Rapport de Projet ALGAV: Tries

Jean-Charles KAING 3808150

Shiyao CHEN 28707756

Décembre 2024

Table des matières

1 Introduction

1.1 Contexte et Motivation

Avec la croissance rapide des volumes de données, la demande pour des solutions efficaces de stockage et de recherche de mots devient de plus en plus importante. Par exemple, une recherche rapide de définitions de mots et une saisie semi-automatique, une indexation et une recherche efficaces de grandes quantités de données textuelles, et bien plus encore.

Afin de résoudre ces problèmes, nous avons choisi l'arbre de `Trie`. Il s'agit d'une structure de données spéciale qui permet de stocker et de rechercher efficacement des chaînes et qui est particulièrement adaptée à la représentation de dictionnaires de mots.

1.2 Objectif du projet

L'équipe de développement du projet est composée de Shiyao Chen et de Jean-Charles Kaing.

L'objectif du projet est d'implémenter deux structures Trie concurrentes (`Patricia Trie` et `Trie hybride`) et leurs fonctions en utilisant le langage C. Ensuite, nous analyserons les avantages et les inconvénients des deux modèles en termes de performance en les comparant expérimentalement

1.3 Représenter un dictionnaire de mots

Le Trie est une structure de données arborescente principalement utilisée pour stocker un ensemble de mots, comme dans un dictionnaire. Chaque nœud de l'arbre représente un caractère ou une chaîne de caractères ayant un préfixe commun, et les mots sont représentés par les chemins parcourus depuis la racine jusqu'aux feuilles.

Ensuite, nous vous présenterons les deux structures de Trie différentes : `les Patricia-Tries` et les `Tries Hybrides`.

1.3.1 Structure 1 : Patricia-Tries

L'objectif des arbres PATRICIA est de réduire la taille des R-tries tout en conservant une recherche efficace. Pour ce faire, au lieu d'associer chaque nœud interne à une seule lettre, un nœud représente la plus longue sous-chaîne commune à plusieurs mots, permettant ainsi de regrouper plusieurs branches en une seule et d'optimiser la structure.

Pour réprésenter l'arbre PAT : - Chaque nœud possède une clé qui représente le plus long préfixe commun aux mots de son sous-arbre, une valeur associée qui indique le nombre de clés associées et un pointer fils pointe vers un tableau de fils représentant les branches suivantes du Trie. - un PAT représente l'arbre complet sous la forme d'un tableau de nœuds.

```c
typedef struct PAT PAT;

typedef struct _Node{
    char* cle; // le plus long prefixe commun
    int valeur; //nb de cle;
    PAT* fils;
} Node;

typedef struct PAT{
    Node** node;
} PAT;
```

En C, le caractère `'\0'` est généralement utilisé pour indiquer la fin d'une chaîne de caractères. Cependant, dans l'implémentation d'un arbre de dictionnaire (`patricia_trie`), pour éviter tout conflit avec ce caractère de fin de chaîne, nous choisissons d'utiliser le caractère `SPACE` (ASCII 32 ou `'\x20'`), qui est peu courant utilisé dans un mot, afin de marquer la fin du mot. Ce choix permet d'assurer qu'aucun mot valide ne sera confondu avec le marqueur de fin.

```c
#define END_OF_WORD "\x20" // terminateur de mot d'un PAT
```

Les primitives de base concernant les `Patricia-tries` :

`PAT* PATVide()` qui crée un PAT vide.

`int EstVide(PAT* A)` qui renvoie 1 ssi l'arbre PAT est vide, 0 sinon.

`PAT* PATCons(Node* n)` qui construit un arbre avec un seul nœud.

`char* Rac(Node* A)` qui renvoie la clé du noeud.

`int Val(Node* A)` qui renvoie le nb associe la clé du noeud.

`int estPrefixe (char*c, char* m)` qui verifie c est préfixe de m, si oui renvoie 1, sinon 0.

`int prefixe(char* c, char* m)` qui renvoie la longeur de préfixe commun de c et m.

`void ajouter_fils(Node* A, Node* fil)` qui ajoute un enfant à un nœud.

`void ajouter_racine(PAT** P, Node* r)` qui ajoute une racine de PAT.

`void PATinsertion(PAT** A, char* m)` qui insère un mot dans l'arbre.

`void libererPAT(PAT* A)` qui libère la mémoire de l'arbre.

```
(A , 1)
(a, 0)
  (ppel , 1)
  ( , 1)
(c, 0)
  (hacune , 1)
  (i , 1)
  (lavier , 1)
  (o, 0)
    (eur , 1)
    (nnait , 1)
(d, 0)
  (actylo, 0)
    (graphie , 1)
    ( , 1)
  (e, 0)
    (s, 0)
      (sous , 1)
      ( , 1)
    ( , 1)
  (u , 1)
(e, 0)
  (crire , 1)
  (lle , 1)
    ...
}
```

Listing 1: pat.json

En prenant l'exemple de notre projet, nous obtiendrons l'arbre ci-dessus.

1.3.2 Structure 2 : Tries Hybrides

Le trie hybride est la seconde structure utilisée pour représenter un dictionnaire de mots. Pour représenter cet arbre ternaire - dont chaque noeud contient un caractère, une valeur (-1 si nulle), et trois pointeurs vers Inf, Eq et Sup, ainsi que la hauteur (notamment pour la partie 6), nous avons défini la structure TrieH et les primitives de tries hybrides dans le fichier `src/tries_hybrides/tries_hybrides.h`

```
typedef struct TrieH {
    char l; // lettre
    int v;  // valeur
    struct TrieH *inf; // sous-arbre gauche
    struct TrieH *eq;  // sous-arbre central
    struct TrieH *sup; // sous-arbre droite
    int hauteur; // hauteur
} TrieH;
```

Les primitives sur les clés des `tries hybrides` sont :
`char prem(char* c)` qui renvoie la première lettre de la chaine de caractères
`char* reste(char* c)` qui renvoie le reste de la chaine de caractères privée de la première lettre

Les primitives sur les tries hybrides sont :
`TrieH* TrieHybride(char l,TrieH* inf,TrieH* eq,TrieH* sup,int v)` qui construit une trie hybride
`TrieH* TH_Vide()` qui renvoie une trie hybride vide
`int EstVide(TrieH* A)` qui renvoie 1 ssi le trie hybride A est vide, 0 sinon
`char Rac(TrieH* A)` qui renvoie le caractère de la racine du trie hybride
`int Val(TrieH* A)` qui renvoie l'entier de la racine du trie hybride, -1 sinon
`TrieH* Inf(TrieH* A)` qui renvoie une copie du sous-arbre gauche de A
`TrieH* Eq(TrieH* A)` qui renvoie une copie du sous-arbre central de A
`TrieH* Sup(TrieH* A)` qui renvoie une copie du sous-arbre droite de A

`void majHauteur(TrieH* A)` met à jour la hauteur du noeud (rajouté 3.8)
`TrieH* TH_Ajout(char* c, TrieH* A, int v)` renvoie le trie hybride resultant de l'insertion de c dans A
`void free_TH(TrieH* A)` libère la mémoire allouée pour un noeud de trie hybride
`void free_chaine(char* chaine)` libère la mémoire allouée pour une chaîne de caractères.
Ces deux dernières fonctions sont importantes afin de libérer la mémoire lorsqu'elle n'est plus utilisé, notamment lorsque l'on crée un dictionnaire comportant beaucoup de mots tel que ceux de l'oeuvre de Shakespeare.

2 Fonctions avancées et complexes

2.1 Fonctions avancées et leurs complexités théoriques.

Nous vous présentent des fonctions avancées pour les deux structures Patricia-Trie et Trie Hybride.

```
/* Fonction permet de vérifier si un mot existe dans
l'arbre Trie, si oui renvoie 1, sinon 0.  */
Recherche(arbre, mot) -> bool

/* Fonction compte le nombre de mots stockés dans l'arbre. */
ComptageMots(arbre) -> int

/* Fonction génère une liste des mots présents dans
l'arbre, triée par ordre alphabétique. */
ListeMots(arbre) -> liste[mots]

/* Fonction compte le nombre de pointeurs
qui pointent vers NULL dans l'arbre. */
ComptageNil(arbre) -> int

/* Fonction calcule de la hauteur de l'arbre. */
Hauteur(arbre) -> int

/* Fonction calcule de la profondeur moyenne de l'arbre. */
ProfondeurMoyenne(arbre) -> int

/* Foction permet de déterminer combien de mots du
dictionnaire commencent par un préfixe donné. */
Prefixe(arbre, mot) -> int

/* Fonction permet de supprimer un mot de l'arbre. */
Suppression(arbre, mot) -> arbre
```

TABLE 1 – Comparaison des complexités entre Patricia-Trie et Trie Hybride

Fonction	**Patricia-Trie**	**Trie Hybride**
Recherche	$O(L * n)$	$O(n)$
Comptage des mots	$O(n)$	$O(n)$
Liste des mots	$O(L * n)$	$O(n \cdot L)$
Comptage des pointeurs Nil	$O(n)$	$O(n)$
Hauteur	$O(n)$	$O(n)$
Profondeur moyenne	$O(n)$	$O(n)$
Préfixe d'un mot	$O(L * n)$	$O(n \cdot L)$
Suppression	$O(L * n)$	$O(n + m)$

Légende :

— L : longueur de mot

— n : Nombre de nœuds dans l'arbre.

2.2 Fonction complexe

On donne un pseudo-code de notre fonction `PATFusion`, qui fusionne deux `Patricia-Tries` en un seul.

La boucle extérieure itère sur n_2 nœuds, et à chaque itération, des comparaisons sont effectuées sur n_1 nœuds, avec un coût de $O(L_{\max})$ par comparaison. Ainsi, la complexité temporelle globale est $O(n_2 \cdot n_1 \cdot L_{\max})$, où n_1 et n_2 représentent les tailles des deux tries, et $L_{\max}$ est la longueur maximale des mots.

Algorithm 1: PATfusion

Input: Deux arbres PAT : A et B
Output: Arbre PAT fusionné A

```
1  if A est vide then
2  |  return B
3  end
4  else if B est vide then
5  |  return A
6  end
7  foreach chaque nœud bNode dans B do
8  |  cle_b ← clé de bNode;
9  |  if cle_b n'existe pas dans A then
10 |  |  Ajouter bNode comme racine dans A;
11 |  end
12 |  else
13 |  |  foreach chaque nœud aNode dans A do
14 |  |  |  cle_a ← clé de aNode;
15 |  |  |  if cle_a commence par cle_b (ou vice versa) then
16 |  |  |  |  if cle_a et cle_b sont identiques then
17 |  |  |  |  |  if les deux clés sont des feuilles then
18 |  |  |  |  |  |  Additionner les valeurs de aNode et bNode;
19 |  |  |  |  |  end
20 |  |  |  |  |  else
21 |  |  |  |  |  |  Fusionner leurs sous-arbres;
22 |  |  |  |  |  end
23 |  |  |  |  end
24 |  |  |  |  else
25 |  |  |  |  |  len_com ← longueur du préfixe commun entre cle_a
   |  |  |  |  |    et cle_b;
26 |  |  |  |  |  pref_com ← préfixe commun;
27 |  |  |  |  |  a_rest ← partie restante de cle_a;
28 |  |  |  |  |  b_rest ← partie restante de cle_b;
29 |  |  |  |  |  F ← nouveau nœud avec clé a_rest et sous-arbres de
   |  |  |  |  |    aNode;
30 |  |  |  |  |  G ← nouveau nœud avec clé b_rest et sous-arbres de
   |  |  |  |  |    bNode;
31 |  |  |  |  |  Créer un sous-arbre fusionné entre F et G;
32 |  |  |  |  |  Mettre à jour aNode avec clé pref_com, valeur 0, et
   |  |  |  |  |    sous-arbre fusionné;
33 |  |  |  |  end
34 |  |  |  end
35 |  |  end
36 |  end
37 end
38 return A
```

Pour implémenter une fonction `TH_AjoutEquilibre`, qui maintient un trie hybride équilibré, nous avons choisi de rééquilibrer après chaque insertion. Donc, la fonction commence par insérer le mot à l'aide de `TH_Ajout`, puis vérifie l'équilibre du trie hybride résultant, et le rééquilibre si besoin à l'aide des rotations gauche et droite.

La fonction utilisera les primitives `majHauteur` qui met à jour la hauteur de la structure, `reequilibrer` qui utilise les primitives `rotationGauche` et `rotationDroite` qui permettent de renvoyer les tries hybrides équilibrés.

3 Format du rendu du code

Pour le format de rendu du code, on a écrit des scripts bash qui appellent des scripts en C.

Pour gérer les fichiers JSON pour les `Patricia-tries`, nous avons utilisé la bibliothèque cJSON.h et nous avons défini les fonctions suivantes dans "src/Patricia-Tries/patricia_json.h" :

```c
/* Ecrit le patricia dans le fichier format JSON  */
int ecrire_patricia(char* namefile, PAT* arbre);

/*Convertir le PAT dans le format JSON*/
cJSON* node_to_json(Node* node);
cJSON* pat_to_json(PAT* pat);

/* Construit le patricia depuis le format JSON */
PAT* json_to_pat(cJSON* json_node);
cJSON* node_to_json(Node* node);
```

Pour gérer les fichiers JSON avec les tries hybrides, nous avons écrit les fonctions suivantes dans “src/tries_hybrides/ecriture_lecture.h” :

```c
/* Ecrit le trie hybride dans le format JSON */
int ecrire_trie(FILE* file, TrieH* arbre, int tabulation);

/* Construit le trie hybride depuis le format JSON */
TrieH* charger_trie(char *content, int *index);
```

4 Étude expérimentale

Dans cette partie, nous utilisons des données extraites des œuvres de Shakespeare pour tester les performances des deux structures. Ces données contiennent un grand nombre de mots de longueurs variées, ce qui les rend idéales pour évaluer l'efficacité des structures de type Trie.

Pour extraire les données, nous avons utilisé la structure `Word` et défini les fonctions suivantes dans `"src/experimentale/experimentale.h"`.

```c
typedef struct words{
    char* data;
    struct words* suiv;
}Words;

/* Crée une liste de mots contenant un seul élément */
Words* create_list_word(char* cle);

/* Insère un mot dans une liste de mots existante */
Words* insert_word_in_List(char* cle, Words* lcle);

/* Insère un mot dans la liste uniquement
s'il n'existe pas déjà */
Words* insertWordsNotExist(char* cle, Words* lcle);

/* Lit les mots à partir d'un fichier contenant
une œuvre de Shakespeare */
Words* read_ouvre_Shakespeare(char* nomFichier);

/* Lit les mots à partir de plusieurs fichiers dans
un dossier contenant des œuvres de Shakespeare */
Words* read_Files_Shakespeare(char* nomDossier);

/* Écrit la liste des mots dans un fichier */
void eciture_words(Words* words);
```

On extrait l'ensemble des mots des œuvres de Shakespeare en utilisant la fonction `read_Files_Shakespeare`, puis on écrit ces mots dans le fichier `"words.txt"`. Ensuite, on peut est exécuter un script Bash avec la commande `./inserer x words.txt` pour construire le `Patricia-Trie` et le `Trie hybride`, dont les résultats sont sauvegardés respectivement dans les fichiers `"pat.json"` et `"trie.json"`. Les tests sont lancés à partir des exécutables `./main_etude_patricia` et `./main_etude_trie`.

4.1 Mesure du temps d'exécution

On utilise de la bibliothèque `<time.h>` pour mesurer le temps en millisecondes pour chaque fonction. Et On a défini plusieurs critères pour comparer les deux structures (`Patricia-Trie` et `Trie Hybride`) :

1. **Temps de construction des structures** :
   Temps nécessaire pour insérer un ensemble complet de mots dans chaque structure.
2. **Temps d'insertion d'un mot** :
   Temps pour insérer un nouveau mot dans la structure.
3. **Temps de suppression d'un mot** :
   Temps pour supprimer un ensemble de mots des structures.
4. **Hauteur de l'arbre** :
   Hauteur de chacune des structures.
5. **Profondeur moyenne des feuilles** :
   Profondeur des structures

Et les résultats obtenus sont présentés dans le tableau ci-dessous pour faciliter la comparaison.

TABLE 2 – Comparaison des temps d'exécuter entre `Patricia-Trie` et `Trie Hybride`

Critère	**Patricia-Trie**	**Trie Hybride**
Temps de construction (s)	0.0181330	0.0744010
Temps d'insertion d'un mot (s)	0.0000020	0.000020
Temps de suppression(s)	0.1040390	0.0422650
Hauteur de l'arbre	11	1986289939
Profondeur moyenne	5	19

En comparant les résultats obtenus, nous observons les points suivants :

- Construction :

Le `Patricia-Trie` est légèrement plus rapide que le `Trie Hybride`. Cela s'explique par la compression des chemins dans le `Patricia-Trie`, qui réduit le nombre de nœuds à parcourir lors de l'insertion des mots.

- Insertion :

Les deux structures nécessitent un temps comparable pour l'insertion, proportionnel à la longueur du mot (`O(L)`). Cependant, le `Patricia-Trie` reste légèrement plus rapide grâce à sa structure optimisée pour les préfixes communs, ce qui réduit les opérations nécessaires dans certains cas.

- Suppression :

La suppression d'un ensemble de mots est plus coûteuse dans le `Patricia-Trie`, en raison de ses caractéristiques structurelles. Plusieurs mots y partagent des préfixes dans un même nœud, ce qui complexifie les opérations. Lorsqu'un mot est supprimé, il est souvent nécessaire de réajuster ou de fusionner les nœuds

restants afin de maintenir la structure compressée. Par ailleurs, si le mot supprimé est le seul représenté dans un chemin compressé, le nœud correspondant doit être supprimé et les nœuds adjacents doivent être réorganisés, nécessitant une traversée et une modification de plusieurs niveaux de l'arbre.

- Hauteur :

La hauteur du `Trie Hybride` est plus élevée, car chaque nœud contient une seule lettre. Plus le nombre de mots insérés augmente, plus le nombre de nœuds croît proportionnellement. De plus, chaque nœud ne peut avoir que trois sous-fils (gauche, milieu et droit), ce qui allonge la hauteur de l'arbre. En revanche, le Patricia-Trie conserve une hauteur plus faible grâce à sa compression, qui combine les chemins partagés en un seul nœud, réduisant ainsi la profondeur globale.

- Profondeur :

La profondeur moyenne est plus grande pour le `Trie Hybride`, car chaque nœud possède au maximum trois sous-fils (gauche, milieu, droit). Cela nécessite davantage de nœuds pour stocker les lettres, augmentant ainsi la profondeur de l'arbre. En revanche, le `Patricia-Trie`, avec sa structure plus compacte, permet de réduire la profondeur, ce qui impacte positivement les performances des opérations de recherche et de parcours.

Enfin, le `Trie Hybride équilibré` sera plus efficace qu'un `Trie Hybride non équilibré`, mais restera moins efficace qu'un `Patricia-Trie`.

4.2 Compteur de complexité

Nous ajoutons un compteur dans les fonctions pour mesurer expérimentalement la complexité des opérations avancées et complexes. Dans la structure `Patricia-trie`, les fonctions d'`insertion`, de `suppression` et de `recherche` sont mesurées en termes de nombre de comparaisons, tandis que les fonctions `ComptageMotsDansPAT`, `ComptageNilDansPAT`, `ListeMotsdansPAT`, `HauteurPAT` et `ProfondeurMoyennePAT` sont mesurées en fonction du nombre d'appels récursifs.

Pour un `Trie Hybride`, nous ajoutons une variable globale `cpt_fct` et incrémentons les fonctions avancées implémentées. Nous récuperons ensuite à l'aide de `getCptFct` et mettrons le compteur à 0 à l'aide de `setCptFct(0)`.

4.2.1 Analyser des résultats

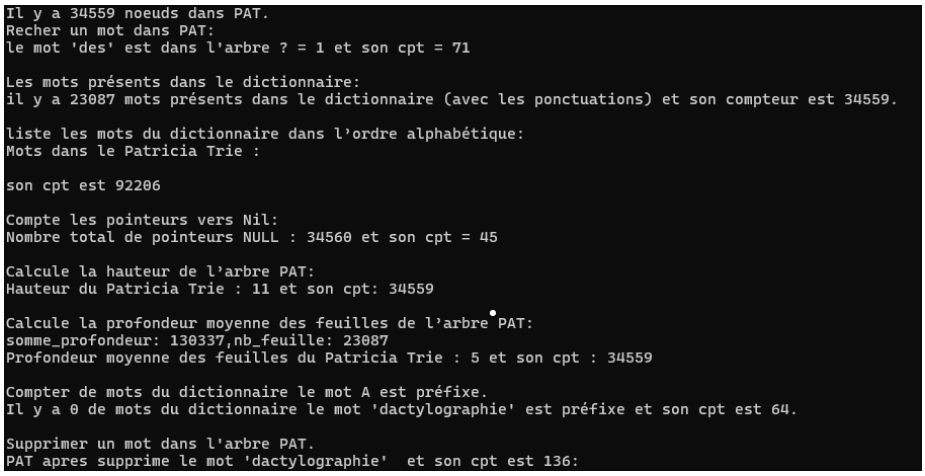

FIGURE 1 – Résultats des compteurs de fonctions de Patricia-Tries

En observant les résultats, nous pouvons constater que la complexité des fonctions `ComptageMotsDansPAT`, `HauteurPAT` et `ProfondeurMoyennePAT` est bien conforme à la complexité théorique. Cependant, un problème est apparu avec la fonction `ComptageNilDansPAT`, dont les résultats sont bien plus faibles que ce qui était attendu selon la complexité théorique.

Cela met en évidence un problème majeur dans cette conclusion, à savoir que nous n'avons testé qu'une seule donnée, sans disposer d'un ensemble de données suffisamment large pour effectuer des tests représentatifs. Par conséquent, les résultats obtenus sont biaisés et ne permettent pas de conclure sur la complexité des fonctions, notamment pour des cas plus complexes tels que $O(nlogn)$.

Il est donc essentiel d'utiliser un plus grand volume de données pour mener des tests plus fiables et obtenir des résultats significatifs.

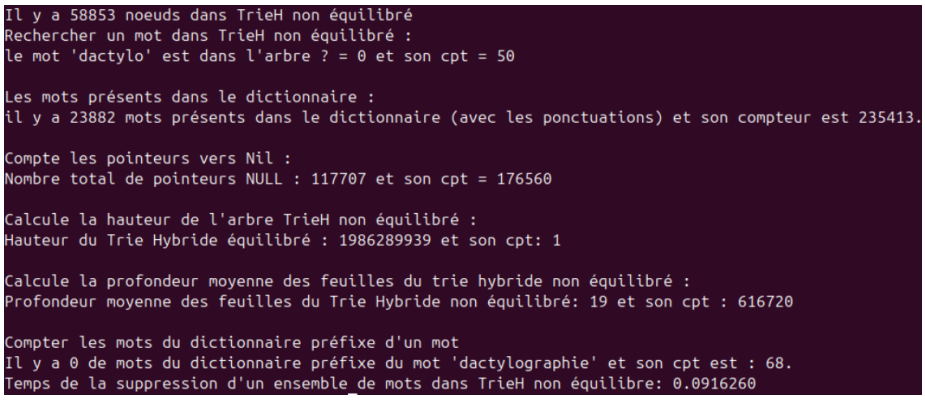

FIGURE 2 – Résultats des compteurs de fonctions de Trie Hybride non équilibré

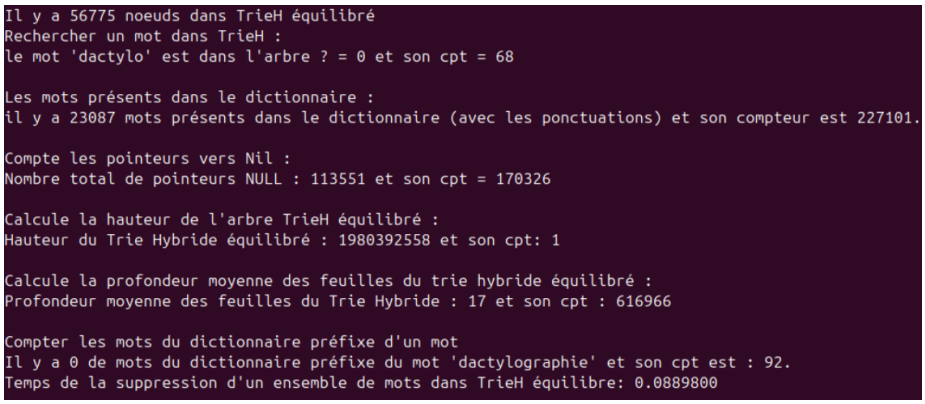

FIGURE 3 – Résultats des compteurs de fonctions de Trie Hybride équilibré

On constate que le trie hybride équilibré ou non reste dans le même ordre de grandeur. Ainsi, certaines fonctions sont relativement plus efficace que que pour l'autre structure et inversement.

4.3 Comparaison les temps de construction des arbres

En comparant le temps de construction des PAT en ajoutant successivement chaque mot ou en fusionnant les arbres de chaque œuvre, j'ai constaté que le temps de construction des PAT était de 0.0164590 secondes, tandis que le temps de fusion des PAT était de 0.0856450 secondes. Cependant, ces résultats diffèrent légèrement de ce que nous avons appris. Selon la complexité théorique, le temps nécessaire pour ajouter successivement les mots devrait être plus long que celui

pour fusionner les arbres. En vérifiant les données utilisées, je me suis rendu compte qu'elles comprenaient tous les mots différents.

J'ai donc recalculé le temps nécessaire pour ajouter successivement tous les mots de l'œuvre de Shakespeare dans l'arbre, et j'ai obtenu un temps de construction de 0.3994350 secondes, tandis que le temps de fusion des PAT était de 0.0802520 secondes, ce qui est relativement plus élevé. Nous avons ainsi observé que, pour un petit nombre de mots, l'insertion est plus rapide, tandis que pour de grandes quantités de données et de nombreux préfixes communs, la fusion des arbres est plus rapide.

Pour le trie hybride, on constate que que les temps de constructions sont beaucoup plus élevé.

5 Conclusion

En conclusion , ce projet a permis de réaliser une comparaison complète des deux structures de tries et d'étudier leur performance.

Le `Patricia-Trie` s'avère plus efficace en termes de consommation de mémoire et de performance pour des ensembles de données volumineux et contenant de nombreux préfixes communs.

Le `Trie Hybride` équilibré ou non reste en deça de `Patricia-Trie`.