Sorbonne Université

Rapport Final

16 Novembre 2022

Projet de détection de plagiat flagrant

Auteurs:
Amaury Curiel
Daniel Sima

Encadrants:
Antoine Genitrini
Emmanuel Chailloux

Table des matières

1 Présentation

L'objectif du projet consiste en la récupération de la plus longue sous-chaîne commune (ou l'une des plus longues si plusieurs) de deux chaînes de caractères. Cette récupération se fera a partir de plusieurs structures de données et méthodes algorithmiques comme la programmation dynamique, l'arbre des suffixes des lettres ou indices de ces lettres, compressé ou non.
Un second objectif sera de voir quelle méthode parmi celles cités précédemment sera la plus adaptée en fonction des données (les deux chaînes) en entrée.

2 Programmation dynamique

Question 2.1
- Une sous-structure optimale permettant de calculer la longueur des plus longs préfixes communs des suffixes de C1 et C2 est un tableau de tableaux, plus exactement une matrice $n*m$, avec n la taille de la chaîne C1 et m la taille de la chaîne C2.

Question 2.2
- Pour calculer la longueur des sous-chaînes les plus longues communes à C1 et C2, en fonction de la matrice choisie, on procède de manière itérative ligne par ligne (i.e. de gauche vers la droite pour chaque ligne).
Soit M cette matrice et M[i][j] la valeur de la plus longue sous-chaîne commune à C1 et C2 aux indices C1[i] et C2[j] dans cette structure. Si i=0 ou j=0, l'une des chaînes est de taille nulle et donc la plus longue longueur des sous-chaînes est nulle.
La valeur de la case M[i][j] (i.e. la plus longue longueur des sous-chaînes communes à C1 et C2 aux indices i et j) vaut M[i-1][j-1]+1 (i.e la plus longue longueur des sous-chaînes communes à C1 et C2 aux indices i-1, j-1) si C1[i] = C2[j].
Dans le cas ou C1[i] != C2[j] alors M[i][j] est remis à 0.
A la sortie de l'algorithme, on récupère la matrice complétée, la longueur de la sous chaîne commune ainsi que les coordonnées de fin de la sous chaîne commune

	A	N	A	N	A	S
B	0	0	0	0	0	0
A	1	0	1	0	1	0
N	0	2	0	2	0	0
A	1	0	3	0	3	0
N	0	2	0	4	0	0
E	0	0	0	0	0	0

Figure 1: Exemple pour "ANANAS" et "BANANE"

- Sachant que les indices de C1 correspond aux colonne et ceux de C2 aux lignes, on a l'indice maximal = 4 qui correspond a la case C1[4] = 4 et C2[5] = 4.

Question 2.3

$$\text{M[i][j]} = \begin{cases} 0 & \text{si i=0 ou j=0} \\ 0 & \text{si C1[i] != C1[j]} \\ M[i-1]M[j-1]+1 & \text{si C1[i] = C2[j]} \end{cases}$$

Figure 2: Équation de caractérisation de la sous-structure

Question 2.4
Afin de retrouver la sous chaîne commune à C1 C2, nous avons fait en sorte que l'algorithme renvoie le numéro de la ligne et de la colonne de l'indice le plus grand dans la matrice.Pour récupérer la sous chaîne commune, il suffit d'extraire la sous chaîne C1[col-taille: col]. (Quitte a échanger, on pose C1 comme étant le mot de taille maximale).

Question 2.5
On obtient une complexité au pire cas en $O(n*m)$ en utilisant comme mesure le nombre de comparaisons. En effet, dans la matrice $n*m$ on compare chaque caractère de la chaîne C1 avec chaque caractère de la chaîne C2.
A noter que la fonction *recupsub* qui permet de récupérer la plus longue sous-chaîne commune a une complexité au pire cas en nombre d'accès aux éléments $M[i][j]$ en $O(n) < O(n*m)$ ou n est la taille du mot le plus petit et qui vient s'ajouter a cette dernière.
On gardera donc pour la complexité totale $O(n*m)$.

Question 2.6
Voir le code source qui se trouve dans le dossier *Sources/Programmation_dynamique/source.ml* et lire le *README.txt* pour savoir comment exécuter le banc de tests.
Parmi les tests on trouve:
- "ANANAS" "BANANE" ce qui doit nous retourner "ANAN".
- "ANANAS" "ANANASSA" ce qui doit nous retourner "ANANAS".
- "BANANE" "BANANEBANANE" ce qui doit nous retourner "BANANE".

3 Arbre des suffixes

Question 3.7
En utilisant l'arbre des suffixes de C1 nous avons tous les sous-chaînes de C1. Il suffit de vérifier que C2 fait partie du chemin des sous-chaînes de C1 dans son arbre de suffixes.

Question 3.8
Nous avons choisit d'utiliser pour cette partie comme structure de données un arbre de type enregistrement (record) qui regroupe le *label* du noeud ainsi qu'une liste de fils $fils$. Ces champs sont mutables, i.e. modifiables par affectation.

```
type arbre = {
    mutable label: string;
    mutable fils: arbre list;
}
```

Figure 3: Structure de données de type arbre

Les primitives dont nous avons eu besoin sont les suivantes:
- **creer_fil** chaine: Permettant de créer un arbre fil dont chaque noeud possède qu'un seul noeud dans sa liste de fils. Cette primitive est nécessaire lorsqu'on veut ajouter la sous-chaîne ou le reste de sous-chaîne qui n'est plus commune a une sous-chaîne dans l'arbre des suffixes.
- **getListLabel** fils: Permettant d'obtenir la liste de tous les fils sous forme de chaîne de caractères pour pouvoir être écrite dans le fichier "$arbre\_suffixes.dot$" afin d'être affichés sous forme graphique.
- **parcours_arbres_suffixes** arbre: Permettant de parcourir l'arbre des suffixes et d'écrire dans le fichier "$arbre\_suffixes.dot$" .
- **tri_rapide** liste: Permettant de trier la liste de noeuds par leur label en utilisant le tri rapide classique.
- **find_perso** arbreList eti i: Permettant de trouver le noeud qui a le label eti.
- **ajout_frere** arbre chaine i: Permettant d'ajouter le suffixe *chaine* à *arbre* tout en tenant compte des noeuds communs (i.e. au dernier fils commun s'il en existe un).

A noter que nous utilisons les caractères "_" pour "#" et "§" pour la racine, car se sont des caractères acceptés par GraphViz.

Question 3.9
Voir le code de la fonction **arbre_suffixes** chaine, dans le dossier *Sources/Arbre_suffixe/source.ml*, dont le pseudo-code est le suivant:

Algorithm 1 Pseudo-code ArbreSuffixes

Entrée(s) la chaîne C1
  racine prend l'arbre vide ayant comme label "§".
  pour tout suffixe SSC de $C1$ **faire**
    si le suffixe SCC n'est pas présent dans l'arbre de racine "§" **alors**
      l'ajouter comme l'arbre fil dans les fils de la racine "§".
    sinon
      parcourir les noeuds jusqu'à tomber sur une différence entre le label du noeud et la i-ème lettre de $C1$ et ajouter aux fils du père de ce noeud l'arbre fil restant de SSC
    fin du si
  fin du pour
Sortie(s) *racine* //Arbre des suffixes de la chaîne $C1$

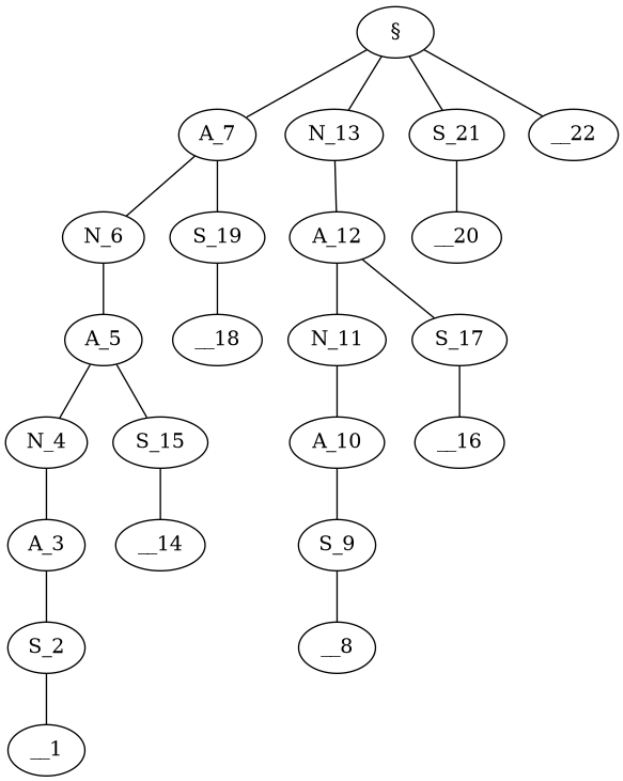


Figure 4: Génération du graphe.dot d'après la fonction **arbre_suffixes** "ANANAS_"

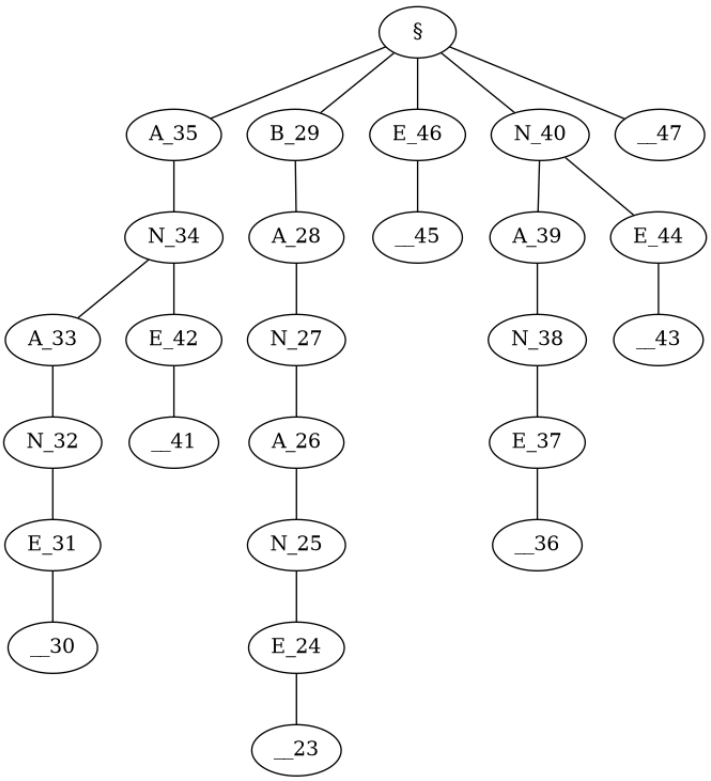


Figure 5: Génération du graphe.dot d'après la fonction **arbre_suffixes** "BANANE_"

A noter a nouveau que tous les noeuds possedent un identifiant lettre_**nombre** (e.g. A_**35**) pour differencier les noeuds dans le fichier .dot et que le caractere de fin est remplacé par "_" au lieu de "#" (donc pour les noeuds qui ont le caractere de fin, on retrouvera deux fois "_" e.g. "__23")

Question 3.10
Voir le code des fonctions **aux_sous_chaine** arbre chaine2 i, et **sous_chaine** chaine1 chaine2, dans le dossier $Sources/Arbre\_suffixe/source.ml$, dont leur pseudo-code est le suivant:

Algorithm 2 Pseudo-code SousChaine

Entrée(s) chaînes $C1$ $C2$
  création de l'arbre des suffixes *arbre* pour $C1$
  pour tout lettre de $C2$, du début vers la fin **faire**
    si la i-ème lettre de $C2$ n'existe pas parmi l'un des chemins empruntes pour vérifier l'existence de $C2$ dans les suffixes de $C1$ dans l'*arbre* **alors**
      false // $C2$ n'est pas une sous-chaîne de $C1$
    sinon
      la i-ème lettre existe, et on passe a la (i+1)-ème lettre et au prochain noeud
    fin du si
  fin du pour
Sortie(s) **true** // si on n'est pas tombé dans le cas false

Question 3.11
Voir dans la question 3.9 les tests pour l'arbre des suffixes avec "ANANAS_" et "BANANE_". Et un troisième test ci-dessous:

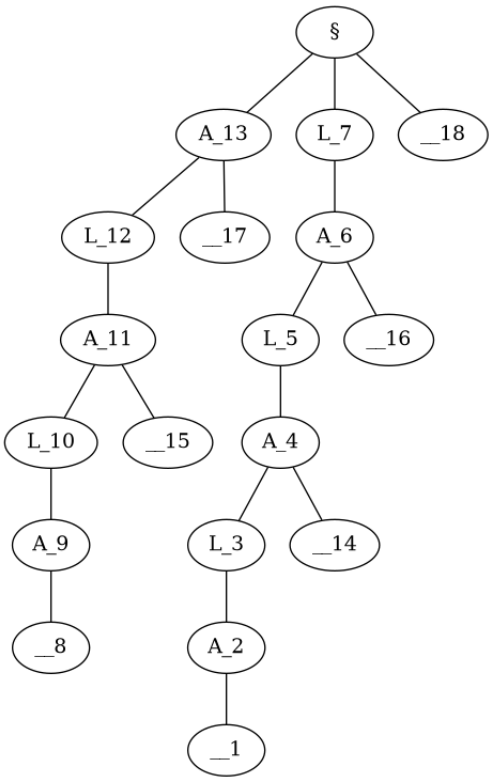


Figure 6: Génération du graphe.dot d'après la fonction **arbre_suffixes** "LALALA_"

Pour les tests de sous chaîne, nous pouvons tester avec:
- **sous_chaine** "BANANE_" "NAN" ce qui doit nous dire que "NAN" est une sous-chaîne de "BANANE_".
- **sous_chaine** "ANANAS_" "BANANE_" ce qui doit nous dire que "BANANE_" n'est pas une sous-chaîne de "ANANAS_".
- **sous_chaine** "ANANASANAANS_" "ANANAS_" ce qui doit nous dire que "ANANAS_" est une sous-chaîne de "ANANASANANAS_".

Pour tester d'autres combinaisons de tests pour l'arbre des suffixes et sous chaîne, voir le code source qui se trouve dans le dossier $Sources/Arbres\_suffixes$ et lire le $README.txt$ pour savoir comment exécuter le banc de tests.
- A noter que si vous possédez .dot alors les graphes des arbres des suffixes seront directement génères dans le dossier $Sources/Arbres\_suffixes/Graphes$. Sinon prendre le contenu de $Sources/Arbres\_suffixes/arbre\_suffixes.dot$ et l'ajouter sur le site de Graphviz.
- Vérifier que vous prenez le contenu qui vous intéresse puisque le fichier $arbre\_suffixes.dot$ est écrasé a chaque fois.

Question 3.12
Pour la fonction **arbre_suffixe**:
- Pour la complexité temporelle au pire des cas nous allons compter le nombre de comparaisons. Il existe n suffixes dans un mot (n = taille nu mot). Au pire des cas a chaque ajout d'un nouveau suffixe nous le comparons a tous les suffixes déjà présents, i.e. (n-1). De plus un tri rapide classique en $O(n.log(n))$ est effectue à chaque ajout pour avoir une liste de fils ordonnés alphabétiquement, donc comme nous faisons n ajouts de suffixes, la complexité temporelle totale au pire des cas est en $O(n * (n + n.log(n)) = O(n^2.log(n))$.
- Pour la complexité en espace, elle est quadratique $O(n^2)$ car au pire nous stockons tout les suffixes (de taille 1 + 2 + 3 + ... + n).

Pour la fonction **sous_chaine**:
- La complexité temporelle au pire des cas est celle de **arbre_suffixe**, i.e. $O(n^2.log(n))$, pour la création de l'arbre de la première chaîne + $O(n^2)$ en nombre de comparaison au pire des cas pour vérifier que chaque lettre de la deuxième chaîne suit un chemin de la première chaîne, car nous avons au pire des cas (n-1) noeuds présents à comparer. Donc on a une complexité temporelle totale en $O(n^2.log(n) + n^2) = O(n^2.log(n))$.
- La complexité en espace est celle de **arbre_suffixe**, i.e. $O(n^2)$ puisque nous avons le même arbre des suffixes.

Question 3.13
Pour cette question nous avons enrichi notre structure de données précédente pour pouvoir accéder au père des noeuds avec *pere* et manipuler un compteur *cpt* pour connaître la plus longue sous chaîne commune. Le choix d'avoir une liste de père qui peut être vite se justifie pour la racine puisqu'elle n'a pas de père.

```
type richArbre = {
        mutable labelR: string;
        mutable filsR: richArbre list;
        mutable pere: richArbre list;
        mutable cpt: int;
 }
```

Figure 7: Structure de données enrichie de type richArbre

Voir le code de **buildPlusGrandeSequence** arbre acc, et **buildArbre_suffixes** ch1 ch2 dont le pseudo-code est le suivant:

Algorithm 3 Pseudo-code SousChainesCommunes

Entrée(s) chaînes $C1$ $C2$

création de l'arbre des suffixes *arbre* pour C1

pour tout fils de $C2$ **faire**

ajouter le fils dans l'*arbre* de $C1$ tout en propageant le compteur *cpt* des sous-chaînes de $C2$

fin du pour

recherche du noeud ayant le compteur *cpt* maximum parmi les fils de la racine de l'*arbre*

parcours du chemin des noeuds ayant ce compteur *cpt* maximal et concaténation du label de ces noeuds

Sortie(s) SSC //Plus longue sous-chaîne commune de $C1$ et $C2$ suite a la concaténation des labels ayant le compteur maximal

Pour les tests de sous chaîne commune, nous pouvons tester avec:
- **buildArbre_suffixes** "ANANAS_" "BANANE_" ce qui doit nous retourner "ANAN".
- **buildArbre_suffixes** "ANANABBBBBS_" "BANANEBBBBB_" ce qui doit nous retourner "BBBBB".
- **buildArbre_suffixes** "ANANAS_ " "ANANASANANAS_" ce qui doit nous retourner "ANANAS_".

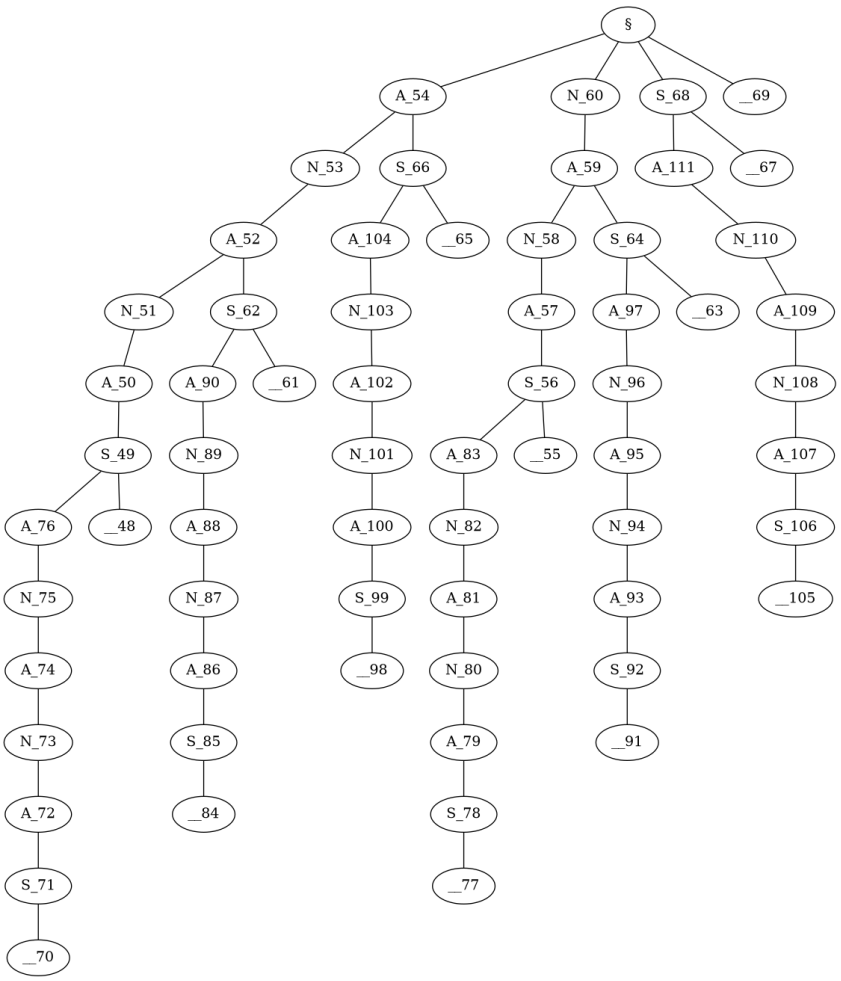


Figure 8: Exemple d'arbre des suffixes non compressé crée avec **buildArbre_suffixes** "ANANAS_ " "ANANASANANAS_"

Pour tester d'autres combinaisons de tests pour la plus longue sous chaîne commune, voir le code source qui se trouve dans le dossier $Sources/Arbres\_suffixes$ et lire le $README.txt$ pour savoir comment exécuter le banc de tests.
- A noter que si vous possédez .dot alors les graphes des arbres des suffixes des deux chaînes seront directement génères dans le dossier $Sources/Arbres\_suffixes/Graphes$. Sinon prendre le contenu de $Sources/Arbres\_suffixes/arbre\_suffixes.dot$ et l'ajouter sur le site de Graphviz.
- Vérifier que vous prenez le contenu qui vous intéresse puisque le fichier $arbre\_suffixes.dot$ est écrasé a chaque fois.

Question 3.14
Nous avons modifié la structure de données pour rajouter en chaque noeud un pointeur vers son père et un compteur. Ceci permet la propagation du compteur vers les noeuds de la racine et de connaître le chemin à suivre pour récupérer la plus grande séquence commune.
- Étant les seules modification, nous stockons principalement les mêmes informations à deux constantes prés, nous avons donc la même complexité en espace au pire des cas que la question 3.12, i.e. $O(n^2)$.
- La construction de l'arbre des suffixes des deux chaînes de caractères se fait comme précédemment, en une complexité temporelle en $O(n^2.log(n)))$. A ceci s'ajoute la complexité du parcours de l'arbre pour récupérer la plus longue séquence commune qui se fait en $O(n^2)$ en nombre de comparaisons dans le pire des cas car nous pouvons comparer a tous les suffixes présents (n-1). Donc nous avons une complexité temporelle totale dans le pire des cas en $O(n^2.log(n)) + n^2) = O(n^2.log(n))$.

4 Compression de l'arbre des suffixes

Question 4.15
Un arbre que non compressé contient énormément de nœuds et à une taille conséquente, par exemple: l'arbre des suffixes de l'alphabet: 351 nœuds pour 26 caractères initiaux!
Solution: Fusionner les fils uniques avec leur père.
Conséquence: ||Arbre des suffixes de l'alphabet|| : 351 —> 26 noeuds
Voir le code récursif de la fonction **compression1** arbre, dans le dossier $Sources/Arbre\_suffixe/source.ml$, dont le pseudo-code est le suivant:

Algorithm 4 Pseudo-code compression

Entrée(s) arbre de suffixes non compressé A
  pour tout noeud N de A **faire**
    si N n'a qu'un seul fils **alors**
      on crée un nouveau noeud ayant la concaténation des labels du noeud N et de son fils, ce nouveau noeud récupérera les fils du fils et le père du noeud N concerné.
    fin du si
  fin du pour
Sortie(s) A //Arbre A compressé récursivement

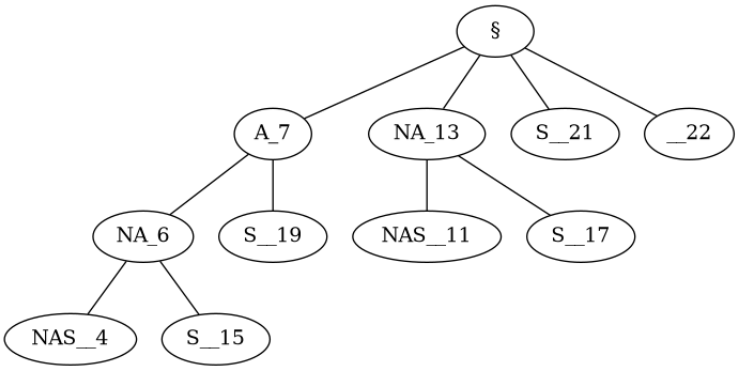


Figure 9: Exemple d'arbre des suffixes compressé avec **compression1** "ANANAS_ "

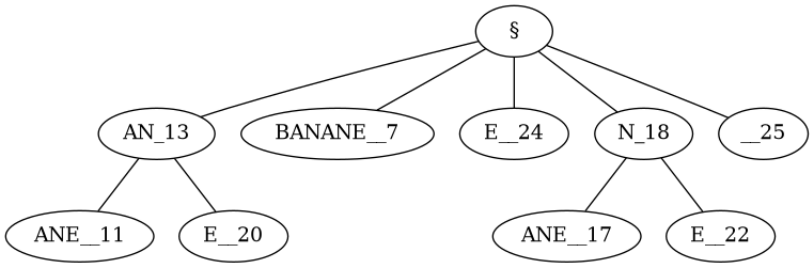


Figure 10: Exemple d'arbre des suffixes compressé avec **compression1** "BANANE_ "

Question 4.16
On a ajouter le symbole "#" (dans notre cas pour l'affichage "_") car si on souhaite compresser l'arbre des suffixes de 2 chaînes C1 C2 et que C1 est un préfixe de C2, on aura un noeud contenant le suffixe de C1 concaténé avec la fin de C2. Ce qui rendrais impossible le fait de récupérer la plus grande séquence.

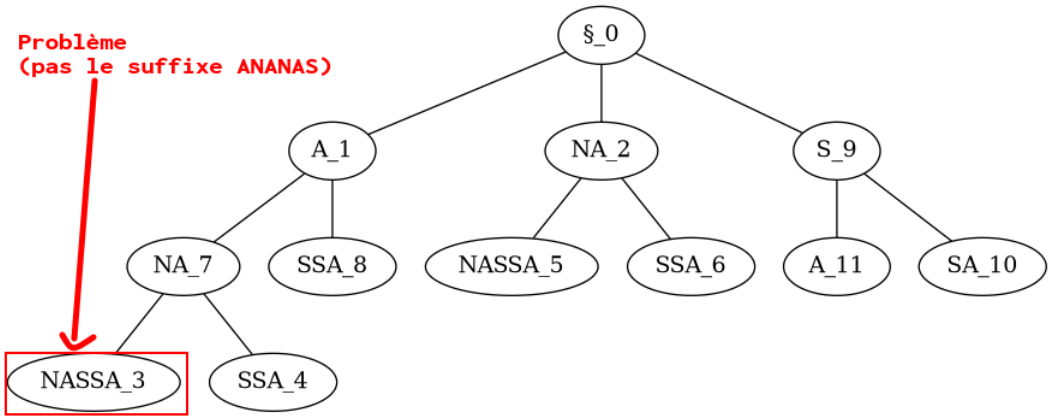


Figure 11: Exemple de compression de "ANANASSA" "ANANAS" sans caractère spécial de fin

- Nous retrouvons un noeud "NASSA" qui ne permet pas de trouver le suffixe "ANANAS".

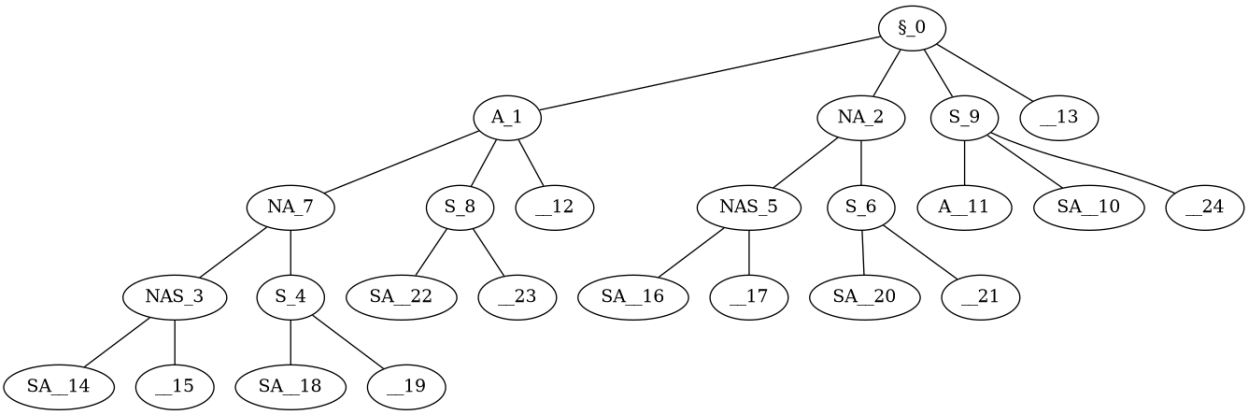


Figure 12: Exemple de compression de "ANANASSA_" "ANANAS_" avec caractère spécial de fin "_"

- Nous retrouvons ici le suffixe "ANANS" qui manquait grâce au caractère spécial de fin, d'où son importance.

Question 4.17
Voir le code de la fonction **buildPlusGrandeSequence2** node acc, dans le dossier $Sources/Compression\_arbre\_suffixe/source.ml$, dont le pseudo-code est similaire au pseudo-code pour SousChaineCommune pour les arbre non compressés (Algorithm 3) a la différence près que lors de la fusion des noeuds on garde le compteur propagé qui sera parcourut a la fin pour récupérer la sous-chaîne commune la plus longue.

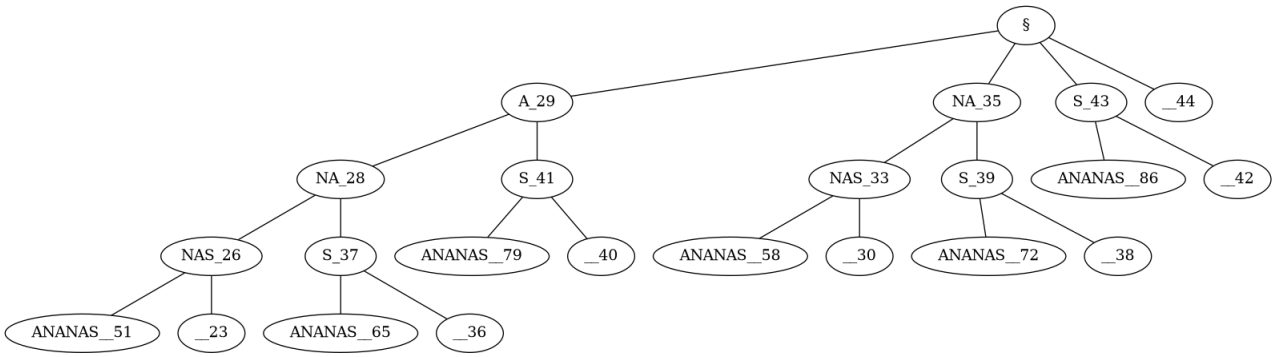


Figure 13: Arbre des suffixes compressées crée pour récupérer la plus longue sous-chaîne commune de "ANANAS""$ANANASANANAS$"

Question 4.18
Le pire des cas apparaît dans un mot ou il est impossible de compresser, i.e. un mot dans lequel chaque suffixe est un préfixe dans lequel nous ajoutons une lettre. Dans ce cas, nous avons n^2 noeuds si n est la taille du mot. L'arbre des suffixes compressés ressemblerait à un arbre peigne.

Question 4.19
A partir de cette question nous avons a nouveau et pour la dernière fois enrichi notre

structure de données précédente. Nous avons ajouté un identifiant *id* pour chaque arbre dans la structure pour nous permettre de plus facilement gérer l'affichage sans intégrer cet identifiant dans le label des arbres comme jusqu'à maintenant.

```
type idArbre = {
        mutable labelI: string;
        mutable filsI: idArbre list;
        mutable id: int;
        mutable pereI: idArbre list;
        mutable cptI: int;
}
```

Figure 14: Structure de données enrichie de type idArbre

Voir le code de la fonction **idArbre_suffixes** chaîne, qui utilise la même technique que la fonction créant l'arbre des suffixes non compressés avec les lettres sauf que nous ajoutons une table de hachage qui nous offre la correspondance entre les indices et les lettres des mots. Le pseudo-code de cette fonction est principalement le même que celui de l'Algorithm 1.

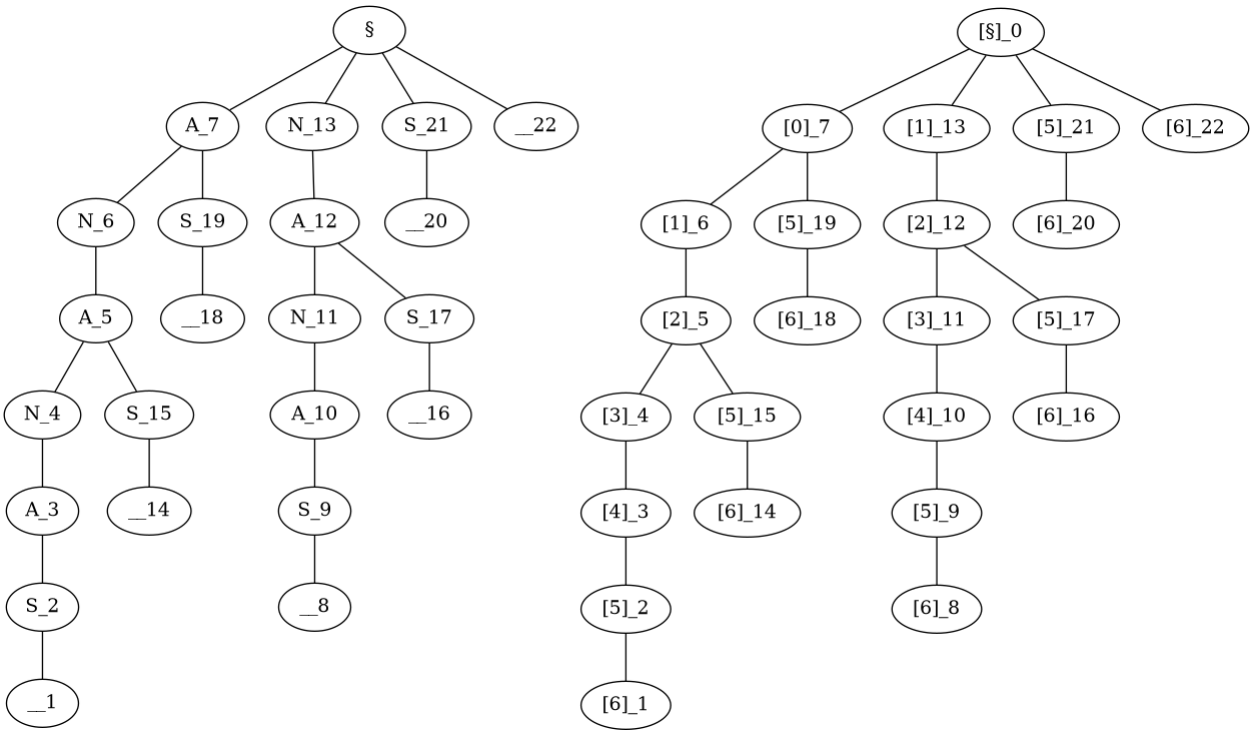


Figure 15: Arbre des suffixes non compressés de "ANANAS_" avec lettres et indices

Voir le code de la fonction **compression_idArbre_numeros** idArbre, pour la compression de l'arbre des suffixes avec des indices. Comme précédemment le pseudo-code de cette fonction est principalement le même que celui de l'Algorithm 4, a la différence près que nous utilisons une table de hachage pour vérifier l'égalité des occurrences des lettres.

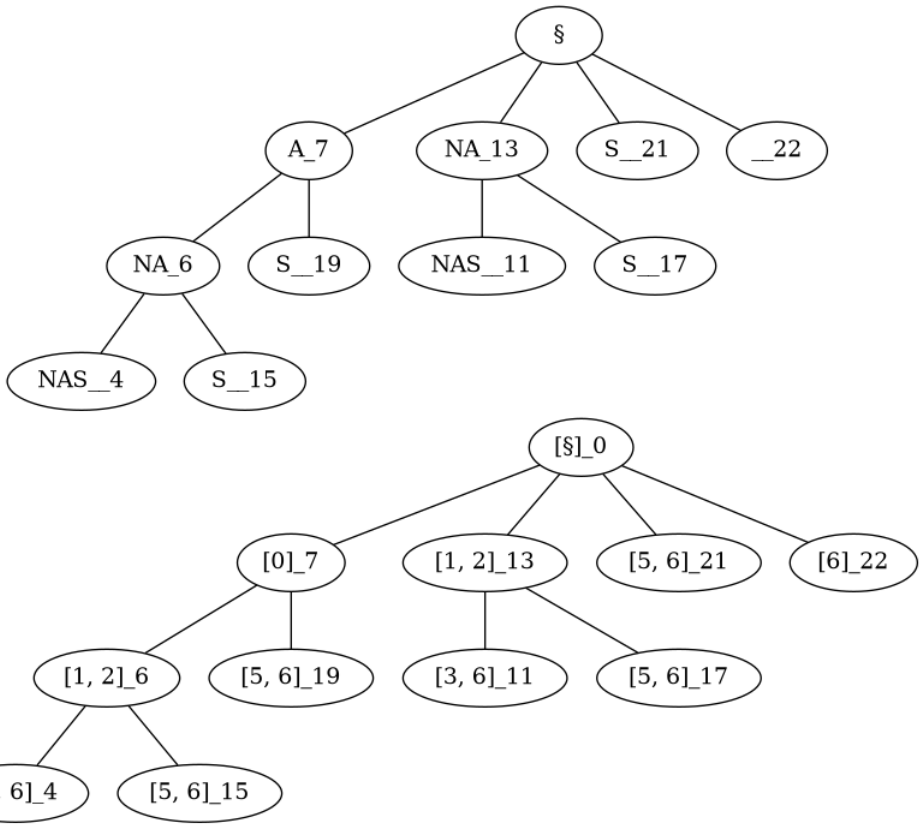


Figure 16: Arbre des suffixes compressés de "ANANAS_" avec lettres et indices

Question 4.20
La complexité est similaire au cas précédent, sauf que l'on passe une fois sur l'arbre afin de le compresser. Le nombre de noeud à visiter étant au pire de n carré, nous avons toujours une complexité quadratique $*$ complexité du tri rapide, donc $O(n^2.log(n))$. Cependant, nous avons gagner en espace (pas au pire).
Pour tester d'autres combinaisons de tests, voir le code source qui se trouve dans le dossier $Sources/Compression\_arbre\_suffixe/source.ml$ et lire le $README.txt$ pour savoir comment exécuter le banc de tests.

5 Construction efficace de l'arbre des suffixes compressé

Question 5.21
Arbre des suffixes compressés directement avec lettres:
- Voir le code de la fonction **arbre_suffixes_compresse** chaîne, dans le dossier $Sources/Compression\_directe\_arbre\_suffixes/source.ml$, dont le pseudo-code est le suivant:

Algorithm 5 Pseudo-code ArbreSuffixesCompresse

Entrée(s) chaîne C
  création de la racine R
  pour tout suffixe de C **faire**
    si le suffixe de C n'existe pas dans les fils de R **alors**
      ajout du suffixe complet de C parmi les fils de R
    sinon
      parcourir les noeuds du chemin jusqu'à tomber sur une différence entre les lettres du noeud concerné et la i-ème lettre du suffixe
      si différence trouvé **alors**
        création de deux noeuds, le premier prendra le label restant du suffixe a partir de la différence de lettres et le second prendra le label restant du noeud ou y a eu la différence de lettres et ce noeud fils prendra les fils de son père
      fin du si
    fin du si
  fin du pour
Sortie(s) A //Arbre des suffixes des lettres compressé directement

Arbre des suffixes compressés directement avec indices:
- Voir le code de la fonction **idArbre_suffixes_indices_compresse** chaîne, dans le dossier $Sources/Compression\_directe\_arbre\_suffixes/source.ml$, dont le pseudo-code est similaire a celui de la compression directe avec les lettres (Algorithm 5), a la différence près qu'il faut utiliser une table de hachage pour vérifier l'égalité des occurrences des lettres.

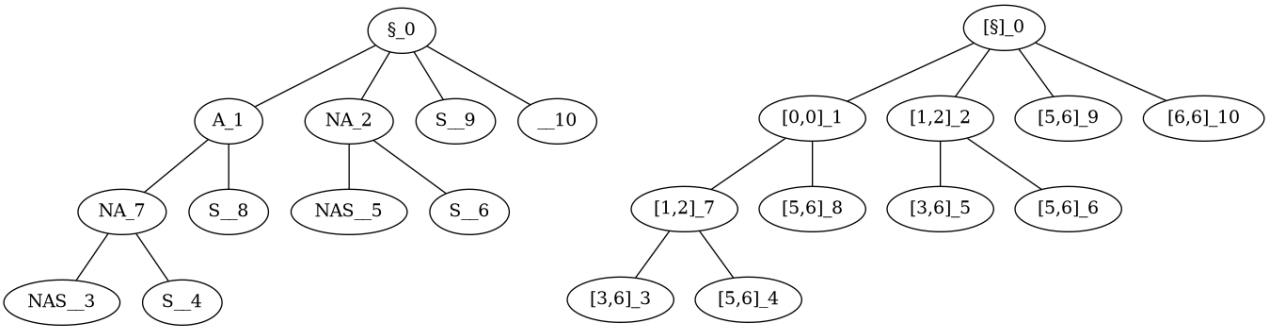


Figure 17: Arbre des suffixes compressé directement de "ANANAS_" avec lettres et indices

Récupération de la plus longue sous-chaîne commune dans arbre des suffixes compressés directement avec lettres:
- Voir le code des fonctions **arbre_suffixes_compresse_2** arbre chaîne, et **sous_chaine_commune_compresse_lettres** arbre, dans le dossier $Sources/Compression\_directe\_arbre\_suffixes/source.ml$, dont le pseudo-code est le suivant:

Algorithm 6 Pseudo-code SousChaineCommuneCompresse

Entrée(s) chaînes $C1$ $C2$
  création de l'arbre des suffixes compressés de $C1$
  pour tout suffixe de $C2$ **faire**
    si le suffixe de $C2$ n'existe pas dans les fils de R **alors**
      ajout du suffixe complet de $C2$ parmi les fils de R avec son compteur = -1 // -1 pour dire qu'il s'agit d'un suffixe de $C2$
    sinon
      itérer tant que les lettres des nœuds et du suffixe de $C2$ sont égales et augmenter un compteur passé par référence
      si différence trouvé **alors**
        créer deux fils, l'un qui prendra le reste du suffixe a partir de la différence et l'autre qui aura comme label l'étiquette restante du nœud ou il y avait la différence et aura comme fils les fils de son père + propager le compteur si compteur != -1
      sinon
        s'arrêter si le suffixe de $C2$ se termine + propager le compteur si compteur != -1.
      fin du si
    fin du si
  fin du pour
  parcours suivant le compteur maximal et concaténation des labels
Sortie(s) SSC //Plus longue sous-chaîne commune de $C1$ et $C2$

Récupération de la plus longue sous-chaîne commune dans arbre des suffixes compressés directement avec indices:
- Voir le code de la fonction **idArbre_suffixes_indices_compresse2** chaine1 chaine2, et **sous_chaine_commune_compresse_indices** arbre tableHash, dans le dossier *Sources/Compression_directe_arbre_suffixes/source.ml*, dont comme précédemment le pseudo-code est similaire a celui de récupération de la plus longue sous-chaîne commune pour les arbres des suffixes compresses avec lettres (Algorithm 6), a la différence près qu'il faut utiliser une table de hachage pour vérifier l'égalité des occurrences des lettres.

Question 5.22

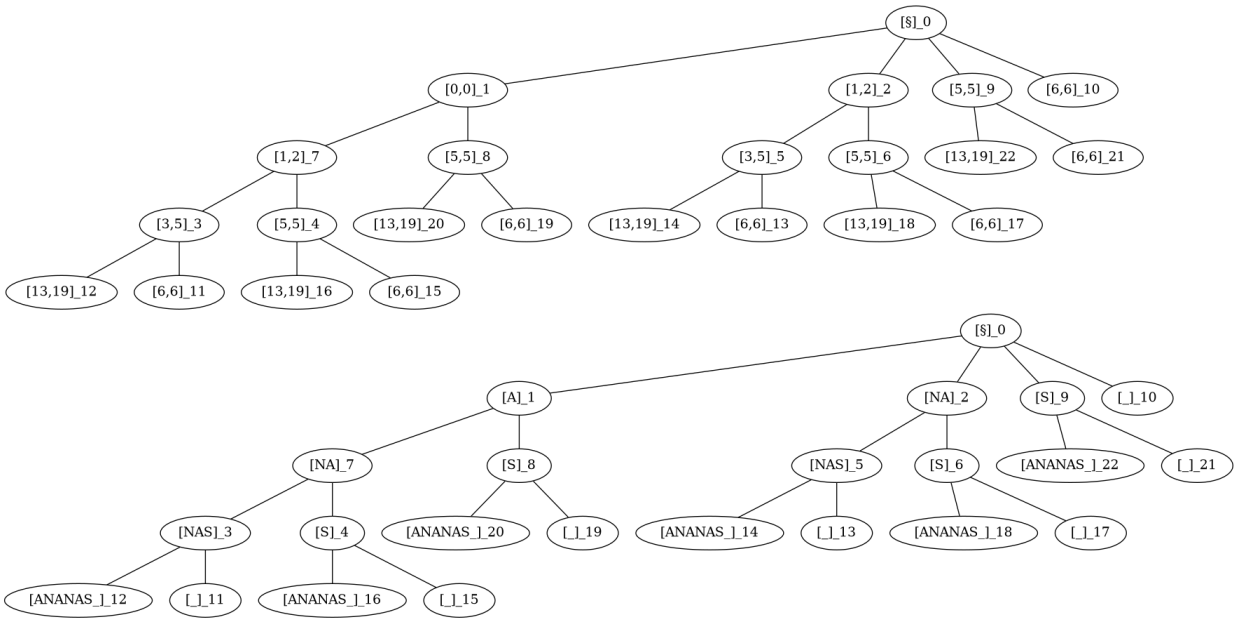


Figure 18: Arbre des suffixes compressé directement avec "ANANAS_" puis avec "ANANASANANAS_" avec lettres et indices pour retrouver la plus longue sous-chaîne commune

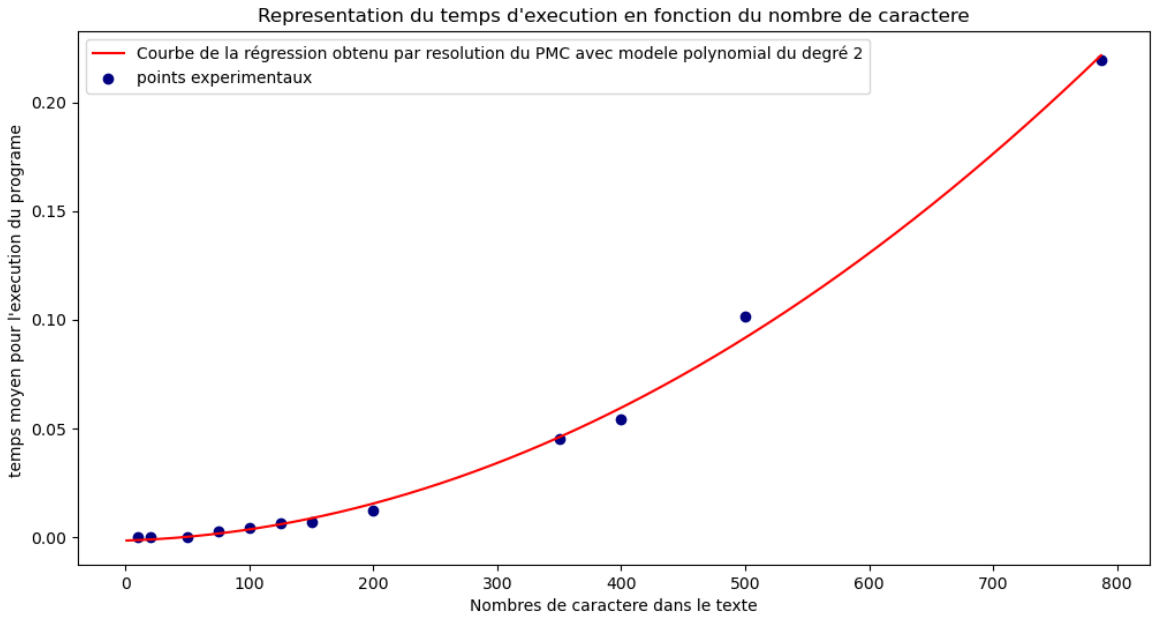
- Dans l'exemple ci-dessus, l'algorithme nous retourne comme plus longue sous-chaîne commune: "ANANAS_"
- Pour "ANANS_" et "BANANE_", l'algorithme nous retourne "ANAN".
- Pour "ANANASBBBBB_" et "BANANBBBBBE_", l'algorithme nous retourne "BBBB".
- Nous pouvons tester plusieurs combinaisons de mots dans le fichier $Sources/Compression\_directe\_arbre\_suffixes/source.ml$ et en lisant le fichier $README.txt$ qui explique comment fonctionnent les tests et comment les executer.

Question 5.23
La construction efficace de l'arbre compressé nous permet de gagner en complexité en moyenne mais pas au pire des cas. En effet, on ne nécessite pas de repasser en moyenne sur tout l'arbre car il est compressé. Bien que au pire des cas la complexité reste quadratique en $O(n^2.log(n))$ (nécessite de comparer chaque suffixes aux autres + tri rapide pour ordonner alphabétiquement).
Nous gagnons en espace pour les arbres compressés avec indices car nous stockons les indices de début et de fin au lieu de tout le suffixe comme pour les arbres avec lettres (complexité en $O(n^2)$).

6 Expérimentations

Question 6.24
Pour les 150 caractères de données0.txt et données1.txt, la plus longue chaîne commune est: "D'abord confinée dans les monastères et limitée".
- A noter que nous prenons tous les caractères en minuscule, donc les différences minuscule majuscule de la même lettre ne seront pas perceptible (comme pour donnee0.txt vs donnee2.txt ou il y a "Les" et "les").

Question 6.25
En résolvant un Problème des Moindre Carrés (PMC) avec un modèle polynomial de degré 2 sur les mesures que nous avons obtenus empiriquement, nous avons obtenu le graphique suivant:



En appliquant la même méthode avec un modèle linéaire, nous obtenons une courbe de régression trop éloignée de la réalité et avec un modèle polynomial de degré 3, nous ne gagnons pas grand chose sur le modèle de degré 2.
Nous avons donc en pratique une complexité quadratique, ce qui est cohérent avec notre analyse théorique.
- A noter que le temps est en secondes.

Question 6.26
En appliquant la même méthode que précédemment, nous avons obtenus les graphiques ci-dessous. Grâce a ces expérimentations, nous avons pu observer que plus les 2 documents comparés présentaient des similitudes, de plus le temps d'exécution en fonction du nombre de caractères semble être quadratique.
Nous obtenons donc la complexité attendue dans le pire cas.

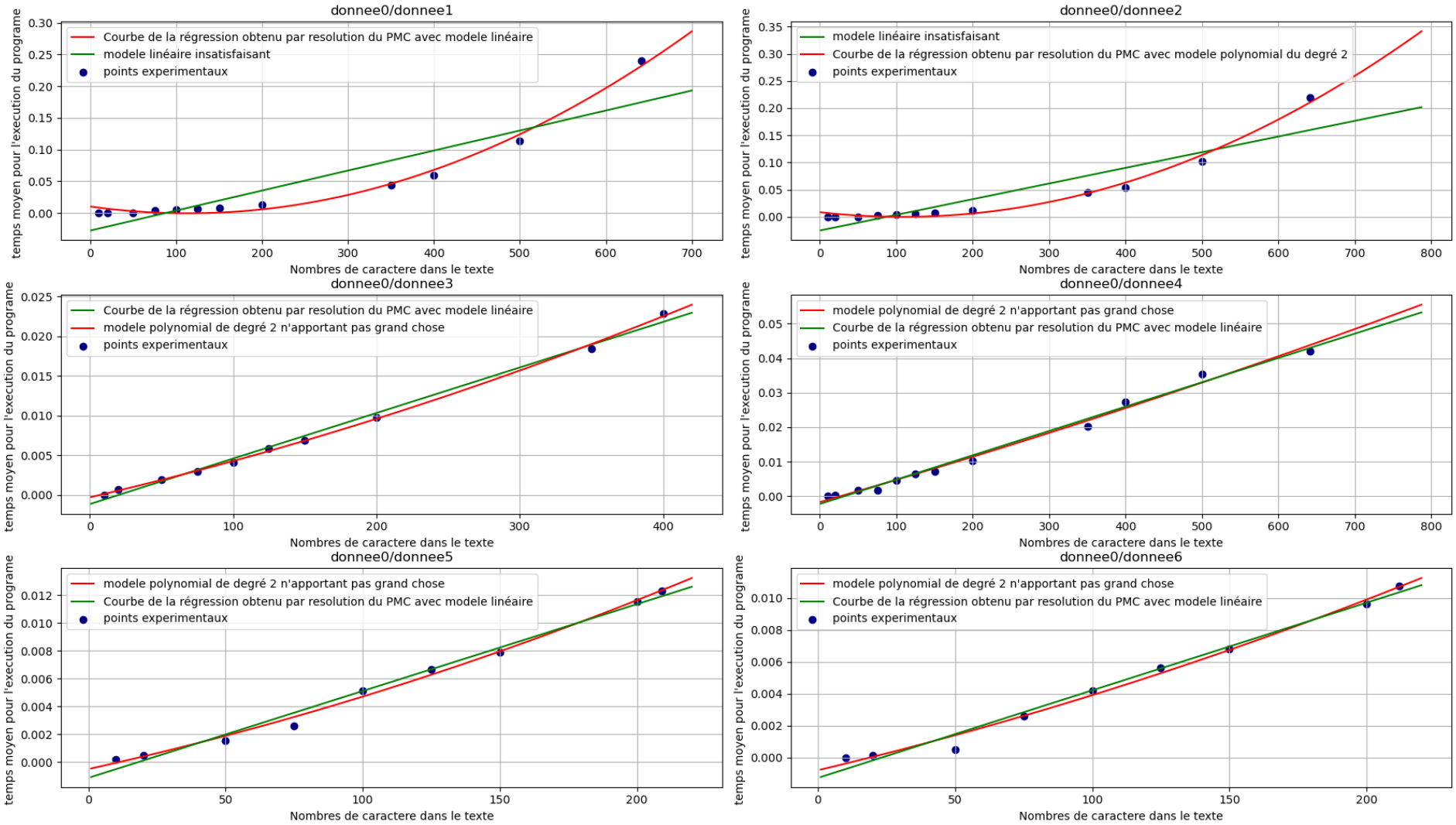


Figure 19: Graphique des temps d'execution du programme sur donnée0 comparé aux autres fichiers classés par ordre croissant de similitude