PROJET ALGAV

Devoir de Programmation : Tries

Mengxiao LI Xue YANG

PLAN

1. STRUCTURES DES TRIES
2. FONCTIONS DES TRIES
3. COMPLEXITÉS
4. ANALYSE EXPÉRIMENTALE
5. CONCLUSION

STRUCTURES DES TRIES

PATRICIA-TRIES

```
class PatriciaTrieNode:  9 usages
    def __init__(self, label=""):
        self.label = label
        self.children = {}
```

Label: Chaîne de caractères représentant le plus long préfixe

Children: Un dictionnaire qui stocke les sous-nœuds. Les clés sont les premiers caractères des label des sous-nœuds, et les valeurs sont les nœuds enfants

```
end_marker = chr(0x00)
```

La fin d'un mot

STRUCTURES DES TRIES

PATRICIA-TRIES

Les Fonctions

- inserer (arbre, mot)
- recherche (arbre)
- comptageMots(arbre)
- listMots(arbre)
- ProfondeurMoyenne(arbre)
- Prefixe(arbre, mot)
- Suppression(arbre, mot)
- fusion(arbreA, arbreB)

Les Primitives de Base

- find_mots_prefix(mot1,mot2)
- json_to_patricia_trie(data)
- to_dict(arbre)
- display_as_json(self)

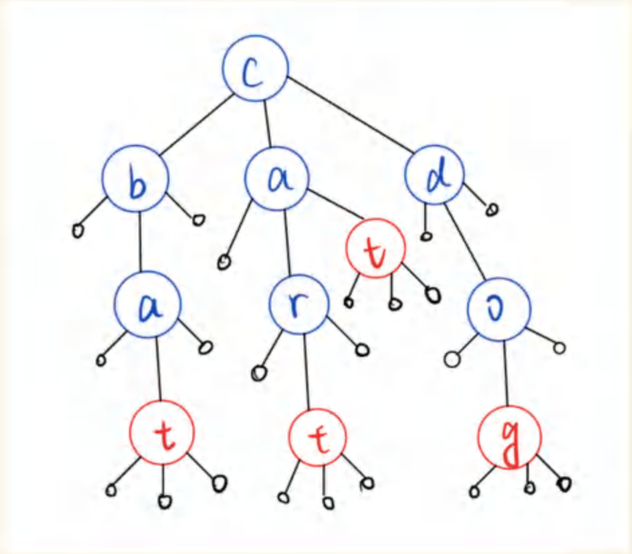
Concepts de base:

- Recherche de préfixe
- Division de nœud
- Fusion de nœuds

STRUCTURES DES TRIES

TRIES HYBRIDES

WORDS = ["CAR", "CAT", "CART", "DOG", "BAT"]



- **GAUCHE : INFÉRIEURS AU CARACTÈRE COURANT**
- **MILIEU : PROCHAIN CARACTÈRE**
- **DROITE : SUPÉRIEURS**

REPRÉSENTÉ EN JSON

CHAQUE NŒUD DU JSON GÉNÉRÉ EST DÉCRIT PAR :

- **LE CARACTÈRE QU'IL CONTIENT (CHAR)**
- **UN MARQUEUR INDIQUANT S'IL S'AGIT DE LA FIN D'UN MOT (IS_END_OF_WORD)**
- **TROIS RÉFÉRENCES VERS SES SOUS-ARBRES (LEFT, MIDDLE,RIGHT)**

```json
{
    "char": "c",
    "is_end_of_word": false,
    "left": {
        "char": "b",
        "is_end_of_word": false,
        "left": null,
        "middle": {
            "char": "a",
            "is_end_of_word": false,
            "left": null,
            "middle": {
                "char": "t",
                "is_end_of_word": true,
                "left": null,
                "middle": null,
                "right": null
            },
            "right": null
        },
        "right": null
    },
```

STRUCTURES DES TRIES

TRIES HYBRIDES

PRIMITIVES DE BASE

- **Insertion(arbre, mot)** **arbre : Ajoute un mot dans le Trie.**
- **IsEmpty(arbre)** **booléen : Vérifie si le Trie est vide.**
- **to_json(arbre, chemin)** **void : Sauvegarde le Trie dans un fichier JSON.**
- **to_dict(arbre)** **dict : Convertit le Trie en un dictionnaire.**
- **from_dict(data)** **arbre : Reconstruit le Trie à partir d'un dictionnaire.**
- **from_json(chemin)** **arbre : Charge un Trie depuis un fichier JSON.**

Algorithm 1 Insertion(*arbre*, *mot*) → *arbre*

```
1:  function INSERTION(arbre, mot)
2:      arbre.root ← INSERTREC(arbre.root, mot, 0)
3:      return arbre
4:  end function
5:  function INSERTREC(node, mot, index)
6:      char ← mot[index]
7:      if node = null then
8:          node ← HYBRIDTRIENODE(char)
9:      end if
10:     if char < node.char then
11:         node.left ← INSERTREC(node.left, mot, index)
12:     else if char > node.char then
13:         node.right ← INSERTREC(node.right, mot, index)
14:     else
15:         if index + 1 = length(mot) then
16:             node.is_end_of_word ← true
17:         else
18:             node.middle ← INSERTREC(node.middle, mot, index + 1)
19:         end if
20:     end if
21:     return node
22: end function
```

exemple de base:

" A quel génial professeur de dactylographie sommes-nous redevables de la superbe phrase ci-dessous, un modèle du genre, que toute dactylo connaît par cœur puisque elle fait appel à chacune des touches du clavier de la machine à écrire ? "

FONCTIONS DES TRIES

TRIES HYBRIDES

FONCTIONS AVANCÉES

Algorithm 2 Recherche(arbre, mot) → booléen

```
1: function RECHERCHE(arbre, mot)
2:     return RECHERCHEREC(arbre.root, mot, 0)
3: end function
4: function RECHERCHEREC(node, mot, index)
5:     if node = null then
6:         return False
7:     end if
8:     char ← mot[index]
9:     if char < node.char then
10:        return RECHERCHEREC(node.left, mot, index)
11:    else if char > node.char then
12:        return RECHERCHEREC(node.right, mot, index)
13:    else
14:        if index + 1 = length(mot) then
15:            return node.is_end_of_word
16:        else
17:            return RECHERCHEREC(node.middle, mot, index + 1)
18:        end if
19:    end if
20: end function
```

Algorithm 3 ComptageMots(arbre) → entier

```
1: function COMPTAGEMOTS(node)
2:     if node = null then return 0
3:     end if
4:     count ← 0
5:     if node.is_end_of_word = true then
6:         count ← count + 1
7:     end if
8:     count ← count+ COMPTAGEMOTS(node.left)
9:     count ← count+ COMPTAGEMOTS(node.middle)
10:    count ← count+ COMPTAGEMOTS(node.right)
11:    return count
12: end function
```

Algorithm 4 ListeMots(arbre) → liste[mots]

```
1: function LISTE_MOTS(self)
2:     result ← []                          ▷ Initialiser une liste vide pour stocker les mots
3:     _LISTE_MOTS(self.root, "", result)   ▷ Appeler la fonction auxiliaire avec le racine
4:     return result
5: end function
6: function _LISTE_MOTS(self, node, prefix, result)
7:     if node = null then
8:         return
9:     end if
10:    _LISTE_MOTS(self, node.left, prefix, result)        ▷ Explorer le sous-arbre gauche
11:    if node.is_end_of_word = true then
12:        APPEND(result, prefix + node.char)              ▷ Ajouter le mot formé à la liste
13:    end if
14:    _LISTE_MOTS(self, node.middle, prefix + node.char, result)   ▷ milieu
15:    _LISTE_MOTS(self, node.right, prefix, result)                ▷ droit
16: end function
```

FONCTIONS DES TRIES

TRIES HYBRIDES

FONCTIONS AVANCÉES

Algorithm 5 Comptage des pointeurs NULL dans un Trie Hybride

```
1: function COMPTAGE_NIL(self)
2:     function _COMPTAGE_NIL(node)
3:         if node = null then
4:             return 0
5:         end if
6:         count ← 0
7:         if node.left = null then
8:             count ← count + 1
9:         end if
10:        if node.middle = null then
11:            count ← count + 1
12:        end if
13:        if node.right = null then
14:            count ← count + 1
15:        end if
16:        count ← count+ _COMPTAGE_NIL(node.left)
17:        count ← count+ _COMPTAGE_NIL(node.middle)
18:        count ← count+ _COMPTAGE_NIL(node.right)
19:        return count
20:    end function
21:    return _COMPTAGE_NIL(self.root)
22: end function
```

Algorithm 6 Calcul de la hauteur d'un Trie Hybride

```
1: function HAUTEUR(self)
2:     return _HAUTEUR(self.root)
3: end function
4: function _HAUTEUR(node)
5:     if node = null then
6:         return 0
7:     end if
8:     return 1 + max(_HAUTEUR(node.left), _HAUTEUR(node.middle), _HAUTEUR(node.right))
9: end function
```

Algorithm 7 Calcul de la profondeur moyenne des feuilles d'un Trie Hybride

```
1: function PROFONDEUR_MOYENNE(self)
2:     result ← {total_depth : 0, leaf_count : 0}
3:     _PROFONDEUR_MOYENNE(self.root, 0, result)
4:     if result.leaf_count = 0 then
5:         return 0
6:     end if
7:     return result.total_depth/result.leaf_count
8: end function
9: function _PROFONDEUR_MOYENNE(node, depth, result)
10:    if node = null then
11:        return
12:    end if
13:    if node.is_end_of_word = true then
14:        result.total_depth ← result.total_depth + depth
15:        result.leaf_count ← result.leaf_count + 1
16:    end if
17:    _PROFONDEUR_MOYENNE(node.left, depth + 1, result)
18:    _PROFONDEUR_MOYENNE(node.middle, depth + 1, result)
19:    _PROFONDEUR_MOYENNE(node.right, depth + 1, result)
20: end function
```

FONCTIONS DES TRIES

TRIES HYBRIDES

FONCTIONS AVANCÉES

Algorithm 8 Calculer le nombre de mots commençant par un préfixe donné

```
1: function PREFIXE(self, prefix)
2:     return _PREFIXE(self.root, prefix, 0)
3: end function
4: function _PREFIXE(node, prefix, index)
5:     if node = null then
6:         return 0
7:     end if
8:     if index ≥ len(prefix) then
9:         return COMPTAGEMOTS(node)
10:    end if
11:    char ← prefix[index]
12:    if char < node.char then
13:        return _PREFIXE(node.left, prefix, index)
14:    else if char > node.char then
15:        return _PREFIXE(node.right, prefix, index)
16:    else
17:        if index + 1 = len(prefix) then
18:            return COMPTAGEMOTS(node.middle)
19:        else
20:            return _PREFIXE(node.middle, prefix, index + 1)
21:        end if
22:    end if
23: end function
```

Algorithm 9 Suppression d'un mot dans un Trie Hybride

```
1: function SUPPRESSION(self, word)
2:     self.root ← _SUPPRESSION(self.root, word, 0)
3:     if self.root ≠ null and self.root.is_end_of_word = false and all children of self.root
   = null then
4:         self.root ← null                ▷ Nettoyage si la racine devient inutile
5:     end if
6: end function
7: function _SUPPRESSION(node, word, index)
8:     if node = null then
9:         return null                     ▷ Retourner null si le nœud est vide
10:    end if
11:    char ← word[index]
12:    if char < node.char then
13:        node.left ← _SUPPRESSION(node.left, word, index)
14:    else if char > node.char then
15:        node.right ← _SUPPRESSION(node.right, word, index)
16:    else
17:        if index + 1 = len(word) then
18:            node.is_end_of_word ← false          ▷ Supprimer la fin du mot
19:        else
20:            node.middle ← _SUPPRESSION(node.middle, word, index + 1)
21:        end if
22:        if node.is_end_of_word = false and node.left = null and node.middle = null
   and node.right = null then
23:            return null                 ▷ Supprimer le nœud inutile
24:        end if
25:    end if
26:    return node
27: end function
```

FONCTIONS DES TRIES

TRIES HYBRIDES

```
>>> Tester la fonctionnalité 3.8:

>>> Arbre non équilibré:
Hauteur de l'arbre : 21
Profondeur moyenne : 9.666666666666666
Tous les mots : ['ant', 'apple', 'ball', 'banana',
lemon', 'lime', 'melon', 'mouse', 'orange', 'peach',

>>> Arbre équilibré:
Hauteur de l'arbre : 13
Profondeur moyenne : 6.966666666666667
Tous les mots : ['ant', 'apple', 'ball', 'banana',
lemon', 'lime', 'melon', 'mouse', 'orange', 'peach',
```

FONCTIONS COMPLEXES Détecter et rééquilibrer

1- is_unbalanced :

- **depth_threshold** : Seuil pour la différence de profondeur maximale
- **balance_threshold** : Rapport entre la profondeur maximale et la profondeur moyenne.

2- rebalance :

- **Extraction des mots**
- **Tri et dédoublage**
- **Construction équilibrée** : en utilisant la méthode par division (binaire).

3- _build_balanced_tree :

- **Milieu** : Choisir l'élément central de la liste des mots comme nœud racine
- **Sous-arbres gauche et droit**

4- _build_tree_from_word :

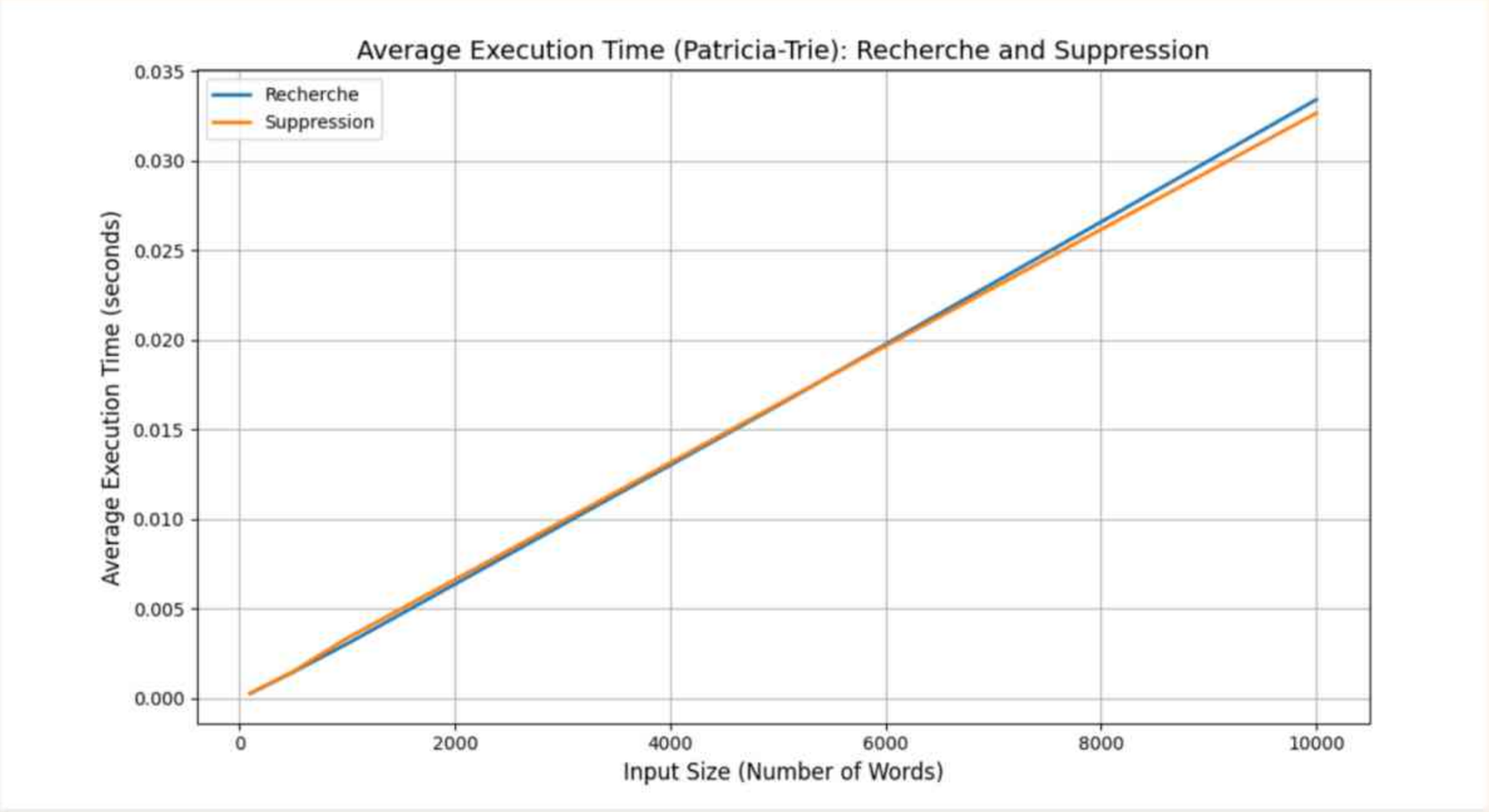
- **Noeuds chaînés** : Insérer chaque caractère du mot dans l'arbre en séquence pour former une structure en chaîne
- **Marqueur de fin de mot** : Marquer la fin d'un mot dans l'arbre.

5- insert_with_balance :

- **Insertion conditionnelle** : Empêcher l'insertion de mots en double.
- **Détection du déséquilibre** : Appeler is_unbalanced pour vérifier l'état d'équilibre de l'arbre
- **Rééquilibrage automatique** : Appeler rebalance pour reconstruire l'arbre lorsqu'il est déséquilibré.

COMPLEXITÉS

PATRICIA-TRIES

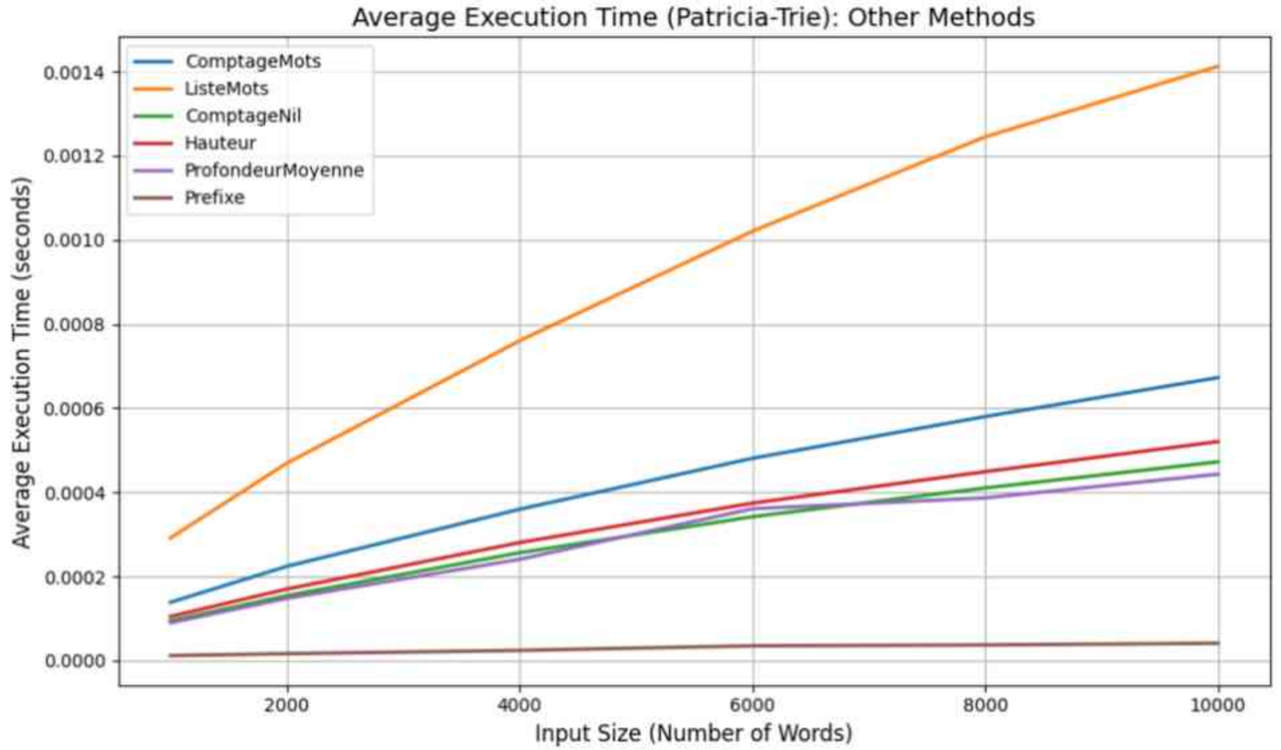


O(k)

k est le longeur de mot

COMPLEXITÉS

PATRICIA-TRIES



ListeMots: O(m)[m: la somme des longueurs de tous les mots dans le trie]

ComptageMots/comptage_nil /hauteur/profondeurMoyenne :O(N) [N :nb de noeuds]

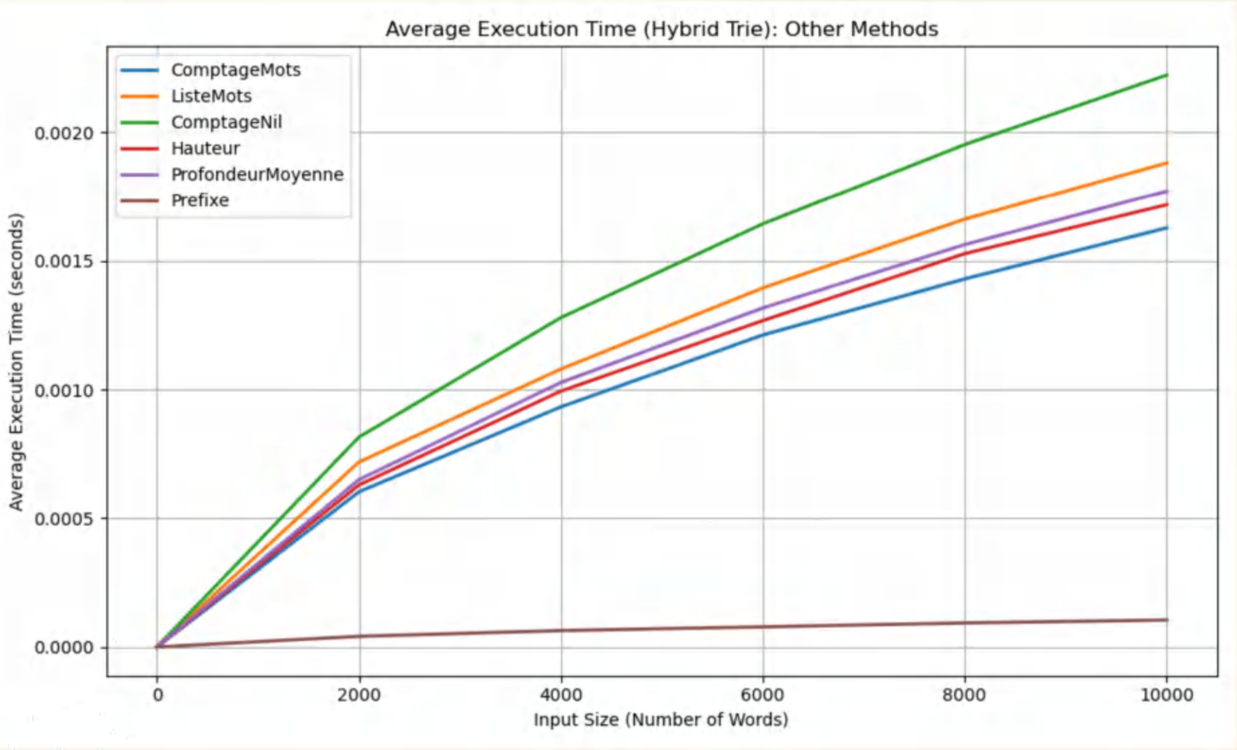
Prefixe: O(k+nbSousNoeud)[k est nb de mots de prefix]

COMPLEXITÉS

TRIES HYBRIDES

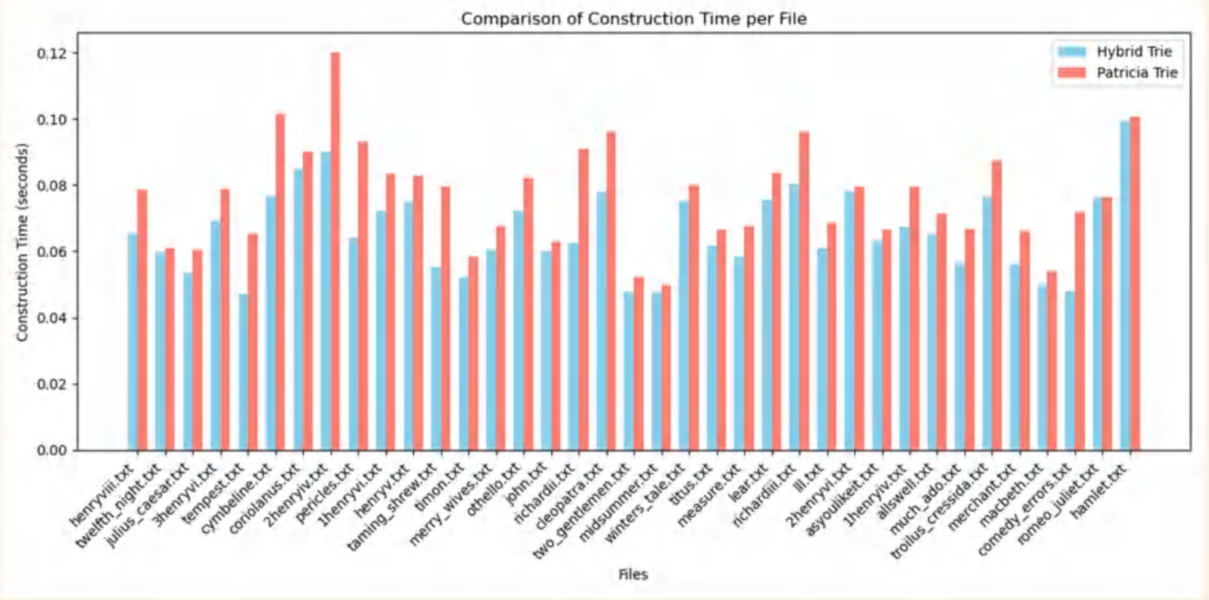
Fonction	Complexité temporelle	Complexité spatiale
Recherche	$O(h)$	$O(1)$
ComptageMots	$O(n)$	$O(h)$
ListeMots	$O(n)$	$O(n)$
ComptageNil	$O(n)$	$O(h)$
Hauteur	$O(n)$	$O(h)$
ProfondeurMoyenne	$O(n)$	$O(h)$
Prefixe	$O(p+k)$	$O(h)$
Suppression	$O(h)$	$O(h)$

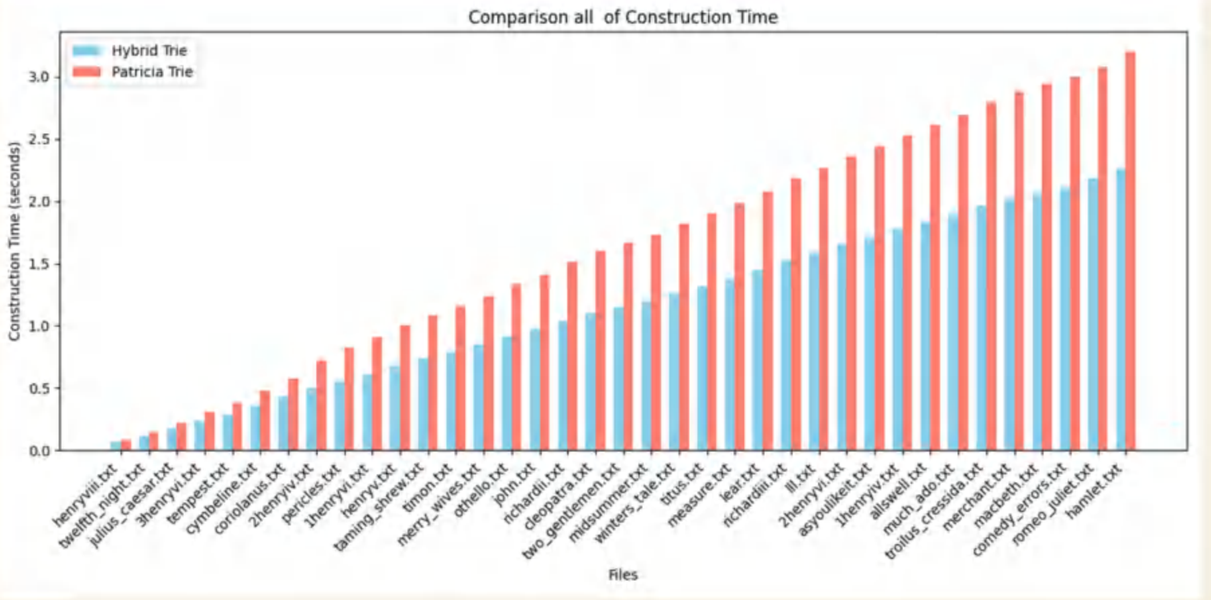
TABLE 1 – Résumé des Complexités des Algorithmes

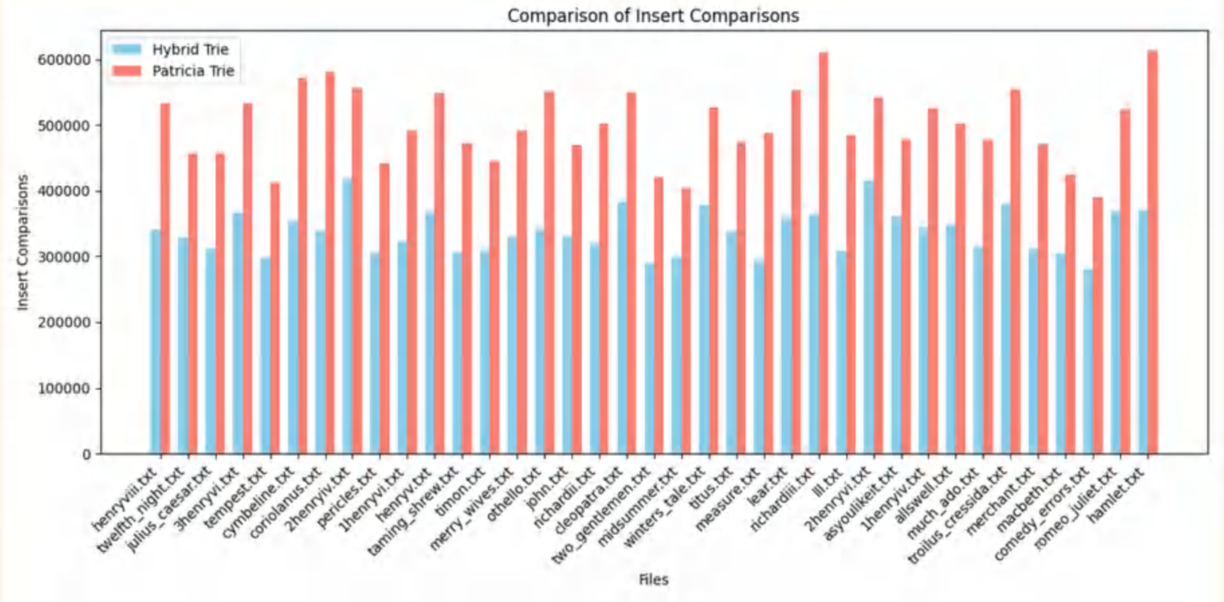


19

ANALYSE EXPÉRIMENTALE

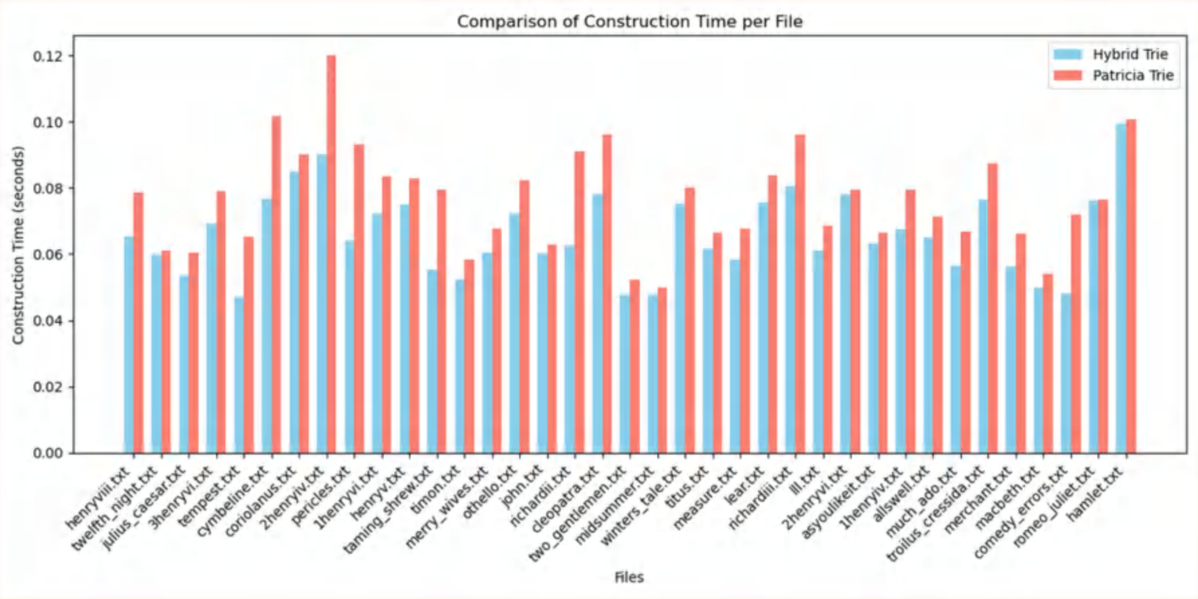


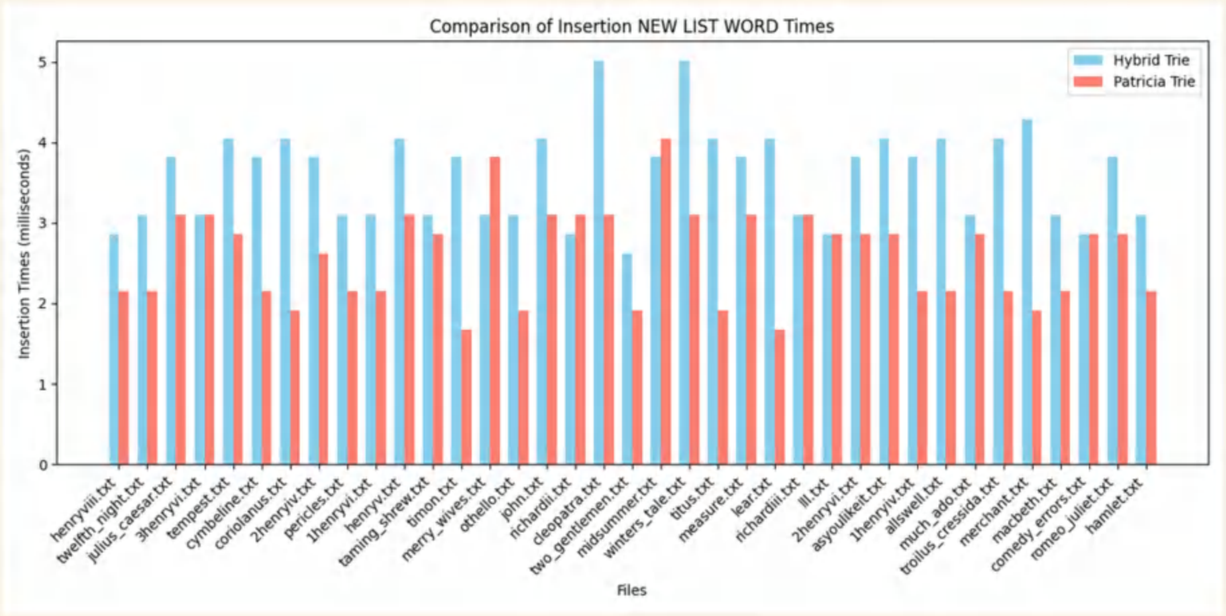




- **Temps de Construction : Hybrid Trie est plus efficace à construire que le Patricia Trie**
- **Nombre de Comparaisons : Patricia Trie complexe, plus de comparaisons**

ANALYSE EXPÉRIMENTALE

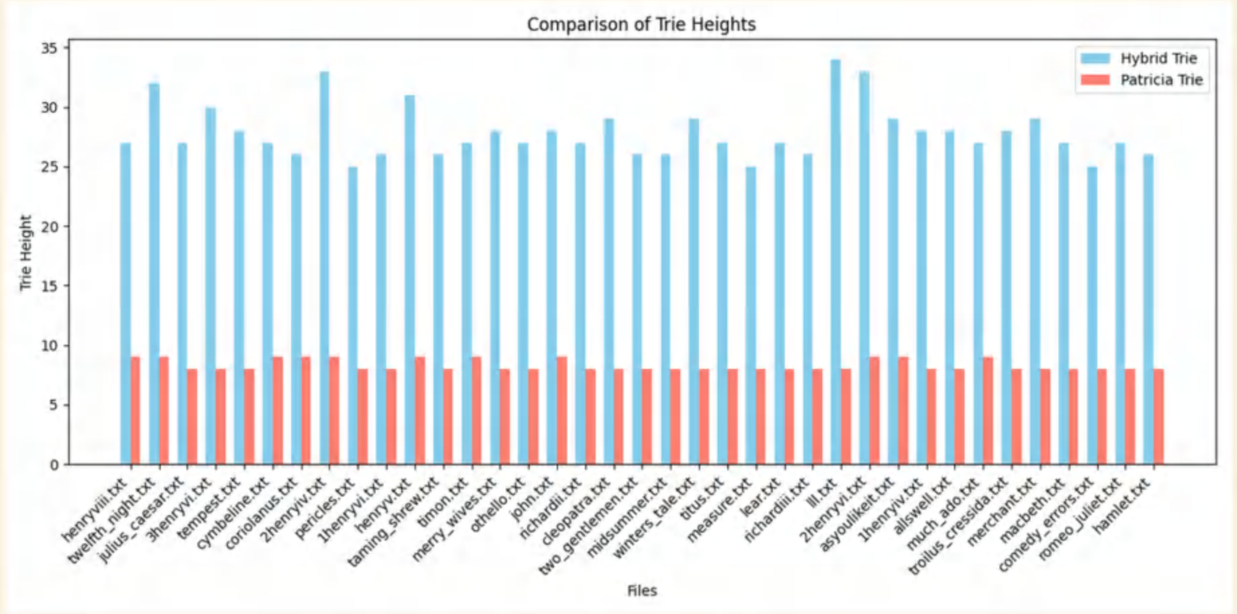


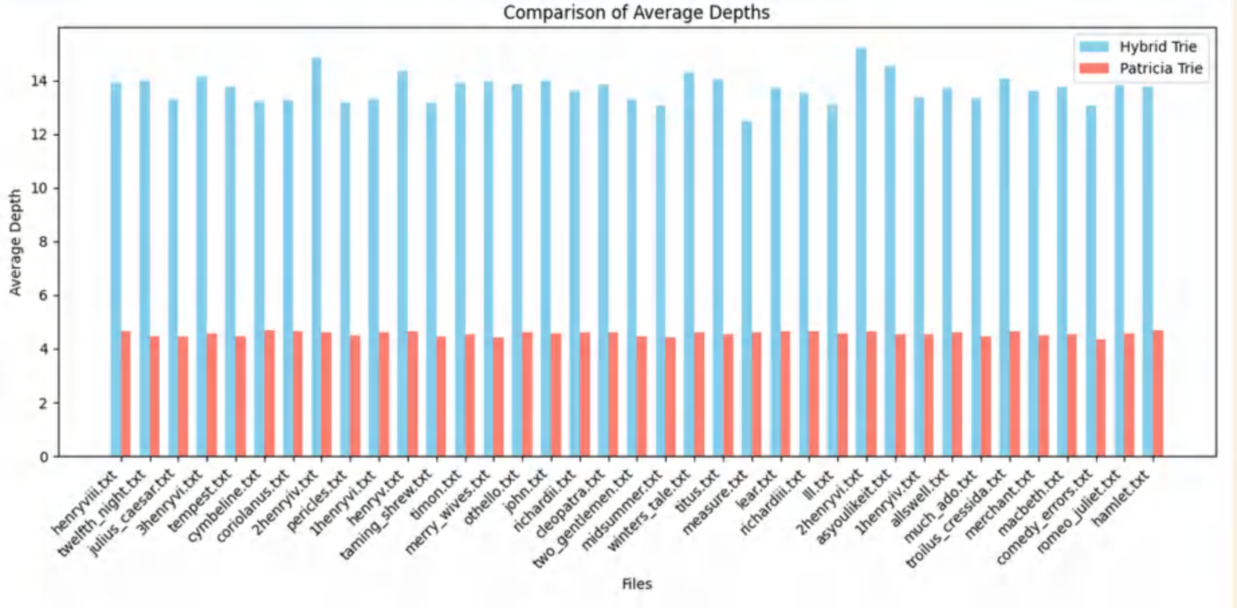



```
19987  hatnte
19988  oln
19989  rnh
19990  eushldnodfne
19991  fhk
19992  sth
19993  oobmh
19994  yt
19995  iefesh
19996  aeaaop
19997  eeih
19998  tm
19999  obreeihhe
20000  ohns
```

- **insérer de nouveaux mots(mélanger) : La plupart du temps, Patricia Trie est plus rapide.**

ANALYSE EXPÉRIMENTALE

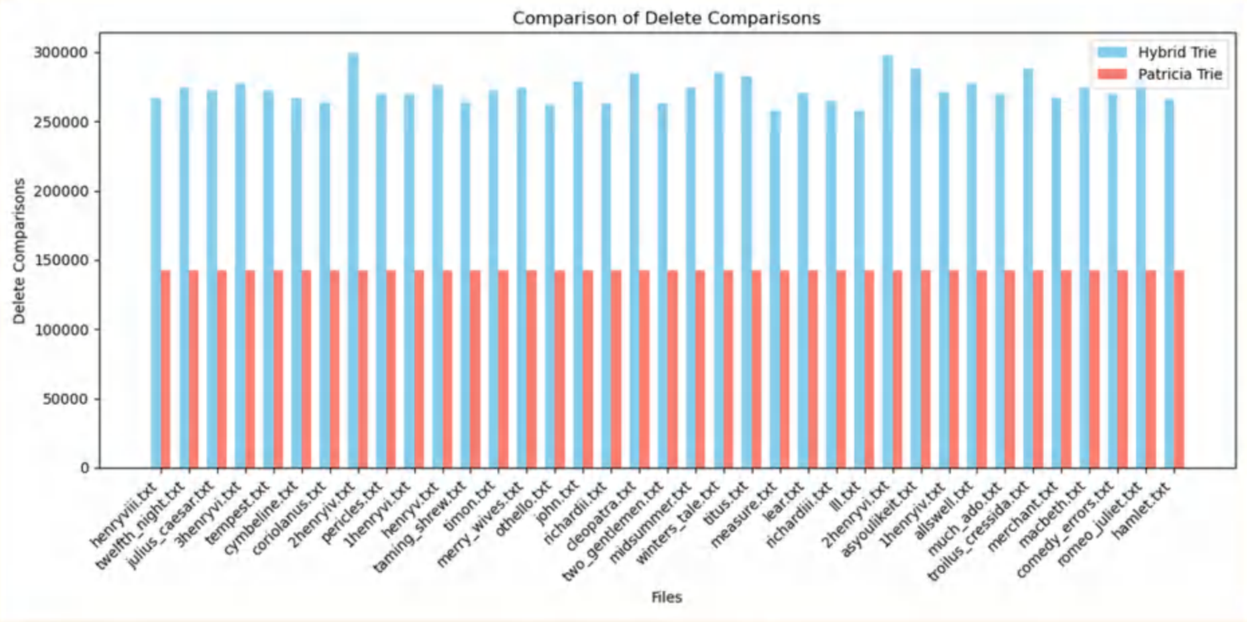




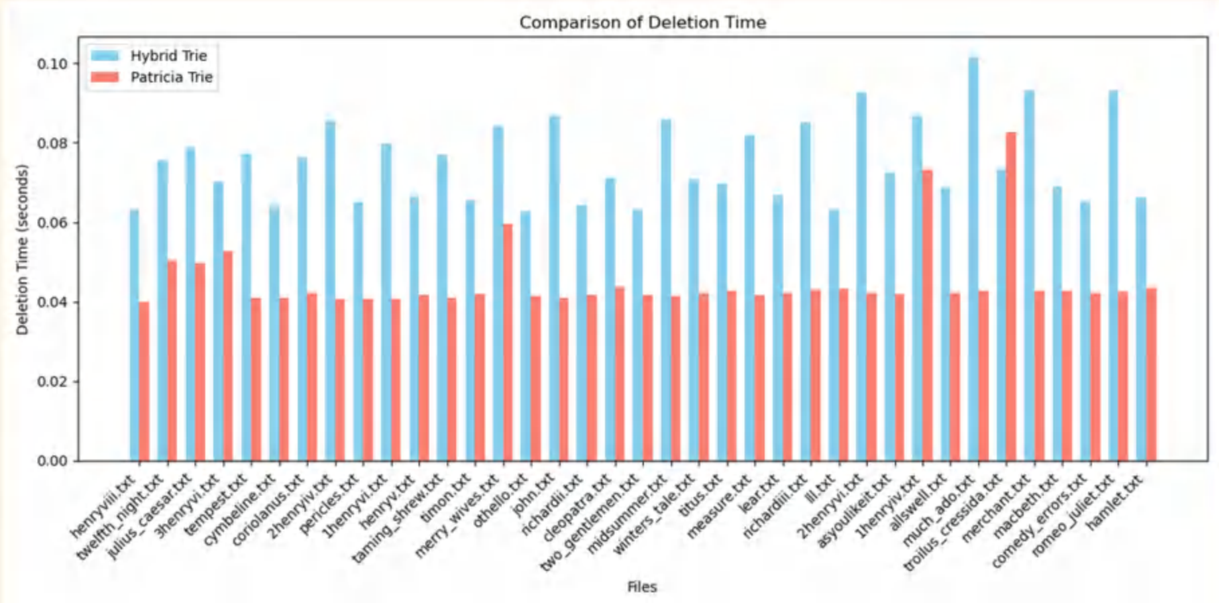
- **Hauteur et Profondeur moyenne : Hybrid Trie plus grandes**

Patricia Trie offre une structure plus compacte avec des chemins plus courts et plus stables

ANALYSE EXPÉRIMENTALE

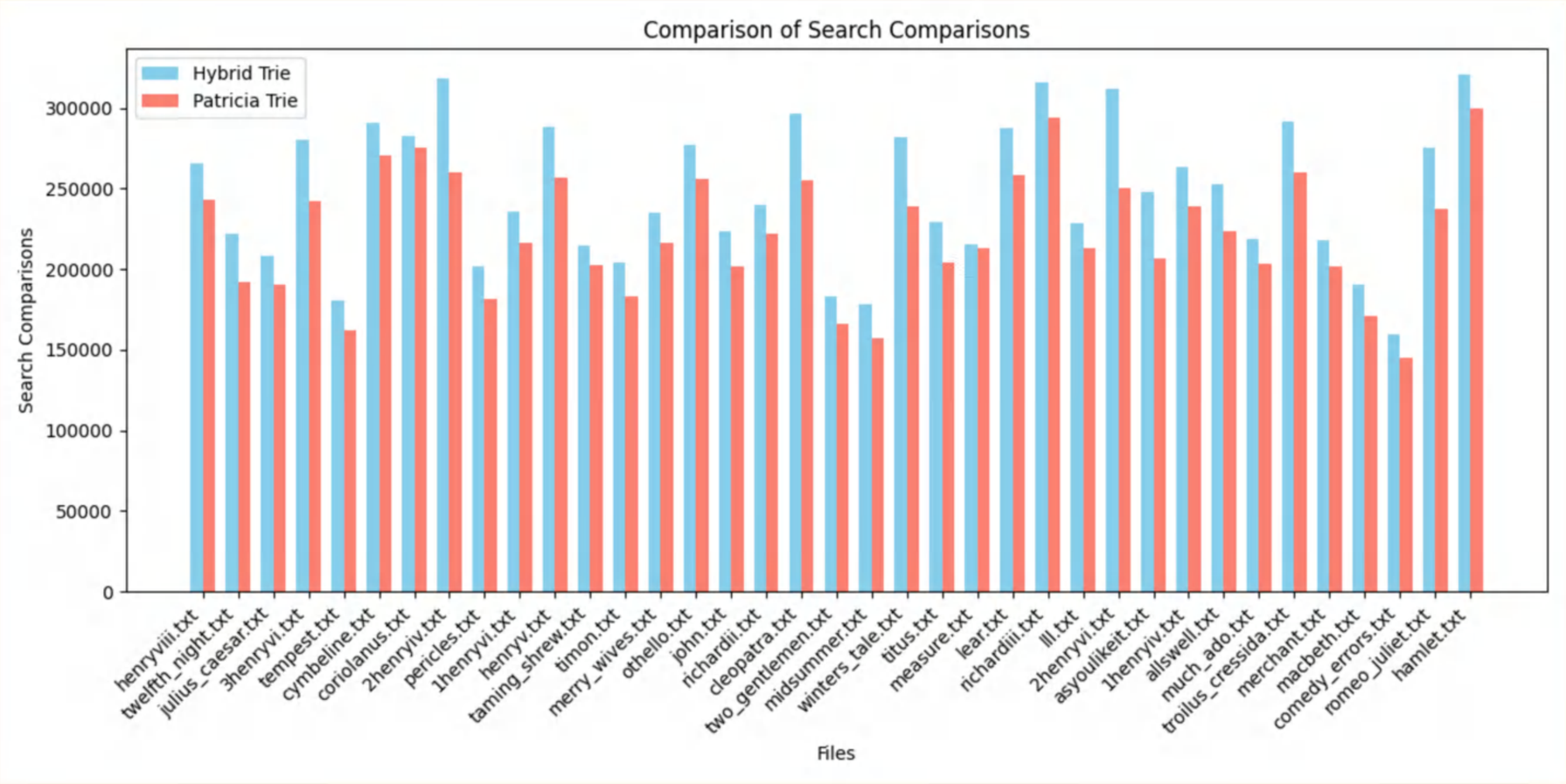


Hybrid Trie présente un coût en comparaisons et un temps de suppression plus élevés en raison de sa structure plus profonde



Patricia Trie est plus efficace pour la suppression grâce à sa structure plus équilibrée et compacte

ANALYSE EXPÉRIMENTALE



- **Patricia Trie : Plus efficace en recherche**

CONCLUSION

- **Hybrid Trie** : gestion flexible, plus rapide dans la construction mais nécessite plus de comparaisons pour les recherches et suppressions

- **Patricia Trie**: plus efficace pour les opérations de recherche et suppression, plus économe en mémoire

MERCI DE VOTRE ATTENTION

Mengxiao LI Xue YANG