajouter des éléments au tableau, l'implémentation des dictionnaires en python utilise l'adressage ouvert afin de gérer les collisions, qui est aussi d'ordre O(1). Par contre, si la gestion des collions avait été implémentée en utilisant le méthode de chaînage et que tous les éléments du tableau se seraient retrouvés malencontreusement dans le même « bucket », la complexité Big-O aurait été O(n) puisque, dans le pire des cas, nous aurions eu à parcourir n éléments afin d'accéder au dernier élément. Si une telle situation survenait, il faudrait revoir la fonction de hachage utilisée afin qu'elle ne renvoie pas le même signature pour tous les éléments.

J'ai aussi considéré l'utilisation d'arbres binaires puisqu'ils auraient facilité l'organisation des n-grammes en fonction de leur fréquence. Au final, j'ai décidé de ne pas opter pour cette option car la complexité Big-O pour ajouter ainsi que pour accéder aux éléments de la structure de données est de O (log n), ce qui est moins efficace que la complexité O(1) des tableaux de hachage.

Puisque les tableaux de hachages n'organisent pas nécessairement les données de la façon souhaitée, j'ai aussi du implémenter une méthode pour trier les n-grammes en fonction de leur fréquence. J'ai opté pour un tri fusion plutôt que pour un tri rapide ou à bulle. Dans les pires cas, les deux derniers ont une complexité $O(n^2)$, tandis que le tri fusion a une complexité O(n log n).

Le tri fusion requiert plus d'espace mémoire, mais puisque nous nous soucions du degré d'efficacité et que nos ordinateurs sont assez puissants, j'ai priorisé le degré de complexité des algorithmes.

2. COMPLEXITÉ DES ALGORITHMES UTILISÉS POUR L'AJOUT DES MOTS

Pour analyser la complexité, il faut tout d'abord comprendre comment le programme est structuré. Il comporte trois classes : *TextParser*, *NgrammeNode* et *Statistics*. La première ne sert qu'à lire un ou plusieurs fichiers textes, puis à le transformer en une liste de mots qui sera utilisée qu'une seule fois pour générer la structure de donnée contenant les n-grammes. Il est important de noter que ce processus n'est effectué qu'une seule fois et qu'on ne travaille pas directement avec cette liste de mots. La classe *Statistics* est celle qui contient la méthode pour générer les n-grammes (des objets de type *NgrammeNode*), ainsi que le dictionnaire qui les regroupe. Un *NgrammeNode* contient le *string* associé au n-gramme, un entier qui dénote la fréquence, puis un dictionnaire de tous les n-grammes prochains possibles.

La méthode qui nous intéresse est *generateStats*() de la classe *Statistics*. Cette méthode parcours la liste de mots fournie par le TextParser, n mots à la fois (selon la taille des n-gramme voulue), en incrémentant de 1 l'index pointant vers le début du n-gramme, jusqu'à ce que la fin de la liste de mots soit atteinte. Cette opération a une complexité de O(n), puisque qu'elle est obligée de parcourir tous les mots fournis du début à la fin.

Imbriqué dans cette boucle *while (index+nsize)<=wordCount*, on retrouve une boucle *for i in range(nsize)* qui possède une complexité O(n) dès que nsize (taille du n-gramme) est plus grand que 1. Si on observe des unigrammes, cette boucle for a une complexité O(1), puisqu'on ne fait qu'assigner une variable a une valeur du tableau avec un index donné. On retrouve aussi deux conditions *if…else…* qui elles ont une complexité de O(1) puisque les actions conséquentes sont soit d'accéder à une des clés du dictionnaires (accès direct, O(1)), d'instancier un objet de type NgrammeNode (aussi O(1)), où d'appeler la méthode qui incrémente de 1 la fréquence d'un objet de type NgrammeNode (O(1)).

Au final, puisque l'on retrouve une boucle for imbriquée dans une boucle while, il faut multiplier leurs complexités respectives afin de calculer l'ordre de complexité de l'algorithme. Si on observe des unigrammes, on obtient un Big-O de O(n) x O(1) = O(1). Par contre, si on observe des n-grammes possèdant plus d'un mot, un obtient un Big-O de O(n) x O(n) = $O(n^2)$.

3. MOTS LES PLUS UTILISÉS PAR AUTEUR

Tableau 1: Unigrammes les plus fréquents pour chaque auteur

Auteur	Unigramme (fréquence)				
	1er unigramme	2e unigramme	3e unigramme	4e unigramme	5e unigramme
Balzac	les (17 629)	une (12 131)	que (11 594)	qui (10 355)	des (9884)
Hugo	les (12 445)	une (9265)	qui (8733)	que (8676)	dans(8024)
Ségur	que (3885)	les (3350)	pas (3031)	vous (2643)	elle (2635)
Verne	les (15 492)	des (9457)	que (9078)	une (8500)	qui (7079)
Voltaire	les (1404)	que (1306)	qui (1008)	des (867)	vous (834)
Zola	les (14 201)	elle (12 092)	des (9467)	une (9406)	pas (6198)

Tableau 2: Bigrammes les plus fréquents pour chaque auteur

Auteur	Bigramme (fréquence)				
	1er bigramme	2e bigramme	3e bigramme	4e bigramme	5e bigramme
Balzac	dans les (976)	que vous (898)	peut être (839)	tous les (817)	une femme (801)
Hugo	jean valjean (1109)	dans les (899)	qui est (632)	est que (594)	tous les (574)
Ségur	des ormes (394)	que vous (384)	mac miche (298)	les enfants (248)	est pas (237)
Verne	dans les (1008)	cyrus smith (856)	que les (743)	sur les (648)	avec une (564)
Voltaire	tous les (122)	saint yves (118)	que vous (118)	lui dit (99)	dit candide (90)
Zola	elle avait (928)	elle était (757)	dans les (589)	sur les (566)	les yeux (524)

4. IDENTIFICATIONS DES TEXTES DE L'AUTOVALIDATION

Tableau 3: Détection et calculs de distances des textes à identifier

Distance avec :	Balzac	Hugo	Ségur	Verne	Voltaire	Zola
Balzac_généré.txt	0.157	0.164	0.161	0.171	0.156	0.180
Hugo_généré.txt	0.176	0.164	0.170	0.174	0.165	0.179
Ségur_généré.txt	0.154	0.167	0.135	0.178	0.154	0.173
Verne_généré.txt	0.204	0.209	0.201	0.200	0.204	0.223
Voltaire_généré.txt	0.151	0.154	0.151	0.163	0.137	0.172
Zola_généré.txt	0.165	0.165	0.180	0.150	0.159	0.154

Vert : Bonne identification

Rouge : mauvaise identification → Jaune : la distance mesurée avec le bon auteur