Rapport projet algorithmique v2 :

Encadré par :Pr.Hafidi

Réalisé par :ISMAIL LBAZRI & MAROUANE NAFII

INTRODUCTION

Problématique :

Comment gérer les doublons ? Cette question est récurrente lorsque l'on souhaite traiter et analyser des données, que ce soit pour les compter, les filtrer, les regrouper ou les supprimer.
La gestion des doublons dans une base de données est très importante. Une mauvaise identification peut nuire aux performances et à l'efficacité des actions menées au sein d'un organisme. Les doublons nuisent à la productivité des équipes commerciales, marketing, de téléprospection, etc. D'après l'étude de 2013 de Dynamics Markets, les doublons représentent 10% du budget gaspillé.

Objectif :

Dans ce projet, On propose trois algorithmes :Algorithme Naïf, Algorithme avec tri par tas , Algorithme Hash en utilisant la distance de LevenShtein afin de répondre à cette problématique.

PRESENTATION DES STRUCTURES DE DONNEES UTILISES :

Tableaux dynamiques :

En informatique, un tableau est une structure de données représentant une séquence finie d'éléments auxquels on peut accéder efficacement par leur position, ou indice, dans la séquence. C'est un type de conteneur que l'on retrouve dans un grand nombre de langages de programmation.
Le temps d'accès à un élément par son index est constant, quel que soit l'élément désiré. Cela s'explique par le fait que les éléments d'un tableau sont contigus dans l'espace mémoire. Ainsi, il est possible de calculer l'adresse mémoire de l'élément auquel on veut accéder, à partir de l'adresse de base du tableau et de l'index de l'élément. L'accès est immédiat, comme il le serait pour une variable simple.
Les limites d'une telle structure viennent de son avantage. Un tableau étant représenté en mémoire sous la forme de cellules contiguës, les opérations d'insertion et de suppression d'élément sont impossibles, sauf si on crée un nouveau tableau, de taille plus grande ou plus petite. Il est alors nécessaire de copier tous les éléments du tableau original dans le nouveau tableau, puis de libérer l'espace mémoire alloué à l'ancien tableau. Cela fait donc beaucoup d'opérations et oblige certains langages fournissant de telles possibilités à implémenter leurs tableaux, non pas sous la forme traditionnelle (cellules adjacentes), mais en utilisant une liste chaînée, ou une combinaison des deux structures pour améliorer les performances.

Tas :

En informatique, un tas est une structure de données de type arbre qui permet de retrouver directement l'élément que l'on veut traiter en priorité. C'est un arbre binaire presque complet ordonné.
Un arbre binaire est dit presque complet si tous ses niveaux sont remplis, sauf éventuellement le dernier, qui doit être rempli sur la gauche. Ses feuilles sont donc à la même distance minimale de la racine, plus ou moins 1. Le Tas est une structure de données utile pour le tri par tas, il permet de construire une file de priorité efficace.
Le temps d'exécution su Tri par Tas est O(n log n).

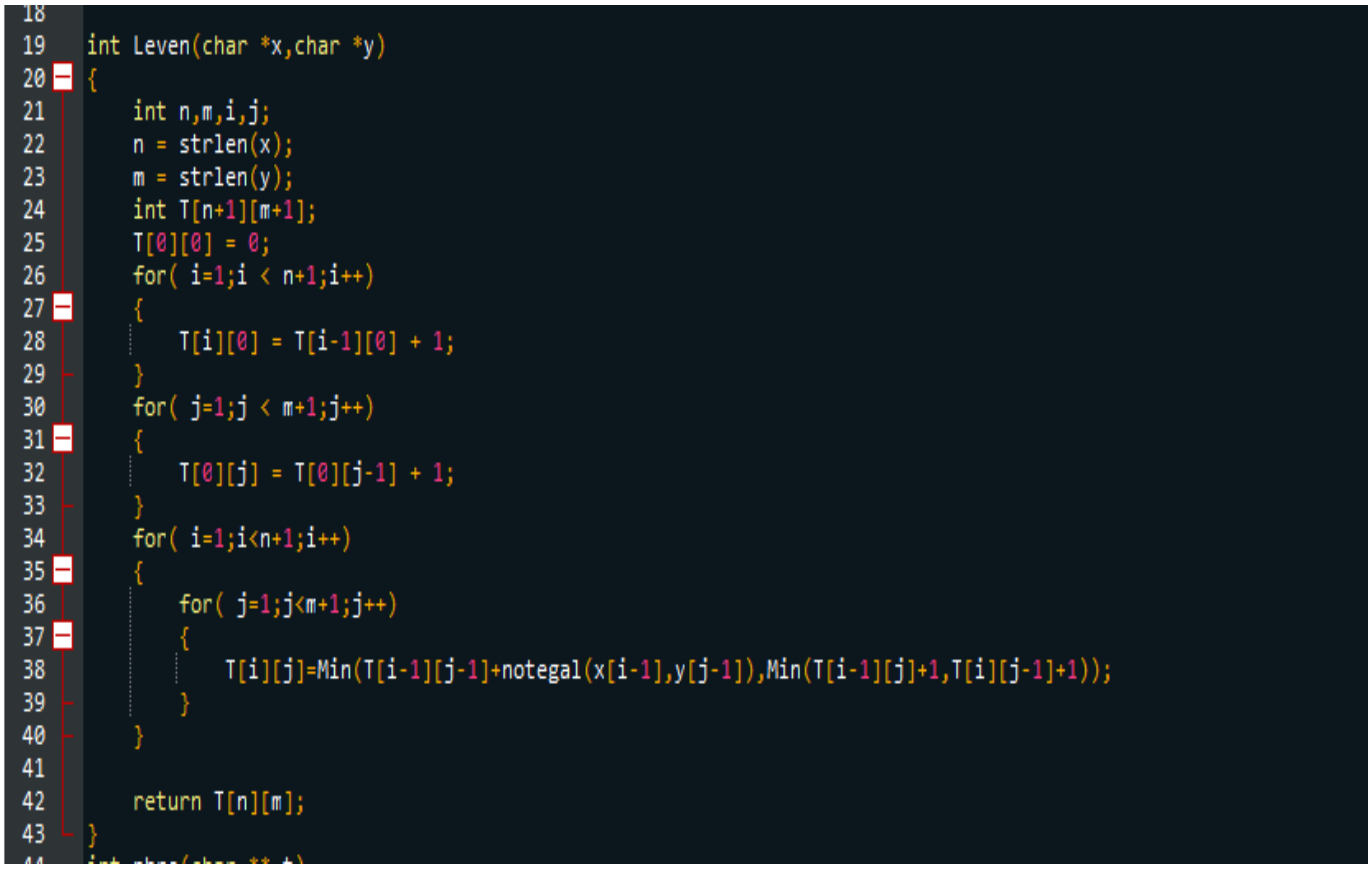
Tableaux de Hachage :

Une table de hachage est, en informatique, une structure de données qui permet une association clé–valeur, c'est-à-dire une implémentation du type abstrait tableau associatif ; en particulier, l'implémentation d'une table des symboles lorsque les clés sont des chaînes de caractères.
Dans une table de hachage, on calcule la position d'un élément à partir de sa propre valeur.
Le principe de hachage n'est pas parfait, il pose certains problèmes :
Il faut nécessairement 0 < h(clé) < taille du tableau
Problèmes de collision : Plusieurs entités ont le même indice.

Dans un tableau de hachage le temps de recherche est constant, l'accès à un élément est indépendant de la taille : Recherche O(1).

Définition de la distance LEVENSHTEIN :

La distance de Levenshtein est une distance, au sens mathématique du terme, donnant une mesure de la différence entre deux chaînes de caractères. Elle est égale au nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer pour passer d'une chaîne à l'autre.
Elle a été proposée par Vladimir Levenshtein en 1965. Elle est également connue sous les noms de distance d'édition ou de déformation dynamique temporelle, notamment en reconnaissance de formes et particulièrement en reconnaissance vocale.
Cette distance est d'autant plus grande que le nombre de différences entre les deux chaînes est grand. La distance de Levenshtein peut être considérée comme une généralisation de la distance de Hamming. On peut montrer en particulier que la distance de Hamming est un majorant de la distance de Levenshtein.
On la représente comme ci-dessous :



```
int Leven(char *x,char *y)
{
    int n,m,i,j;
    n = strlen(x);
    m = strlen(y);
    int T[n+1][m+1];
    T[0][0] = 0;
    for( i=1;i < n+1;i++)
    {
        T[i][0] = T[i-1][0] + 1;
    }
    for( j=1;j < m+1;j++)
    {
        T[0][j] = T[0][j-1] + 1;
    }
    for( i=1;i<n+1;i++)
    {
        for( j=1;j<m+1;j++)
        {
            T[i][j]=Min(T[i-1][j-1]+notegal(x[i-1],y[j-1]),Min(T[i-1][j]+1,T[i][j-1]+1));
        }
    }

    return T[n][m];
}
```

Algorithme naïf

Structure de données:

Structure utilisée dans l'algorithme naïve est un tableau des pointeurs :

L'algorithme naïve est la version plus évidente pour résoudre problème des doublons qui base sur les fonctions suivant :

- Char** MappingToSet (char * nomfichier) copie les noms du fichier dans un tableau dynamique
- Char** purge(char* nomfichier) compare les éléments du tableau en utilisant la distance de livenshtein puis supprime les doublons

Complexité :

n : le nombre des mots dans fichier . et $\mathbf{m_i}$ la taille de chaque mots :

- char** MappingToSet(char * nomfichier) :
  $T(n) = n\sum_{i=1}^{n} \mathbf{mi}$
  au pire : $T(n) = n^*\max(mi)=n^*M$
  donc $T(n)= Q(n^*M)$

- char** purge(char* nomfichier) : $T(\mathbf{n}) = \mathbf{O(n^2)}$

Avantage :

Traite toute possibilité et il retourne un fichier filtré en effet il compare chaque mots avec chaque mot filtré .

Inconvénient :

La complexité augment proportionnellement avec la taille de donnes.

Algorithme tri par tas

Structure de de données :

void Ajout(char *s,char **ta) : fonction qui ajout element dans le tas et le comparer pour conserver la structure de tas

Char *ExtraireMin(char **ta) : fonction qui retourne le minimum (le racine de tas)

et réordonner pour conserver la structure de tas.

Char **LireToTas(char *fichier) : remplir le tableau sous forme de tas à partir d'un fichier.

Char** MappingToTable(char *fichier) : return un tableau trier il s'agit ici de tri par tas en utilisant les deux premiers fonction ci-dessus.

Char **purge(char *fichier) : supprimer les mots dupliqués et return un nouveau fichier ordonner sans dupliques

Complexité :

void Ajout(char *s,char **ta) : O(log(taille)): la hauteur de tas.

char *ExtraireMin(char **ta) : O(log(taille)) .

char **LireToTas(char *fichier) : O(taille*log(taille)).

char **purge(char *fichier) : O(taille2).

Avantage :

Le tri par tas nous permet de rassembler tous les doublons(dupliques) d'un mot l'un a cote de l'autre, donc il est facile de trouver les dupliques de chaque mots, il n'y a pas besoin de parcours tout tableau.

Inconvénient :

Le tri par tas cout beaucoup de temps pour trier le tableau il a besoin de O(taille*log(taille)).

Algorithme de tableau de hachage :

méthode pour résoudre le problème de duplique nous présentons algorithme de hash qui sera servie à la recherche profitons le cout de recherche qui indépendant de la taille de jeu de donnée.

L'algorithme est divisé en sous-programme qui sont :

int calculeHache(char* name) :calcule l'indice du mot name;
void ajoutMot(Maillon** hashTab, char* name) :ajoute un mot name au maillon hashtable ;
char** HashTolist(maillon** hashTab,int n,int N) : écrit les éléments de tableau de hachage dans un tableau de chaine de caractères ;
Maillon** MappingToHashTable(char* nomfichier) :liste les noms d'un fichier dans le tableau de hachage en utilisant la fonction ajoutmot ;
void afficher(maillon **t,int taille) :affiche les éléments du tableau de hachage ;
char** lire(char * nomfichier) :lit les noms du fichier et les insère dans un tableau chaine de caractère ;

char** purge(char* nomfichier) :supprime les doublons du tableau de chaine de caractère en utilisant les fonctions précédentes ;

Structure de hash

Les case de tableau de hachage contient deux champs l'une pour stocker la chaine de caractère est l'autre pour stocker l'adresse d'autre cellule qui sont la même clé.

```
typedef struct maillon
{
        char *val;
        struct maillon *suiv;
} maillon;
```

Calcule de clé (indice) :

Pour calcule indice de chaque chaine nous avons utilisé algorithme suivante : on calcule pour chaque mot la somme des ***code ascii*** après on divise sur la taille de donne :

```
int calculhach(char *mot,int taille)
{
        int c,s=0,i=0;
        while(mot[i]!='\0')
        {
                s=s+mot[i];
                i++;
        }
        c=s%taille;
        return c;
}
```

Complexité

Complexité de recherche : Q(1).

Complexité de l'algorithme :

Soit D un jeu de donne de taille **n** : le nombre des mots dans fichier . et soit $\mathbf{m_i}$ la taille de chaque mots :

- **int** calculeHache(char* name) : pour chaque mots il y a Q($\mathbf{m_i}$).
- char** Lire char * nomfichier) :
  T(n) = $n\sum_{i=1}^{n} \mathbf{mi}$
  au pire : T(n) = n*max(mi)=n*M
  donc T(n)= Q(n*M)
- void ajoutMot(maillon** hashTab, char* name) :
  au meilleur : O(1)
  au pire : O(n) tous les mots ayant le même clé.(cas d'une liste chainé).
- maillon** MappingToHashTable(char* nomfichier) :
  T(n) = n*(1+2+3…n) = n^2*(n+1)/2
  Alors Q(n^2)

char** purge(char* nomfichier) : Q(n^2)(au pire cas)

Avantage :

Le temps de rechercher est O(1) est indépendant de la taille donnée.

Inconvénient :

Il peut avoir de duplique dans le fichier filtrer car l'algorithme compare chaque mot que avec les case qui lui lié et les cinq case qui lui suive.

Simulation et résultats numériques :

*algorithme naïf :

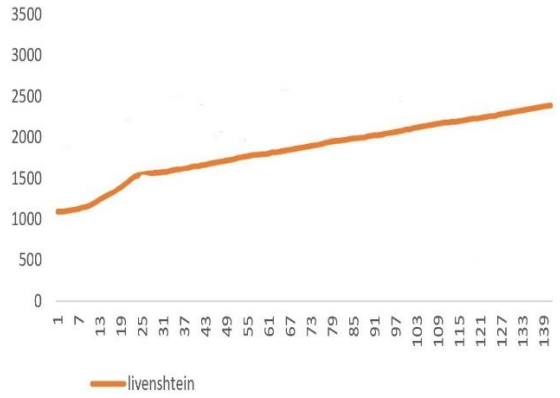
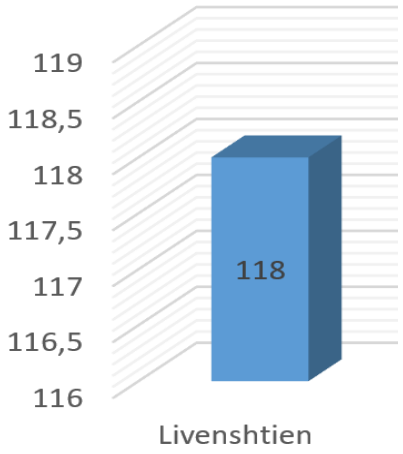


Fig :graphe d'évolution de temps exécution



Nombre de mots supprime avec 142 mots au total

*algorithme tas :

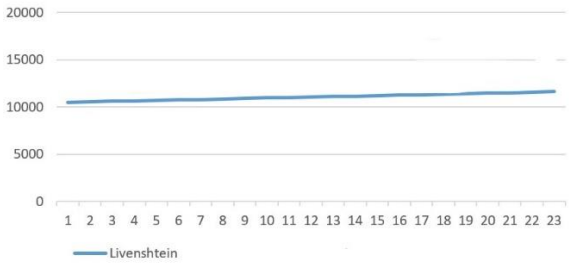
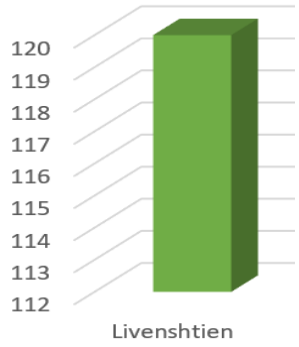


Fig :Graphe : d'évolution de temps exécution



Nombre de mots supprime avec 142mots au total

*algorithme hachage :

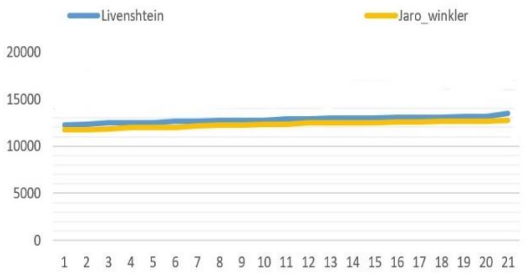
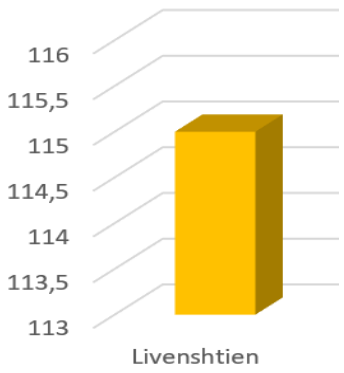


Fig :Graphe : d'évolution de temps exécution



Nombre de mots supprime avec 142mots au total

- ## Discussion de résultats :

Nous remarquons que pour l'algorithme naïf la complexité est élevée dû à l'algorithme compare chaque mot avec tous les mots qui précèdent ce qui coûte plus chère lorsque la taille de données augmente.

Deuxièmement, d'après les graphes d'évolution de temps nous constatons que les graphes ont presque le même temps d'évolution augment après chaque itération dû que le nombre de mots dans tableau filtrer augment donc nombre de comparaison augment ainsi temps évolution croit,.

Troisièmement, on ce qui concerne les graphes de nombre mots supprimés, on remarque que nombre de mots supprimée dans algorithme de hachage moins que les autres a cause que l'algorithme ne compare chaque mot qu'avec les mots qui lui associe a d'autre de 5 case qui lui suive. Il est possible de existence des doublons dans fichier filtrer, pour l'algorithme de tas on constate qu'il moins couteux au niveau de suppression car il parcours pas le tableau, il recherche juste dans les mots suivant jusqu'à ne trouve pas des dupliques, mais le problème c'est que dans certains cas ce algorithme il peut ne pas ordonner les mots dupliquées à côte entre eux donc il va lire des duplique par exemple (mustafa, mostafa, muhamed) il va les ordonner comme(mostafa, muhamed, mustafa) donc on remarque que le mot dupliqué (mustafa) n'est pas à coté de mot (mostafa) donc il va pas le supprimer.

Conclusion :

Finalement, et pour conclure on peut dire que le problème des dupliqués est l'un des problèmes majeure dans l'informatique, avec l'avancement technologique la taille des données augmente proportionnellement, donc la probabilité d'existence des doublons est de plus en plus élevée, et il y a des données qui sont sensibles aux erreurs dans les données d'où les informaticiens et mathématicien cherchent à trouver des solutions pour ce problème-là. Et dans ce projet nous avons travaillé avec trois algorithmes pour comparer les données, après l'étude de ces algorithmes, chacun a ses avantages et inconvénients, on remarque que l'algorithme naïve malgré sa grande complexité est le plus efficace pour supprimer tous les doublons mais il a besoin d'optimisation.