UNIVERSITÉ DE NANTES

Algorithmique et Alignement de Chaînes

Distanciel

Recherche de motif avec pré-traitement de T

Auteurs :
Adrien BAZOGE
Maëlle BRASSIER
Solène CATELLA

Référent :
M. Guillaume FERTIN

Table des matières

1 Introduction

L'exercice proposé dans le cadre de ce distantiel *Algorithmique et Alignement de chaînes* s'attache à implémenter et tester trois méthodes appliquées à la recherche exacte de motifs avec pré-traitement du texte T :

- l'arbre des suffixes ;
- le tableau des suffixes ;
- la transformée de Burrows-Wheeler et FM-index.

À l'issue de ce travail, il s'agira plus particulièrement de s'intéresser à l'efficacité de ces algorithmes au travers des indicateurs suivants :

- la taille (en mémoire) de la structure ;
- le temps de construction de la structure ;
- le temps de recherche des motifs.

Les fonctions seront alors testées sur un jeu de 10 textes de tailles différentes. Pour chacun des textes considérés, 1000 recherches de motifs différentes seront effectuées, pour des tailles de motifs allant de 4 à 13, présents ou non dans le texte.

2 Choix de modélisation

2.1 Choix du langage

Les algorithmes ont été ici développés en Python (version 3). Ce choix a été motivé par notre connaissance et accoutumance au langage essentiellement.
Des langages comme C++ auraient sans doute été plus appropriés pour coder certaines structures (cf. arbre des suffixes), permettant par là même occasion d'obtenir des temps d'exécution moindres, mais ne maîtrisant pas tous ce langage, nous avons préféré nous en tenir à Python.

2.2 Description (rapide) des algorithmes implémentés et/ou provenance

Arbre des suffixes

Pour l'implémentation de la construction de l'arbre des suffixes, nous nous sommes appuyés sur une implémentation en temps linéaire [1] (Python), basé sur l'algorithme de Ukkonen. Cependant, les index des suffixes sur les feuilles de l'arbre ne sont pas implémentés dans cette version. Nous avons donc tiré profit d'une seconde ressource [2] - cette fois-ci implémentée en C - pour ajouter les index manquants. Une fonction permettant de parcourir l'arbre en profondeur et d'ajouter les index des suffixes sur les feuilles a ainsi été récupérée et adaptée.

Les fonctions de recherche de motifs ont été implémentées en s'appuyant sur ces deux mêmes ressources. La première ressource disposait déjà d'une fonction permettant de rechercher l'existence d'un motif P dans un texte T : celle-ci a donc été réutilisée. Pour rechercher toutes les occurrences d'un motif dans un texte, nous nous sommes appuyés sur la fonction booléenne précédemment mentionnée, que nous avons ensuite associée à une fonction de parcours en profondeur de l'arbre. Cette dernière a été ré-implémentée afin de récupérer les index des occurrences du motif.

Tableau des suffixes

L'implémentation du tableau des suffixes et des fonctions de recherche de motifs sans LCP se sont appuyées sur le cours. La recherche de toutes les occurrences d'un motif M au sein d'un texte T a

1. https://github.com/pombredanne/suffixTree-3/blob/master/SuffixTree.py
2. https://www.geeksforgeeks.org/suffix-tree-application-2-searching-all-patterns/

néanmoins été reprise à partir d'un post existant [3].
Il a s'agit ensuite d'implémenter ces mêmes fonctions en utilisant cette fois-ci le tableau LCP. Une première version du tableau a été réalisée sans ressources externes, mais enregistrant une complexité supérieure à celle obtenue à partir de l'algorithme de Kasai, nous avons préféré l'implémenter selon cette dernière approche [4] (algorithme repris et adapté). L'implémentation des fonctions de recherche d'existence et d'occurrence de motifs avec LCP n'ont néanmoins pas été menées à terme, faute de temps et de ressources disponibles. Une version inachevée de cette dernière a été proposée, mais celle-ci ne renvoie pas tous les résultats d'occurrences escomptés, raison pour laquelle nous avons ici décidé de ne pas l'inclure dans notre évaluation.

Transformée de Burrows-Wheeler et FM-index

L'implémentation de la Transformée de Burrows-Wheeler a été menée en deux temps : la création de la Transformée (et son inverse) puis la recherche du motif par le FM-index. La première a été implémentée à partir des connaissances et instructions du cours tandis que la seconde s'inspire de [Langmead, 2013]. Ce papier décrit et explique en détails le processus de recherche du nombre d'occurrences d'un motif P dans un texte T. Nous avons également implémenté une fonction capable de renvoyer la position des occurrences de P dans T et une autre indiquant l'existence de ce motif dans le texte.

3 Détails des algorithmes

3.1 Arbre des suffixes

Fonctions implémentées et complexités

Construction de l'arbre des suffixes

La nature arborescente de l'arbre des suffixes contraint à utiliser une classe *Node* pour représenter les noeuds de l'arbre. Chaque noeud dispose d'un dictionnaire avec tous ses noeuds descendants/fils ainsi que les indices de début et de fin de chaîne de caractères présente entre le noeud courant et le noeud parent.

```
__addChar(self, c)
```

Couplée à une boucle pour parcourir le texte, la fonction *__addChar* permet d'ajouter les caractères un à un dans l'arbre des suffixes. Cette fonction se base sur l'algorithme de construction de Ukkonen.

```
setSuffixIndexByDFS(self, node, labelheight)
```

Après avoir ajouté tous les caractères du texte dans l'arbre, la fonction *setSuffixIndexByDFS* est appelée afin d'ajouter les index des suffixes, en parcourant en profondeur l'arbre à partir du noeud *node*.

Construction de l'arbre des suffixes : O(n)
Espace requis : O(n$\times$log(n))
avec n la taille du texte T

3. https://lars76.github.io/various/searching-suffix-array/
4. www.hackerrank.com/topics/lcp-array

Recherche de motif(s)

```
findSubString(self, P)
```

Cette fonction retourne sous forme d'un booléen si un motif *P* est présent dans un texte *T*. L'arbre est parcouru pour chaque caractère du motif : si tous les caractères sont retrouvés successivement dans l'arbre, le motif est présent et la fonction retourne *Vrai* ; dans le cas contraire, elle retourne *Faux*.

Recherche de motif(s) : O(m)
avec *m* la taille du motif P

```
findAllSubString(self, P)
```

Cette fonction retourne toutes les occurrences d'un motif *P* dans un texte *T*. Elle repose sur le même principe que la fonction *findSubString* : on parcourt l'arbre pour chaque caractère du motif et une fois tous les caractères du motif retrouvés, un parcours en profondeur est effectué à partir du noeud du dernier caractère du motif pour récupérer tous les index/occurrences sur les feuilles.

Recherche de motif(s) : O(m+occ)
avec *m* la taille du motif P
et *occ* le nombre d'occurrences du motif

3.2 Tableau des suffixes

Fonctions implémentées et complexités

Construction du tableau des suffixes

```
build_suffix_array(self, T)
```

La fonction présentée ici est une première implémentation du tableau des suffixes. Le texte passé en entrée est parcouru caractère après caractère, de façon à collecter l'ensemble de ses suffixes. Chaque suffixe est alors stocké avec son indice de position dans le texte. Une fois la liste des suffixes acquise, celle-ci est triée par ordre alphabétique puis retournée sous forme de liste d'indices de position.

Construction de l'index : O(n + n×log(n)) donc O(n×log(n))
Espace requis : O(n×log(n))

```
build_LCP_array(self, T, SA)
```

Cette fonction retourne le tableau LCP d'un texte donné à partir de l'algorithme de Kasai.
Le LCP du premier suffixe est d'abord calculé en considérant le suffixe suivant, récupéré à partir du tableau inverse des suffixes. La valeur du LCP est mise à 0 à l'index correspondant si les suffixes ne partagent pas de préfixe commun ou à la longueur du plus long préfixe commun si c'est le cas. L'astuce utilisée par l'algorithme lorsque la valeur du LCP n'est pas nulle consiste à s'appuyer sur la valeur du LCP du suffixe T[i] pour calculer celle du suffixe suivant T[i+1]. La valeur du LCP du suffixe T[i+1] est alors égale à celle du suffixe T[i] diminuée de 1, car l'ordre relatif des caractères reste le même : les caractères partagés par ces deux suffixes en excluant le premier sont donc les mêmes (d'où le -1). Cette façon de faire permet de gagner en complexité.

Construction de l'index : O(n)
Espace requis : O(n×log(n))
avec *n* la taille du texte T

Recherche de motif(s)

```
check_pattern(T, P, SA)
```

Cette fonction retourne la première occurrence d'un motif P dans un texte T à partir du tableau des suffixes SA. Toutes les occurrences du motif P dans le texte T se trouvent dans un intervalle du texte délimité par une borne droite et une borne gauche. L'identification de ces bornes se fait à l'aide d'une recherche dichotomique. L'algorithme a été conçu conformément au raisonnement présenté en cours.

Recherche de motif : O(m×log(n))
avec n la taille du texte T
et m la taille du motif P

```
find_patterns(T, P, SA)
```

Cette fonction retourne toutes les occurrences d'un motif P dans un texte T à partir du tableau des suffixes SA. Elle repose sur une double recherche dichotomique.

Recherche de motif(s) : O(2×m×log(n) + occ) donc O(m×log(n) + occ)
avec n la taille du texte T
m la taille du motif P
et occ le nombre d'occurrences du motif

3.3 Transformée de Burrows-Wheeler et FM-index

Fonctions implémentées et complexités

Création de la transformée BWT

```
rotation(T)
```

Cette fonction prend en entrée un texte T et en dérive toutes les rotations possibles - les rotations étant les chaînes de caractères obtenues en décalant chaque caractère vers la droite. On représente généralement ces rotations sous forme de matrice $m \times m$, mais nous traitons ici une liste R, que nous ordonnons par ordre lexicographique - le symbole $ étant en premier.

```
last_characters(R)
```

Cette fonction retourne les derniers caractères de chaque élément (i.e rotation) de la liste. Cela équivaut à récupérer la dernière colonne de la matrice de rotations ou L (Last). Elle prend donc en paramètre la liste R.

```
create_BWT(T)
```

Cette fonction crée la transformée de Burrows-Wheeler à partir des deux fonctions décrites précédement ; celles-ci vont simuler la création de la matrice de Burrows-Wheeler. À partir d'une chaîne de caractères T - qu'on traite en substituant les espaces vides par le symbole '_' - on récupère sa transformée (dernière colonne L).

Construction de la transformée : $O(n^2 + n \times log(n))$
avec $O(n^2)$ pour la construction du FM-index
et $O(n \times log(n))$ pour la construction du tableau de suffixes
Espace requis : O(n×log(n))
avec n la taille du texte T

Recherche de motif(s)

```
rankAllBwt(bw)
```

Cette fonction permet de renvoyer deux dictionnaires utilisés lors de la recherche de motifs : *rankAll* et *tots*. Les clés de ces dictionnaires sont les caractères de la transformée *bw* (et donc du texte *T*) tandis que leurs valeurs sont respectivement :

— le nombre d'occurrences cumulé et observé pour chaque ligne où *c* est le caractère courant de la colonne L ;
— le nombre d'occurrences total.

```
firstCol(tots)
```

Cette fonction prend en entrée le dictionnaire *tots* créé par la fonction **rankAllBwt**. Grâce au nombre d'occurrences de chaque caractère, nous pouvons retrouver son étendue (i.e son range) dans la matrice puisque chaque rotation/ligne est triée de manière lexicographique. Cette fonction renvoie donc, pour chaque caractère, un tuple représentant son indice de départ et son indice de fin.

```
countMatches(bw, p)
```

Cette fonction compte le nombre d'occurrences d'un motif *p* à partir de la transformée *bw*. Le motif est parcouru de droite à gauche, en récupérant pour chaque caractère courant de *p* son étendue (indice de départ et de fin) dans la transformée. Comme la colonne F de la matrice est triée de façon lexicographique, obtenir l'étendue permet d'obtenir le nombre d'occurrences en soustrayant l'indice de début à l'indice de fin.

```
findPattern(bw, p)
```

Cette fonction retourne une valeur booléenne, indiquant simplement si le motif *p* est présent ou non dans le texte via la transformée *bw*.

```
positionOcc(bw, p)
```

Cette fonction retourne les positions de chaque occurrence dans le texte *T*. Elle utilise le même principe que la fonction *countMatches* mais va faire appel au tableau des suffixes. Comme ce dernier stocke les positions de chaque suffixe, il suffit de l'aligner avec notre matrice et ainsi récupérer chaque indice de position du tableau des suffixes, qui seront nos offsets.

Recherche de motif(s) : O(occ)
avec *occ* le nombre d'occurrences du motif P
on obtient cette complexité grâce à l'implémentation de rankAll qui fait passer d'une complexité de O(n)) à O(1)) en utilisant une matrice $m \times |\sum|$ avec $|\sum|$ la taille de l'alphabet du texte T

4 Choix et méthodes de génération des instances

Génération des instances

Les instances ont été générées à partir du corpus *europarl* fourni sur la plateforme Madoc. Nous souhaitions à la base traiter le corpus dans son entièreté et suivre la procédure décrite dans l'énoncé à savoir le segmenter en 10 sous-textes de tailles égales. Néanmoins, suite à des problématiques de taille de mémoire et temps d'exécution (que nous traiterons plus en détails en *Discussion & Conclusion*), nous avons finalement réduit notre texte à une sous partie du corpus *europarl* contenant 100 000 caractères, que nous nommerons *Texte*1. Chacun des 10 sous-textes de *Texte*1 est donc composé d'environ 10

000 caractères, que nous avons ensuite concaténé itérativement de façon à avoir 10 textes de tailles croissantes. Au vu de nos temps d'exécution très rapides, nous avons choisi d'incrémenter notre taille de texte en passant à un nombre de 500 000 caractères, obtenant un deuxième texte *Texte*2. Les résultats obtenus sur ce deuxième jeu de données sont présentés en *Annexe*), sous forme de tableaux.

Génération des motifs

Suite à la génération de nos instances, il nous a fallu créer 1000 motifs différents, avec des tailles allant de 4 à 13. Ces motifs proviennent de deux ensembles :

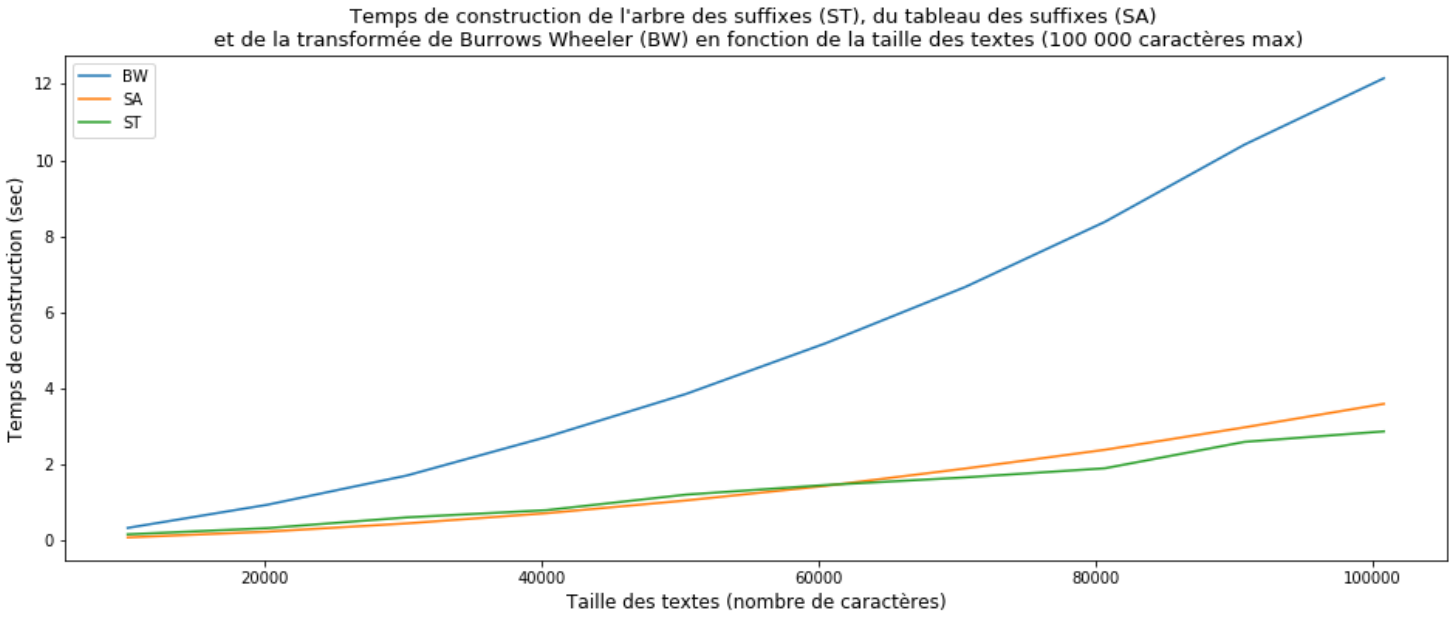
— **un ensemble de 500 motifs** (= 50 motifs x 10 tailles) **contenus dans le texte**. Pour ce faire, nous avons récupéré tous les mots dont la taille était comprise entre 4 et 13, puis nous avons choisi aléatoirement 50 d'entre eux pour chacune des tailles ;

— **un ensemble de 500 motifs** (= 50 motifs x 10 tailles) **pouvant être présents ou non dans le texte**, auquel cas générés aléatoirement. Cette sélection s'est faite sur la base d'un indice binaire tiré aléatoirement entre 0 et 1 (0 : motif généré aléatoirement par concaténation de caractères ; 1 : motif existant dans le texte).

5 Figures et tableaux de résultats

Dans cette section, nous présentons notre ensemble de résultats suivant deux modalités de représentations : les graphes reproduisent les performances obtenues sur notre corpus *Texte*1 (100 000 caractères) tandis que les tableaux affichent celles du corpus *Texte*2 (500 000 caractères).

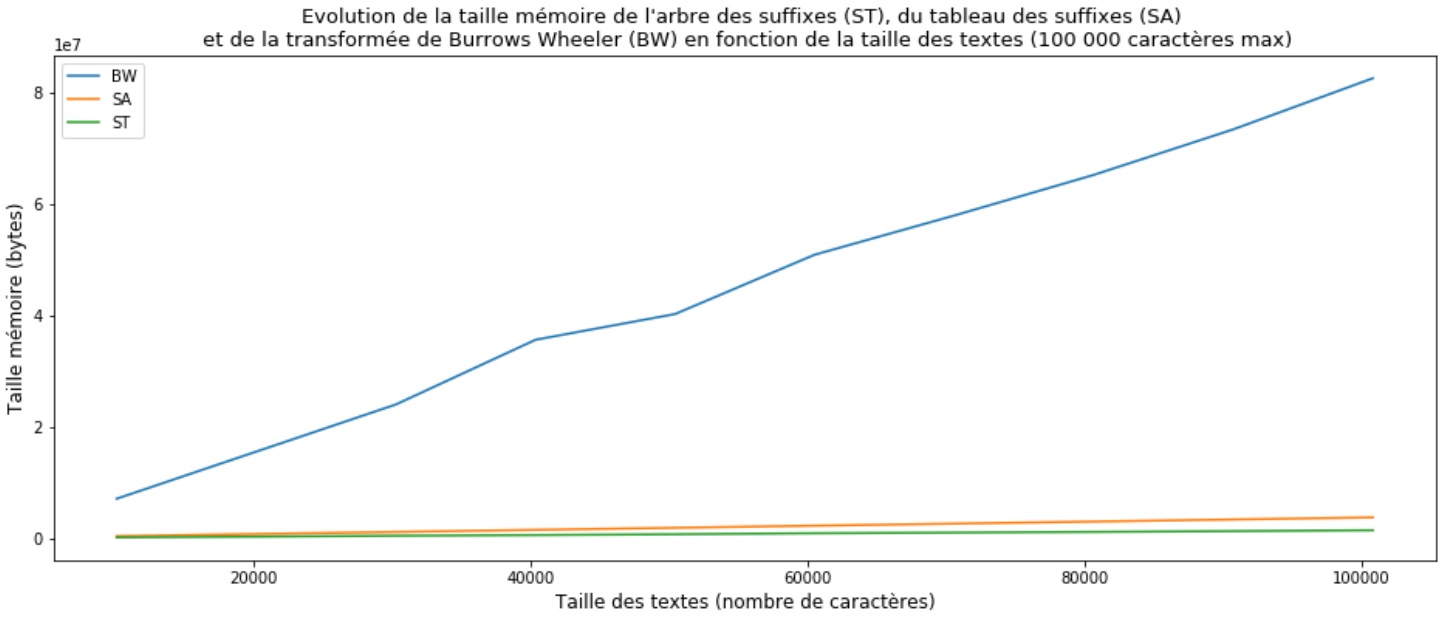
Temps de construction



Le temps nécessaire à la construction de l'arbre des suffixes et du tableau des suffixes est ici semblable, à 1 seconde près (3.6 secondes pour le tableau des suffixes et 2.9 secondes pour l'arbre des suffixes pour un texte de 100 000 caractères). Pour la transformée de BW, il en est tout autrement. Le temps de construction de cette structure est ici presque six fois supérieur et apparaît comme exponentiel, atteignant 12 secondes pour un texte de même taille. Dans l'ensemble, ces temps de construction sont cohérents avec ceux estimés théoriquement, d'une part pour le tableau et l'arbre des

suffixes, qui ont une même complexité de construction théorique, d'autre part pour la transformée de BW, puisque le coût de construction de celle-ci intègre celui du FM-index et du tableau des suffixes. Il s'agit là d'un biais inhérent à la construction de la transformée de BW dont nous sommes conscients. En effet, ces deux éléments sont nécessaires pour chaque recherche de motifs dans le texte et sont instanciés dès le départ et une fois pour toute afin d'optimiser le temps d'exécution. Par conséquent, il était préférable de mesurer ces éléments au départ lors de la construction plutôt qu'à chaque recherche de motif dans le texte.

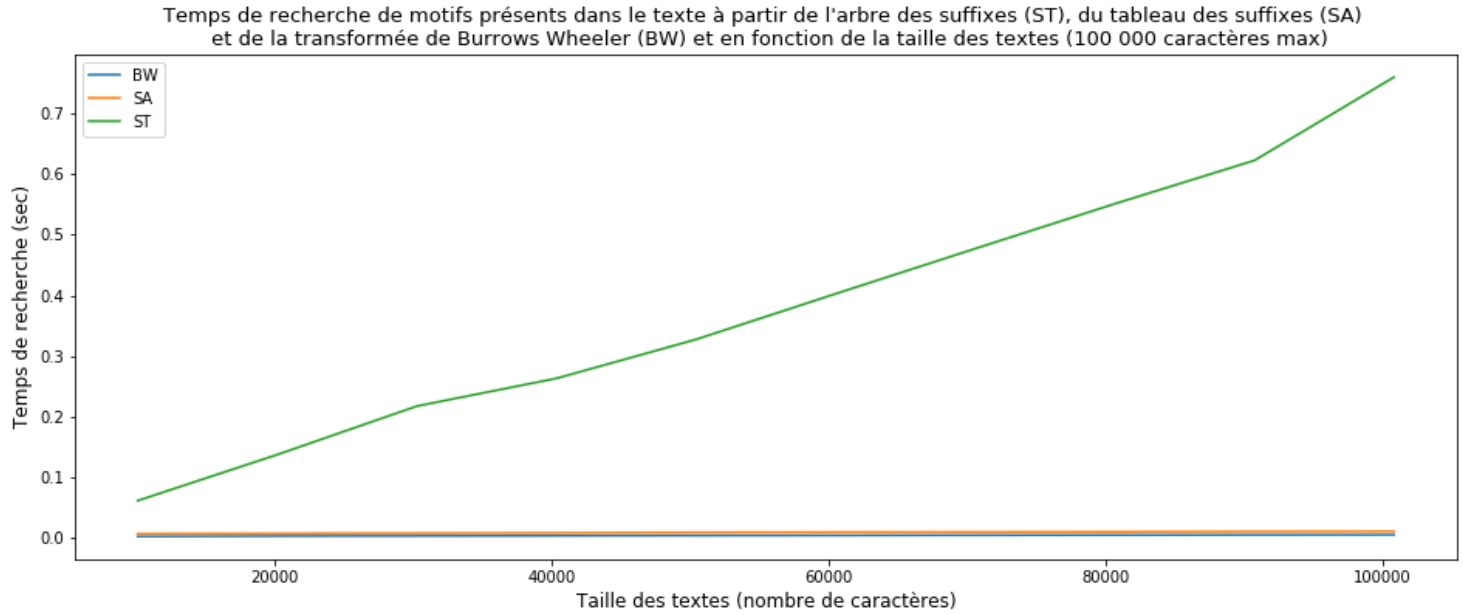
Taille mémoire

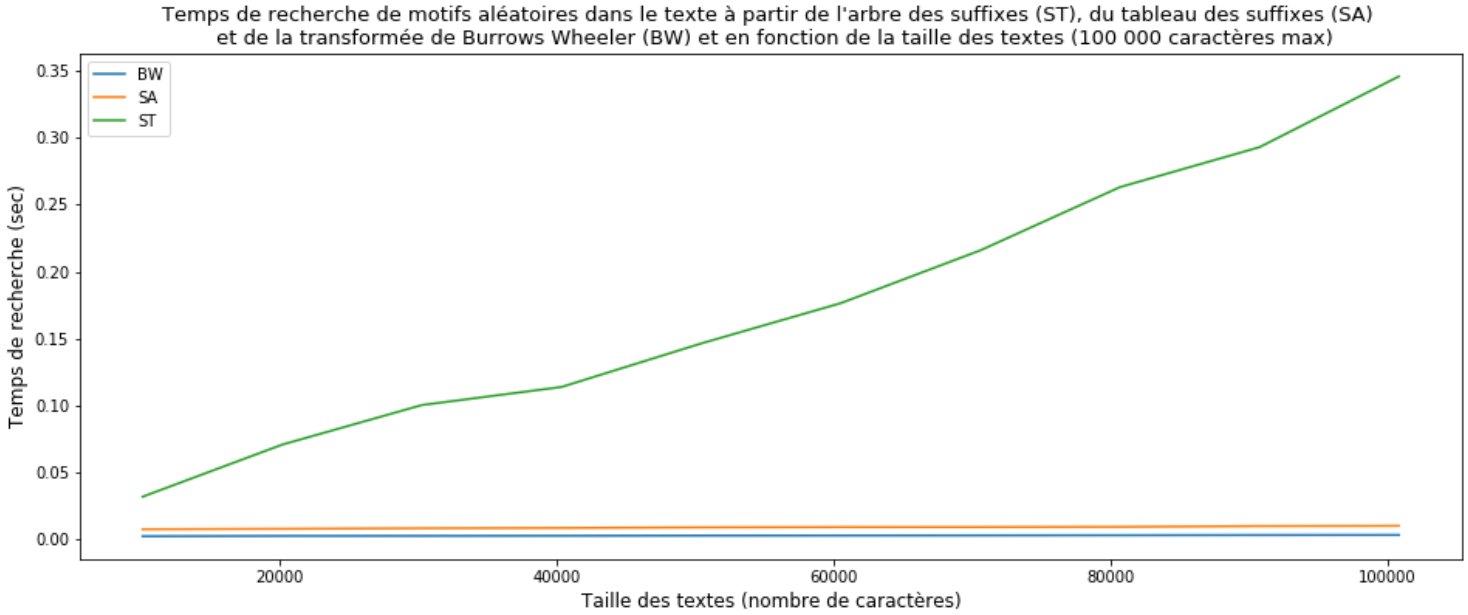


Nous constatons que l'arbre des suffixes est la structure qui occupe le moins d'espace mémoire, avec 1422126 bytes pour 100 000 caractères. Le tableau des suffixes enregistre une taille mémoire légèrement supérieure (3730765), bien qu'il soit plus coûteux de par sa construction, dans la mesure où celle-ci inclut le tableau LCP en plus de la structure en elle-même.

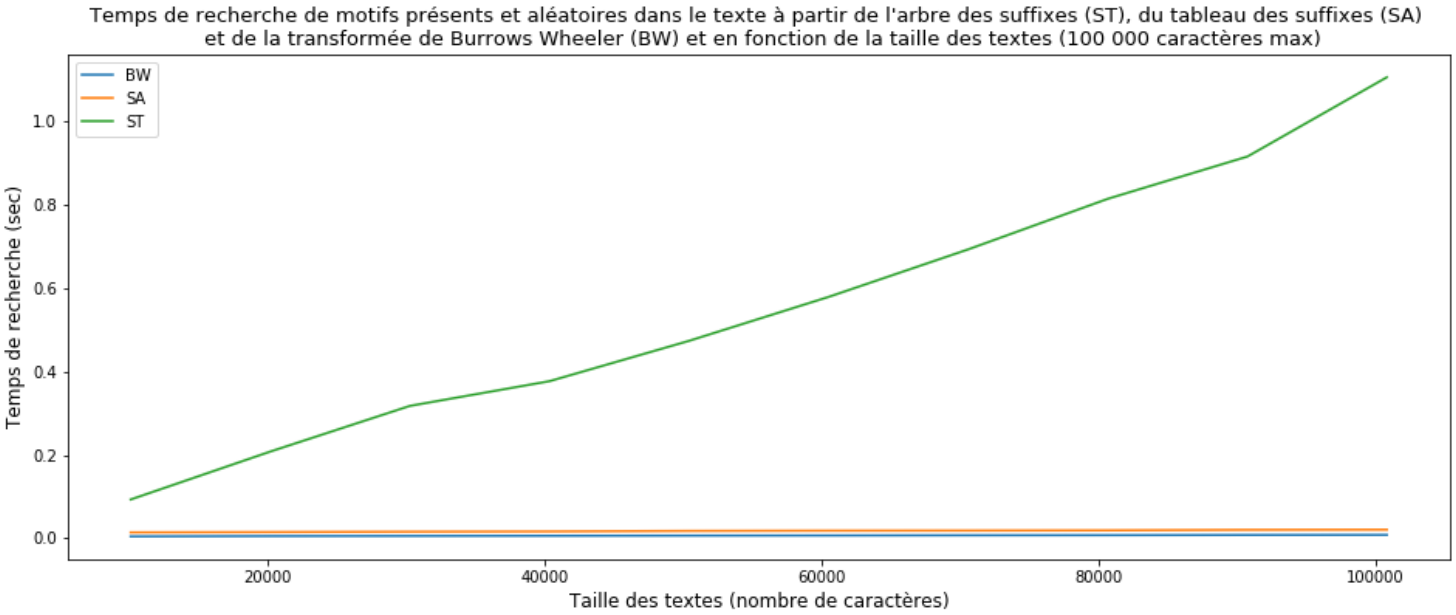
À l'inverse, la taille mémoire occupée par la transformée de Burrows-Wheeler explose (82623307 bytes pour 100 000 caractères). En effet, en plus de créer sa propre transformée - dont la taille dépend du nombre de permutations checkpoints exclus - ainsi que son FM-index, celle-ci inclut la construction du tableau des suffixes. C'est donc nécessairement ici la structure la plus coûteuse en mémoire.

Temps de recherche de tous les motifs en fonction de l'ensemble de motifs





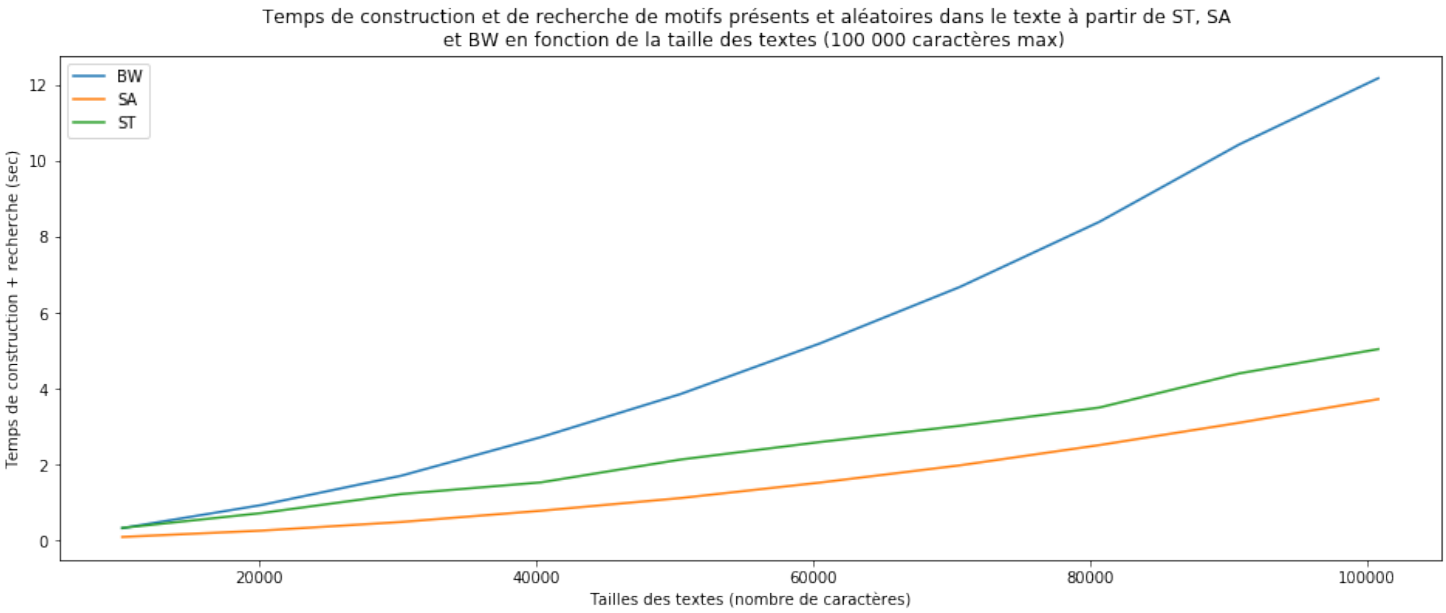
Les deux graphes ci-dessus représentent le temps de recherche sur nos deux ensembles de motifs (motifs présents dans le texte ; motifs générés aléatoirement).
Un même constat peut être observé pour les 3 structures : les mots générés au hasard sont identifiés (ou non) plus rapidement que les mots effectivement présents dans le texte. Les mots générés aléatoirement ont en effet une probabilité d'apparition dans le texte bien plus faible que les motifs existants, dans la mesure où ils sont issus d'une concaténation aléatoire de caractères, ce qui les rend d'autant plus facile et rapide à chercher - ils n'ont pas de sémantique particulière. Ce constat est d'autant plus marquée pour l'arbre des suffixes qui fait cette recherche en 0.759 secondes pour les motifs présents contre 0.346 secondes pour repérer la présence/absence de motifs aléatoires (texte de 100 000 caractères référent).



Au vu des résultats obtenus sur l'ensemble concaténé des motifs, nous pouvons dresser quelques conclusions sur le temps de recherche de chaque structure.
Tout d'abord, l'arbre des suffixes s'avère être le plus chronophage, atteignant un temps de 1.105 secondes pour faire cette recherche sur un texte de 100 000 caractères. Ce résultat entre en adéquation avec sa complexité de $O(n + occ)$, la plus haute des 3 structures.
À l'inverse, le tableau des suffixes offre de meilleures performances avec un temps de 0.020 secondes pour une même taille de texte. On peut également expliquer cette différence *via* sa complexité qui est en $O(m \times log(n) + occ)$. En effet, puisque le $log(n)$ réduit la complexité et que m est largement inférieur à n, le temps de recherche à partir du tableau de suffixes apparaît comme significativement plus rapide que celui en utilisant l'arbre. L'algorithme de Burrows-Wheeler suit la même logique, avec une complexité encore plus faible de $O(occ)$.

Temps de recherche de l'existence du motif en fonction de l'ensemble de motifs

En plus des performances évaluées sur l'algorithme de recherche totale des occurrences, nous complétons ici notre analyse avec un algorithme de recherche de l'existence du motif. Cet algorithme permet d'indiquer la présence ou l'absence du motif dans le texte, et en outre d'expérimenter de nouvelles approches sur nos structures. Dans un souci de place et lisibilité, les graphes associés se trouvent en *Annexe*. En comparant les 3 ensembles possibles de motifs, il semble que la nature du motif (aléatoire ou présent) n'influence pas les performances du tableau de suffixes et de Burrows-Wheeler, leurs résultats restant très similaires (différences respectives de 0.001 et 0.0008687973 secondes). Cependant, l'arbre de suffixes y est plus sensible avec une différence de 0.393 secondes dû au fait que sa complexité se base sur la taille du motif m.



Ce dernier graphe représente le temps total d'exécution, à savoir l'addition du temps de construtions et le temps de recherche. Nous observons une inversion des performances. Burrows-Wheeler qui s'érigeait premier dans tous les scénarios (excepté le temps de construction) se retrouve ici dernier, avec une courbe progression et exponentielle, partant de 0.329 secondes pour 10 000 caractères et terminant à 12.166 secondes pour 100 000 caractères.

À l'inverse, l'arbre de suffixes qui souffrait d'un temps de recherche plus lent se place désormais second avec un temps cumulé de 5.041 secondes pour 100 000 caractères, en partie grâce à son faible temps de construction.

Enfin, le tableau de suffixes qui offrait de bonnes performances, à la fois en temps de construction et en temps de recherche, a joui de ce rapport équilibré et a pu se hisser en tête des algorithmes, atteignant les 3.72 secondes pour une même taille de texte.

Résultats des algorithmes sur un texte de 500 000 caractères

Comme présenté dans la section *Choix et méthodes de génération des instances*, nous avons élargi notre texte *Texte*1 de 100 000 caractères à un nouveau texte *Texte*2 de 500 000. Ces résultats sont présentés dans 3 tableaux, en Annexe. Les tendances observées précédemment dans nos graphes sont retrouvées ici, avec un temps de construction linéaire pour l'arbre de suffixes et un temps exponentiel pour Burrows-Wheeler. L'ordre des temps de recherche est également conservé avec Burrows-Wheeler, suivi du tableau des suffixes puis de l'arbre des suffixes.
Quelques points d'intérêt peuvent être soulevés : la dernière itération (correspondant à 500 000 caractères) multiplie le temps de construction des tableaux des suffixes et de la transformée de BW par 6 et 4 (resp.). Nous n'avons pas d'hypothèse concrète pour expliquer ce phénomène ; la seule piste probable pourrait provenir du GPU sur lequel nous avons lancé nos expérimentations (notamment le trafic). Pour cette même raison (éventuelle), nous n'avons pas pu finir l'exécution de Burrows-Wheeler, dû à son temps de construction qui explose. Enfin, il faut noter que les valeurs de la taille mémoire sont erronées car non mises à jour suite à notre changement de mesure de taille de mémoire, qui est décrit en détails dans la section *Discussion & Conclusion*. Elles ne sont donc pas à prendre en compte.

6 Discussion & Conclusion

Cet exercice s'est avéré particulièrement formateur puisqu'il nous a permis de mieux appréhender la recherche de motif avec pré-traitement. Néanmoins, nous avons rencontré un certain nombre de limites non négligeable au cours de notre travail :

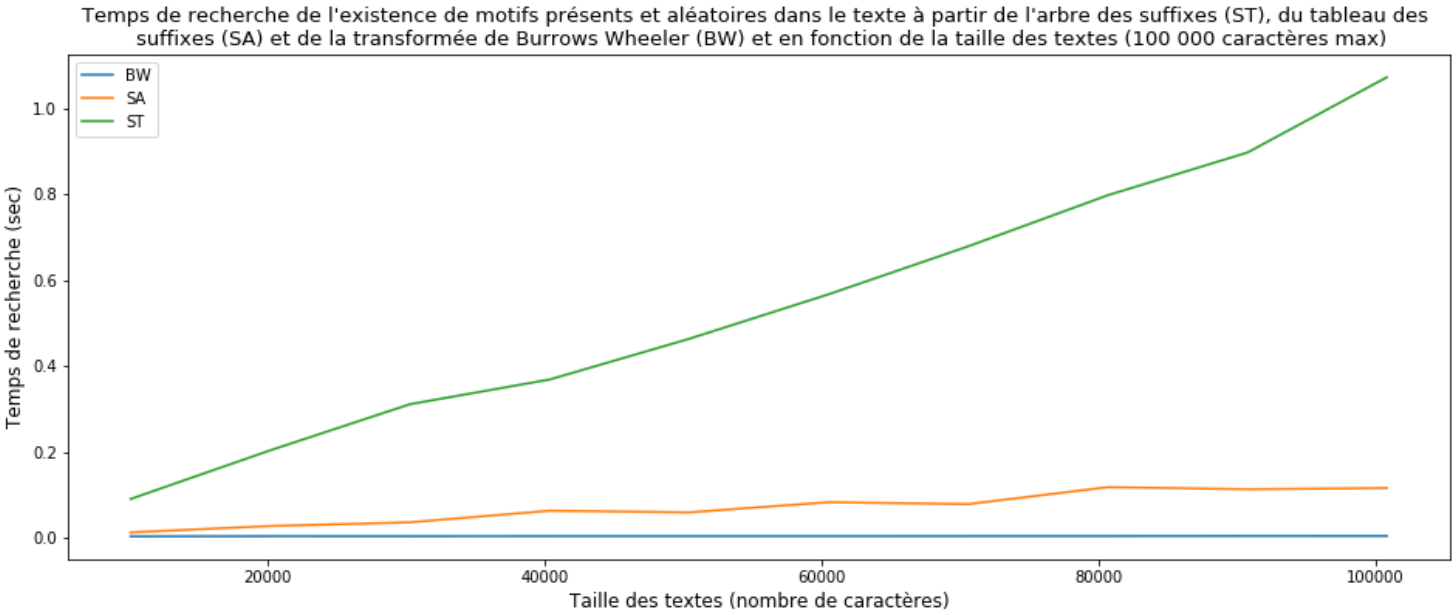
1. D'abord au niveau du calcul de la taille mémoire, nous avons eu une première estimation "biaisée" des mesures. La fonction - *getsizeof()* du module *sys* - utilisée initialement retournait uniquement la taille mémoire de l'objet (donc de notre classe) et non la somme des objets contenus dans l'instance de la classe. Finalement, la fonction *deep_getsizeof*[5] a été après-coup utilisée pour mesurer l'espace mémoire occupé par les différentes structures. Cette fonction mesure de façon récursive tous les éléments contenus dans un objet.
2. Une deuxième limitation a été rencontrée, cette fois-ci relative à l'exécution des scripts, bien trop coûteuse (en temps et en espace) sur nos machines en raison du calcul et de la taille occupée par nos structures. Nous avons donc dû migrer nos scripts sur les serveurs GPU du Mans, bien que fortement ralentis par le trafic des derniers jours. Nous nous sommes rendus compte *a posteriori* que ce même trafic pouvait impacter les calculs relatifs à nos temps de construction de structures, en particulier pour la transformée de BW, dans certains cas deux fois supérieur d'une expérience à l'autre.
3. La taille du corpus ensuite fait partie intégrante des limites auxquelles nous avons été ici confrontés. Nous avons en effet dû réduire drastiquement celle-ci, comportant plus de 24 millions de caractères à 100 000 caractères dans un premier temps puis 500 000 dans un second temps, car l'exécution des scripts sur la taille du corpus d'origine était trop coûteuse en temps et mémoire. Dans le cas des 500 000 caractères, nous ne sommes déjà pas parvenus à aller au bout de l'exécution pour BW (cf. Annexes).
4. Nous aurions aimé pouvoir implémenter la recherche de motifs à partir du LCP mais ne n'y sommes pas parvenus complètement (omission de certains motifs). Le peu de ressources en ligne ne nous a en effet pas permis de trouver à temps une solution à notre problème.
5. Enfin, afin de réduire le temps d'exécution (et particulièrement le temps de construction) de Burrows-Wheeler, nous aurions pu implémenter une approche de checkpoint, qui aurait permis de réduire la taille de la matrice RankAll et par conséquent réduire le temps de construction ainsi que la taille mémoire.

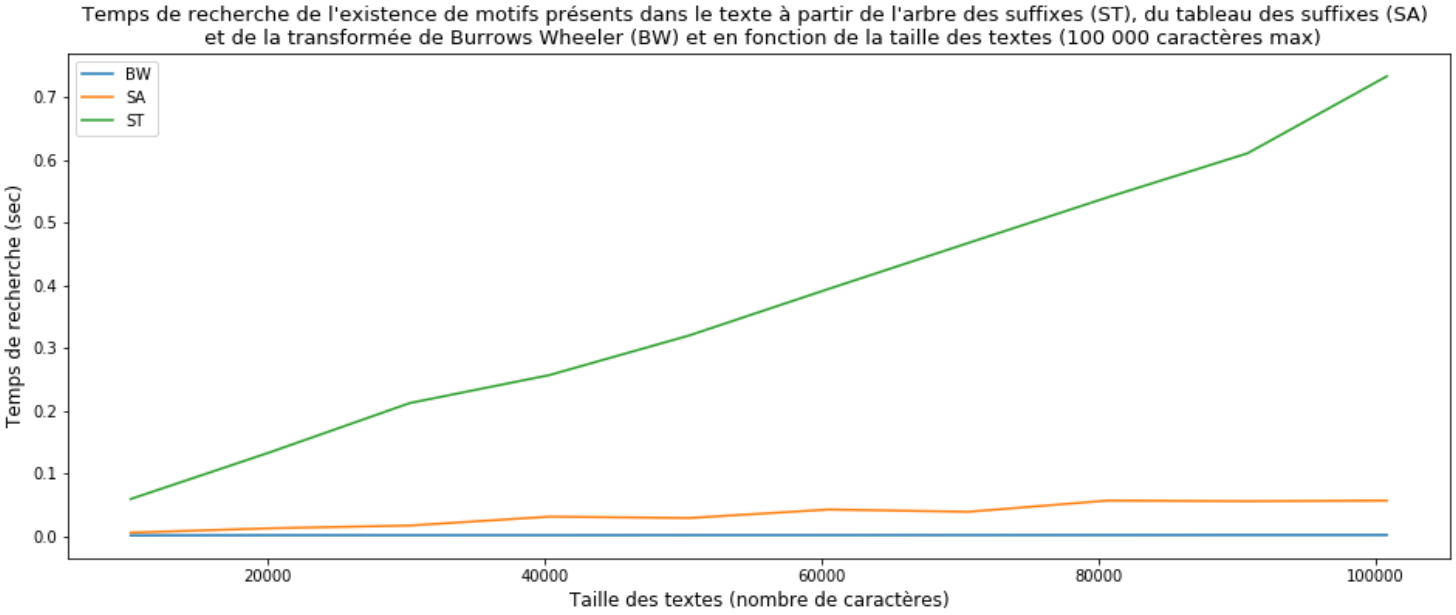
Nous aurions pu limiter la portée de certaines de ces limites en lançant nos scripts sur le serveur GPU plus tôt. Le manque de coordination et de temps ne nous ont cependant pas permis d'y parvenir suffisamment à l'avance. Une semaine de travail supplémentaire nous aurait également sans doute donné la possibilité de résoudre notre problème relatif à l'implémentation de la fonction de recherche à base de LCP.

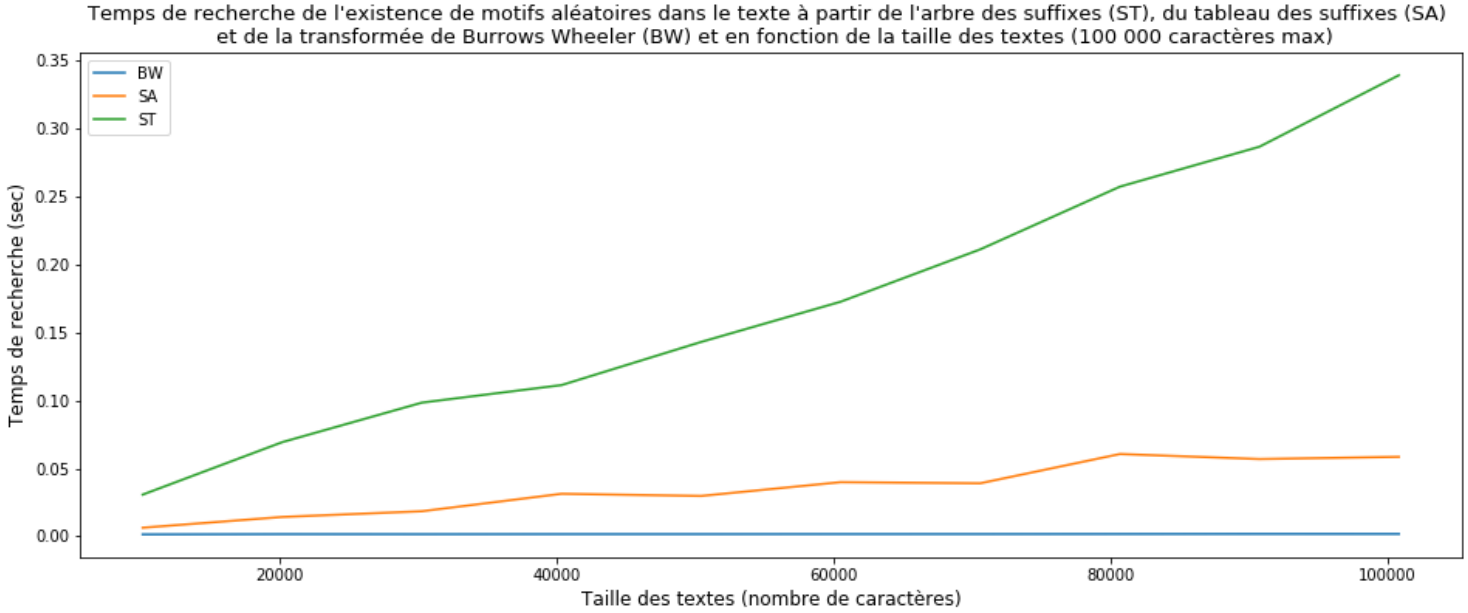
5. https://github.com/the-gigi/deep

7 Annexes

Temps de recherche relatifs à la recherche de l'existence de motif(s)






Temps de recherche de l'existence de motifs aléatoires dans le texte à partir de l'arbre des suffixes (ST), du tableau des suffixes (SA)
et de la transformée de Burrows Wheeler (BW) et en fonction de la taille des textes (100 000 caractères max)
BW
SA
ST
Temps de recherche (sec)
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
20000
40000
60000
80000
100000
Taille des textes (nombre de caractères)

Tableaux des résultats obtenus sur des fichiers de taille de 500 000 caractères maximum

Nb carac.	Taille Mémoire	Temps Construct	Temps Recherche All_Present	Temps Recherche All_Random	Temps Recherche All	Temps Existence Present	Temps Existence Random	Temps Existence
50412	702222	0.77	0.26	0.12	0.40	0.22	0.12	0.34
101330	1422636	1.53	0.27	0.18	0.45	0.25	0.18	0.43
152395	2033661	2.43	0.44	0.31	0.75	0.33	0.25	0.59
203645	2882199	3.20	0.55	0.37	0.93	0.39	0.32	0.72
254120	3644138	3.86	0.74	0.47	1.21	0.63	0.35	0.98
305281	4119043	4.73	0.87	0.51	1.39	0.74	0.42	1.16
355860	5182630	5.53	0.94	0.59	1.53	0.79	0.44	1.23
406473	5836579	6.45	1.11	0.65	1.77	0.85	0.47	1.32
457030	6565888	7.14	1.12	0.71	1.82	0.89	0.51	1.41
504055	7376505	7.90	1.14	0.80	1.95	0.92	0.68	1.60

TABLE 1 – Tableau des résultats du Suffix Tree, en fonction du nombre de caractères (500 000 max)

Nb carac.	Taille Mémoire	Temps Construct	Temps Recherche All_Present	Temps Recherche All_Random	Temps Recherche All	Temps Existence Present	Temps Existence Random	Temps Existence
50412	857204	0.66	0.020	0.014	0.036	0.02	0.02	0.05
101330	1722812	2.65	0.022	0.018	0.039	0.05	0.04	0.09
152395	2590916	4.94	0.022	0.021	0.044	0.08	0.08	0.16
203645	3462164	6.99	0.023	0.022	0.047	0.09	0.08	0.18
254120	4320244	8.99	0.024	0.023	0.047	0.14	0.15	0.29
305281	5189980	13.60	0.024	0.024	0.048	0.16	0.16	0.32
355860	6049820	16.09	0.026	0.024	0.050	0.19	0.19	0.38
406473	6910244	19.92	0.026	0.025	0.050	0.19	0.20	0.40
457030	7769708	25.99	0.027	0.025	0.053	0.22	0.21	0.43
504055	8569132	155.81	0.43	0.38	0.47	0.24	0.24	0.49

TABLE 2 – Tableau des résultats du Suffix Array, en fonction du nombre de caractères (500 000 max)

Nb carac.	Taille Mémoire	Temps Construct	Temps Recherche All_Present	Temps Recherche All_Random	Temps Recherche All	Temps Existence Present	Temps Existence Random	Temps Existence
50412	523021	2.61	0.0007	0.004	0.012	0.0042	0.0034	0.007
101330	1032203	149.34	0.009	0.005	0.015	0.0054	0.0035	0.0090
152395	1542852	211.39	0.013	0.006	0.019	0.0055	0.0036	0.0091
203645	2055350	292.69	0.014	0.006	0.021	0.0057	0.0036	0.0093
254120	2560105	371.80	0.015	0.007	0.023	0.0057	0.0037	0.0093
305281	3071714	498.77	0.016	0.008	0.025	0.006	0.0039	0.010
355860	3577501	2069.72	2.33	0.71	3.057	0.012	0.0049	0.017
...	...	...	...	...	...	...	...	...

TABLE 3 – Tableau des résultats de Burrows-wheeler, en fonction du nombre de caractères (500 000 max)

Références

[Langmead, 2013] Langmead, B. (2013). Introduction to the burrows-wheeler transform and fm index.