Compte rendu projet MU4IN511

Lucas Fumard – Carla Giuliani

Pour les jeux de test, voir les fichiers test_*.ml

Exercice 2 programmation dynamique

Les fichiers sont dans `ex2_dynamic/`*, les tests sont exécutés en lançant* `compile_and_run.sh`

2.1) La structure optimale est un tableau S à deux dimensions de taille $n*m$ où n est la taille du premier mot et m celle du second mot.

Ainsi, $S[i][j] \forall 0 \leq i < n, 0 \leq j < m$ représente la longueur du plus long préfixe commun de $C_{1,i}C_{1,i+1}...C_{1,n-1}$ et $C_{2,i}C_{2,i+1}...C_{2,n-1}$

2.2) Cette structure est initiée à 0. Pour remplir cette structure, il faut parcourir toutes les paires de suffixes et il faut incrémenter la case de 1 par rapport à la taille du préfixe le plus long commençant à la lettre d'après si les deux lettres lues sont les mêmes.
La solution est le maximum de cette structure.

2.3)C'est à dire $S[i][j] = \begin{cases} S[i+1][j+1] \; si \; C_{1,i} = C_{2,j} \; et \; i < n-1 \; et \; j < m-1 \\ 1 \; sinon \; si \; C_{1,i} = C_{2,j} \\ 0 \; sinon \end{cases}$

et Solution : $max(S)$

2.4) On ajoute l'information d'index de début dans le mot C_1

Ainsi, $S[i][j] = \begin{cases} (S[i+1][j+1], i) \; si \; C_{1,i} = C_{2,j} \; et \; i < n-1 \; et \; j < m-1 \\ (1, i) \; sinon \; si \; C_{1,i} = C_{2,j} \\ (0,0) \; sinon \end{cases}$

2.5) Pour remplir le tableau et trouver le maximum, il faut comparer toutes les paires de lettres des mots avec une complexité en comparaison en $\Theta(n*m)$.
Pour obtenir la solution, il suffit de lire le contenu de la variable `maxi` modifiée à chaque fois que nous calculons un nouveau maximum. Ainsi, la complexité dans tous les cas en comparaison de cet algorithme est de l'ordre de $\Theta(n*m)$

Exercice 3 arbre des suffixes

Les fichiers sont dans `ex3_arbre/`*, les tests sont exécutés en lançant* `compile_and_run.sh`

3.7) C_2 est une sous chaîne de C_1 si C_2 est un préfixe des mots présents dans l'arbre des suffixes des C_1

3.8) On défini `'a sufftree = ('a * 'a sufftree list * int)`, avec `'a` le type `char` (puis `string` et `int*int` par la suite) c'est à dire un nœud correspond à un triplet contenant une lettre (un mot ou des indices par la suite), une liste de nœud et un entier correspondant à la taille du plus grand mot que l'on peut former à partir de ce nœud.

3.12) Soit n la taille de la chaîne et α la taille de l'alphabet de la chaîne (le nombre de caractères distincts de la chaîne).
Complexité temps pire cas de `arbreSuffixes` en nombre de nœuds accédés, modifiés ou crées :
$C(creerSuffixes)=n$ car on parcours la chaîne du début à la fin pour créer une liste des suffixes.

$C(ajoutChaine(i))=\alpha*i$ où $i=|suffixe|$ car on parcours une liste de nœuds de taille maximale de l'alphabet pour trouver une bonne lettre et on descend en profondeur de au maximum i nœuds (jusqu'à la fin du suffixe).

Ainsi, $C(arbreSuffixes)=\sum_{i=1}^{n} ajoutChaine(i)=\alpha*\frac{n(n+1)}{2}=O(\alpha*n^2)$ car on ajoute chaque mot créé par `creerSuffixe` dans l'arbre avec `ajoutChaine`, $O(n^2)$ si on considère α borné par une constante.

Il y a au maximum $n+1$ (+1 pour #) suffixes, de taille 1 à n+1, donc la complexité en espace en nombre de nœuds est en $O(n^2)$.

Pour `sousChaine`, complexité pire cas en nombre de comparaisons :
Soit n la taille de s_1 et m la taille de s_2
(aux1) :

- on a une sous fonction qui produit une boucle qui parcours s_2 à partir de l'indice l
- durant cette boucle il y a 2 comparaisons
- l commence tout le temps à 0

=> complexité $(m-l)*2$

(aux2) :

- cette sous fonction ne fait qu'un tour de boucle et produit 3 comparaisons au maximum
- elle appelle aux1 avec l=0

=> complexité $3*C(aux1)=6*m$

(aux1) :

- cette sous fonction fait n tours de boucle (contrôlé par i) et fait 2 comparaison par tour
- elle appelle aux2 à chaque tour

=> complexité $2*n*C(aux2)=12*n*m$

ainsi, on a $C(sousChaine)=O(n*m)$

3.14) Complexité pire cas en nombre de comparaisons :

$C(sousChaineMot)=m^2$ car on crée tous les préfixes de tous les suffixes de s_2, au nombre de $\frac{m(m+1)}{2}-1$, qu'on arrondi à m^2.

$$C(sousChainesCommunes)=C(sousChaineMot)+\sum_{i=1}^{m^2}(C(sousChaine(n,i))+1+C(ajoutChaine(i)))+C(motArbre)$$

car dans la fonction `sousChainesCommunes`, on teste si chacun des préfixes des suffixes de s_2 appartient à s_1 avec `sousChaine` et on l'ajoute avec `ajoutChaine` si oui. Ensuite, on cherche le mot le plus long avec `motArbre`.

Or, $C(motArbre)=hauteur(arbre)*\alpha=min(n,m)*\alpha$ avec n le nombre de nœuds dans l'arbre, car on a ajouté une troisième donnée qui symbolise la plus longue chaîne possible à partir d'un nœud donné. Ainsi, on peut faire une recherche d'un mot le plus long en complexité linaire d'accès aux nœuds.

Donc $C(sousChainesCommunes)=m^2+\sum_{i=1}^{m^2}(n*i+1+\alpha*i)+\alpha\,min(n,m)$ (on substitue m par i dans la complexité de `sousChaine` car c'est la taille de la chaîne à vérifier)

$$C(sousChainesCommunes)=m^2+n\sum_{i=1}^{m^2}i+m+\alpha\sum_{i=1}^{m^2}i+\alpha\,min(n,m)$$

$$C(sousChainesCommunes)=m^2+n\frac{m^2(m^2+1)}{2}+m+\alpha\frac{m^2(m^2+1)}{2}+\alpha\,min(n,m)$$

Donc $O(sousChainesCommunes)=O(\alpha\,m^4+n*m^4)=O(n*m^4)$ avec $\alpha\leq n+m$ car c'est l'ensemble des caractères distincts dans s_1 et s_2 et α borné par un nombre fini.

Cette complexité est trop élevée et des optimisations seront faites dans le cadre d'un arbre compressé, notamment en utilisant des indices pour ajouter seulement un préfixe d'un suffixe dans le deuxième arbre dans le pire cas, au lieu de tous les préfixes des suffixes comme c'est le cas avec cet algorithme.
Par ailleurs, la complexité est asymétrique. En effet, l'ordre des chaîne a son importance lors de l'appel de la fonction. On remédiera à cela en définissant n = min($|s_1|$, $|s_2|$) et m = max($|s_1|$, $|s_2|$).

La complexité espace est en $O(m^2)$ nombre de lettres car on construit tous les préfixes des suffixes de s_2 de taille m.

Exercice 4 compression de l'arbre des suffixes

Les fichiers sont dans `ex4_compress/`*, les tests sont exécutés en lançant* `compile_and_run.sh`

4.15) La méthode demandée est implémentée sous le nom '`compression_str`'. La même méthode a été implémentée avec la structure proposée en 4.19 sous le nom '`compression_int`'.

4.16) Le symbole # permet de savoir connaître une fin du mot, sans quoi il serait impossible de déterminer une fin avant la fin de la branche parcourue. Par exemple, 'NENE' et 'NE' ne serait qu'un seul nœud 'NENE' alors qu'avec le symbole #, il y le nœud 'NE' et ses fils '#' et 'NE'.

4.18) Considérons le cas où toutes les lettres sont différentes, alors il y a n+1 nœuds de tailles 1 à n+1. La complexité en nombre de cases mémoires est donc de l'ordre de $O(n^2)$. Si on utilise des indices au lieu de stocker une chaîne dans les nœuds, la complexité est de l'ordre de O(n) car on stocke 2 entiers au lieu de k caractères.

4.20) Soit n et m le nombre de caractères des deux chaînes. On traite la complexité en nombre de nœuds à accéder ou créer (une création compte pour un accès de nœud). Pour trouver une chaîne commune de longueur maximale, on crée l'arbre de la plus petite chaîne de taille n, en parcourant les lettres de cette chaîne à partir de la fin pour insérer les indices du suffixe dans l'arbre, qui est vide au départ.
Ainsi, la complexité de cette étape en temps dans le pire cas en accès à un nœud de la chaîne est en $\sum_{k=n-1}^{0}(C_{ajout}(k))$ avec $C_{ajout}(k)=\sum_{i=1}^{k}\alpha+\sum_{i=1}^{k}1=(\alpha+1)k$ car il faut α accès à un nœud pour trouver la bonne première lettre dans le pire cas, c'est à dire si la lettre est à la fin de la liste existante, puis i pour créations de nœuds, soit $(\alpha+1)\sum_{k=n-1}^{0}k=(\alpha+1)\frac{n(n+1)}{2}$ accès à un nœud donc en $O(\alpha*n^2)$ accès de nœuds, $O(n^2)$ si on considère α borné par une constante.

Puis, il faut autant d'accès aux nœuds que de nœuds pour compresser l'arbre. La compression se fait donc en $O(n^2)$ accès de nœuds.

Il faut ensuite obtenir l'indice de fin de chaque suffixe de la deuxième chaîne dans l'arbre de la première chaîne. Cette opération se fait en $O(\alpha*n*m^2)$ accès de nœuds, $O(m)$ du mot pour obtenir les indices des suffixes puis, pour chaque paire d'indices de suffixes, il faut comparer à au maximum $\alpha*n$ nœuds (avec le même raisonnement que la complexité d'ajout) soit $\alpha*n$ accès aux nœuds.

Enfin, il faut ajouter la chaîne des indices obtenus, en $O(\alpha*min(n,m))$ accès aux nœuds, avec le même raisonnement que l'ajout, car il s'agit d'une copie du premier arbre dans le pire cas.

Ainsi, la complexité temps en pire cas de `sousChainesCommunes` en accès aux nœuds est en $O(\alpha*n*m^2)=O(n*m^2)$ si on considère que α ≤ constante.

D'après la question 4.18, le nombre de nœuds est linéaire selon n.
Ainsi, si on utilise des chaînes dans les nœuds, il y a de l'ordre de $O(n^2)$ lettres (on stocke autant d'informations que dans un arbre non compressé, sur moins de nœuds). Cependant, lorsqu'on utilise des indices, un nœud contient un nombre constant de données, quelque soit la taille des entrées. Cette complexité en temps est linéaire, de l'ordre de O(n). C'est ce type d'arbre qu'on utilise pour nos expérimentations.

Exercice 5 construction efficace de l'arbre des suffixes compressé

Les fichiers sont dans `ex5_compresstree/`*, les tests sont exécutés en lançant* `compile_and_run.sh`

5.23) Pour créer un arbre compressé, nous avons besoins de la liste des paires indices des suffixes, n paires.
Puis, pour chaque paire d'indices, il faut insérer la paire dans l'arbre compressé.
Avec les mêmes raison que pour la question 4.20, cette opération se fait en $C_{ajout}(i)=\alpha\, i(i+1)$ accès à un nœud. Fondamentalement, un arbre compressé dans le pire cas ne change pas d'un arbre non compressé

Ainsi, la complexité en temps dans le pire cas est en $\mathrm{O}(\alpha * n^2)$.

La complexité en espace est en $\mathrm{O}(n)$ caractères car d'après la question 4.18 et 4.20, il y a de l'ordre de $\mathrm{O}(n)$ nœuds dans l'arbre, chacun ne gardant en mémoire que l'indice de début et l'indice de fin du bout de chaîne, donc un nombre constant de données quelque soit les entrées.

Exercice 6 expérimentations

Les fichiers sont dans ex6_experiments/*, les tests sont exécutés en lançant* `compile_and_run.sh`

6.24) La sous-chaîne la plus longue entre les 150 premiers caractères de donnee0 et donnee1 est « D'abord confinée dans les monastères et li » de taille 42.

6.25) Sont joints en annexe les graphiques demandés (figures 1 à 8). On remarque que la méthode dynamique suit une courbe quadratique, représentant la complexité en $\Theta(n*m)$ mais la méthode par arbre ne suit pas une courbe représentant sa complexité. En effet, la méthode dynamique est constamment en $\Theta(n*m)$ alors que la complexité de la méthode par arbre compressé a été calculée pour le pire des cas, qui n'est pas le cas présent, et qui est très rare.

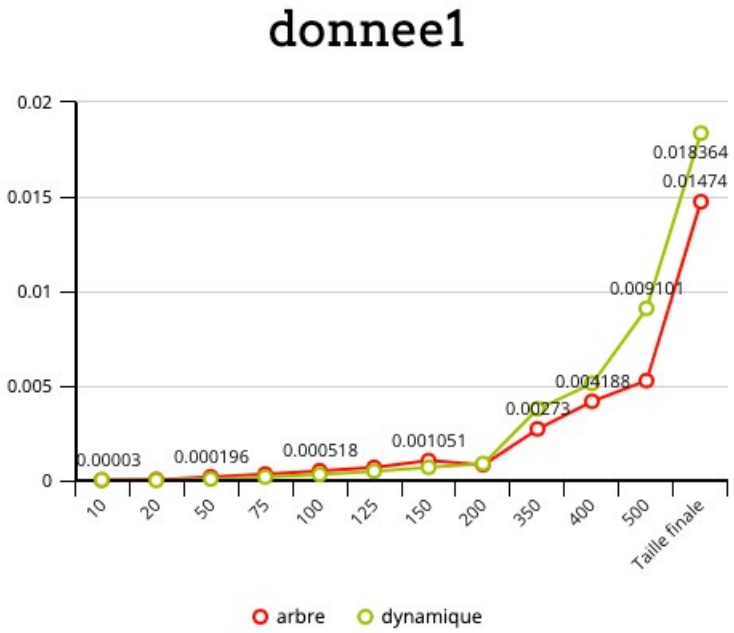
De plus, on peut constater que la méthode dynamique est plus rapide sur de petites entrées alors que la méthode par arbre compressé est plus rapide sur de grandes entrées. Cela est due à la simplicité de la structure de la méthode dynamique, un simple tableau de n*m cases alors que la méthode par arbre compressé demande de créer un arbre des suffixes, qui demande beaucoup plus de temps de calcul.

On peut aussi remarquer que la taille en espace (Figures 9 à 14) de l'arbre compressé avec des indices est linéaire en fonction de la taille de la chaîne la plus petite, et est bien moindre que la taille en espace de la méthode dynamique, qui grandit de façon quadratique.

La méthode par arbre avec indices est donc à préférer pour des entrées de grandes tailles, pour un temps de calcul et une utilisation mémoire moindre par rapport à la méthode dynamique.

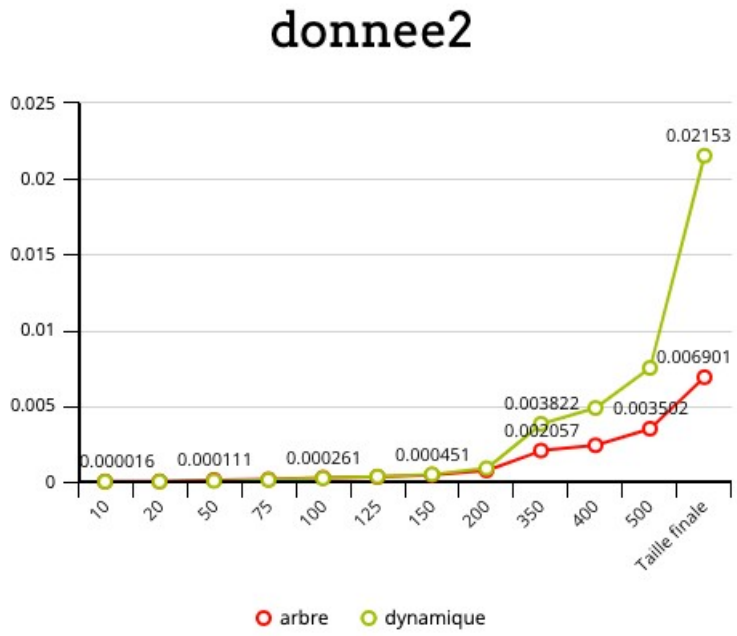
6.27) Les figures 15 à 17 sont les représentations en graphe de DOT des mots « ANANAS_ », « BANANE_ » et « CARAMBAR_ » créées à l'aide du fichier dot.ml.
On a remplacé le # par un _ pour des soucis de cohérence avec DOT.

Annexe



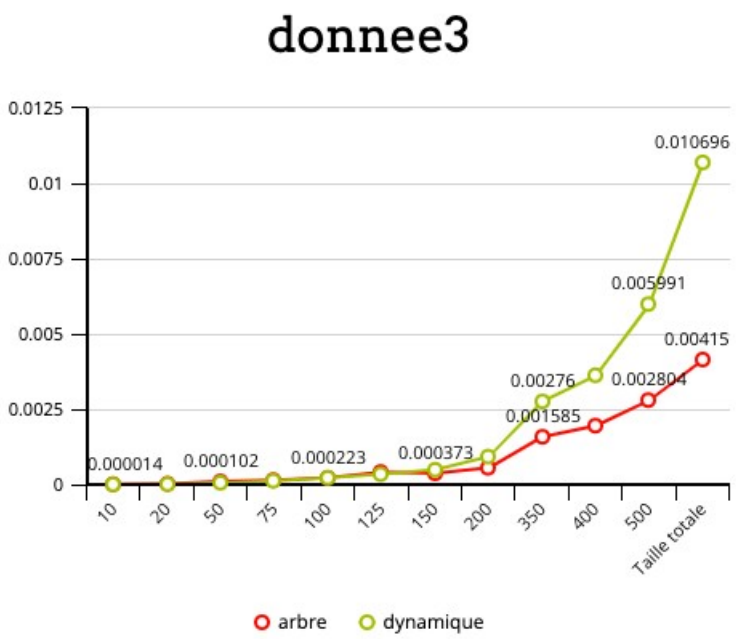
Source: projet_ouv Lucas Fumard et Carla Giuliani

Figure 1 :Graphique des résultats des méthodes dynamique et par arbre compressé entre donnee0 et donnee1



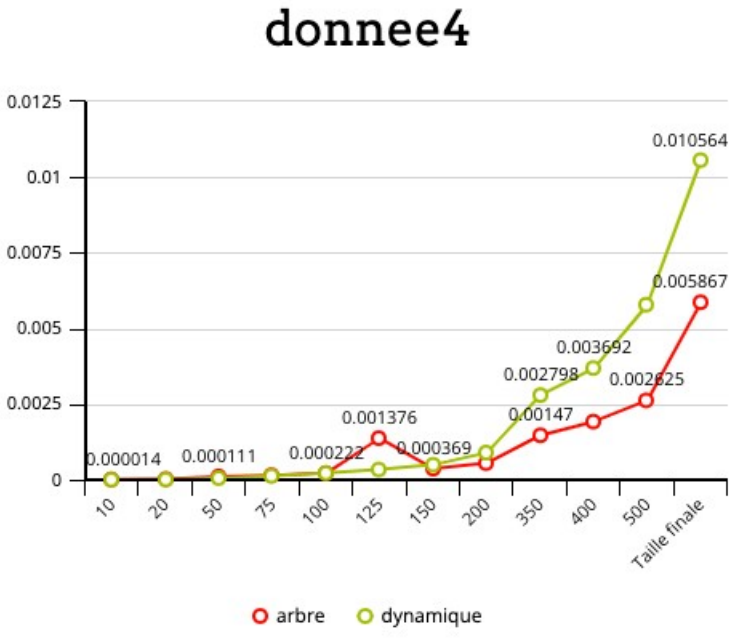
Source: projet_ouv Lucas Fumard et Carla Giuliani

Figure 2 :Graphique des résultats des méthodes dynamique et par arbre compressé entre donnee0 et donnee2



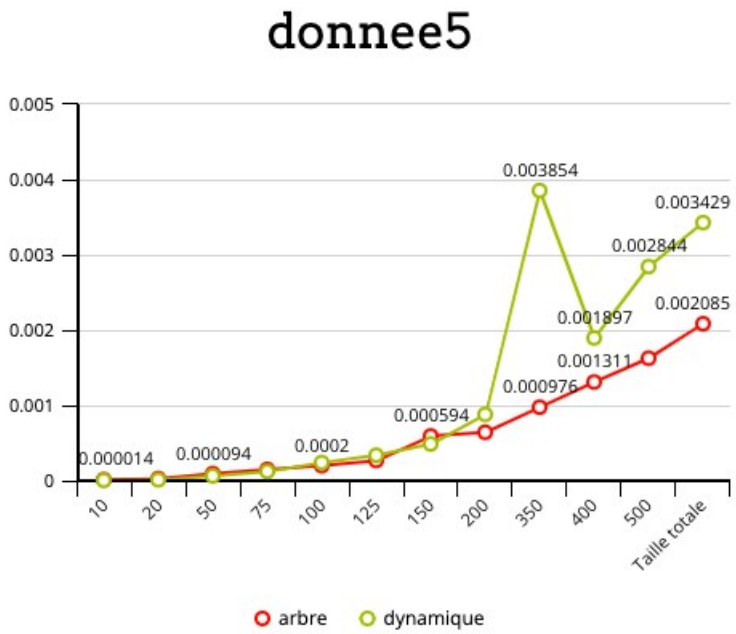
SoSource: projet_ouv Lucas Fumard et Carla Giuliani

Figure 3 :Graphique des résultats des méthodes dynamique et par arbre compressé entre donnee0 et donnee3



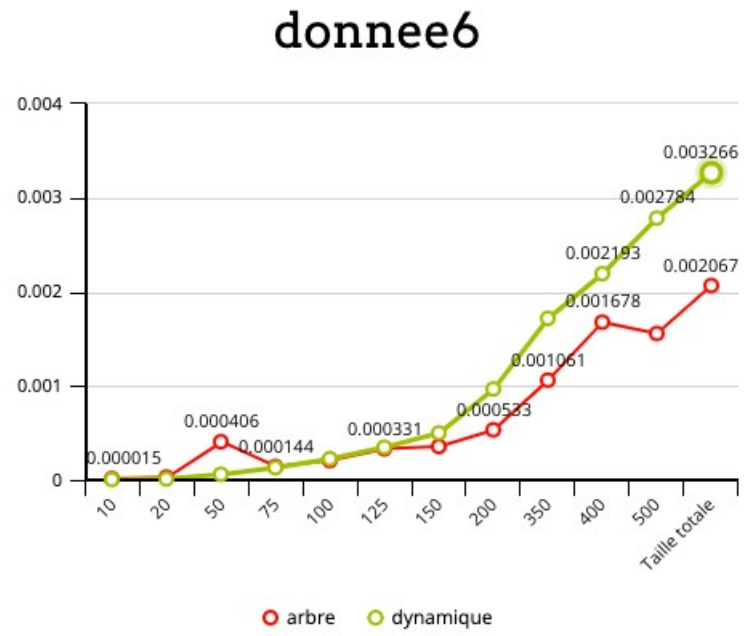
Source: projet_ouv Lucas Fumard et Carla Giuliani

Figure 4 :Graphique des résultats des méthodes dynamique et par arbre compressé entre donnee0 et donnee4



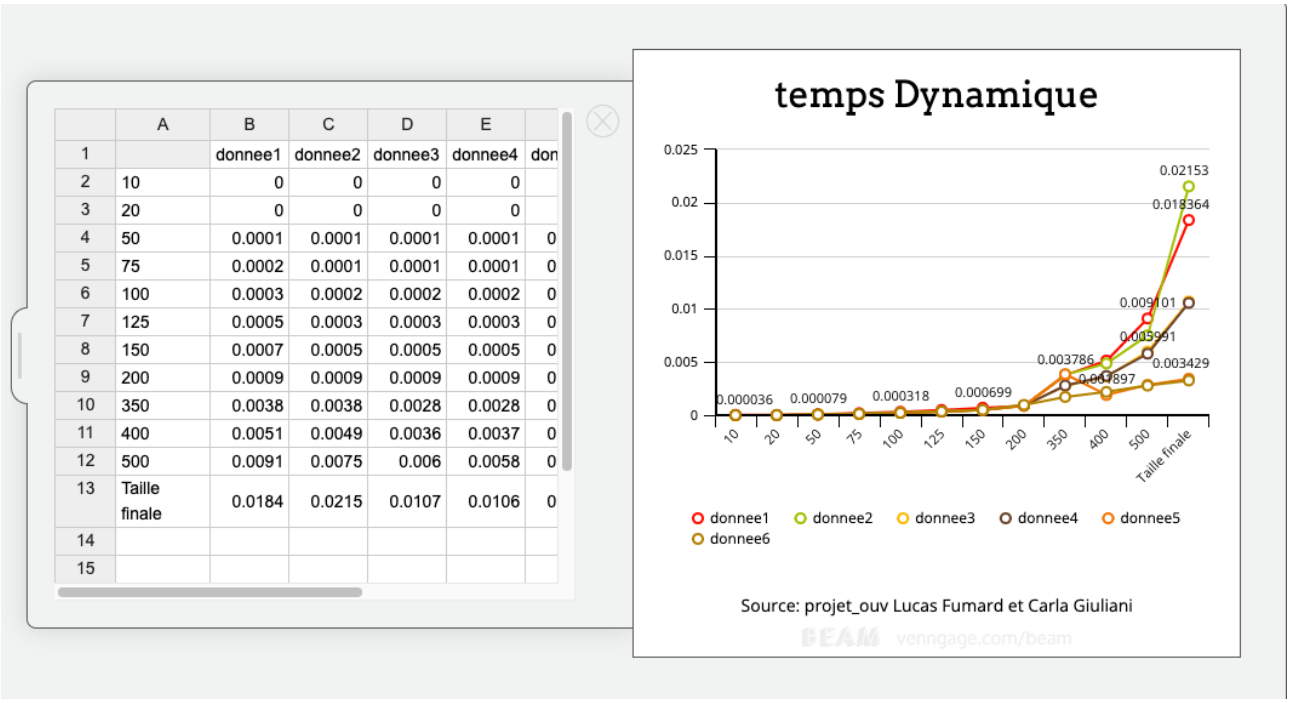
Source: projet_ouv Lucas Fumard et Carla Giuliani

Figure 5 :Graphique des résultats des méthodes dynamique et par arbre compressé entre donnee0 et donnee5



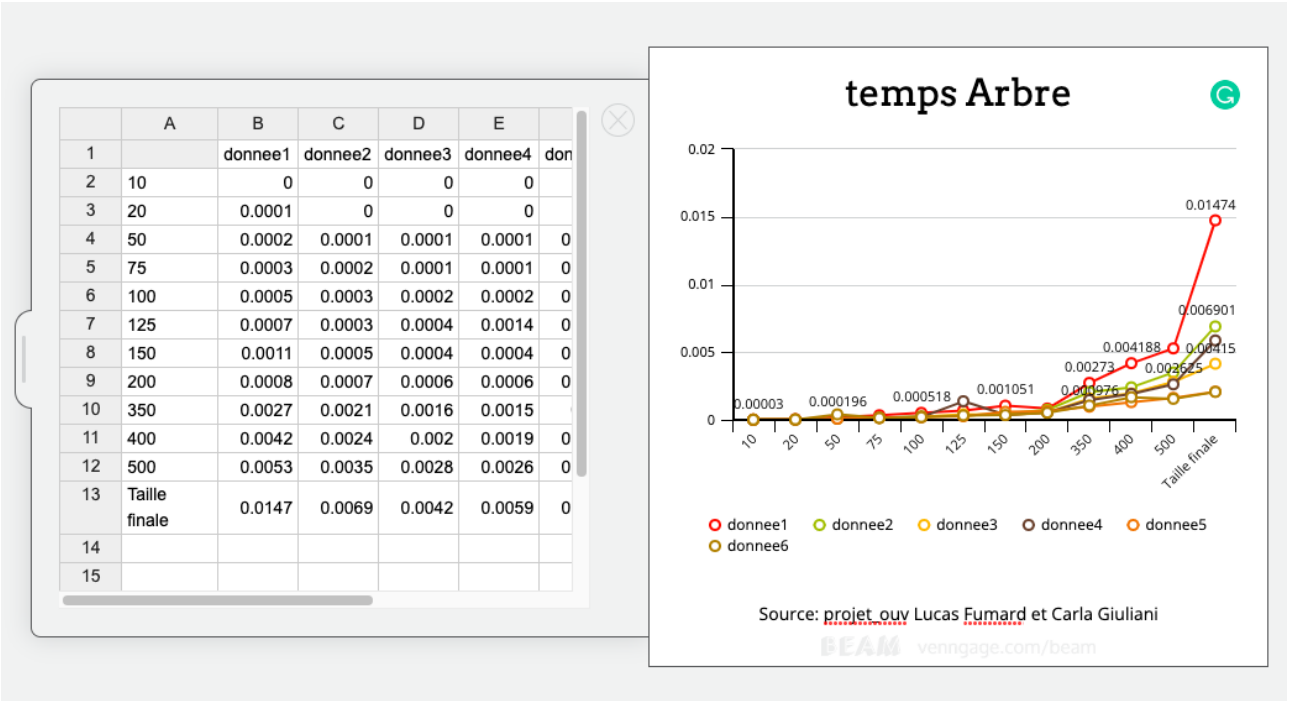
Source: projet_ouv Lucas Fumard et Carla Giuliani

Figure 6 :Graphique des résultats des méthodes dynamique et par arbre compressé entre donnee0 et donnee6



	A	B	C	D	E	
1		donnee1	donnee2	donnee3	donnee4	don
2	10	0	0	0	0	
3	20	0	0	0	0	
4	50	0.0001	0.0001	0.0001	0.0001	0
5	75	0.0002	0.0001	0.0001	0.0001	0
6	100	0.0003	0.0002	0.0002	0.0002	0
7	125	0.0005	0.0003	0.0003	0.0003	0
8	150	0.0007	0.0005	0.0005	0.0005	0
9	200	0.0009	0.0009	0.0009	0.0009	0
10	350	0.0038	0.0038	0.0028	0.0028	0
11	400	0.0051	0.0049	0.0036	0.0037	0
12	500	0.0091	0.0075	0.006	0.0058	0
13	Taille finale	0.0184	0.0215	0.0107	0.0106	0
14						
15						

Figure 7 :Graphique des résultats de la méthode dynamique entre donnee0 et donnee1, donnee2, donnee3, donnee4, donnee5, donnee6



	A	B	C	D	E	
1		donnee1	donnee2	donnee3	donnee4	don
2	10	0	0	0	0	
3	20	0.0001	0	0	0	
4	50	0.0002	0.0001	0.0001	0.0001	0
5	75	0.0003	0.0002	0.0001	0.0001	0
6	100	0.0005	0.0003	0.0002	0.0002	0
7	125	0.0007	0.0003	0.0004	0.0014	0
8	150	0.0011	0.0005	0.0004	0.0004	0
9	200	0.0008	0.0007	0.0006	0.0006	0
10	350	0.0027	0.0021	0.0016	0.0015	
11	400	0.0042	0.0024	0.002	0.0019	0
12	500	0.0053	0.0035	0.0028	0.0026	0
13	Taille finale	0.0147	0.0069	0.0042	0.0059	0
14						
15						

Figure 8 :Graphique des résultats de la méthode par arbre compressé entre donnee0 et donnee1, donnee2, donnee3, donnee4, donnee5, donnee6

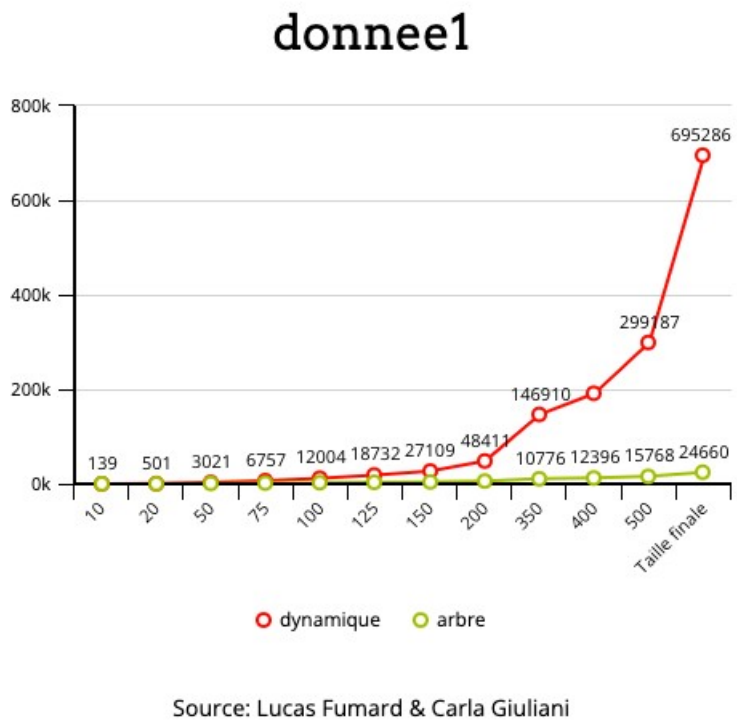


Figure 9 : Mots accessibles à partir de la racine de la matrice crée par la méthode dynamique et les deux arbres de la méthode par arbre pour donnee1

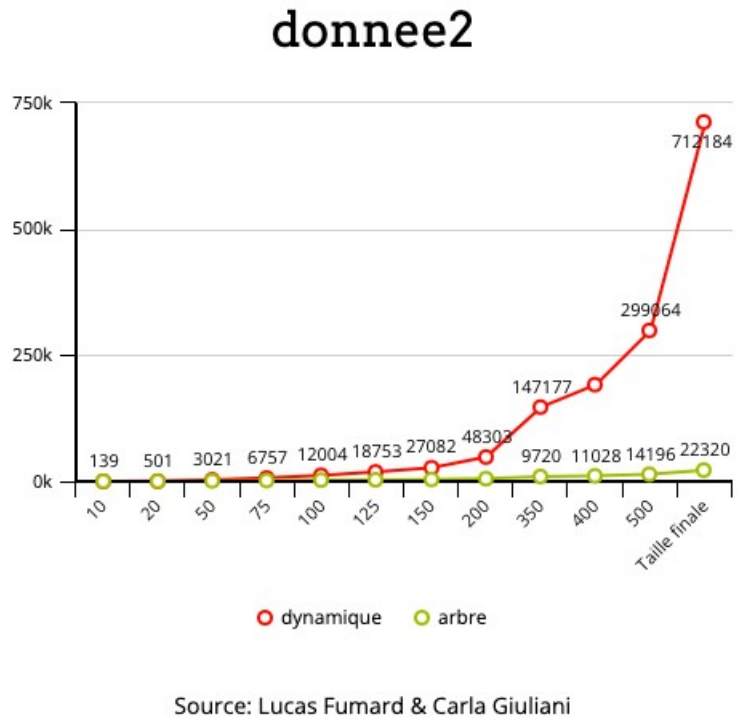
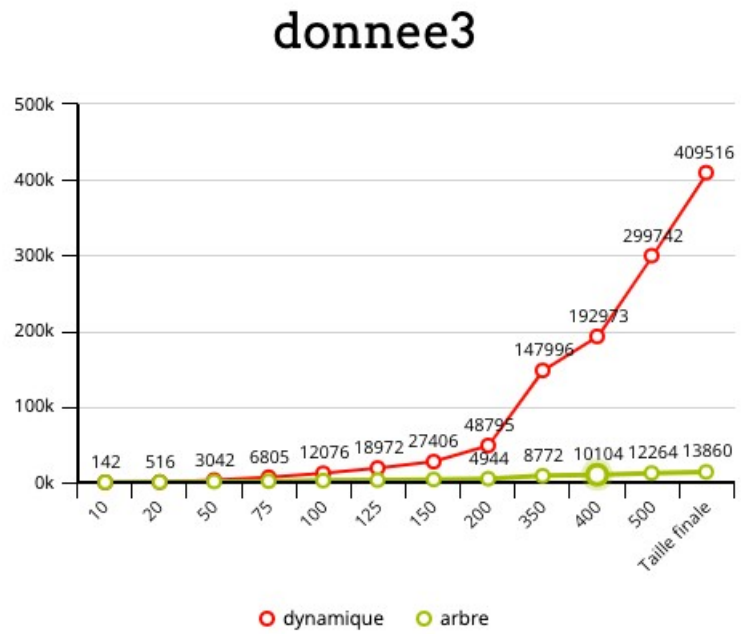
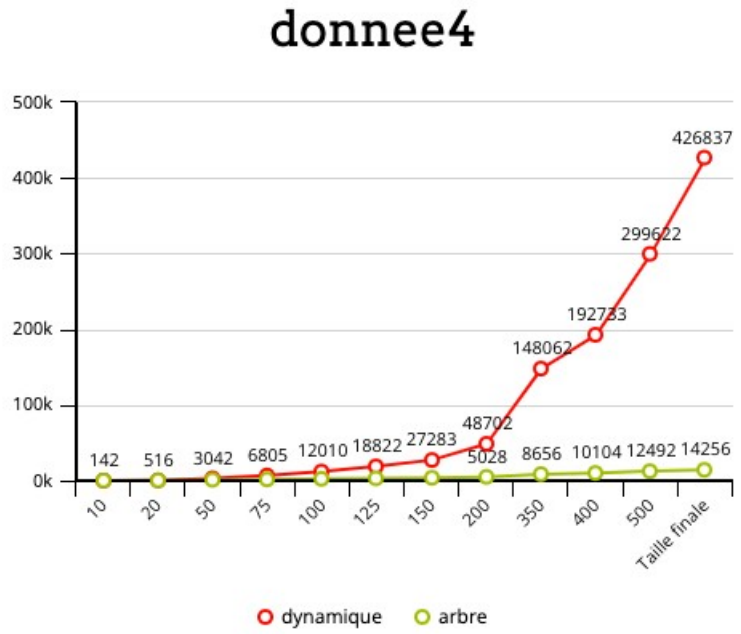


Figure 10 : Mots accessibles à partir de la racine de la matrice crée par la méthode dynamique et les deux arbres de la méthode par arbre pour donnee2



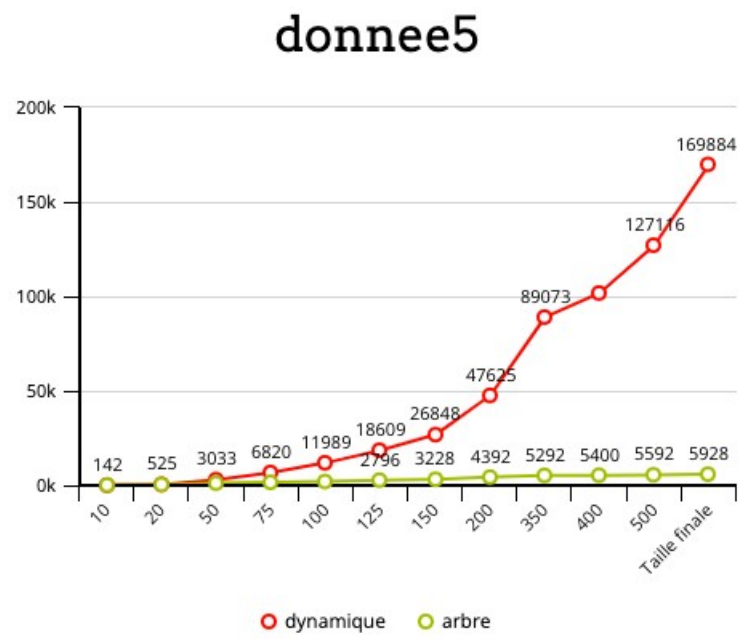
Source: Lucas Fumard & Carla Giuliani

Figure 11 : Mots accessibles à partir de la racine de la matrice crée par la méthode dynamique et les deux arbres de la méthode par arbre pour donnee3



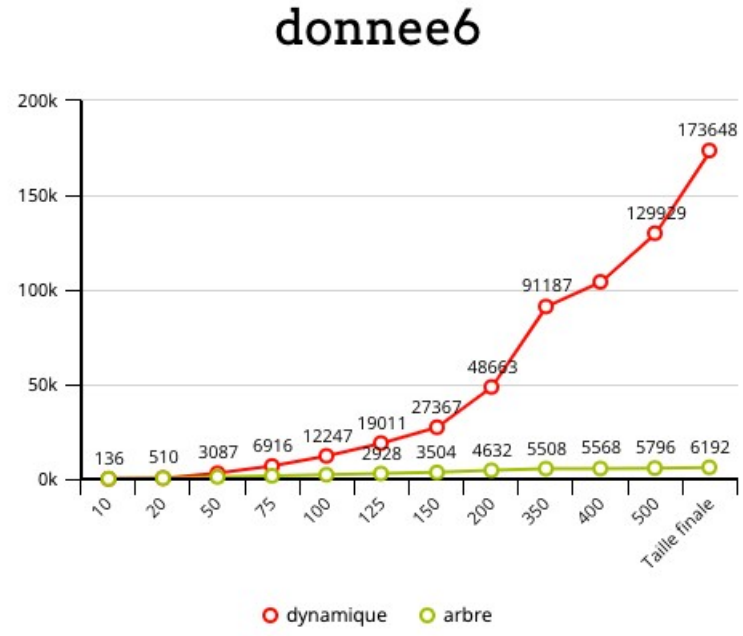
Source: Lucas Fumard & Carla Giuliani

Figure 12 : Mots accessibles à partir de la racine de la matrice crée par la méthode dynamique et les deux arbres de la méthode par arbre pour donnee4



Source: Lucas Fumard & Carla Giuliani

Figure 13 : Mots accessibles à partir de la racine de la matrice crée par la méthode dynamique et les deux arbres de la méthode par arbre pour donnee5



Source: Lucas Fumard & Carla Giuliani

Figure 14 : Mots accessibles à partir de la racine de la matrice crée par la méthode dynamique et les deux arbres de la méthode par arbre pour donnee6

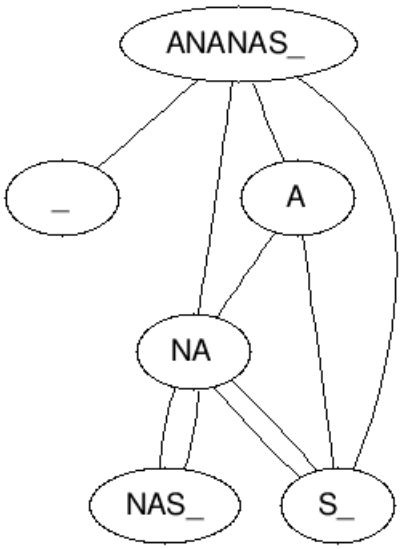


Figure 15 : Graphe DOT du mot « ANANAS_ »

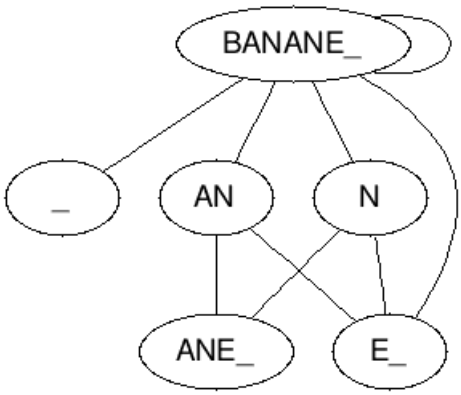


Figure 16 : Graphe DOT du mot «BANANE_ »

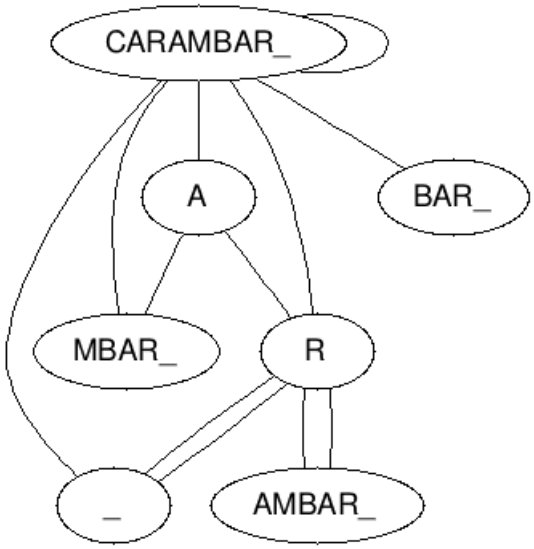


Figure 17 : Graphe DOT du mot «CARAMBAR_ »