Rapport Projet d'Algorithmique Avancée - MU4IN500

Loïc Daudé Mondet

n°21304461

Boudrouss Réda

n°28712638

Sorbonne Université
France

Contents

1. Introduction

Dans le carde de l'unité d'enseignement d'Algorithmique Avancée (MU4IN500 - AlgAv), nous avons réaliser ce projet portant sur les structures de trie : Trie Hybride, et Patricia Trie afin de représenter un dictionnaire de mot.

Langage et interpréteur

Nous avons réalisé ce projet en utilisant un sur-ensemble de Javascript, à savoir **Typescript**. Le projet a été codé et testé avec **Deno 2.0.5** mais n'importe quelle version de Deno 2.x.x devrait marcher.

Javascript est un langage multiparadigme, pour notre cas son coté fonctionnel et orienté objet nous a permis de réaliser ce projet de manière efficace. Il dispose d'une bibliothèque standard très étoffée notamment en ce qui concerne la manipulation des chaînes de caractères et supporte nativement le JSON. Les moteurs Javascript sont très performants ce qui est crucial quand il s'agit de traiter de grandes quantités de données.

Nous avons choisi d'utiliser **Typescript** car il permet de définir des types pour les variables et les fonctions, ce qui permet de détecter plus facilement les erreurs de programmation et avoir un code plus robuste.

Deno est un environnement d'exécution pour Javascript et Typescript, et un des seuls qui supporte nativement le Typescript sans avoir besoin de transpiler le code. Il est sécurisé par défaut, et ne permet pas l'accès au système de fichiers ou au réseau sans autorisation explicite. Il est également très performant et dispose d'une bibliothèque standard très complète.

Utilisation

`deno run cli` vous permet d'intéragir avec les différentes commandes demandé par l'énoncer par exemple :

- `deno run cli inserer 1 temp.txt` permet d'insérer les mots dans le fichier temp.txt dans un Trie Hybrid

`deno test` permet de lancer les tests unitaires. Les test sont reconnaissable par leur extension sous la forme `.test.ts`. Il y en a un pour chaque trie dans le dossier correspondant.

`deno run bench` permet de lancer les benchmarks. Les benchmarks sont reconnaissable par leur extension sous la forme `.bench.ts`. Il y en a un pour chaque trie dans le dossier correspondant.

<!> Exécutez les commandes au root du projet. soit le **parent** de /src (là où se trouve les fichiers de configuration de Deno et le README.Md).

Arborecense

- `src/` contient le code source du projet
- `src/cli.ts` contient le code de l'interface en ligne de commande
- `src/dev.ts` contient un playground pour tester les tries (executable avec `deno run dev`)
- `src/HybridTrie/` contient le code du Trie Hybride (incluant les tests et les benchmarks)

- `src/PatriciaTrie/` contient le code du Patricia Trie (incluant les tests et les benchmarks)
- `src/helpers/` contient des fonctions utilitaires
- `docs/` contient le code source du rapport
- `docs/diapos/` contient les fichiers sources des diapositives
- `docs/report/` contient les fichiers sources du rapport
- `docs/scripts/` contient les scripts pour générer les images du rapport
- `Shakespeare/` contient les texts de Shakespeare (fournis par l'énoncé et utilisés pour les tests et les benchmarks)

Au root du projet, on trouve aussi :

- `deno.json` et `deno.lock` les fichiers de configuration de Deno
- `README.md`
- `rapport.pdf` le rapport généré (que vous êtes en train de lire)
- `diapos.pdf` les diapositives générées

Tous les fichiers générés et lu par le cli doivent être dans le root du projet. (pour les fichiers lu le chemain doit être relatif au root du projet).

2. Patricia Trie

Présentation

Le Patricia Trie est une structure de données qui détient les éléments suivant :

- un label qui est une chaine de texte
- un tableau de pointeur vers les fils

Le Patricia Trie est un arbre de recherche qui permet de stocker des mots. Chaque noeud de l'arbre est un label qui est une chaine de caractère. Les fils d'un noeud sont stockés dans un tableau de pointeur. Chaque fils est associé à un caractère de l'alphabet.

Le principe de base du Patricia Trie est de stocker les mots dans l'arbre en fonction de leur préfixe. Ainsi, les mots qui ont un préfixe commun sont stockés dans le même sous-arbre.

Implémentation

Notre implémentation du Patricia Trie contient les éléments suivants :

```
class PatriciaTrie {
  root: PatriciaTrieNode;
}

class PatriciaTrieNode {
  label: string;
  is_end_of_word: boolean;
  children: Map<string, PatriciaTrieNode>; // Map c'est une hashmap
}
```

Le Patricia Trie est implémenté en utilisant une classe `PatriciaTrie` qui contient un attribut `root` qui est un pointeur vers le noeud racine de l'arbre.

Les noeuds de l'arbre sont implémentés en utilisant une classe `PatriciaTrieNode`. Chaque noeud contient un label qui est une chaine de caractère, un attribut `is_end_of_word` qui indique si le noeud est la fin d'un mot, et un attribut `children` qui est une hashmap qui contient les fils du noeud.

Nous avons opté plutôt pour un boolean `is_end_of_word` pour indiquer la fin d'un mot, car cela permet de simplifier l'implémentation de certaines opérations et de ne pas se réduire à un ensemble de charactère. Dans notre cas notre code supporte tout les caractères UTF-8 car c'est le standard en Javascript.

De même, si nous étions dans un langage bas niveau, nous aurions pu utiliser un tableau de pointeur pour les enfants où la position représente un charactère unicode; Mais en Javascript, nous avons préféré utiliser une hashmap pour simplifier l'implémentation, les clés de cette hashmaps sont le premier charactère du label (Créer une case pour chaque caractère UTF-8 n'est pas envisageable). Les différentes opérations des hashmaps et des tableaux sont sensiblement les mêmes en terme de complexité (O(1) pour l'accès et l'insertion). De plus les Map en Javascript retienne la taille, ce qui simplifie l'implémentation de certaines opérations et permet de ne pas avoir à parcourir l'ensemble des enfants.

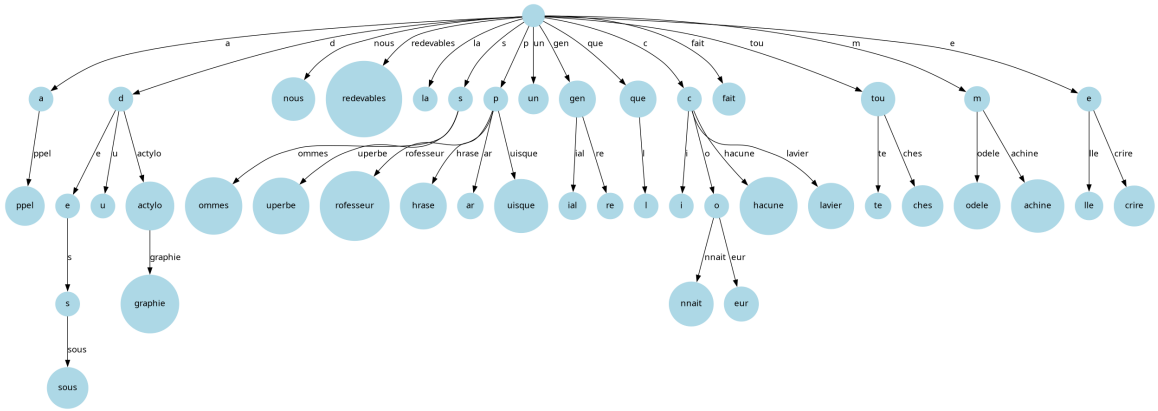

Figure 1: Arbre représentant l'exemple de base avec un Patricia Trie

Analyse

ComptageMot, ComptageNil, Hauteur, ProfondeurMoyenne et ListeMots

Ces opérations procédent de la même manière, elles parcourent l'arbre en profondeur une fois en utilisant une récursion et receuille les informations dont elles ont besoin. Que ce soit dans le pire des cas ou dans le meilleur des cas, nous sommes obligés de parcourir tous les noeuds de l'arbre pour effectuer ces opérations. La complexité de ces opérations est en $O(n)$ où n est le nombre de noeuds de l'arbre.

NB: Pour la méthode ComptageNil, au vu de notre implémentation les pointeurs nulles sont abstractisés par l'objet `Map` de Javascript, pour le bien de l'exercice nous considérons un `Map` vide comme un pointeur nul.

Nous remarquon effectivement que la complexité est en $O(n)$ dans nos tests.

Recherche et Suppression

La supression est semblable à la recherche, car le gros du calcul est fait par la recherche du noeud à supprimer.

Il y a 2 variables qui influent sur la complexité de la recherche et de la suppression :

- n : la longueur du mot recherché
- m : Nombre moyen de caractères dans un label de nœud.

Pour chaque niveau de récursion, nous devons récupérer l'enfant dans `children` ce qui est en $O(1)$ en moyenne, et comparer le label du noeud avec le mot recherché ce qui est en $O(m)$ (dans le code, cela est représenté par la fonction `startsWith`).

Le nombre maximal de niveaux de récursion est proportionnel à la longueur de la chaîne n, divisée par la longueur moyenne des labels m. Cela donne environ n/m appels récursifs.

Ainsi, la complexité totale est :

$$O\left(\sum_{i=1}^{n/m}(1+m)\right) \approx O(n)$$

Dans nos benshmarks, les méthodes sont assez ébruitées, nous supposons qu'au vu du fait que les fonctions s'exécutent très rapidement (de l'ordre de la centaine de nanosecondes), les mesures sont sujettes à des erreurs de mesure et de bruit. Mais si une tendance devait être dégagée, elle aurait l'air logarithmique ?

Prefixe

La méthode traverse plusieurs niveaux de l'arbre jusqu'à atteindre un nœud où `prefix` devient vide ou non trouvé. À chaque niveau :

- La recherche dans `children` est en $O(1)$ en moyenne
- La comparaison de label (avec `startsWith`) est en $O(m)$ (avec m la longueur moyenne des labels)

La profondeur maximale de l'arbre est n/m (avec n la longueur du mot le plus long et m la longueur moyenne des labels), ce qui nous donne alors en moyennne n/m récursion, et ensuite la methode `ComptageMot` est appelée une fois.

La complexité de l'appel final `ComptageMot` dans le pire des cas est $O(n-k) = O(n)$ où k est les noeuds traversés.

Donc, la complexité totale est :

$$O\left(\sum_{i=1}^{n/m}(1+m)\right) + O(n) \approx O(n)$$

Dans nos benshmarks, nous remarquons bien la tendance linéaire de la méthode.

Fusion

1. Parcours des enfants Pour chaque noeud du second Patricia-Trie (node), on effectue les opérations suivantes :

- Vérification si la clé existe déjà dans this.children. Cette opération est $O(1)$ grâce à la structure Map.
- Si elle n'existe pas, l'enfant est ajouté directement, ce qui est également $O(1)$.
- Si elle existe, on appelle récursivement merge sur les sous-arbres correspondants.

2. Gestion des préfixes communs En cas de conflit (préfixes communs), on :

- Calcule la longueur du préfixe commun. Cette opération est proportionnelle à la longueur du label, soit $O(m)$.
- Réorganise les labels et les sous-arbres, ce qui implique de modifier ou de créer de nouveaux nœuds. Ces opérations sont constantes par nœud, soit $O(1)$.

3. Récursion La fonction est appelée récursivement pour chaque enfant. Le nombre total d'appels récursifs est proportionnel au nombre total de noeuds n_2 dans le second Patricia-Trie.

4. Conclusion Chaque enfant dans node est traité en $O(m)$ dans le pire cas, en tenant compte de la vérification et du traitement des préfixes communs. La fonction est appelée récursivement pour tous les enfants, donc pour n_2 noeuds au total. On a donc :

$$O\left(n_2 \times m\right)$$

Dans le pire cas où tous les labels ont des longueurs similaires, m est une constante, et la complexité devient linéaire en fonction du nombre de noeuds dans le second Patricia-Trie :

$$O(n_2)$$

Dans nos benshmarks, nous remarquons bien la tendance linéaire de la méthode.

3. Hybrid trie

Présentation

Le Hybrid trie est une structure de données qui détient les éléments suivant :

- un caractère
- un pointeur left
- un pointeur right
- un pointeur middle

Le Hybrid trie est un arbre de recherche qui permet de stocker des mots. Chaque noeud de l'arbre est un character. Le fils est stocké dans le pointeur middle. Les caractères plus petits que le noeud

sont stockés dans le pointeur left et les caractères plus grands que le noeud sont stockés dans le pointeur right.

Le principe de base du Hybrid trie est de stocker les mots dans l'arbre en fonction de leur préfixe. Ainsi, les mots qui ont un préfixe commun sont stockés dans le même sous-arbre.

Implémentation

Notre implémentation du Hybrid trie contient les éléments suivants :

```
class HybridTrie {
  root: HybridTrieNode | null;
}

class HybridTrieNode {
  character: string;
  is_end_of_word: boolean;
  left: HybridTrieNode | null;
  right: HybridTrieNode | null;
  middle: HybridTrieNode | null;
}
```

Le Hybrid Trie est implémenté en utilisant une classe `HybridTrie` qui contient un attribut `root` qui est un pointeur vers le noeud racine de l'arbre.

Les noeuds de l'arbre sont implémentés en utilisant une classe `HybridTrieNode`. Chaque noeud contient un caractère, un attribut `is_end_of_word` qui indique si le noeud est la fin d'un mot, et trois pointeurs `left`, `right` et `middle` qui pointent respectivement vers les noeuds dont le caractère est plus petit, plus grand ou égal au caractère du noeud.

De même que pour le Patricia Trie, nous avons opté pour un boolean `is_end_of_word` pour indiquer la fin d'un mot, car cela permet de simplifier l'implémentation de certaines opérations et de ne pas se réduire à un ensemble de charactère.

Analyse

ComptageMot, ComptageNil, Hauteur, ProfondeurMoyenne et ListeMots

Ces opérations procédent de la même manière, elles parcourent l'arbre en profondeur une fois en utilisant une récursion et receuille les informations dont elles ont besoin. Que ce soit dans le pire des cas ou dans le meilleur des cas, nous sommes obligés de parcourir tous les noeuds de l'arbre pour effectuer ces opérations. La complexité de ces opérations est en $O(n)$ où n est le nombre de noeuds de l'arbre.

Dans nos benshmarks, nous avons effectivement observé que la complexité est en $O(n)$.

Recherche et Suppression

La supression est semblable à la recherche, car le gros du calcul est fait par la recherche du noeud à supprimer.

Pour chaque caractère du mot on peut soit rechercher à droite ou à gauche soit au milieu :

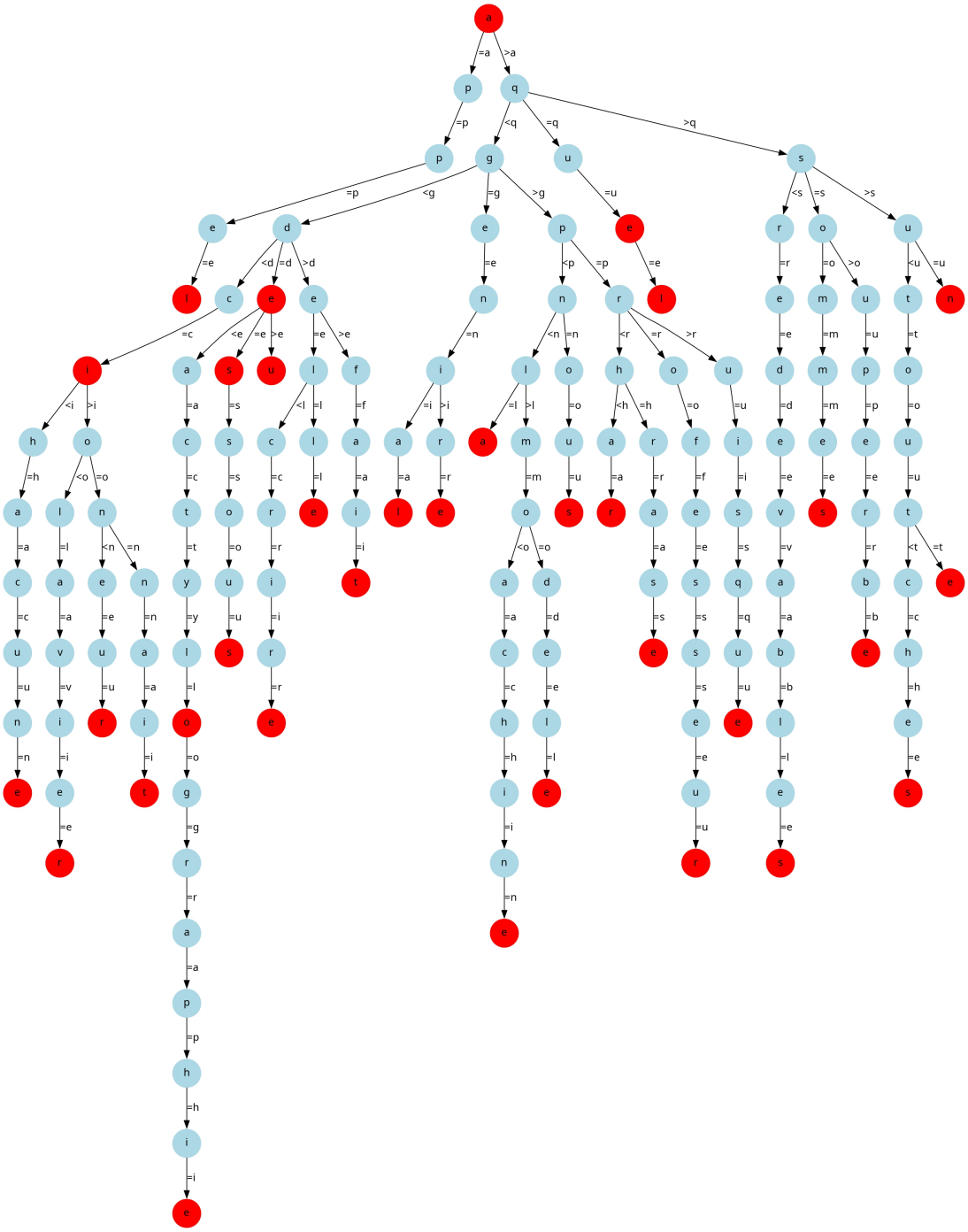

Figure 2: Arbre représentant l'exemple de base avec un Hybrid Trie

- Dans le cas gauche/droite (recherche d'un caractère), la structure de l'arbre est semblable à un arbre binaire de recherche, la complexité est en $O(h)$ où h est la hauteur de l'arbre. La hauteur moyenne d'un arbre binaire de recherche est $h\ logn$ où n est le nombre de noeuds de l'arbre. La complexité de la recherche d'un caractère est donc en $O(logn)$.
- Dans le cas du milieu, la complexité est en $O(m)$ où m est la longueur du mot.

Pour un mot de longueur m, la recherche nécessite m recherches de caractères. La complexité de la recherche est donc:

$$O(m \times logn)$$

À noter que dans le pire des cas (un arbre dégénéré, où $h\ n$), la complexité de la recherche est en $O(n)$ car il est en effet possible de parcourir tous les noeuds de l'arbre pour trouver le mot. (ex: trouver le mot "z" dans un arbre où on a inséré les mots "a", "b", "c", ..., "y", "z")

Dans nos benchmarks, la méthode de recherche est ébruitée, mais nous pouvons observer la complexité logaritmique.

Ceci dit pour la méthode de suppression, notre graphique pointe plutôt vers une complexité en $O(n)$. Nous ne saurons pas expliquer pourquoi la suppression est plus lente que la recherche, alors qu'une bonne partie de l'algorithme est commune aux deux méthodes.

Prefixe

La fonction Prefixe est une combinaison de la methode de recherche et de la methode de comptage, vu qu'elle fonctionne en cherchant le prefixe et en comptant le nombre de mots qui ont ce prefixe. La complexité de cette fonction est donc en :

$$O(m \times logn) + O(n_{sousarbre})$$

où $n_{sousarbre}$ est le nombre de noeuds du sous-arbre qui contient les mots qui ont le prefixe.

À noter qu'il y a plusieurs cas particuliers :

- Si le prefixe n'est pas dans l'arbre, la complexité est en $O(m \times logn)$ vu qu'il y a pas d'appel à la fonction `ComptageMot`.
- Cas où l'arbre est dégénéré, la complexité de la recherche est ein $O(n)$ donc la complexité de la fonction est en $O(n)$ car n prédomine $n_{sousarbre}$.
- Cas grand sous arbre (peut arriver si le prefix est de petite taille), la complexité de la fonction est en $O(n_{sousarbre})$ car $n_{sousarbre}$ prédomine $m \times logn$.

Dans nos benchmarks, nous avons observé une complexité proche du $O(m \times logn)$ mais pas assez claire pour être affirmatif.

Equilibrage

Au début du fichier `src/HybridTrie/HybridTrieNode.ts` nous pouvons trouver constante `BALANCE` qui si elle est définie à `true` permet d'équilibrer l'arbre à chaque insertion. Cela permet de réduire

la hauteur de l'arbre et donc de réduire la complexité des opérations de recherche, de suppression et de prefixe.

L'équilibrage est fait en utilisant la méthode `balance` qui est appellé pour chaque noeud où l'algorithme d'insertion est passée et elle procéde comme suit :

- On calcule le facteur d'équilibre du noeud courant (la différence de hauteur entre le sous-arbre gauche et le sous-arbre droit)
- Si le facteur d'équilibre est supérieur à 1, on fait une rotation droite
- Si le facteur d'équilibre est inférieur à -1, on fait une rotation gauche

La fonction `getBalanceFactor` a pour complexité $O(h)$ où h est la hauteur du sous-arbre. En effet, elle fait appel à la fonction `height` qui a une complexité en $O(h)$.

À chaque appel de la méthode `balance` nous faisons :

- Un appel à `getBalanceFactor` qui a une complexité en $O(h)$
- Si cela correspond à un cas de rotation, nous faison une rotation qui ont une complexité en $O(1)$

La complexité de l'équilibrage est donc dominée par la complexité de `getBalanceFactor` qui est en $O(h)$.

Sachant que notre algorithme d'équilibrage est appelé à noeud où passe l'insertion, la complexité de l'équilibrage est en $O(h^2)$ où h est la hauteur de l'arbre.

À noter que notre méthode d'équilibrage peut être améliorée en stockant la hauteur de chaque noeud pour éviter de recalculer la hauteur à chaque appel de la méthode `height` (appelé par `getBalanceFactor`). Cela permettrait de réduire énormément le temps de calcul de l'équilibrage.

Dans nos benchmarks, nous avons observé que l'équilibrage permet de réduire la hauteur de l'arbre et donc de réduire la complexité des opérations de recherche, de suppression et de prefixe. Cependant pas de manière significative. En effet, un arbre Hybride par ajout de mots est proche d'un arbre équilibré, le cas des arbres dégénérés est rare,donc l'équilibrage n'apporte pas de grand changement.

Ceci dit, l'équilibrage ralenti remarquablement l'insertion, jusqu'à 1000 fois plus lent que sans équilibrage.

4. Comparaison

5. Conclusion

6. Annexe

PatriciaTrie

HybridTrie

Balance

Comparaison

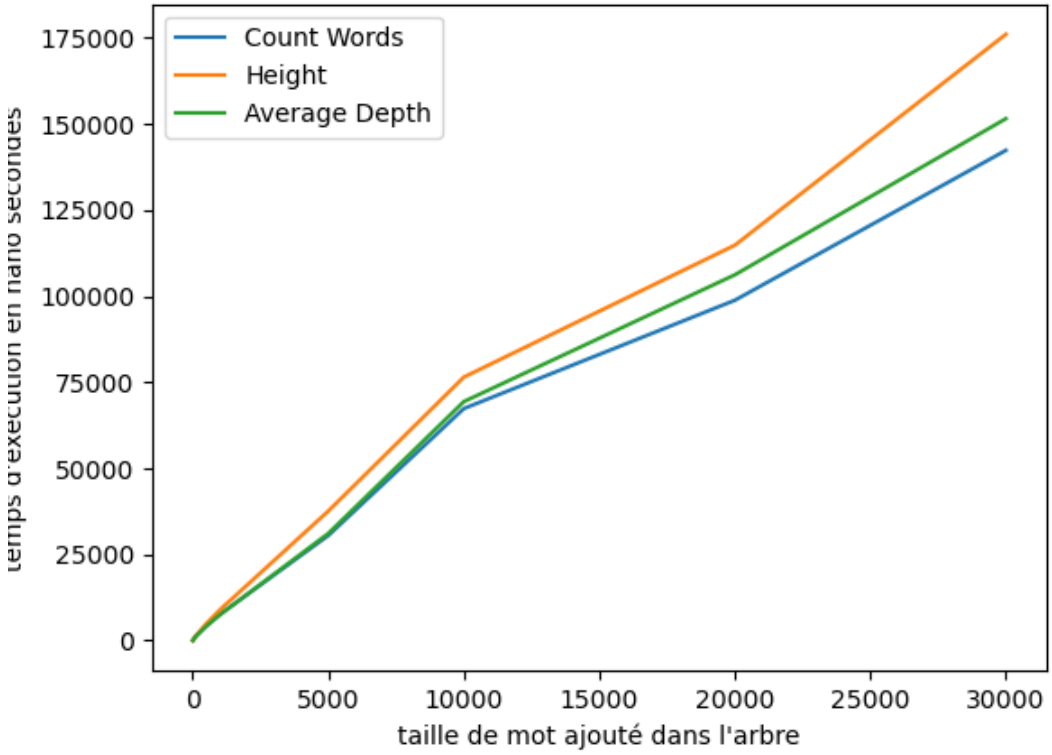

Figure 3: Benchmark des méthodes ComptageMot, Hauteur, ProfondeurMoyenne

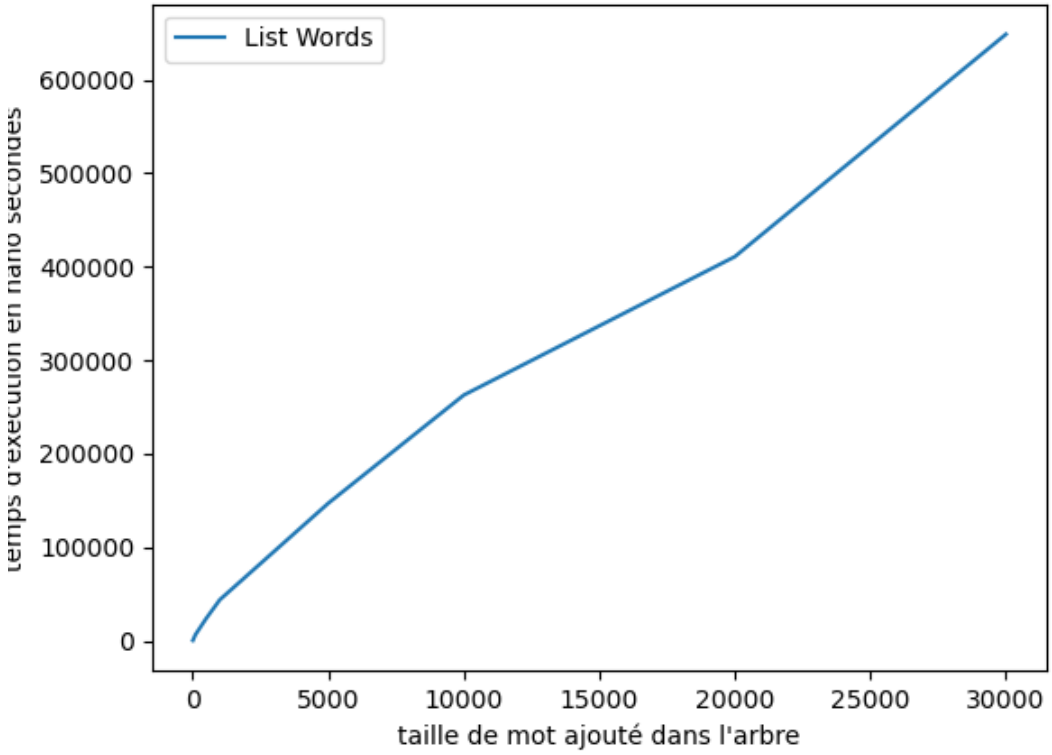

Figure 4: Benchmark de la méthode ListeMots

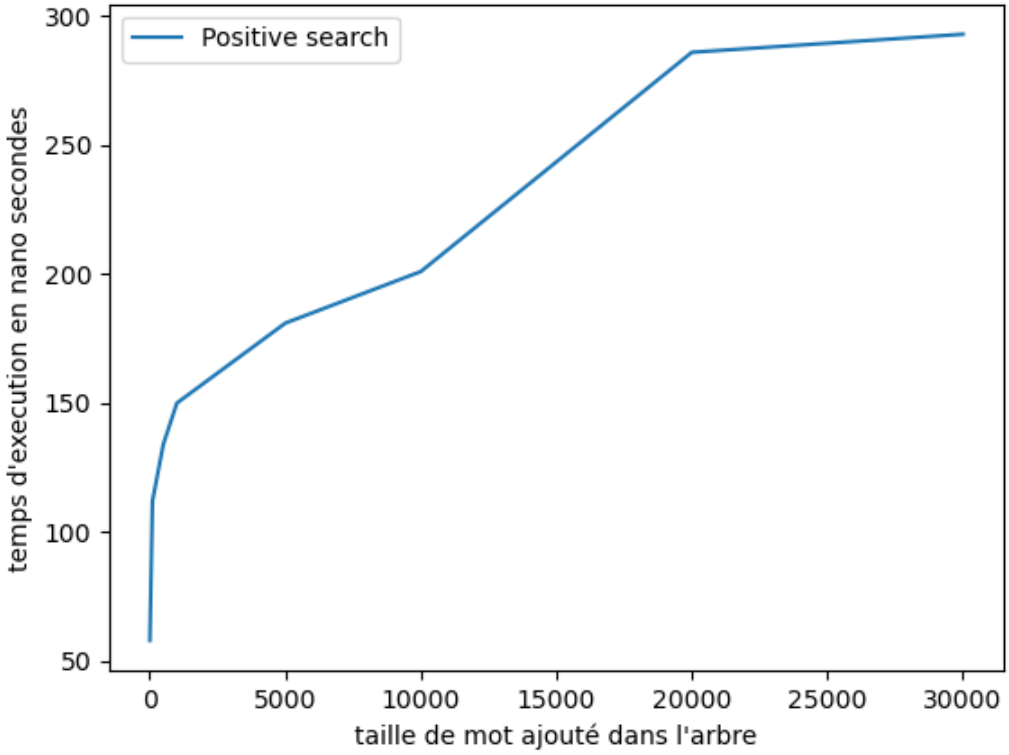

Figure 5: Benchmark de la méthode Recherche

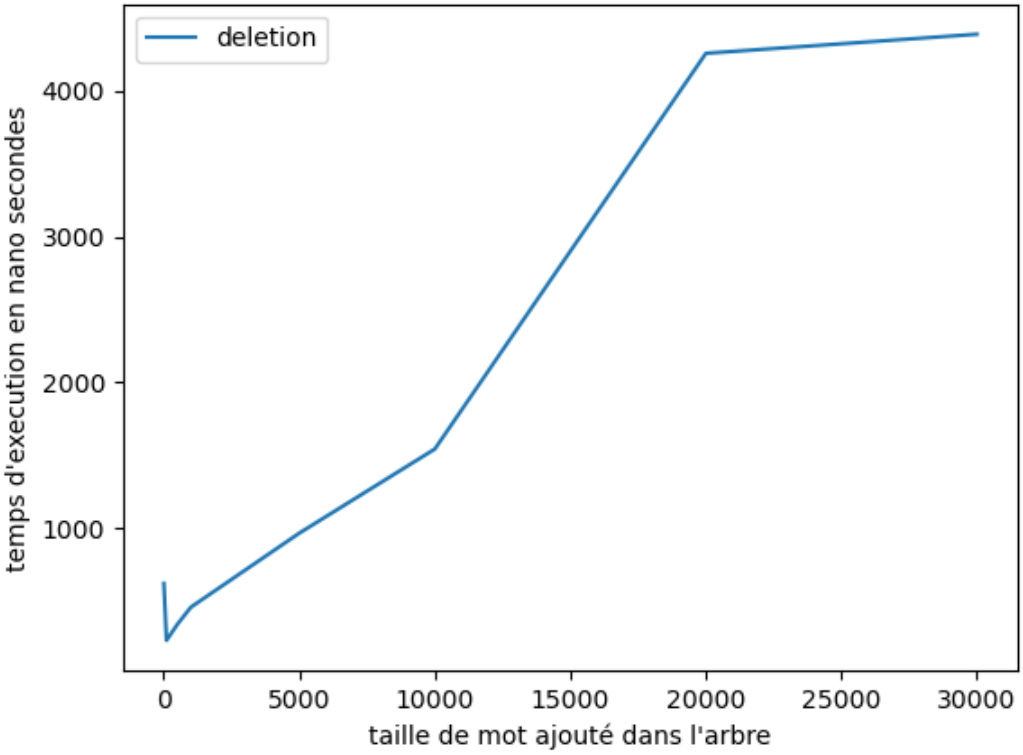

Figure 6: Benchmark de la méthode Suppression

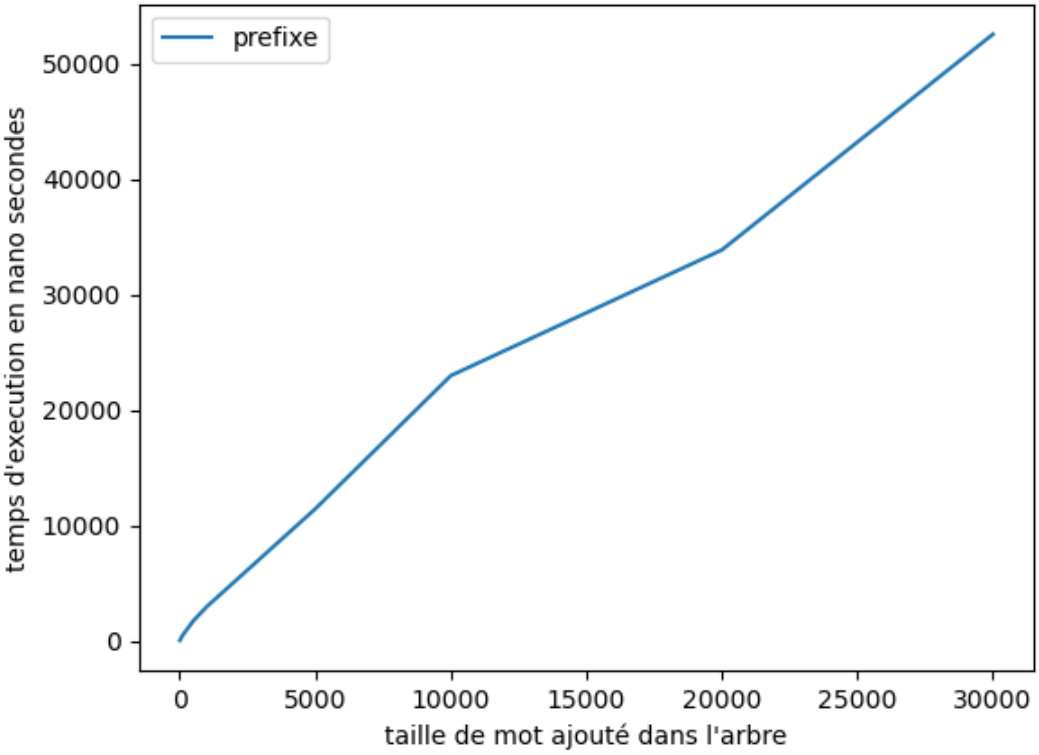

Figure 7: Benchmark de la méthode Prefixe

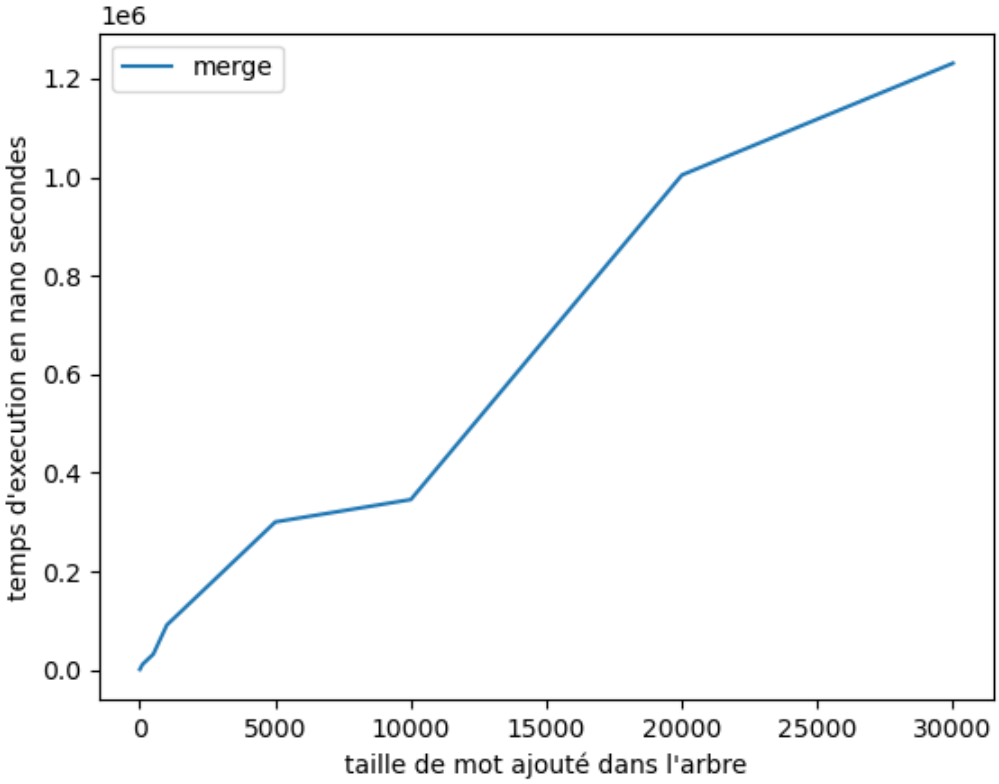

Figure 8: Benchmark de la méthode Fusion

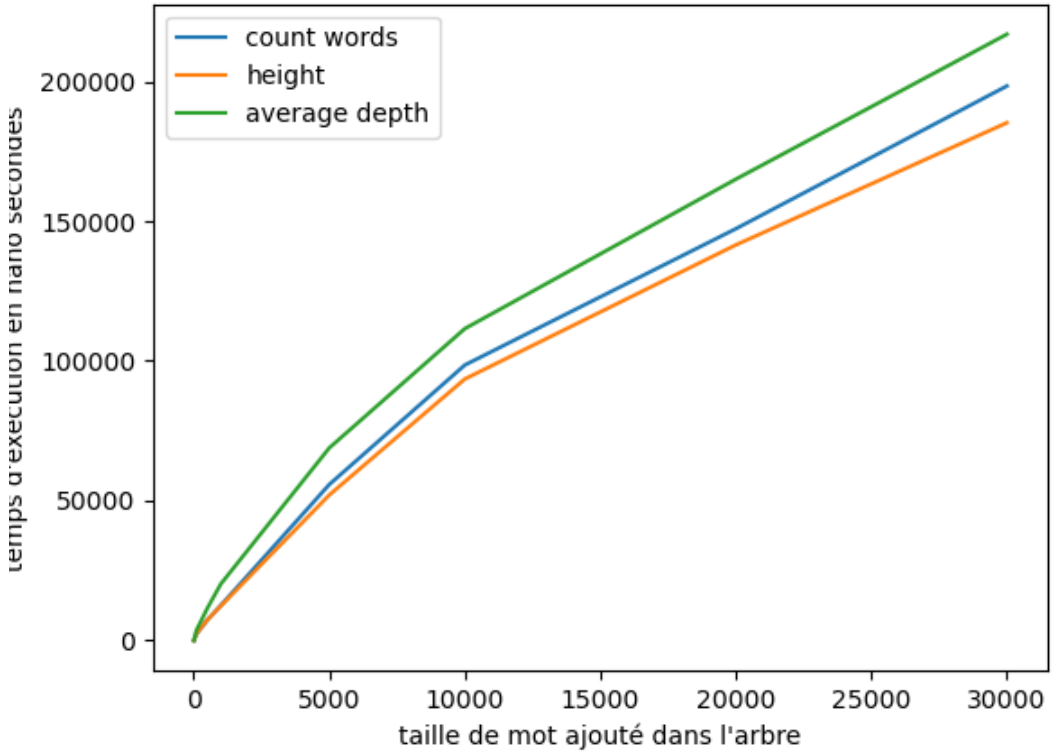

Figure 9: Benchmark des méthodes ComptageMot, Hauteur, ProfondeurMoyenne

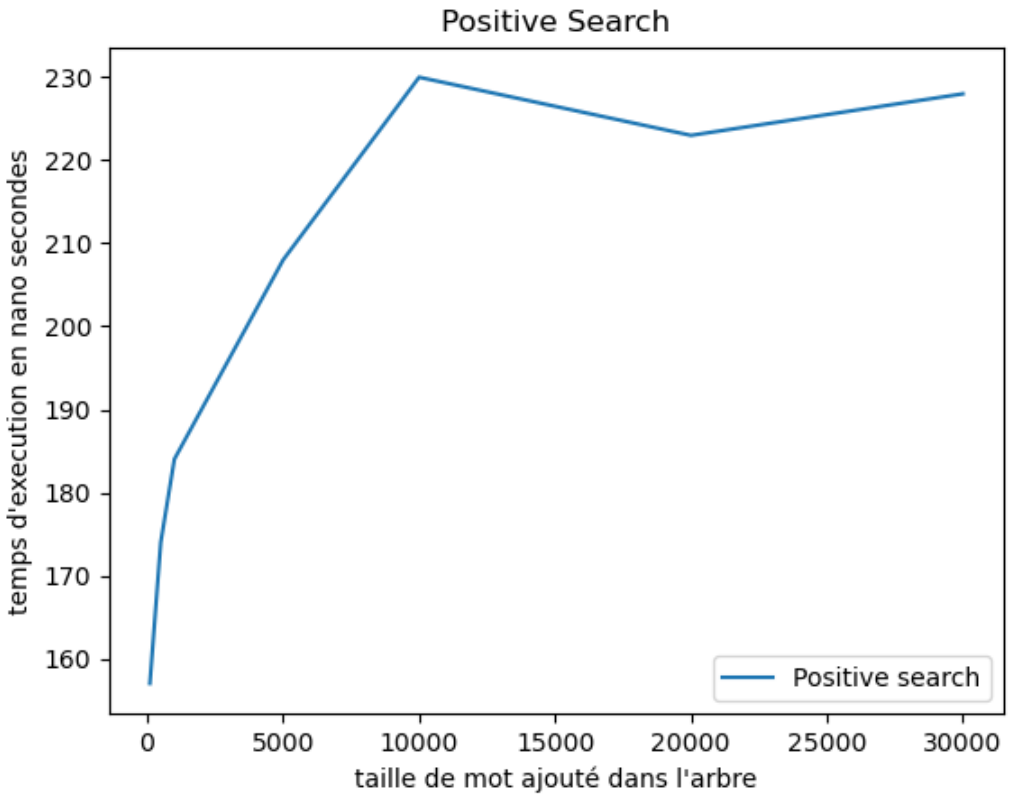

Figure 10: Benchmark de la méthode Recherche

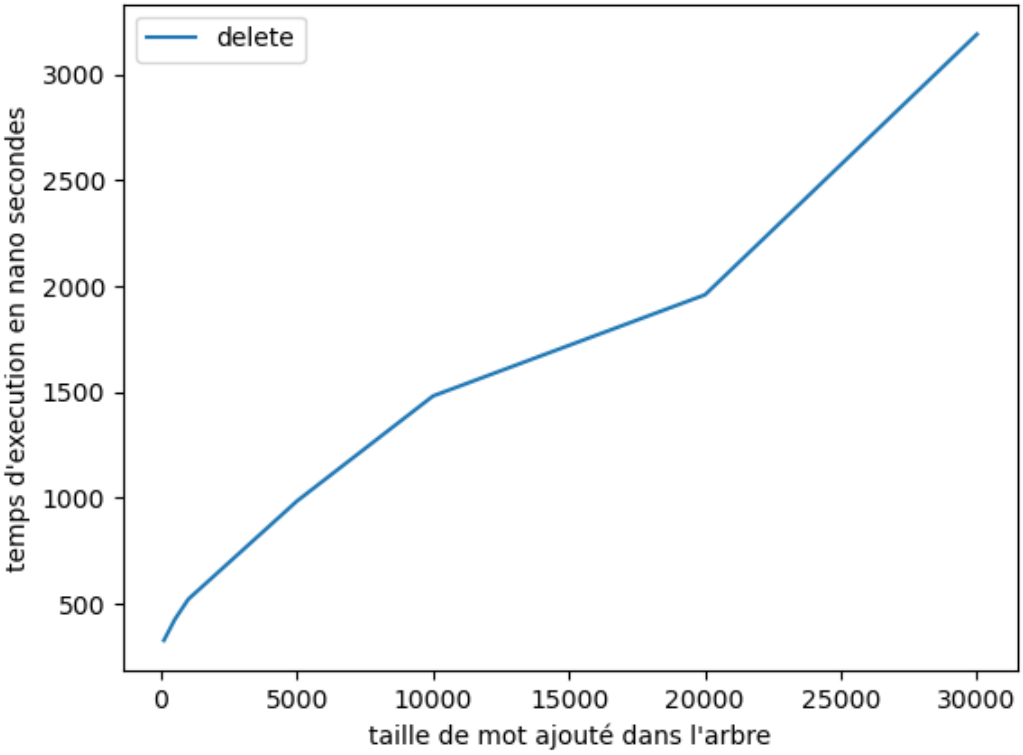

Figure 11: Benchmark de la méthode Suppression

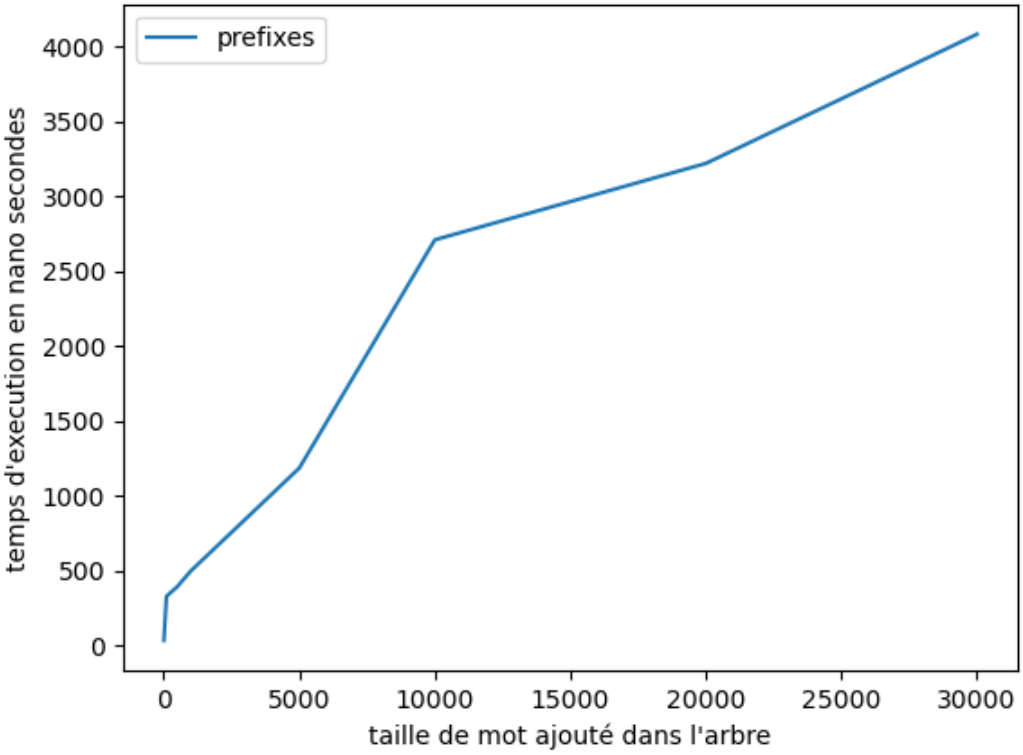

Figure 12: Benchmark de la méthode Prefixe

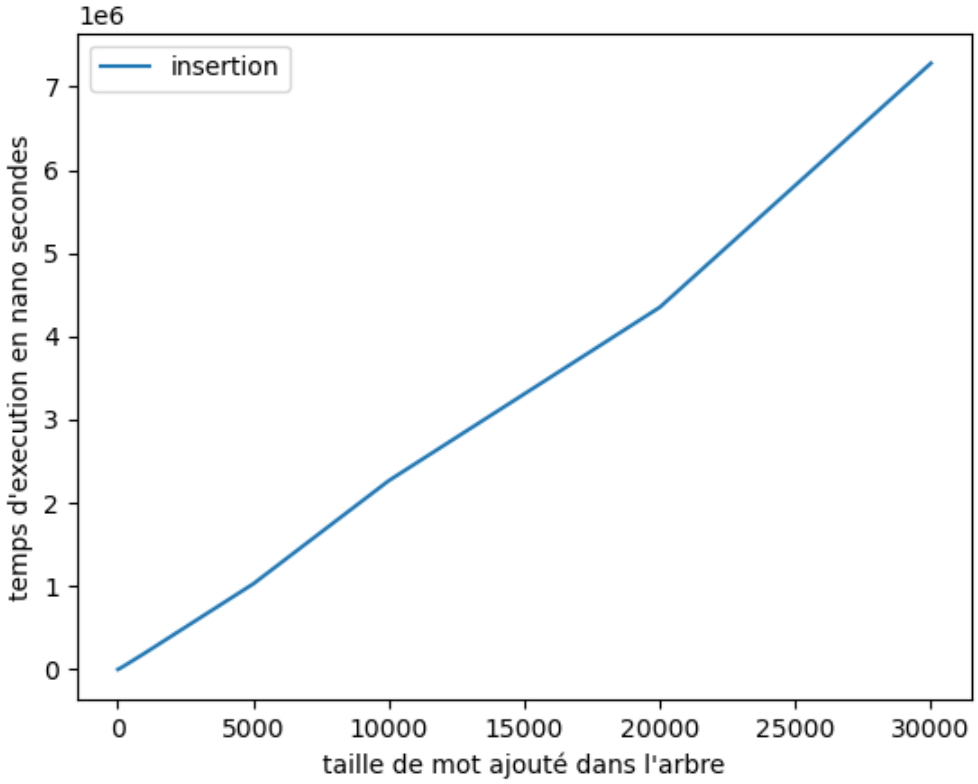

Figure 13: Benchmark de la méthode Insertion sans équilibrage

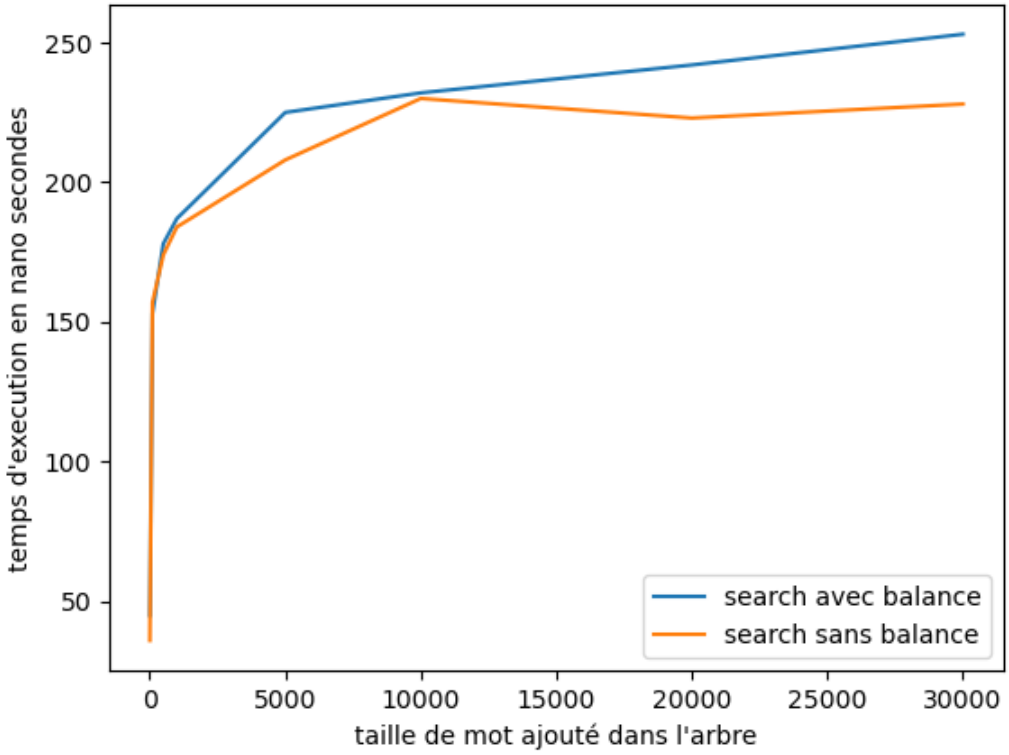

Figure 14: Benchmark Comparaison de la recherche

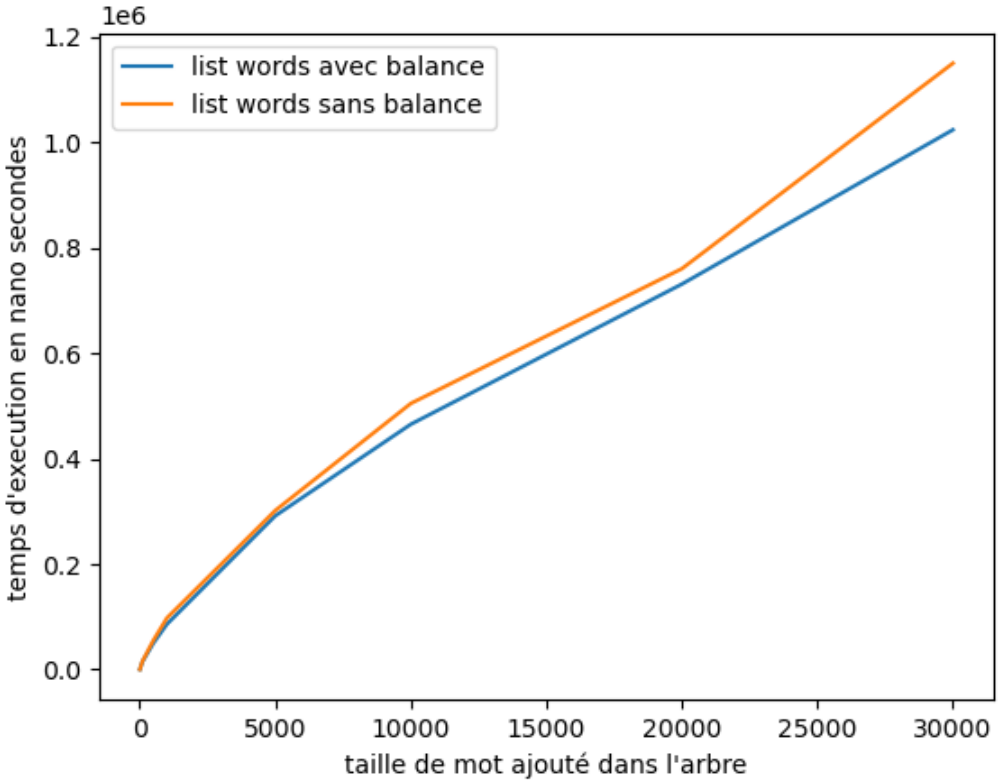

Figure 15: Benchmark Comparaison de la methode list words

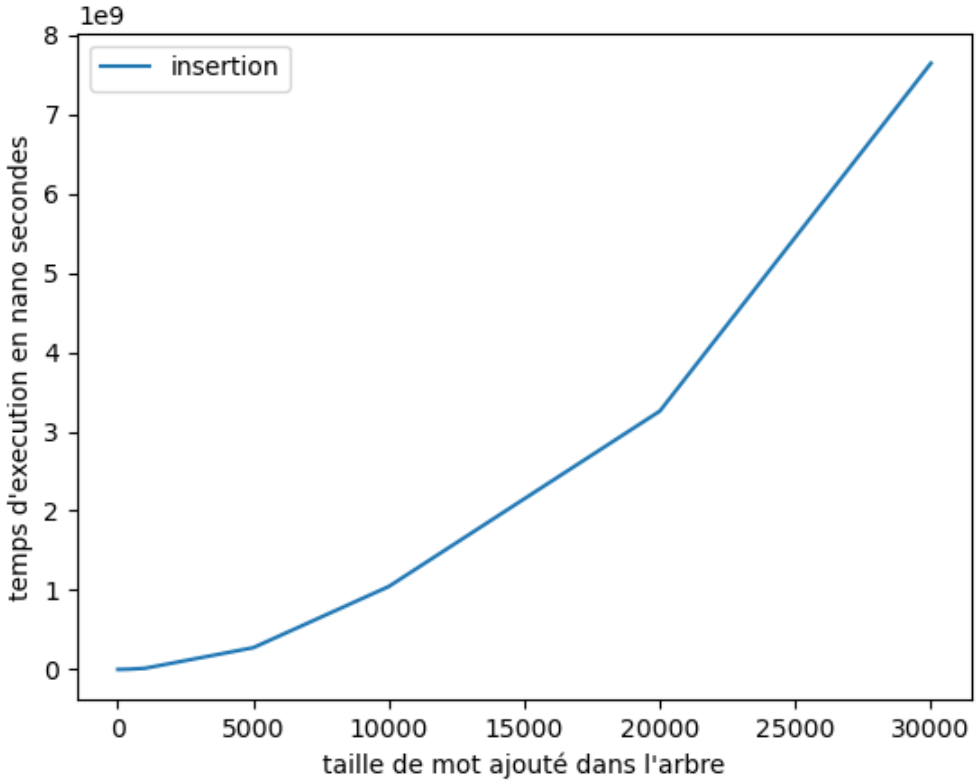

Figure 16: Benchmark de la méthode insertion avec équilibrage