Analyse des sentiments à partir de critiques de films

PPII

Auteurs :
Elisa DARGENT
Niels TILCH
Léa TOPRAK

Responsables du PPII :
Sébastien DA SILVA
Olivier FESTOR
Gérald OSTER

20 Avril 2020 - 7 juin 2020

TELECOM Nancy

Table des matières

Notations du rapport : On parle de hash ou de hashcode pour le résultat d'une fonction de hachage et on notera n la taille de la table dans l'état de l'art.

1 Introduction

Longtemps les émotions et les sentiments furent méconnus par une partie des intellectuels et pour cause, la majorité des philosophes ; tels qu'Aristote, les stoïciens ou Kant ; défendirent à travers les temps une impassibilité, une modération, voire une déformation de ces derniers afin d'atteindre le bonheur. Et ces courants de pensée favorisant la raison aux sentiments ont eu une influence jusqu'à nos sociétés modernes comme en témoignent certains grands films tels que Star Wars avec le code Jedi :*"Il n'y a pas d'émotions, il y a la paix"*.

Cependant, les sentiments sont actuellement au centre de notre société et régissent nombre de nos actions : les réseaux sociaux jouent sur notre fierté et notre sentiment d'appartenance, les publicités ciblées s'attaquent à nos envies ou à nos peurs et les applications de rencontre jouent sur l'amour. Nos sentiments sont donc au coeur de notre économie. Par conséquent de nombreuses recherches s'ouvrent et pour de nombreux neuroscientifiques actuels, tels qu'Antonio Damasio, les émotions sont des expériences qui nous apportent des informations sur notre état mais aussi sur notre environnement. C'est pourquoi récupérer et analyser ces sentiments s'avère primordial pour l'économie et le développement de nombreux domaines. Mais en pratique comment faire ? Comment un programme, tout a fait opposé aux concepts de sentiment et de sensibilité, peut-il interpréter les émotions derrière un simple commentaire ?

2 État de l'art.

2.1 Le lien entre masses de données et sentiments personnels : Les publicités

En 2021, il est estimé que la part des publicités en ligne représentera environ 52% du marché publicitaire mondial, soit près de 330 milliards de dollars d'après Adventising Expenditure Forecasts.

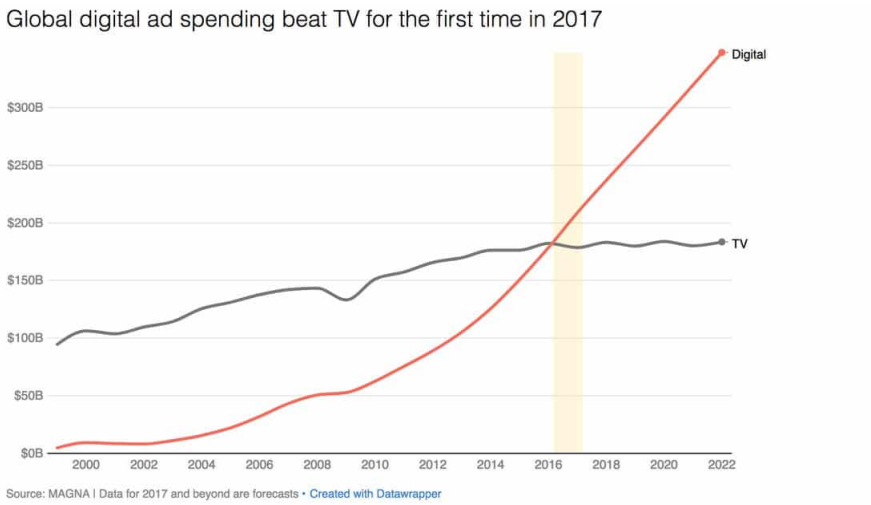


FIGURE 1 – Courbe des dépenses publicitaires

Cet accroissement aussi rapide de ce marché en ligne est dû à la précision des systèmes informatiques à présenter les publicités à une population susceptible d'être conquise. Cette précision recherchée par les entreprises permet de réduire les coûts tout en augmentant l'efficacité de leurs publicités.

Les deux entreprises représentant les grosses parts de publicités en ligne sont Facebook avec Facebook Ads et Google avec Google Ads. En utilisant leur propre système de base de données, nommée Google F1, Google permet aux entreprises de montrer leurs produits à des consommateurs pertinents selon leur historique et leurs recherches grâce aux cookies.

Lorsque les cookies vont détecter la page web, ils vont enregistrer vos derniers sites web visités et, en fonction de la catégorie du site, proposer des pubs en relation avec sa catégorie. Prenons l'exemple de Monsieur M visitant des sites pour pouvoir acheter une maison. Après plusieurs visites et recherches sur des sites spécialisés, Google Analytics va remarquer que Monsieur M est intéressé par les maisons à vendre. Ainsi, l'algorithme va lui afficher des publicités concernant des sites immobiliers.

Néanmoins, la demande accrue d'afficher des publicités par les compagnies ont nécessité une hausse de la collecte d'informations de la part de Google afin de rencontrer les espérances de visibilité des entreprises. Aujourd'hui Google traite plus de 20 petabytes d'informations par jour [1]. Certaines informations sont collectées grâce aux utilisateurs de Google Home ce qui a suscité des controverses vis-à-vis de la vie privée. Aujourd'hui, Google (plus particulièrement la firme Mountain View) est menacée d'une plainte de dédommagement de 5 milliards de dollards pour avoir abusé de

la collecte d'informations sur le mode de navigation privée contrairement à leurs déclarations sur ces onglets [2].

2.2 Les fonctions de hachage

2.2.1 Définition

Une fonction de hachage cryptographique est une fonction prenant en entrée un message (généralement une suite de bits de longueur indéfinie) et donnant un hash (généralement une suite de bits) de longueur propre à la fonction.
De manière plus formelle [3], une fonction de hachage h est définie par :

$$h : \Sigma^* \to \Sigma^n$$

Bien que cette définition soit simple, une fonction de hachage cryptographique ne serait pas sécurisée sans ses diverses propriétés.

2.2.2 Critères d'une fonction de hachage cryptographique

Propriétés fondamentales En théorie une fonction de hachage doit vérifier les 4 propriétés suivantes :

1. La valeur d'un hash se calcule très facilement.
2. La fonction est à sens unique : pour tout hash donné , il est impossible de retrouver le message.
3. La moindre perturbation en entrée se répercute grandement sur le hash.
4. La fonction est injective : pour deux messages distincts, leur hash le sont aussi. De manière plus formelle :
$$\forall (m, m') \in (\Sigma^*)^2, m \neq m' \Rightarrow h(m) \neq h(m')$$

On notera que dans la théorie la propriété 4 n'est pas réalisable ! On part en effet d'un ensemble infini (Σ^*) pour aller dans un ensemble de cardinal n (Σ^n). Il existera donc forcément deux messages m et m' tels que $h(m) = h(m')$. C'est ce que l'on appelle une collision.

Confusion La confusion [4] est un critère important d'une fonction de hachage cryptoraphique car bien que le hash doit être facile à calculer il ne faut cependant pas que la relation entre le message et le hash soit simple ! Si tel était le cas, la sécurité serait en péril. La confusion est donc le fait d'avoir une relation complexe entre m et $h(m)$.

Diffusion La diffusion [4] est le fait qu'un élément remarquable en entrée ne le soit plus en sortie, toute la particularité du message est diffusée dans le hash et aucune corrélation entre l'entrée et la sortie de la fonction de hachage ne peut être repérée.

Critère de l'avalanche et critère strict de l'avalanche Ces critères descendent tous deux de la diffusion et le second est juste une précision du premier.

Le critère de l'avalanche est un critère permettant de renforcer la difficulté d'inverser une fonction de hachage. Ce dernier assure que la moindre modification d'entrée se répercute bien dans l'ensemble

du hash.

Le critère strict de l'avalanche est plus précis que le critère de l'avalanche et établit que changer un seul bit sur l'entrée change chaque bit du hash avec une probabilité de 0,5 d'être changé. Sur la figure 2 les bits inchangés sont représentés en noir, le bit changé de l'entrée est représenté en rouge et les bits blancs sont les bits changés au cours de l'algorithme de SHA1, chaque ligne représentant un des 80 tours.

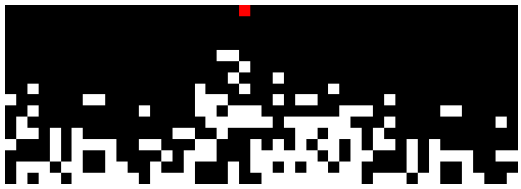


FIGURE 2 – Critère strict de l'avalanche sur SHA-1 sur les 46 premiers tours.

Le salage Le salage [5] est une protection supplémentaire au hachage protégeant notamment des attaques utilisant les tables arc-en-ciel (cf sous section 2.2.4). Le but de ce salage est d'éviter (ou plutôt de limiter) l'apparition de collisions en ajoutant un sel. Ce sel peut être soit aléatoire (on parle alors de salage dynamique) soit constant, tout les messages ont donc le même sel (on parle alors de salage statique). Actuellement le salage statique est mis caduc par les cryptanalystes et seul le salage dynamique est utilisé. L'utilisation du sel se fait de la manière suivante :

$$h_{salé}(m) = h\left(message \parallel sel\right) \parallel sel$$

Mais dans quel contexte le salage sert-il ? Ce dernier est généralement utilisé pour le stockage des mots de passe, permettant ainsi que deux informations identiques n'aboutissent pas au même hash final et permettant aussi la protection lors d'attaques.

2.2.3 Le hachage parfait et le hachage universel

Compte tenu des propriétés précédentes on pourrait se laisser rêver à avoir une fonction de hachage aléatoire ! Mais cela n'est pas possible, du moins pas de cette manière.

Certaines fonctions regroupées forment des familles de hachage universel [6] et on choisit aléatoirement des fonctions de hachage dans cette famille ayant des propriétés mathématiques particulières pour remplir la table de hachage, la plus importante est, en notant $\mathscr{H}$ l'ensemble fini de ces fonctions universelles :

$$\forall (m_1, m_2) \in \Sigma^*,\ m_1 \neq m_2,\ \mid \{h \in \mathscr{H}; h(m_1) = h(m_2)\} \mid \leq \frac{\mid \mathscr{H} \mid}{n}$$

Ainsi, la probabilité d'avoir des collisions en utilisant une famille de hachage universelle est de $\frac{1}{n}$ et la répartition est bien uniforme.

Le hachage parfait [7] est bien plus élaboré qu'une simple fonction de hachage et une table. Ce dernier ce fait en effet sur deux niveaux : les clefs sont d'abord hachées dans une première table avec une fonction de hachage universelle puis pour chaque 'case' de la table on crée une nouvelle table que l'on remplit avec une fonction de hachage unique selon la case de la table.

2.2.4 Attaques possibles

Pour les fonctions de hachage de nombreuses attaques sont possibles, certaines sont évitables et d'autres non, mais l'important et de s'en prémunir et d'en être conscient.

Attaque des anniversaires ou des collisions L'attaque des anniversaires [8] est une attaque par force brute s'appuyant sur le paradoxe des anniversaires : " Pour une classe de 23 élèves quelle est la probabilité qu'au moins deux élèves aient leur anniversaire le même jour ? " L'année comptant 365 jours on pourrait penser que cette probabilité est faible mais bien au contraire ! En calculant la probabilité que 2 personnes aient leur anniversaire le même jour dans un groupe de k personnes on trouve :

$$P(A_k) = 1 - \prod_{i=1}^{k-1} \left(1 - \frac{i}{365}\right)$$

Pour 23 élèves cette probabilité monte à 51% !

Et le rapport avec les fonctions de hachage ? Ce problème est tout a fait similaire avec la probabilité que nos messages aient le même hash ! Les élèves représentent les messages, le nombre de personnes représente le nombre de hashs disponibles et la date d'anniversaire représente le hash. Seulement il n'y a plus 365 possibilités de hash différents mais 2^n (on rappelle que n est la taille du hash et que ce dernier est un binaire) on aura donc :

$$P(A_k) = 1 - \prod_{i=1}^{k-1} \left(1 - \frac{i}{2^n}\right)$$

Ainsi pour k hashs si on veut une chance sur deux de trouver une collision il nous faut (approximativement car on considère que $\left(1 - \frac{1}{x}\right) = exp(x)$) :

$$exp\left(\frac{-k(k-1)}{2^n})\right) < \frac{1}{2}$$
$$\implies k(k-1) < 2^n ln(2)$$

On généralisera en disant qu' $\boxed{\text{une collision se trouve en } 2^{\frac{n}{2}}}$. Cette attaque par force brute doit être dans l'idéal la seule possible pour une fonction de hachage.

La première chose qu'il faut retenir de cette attaque est que la taille des hashs est très importante ! Pour un hash de longueur 160 on a $k = 2^{80}$ (pour donner un ordre de grandeur cela correspond à la masse des océans en kg) et on considère que le nombre de messages à calculer est grand pour un hash de longueur 128, au-dessous d'un certain seuil, l'algorithme sera vite mis caduc. La seconde chose à retenir est que toute fonction de hachage aura un jour une collision trouvée, mais avec plus ou moins de temps.

Préimage et second préimage L'attaque par préimage [9] est une attaque qui, pour un hash donné (notons le h_1), cherche une entrée m telle que : $h(m) = h_1$.

Une attaque de second préimage est donc une attaque qui pour un message d'entrée (noté m_1) consiste à trouver un second message,m_2, tel que $h(m_1) = h(m_2)$. La différence entre le second préimage et le préimage est la connaissance de m_1 qui apporte une information supplémentaire pour le cryptanalyste.

Attention, ces attaques ne sont pas à confondre avec l'attaque des anniversaires. Cela est plus visible avec une définition formelle de ces attaques :

$$\underline{\text{Attaque des anniversaires :}}\ \exists (m_1, m_2) \in (\Sigma^*)^2, h(m_1) = h(m_2)$$
$$\underline{\text{Attaque par préimage :}}\ \text{Pour } y \in \Sigma^n, \exists m \in \Sigma^*, h(m) = y$$
$$\underline{\text{Attaque par second préimage :}}\ \text{Pour } m_1 \in \Sigma^*, \exists m_2 \in \Sigma^*, h(m_1) = h(m_2)$$

Table arc-en-ciel Les tables arc-en-ciel [10] sont une structure de données inventée en 2013 et permettant pour un hash connu de retrouver le mot associé. On commence à prendre un message (quelconque, "qwer" sur la figure 3) que l'on hache puis réduit un nombre de fois prédéfini (ici 4). A la fin de ces hachages-réductions on récupère un nouveau hash (ici $269c241b$) que l'on rentre dans la table. On réitère ensuite avec un autre mot. Dans le futur si on veut récupérer le mot associé au hash $269c241b$ on sait qu'il faut hacher puis réduire 3 fois ce hash pour retrouver le message associé. Mais si le hash n'est pas directement dans la table on réduit puis hache ce hash jusqu'à avoir une correspondance dans la table : on est de retour au cas 1.

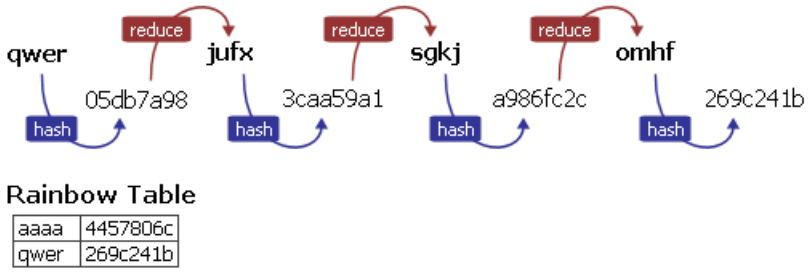


FIGURE 3 – Principe de fonctionnement des rainbow tables.

2.2.5 Les familles de fonctions de hachage

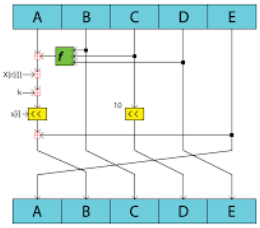


FIGURE 4 – Schéma de l'algorithme RIPEMD

RIPEMD RIPEMD ou RACE Integrity Primitives Evaluation Message Digest est une fonction de hachage cryptographique créée en 1992 par le contorsium européen "RIPE Consortium" dont la première collision a été trouvée en 1992 [11]. Cet algorithme est un algorithme de chiffrement par bloc et est donc précédé de l'algorithme de Merkle Damgard [12] consistant à compléter le message en binaire par un bit de 1 puis d'autant de bits de 0 qu'il est nécessaire pour que le message soit de taille voulue. Cet algorithme en a inspiré de nombreux autres comme RIPEMD-160.

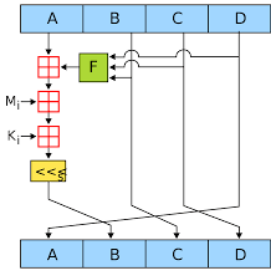


FIGURE 5 – Schéma de l'algorithme MD4

MD Les algorithmes MD sont très connus dans le monde de la cryptographie et cette famille est assez étendue et comporte notamment les algorithmes MD4 (cf figure 5 ci-contre) et MD5. L'algorithme MD5 a été inventé par Ronald Rivest, (également un des trois inventeurs du très connu algorithme RSA) en 1991 et une faille a été trouvée en 1996 permettant de trouver des collisions. Actuellement les tables arc-en-ciel permettent de craquer MD5 en moins d'une seconde.

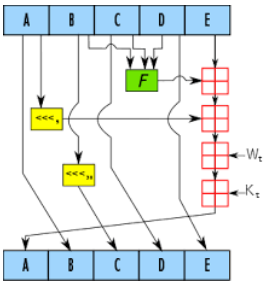


FIGURE 6 – Schéma de l'algorithme SHA-1

SHA La famille de SHA se compose en partie de SHA-0, SHA-1, SHA2, SHA-256 et SHA-3 (on étudiera en détail SHA-1 avec la sous-section 5.1.4). SHA-1 [3] est une fonction de hachage cryptographique conçue en 1995 par la NSA et dérive de SHA-0 pour lequel le NIST a jugé que la sécurité associée était insuffisante. SHA-1 a longtemps été utilisé par Google mais les certificats de SHA-1 ont été refusés à partir de 2017. En février de cette même année Google et le Centrum voor Wiskunde en Informatica trouve une collision. SHA-1 reste cependant utilisé pour de petites tâches comme la vérification de l'intégrité de fichier.

2.3 Les paramètres agissant sur le score d'un commentaire

2.3.1 La sensibilité à la casse

La sensibilité à la casse conduit à l'ajout d'un même mot comme un nouveau mot dans la table de hachage si celui-ci contient des majuscules à des positions différentes du mot initialement présent dans la table. Par exemple, considérons une table de hachage contenant les mots *loved*, *every*, *minute*, *of* et *it*. Nous souhaitons évaluer le score du commentaire suivant : *Loved every minute of it*. Puisque le programme est sensible à la casse, *Loved* sera considéré non présent dans la table et la note qui lui sera associée sera nulle. De plus, comme *Loved* influe positivement alors que les autres mots du commentaire sont neutres, le score obtenu sera faible. La sensibilité à la casse influence donc très fortement le score du commentaire et le sentiment qui lui sera associé.

2.3.2 La ponctuation

La ponctuation a quatre fonctions principales : une fonction syntaxique, une fonction sémantique, une fonction expressive et intonative et une fonction rythmique. Dans le cas de l'analyse des sentiments, la ponctuation peut influencer le score des commentaires.

Fonction sémantique Les signes de ponctuation facilitent la compréhension d'un texte et peuvent changer le sens d'une phrase. Utilisés seuls, ils peuvent créer du sens. Les exemples ci-dessous illustrent la fonction sémantique des signes de ponctuation.

— « On va manger les enfants ! »« On va manger , les enfants ! »
La première phrase signifie que les enfants vont être mangés tandis que dans la seconde on appelle les enfants à venir manger.

— « ... »signifient qu'on s'abstient de faire un commentaire.

— « - »signifie une absence de réponse d'un personnage lors d'un dialogue.

Fonction expressive et intonative La ponctuation permet de renforcer les sentiments et les émotions exprimés par l'auteur. L'utilisation des points d'exclamation permet de marquer la joie, la colère, l'impatience ou l'admiration. Les points d'interrogation permettent d'exprimer le doute, le scepticisme ou un questionnement. Quant aux points de suspension, ils peuvent exprimer l'ironie, la réserve ou un état de rêverie. Les exemples ci-dessous illustrent l'influence de la ponctuation dans les sentiments.

— Ils m'ont offert leurs plus sincères (?) excuses. Le point d'interrogation permet d'exprimer le doute sur la sincérité des excuses.

— Oui ! Ça y est ! Il a réussi son examen ! Les points d'interrogation permettent de renforcer l'expression de la joie.

2.3.3 Les contractions, les abréviations et les synonymes

Contractions Une contraction est la formation d'un mot à partir de deux autres à l'aide d'une apostrophe. En anglais, les contractions sont couramment utilisées que ce soit à l'oral ou à l'écrit

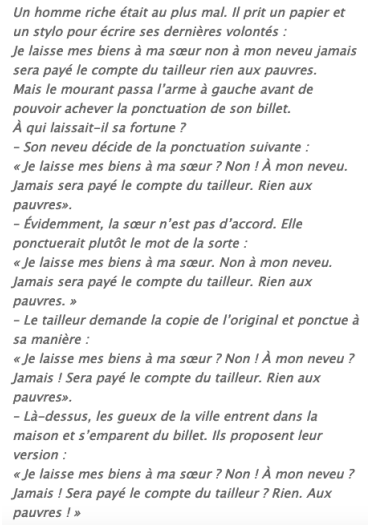


FIGURE 7 – Exemple illustrant l'influence de la ponctuation sur le sens d'une phrase.

lorsque le contexte n'est pas formel. Ils permettent de rendre le discours plus naturel. Le tableau ci-dessous regroupe une liste non exhaustive de contractions.

Forme contractée	Forme complète	Exemples
'm	am	I**'m** sorry.
's	is	He**'s** walking to school.
's	has	She**'s** been to England many times.
's	of	John**'s** car
're	are	They**'re** playing basketball.
've	have	I**'ve** been waiting an hour already.
'll	will	I**'ll** give you a lift.
shouldn't	should not	You **shouldn't** do things like that.
won't	will not	I **won't** get mad.
can't	can not	I **can't** carry that.
'd	would	I**'d** like to go.
'd	had	If you**'d** told me what was wrong I could have helped.
let's	let us	**Let's** go out to dinner.

Dans l'analyse des sentiments, il est plus judicieux d'ajouter dans la table de hachage tous les mots sous leur forme complète. Cependant certaines contractions ont plusieurs formes non contractées. Par exemple, *'s* peut correspondre à la troisième personne du singulier du verbe **BE** ou du verbe **HAVE** mais elle peut être également la marque du possessif. Dans ce cas, il est difficile de savoir quelle est la forme complète de la forme contractée. Pour pallier à ce problème, il faudrait pouvoir étudier la structure de la phrase.

Abréviations Comme dans toute langue, il existe des abréviations en anglais. Le tableau ci-dessous contient les principales abréviations.

Abréviation	Signification
ASAP	as soon as possible
idk	I don't know
idc	I don't care
aka	alias
btw	by the way

Tout comme les contractions, il faudrait ajouter dans la table de hachage la forme complète des abréviations. Cela conduirait à une amélioration de l'analyse des sentiments.

Synonymes Les synonymes sont extrêmement nombreux et avoir 8 mots à la place d'un seul, réduit la prise en compte de l'influence de toutes les notes : il est possible d'avoir 'escroquer' avec une moyenne de 2 et 'duper' avec une moyenne de 1. Ce problème réduit donc la fiabilité du modèle lors de l'évaluation d'un commentaire.

2.3.4 Les figures de style

La majorité des langues germaniques et latines regorgent de figures de style. Des plus simples et habituelles (euphémisme et comparaison) aux plus sophistiquées et improbables (enallage et synecdoque) ces dernières rythmes nos phrases sans même que nous nous en rendions compte.

Cependant certaines de ces figures de style jouent sur l'interprétation que nous faisons de ces phrases [13], nous pouvons par exemple penser aux oxymores. Prenons la phrase : « Je me sens définitivement triste comme une Bamba triste »de la chanson Bamba Triste de Pierre Billon, cette phrase, notée par un humain aurait une note de 0 ou 1, (prenons 0). Ainsi pour l'ordinateur tous les mots la composant auraient une note de 0 même « Bamba »qui est pourtant un mot très positif. Ainsi lorsque nous voudrons évaluer le commentaire "Nous avons fait la Bamba toute la nuit" , les autres mots que Bamba étant relativement neutres, cette phrase aura un sentiment triste alors qu'elle était en réalité très positive. Pour corriger ce problème il ne reste plus qu'à avoir une grande base de données et espérer que le mot déprécié revienne souvent afin de contrebalancer sa mauvaise note. La seconde solution serait aussi lors de la lecture d'un commentaire d'apprendre au programme à reconnaître les oxymores : un mot avec une très bonne note à côté d'un mot avec une très mauvaise note a beaucoup de chance d'être un oxymore, il faudra donc adapter le calcul du score.

Cette problématique des figures de style est aujourd'hui un défi des programmes et de l'intelligence artificielle : comment repérer le sarcasme, l'ironie, le cynisme ? Des études [14] ont montré que pour la plupart des usagers l'ironie se manifestait par deux propositions en contradictions qui peuvent être explicites ou implicites. Les contradictions sont dites explicites (23,58% des cas) lorsque les deux propositions sont présentes dans le commentaire de l'usager et implicites (76,42% des cas) si une des deux propositions n'est pas directement dans le commentaire. Par exemple « J'aime gagner 30% de moins qu'un homme à poste égal »est un commentaire à ironie implicite, ce même commentaire en ironie explicite serait : « J'aime gagner 30% de moins qu'un homme à poste égal alors que l'homme et la femme sont des égaux. », ici les deux propositions "Gagner moins" et "Être humain à égalité" sont clairement en opposition alors que dans le premier commentaire la seconde proposition n'apparaît pas et est déduite de l'utilisateur par le contexte et les problématiques actuelles.

Maintenant comment appliquer cela à un algorithme ? Comment un algorithme pourrait-il

connaître les problématiques de la société qui est en constante évolution ? Ce dernier devrait analyser le contexte du commentaire, capter ce dernier à l'aide d'autres commentaires autour de ce dernier.

2.3.5 Opinions explicites et implicites

De pair avec les figures de style les opinions peuvent être explicites ou implicites comme le montre la thèse de Jihen KAROUI (référence [14]) avec cette exemple parlant :

— Opinion explicite : Quel film magnifique. J'étais tellement captivé que JE N'AI PAS BOUGÉ UNE SECONDE DE MON SIÈGE.

— Opinion implicite : Nous avons acheté en mars ce matelas. Après l'avoir essayé plusieurs jours, sur-prise :RÉVEIL DANS UNE CUVETTE CREUSÉE AU COURS DE LA NUIT.

Nous voyons bien avec cette exemple que le programme aura bien du mal à déterminer si le second commentaire est positif ou négatif alors que le premier donne des indices clairs avec les mots 'magnifique' et 'captivé'. Le fait de comprendre et pouvoir analyser les commentaires implicites reste un défi pour l'intelligence artificielle.

3 Évaluation d'un commentaire

3.1 Création de la table et structures utilisées

3.1.1 Des structures simplistes

Présentation et programmation Pour commencer le projet on reprend les structures de liste et de table faites lors des TPs de SD. On construit donc la table de hachage avec une liste remplie d'éléments et une taille fixe. Cependant, on modifie la structure de l'élément en lui-même. En effet, ce dernier stockera maintenant une $clef$, une moyenne mais aussi le nombre de notes notés respectivement moy et $nbNote$. Ainsi lorsque l'on voudra modifier la moyenne en lui ajoutant une nouvelle note, il suffira de faire :

$$moy = \frac{moy.nbNote + note}{nbNote + 1}$$

On définit un élément vide/nul comme ayant une moyenne de 0 et un nombre de notes de 0, ainsi aucune confusion ne sera possible avec une clef inexistante.

Problèmes Le principal défaut de ces structures est le fait qu'elles ne permettent pas la gestion des collisions. Ainsi lorsqu'une collision apparaît avec un nouveau mot, ce dernier remplace purement et simplement l'ancien et initialise sa moyenne par sa note et son nombre de notes par 1. Mais ces collisions sont-elles fréquentes et handicapantes ou pouvons-nous passer outre ?

Là se pose le choix de la fonction de hachage ! Pour une table de hachage de taille N et un mot m de longueur n noté $m = (x_1 x_2 ... x_n)$, avec $(x_i)_{i \in [\![1;n]\!]}$ chacun des caractères de m, la fonction de hachage actuelle donne le résulat suivant :

$$hash(m) = \left(n + \sum_{i=1}^{n} int(x_i) \right) \ mod\ N$$

Est-il donc possible de jouer sur la taille du tableau pour éviter les collisions ? Malheureusement, ici, cela ne suffit pas. Et en voici la preuve : chaque caractère est codé en ASCII entre 33 et 126. Supposons que le mot ne soit composé que de caractères codés en 126 et que sa longueur soit de 15 (en anglais la moyenne de la longueur d'un mot est de 8.23 lettres, on choisit consciemment un mot long). Le hash serait donc de 2016 $mod\ N$. En prenant donc un tableau de taille supérieur à 2017, ce mot serait à l'indice 2016 dans la table. Ce mot étant le plus gros, tous les autres seraient dans des indices inférieurs.
Cela n'est pas inquiétant si on prend un nombre de mots inférieur à 2016 mais nos données sont dans un dossier de 8529 lignes ayant au plus 300 caractères. Nous pouvons alors en déduire qu'il y a un peu moins de 8529*(300/8.23) mots, soit un total de 310 900 mots. Ainsi seulement une très petite partie des mots pourra être stockée mais sans même l'assurance que leur moyenne soit la bonne. En effet si une collision survient, le nouveau mot remplace totalement l'ancien. Si un mot ayant précédemment été supprimé vient à reprendre sa place, sa moyenne sera réinitialisée de même que ses notes, ce qui fausse totalement les résultats.

Analyse et solutions La combinaison de ces structures et de cette fonction de hachage, en l'état, n'est pas viable ! Pour régler le problème deux solutions s'offrent à nous : modifier les structures en

incluant une gestion des collisions afin d'être sûr que tous les mots présents dans le fichier le soit dans la table ou bien faire une autre fonction de hachage, bien plus performante afin de minimiser le risque de collisions. Cependant le risque zéro n'existe pas ! En effet une fonction de hachage part d'un ensemble infini vers un ensemble fini, les collisions sont donc inévitables et il faut se préparer à les gérer.

3.1.2 De nouvelles structures avec la gestion des collisions

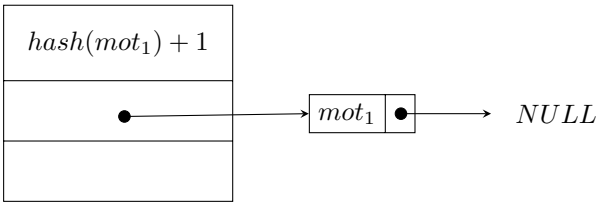
Présentation et programmation La gestion des collisions est un aspect fondamental pour l'intégrité de nos résultats dans la table de hachage. Prenons l'exemple suivant :

Soit un mot $m_1 = x_1x_2...x_n$ et un autre mot $m_2 = x_nx_{n-1}...x_1$ avec n un entier quelconque. On remarque d'après le calcul du hachcode, on a hash(m_1)=hash(m_2) d'après l'équation du paragraphe précédent. Cette égalité est observée en particulier si :

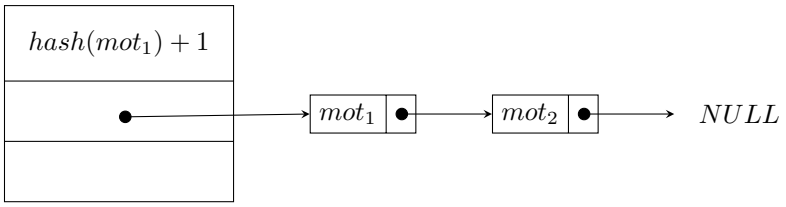
$$(0 \leq) Espace\_Mémoire_{Tablehachage} < Espace\_Mémoire_{Données}$$

Le programme initial que l'on a créé ne peut pas gérer ce type de désagrément. Ainsi, il est nécessaire de faire évoluer le programme pour éviter toute perte d'informations à cause des collisions.

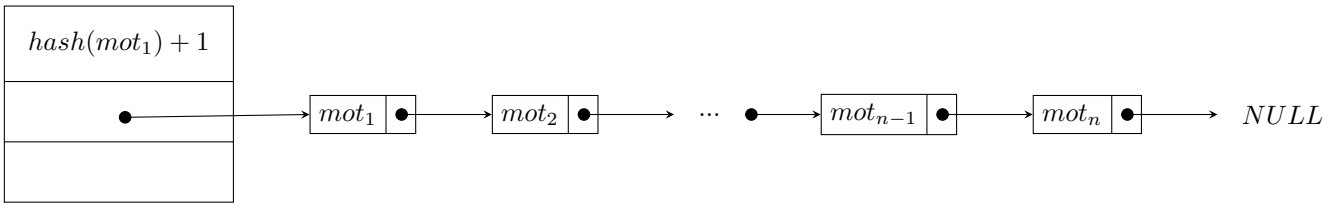
Afin de conserver les informations ayant le même hachcode, nous allons introduire une "nouvelle" dimension à la table. En effet, on va pour cela créer des sous-tables si une collision se produit. Reprenons le cas du mot m_1 et m_2, sous la forme d'un schéma. Lors de l'entrée du mot m_1 (en supposant que la table est vide), on a :



Dans l'ancienne version, lors de l'entrée du deuxième mot, l'information du premier mot se faisait remplacer par le deuxième mot. Ainsi, l'information du premier disparaissait. C'est ainsi, que l'on ajoute les sous-tables : lors de l'apparition d'un mot ayant le hashcode d'une case déjà remplie, le programme va créer une sous-table dont le pointeur est sauvegardé dans la "case" de mot_1.



Et ainsi de suite, dès que la table rencontre une collision avec un nouveau mot, il va à la fin de la sous-table pour l'ajouter.



Analyse Il y a plusieurs étapes dans l'ajout d'un mot dans la table de hachage :

- Calcul du hachcode de l'élément
- Regarder si l'élément existe ou non dans la table de hachage
- Mettre à jour la table de hachage

Lorsque l'on met un élément dans la table de hachage, on calcule d'abord sa valeur (son hachcode) afin de le mettre et de reprendre cette valeur très facilement. Le calcul (qui se fait une fois par mot) a une complexité qui diffère selon la fonction que l'on utilise. Soit m la taille du mot. Dans le cas de la fonction naïve on a une complexité de la taille du mot : on a une complexité de l'ordre de $\boxed{\text{O(m)}}$ (cf. 5.1.1). Dans le cas de la fonction SHA-1, l'ensemble des calculs devient plus complexe. Néanmoins, la complexité reste toujours de l'ordre de $\boxed{\text{O(m)}}$ (cf. 5.1.4). On a une complexité linéaire.

Après avoir calculé le hachcode de l'élément, il faut regarder la table, si on ajoute un nouveau mot ou s'il faut mettre à jour les informations d'un mot : c'est la fonction IsIn qui nous renvoie soit NULL (le mot n'existe pas) ou bien il renvoie le pointeur vers l'adresse de l'élément (le mot existe déjà dans la table).
La fonction IsIn utilise la fonction tableHachage_get demandant une adresse vers une case d'un élément de la table de hachage. Celui-ci va d'abord aller jusqu'au hachcode correspondant. On regarde si la mot correspond à la clé de la case. Il y a trois issues possibles :

- La clé correspond et on renvoie l'adresse de la "case" (la structure contenant l'information du mot).
- La clé ne correspond pas au mot :
  - Il existe une sous-table, on va alors regarder celle-ci.
  - Il n'existe pas de sous-table, alors on renvoie la valeur NULL.

Ainsi, la fonction IsIn renvoie bien deux valeurs possibles, l'adresse vers une structure contenant les informations de la clé ou bien la valeur NULL. Dans le pire des cas (le cas où on ne trouve pas de valeur), avec n, le hachcode de la valeur et s, le nombre de sous-tables, on obtient une complexité de l'ordre de $\boxed{\text{O(n+s)}}$, soit une complexité linéaire. Dans le cas où la fonction IsIn renvoie une valeur non NULL, la fonction ajouteAncienMots est appelée : il suffit de mettre à jour les valeurs de cette structure. Dans le cas où IsIn nous renvoie une valeur NULL, il faut ajouter une structure contenant le mot dans la table de hachage avec la fonction ajouteNouveauMots. On a alors deux possibilités :

- Aucun mot n'a la même valeur dans la table de hachage. On ajoute alors ce mot dans cette structure sans aucune difficulté.

— Il existe un ou plusieurs mots ayant la même valeur. On va ainsi à la fin des sous-tables pour y ajouter une structure contenant les informations du mot.

L'ensemble des cas possibles sont programmés dans la fonction. Pour x mots à traiter par la fonction, on obtient ainsi une complexité linéaire de l'ordre de $\boxed{\text{O(x*s)}}$

3.2 Remplissage de la table

3.2.1 1ère version : sans gestion des collisions

Bien que cette version sans gestion des collisions ne pourra pas être utilisée comme version finale elle permettra de fournir une structure globale pour remplir la table avec la gestion des collisions et certaines fonctions pourront être réutilisées.

Présentation Pour remplir la table de manière claire nous décidons de décomposer le programme en 3 grandes sous-fonctions : le découpage des commentaires, l'insertion des mots et la lecture de tous les commentaires.

Programmation

1. Découper un commentaire : Pour découper un commentaire selon les espaces, la ponctuation, la tabulation et les symboles mathématiques nous utilisons la fonction *strtok*. Cette fonction prend en entrée la chaîne de caractères à découper et les délimiteurs et renvoie le premier mot avant ces délimiteurs. Cependant la fonction *strtok* a une "mémoire" : si lors d'un second usage on rentre $NULL$ à la place d'une chaîne de caractères, *strtok* reprendra son découpage au mot après le mot renvoyé la dernière fois. Une fois arrivée au bout de la chaîne de caractères, la fonction renvoie $NULL$.
   En sachant cela il suffit pour découper le commentaire de créer un pointeur de chaîne de caractères de taille suffisamment grande (on prendra la constante $SIZE$) pour stocker les mots du commentaire. On crée ensuite une variable *mot* qui est de type $char*$ et qui stockera le résultat de *strtok*. On récupère la taille de cette variable mot et avec un malloc, on crée une chaîne de caractères de taille suffisante. On copie ensuite (avec *strcpy*) le contenu de *mot* dans le malloc. Puis *mot* prend la valeur suivante de *strtok*. On réitère cette étape jusqu'à ce que mot soit $NULL$ ou que l'on ait dépassé le nombre de mots maximum autorisé par la constante $SIZE$

2. Ajouter des mots : Une fois le mot récupéré il faut l'ajouter dans la table et deux cas s'ouvrent à nous : le mot est déjà dans la table et il suffit de modifier sa moyenne et son nombre de notes ou bien le mot n'est pas encore dans la table et on crée un nouvel $ELEMENT$ qui stocke les informations de ce mot puis on appelle la fonction $tableHachage\_ajout$ qui ajoute un mot dans la table. Pour savoir si le mot est déjà dans la table ou non on appelle la fonction $IsIn$

3. Remplir la table : Dans un premier temps, on vérifie que le fichier a bien été ouvert. Si c'est le cas, on parcourt le fichier ligne par ligne avec la fonction *fgets*, sinon on affiche un message d'erreur. Pour chaque ligne, on regarde s'il y a un retour à la ligne à l'aide de la fonction *strchr*. Si c'est le cas, on le remplace par le caractère nul terminal.

Ensuite on ajoute ce même caractère à la fin de chaque ligne et on effectue les pré-traitements additionnels qui sont détaillés dans la sous-section 4.1. À l'aide de la fonction *split*, on découpe le commentaire selon les délimiteurs choisis. La fonction nous renvoie un tableau contenant tous les mots. Comme la note correspond au premier mot du tableau, on convertit celui-ci en entier avec la fonction *atoi* et on la stocke dans une variable.
On parcourt ensuite le tableau et on ajoute un par un tous les mots dans la table de hachage avec la fonction *rentreMot* excepté les contractions et le caractère &. En effet, si le mot est une esperluette, on ajoute son équivalent, and, dans la table et si le mot est une contraction, on ajoute sa forme complète.

Tests Afin de s'assurer, malgré la courte durée de vie de ces fonctions, de leur véracité des tests ont été effectués sur un fichier *datatest.txt* qui a été un fichier test du projet contenant des phrases simples et couvrant les différents cas de ponctuation.

3.2.2 2nd version : avec gestion des collisions

Programmation et explications Lors du remplissage de la table, comme dit dans le paragraphe précédent, il y a deux cas possibles :

— Le mot existe dans la sous-table
— Le mot n'existe pas dans la sous-table

La particularité de ce programme est qu'il résout le problème du second point évitant ainsi la perte d'informations tout en respectant les fondamentaux d'une table de hachage.

Fonctions intermédiaires Pour éviter de créer des collisions, il faut tout d'abord créer une nouvelle structure afin de créer les sous-tables. La structure _next contient les éléments suivants :

— ELEMENT* val ; : il contient les valeurs d'un mot.
— next* nextWord ; : c'est un pointeur vers la prochaine structure _next, qui est le prochain élément de la sous-table.

On a la modification de la fonction tableHachage_get, ajouteAncienMots et ajouteNouveauMots avec l'ajout de la même boucle while nous permettant d'aller dans les sous-tables. Cette boucle while se termine car l'ensemble des sous-tables possède un ensemble de mot fini.

Problème et solution Le problème principal de la création de sous-tables est la gestion des fuites mémoires au fur et à mesure que l'on crée des sous-tables. Pour remédier à ce problème, il nous faut libérer l'espace mémoire inutilisé tel que la création de structures contenant la clé NULL dans la fonction tableHachage_get.

Tests L'ensemble des tests sont exactement les mêmes que ceux de la première version du programme à l'exception qu'il faut s'assurer que l'insertion de deux mots ayant la même valeur soit retrouvée par la suite (ce qui est le cas).

3.3 Calcul du score prédit pour le commentaire saisi par l'utilisateur

Présentation Dans un premier temps, on demande à l'utilisateur de saisir un nouveau commentaire sur l'entrée standard du terminal. Puis on calcule le score prédit pour ce nouveau commentaire

avec la fonction *score*. Enfin, on affiche le score obtenu et le qualificatif associé au commentaire (négatif, quelque peu négatif, neutre, quelque peu positif, positif).

Programmation La fonction *score* prend en paramètres le commentaire, la table de hachage et le numéro de la fonction choisie. Tout d'abord, on effectue les prétraitements sur le commentaire. On convertit toutes les lettres du commentaire en minuscule avec la fonction *maj2min* puis on supprime tous les chiffres avec *removeNumber*. Ensuite on découpe le commentaire avec la fonction *split* qui renvoie un tableau de chaînes de caractères dépourvues de signes de ponctuation, de symboles mathématiques, de caractères spéciaux et de la tabulation. Puis on initialise la variable **score** à 0 et on parcourt tous les mots du commentaire avec une boucle **while**. Si le mot parcouru est une esperluette, alors on ajoute à la variable **score** la moyenne du mot "and" et si c'est une contraction, on lui ajoute la moyenne du mot ou des mots qui constitue(nt) sa forme complète. Enfin, on retourne le résultat de la division de **score** par le nombre de mots dans le commentaire.

4 Prétraitements additionnels et études statistique des données

4.1 Prétraitements additionnels

Pour améliorer la qualité des résultats d'analyse des sentiments, nous avons effectué des prétraitements additionnels.

Nous avons d'abord ajouté la fonction *maj2min* qui convertit toutes les lettres en minuscule. Cette fonction parcourt la chaîne de caractères en paramètre et convertit chacun de ses caractères en minuscule en utilisant la fonction *tolower*. Lors de l'allocation mémoire de la chaîne convertie, on n'oublie pas de compter le caractère nul terminal. La complexité de cette fonction est de $\boxed{O(n)}$ avec **n** la taille de la chaîne passée en paramètre.

Nous avons ensuite implémenté les fonctions *isNumber* et *removeNumber* afin de supprimer tous les nombres dans les commentaires excepté la note. La fonction *isNumber* renvoie 1 si le caractère en paramètre est un chiffre, 0 sinon. La complexité de cette fonction est de $\boxed{O(1)}$. La fonction *removeNumber* conserve la note associée au commentaire puis parcourt le reste de la chaîne de caractères en paramètre. Si le caractère parcouru n'est pas un chiffre, on l'ajoute à la variable allouée. On n'oublie pas d'ajouter le caractère nul terminal à la fin. La complexité de *removeNumber* est de $\boxed{O(n)}$.

Nous avons également supprimé les signes de ponctuation, **.!?, :;-'’**, des caractères spéciaux, **#$^**, les symboles mathématiques, **+/*=**, et la tabulation. Pour cela, nous avons ajouté les caractères que nous voulions supprimer en paramètre de la fonction *strtok* utilisée pour découper une chaîne de caractères dans *split*.

Nous avons remarqué que les critiques de films contenaient de nombreuses contractions telles que *'m*, *'s* et *'ll*. En effet, les contractions sont couramment utilisées en anglais à l'oral et également à l'écrit lorsque le contexte n'est pas formel. Afin d'améliorer la qualité des résultats d'analyse, nous avons ajouté dans la fonction *filledTab* des comparaisons permettant d'ajouter dans la table de hachage la forme non contractée des principales contractions verbales que nous avons pu retrouver dans les commentaires. Le tableau ci-dessous regroupe les principales contractions.

Forme contractée	Forme complète
'm	am
's	is ou of
're	are
've	have
'll	will
won't	will not
can't	can not
n't	not
'd	would
let's	let us

4.2 Études statistiques des données

Pour les études statistiques effectuées, nous avons utilisé la table de hachage obtenue avec gestion des collisions.

4.2.1 Étude des mots les plus récurrents et des moins présents

Présentation La première étude que nous avons réalisée concerne les mots les plus récurrents et les moins présents dans la table. Pour cela, nous nous sommes appuyés sur l'occurrence des mots dans la table. Dans un premier temps avec la fonction *avl_sortedOccurrence*, nous avons trié toutes les occurrences présentes dans la table en utilisant une structure de données efficace pour ce type de problème : les arbres et en particulier les AVL. Nous avons ensuite inséré toutes les occurrences triées dans une liste avec la fonction *list_sortedOccurrence* afin de pouvoir parcourir ses éléments par la suite. Puis nous avons implémenté la fonction *getSameOcurrenceWords* qui renvoie tous les mots de la table qui ont l'occurrence passée en paramètre. Enfin, nous avons ajouté la fonction *orderedWords* qui renvoie un tableau contenant tous les mots rangés du moins présent au plus présent dans le fichier de données. Cette fonction parcourt la liste de toutes les occurrences triées dans l'ordre croissant puis ajoute dans le tableau tous les mots qui ont l'occurrence parcourue. Les fonctions *print_leastRecurrentWords* et *print_mostRecurrentWords* affichent respectivement les n mots les moins récurrents et les n mots les plus récurrents.

Programmation Pour ranger les mots du moins présent au plus présent, nous avons utilisé deux structures de données différentes : les arbres et les listes simplement chaînées. Nous nous sommes donc servis des fonctions sur les arbres que nous avions réalisées lors du lab3 de SD. Nous nous sommes également servis des fonctions sur les listes simplement chaînées réalisées lors du lab1 de SD mais nous avons tout de même dû les modifier afin de les adapter à notre problème. En effet, dans le fichier *listSimpleChain_int.c*, nous avons modifié le type de la valeur contenue dans les cellules en entier. Nous avons également ajouté la fonction *copy* qui copie une liste, la fonction *concat* qui concatène deux listes et la fonction *freeList* qui libère une liste en mémoire.

4.2.2 Calcul du nombre de mots dans la table

Présentation et programmation La fonction *getTableNumberWords* renvoie le nombre de mots contenus dans la table de hachage. Pour cela, on initialise un compteur à 0 puis on parcourt tous les éléments de la table et des sous-tables un à un tout en incrémentant le compteur. Pour parcourir la

table et les sous-tables ou crée tout d'abord une boucle *for* qui parcourt la table cellule par cellule et pour chaque cellule on regarde combien de mots il y a dans la sous-table avec une boucle *while* ayant pour condition d'arrêt le pointeur vers le mot suivant de la sous table $NULL$.

Analyse La complexité de la fonction *getTableNumberWords* est de $\boxed{O(n^2)}$ car dans le pire cas tout les mots de la table ont le même hash. Il y aurait donc un mot et $n-1$ sous-tables. Le parcours se ferait donc en $O(n^2)$.

Test Le nombre de mots contenus dans la table de hachage est de $\boxed{15081}$.

4.2.3 Affichage des mots les moins et les plus récurrents

Présentation et implémentation La fonction *print_leastRecurrentWords* affiche les n premiers mots du tableau obtenu avec la fonction *orderedWords* qui contient les mots rangés du moins récurrent au plus récurrent. La fonction *print_mostRecurrentWords* quant à elle affiche les n derniers mots du tableau.

Tests

Les 10 mots les moins présents	Les 10 mots les plus présents
y	the
ba	a
ka	and
ke	of
bv	to
ou	s
r&b	is
r&d	it
s&m	that
aaa	in

4.2.4 Calcul de la taille du mot le plus long pour la fonction SHA1

Présentation et programmation La fonction *sizeLongestWord* prend en paramètre une table de hachage et renvoie la taille du mot le plus long. Pour cela, on initialise la variable **sizeMax** à 0 puis on parcourt tous les éléments de la table et des sous-tables. Dès que l'on rencontre un mot ayant une taille supérieure à **sizeMax**, on met à jour sa valeur. La taille d'un mot est obtenu avec la fonction *strlen*. Après avoir parcouru toute la table, on renvoie **sizeMax**.

Analyse La complexité de la fonction *sizeLongestWord* est de $\boxed{O(n^2)}$ car dans le pire cas tout les mots de la table ont le même hash. Il y aurait donc un mot et $n-1$ sous-tables. Le parcours se ferait donc en $O(n^2)$.

Test La taille du mot le plus long est de $\boxed{18}$ caractères.

4.2.5 Études statistiques sur le score des mots

Présentation Pour afficher les mots positifs, négatifs et neutres, nous nous sommes appuyés sur le score des mots. Pour cela, nous avons d'abord trié les différents scores présents dans la table avec la fonction *avl_sortedScore* qui fonctionne de la même façon que *avl_sortedOccurrence* présentée dans la partie 4.2.1. La seule différence est le type des éléments triés : float pour les scores et int pour les occurrences. Nous avons ensuite inséré tous les scores triées dans une liste simplement chaînée de float avec la fonction *list_sortedScore* afin de pouvoir parcourir ses éléments par la suite. La fonction *getWords* prend en paramètres une table de hachage, une valeur minimale et une valeur maximale correspondants aux bornes à l'intérieur desquelles les mots sont considérés positifs, négatifs ou neutres. Cette fonction retourne un tableau contenant les mots dont le score est compris compris entre les deux bornes.

Programmation Tout comme dans la partie 4.2.1, nous avons réutilisé les fonctions sur les arbres et les listes simplement chaînées réalisés lors des labs de SD mais nous les avons réadaptées au float.

Test

Mots négatifs	Mots quelque peu négatifs	Mots neutres	Mots quelque peu positifs	Mots positifs
incoherent	didactic	for	young	joyous
disappointment	achieved	glacial	fans	heartwarming
incomprehensible	hawaiian	mystery	entertainment	delightfully
unnecessary	unless	intrigue	romantic	enthusiastic
frustratingly	tourists	twenty	dance	brilliantly
unsympathetic	ballerina	solution	memorable	gorgeous
rut	misfortune	narrative	epic	romantics
creek	protestors	institutionalized	technological	tumultuous
bygone	coincidence	opportunities	contemporaries	powerful
garbage	brilliance	practically	summertime	charmer

5 Amélioration de la table

Avec la création des sous-tables, les collisions n'impactent plus l'évaluation d'un commentaire. Mais comment s'assurer que l'évaluation de ce commentaire soit optimal ? Comment être sûr que le nombre de sous-tables, et donc la place prise pour stocker une table, le temps de recherche d'un élément ou bien le temps de la construction d'une table, soit raisonnable ?

Il parait évident que plus le nombre de sous-tables est important moins la table sera de bonne qualité. Et pour éviter ces sous-tables il faut éviter les collisions. Le levier pour améliorer la table sera donc la fonction de hachage !

5.1 Présentation des fonctions de hachage

Dans cette partie on considérera les notations suivantes : la taille de la table de hachage est notée N, un mot est noté m et sa longueur est n.

Le mot est représenté avec ses caractères $(x_i)_{i\in[\![1;n]\!]}$: $m = (x_1x_2...x_n)$.

Pour les fonctions où apparaissent un poids, ce dernier est noté p.

5.1.1 Fonction naïve

Présentation On appelle fonction naïve la fonction de hachage présentée au début donnant le résultat suivant :

$$hash(m) = \left(n + \sum_{i=1}^{n} int(x_i) \right) \ mod\ N$$

Programmation Cette fonction, donnée par le TP de SD sur les tables, est constituée d'une boucle while additionnant la valeur des caractères jusqu'à la fin du mot puis ajoutant au résultat la longueur du mot.

Analyse du programme Sa complexité est donc en $O(n)$. Elle est précisément de $n + 2$: le while, l'addition de la longueur du mot et le modulo.

Dans cette fonction de hachage aucun malloc n'a été nécessaire.

5.1.2 Fonction naïve avec les poids

Présentation Comme vu en 3.1.1 la fonction précédente est mauvaise car bornée et générant énormément de collisions. Mais pourquoi en génère-t-elle autant ? Quels sont les critères faisant que chaque mot est considéré comme unique et devrait avoir une place unique dans notre table ?

Ces critères sont au nombres de deux : les lettres (prises en compte par la fonction naïve) et leur place dans le mot (ignorée par la fonction naïve). Si la fonction naïve génère autant de collisions c'est tout d'abord parce qu'elle ne prend pas en compte l'ordre des lettres. En effet il existe dans chaque langue nombre d'anagrammes (par exemple 'police' et 'picole' en français) et sans tenir compte de l'ordre des lettres dans un mot ces derniers ont le même hashcode.

Pour remédier à ce problème nous avons eu l'idée d'attribuer un poids à chaque place dans le mot. Ce poids dépendra de la longueur du mot le rendant encore plus unique pour chaque mot. Ainsi, le hash sera de la forme :

$$hash(m) = \left(n + \sum_{i=1}^{n} int(x_i) \cdot p_{n,i} \right) \ mod\ N$$

Mais comment choisir ce poids p ? La contrainte de ce choix est liée au codage des nombres par un ordinateur. En effet aller à plus de $2^{32} - 1$ n'est pas possible (ou alors en utilisant des structures telles que des listes, mais ce n'est pas le choix fait ici). Il est donc nécessaire de faire attention au résultat. Pour le poids, nous avons décidé de le mettre sous forme de $k^{n-i} avec\ k \in \mathbb{R}$ et pour ne pas dépasser la limite nous nous sommes aidés de la longueur des mots présents dans les données $movie_reviews.txt$. Avec une fonction décrite en 4.2 il vient que le mot le plus long est de taille 18. Ainsi, après quelques tests et quelques hypothèses sur le mot (il ne contient pas plus de 2/3 fois z, il est de longueur 18, les lettres les plus utilisées sont globalement au début de l'alphabet [15] et ont donc une valeur plus faible que 110 /111 ...) nous avons choisi de prendre $k = \sqrt{5}$. Finalement :

$$hash(m) = \left(n + \sum_{i=1}^{n} int(x_i) \cdot 5^{\frac{n-i}{2}} \right) \ mod\ N$$

Programmation La première chose à faire a été la création de la fonction *int psc*(*int a*, *int b*) calculant a^b via une boucle *while*.
Cette fonction de hachage se base sur une boucle for et non plus sur une boucle *while* car l'exposant de la puissance dépend du tour de la boucle.

Analyse du programme Pour cette fonction la complexité est en $O(n^2)$ et plus précisément, en notant $C(n)$ sa complexité :

$$\begin{aligned} C(n) &= 1 + 1 + \sum_{i=1}^{n}(n-i) \\ &= 2 + \frac{n(n-1)}{2} \end{aligned}$$

Dans cette fonction de hachage aucun malloc n'a été nécessaire.

5.1.3 Fonction récursive

Lors de nos recherches sur les fonctions de hachage [16] nous avons été intrigués de trouver une fonction de hachage récursive et par curiosité et étant donné sa facilité de programmation nous nous sommes lancés dans sa programmation et son analyse.

Présentation La fonction de hachage récursive se présente simplement par :

$$f(x) = \begin{cases} 0 & \text{si } x = 0 \\ 8f(\frac{x}{2}) + x & \text{sinon} \end{cases}$$

Programmation Pour la programmer nous avons codé la fonction récursive en elle-même (f) à part avec un cas d'arrêt via un if si x vaut 0 et l'appel à cette même fonction avec le paramètre divisé par deux dans le cas contraire.

On crée ensuite la fonction générale qui transforme le mot en entier avec le même fonctionnement que dans la fonction naïve avec poids et qui appelle ensuite la fonction f avec cette valeur. On fait un modulo pour s'assurer que le hash soit dans les bornes de la table.

5.1.4 SHA-1 et ses adaptations

Présentation Comme vu dans la sous-section 2.2.5 SHA-1 est une fonction de hachage qui a fait ses preuves et nous nous sommes lancés dans le défis de sa programmation ! Cette fonction de hachage travaillant sur des binaires le premier réflexe a été de créer une fonction transformant un mot en binaire (cf point 1), puis d'appliquer par-dessus l'algorithme de Merkle-Damgard (cf point 2), de découper le bloc obtenu (cf point 3), d'appliquer la suite définie sur les blocs 80 fois (cf point 4) et enfin d'adapter.

1. Transformation du mot en binaire : Pour transformer un mot en binaire deux solutions s'offrent à nous : additionner chaque caractère et le coder en binaire (ce qui provoquerait des collisions) ou prendre chaque caractère du mot et le coder en binaire sur 9 bits. En effet, le caractère avec la valeur la plus grande ('~ ' pour une valeur de 126) est codé par 0b1111110,

sur 7 bits. Ainsi 9 bits suffisent largement. On concatène ensuite les résultats. Chaque lettre aura alors un poids différent dans le résultat binaire final, ce qui assure l'unicité du mot.

2. Algorithme de Merkle Damgärd : L'algorithme de Merkle Damgärd, comme vu en 2.2.5 a pour but de compléter un message afin que sa longueur soit un multiple de 512 en ajoutant un 1 puis une suite de 0. Ici, le plus long mot étant de 18 caractères et chacun étant codé sur 9 bits, le plus grand mot en binaire sera de $9 \times 18 = 162 < 512$. Ainsi une fois le mot en binaire passé dans l'algorithme de Merkle Damgard, sa taille sera de 512.

3. Découpage des blocs $(M_i)_{i\in[\![0;15]\!]}$ et création des blocs $(W_i)_{i\in[\![0;79]\!]}$ Une fois le mot binaire passé par Merkle Damgard (on l'appellera dès maintenant le mot allongé) on le découpe en 16 blocs de 32 bits que l'on appelle $(M_i)_{i\in[\![0;15]\!]}$ et à partir desquels on crée les $(W_i)_{i\in[\![0;79]\!]}$ de 32 bits avec :

$$W_i = \begin{cases} M_i & \text{si } i \in [\![0;15]\!] \\ (W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16}) \ll 1 & \text{si } i \in [\![16;79]\!] \end{cases}$$

Où on a noté $\ll k$ la rotation de k-bits (les k premiers bits sont déplacés à la fin de la chaîne de caractères et les autres sont poussés à l'avant de la chaîne) et $\oplus$ le xor. Ainsi tous les $(W_i)_{i\in[\![0;79]\!]}$ dépendent du mot binaire allongé, ce qui garanti que le moindre changement dans le mot initial se répercute sur l'entièreté du hash.

4. Suite sur 80 tours de SHA-1 Pour calculer le hash de notre mot, on définit 5 suites : $(A_i)_{i\in[\![0;79]\!]}$, $(B_i)_{i\in[\![0;79]\!]}$, $(C_i)_{i\in[\![0;79]\!]}$, $(D_i)_{i\in[\![0;79]\!]}$, $(E_i)_{i\in[\![0;79]\!]}$ que l'on initialise avec des constantes données (H_0 sur [17]) Notre hash sera alors la concaténation de l'addition des premiers et derniers résultats des suites.
Pour calculer les termes des suites, on définit une donnée K qui intervient dans la définition des suites et qui change au tour k et une fonction ϕ :

$$K_i = \begin{cases} 0x5a827999 & \text{si } i \in [\![0;19]\!] \\ 0x6ed9eba1 & \text{si } i \in [\![20;39]\!] \\ 0x8f1bbcdc & \text{si } i \in [\![40;59]\!] \\ 0xca62c1d6 & \text{si } i \in [\![60;79]\!] \end{cases}$$

$$\phi_i(x,y,z) = \begin{cases} ITE(x,y,z) = xy \oplus \bar{x}z & \text{si } i \in [\![0;19]\!] \\ MAJ(x,y,z) = xy \oplus xz \oplus yz & \text{si } i \in [\![40;59]\!] \\ XOR(x,y,z) = x \oplus y \oplus z & \text{sinon} \end{cases}$$

En notant $\square$ l'addition modulo 2^{32}, les suites sont ensuite définies par :

$$\begin{aligned} A_{i+1} &= (A_i \ll 5) \square \phi(B_i, C_i, D_i) \square W_i \square K_i \\ B_{i+1} &= A_i \\ C_{i+1} &= B_i \ll 30 \\ D_{i+1} &= C_i \\ E_{i+1} &= D_i \end{aligned}$$

Une fois les 80 tours effectués, on fait pour chaque suite l'addition modulo 2^{32} du terme 0 et du terme 80 de la suite. Si le mot allongé vaut 512 bits, on concatène alors tous les résultats et le hash sera donc :

$$hash(m) = A_0 \square A_{80} \parallel B_0 \square B_{80} \parallel C_0 \square C_{80} \parallel D_0 \square D_{80} \parallel E_0 \square E_{80}$$

Mais si le mot sortant de Merkle Damgard est de longueur $512k$ avec $k \in \mathbb{N}, k > 1$ on recommence alors depuis l'étape 3 jusqu'à faire tous les blocs de taille 512 du mot allongé mais cette fois on initialise les suites avec les résultats du bloc précédent :

$$\begin{aligned} A_0 &= A_0 \square A_{80} \\ B_0 &= B_0 \square B_{80} \\ C_0 &= C_0 \square C_{80} \\ D_0 &= D_0 \square D_{80} \\ E_0 &= E_0 \square E_{80} \end{aligned}$$

Cependant comme expliqué précédemment dans notre cas le bloc allongé aura toujours une longueur de 512, ainsi on a besoin de définir la suite une unique fois.

5. Utilisation du hash Une fois le hash calculé nous sommes fasse à un binaire codé sur 160 bits mais comment l'utiliser ? Ce binaire est exagérément grand par rapport à nos tables... Le réflexe serait donc de prendre le modulo, cela réduirait drastiquement la taille du hash et permettrais de l'utiliser. Mais pour une fonction de hachage cryptographique c'est l'entièreté du hash qui est indispensable comme le montre le critère strict de l'avalanche (vu en 2.2.2 dans l'état de l'art). Il nous faut alors considérer chaque bits mais aussi réduire la valeur du hash. L'idée du poids est alors revenu, bien qu'alors certains bits influerons plus que d'autre, ces derniers auront tout de même une influence. Pour choisir le poids on considère la taille de la table dans laquelle le mot sera rentré et on trouve sa puissance de 2 inférieur (on la note $N_{Psc2Inf}$) et en notant $hash_{completBinaire}$ le hash initial en binaire de 160 bits et $hash_{reduitPoids}$ le hash avec des poids amménagés :

$$hash_{reduitPoids}(m) = \left(\sum_{i=1}^{160} hash_{completBinaire}(m)[i] \cdot 2^{\frac{i * N_{Psc2Inf}}{160}} \right) mod\ N$$

On s'assure toute de même que le hash soit inférieur à N avec un modulo.

Questionnement de démarrage Une fois l'algorithme compris, il a fallut décider de la manière dont nous allions travailler. En décimal ? L'ordinateur codant en binaire, cela ne changerais pas le résultat et réduirait peut être le complexité de l'algorithme... En hexadécimal ? En binaire ? Cela faciliterais la programmation de l'algorithme mais il faudrait pouvoir accéder a un bit via sa position, ce qui est faisable avec des opération mais coûteux...

Le compromis trouvé a été de codé en binaire mais sous la forme d'une chaîne de caractère ainsi la bibliothèque ¡string.h¿ sera disponible et nous aidera (en particulier strcat,strcpy et strlen) et nous pourrons accéder a chaque bits du mot allongé.

Programmation

1. Transformation du mot en binaire Pour transformer un mot en binaire nous avons décomposé en fonction intermédiaire : une transformant un entier en binaire, une transformant une lettre en binaire et une dernière transformant un mot en binaire.
   Pour la première fonction (classique) on crée une chaîne de caractère remplie de 0 puis divise l'entier passé en paramètre par 2 jusqu'à ce que ce dernier soit nul, enfin on remplace à chaque division le 0 dans la chaîne de caractère par le reste de la division par 2 : 0 ou 1. Enfin on retourne la chaîne de caractère pour avoir le binaire codé correctement.

2. Algorithme de Merkle Damgärd L'algorithme de Merkle Damgärd est rendu facile à programmer grâce au choix de mettre le binaire dans une chaîne de caractère. On commence par créer une chaîne de longueur de taille 512 et on la remplie tout d'abord avec le mot en binaire, ensuite avec un 1 puis avec $512 - 1 - 9 * n = 512 - 1 - strlen(motBinaire)$ zéros.

3. Découpage des blocs $(M_i)_{i\in[\![0;15]\!]}$ et création des blocs $(W_i)_{i\in[\![0;79]\!]}$ Pour découper le bloc allongé et créer les $(M_i)_{i\in[\![0;15]\!]}$ on déclare un char** contenant 16 mots de 32 bits et on vérifie tout d'abord que la longueur de la chaîne passée en entrée soit bien de 512. Pour les remplir ce char** on crée une double boucle for : la première, de paramètre k, de 0 à 15, permettant de changer de mot et la seconde de k*32 à (k+1)*32 permettant de remplir le mot k caractère par caractère par le mot allongé.
   Pour la création des $(W_i)_{i\in[\![0;79]\!]}$ on crée la fonction $char*\ \ xorStr(char*\ \ x, char*\ \ y)$ permettant de faire un xor bit à bit entre deux chaînes de caractère grâce à une boucle for et l'opérateur ^ permettant de faire le xor entre deux entiers. Il ne faut alors pas oublier que les 0 et les 1 dans les char* sont des caractères, que l'opérateur ^ prend des entiers et que les casts entre entier et char* se font automatiquement. Ainsi on prend soin de retirer 48 au code ASCII du caractère avant d'appliquer l'opérateur et on rajoute 48 avant d'ajouter le résultat à la chaîne de caractère de retour contenant le xor bit à bit de deux char*.
   Pour la rotation (imaginons une rotation de j bits) on commence par parcourir la chaîne d'entrée de $j+1$ jusqu'à sa fin et de stocker chacun des caractères dans une nouvelle chaîne. Ensuite on la parcourt de 0 jusqu'à j et on concatène chacun des caractères à la suite de la chaîne débuté dans le premier parcourt.

4. Suite sur 80 tours de SHA-1 Pour faire cette fonction on découpe par étape. La première est de définir les $(K_i)_{i\in[\![0;79]\!]}$.Pour ce faire, on crée une fonction prenant en paramètre un entier i (correspondant au tour de la suite) et selon sa valeur on retourne la constante associé au tour.
   La seconde étape a été de programmer la fonction ϕ. On décompose alors en les trois fonction $ITE,\ MAJ,\ ,XOR$ prenant chacune en entrée un entier (qui sera représentatif de la valeur du caractère '0' ou '1'). Pour chacune on utilisera l'opérateur ^, il faudra alors faire particulièrement attention au conversion entre la valeur en ASCII et la valeur réelle du bit. Une fois les fonctions $ITE,\ MAJ,\ ,XOR$ fonctionnelle on construit la fonction ϕ. Elle prend en argument 3 chaînes de caractère et un entier i, correspondant au numéro de tour des suites. On applique ensuite la fonction correspondante suivant l'entier i pour chaque bit (i.e. pour chaque caractère des chaînes d'entrée). Afin de ne pas avoir de mauvaises surprises on vérifie d'avoir les chaînes d'entrée de même longueur.

La dernière fonction secondaire à créé est la fonction permettant de faire des additions modulo 2^{32} entre 2 chaîne de caractère représentant des binaires. D'après [18]faire un modulo par 2^n sur un binaire consiste en fait à garder les n premiers bits. On se contente donc de faire une addition classique entre les deux binaires avec des conditionnelles pour prendre en compte l'éventuelle retenu et on prend pas en compte la dernière retenu.
Nous avons à présent tout les éléments pour construire une fonction SHA-1, prenant en entrée un mot et donnant en sortie une chaîne de caractère de taille 160 représentant un entier en binaire. On commence par transformer le mot en binaire, à l'allonger et à le découper en 16 blocs de 32bits via les fonctions précédemment créées, puis on initialise nos suites avec $A_0,\ B_0,\ C_0,\ D_0,$ et E_0. On prend soin de libérer toutes les allocations mémoires effectuées. Afin de pouvoir construire les suites on effectue à chaque tour i des copies de $A_i,\ B_i,\ C_i,\ D_i,$ et E_i. On applique simplement les définitions des suites sur 80 tours à l'aide d'une boucle for. Une fois obtenu les $A_{80},\ B_{80},\ C_{80},\ D_{80},$ et E_{80} on les additionne (modulo 2^{32}) respectivement à $A_0,\ B_0,\ C_0,\ D_0,$ et E_0. À l'aide d'un strcat on concatène ensuite chacun des résultats. Le hash est prêt !

5. Utilisation du hash Pour la première solution du modulo, on commence par calculer la puissance de deux supérieure à N c'est a dire qu'on cherche le m tel que $2^{m-1} < N < 2^m$. On construit pour cela une fonction prenant en entré un entier N et se construisant avec une boucle *while* ayant pour condition d'arrêt le dépassement de la valeur N. A chaque boucle on incrémente un compteur que l'on retourne une fois la condition d'arrêt dépassée. Une fois cette puissance de deux récupérée (notons la m) on récupère les m premiers bit du hash, on transforme ces derniers en entier et on applique un modulo N. Ainsi on ne se heurte pas au problème de codage des entiers se limitant à 2^{32}.
Pour programmer la seconde transformation du hash on calcule cette fois la puissance de 2 inférieur, avec la même fonction que précédemment et à l'aide d'une boucle for on récupère le bit de poids i, on le multiplie par son poids et on l'ajoute au résultat.

Tests Les tests ont été effectués et ont parfois révélé des erreurs, souvent bêtes, qui ont été corrigées immédiatement. Cependant une erreur persistait : $malloc\ consolidate\ :\ invalid\ chunk\ of\ size$

Premières difficulté rencontrées Lors des premiers tests et une fois les erreurs bêtes de programmation éliminées (par exemple oublier de changer la valeur ASCII en entier) une nouvelle erreur est apparue : $malloc\ consolidate\ :\ invalid\ chunk\ of\ size$. Afin de régler cette erreur nous nous sommes retrouvé sur Discord avec un partage d'écran afin de se conseillé et lorsque nous avons lancé le programme sur un mac, l'erreur disparaissait. Nous avons trouvé cela particulièrement étrange et en avons déduit que c'était probablement une erreur mémoire. Après quelques tests, il s'est avéré que l'erreur venait de *strcat*.

Solution trouvée. Pour rêgler les prblème de conversion des entiers avec l'ASCII, une simple soustraction suffit et pour les *strcat*, ces derniers furent changés par des concaténations manuelles à l'aide de boucles for.

Analyse du programme

1. Transformation du mot en binaire La complexité de l' étape de la transformation du mot est :

$$\begin{aligned} C(n) &= \sum_{i=1}^{n} \left(9 + \left(1 + +9 + \sum_{i=1}^{7} 4\right)\right) \\ &= 56n \end{aligned}$$

   La complexité de l'étape de la transformation du mot est donc en $O(n)$
   Plusieurs allocations mémoires ont été faites pour les chaînes de caractères et toutes (sauf celui contenant le mot en binaire) ont été libérées aux travers des fonctions.

2. Algorithme de Merkle DamgärdLa complexité de la transformation de Merkle Damgard est en O(n). Lors de l'algorithme de Merkle Damgard, on n'oublie pas de $free$ les $mallocs$.

3. Découpage des blocs $(M_i)_{i\in[\![0;15]\!]}$ et création des blocs $(W_i)_{i\in[\![0;79]\!]}$ La complexité de la fonction découpant les blocs est de $O(n)$ car on parcourt tout le message en entrée à l'aide de deux boucles for. La fonction en charge de créer les W_i est de : $O(n^2)$ car on appelle via une boucle for la fonction $xorStr$ elle même de complexité $O(n)$.

4. Suite sur 80 tours de SHA-1 et adaptation Lors de ce programme nous avons utilisé énormément de malloc, mais ces derniers ont ils été bien $free$? Nous avons vérifié cela avec Valgrind (cf figure 8) et cela est bien le cas.

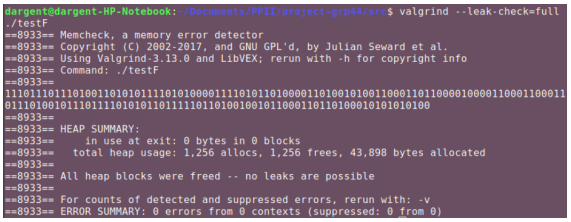


FIGURE 8 – Impact de SHA-1 sur la mémoire

Difficultés lié au remplissage de la table avec SHA-1 Lors du remplissage de la table avec les fonctions inspirées de SHA-1 cela a révélé des erreurs sur des fonctions qui n'avaient pas encore été testée. La première erreur (une segfault) se manifesta dans le $tableHachage\_get$: un élément pouvait avoir une sous table mais la valeur de cette sous table n'existait pas... En fait pour les cellules les pointeurs vers les sous tables (notés $nextWord$) n'était pas initialisés à $NULL$, la sous table pouvait alors exister sans raison d'être.

Nous nous sommes aperçus de la seconde erreur en regardant la table une fois remplie, certains mots avaient des symboles étranges, semblables à ceux rencontrés lors de fuites mémoires. Après vérification l'erreur venait de la fonction $maj2min$, nous avions oublié de terminer la conversion en ajoutant un '\0'.

5.2 Analyse des fonctions de hachage

Une fois ces fonctions programmées et fonctionnelles comment choisir le meilleure pour le remplissage de la table ? Compte tenu du nombre important de mot trouvé dans la table (15 085 mots distincts d'après la fonction $getTableNumberWords$ expliquée en sous section 4.2.2) il sera important de prendre en compte le temps de calcul comme paramètre du choix de la fonction mais aussi la dispersion des valeurs dans la table (c'est à dire s'assurer que la répartition soit uniforme et ne forme pas de 'grappe') et le facteur de compression de ces fonctions entreront en jeu.

5.2.1 Temps de calcul

En premier lieu nous allons examiner le temps de calcul des hashs selon les différentes fonctions de hachage. Pour essayer d'avoir des résultats fiables et généraliste nous n'utiliserons pas les mots de la base de données *movie_reviews.txt* mais nous en créeront en considérant qu'un mot est une suite de lettre d'une longueur donnée sans forcément de sens.

Hypothèses Avant de commencer les études sur le temps de calcul de ces fonctions de hachage, nous supposions de les fonctions utilisant SHA-1 seraient bien plus consommatrices de temps que les fonctions naïves ou récursives, cependant nous n'avions aucune idée de combien l'augmentation de la longueur des mots influerais sur ces fonctions.

Expérience On commence donc par choisir sur combien d'essai la moyenne du temps de hachage doit être faite. Au vu de la complexité des fonctions et de l'envie d'avoir des résultat moyens fiables il a été décidé de faire lune moyenne sur 1 000 essais. Pour encore plus de précision nous avons observé ces temps de calcul en fonction de la longueur du mot.

L'expérience consiste donc à générer, pour une longueur de mot donné et variant de 1 à 15, 1 000 mots et de mesurer leurs temps de hash pour chacune des fonctions et d'en faire la moyenne.

Programmation

1. Génération de mot : Pour générer un mot de longueur donnée (notons là n') on utilise un malloc de la taille du mot voulu +1 et avec une boucle for on ajoute à chaque tour un entier aléatoire compris entre 97 et 121 pour ne garder que des lettres en code ASCII, le cast se faisant automatiquement. On n'oubliera pas de $free$ le mot à la fin du calcul de ses hashs et de mettre $srand(time(NULL))$ au commencement.

2. Calculer un temps en C : Pour calculer le temps on décide de se servir de *clock*() dans le module $< time.h >$ qui compte le nombre de tour d'horloge du CPU et de diviser ce nombre de coup d'horloge par la constante $CLOCKS\_PER\_SEC$ afin d'avoir le résultat en seconde. Pour calculer le temps d'un hash il suffit alors de prendre le temps avant le calcul du hash (noté t_1), de calculer le hash, de reprendre le temps (noté t_2) puis de faire la différence des deux (qui est $t_2 - t_1$) afin d'avoir le nombre de coup d'horloge passé durant ce temps de calcul.

3. Expérience : Pour stocker la moyenne du temps de calcul pour chaque longueur de mot et pour chaque fonction on crée 6 tableaux de *double* (un pour chaque fonction de hachage), avec une allocation mémoire statique, chacun de taille correspondant à la taille maximal du mot (ici 15). La case i de ce tableau contiendra la moyenne du temps de calcul pour un mot

de taille i sur 1 000 essais. On initialise donc toutes les cases des tableaux à 0.0 à l'aide d'une boucle for.
Une fois les variables initialisées on commence l'expérience avec une boucle for pour le nombre d'essais (ici 1 000) et une seconde pour la taille du mot. Pour chaque taille de mot et pour chaque fonction on calcule le temps de fabrication du hash, on l'additionne au temps déjà présent dans la case correspondante.
Pour obtenir les résultats on affiche pour chaque tableau la somme contenu dans les case en la divisant par le nombre d'essai.

Difficultés Lors du premier lancement du programmes nous avons remarqué énormément de valeurs nulles dans les temps, voire des fonctions dont les temps étaient tous nuls (tel que la fonction naïve). Nous avons donc supposé que les valeurs étaient trop petites et subissaient un $underflow$.

Solutions Pour régler ce problème nous avions deux solutions : changer le mode de calcul du temps ou multiplier les résultats par 10^4. C'est cette deuxième solution plus simple et plus sûre que nous avons choisit ! Ainsi les résultats sont en seconde 10^{-4}.

Résultats Lors de la mise en forme de nous résultats nous obtenons les courbes de la figure 9
N.B. : Les trois courbes des adaptations de SHA-1 sont quasi confondues.

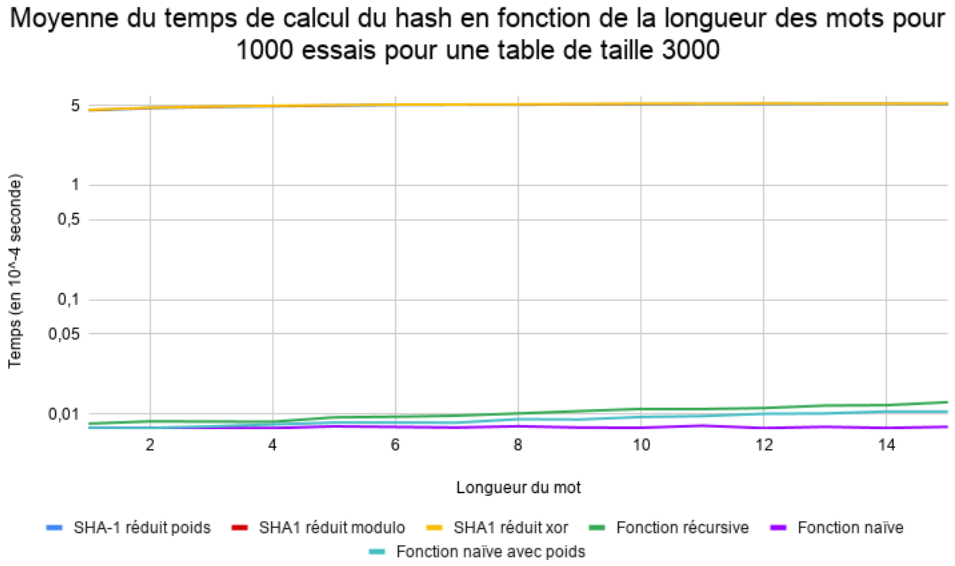


FIGURE 9 – Évolution du temps de calcul d'un hash pour les différentes fonctions de hachage

Interprétation des résultats Ce graphique confirme nos hypothèses plaçant les fonctions découlant de SHA-1 en tête de celles consommant le plus avec très peu de différence entre ces dernières (on peut cependant supposer que la différence existe mais qu'elle est minime et que l'échelle logarithmique ne permet pas de la voir). Cependant nous en apprenons plus sur l'augmentation du temps de calcul en fonction de la longueur d'un mot qui est moins grande que ce que nous aurions pu penser au premier abord.

La seconde surprise de ces analyse de temps est le rapport entre le temps de calcul des fonctions du groupe SHA-1 et des autres fonctions : ce dernier est de plus de 500 ! !

Conclusion intermédiaire Au vu du temps de calcul du groupe de fonction de SHA-1 par rapport au reste des fonctions ces dernières ne serait (pour le moment !) pas à conseiller. Reste cependant la question de la dispersions de valeurs dans la table...

5.2.2 Dispersion des hashs dans la table

En effet, une fonction aura beau se calculer vite si les valeurs de ses hashs ne se répartissent pas suffisamment uniformément dans la table cette dernière sera caduc ! Nous somme donc sur un critère éliminatoire. Là aussi pour avoir les résultats les plus fiables possible nous n'utiliseront pas les mots de *movie_reviews.txt* mais des mots aléatoire avec une longueur aléatoire.

Hypothèses Avant de commencer nos expériences nous nous attendions à avoir une très mauvaise répartition uniforme pur la fonction naïve (cf sous section 3.1.1) et une très bonne pour les fonctions dérivant de SHA-1 compte tenu de sa réputation et de son utilisation. Les grandes inconnues étaient donc la fonction naïve avec les poids et la fonction récursive. Rajouter des poids suffisait-il à assurer une répartition uniforme ? Est ce que cette récursivité jouait sur la répartition des hashs ?

Expérience Pour vérifier la bonne répartition uniforme des hashs nous voulons voir les hashs dans la table. Ainsi nous décidons de créer 6 tableaux de taille N (un pour chaque fonction de hachage) qui représenteront 6 tables de taille N. On génère ensuite un nombre N de mot aléatoire et on calcul leurs hashs. Pour chaque table on rajoute un à l'indice du hash. On visualise ensuite la table.

Pour que cette fonction ne soit pas trop longue à exécuter et que les résultats aient un minimum de sens nous prenons une table de taille 3 000.

Programmation Comme expliqué précédemment on crée 6 tableaux de *int* (un pour chaque fonction de hachage), avec une allocation mémoire statique, chacun de taille correspondant à la taille de la table (N et ici 3 000) que l'on initialise à 0. À l'aide d'une boucle for on génère N mots de taille aléatoire dont on calcule le hash pour les différentes fonctions de hachages (on stock ce hash dans une variable temporaire pour plus de visibilité). On se contente ensuite de rajouter 1 à l'indice du hash pour chaque table correspondante (on rajoute 1 dans le tableau de la fonction numéro x à l'indice du hash du mot de la fonction numéro x).

On n'oublie pas de $free$ le mot à la fin de chaque tour de boucle.

Résultats On affiche ensuite les résultats dans un graphique (cf figure 10, 11 et 12)

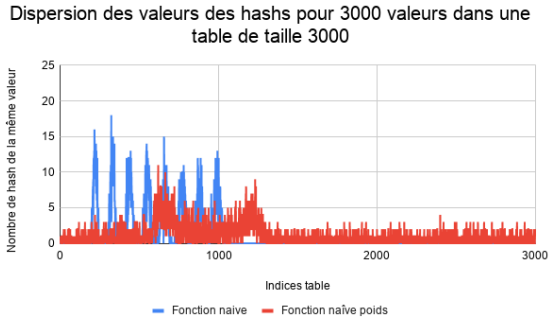


FIGURE 10 – Visualisation de la dispersion des valeurs des hashs pour les fonctions utilisant la fonction naïve

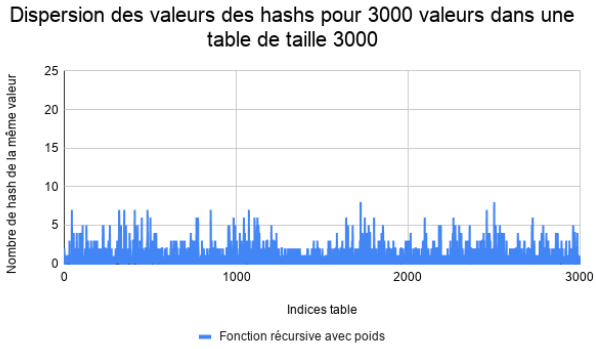


FIGURE 11 – Visualisation de la dispersion des valeurs des hashs pour la fonction récursive

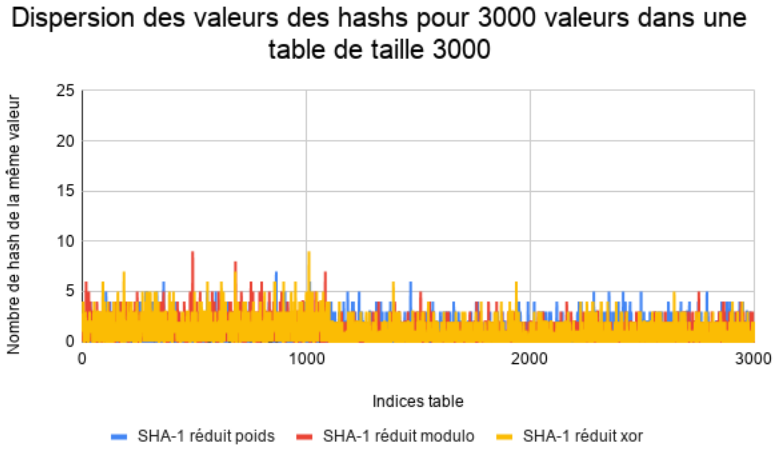


FIGURE 12 – Visualisation de la dispersion des valeurs des hashs pour les fonctions utilisant SHA-1

Interprétation des résultats

1. Figure 10, les fonctions naïves : Comme attendu pour la fonction naïve simple, les valeurs ne sont absolument pas réparties uniformément et forme de gros blocs entre 250 et 1 000. Pour la fonction naïve avec poids les hashs sont déjà mieux répartis à l'exception de 2 pics en 700 et 1 200 que l'on peut supposer être dût à l'attribution des poids suivit du modulo : même si les poids limitent les paquets de hash, les lettres n'ont pas toutes le même poids donc pas toutes la même influence (donc l'ordre des lettres compte mais n'est pas exhaustif) et les pondérer permettent d'étaler les hash mais pas d'écarter toutes possibilité de regroupement.

2. Figure 11, la fonction récursive : À première vu la fonction récursive répartit mieux les hashs que la fonction naïve avec les poids, cependant en y regardant de plus près on observe des

cycles plus ou moins réguliers .

3. Figure 12, les adaptations de SHA-1 : Le graphique montrant la dispersion des hashs des fonctions adaptées de SHA-1 est le meilleur : les valeurs sont toutes réparties uniformément, on ne remarque ni cycle, ni grappe de hash. De plus cela est vrai pour toutes les fonctions dérivant de SHA-1, l'une n'est pas meilleure que l'autre, on le remarque particulièrement car les graphiques sont presque superposés.

Conclusion intermédiaire Ainsi la fonction naïve est éliminée du choix de la fonction finale, cette dernière ne répartie pas uniformément les hashs. Pour les autres fonctions aucune n'est disqualifié mais il y a une nette préférence pour les fonctions adaptées de SHA-1.

5.2.3 Facteur de compression

Le derniers critère à examiner pour les fonctions de hachages est le facteur de compression de ces dernières. Autrement dit, pour une table de taille N avec N mots combien de case quel est le rapport de case occupé sur le nombre de case disponible ?

Hypothèses À la vue de la dispersion des valeurs pour les différentes fonctions de hachage dans la sous section 5.2.2 on peut facilement déduire quelles sont les fonctions de hachage avec un bon facteur de compression. Mais comment ce dernier évolue en fonction de la taille de la table ?

Expérience Pour calculer le facteur de compression on se sert de la fonction précédente : une fois les tableaux contenant la valeurs des hash obtenu il suffit de sommet le nombre de case n'ayant pas pour valeur 0. En le divisant par la taille de la table, on obtient le facteur de compression. Il reste ensuite à effectuer cela pour différentes taille de table.

Toujours avec cette volonté d'avoir des résultats sur une moyenne mais avec la contrainte du temps d'exécution du programme nous avons décidé de faire une moyenne sur 5 : ainsi un cas aléatoire particulier sera quelque peu masqué et le temps de l'algorithme quoi que déjà long sera acceptable. De plus la table de la taille variera de 1200 à 16200 avec un pas de 500, ainsi on commence avec une table de taille relativement faible jusqu'à arriver au delà de 15085, la taille de la table souhaité pour rentrer les mots de *movie_reviews.txt*.

Programmation On répartit le programme en deux fonctions : une calculant le facteur de compression et l'autre appelant la première avec différentes tailles de table et cela pour le nombre d'essais souhaité.

Pour la première fonction on reprend exactement celle de la sous section 5.2.2 et on crée un tableau de flottant qui contiendra le facteur de compression pour chaque fonction. Une fois les tableaux de dispersions créé on les parcours et on rajoute 1 à un compteur (une fonction a un compteur) à chaque fois que l'indice est non nul. On rentre ensuite ces compteurs divisés par la taille de la table au tableau de retour.

Pour la seconde fonction on initialise là aussi 6 tableaux qui contiendront la moyenne du facteur de compression pour une taille de table donnée. Le programme est alors composé de deux boucles for : la première sur la taille de la table et la seconde pour le nombre d'essais. À chaque

essais on appelle la première fonction et on associe le facteur de compression à la case du tableau correspondant. On termine en divisant chaque case des tableaux par le nombre d'essai effectués.

Résultats Une fois les résultats mis sous forme graphique on obtient la figure 13. On a mis les

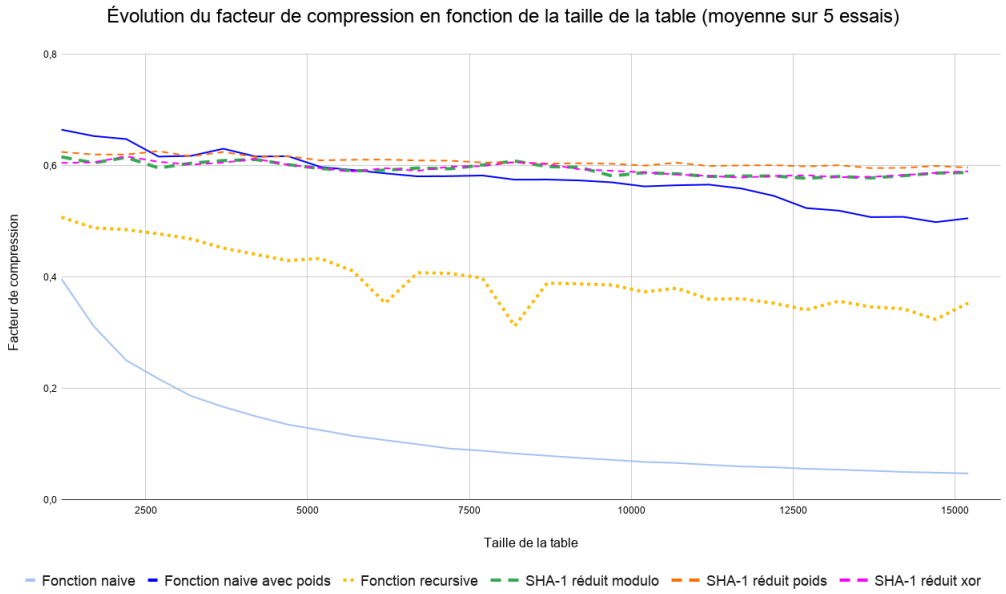


FIGURE 13 – Évaluation du facteur de compression des différentes fonctions de hachage en fonction de la taille de la table.

fonctions dérivant de SHA1 avec des tirets, la fonction récursive en pointillé et les fonctions dérivant de la fonction naïve avec des traits pleins bleus.

Interprétations Bien que la fonction naïve (courbe bleu clair pleine) soit déjà éliminée du choix pour la fonction de hachage finale il est intéressant de voir à quelle vitesse le facteur de compression diminue, ce dernier semble même tendre vers 0 (ce qui est logique car les hashs sont bornés à 2016 − cf sous section 3.1.1 − mais la taille de la table augmente) et sa décroissance est bien plus rapide que prévu à un point que pour une table de taille 15 000 le facteur de compression est de 0.04 !

Cependant on remarque que la fonction naïve avec poids (courbe bleu foncé pleine) subit une décroissance bien plus faible et a un facteur de compression de 0.56 pour une table de talle 15 000. Bien que le facteur de compression ne soit pas optimal, cette fonction n'est donc pas éliminée.

La seconde fonction éliminée est la fonction récursive (courbe orange pointillée), bien que son facteur de compression ait une décroissance plus faible que la fonction naïve ce dernier reste bien inférieur à 0.5 ce qui n'est pas suffisant par rapport au nombre de sous table que nous devrions créer si cette fonction venait à être prise.

Pour les fonction dérivant de SHA-1 on remarque en premier lieu leurs stabilités ! Quel que soit la table de la taille (et pour toutes les fonctions dérivant de SHA-1) leurs facteur de compression

tourne autour de 0.6. On remarque que SHA-1 réduit poids à un facteur de compression légèrement au dessus.

Conclusion partielle Comme supposé dans les hypothèses la fonction récursive est celle ayant le moins bon facteur de compression, mais la surprise de cette expérience a été de voir la décroissance tout de même importante de la fonction récursive qui ne sera donc pas choisit comme fonction finale. Nous avons cependant eu une déception sur le facteur de compression des fonctions dérivant de SHA-1, nous le pensions plus haut.

5.2.4 Choix de la fonction de hachage

Avec toutes ces expériences, toutes indépendantes de la création de table ou des données de $movie\_reviews.txt$ nous pouvons enfin choisir la meilleure des 6 fonctions ! La fonction naïve et la fonction récursive s'étant faite éliminées il nous reste donc à choisir entre la fonction naïve avec poids et les fonctions dérivant de SHA-1.

Pour les fonctions dérivant de SHA-1, compte tenu de leurs temps de calcul et de leurs dispersions quasi identiques on préférera la fonction SHA-1 réduit poids qui a un facteur de compression légèrement meilleure que la fonction SHA-1 réduit modulo ou bien la fonction SHA-1 réduit xor.

Les fonctions choisies sont : la fonction naïve avec poids et SHA-1 réduit poids

5.3 Analyse de la table avec cette fonction de hachage

Nous avons à présent deux fonctions de hachage qui sont susceptibles de remplir correctement (ou du moins pas trop mal) la table. Mais qu'en est-il réellement ?

5.3.1 Temps de remplissage de la table

Présentation Le premier facteur qui est intéressant d'explorer est le temps de remplissage de la table, on sait bien que calculer un hash de SHA-1 réduit poids est bien plus coûteux qu'un hash de la fonction naïve avec poids. Mais à quel point cela se répercute dans le temps de remplissage de la table ? Pour savoir cela on utilisera la même méthode de calcul du temps que dans les expériences précédentes : $clock()$ de $< time.h >$

Programmation Pour programmer cela on se contente de modifier la fonction $filledTable$, en charge du remplissage de la table en initialisant deux variables t_1 et t_2 de type $clock\_t$. Dès le début de $filledTable$ on associe à t_1 la valeurs de $clock()$ et seulement à la fin on associe à t_2 la valeur $clock()$. On fait ensuite la différence, on divise par $CLOCKS\_PER\_SEC$ et on obtient le temps de remplissage de la table que l'on affiche.

Analyse La modification de cette fonction n'a fait qu'ajouter 4 à sa complexité (2 opérations sur l'initialisation de t_1 et t_2, une sur la différence et une dernière sur la division).

5.3.2 Facteur de compression de la table

La dernière fois que nous avions évalué le facteur de compression d'une table nous l'avions fait en nous affranchissant de la structure de donnée utilisée : les tables ! Cette fois ci nous créons un programme propre à la structure des tables.

Présentation Pour trouver le facteur de compression on va tout simplement parcourir la table et compter le nombre de case non vide. On divise ensuite cette donnée par la taille de la table.

Programmation Pour programmer ce parcours de la table et le comptage des cases non vide on crée deux compteurs : un pour s'assurer qu'on ne dépasse pas la taille de la table, ce sera notre condition d'arrêt dans la boucle *while*, et un second (de type $float$) pour compter le nombre de case non vide. Un fois ces deux compteurs initialisés à 0 on parcours la table en regardant à chaque fois si le nombre de note est non nul, si tel est le cas, on incrémente le second compteur. On passe ensuite à la cellule suivante grâce à l'attribut *suiv* de la structure des cellules puis on incrémente le premier compteur. On retourne le second compteur divisé par la taille de la table.

Analyse Cette fonction a une complexité en $O(N)$ et plus précisément, avec N la taille de la table :

$$\begin{aligned} C(N) &= 1 + 1 + 1 + \sum_{i=1}^{N} 3 \\ &= 3(N+1) \end{aligned}$$

5.3.3 Temps moyen de recherche d'un mot

Le dernier critère que nous étudierons sur la table est le temps moyen de recherche d'un mot dans une table de taille 15 085.

Présentation Pour trouver le temps moyen de recherche d'un mot on va chercher un à un les mots présents dans la table et mesurer le temps de recherche (on a déjà une fonction $tableHachage\_getHash$ permettant la recherche d'un mot sans calculer son hash). On somme toutes ces valeurs puis on divise le résultat par le nombre de mots recherchés.

Programmation Pour créer cette fonction nous avons besoin d'une table remplie des mots du fichier *movie_reviews.txt*, du fichier lui même et du numéro de fonction utilisé pour remplir la table. Toutes ces données seront passées en paramètre. On procède ensuite de la même manière que dans la fonction $filledTable$ 3.2.1 pour lire les mots du fichier ligne par ligne. Pour chaque mot on calcule le temps que prend $tableHachage\_getHash$ à retourner un élément avec les mêmes méthode que précédemment (2 variables de type $clock\_t$,une avant le $tableHachage\_getHash$ et l'autre après, la différence des deux divisée par $CLOCKS\_PER\_SEC$ donne le temps pris pour la recherche d'un élément) et on rajoute cette valeur à une variable de type $float$ (noté res) qui contiendra la somme de tous les temps à la fin du parcours des mots du fichier. On retourne la valeur de res divisé par le nombre de mot (non distinct du fichier)

Analyse En notant n le nombre de mot du fichier on a une complexité en :

$$C(l) = \sum_{i=1}^{l} (14 + C_{hachage} + C_{get})$$

On notera que dans *movie_reviews.txt* on a 147 197 mots non distincts ce qui est moitié moins que prévu dans la section 3.1.1

5.3.4 Résultats et résumé

Une fois toutes ces fonctions programmées et exécutées on peut finalement condenser toutes les informations obtenues pour la fonction naïve avec poids et la fonction SHA-1 réduit avec poids :

	Fonction naïve avec poids (1)	SHA-1 réduit poids (2)	Rapport (2)/(1)
Temps de remplissage de la table	12.573888 secondes	1 min 42	8.0763
Temps moyen de recherche d'un mot	0.000031 seconde	0.000070 seconde	2.258
Temps minimum de recherche d'un mot	0.0000 seconde	0.0000 seconde	−−
Temps maximun de recherche d'un mot	0.000819 seconde	0.001519 seconde	3.4289
Facteur de compression	0.508319	0.633411	1.24608

La première chose que l'on peut observer est le facteur de compression de la table qui correspond à l'étude faite précédemment (cf figure 13), ainsi on remarque que SHA-1 réduit poids a un facteur de compression bien meilleur que la fonction naïve réduit poids.

Le second résultat auquel nous nous attendions est le temps de remplissage de la table : ce dernier est multiplié par 8 lorsque l'on utilise SHA-1 réduit poids ! On avait vu dans la sous section 5.2.1 que le temps de calcul de SHA-1 réduit poids pouvais être 500 fois supérieur au temps de calcul de la fonction naïve avec poids, le résultat est donc plutôt bon et cohérent. On peut expliquer en partie le fait que le temps de remplissage soit 8 fois (et non 500 fois) supérieur par le facteur de compression : ce dernier étant plus élevé pour SHA-1 réduit poids il est moins nécessaire de créer des sous tables, ce qui est un gain de temps.

La surprise de cette étude vient du temps moyen de recherche d'un mot, bien qu'il y ait moins de sous table pour SHA-1 réduit poids que pour la fonction naïve le temps moyen de recherche d'un mot est 2.2 fois supérieur pour SHA-1 réduit poids que pour la fonction naïve... On pourrait expliquer cela en supposant que bien qu'il y ait moins de sous table ces dernières sont peut être

plus longues.

On privilégiera tout de même le facteur de compression au temps moyen de recherche car dans les deux cas, le temps de recherche d'un mot reste faible. De plus SHA-1 réduit poids à une répartition plus uniforme de ses hashs.

On privilégiera la fonction SHA-1 réduit poids

6 Applications à d'autres jeux de données

6.1 Introduction

Initialement, le jeu de données que nous avions était en anglais. Néanmoins, sur le site de Rotten Tomatoes, il existe plusieurs langues différentes comme le Français, l'Espagnol, l'Allemand,etc. Le but de cette partie de programme est d'avoir un calcul du poids de chaque mot aussi juste possible : cela concerne la partie de la mesure de fiabilité du modèle de prédiction. Beaucoup de langues racines du latin utilisent des déterminants, des pronoms personnels (etc ...) ne prenant pas partie dans le sentiment du commentaire de l'utilisateur.

6.2 Méthode

Pour éliminer ces mots indésirables pour la table table de hachage, il suffit de créer une "blacklist" des mots dont on ne veut pas être dans la table de hachage. Néanmoins, cela réduit l'automatisme du logiciel : on est obligé de mettre manuellement l'ensemble des mots dont on ne veut pas.

Après avoir rempli le fichier texte des éléments indésirables , nous pouvons utiliser la fonction split utilisé pour détecter chaque mot des commentaire de la base de donnée initial. L'ensemble des mots indésirable seront mis dans une liste. Ainsi, à chaque passage d'un mot voulant être mis dans la table de hachage, on "compare" ce mot à la liste des mots indésirable.

6.3 Améliorations

Un problème de taille s'installe : la langue française contient des milliers de mots supposé indésirable. Cela pourrait affecter le temps d'exécution du programme en ajoutant plusieurs millions de calculs supplémentaires. Afin d'éviter ce désagrément , il est bien de "trier" la liste des indésirables.

Pour trier, on va tout d'abord calcul leur valeur (hachcode) selon une fonction choisi. À partir des valeurs, une liste de couples mot-valeur est créée. À partir, de ces valeurs, il suffit de les trier à partir d'un tri Rapide.L'ensemble des valeurs peuvent ainsi être comparé dans un premier temps, par leurs valeurs (hachcode) puis s'il y a un ou plusieurs mots ayant la même valeur, comparé dans un deuxième temps, les mots eux-mêmes. Cette comparaison se fera grâce à une dichotomie. L'utilisation de la dichotomie permettra dans le cas d'une longue liste (avec plus de 1000 mots), de réduire massivement le temps de calcul. En effet, on peut comparé la méthode de recherche linéaire (on compare un à un les élements) et la méthode de dichotomie grace à leur complexité : dans le premier cas, on a une complexité de $\boxed{O(n)}$ alors que pour la dichotomie, on a une complexité de $\boxed{O(\ln(n))}$ avec n la taille de la liste. Pour n=1000, on a ln(n)=6.9. Ainsi, on a un rapport de nombre de calcul de plus de 150 par mot ce qui diminue considérablement le temps d'éxecution du programme.

Si l'on voulait avoir une base de données unifier entre les langues, il aurait fallu coder un dictionnaire de traduction. Néanmoins, il nous semble infaisable d'effectuer une telle tâche dans une période de temps aussi restreint.

7 Gestion de projet

7.1 L'équipe

Répartition des rôles Lors de la première réunion, nous avons fait connaissance avec les différents membres du groupe et nous avons partager nos expériences sur le premier projet. Nous avons également discuter des points forts et des points faibles de chacun ce qui a amené à la conclusion que nous étions un groupe complémentaire. Élisa Dargent a été nommée chef de projet du groupe et nous avons décidé de ne pas nommer de secrétaire mais de rédiger les comptes rendus chacun notre tour.

Points forts et points faibles

Matrice SWOT	Positif	Négatif
Interne	Groupe complémentaire Expérience d'un projet Connaissances d'Élisa sur les fonctions de hachage	Pas de recul sur le langage
Externe	Entraide avec les différents groupes Flexibilité	Contraintes dues à la crise sanitaire Confinement

Solutions et optimisations Lorsqu'un membre du groupe rencontrait des difficultés en $\#C$, nous nous réunissions dans le salon vocal sur Discord afin de résoudre les problèmes tous ensemble via le partage d'écran. Pour pallier aux contraintes dues à la situation inédite dans laquelle nous sommes, nous avons créé un groupe Messenger et un groupe Discord (cf. la sous-section 7.2.2 pour plus de détail) afin de faciliter la communication dans le groupe. Pour ce qui est du confinement, nous n'avons rien pu faire si ce n'est utiliser les moyens de communications que nous avons mis en place.

7.2 L'environnement de travail

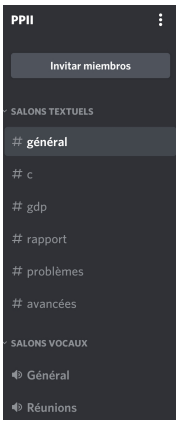
Ce projet s'illustrera par le contexte exceptionnel dans lequel il s'est effectué. C'est en pleine crise sanitaire que ce dernier a commencé et cela à profondément modifié notre manière de travailler et de communiquer.

7.2.1 Confinement et rythme de travail

Avec le confinement les habitudes de projet prisent lors du premier semestre ont du être adaptées. Dans nos anciens projets, les stand up meeting avait un rôle clef : ils nous permettaient de faire part de nos avancées, de nos idées, de nos problématiques et d'avoir un avis ou une aide extérieur en peu de temps. Ainsi, chaque membre avait une vision globale du projet à tout instant. La suppression des stand up meeting fut une des conséquences du confinement et cela a joué sur la communication qu'il a fallut rétablir (il est expliqué comment dans la section 7.2.2). Cependant, il y eu des aspects positifs suite à ce confinement. En effet, nous étions forcément chez nous jour et nuit et cela a permis d'avoir en permanence (ou presque) la possibilité d'obtenir de l'aide en cas de doute ou de difficulté là ou il fallait avant prévoir un rendez vous à heure fixe.

7.2.2 La communication

La communication fut la première chose qu'il a fallut adapter. C'est en effet un élément crucial lors de la gestion d'un projet et une mauvaise communication peut ruiner un projet.
Pour s'adapter à cette situation notre premier réflexe a été la création d'un Discord avec différents canaux. Certains textuels propres à la gestion de projet et au rapport permettant l'organisation de réunions ou des discussions sur l'organisation du rapport. Pour effectuer les réunions, un canal vocal *#Réunions* a été créé.
D'autres canaux sont plus centrés sur la programmation comme le canal *#C* ou *#Problèmes* permettant respectivement d'échanger sur des problèmes de compréhension du C et de faire part de divers problèmes autant sur l'algorithmique, la compilation ou sur des résultats non cohérents. Afin de pouvoir être plus efficace sur la résolution de ces difficultés, le canal visuel *#Général* permettra un partage d'écran.
Le dernier canal est le canal textuel *#Avancées* permettant de retrouver les différents points clés du projet afin de suivre son avancement et de pouvoir évaluer sa progression.



7.2.3 Le matériel

Là ou l'école nous offrait une norme avec des ordinateurs sous Linux et disponible 6 jours sur 7 jusqu'à 22h, cette dernière est dorénavant plus dure à suivre malgré la mise à disposition d'ordinateurs par l'école. En effet, grâce à cette démarche, chaque membre du groupe dispose d'un ordinateur fonctionnel mais sous des systèmes différents.

Plusieurs difficultés ont donc découlé de cette diversité d'OS. En effet, Léa TOPRAK utilise un MAC et il a fallut adapter pour l'utilisation de Valgrind à la compilation. Le second problème posé par cette multitude d'OS a été lors de la gestion des segFault par exemple avec l'erreur expliqué en 5.1.4 (*consolidate_malloc : invalid chunk of size*) qui n'apparaissait que sous Linux et Windows, cela n'a pas facilité son débeugage.

7.3 Lancement du projet

Formalisation de réunions Étant donné la situation sanitaire actuelle, les réunions seront extrêmement importantes et devront récapituler l'ensemble des tâches faites durant le laps de temps écoulé. Ces dernières se feront les samedi ou dès que la nécessité s'en fait resentir pour un membre du groupe.

Organisation Pour le début de ce projet, nous avons décidé de créer une branche test afin de pouvoir rendre une branche master propre avec nos programmes finaux.

Nous nous sommes ensuite répartit les rôles selon nos affinités avec le sujet ainsi Léa commencera à travailler sur l'évaluation d'un commentaire, Élisa s'occupera de remplir la table et Niels commencera à travailler sur la gestion des collisions (cf 14 et 17. Nous nous sommes également tous penché sur l'état de l'art.

Premières difficultés À cause de la crise sanitaire, les rencontres entre les membres du groupe de projet a été impossible. À cause de cette situation exceptionnelle, la communication fut très

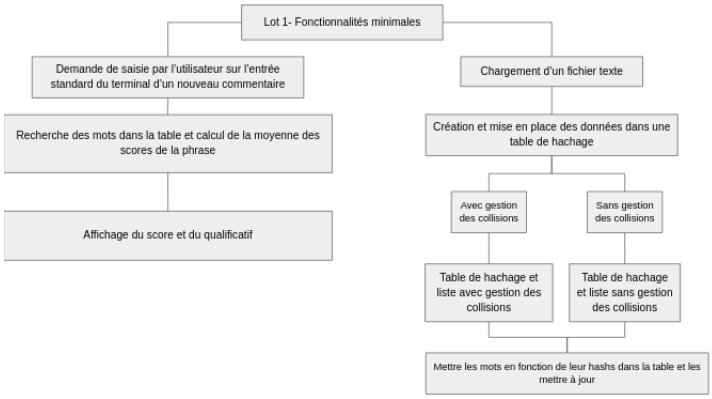


FIGURE 14 – Première partie du WBS

dur à mettre en place, surtout pour la répartition des tâches. Après quelques temps d'adaptation, l'organisation des réunions et la répartition du travail fut plus facile.

7.4 Poursuite du projet

Lorsque la deuxième phase du projet s'est amorcé nous étions déjà en capacité de remplir une table avec une fonction de hachage naïf mais la gestion des collisions n'étaient pas encore optimale pour le remplissage de la table.

Organisation Les rôles se sont donc répartit avec Niels qui s'occupait de modifier les fonctions caducs de *HashProg.c* (fichier permettant le remplissage de la table) pour les adapter à la gestion des collision. Cependant, Léa devait pouvoir se servir des fonction de *HashProg.c* car elle commencait l'analyse statistique de la table. Pour pallier à cela, Niels modifiera les fonction dans un nouveau fichier appelé *HashProg_collision.c*. En parallèle, Élisa s'occupera de créer de nouvelle fonction de hachage. Dans les WBS, (cf figures 15 et 16) cela est encadré en rouge.

7.5 Fin du projet

La fin de projet requiert toujours une organisation particulière et c'est généralement à ce moment que la gestion de projet qui a été faite durant le projet trouve sa résonance.

Contexte La fin de ce projet marque la fin de l'année, les partiels mais aussi la fin du confinement et le retour progressif à la normale cependant l'école reste fermé et le projet se finira aussi à distance, rajoutant une difficulté.

Organisation Pour cette fin de projet l'organisation a été la suivante : tous les membres du groupe rédigent le projet en plus de certaines tâche spécifiques. Élisa s'occupera de finir son analyse sur la table, Léa finira les dernière études statistique et les tests et Niels créera une petite interface avec l'utilisateur et fera ses tests. La dernière étape sera celle du merge : nous avions décidé en début de projet de programmer sur la branche Test, afin de ne pas polluer la branche master de

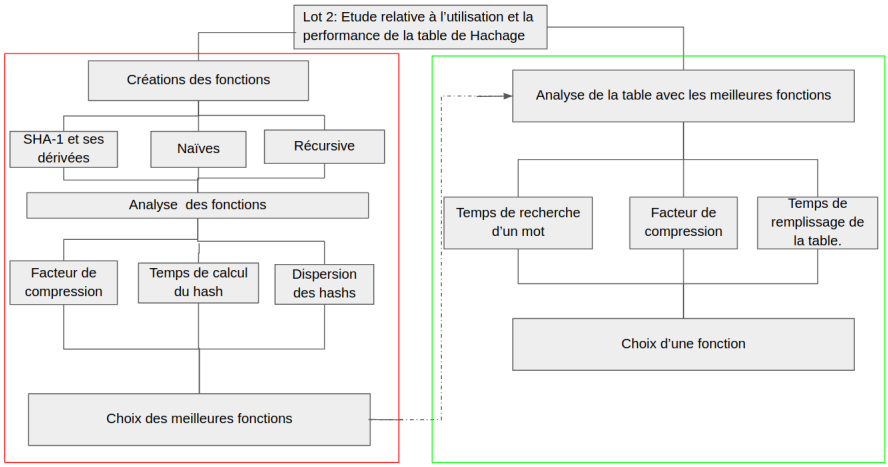


FIGURE 15 – Deuxième partie du WBS

programmes faux. Il est donc mainteant temps de la regrouper avec la branche master que les programmes sont fonctionnels. Dans les WBS, (cf figures 15 et 16) cela est encadré en vert.

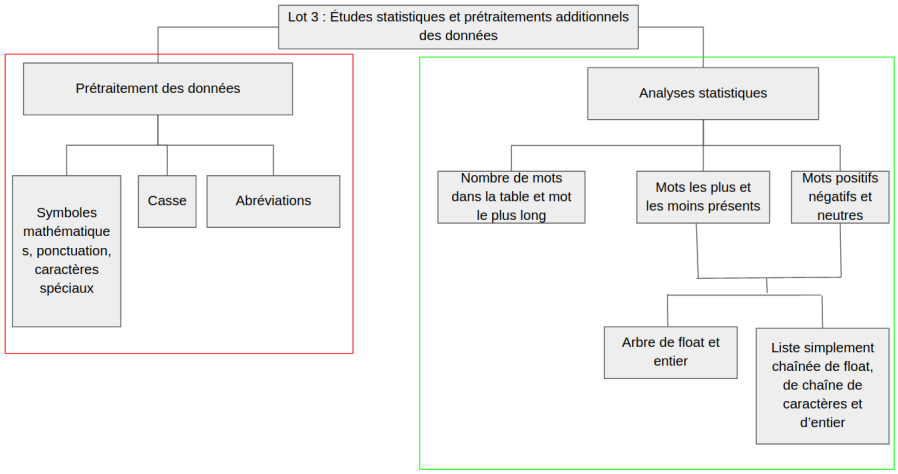


FIGURE 16 – Troisième partie du WBS

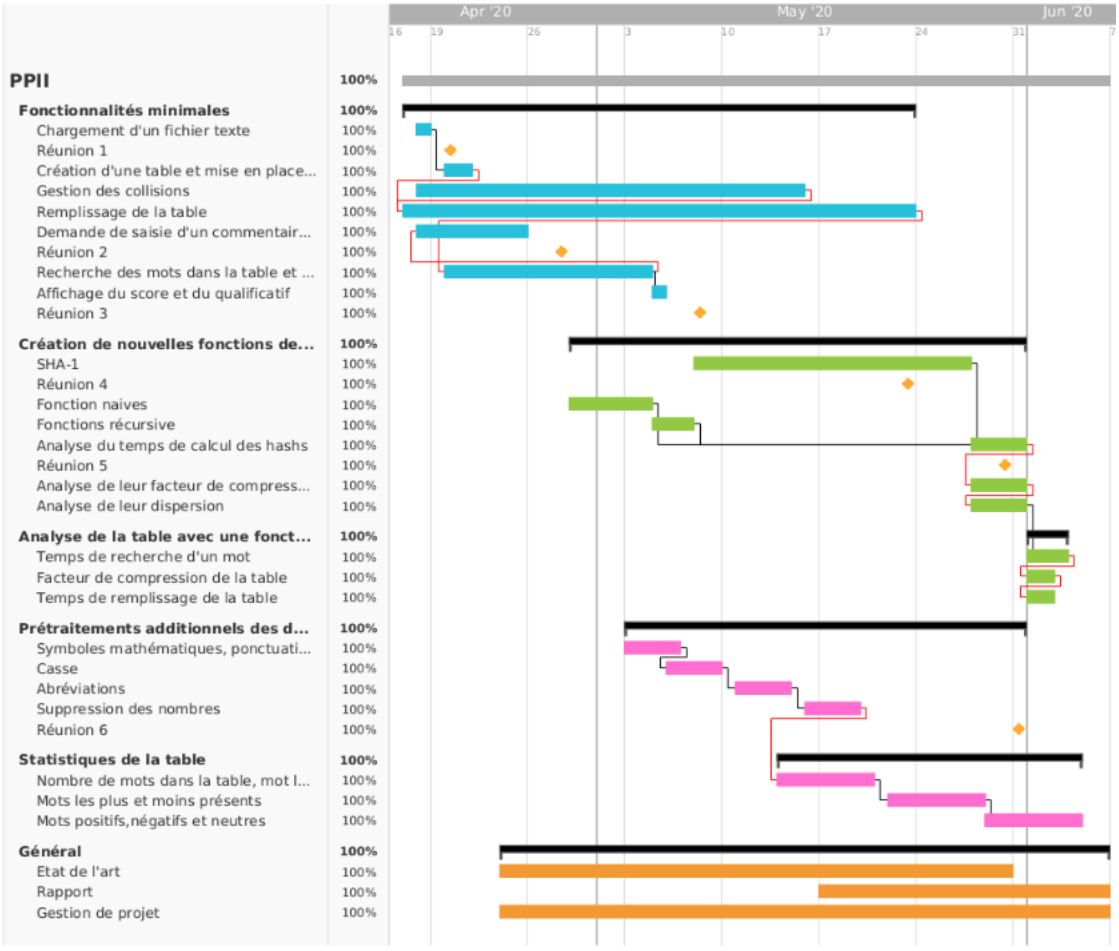


FIGURE 17 – Gantt final du projet

8 Conclusions

8.1 Conclusion générale

Lors de ce projet nous nous sommes aperçus de la difficulté de créer un programme qui pouvait interpréter des sentiments, notamment à cause de la syntaxe des langues, de la véracité du modèle de prédiction et de la complexité du programme. Par exemple le commentaire "Loved every minute of it" a une note d'environs 2.016 sur 4, ce qui correspond à un sentiment positif mais qui est un résultat plutôt timide par rapport au score que pourrait attribuer un être humain à ce commentaire. Ainsi, le programme pourrait largement être amélioré, et on pourrait penser à le faire apprendre des commentaires qu'il lit.
Mais un programme peut-il réellement interpréter et reproduire les sentiments humains ? Pour Philippe Guillaud, cofondateur avec André Manoukian du générateur musical Muzeek, non, la

machine n'en est pas capable (cf article [19])" *Certains de nos concurrents proposent des musiques réalisées à 100% par des ordinateurs, mais on trouve qu'il leur manque un élément-clé que la machine n'est pas capable de créer aujourd'hui : l'émotion.*". Un programme ne pourrait qu'estimer les sentiments mais pas les interpréter et encore moins les reproduire. Cela pourrait être un défis pour le futur de l'IA mais soulèverais énormément de questions éthiques : les sentiments ne sont ils pas le propres des êtres vivants ?

8.2 Conclusions personnelles

Conclusion d'Élisa Dargent Bien que le sujet fut particulièrement intéressant et propre à la découverte de nombre de domaines différents (fonctions de hachage, statistiques, algorithme qui apprend) je suis déçue du résultat final de ce projet. Les fonctions de hachage sont un sujet qui me passionnent et j'ai particulièrement apprécié de programmer SHA-1. Je reste cependant frustrée de l'étude réduite que j'ai pu en faire et de pas avoir pu mener mes expériences à terme à cause de manque de tests systématiques menant à des segfaults déclenchés par des expériences aléatoires. Le temps de les résoudre (ou de communiquer pour les faire régler par les constructeurs de la fonctions) a été une perte que je regrette.

Malgré ces déceptions je retire de ce projet beaucoup d'enseignement tout d'abord en de gestion de projet et de son importance toute particulière en situation de crise mais aussi en programmation avec la prise d'expérience de ce projet et avec la certitude de la nécessité absolue des tests systématiques et rapides lors d'un projet.

Conclusion de Niels Tilch Le sujet fut très intéressant mais très compliqué en vue d'un projet de groupe à cause de la crise sanitaire. La communication a été très difficile à mettre en place au début (car nous étions dans une situation exceptionnelle) mais elle s'est stabilisé au fur et à mesure du projet. D'un point de vue raisonnements, j'ai trouvé le sujet stimulant car il nous ouvrait un champs de possibilités assez extraordinaire. J'ai pu beaucoup m'améliorer sur le langage C grâce aux raisonnements que nous avions créé. Bien que la gestion de la mémoire soit cruciale, elle a été néanmoins difficile.

De par les déceptions que j'ai eu eu tout au long du projet, il m'a beaucoup apporté, surtout en gestion de projet lorsque nous ne pouvions pas nous voir physiquement.Ceci nous permettra d'avoir une expérience pour le télé-travail ce qui me semble important surtout en ingénierie informatique.

Conclusion de Léa Toprak Le projet fût une expérience inédite et très enrichissante du fait de la crise sanitaire. En effet, nous avons dû nous adapter à la situation en travaillant sur le projet à distance des autres membres du groupe ce qui ne fût pas simple au départ. Il nous a fallu un certain temps d'adaptation pour mettre en place une bonne communication dans le groupe. Malgré la situation exceptionnelle dans laquelle nous étions, j'ai beaucoup aimé travaillé sur ce projet car l'analyse des sentiments est une technologie qui utilisée dans notre quotidien si ce n'est posté un simple message sur les réseaux sociaux. Le projet nous a également permis de mettre en application les nouvelles connaissances que nous avons acquises ce semestre notamment la programmation en C, l'utilisation des structures de données et la réalisation des tests avec la bibliothèque Critérion.

9 Annexe

9.1 Attestation de non plagiat.

9.1.1 Élisa DARGENT

Je soussignée Élisa DARGENT,
Élève-ingénieure régulièrement inscrite en 1ère année à TELECOM Nancy
Numéro de carte de l'étudiante : 31916989
Année universitaire : 2019–2020
Auteur du rapport et code informatique associé au projet intitulé :

Analyse des sentiments

Par la présente, je déclare m'être informée sur les différentes formes de plagiat existantes et sur les techniques et normes de citation et référence. Je déclare en outre que le travail rendu est un travail original, issu de ma réflexion personnelle, et qu'il a été rédigé entièrement par mes soins.
J'affirme n'avoir ni contrefait, ni falsifié, ni copié tout ou partie de l'œuvre d'autrui, en particulier texte ou code informatique, dans le but de me l'accaparer.
Je certifie donc que toutes formulations, idées, recherches, raisonnements, analyses, programmes, schémas ou autre créations, figurant dans le document et empruntés à un tiers, sont clairement signalés comme tels, selon les usages en vigueur.
Je suis consciente que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, que le plagiat est considéré comme une faute grave au sein de l'Université, et qu'en cas de manquement aux règles en la matière, j'encourrais des poursuites non seulement devant la commission de discipline de l'établissement mais également devant les tribunaux de la République Française.

Fait à Nancy , le 27 mai 2020

9.1.2 Niels TILCH

Je soussignée Niels TILCH,
Élève-ingénieur régulièrement inscrit en 1ère année à TELECOM Nancy
Numéro de carte de l'étudiante :
Année universitaire : 2019–2020
Auteur du rapport et code informatique associé au projet intitulé :

Analyse des sentiments

Par la présente, je déclare m'être informé sur les différentes formes de plagiat existantes et sur les techniques et normes de citation et référence. Je déclare en outre que le travail rendu est un travail original, issu de ma réflexion personnelle, et qu'il a été rédigé entièrement par mes soins.
J'affirme n'avoir ni contrefait, ni falsifié, ni copié tout ou partie de l'œuvre d'autrui, en particulier texte ou code informatique, dans le but de me l'accaparer.
Je certifie donc que toutes formulations, idées, recherches, raisonnements, analyses, programmes, schémas ou autre créations, figurant dans le document et empruntés à un tiers, sont clairement signalés comme tels, selon les usages en vigueur.
Je suis conscient que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, que le plagiat est considéré comme une faute grave au sein de l'Université, et qu'en cas de manquement aux règles en la matière, j'encourrais des poursuites non seulement devant la commission de discipline de l'établissement mais également devant les tribunaux de la République Française.

Fait à Nancy , le 27 mai 2020

9.1.3 Léa TOPRAK

Je soussignée Léa TOPRAK,
Élève-ingénieure régulièrement inscrite en 1ère année à TELECOM Nancy
Numéro de carte de l'étudiante : 31917129
Année universitaire : 2019–2020
Auteur du rapport et code informatique associé au projet intitulé :

Analyse des sentiments

Par la présente, je déclare m'être informée sur les différentes formes de plagiat existantes et sur les techniques et normes de citation et référence. Je déclare en outre que le travail rendu est un travail original, issu de ma réflexion personnelle, et qu'il a été rédigé entièrement par mes soins.
J'affirme n'avoir ni contrefait, ni falsifié, ni copié tout ou partie de l'œuvre d'autrui, en particulier texte ou code informatique, dans le but de me l'accaparer.
Je certifie donc que toutes formulations, idées, recherches, raisonnements, analyses, programmes, schémas ou autre créations, figurant dans le document et empruntés à un tiers, sont clairement signalés comme tels, selon les usages en vigueur.
Je suis consciente que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, que le plagiat est considéré comme une faute grave au sein de l'Université, et qu'en cas de manquement aux règles en la matière, j'encourrais des poursuites non seulement devant la commission de discipline de l'établissement mais également devant les tribunaux de la République Française.

Fait à Nancy , le 27 mai 2020

9.2 Liste des figures

Table des figures

9.3 Bibliographie

Références

[1] maheshmore1. How much data does google handle ? ? 06 2017.

[2] Reuters. Google faces $5 billion lawsuit in u.s. for tracking 'private' internet use. 06 2020.

[3] Thomas Fuhr. *Conception, preuves et analyse de fonctions de hachage cryptographiques. Cryptographie et sécurité*. HAL Id, 2011.

[4] Christina Boura. *Analyse de fonctions de hachage cryptographiques.. Cryptographie et sécurité*. HAL Id,université Pierre et Marie Curie - Paris VI, 2012.

[5] *Fonction de hachage*. Wikipédia, https ://fr.wikipedia.org/wiki/Fonction_de_hachage.

[6] Antoine Genitrini. *Algorithmique Avancée : MÉTHODES DE HACHAGE*. 2018-2019.

[7] Manon RUFFINI. Hachage parfait. http ://perso.eleves.ens-rennes.fr/ mruffini/Files/Other/hachage.pdf.

[8] Frédéric Bayart. *Le paradoxe des anniversaires et la cryptographie*. BiblMath, http ://www.bibmath.net/crypto/index.php ?action=affichequoi=chasseur/anniversaire.

[9] Ediciones anónimas. *Función Hash*. http ://seguridad.unicauca.edu.co/criptografia/funcion_hash_wikipedia.pdf.

[10] Aranud. *Rainbow tables*. https ://mieuxcoder.com/2008/01/02/rainbow-tables/.

[11] Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. Ripemd-160 : A strengthened version of ripemd. 02 2000.

[12] Harshvardhan Tiwari. Merkle-damgård construction method and alternatives : A review. *Journal of Information and Organizational Sciences*, 41 :283–304, 12 2017.

[13] La langue française. *Figures de style : le guide complet*. https ://www.lalanguefrancaise.com/litterature/figures-de-style-guide-complet/.

[14] Jihen Karoui. *Automatic Irony Detection in Users Generated Content*. Theses, Université de Toulouse 3 Paul Sabatier ; Faculté des Sciences Economiques et de Gestion, Université de Sfax (Tunisie), September 2017.

[15] https ://fr.sttmedia.com/frequences-de-lettres anglais. *Alphabet et Fréquences de Lettres : Anglais (English)*. sttmedia.com, Consulté le 16 mai 2020.

[16] Anonyme. *Algorithmique-ROB3TD4-Tablesdehachage*. http ://www-desir.lip6.fr/ spanjaard/cours/FeuilleTD4.pdf.

[17] *SHA-1*. Wikipédia, https ://fr.wikipedia.org/wiki/SHA-1.

[18] Pierre Langlois. *Représentation binaire de nombres entiers et opérations arithmétiques de base*. Polytechnique Montréal, page 11.

[19] Noémie Lecoq. *Jean-Michel Jarre, musique et IA : «Des algorithmes vont pouvoir déterminer quelle chanson vous touche»*. Korii, 15/05/201.

9.4 Comptes rendus de réunions.

PPII

Compte-rendu du 20 avril 2020

Présent: Élisa DARGENT, Niels TILCH, Léa TOPRAK

Absent: Aucun

Ordre du jour

1. Discuter du sujet.
2. Répartir le travail.

Informations échangées

1. Chacun a partagé son ressenti sur le sujet.
2. Nous avons réfléchi sur les différents attendus du projet notamment les méthodes que nous pourrions mettre en place pour l'implémentation de chacun.
3. Nous avons décidé d'un horaire pour coder ensemble les attendus du projet.
4. Nous avons réparti les différentes tâches.

Remarques/questions

1. Le sujet est très intéressant et les attendus semblent faisables.
2. Est-ce qu'un état de l'art est attendu ?

Actions à suivre

Description	Responsable	Délai	Livrable	Validé par
Rédiger le README	Élisa	1 jour	README	Niels et Léa
Ajouter un fichier .gitignore	Léa	1 jour	.gitignore	Élisa et Niels
Rédiger le CR1	Léa	1 jour	CR1	Élisa et Niels

Prochaine réunion: Mardi 21 avril 2020 à 14:00

PPII

Compte-rendu du 20 avril 2020

Présent: Élisa DARGENT, Niels TILCH, Léa TOPRAK

Absent: Aucun

Ordre du jour

1. Discuter du sujet.
2. Répartir le travail.

Informations échangées

1. Chacun a partagé son ressenti sur le sujet.
2. Nous avons réfléchi sur les différents attendus du projet notamment les méthodes que nous pourrions mettre en place pour l'implémentation de chacun.
3. Nous avons décidé d'un horaire pour coder ensemble les attendus du projet.
4. Nous avons réparti les différentes tâches.

Remarques/questions

1. Le sujet est très intéressant et les attendus semblent faisables.
2. Est-ce qu'un état de l'art est attendu ?

Actions à suivre

Description	Responsable	Délai	Livrable	Validé par
Rédiger le README	Élisa	1 jour	README	Niels et Léa
Ajouter un fichier .gitignore	Léa	1 jour	.gitignore	Élisa et Niels
Rédiger le CR1	Léa	1 jour	CR1	Élisa et Niels

Prochaine réunion: Mardi 21 avril 2020 à 14:00

PPII

Compte-rendu du 28 Avril 2020 à 19h

Présent: Élisa DARGENT, Niels TILCH, Léa TOPRAK

Ordre du jour

1. La gestion des collisions par Niels.
2. Rappels sur l'importance des tests unitaires et de la gestion des fuites mémoires.
3. Analyse fonctions de hachage par Élisa.

Informations échangées

1. Explications des codes par chacun.
2. Vérifications des fuites de mémoires

Remarques/questions

1. Importance de la gestion des collisions.
2. Importance des tests unitaires au fur et à mesure.

Actions à suivre

Description	Responsable	Délai	Livrable	Validé par
Corriger les fuites mémoires	Groupe	5 jours	–	Groupe
Fin des collisions + tests	Niels	5 jours	code	Léa et ÉLisa
Diagramme de Gantt	Niels	3 jours	Gantt	Léa et ÉLisa
GdP : l'équipe	Léa	3 jours	Rapport	Élisa et Niels
Tests de la fonction score	Léa	5 jours	code	Élisa et Niels
Suite de la fonction de hachage + tests	Élisa	5 jours	code et graphique	Niels et Léa

Prochaine réunion: Mardi 5 mai 2020

PPII

Compte-rendu du 5 mai 2020 à 19h

Présent: Élisa DARGENT, Léa TOPRAK, Niels TILCH

Ordre du jour

1. Rapport
2. Avancement de la gestion des collisions, retour sur la casse.
3. Début des travaux complémentaires pour Léa.
4. Rappels de la nécessité des tests unitaires et de la gestion des fuites mémoires.

Informations échangées

1. Commandes pour vérifier les fuites mémoires.
2. Idées de nouvelles fonctions de hachages et des études sur ces dernières.

Remarques/questions

1. Pour la gestion de la mémoire : *valgrind –leak-check=full ./toto* et *gcc -Wall -pedantic -fsanitize=address -o toto toto.c**
2. Attention à ne pas traîner pour la rédaction du rapport et à rendre les résultats visuels.

Actions à suivre

Description	Responsable	Délai	Livrable	Validé par
Corriger les fuites mémoires	Groupe	5 jours	–	Groupe
Suite des collisions et de ses tests	Niels	5 jours	code	Léa et ÉLisa
Gantt	Niels	3 jours	Gantt	Léa et ÉLisa
GdP : l'équipe	Léa	3 jours	Rapport	Élisa et Niels
Étude statistique des données	Léa	5 jours	code et graphique	Élisa et Niels
Tests de la fonction : eval d'un com	Léa	5 jours	code	Élisa et Niels
Rédiger le CR	Élisa	1 jour	CR	Niels et Léa
Rédiger l'introduction du rapport	Élisa	2 jours	Introduction	Niels et Léa
Suite de la fonction de hachage/les tests	Élisa	5 jours	code et graphique	Niels et Léa

Prochaine réunion: Samedi 09 mai 2020

PPII

Compte-rendu du 9 mai 2020

Présent: Élisa DARGENT, Niels TILCH, Léa TOPRAK

Ordre du jour

1. La gestion des collisions par Niels.
2. Avancement sur l'étude statistique des données par Léa.
3. Analyse des différentes fonctions de hachage par Élisa.
4. Rappels sur l'importance des tests unitaires et de la gestion des fuites mémoires.

Informations échangées

1. Explications des codes par chacun.
2. Idées pour réduire la taille du hash renvoyé par la fonction SHA-1.

Remarques/questions

1. Importance de la gestion des collisions.
2. Soucis de rendre les résultats visuels.

Actions à suivre

Description	Responsable	Délai	Livrable	Validé par
Corriger les fuites mémoires	Groupe	5 jours	–	Groupe
Fin des collisions + tests	Niels	5 jours	code	Léa et ÉLisa
Diagramme de Gantt	Niels	3 jours	Gantt	Léa et ÉLisa
Mettre le CR2 dans le rapport	Niels	3 jours	CR	Élisa et Léa
Choisir une partie complémentaire	Niels	3 jours	–	Élisa et Léa
Installer Valgrind	Léa	1 jour	–	Élisa et Niels
GdP : l'équipe	Léa	3 jours	Rapport	Élisa et Niels
Tests de la fonction score	Léa	5 jours	code	Élisa et Niels
Suite de l'étude statistique des données	Léa	5 jours	code et graphique	Élisa et Niels
Rédiger le CR	Léa	1 jour	CR	ÉLisa et Niels
Revoir l'introduction du rapport	Élisa	2 jours	Introduction	Niels et Léa
Suite de la fonction de hachage + tests	Élisa	5 jours	code et graphique	Niels et Léa

Prochaine réunion: Mercredi 13 mai 2020

PPII

Compte-rendu du 23 mai 2020 à 19h

Présent: Élisa DARGENT, Niels TILCH, Léa TOPRAK

Ordre du jour

1. La construction du main par Niels.
2. Avancement de la première partie de l'étude statistique des données par Léa.
3. Fin de l'analyse des différentes fonctions de hachage par Élisa et début de l'analyse de la table. Suite du rapport

Informations échangées

1. Explications de la construction du main par Niels : possibilité de rentrer un commentaire, afficher les mots les plus/moins présents, positifs, négatifs, affichage du facteur de compression, du temps de recherche des mots du commentaire et éventuellement du temps d'ajout de commentaire dans la base de donnée.
2. Léa va commencer à calculer la moyenne des notes de la table de même que l'espérance et d'autres données pertinentes.
3. Élisa rencontre une segfault dans le remplissage de la table avec SHA1. La 'chasse' de la segFault mène a la fonction GET de la table de hachage mais aucune incohérence n'a été relevé. De plus, cette fonction rempli son rôle lors l'usage de d'autre fonction. Cependant SHA-1 a été testé,a fuites mémoire et a déjà été utilisé plusieurs fois. Plus étonnant, remplir une table avec une base de données petite (mais contenant des collisions) fonctionne.

Remarques/questions

1. Rappels sur l'importance des tests unitaires et de la gestion des fuites mémoires.

Actions à suivre

Description	Responsable	Délai	Livrable	Validé par
Corriger les fuites mémoires	Groupe	5 jours	–	Groupe
Continuer le main	Niels	5 jours	code	Léa et ÉLisa
Rapport	Niels	3 jours	Gantt	Léa et ÉLisa
Tests de ces fonctions	Niels	5 jours	CR	Élisa et Léa
Rapport	Léa	5 jours	Rapport	Élisa et Niels
Suite de l'étude statistique des données	Léa	5 jours	code et graphique	Élisa et Niels
Rédiger le CR	ÉLisa	1 jour	CR	ÉLisa et Niels
Continuer le rapport	Élisa	2 jours	Introduction	Niels et Léa
Suite de l'étude de la table + tests	Élisa	5 jours	code et résultats	Niels et Léa

Prochaine réunion: Samedi 27 mai 2020 19h

PPII

Compte-rendu du 27 mai 2020 à 19h

Présent: Élisa DARGENT, Niels TILCH, Léa TOPRAK

Ordre du jour

1. Les avancements depuis la dernière réunion
2. Création des tests
3. Création de l'interface finale
4. Étude de la mémoire
5. Gestion de Projet

Informations échangées

Les avancement depuis la dernière réunion

- Léa a continué à programmer les fonctions concernant la partie statistiques avec des erreurs mais qui ont été facilement corrigé.
- Il y a un ajout de quelques lignes de codes dans la fonction filledTable de la part de Léa et Élisa afin de prendre en compte de la casse. Ainsi, l'ensemble des clef sont en minuscule.
- Il y a eu des ajouts de plus de contractions ('ll, 'n,...) de la part d'Élisa pour éviter d'avoir de fausses données.

Création des Tests

- Il est remarqué de la part d'Élisa d'effectuer les tests au dur et à mesure pour vérifier le bon fonctionnement des programmes et éviter des effets néfaste sur le bon fonctionnement du programme.
- Léa ayant presque terminée les programmes sur les statistiques, elle se propose de faire ses tests pour ses programmes.

Création de l'interface finale

- Niels se propose de programmer l'interface finale afin de regrouper tout les fonctions. Ce sera un carnet interactif. La forme du programme sera décider dans les jours à venir.

Étude de la mémoire

- Nous avons débattu sur ce qu'une bonne fonction de hachage es: Est-ce une fonction qui a peu de collision ou bien peu de hash pour économiser de l'espace mémoire inutilisé ?

  De par le groupe, on favorise les fonctions ayant le moins de collisions une fois que le hashcode est calculé.

- Élisa a créé une fonction de hachage Hachage poids permettant de mettre de l'importance sur la place de la lettre dans le mot. Néanmoins, le facteur de compression n'a pas changé de celui des autres : on n'a pas d'amélioration notable.
- Élisa rappelle l'importance de régler les fuites de mémoires.

Gestion de projet

- Niels a terminé la matrice RACI et a terminé le Gantt pour la fin du projet.

Actions à suivre

Description	Responsable	Délai	Livrable	Validé par
Corriger les fuites mémoires	Groupe	5 jours	–	Groupe
Descriptions des fonctions dans le code	Groupe	5 jours	Rapport	Groupe
Continuer le main	Niels	5 jours	code	Léa et Élisa
Rapport	Niels	3 jours	Descriptions Fonctions	Léa et Élisa
Tests de ces fonctions	Niels	5 jours	CR	Élisa et Léa
Rapport	Léa	5 jours	Rapport	Élisa et Niels
Suite de l'étude statistique des données	Léa	5 jours	code et graphique	Élisa et Niels
Continuer le rapport	Élisa	2 jours	Introduction	Niels et Léa
Continuer l'analyse des fonctions de hashage	Élisa	5 jours	code et résultats	Niels et Léa

Prochaine réunion: Lundi 1er Juin 2020 - 18h

PPII

Compte-rendu du 1 Juin 2020 à 19h30

Présent: Élisa DARGENT, Niels TILCH, Léa TOPRAK

Ordre du jour

1. Bilan de ce qu'on a fait
2. Rappel de l'importance des tests et des fuites mémoires par Élisa
3. Répartition du travail sur la rédaction du rapport

Informations échangées

Dans la fonction *ajouteNouveauMot*, Niels avait oublié d'initialiser la variable nextW à **NULL**.
La fonction SHA1 ajoute un symbole étrange dans certains commentaires.
La fonction *split* occupe beaucoup de place en mémoire.

Remarques/questions

1. Comment répartir le travail dans le rapport ?
2. Est-ce qu'on aura le temps de tout finir ?

Actions à suivre

Description	Responsable	Délai	Livrable	Validé par
Corriger les fuites mémoires	Groupe	5 jours	–	Groupe
Descriptions des fonctions dans le code	Groupe	5 jours	–	Groupe
Continuer le main	Niels	5 jours	code	Léa et Élisa
Rapport	Niels	3 jours	Descriptions Fonctions	Léa et Élisa
Tests de ces fonctions	Niels	5 jours	CR	Élisa et Léa
Rapport	Léa	5 jours	Rapport	Élisa et Niels
Suite de l'étude statistique des données	Léa	5 jours	code et graphique	Élisa et Niels
Rédiger le CR	ÉLisa	1 jour	CR	ÉLisa et Niels
Continuer le rapport	Élisa	2 jours	Introduction	Niels et Léa
Suite de l'étude de la table + tests	Élisa	5 jours	code et résultats	Niels et Léa

Prochaine réunion: -