ASD2 Laboratoire 4

Introduction

Les tables de hachage sont des structures de données permettant de lier une clé à un élément. Dans ce laboratoire, nous allons traiter deux jeux de données identiques dans la forme (AVS, Nom, Prénom, Genre, Date de naissance), mais avec des longueurs différentes (10'000 et 1'000'000 d'entrées).

Le but de ce laboratoire est d'analyser différentes fonctions de hachage, essentielles pour créer la table. Une « bonne » fonction est idéalement une fonction qui ne crée pas ou peu de collisions, tout en ayant des temps d'insertion et de recherche très bas.

Le but de la deuxième partie est d'implémenter une fonction de hachage, qui soit relativement efficace.

Partie 1

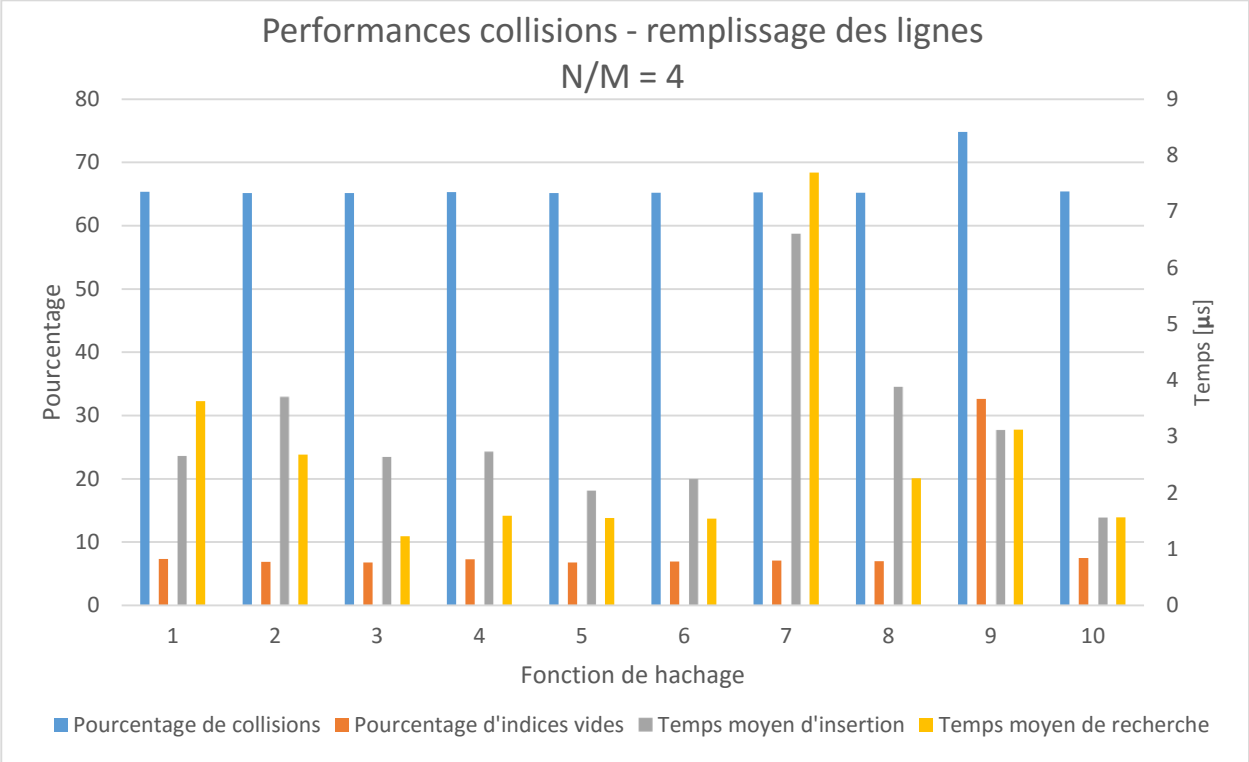
Pour cette partie, nous avons analysé les performances des différentes fonctions de hachage fournies. L'implémentation du hash a été faite sous forme de liste chainée. Nous avons fait différents tests, en faisant varier le MAX_LOAD_FACTOR (2, 4 ou 6). La valeur de 4 représente le MAX_LOAD_FACTOR habituellement utilisé pour des listes chaînées.

Numéros d'algorithme	**Algorithme**
1	Int
2	Long
3	PolILj2EE
4	PolILj31EE
5	PolILj37EE
6	Stl
7	Sha256
8	City
9	WithoutAVS
10	WithoutAVS (Modifié)

Notons cependant que l'algorithme 10 ne sera pas pris en compte dans la partie 1 car il correspond à l'algorithme demandé dans la partie 2.

Premier jeu de données (10'000 entrées)

Max Load Factor = 4



Si l'on regarde en particulier le nombre de collisions et de buckets vides, leur nombre est approximativement semblable partout sauf pour le 9ième algorithme qui est légèrement derrière. Cela s'explique par le fait que cet algorithme est un algorithme très simple (Uniquement le hash du nom). Le fait de n'utiliser que le nom de la personne pour le hash est une assez mauvaise idée, puisque plusieurs personnent peuvent avoir le même nom de famille. En réalité, le nom n'est pas une clé primaire à un individu, autrement dit cela ne caractérise pas obligatoirement un individu en particulier.

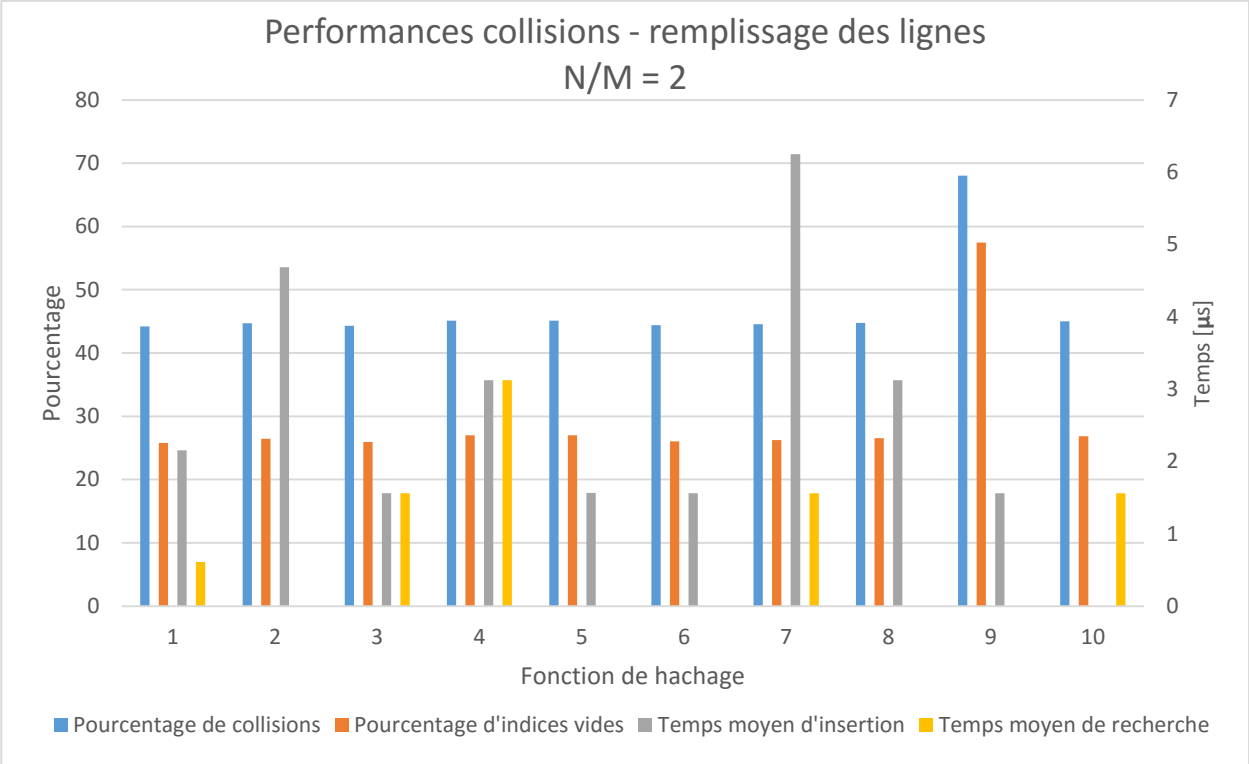
Vu que le nombre d'entrées est fixe, plus il y a de collisions, plus il y a d'entrées qui sont, par définition, dans le même bucket. Ainsi, cela implique donc aussi un plus grand nombre de buckets vides.

Au niveau des temps d'insertion et de recherche, on voit clairement que c'est l'algorithme 7 qui est le moins performant. Cet algorithme correspond au SHA-256, une fonction de hachage couramment utilisée aujourd'hui. Le fait que cette fonction soit aussi lourde est dû à ses nombreuses opérations couteuses (Un prétraitement puis un condensé du hash). Malgrès ceci, on voit qu'elle n'a pas un score de collisions moins élevé que les autres tout en ayant des performances bien moins bonnes. Son utilisation est dû au fait qu'elle est très sécurisée et (presque) impossible à reverse engineerer. Il est fortement déconseillé d'utiliser cette fonction-ci pour le traitement des hash dans notre structure de données.

Les autres fonctions ont des performances à peu près similaires mais la fonction 3 se démarque par son temps de recherche acceptable et la 5 pour son temps d'insertion.

Max Load Factor = 2

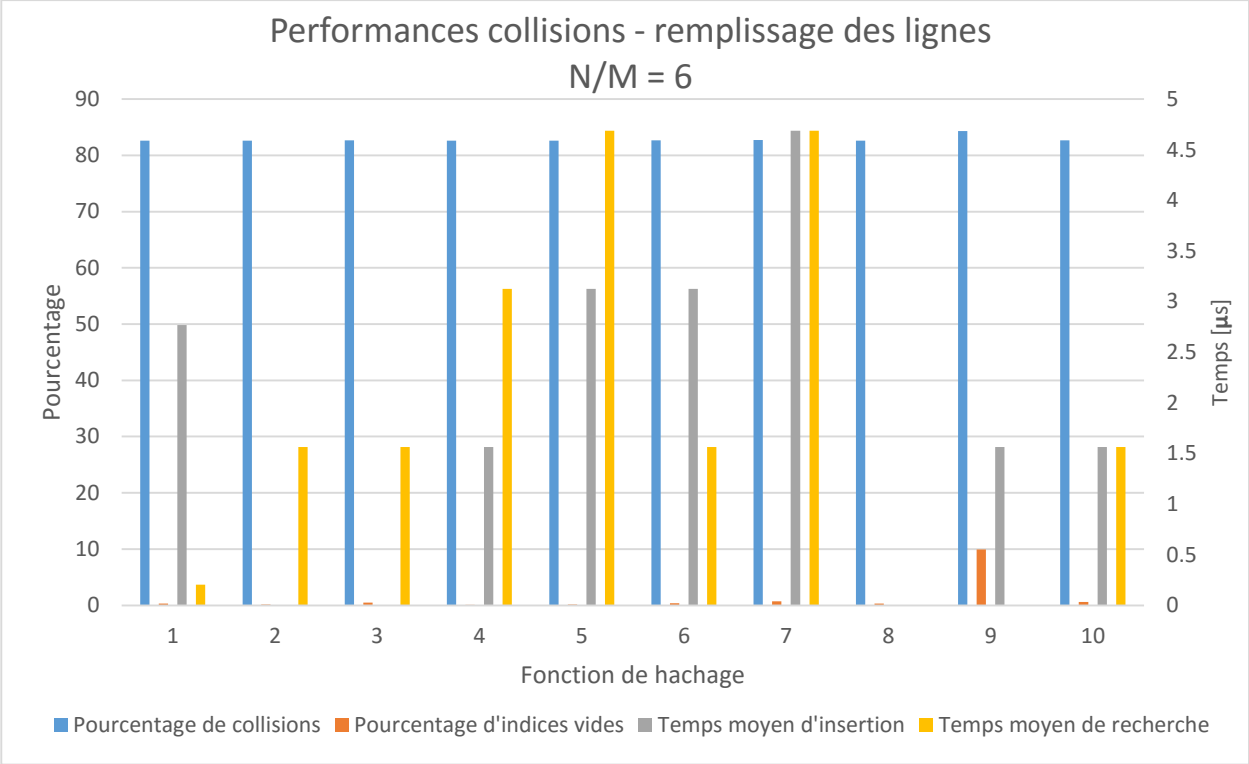
Nous avons cette fois changé le MAX_LOAD_FACTOR à 2 afin d'analyser les différences de performances entre les différentes fonctions de hachage.



Globalement, au niveau des collisions et des buckets vides, il n'y a pas de différences majeures, relativement aux fonctions. Toutefois, on voit quand même une augmentation du nombre de buckets vides et un pourcentage de collisions moindre. Cela est dû au fait que les buckets deviennent moins grand mais qu'il y en a plus et donc ils sont moins souvent remplis car il y a bien plus de valeurs de clé possible pour chaque hash. Si nous représentons la liste chainée comme un tableau, cela voudrait dire que nous avons plus de ligne mais moins de colonnes possibles pour chaque ligne. L'algorithme 9 reste le moins bon pour les mêmes raisons que précédemment.

Au niveau des temps d'insertion et de recherche, il y a quelques différences par rapport au factor qui valait 4. La fonction 2 par exemple devient bien moins efficace pour l'insertion mais devient super efficace au niveau de la recherche. La 4 quand à elle, devient moins efficace aussi bien pour l'insertion que pour la recherche.

Max Load Factor = 6



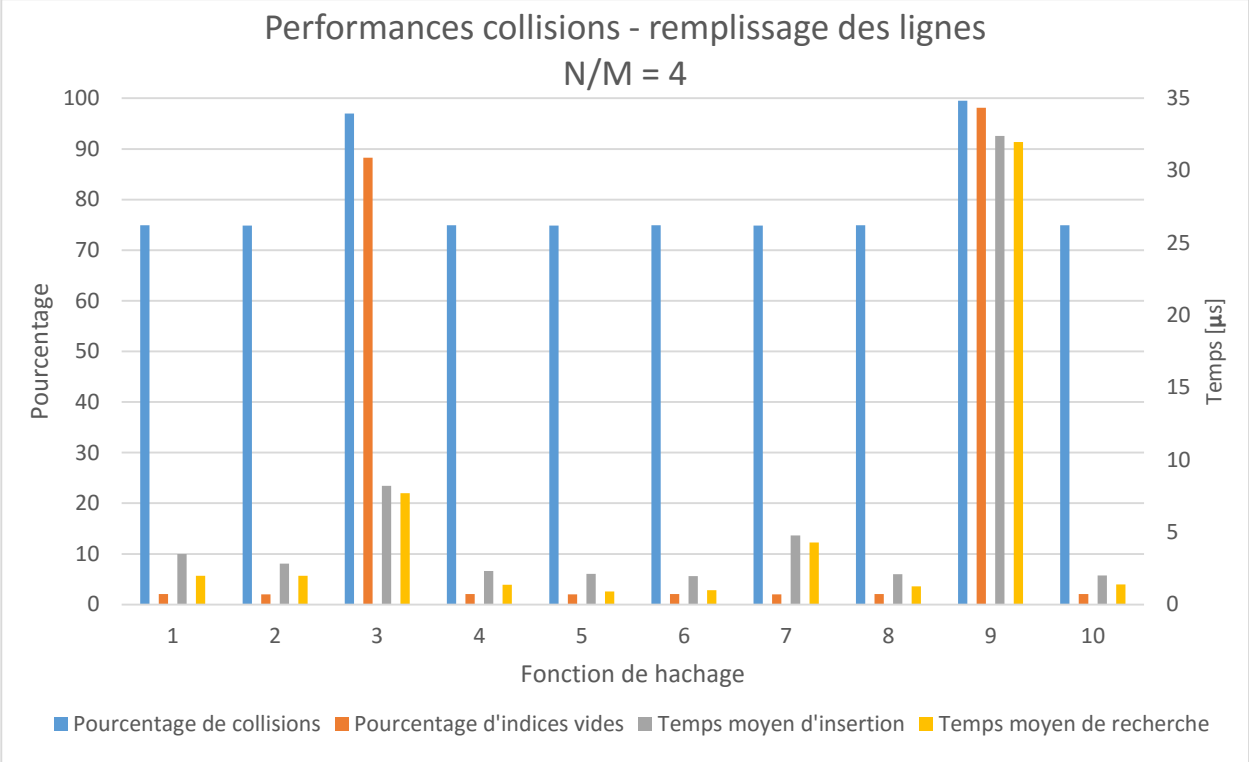
Nous constatons ici une très clair hausse du pourcentage de collisions avec une extrême diminution des buckets vides. Cette fois, si nous représentons notre liste comme un tableau, nous avons alors moins de lignes mais beaucoup plus de colonnes par ligne. Cela implique donc qu'il y aura beaucoup moins de possibilité de clé pour chaque hash et donc la quasi-totalité des buckets seront plein. Mais cela veut aussi dire que beaucoup auront la même clé, et donc il y aura un nombre de collisions élevé.

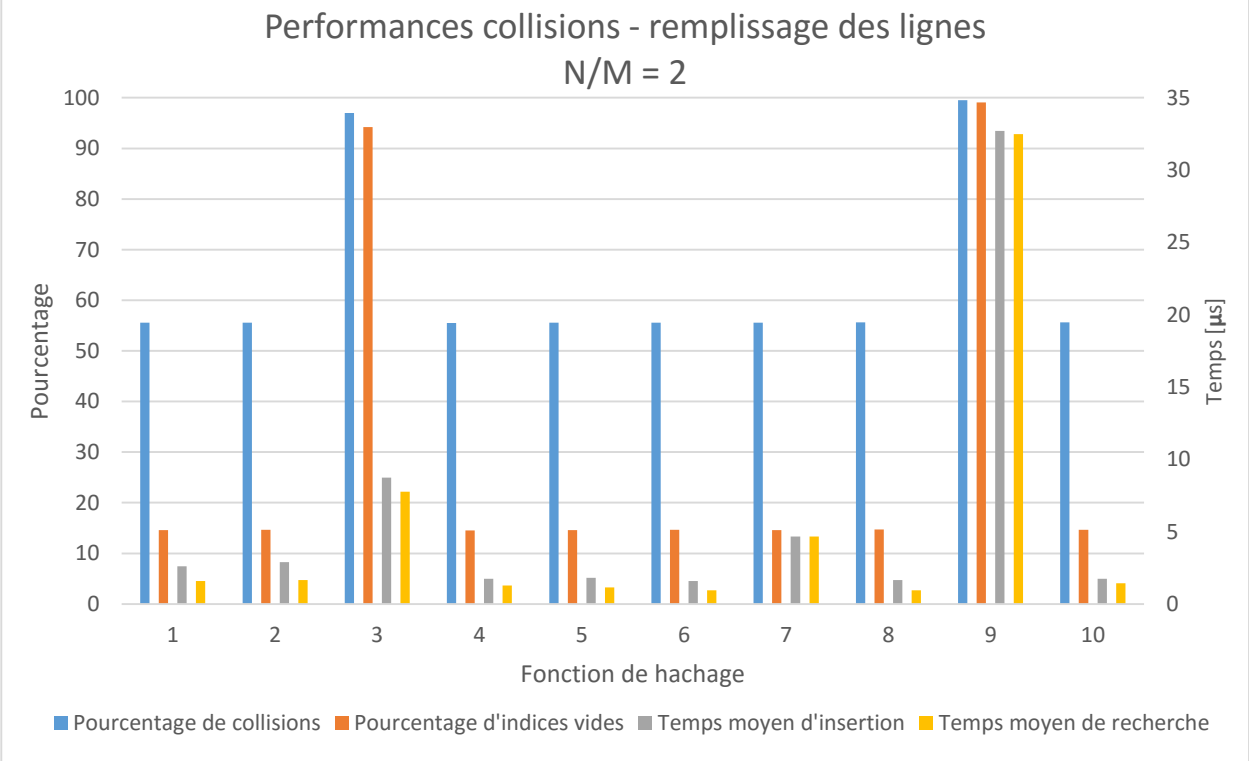
Pour les temps, on peut voir que les fonctions 2,3 et 8 ont des temps d'insertion quasi-instantanés. Pour la recherche, ce sont les algorithmes 1, 8 et 9 qui se démarquent, alors que d'autres comme le 5 ou 7 sont assez mauvais.

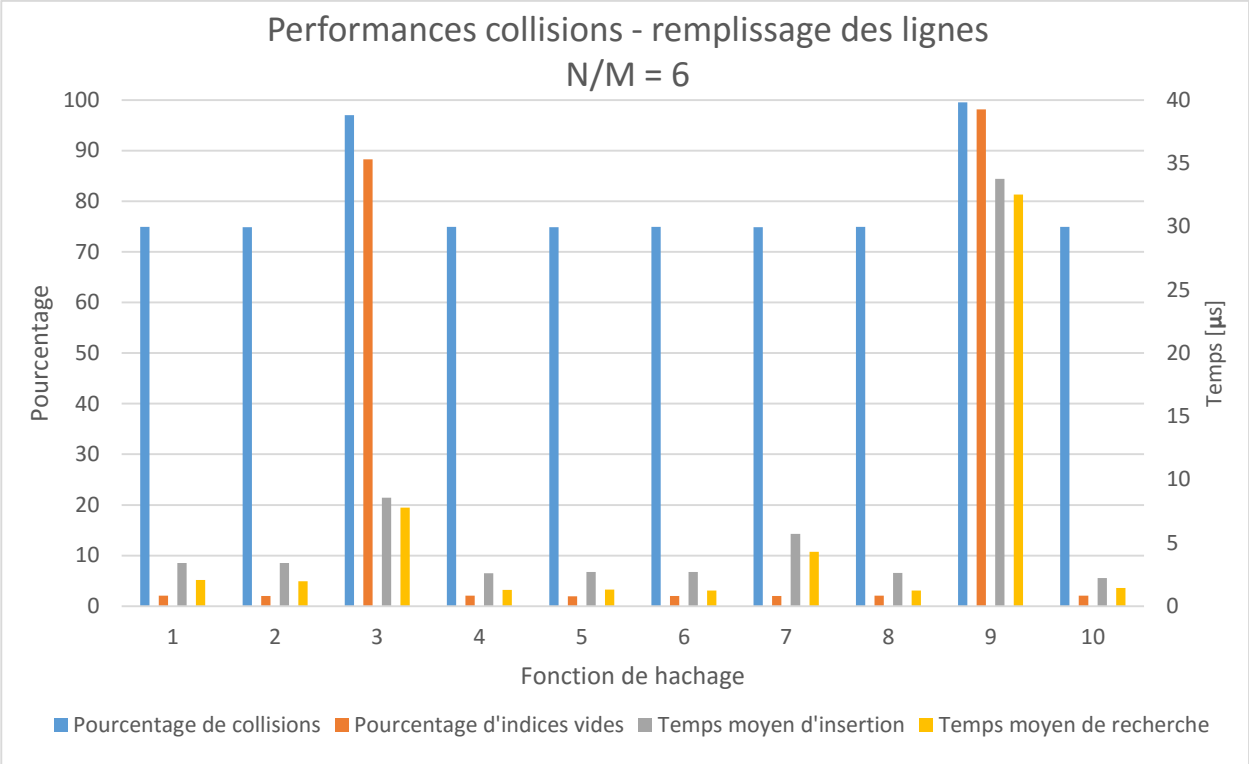
Bien que les performances d'insertion et de recheche sont assez incroyables pour certains algorithmes sur ce jeu de données, ce *Max Load Factor* de 6 est difficilement exploitable puisqu'il génère un beaucoup trop grand nombre de collisions.

Deuxième jeu de données (1’000'000 entrées)

Voici les trois graphiques identiques à ceux présentés précédemment, mais cette fois représentant les fonctions de hachage opérant sur un jeu de données 100 fois plus important.







Alors que la fonction 9 se démarquait légèrement des autres pour le 1er fichier, nous pouvons voir ici que cet algorithme est clairement plus mauvais que les autres. Le n°3, qui était un des meilleurs pour le premier fichier, n'est visiblement pas adaptés aux grands nombre d'entrées. Il s'approche tout près des 100% de collisions, tout en ayant des temps d'insertion et de recherche approximativement 4 fois plus élevés que les autres algorithmes (hormis le 9 bien sûr).

Nous pouvons voir qu'avec un *Max Load Factor* de 2, le nombre de collisions reste en moyenne assez faible (~57%), tout en ayant des temps d'insertion et de recherche relativement identiques aux autres.

Meilleur algorithme

En conclusion à cette partie 1, nous pouvons dire que l'algorithme 3 (PolILj2EE) est probablement le plus adapté pour notre structure de données pour un faible jeu de données (donc le premier fichier) et tant que le MAX_LOAD_FACTOR est assez grand (>= 4). En effet, il a des pourcentages de collisions et de buckets vides identiques aux autres fonctions, tout en ayant des performances meilleurs que les autres, notamment sur la recherche.

Pour les très grands jeux de données, il semblerait que les alogorithmes 4,5,6 et 8 soient tous les quatre très efficaces et relativement identiques du point de vues des performances.

Partie 2

Pour cette partie, il est demandé de modifier l'algorithme 9 (DirectoryWithoutAVS) qui crée un hash à partir du nom d'une personne. Pour cette partie, nous n'avons pas le droit d'utiliser le numéro AVS afin de crée le hash.

Nous avons opté pour une solution simple et performante qui consiste tous simplement à faire un XOR logique entre le hash de la stdLib du nom, du prénom, de la date de naissance et du genre. En effet, si nous prenons toutes ces données, cela corresponds à 99.99% de chance à une personne unique et cela

peut donc être une manière efficaces de crée un hash autant en terme de collisions que de performances. Nous pouvons voir sur les graphes de la partie 1 que dans la quasi-totalité des tests effectués, notre algorithme arrive en tête en ayant en moyenne le même nombre de collisions que les autres mais avec des performances accrues. Cela s'explique par le fait qu'un simple XOR est très rapide à faire pour un ordinateur et cela ne requiert pas d'opérations compliquées comme des compressions. Les seules opérations compliquées sont celle qui sont nécessaire durant le hash définis par la std lib.

Il est cependant important à noter que cela n'est pas un hash sûr. La raison est que si nous possédons 3 hashs sur 4, le quatrième hash peut être trouvé en faisant un simple XOR entre les 3 hash XORé dont on dispose et le hash final. En d'autre terme, si on possède hashNom, hashPrenom et hashDate ainsi que hashFinal, alors hashGenre peut être trouvé en faisant :

hashGenre = (hashNom ^ hashPrenom ^ hashDate) ^ hashFinal

Et cela est valide pour toutes les informations. De plus, cet algorithme utilise la std lib, fonction de hachage n'étant absolument pas secret et qui peut être reverse engineeré par n'importe qui ayant des connaissances de base en sécurité cryptographique. Cependant, ceci n'étant pas un cours de sécurité cryptographique, nous avons privilégié la performance des algorithmes, plutôt que de la bonne sécurité des hash.

Conclusion

Nous avons pu voir dans ce laboratoire que certaines fonctions de hachage étaient bien meilleures que d'autres, au niveau des collisions ou des performances. Il est absolument impossible de mettre au point une bonne fonction de hachage en utilisant uniquement le nom de la personne par exemple.

De plus, bien que certaines fonction paraissaient très efficaces à première vue (pour 10'000 entrées), elles devenaient bien moins bonnes lorsque le nombre d'entrées augmentait. Ainsi, lors de l'implémentation d'une fonction de hachage, il est nécessaire de bien évaluer les besoins de l'applications afin de trouver la fonction idéale.