

Université Nazi BONI

Burkina Faso

Ecole Supérieure d'Informatique

Unité-Progrès-Justice



Département Systèmes d'Information

Master 1 en Sciences de Données

Projet d'Heuristique et Algorithme du Texte

Professeur : Dr Zakaria SAWADOGO

Date : Remise 18/07/24

Nom	Prénom
BAMOGO	Abdallah
KOUTOU	Madi
TOURE	P. Brice
TRAORE	Oumar
ZONON	Hamidou

Année Académique 2023-2024

Sommaires

1	Introduction.....	5
2	PARTIE I : Algorithme de Boyer-Moore	5
2.1	Définition	5
2.2	Fonctionnement.....	5
2.2.1	Principe.....	5
2.3	Contraintes d'implémentation	6
2.4	Pré-traitement	6
2.4.1	Première table de sauts	7
2.4.2	Seconde table de saut	9
3	PARTIE II : Implémentation de l'algorithme de Boyer-Moore.....	11
3.1	Prétraitement	11
3.1.1	Table des mauvaises correspondances	11
3.1.2	Table des bons suffixes	11
3.2	Recherche	12
4	PARTIE III : Implémentation de l'application	13
4.1	Création du projet Django.....	13
4.2	Ajout de l'application AlgoApp	13
4.3	Configuration de l'application	14
4.4	Configuration de l'application dans urls.py du projet :	14
4.5	Création de urls.py dans le répertoire de l'application AlgoApp :	14
5	PARTIE IV : Utilisation de l'application	14
5.1	Installation	15
5.2	Exécution.....	15
6	Conclusion.....	16

LISTE DES TABLEAUX

Tableau 1 : Tableau de comparaison première table de sauts	8
Tableau 2 : Tableau de comparaison deuxième table de sauts	10

LISTE DES FIGURES

Figure 1 : fonction pour la table des mauvaises correspondances.....	11
Figure 2 : fonction pour la table des bons correspondances.....	12
Figure 3 : Recherche.....	13
Figure 4 : Configuration de l'application dans urls.py du projet	14
Figure 5 : Création de urls.py dans le répertoire de l'application AlgoApp	14
Figure 6 : Ecran de chargement des fichiers.....	15
Figure 7 : Resultat de similarité.....	16

1 Introduction

L'algorithme de Boyer-Moore est un puissant algorithme de recherche de sous-chîne utilisé pour trouver des occurrences d'un motif (pattern) dans un texte. Il est particulièrement apprécié pour son efficacité et ses performances dans les cas pratiques. Dans les lignes qui suivent nous ferons tout d'abord une documentation de cet algorithme, ensuite nous implémenterons une application sur basé sur cet algorithme, enfin nous donnerons une directive sur l'utilisation de cette application.

2 PARTIE I : Algorithme de Boyer-Moore

2.1 Définition

L'algorithme de **Boyer-Moore** est un algorithme de recherche de sous-chîne particulièrement efficace, qui est utilisé comme référence avec lequel on compare d'autres algorithmes quand on réalise des expériences de recherche de sous-chîne. Il a été développé par Robert S. Boyer et J Strother Moore en 1977.

2.2 Fonctionnement

2.2.1 Principe

L'algorithme de Boyer-Moore est assez surprenant car il effectue la vérification, c'est-à-dire qu'il tente d'établir la correspondance de la sous-chîne à une certaine position, à l'envers. Par exemple, s'il commence la recherche de la sous-chîne WIKIPEDIA au début d'un texte, il vérifie d'abord la neuvième position en regardant si elle contient un A. Ensuite, s'il a trouvé un A, il vérifie la huitième position pour regarder si elle contient le dernier I de la sous-chîne, et ainsi de suite jusqu'à ce qu'il ait vérifié la première position du texte pour y trouver un W.

La raison pour laquelle l'algorithme de Boyer-Moore utilise cette approche à rebours est plus claire si l'on considère ce qui se produit quand la vérification échoue, par exemple si au lieu de trouver un A en neuvième position, un X est lu. Le X n'apparaît nulle part dans la sous-chîne WIKIPEDIA, ce qui signifie qu'aucune correspondance avec la sous-chîne n'existe au tout début du texte, ainsi que dans les huit positions qui la suivent. Après la vérification d'un seul caractère, l'algorithme est capable de passer ces huit caractères et de rechercher le début d'une correspondance à partir de la dixième position dans le texte, juste après le X.

Ce principe de fonctionnement à rebours explique l'affirmation contre-intuitive disant que plus la sous-chaîne est longue, et plus l'algorithme est efficace pour la trouver.

2.3 Contraintes d'implémentation

Il faut noter que, pour que l'algorithme de Boyer-Moore puisse fonctionner de façon efficace, il est nécessaire de pouvoir parcourir le texte (ainsi que la clé cherchée) en le parcourant linéairement en sens inverse, et de pouvoir sauter directement à une position ultérieure du texte sans avoir à le lire intégralement entre les deux positions, ni à devoir relire le texte ou la clé depuis le début, et sans avoir à utiliser de coûteux tampons mémoire compliqués à gérer. Ce n'est pas toujours le cas de tous les flux de fichiers texte à lecture unidirectionnelle.

Et dans le cas où la recherche doit utiliser des comparaisons basées sur la collation (en) de chaînes conformes à des règles linguistiques dans lesquelles certaines différences sont ignorées dans la recherche des correspondances, les éléments à comparer ne seront pas les caractères eux-mêmes mais les éléments de collation précalculés sur la clé et ceux obtenus au fil de l'eau dans le texte, dans lequel il doit être nécessaire de déterminer si les positions sont celles marquant la séparation des éléments de collation (afin de ne pas trouver de faux positifs ni oublier des correspondances lorsqu'on saute directement certaines positions ou quand on lit le texte ou la clé en sens inverse) : cela pose certaines difficultés dans les collations linguistiques comprenant des contractions et expansions, ou encore dans des textes Unicode non normalisés pour lesquels plusieurs codages sont possibles. Mais des algorithmes complémentaires ont été développés pour traiter efficacement cette difficulté (et quelques autres liées à l'étendue du jeu de caractères Unicode (ou l'étendue numérique encore plus grande des poids de collation multiniveau).

2.4 Pré-traitement

À partir de la clé, l'algorithme précalcule deux tableaux indiquant un nombre de positions à sauter après chaque vérification dans le texte. Ces tableaux (parfois appelés « tables de sauts ») exploitent l'information tirée de la vérification effectuée.

- Le premier se base sur le caractère du texte qui a été comparé au dernier caractère de la clé (c'est la première comparaison effectuée puisque les vérifications se font de droite à gauche).
- Le second se base sur le nombre de caractères vérifiés avec succès (ce nombre peut évaluer la taille de la clé, auquel cas la vérification a réussi).

2.4.1 Première table de sauts

Le premier tableau utilise le constat suivant : si on cherche par exemple la clé WIKIPEDIA dans le texte ENCYCLOPEDIA, alors la première vérification sera :

ENCYCLOPEDIA
WIKIPEDIA

Sachant que la première lettre du texte qu'on a comparée est un **E**, il est inutile de vérifier les appariements suivants :

ENCYCLOPEDIA
. WIKIPEDIA

ENCYCLOPEDIA
.. WIKIPEDIA

On peut sauter directement à celui-ci :

ENCYCLOPEDIA
... WIKIPEDIA

Ici, on a donc avancé de trois positions au lieu d'une seule, c'est-à-dire la distance entre la dernière lettre **E** dans la clé et la fin de la clé. Si cette lettre n'apparaissait pas dans la clé, alors on aurait pu sauter toute la longueur de celle-ci.

La première table de sauts est donc facile à calculer : elle associe à chaque caractère de l'alphabet la distance, mesurée depuis la fin de la clé, de la dernière occurrence de cette lettre dans la clé. La dernière position de la clé n'est pas comptée dans les occurrences ; autrement, la distance associée à la dernière lettre de la clé (la lettre **A** dans l'exemple) serait nulle et on resterait sur place après l'avoir lue dans le texte. Les lettres n'apparaissant pas dans la clé sont associées à la longueur de la clé. Un algorithme de calcul est donc :

- pré-remplir toutes les valeurs du tableau avec la longueur de la clé ;
- démarrer de l'avant-dernière position dans la clé avec le compteur à 1, et aller vers le début en incrémentant le compteur ;
- pour chaque position, si le caractère courant n'est pas déjà dans le tableau, l'ajouter avec la valeur actuelle du compteur.

Exemple : avec la clé WIKIPEDIA, le premier tableau est rempli comme suit (pour plus de clarté, les entrées sont données dans l'ordre où elles sont ajoutées dans le tableau) :

Caractère de l'alphabet	Distance à la fin de clé
I	1
D	2
E	3
P	4
K	6
W	8
<i>autres caractères</i>	9

Tableau 1 : Tableau de comparaison première table de sauts

Le lecteur attentif remarquera que le **A**, le dernier caractère de la clé, n'a pas été ajouté dans le tableau. En effet, il ne présente pas d'autre occurrence dans la clé.

Notes de performance :

- ✓ Ce tableau a une taille (nombre total d'entrées) indépendante de la longueur de la clé, mais proportionnelle à la taille de l'alphabet.
- ✎ Si l'alphabet est très grand (par exemple le répertoire universel UCS d'Unicode et ISO/CEI 10646 dont l'alphabet comprend plus d'un million de points de code possibles) sa taille pourrait devenir prohibitive, alors que la plus grande partie de ce tableau contiendrait la valeur par défaut (la longueur de la clé), et son pré-remplissage peut prendre du temps. Cela peut s'optimiser en remarquant que la table ne contient pas de valeur nulle. la valeur par défaut pourra donc être codée par 0 sans ambiguïté. Cette astuce permet d'économiser l'initialisation du tableau dans un environnement qui pré-remplit les tableaux à zéro.
- ✎ De plus, l'algorithme de Boyer-Moore doit son efficacité à sa capacité de sauter des longueurs de texte suffisantes. Il n'est pas nécessaire de sauter la distance maximale, une distance raisonnablement grande (par rapport à la longueur de la clé) suffit. Quand l'alphabet est bien plus grand que la clé, l'algorithme restera efficace si on réduit l'alphabet à des classes de caractères (ou d'éléments de collation) avec une bonne répartition.

- 🔗 Dans ce cas, le tableau ne sera pas indicé par les caractères mais par les classes de caractères : une fonction de hachage simple réduisant ce grand alphabet à (par exemple) un ensemble réduit à 256 classes convenablement distribuées suffira et fonctionnera très efficacement pour des longueurs de clé pouvant aller jusqu'à plusieurs milliers de caractères, la table permettant alors d'effectuer des sauts de 1 à 256 caractères.
- ✓ En revanche, si l'alphabet est extrêmement réduit (par exemple, un alphabet binaire), l'efficacité de l'algorithme sera totalement nulle (par rapport à un algorithme de recherche naïf) avec cette table de sauts. L'astuce consistera à lire le texte et la clé non pas caractère par caractère, mais par groupes de caractères afin d'augmenter l'alphabet à un cardinal suffisant. Par exemple, avec un alphabet binaire, on pourra lire par paquets de 8 caractères, en fixant un caractère de bourrage arbitraire (ou moins probable) pour les caractères (du petit alphabet) qui manquent au début de la clé ou au début du texte mais qui sont nécessaires à la formation de groupes complets de lettres convertis en lettres équivalentes du nouvel alphabet augmenté. On tiendra ensuite compte, lorsque des correspondances sont trouvées entre des groupes de caractères, du nombre de caractères de bourrage utilisés pour ajuster la position du groupe trouvé.

2.4.2 Seconde table de saut

Le second tableau est sensiblement plus compliqué à calculer : pour chaque valeur de N inférieure à la longueur K de la sous-chaîne clé, il faut calculer le motif composé des N derniers caractères de la sous-chaîne K , précédé d'un caractère qui ne correspond pas. Puis, il faut trouver le plus petit nombre de caractères pour lesquels le motif partiel peut être décalé vers la gauche avant que les deux motifs ne correspondent. Par exemple, pour la sous-chaîne clé ANPANMAN longue de 8 caractères, le tableau de 8 lignes est rempli de cette manière (les motifs déjà trouvés dans le texte sont montrés alignés dans des colonnes correspondant à l'éventuel motif suivant possible, pour montrer comment s'obtient la valeur de décalage qui est la seule réellement calculée et stockée dans la seconde table de saut) :

Indice							Motif suivant								Décalage obtenu
							A	N	P	A	N	M	A	N	
0													N		1
1					A	N									8
2									M	A	N				3
3					N	M	A	N							6
4				A	N	M	A	N							6
5			P	A	N	M	A	N							6
6		N	P	A	N	M	A	N							6
7	A	N	P	A	N	M	A	N							6

Tableau 2 : Tableau de comparaison deuxième table de sauts

Notes relatives à la complexité de calcul cette table :

- On remarque que cette table contient autant de lignes que la longueur de clé. Si la clé est longue, les valeurs de décalage dans la table peuvent être elles aussi assez importantes, ce qui va nécessiter une allocation de mémoire de travail peut être importante, souvent plus grande en taille elle-même que la chaîne clé qui utilise un alphabet plus réduit (par exemple 1 octet par caractère) que les entiers alors qu'en fin de compte les longueurs de clé seront importantes (typiquement des entiers codés sur 4 octets).
- La constitution de cette table nécessite là aussi de rechercher des sous-chaînes (toutes les sous-chaînes possibles de la fin de clé) pour en trouver pour chacune la **dernière** occurrence dans la clé, et l'algorithme de Boyer-Moore pourrait être utilisé récursivement (mais en utilisant une recherche sur des textes et clés de direction inversée).
- Au-delà d'une certaine longueur raisonnable (par exemple jusqu'aux 256 derniers caractères de la clé), il peut être inutile d'augmenter la taille de cette table puisque le seul but sera de savoir si des décalages plus grands peuvent être obtenus pour sauter plus vite dans le texte principal. En ne pré-traitant que les 256 derniers caractères en fin de clé, on obtiendra une longueur déjà suffisante pour accélérer la majorité des recherches (mais si on veut aller au-delà, on pourra, lorsque cette table contient déjà la longueur maximale retenue pour le saut et dans les rares cas où cette longueur serait atteinte, et si l'alphabet est assez discriminant, ce que peut indiquer le taux de remplissage de la première table, employer l'algorithme

pour rechercher par récursion (plutôt que par pré-calcul dans cette table) les sous-chaînes dans la clé.

- Mais le plus simple est de borner les décalages à une valeur raisonnable comme 256 et ne pas chercher à aller au-delà. Dans ce cas, cette table aura une taille prédéfinie et ne coûtera rien en termes de complexité pour les clés très longues, puisque son remplissage prendra un temps constant.
- Différentes stratégies sont possibles selon un compromis espace/temps (des stratégies dites « avec cache », ne font aucun pré-calcul des tables, même si elles en utilisent, mais remplissent cette table à la demande en fonction des longueurs de concordances effectivement trouvées à rebours lors de la vérification des occurrences finales déjà trouvées par la première table ci-dessus).

3 PARTIE II : Implémentation de l'algorithme de Boyer-Moore

Dans la partie précédente nous avons fourni une documentation globale de l'algorithme de Boyer-Moore. Dans cette partie, une implémentation de bout en bout de cet algorithme sera faite.

3.1 Prétraitement

3.1.1 Table des mauvaises correspondances

La table des mauvaises correspondances stocke la dernière occurrence de chaque caractère dans le motif.

```
def preprocess_bad_character(pattern):  
    # Initialisation de la table de mauvaises correspondances avec -1 pour chaque caractère ASCII  
    bad_char_table = [-1] * 256  
    # Parcourir le motif pour remplir la table avec les dernières occurrences des caractères  
    for i in range(len(pattern)):  
        bad_char_table[ord(pattern[i])] = i  
    return bad_char_table
```

Figure 1 : fonction pour la table des mauvaises correspondances

3.1.2 Table des bons suffixes

La table des bons suffixes utilise les positions des suffixes dans le motif pour déterminer les décalages.

```

def preprocess_good_suffix(pattern):
    m = len(pattern)
    # Initialisation de la table des bons suffixes avec des zéros
    good_suffix_table = [0] * (m + 1)
    last_prefix_position = m

    # Remplir la table pour les bons suffixes qui sont aussi des préfixes
    for i in range(m - 1, -1, -1):
        if is_prefix(pattern, i + 1):
            last_prefix_position = i + 1
            good_suffix_table[m - i] = last_prefix_position - i + m - 1

    # Remplir la table pour les autres bons suffixes
    for i in range(m):
        slen = suffix_length(pattern, i)
        good_suffix_table[slen] = m - 1 - i + slen

    return good_suffix_table

def is_prefix(pattern, p):
    m = len(pattern)
    # Vérifier si la partie du motif à partir de p est un préfixe du motif entier
    for i in range(p, m):
        if pattern[i] != pattern[i - p]:
            return False
    return True

def suffix_length(pattern, p):
    m = len(pattern)
    length = 0
    i = p
    j = m - 1
    # Calculer la longueur du suffixe qui correspond au préfixe
    while i >= 0 and pattern[i] == pattern[j]:
        length += 1
        i -= 1
        j -= 1
    return length

```

Figure 2 : fonction pour la table des bons correspondances

3.2 Recherche

Avec les tables de mauvaise correspondance et de bons suffixes, nous pouvons maintenant implémenter l'algorithme de Boyer-Moore.

```

def boyer_moore(text, pattern):
    n = len(text)
    m = len(pattern)
    if m == 0:
        return []

    # Prétraitement pour obtenir les tables de mauvaises correspondances et de bons suffixes
    bad_char_table = preprocess_bad_character(pattern)
    good_suffix_table = preprocess_good_suffix(pattern)

    s = 0 # Position du motif par rapport au texte
    matches = []

    while s <= n - m:
        j = m - 1

        # Comparer le motif avec le texte de droite à gauche
        while j >= 0 and pattern[j] == text[s + j]:
            j -= 1

        # Si le motif est trouvé
        if j < 0:
            matches.append(s)
            s += good_suffix_table[0] # Décalage pour la prochaine comparaison
        else:
            # Utiliser la table de mauvaises correspondances et de bons suffixes pour déterminer le décalage
            s += max(good_suffix_table[j + 1], j - bad_char_table[ord(text[s + j])])

    return matches

```

Figure 3 : Recherche

4 PARTIE III : Implémentation de l'application

Django est un framework web open-source de haut niveau, écrit en Python, qui encourage le développement rapide et une conception propre et pragmatique. Développé initialement pour des sites de presse à forte charge de travail, Django offre une multitude de fonctionnalités intégrées, permettant aux développeurs de se concentrer sur la création de fonctionnalités plutôt que de réinventer la roue. Django a une architecture en 3 couches permettant de découper un projet en plusieurs briques facilitant le développement et la maintenabilité.

4.1 Création du projet Django

pip install django

django-admin startproject TpAlgo

4.2 Ajout de l'application AlgoApp

Cd TpAlgo

```
python manage.py startapp AlgoApp
```

4.3 Configuration de l'application

```
INSTALLED_APPS = [  
    # autres applications Django  
    'AlgoApp',  
]
```

4.4 Configuration de l'application dans urls.py du projet :

```
from django.contrib import admin  
from django.urls import path, include  
  
urlpatterns = [  
    path('admin/', admin.site.urls),  
    path('', include('AlgoApp.urls')),  
]
```

Figure 4 : Configuration de l'application dans urls.py du projet

4.5 Création de urls.py dans le répertoire de l'application AlgoApp :

```
from django.urls import path  
from . import views  
  
urlpatterns = [  
    path('', views.index, name='index'),  
    path('compare/', views.compare_documents, name='compare_documents'),  
]
```

Figure 5 : Création de urls.py dans le répertoire de l'application AlgoApp

Les différents captures vous seront présentés dans la partie utilisation

5 PARTIE IV : Utilisation de l'application

Pour ce qui est de l'utilisation de l'application, il faut d'abord l'installer puis l'exécuter.

5.1 Installation

L'application a été développée en utilisant l'outil git et GitHub pour la gestion des versions. Ainsi pour l'installer il faut taper la commande suivante :

```
git clone https://github.com/AbdallahBAMOGO/TpAlgoTexte/tree/main/Tp\_algo
```

Après l'installation il faut se diriger dans le répertoire du projet à partir de la commande cd. Puis vient maintenant l'installation des dépendances. Ces dernières ont été installées pour pouvoir lire les contenus des différents types de fichiers supportés qui sont le PDF, le txt, le docx. Pour installer ces dépendances il faut taper les lignes suivantes dans l'invite de commande (cela nécessite d'être en mode administration ou d'avoir certains privilèges) :

```
pip install python-docx PyPDF2
```

Maintenant on peut passer à l'exécution.

5.2 Exécution

Pour exécuter l'application, après s'être dirigé dans le répertoire de l'application, on tape la commande suivante pour lancer le serveur :

```
python manage.py runserver
```

On pourra maintenant accéder aux différentes pages (page accueil et page résultat) de l'application.

D'abord on est dirigé sur la page index.html qui contient le formulaire pour uploader les deux fichiers à comparer :

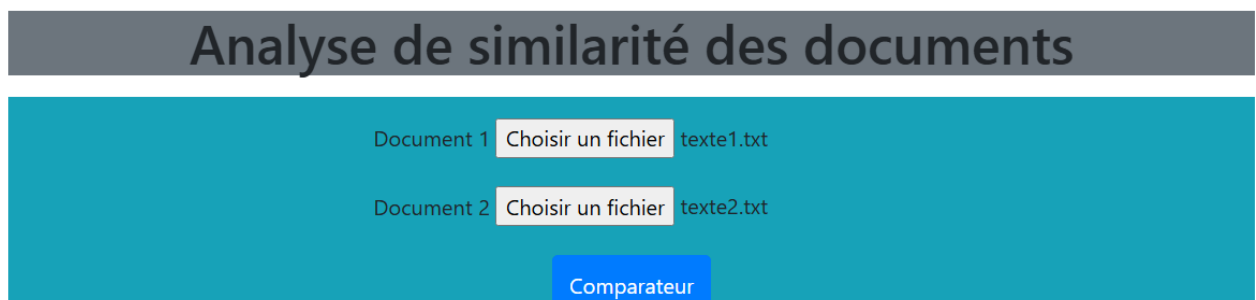


Figure 6 : Ecran de chargement des fichiers

Après avoir insérer les fichiers, on clique ensuite sur le bouton comparateur pour les soumettre pour analyse. Après analyse le résultat sera affiché sur la page result.html sous la forme :



Figure 7 : Resultat de similarité

6 Conclusion

L'algorithme de Boyer-Moore est une méthode de recherche de sous-chaîne extrêmement efficace, connue pour sa capacité à traiter rapidement de grandes quantités de texte en minimisant le nombre de comparaisons nécessaires. Son efficacité repose sur deux heuristiques principales : la heuristique du mauvais caractère et la heuristique des bons suffixes. Ces heuristiques permettent à l'algorithme de sauter intelligemment des portions du texte, ce qui le rend souvent plus rapide que les approches naïves ou traditionnelles de recherche de sous-chaîne.