Rapport Technique

Moteur de Génération et de Validation
Morphologique de la Langue Arabe

Février 2026

1. Introduction

Ce projet implémente un **moteur morphologique** pour la langue arabe, capable de **générer** des mots dérivés à partir d'une racine trilitère et d'un schème (pattern), et de **valider** l'appartenance d'un mot à une racine donnée. L'architecture repose sur trois structures de données fondamentales : un *arbre binaire de recherche* (BST) pour les racines, une *table de hachage* avec chaînage pour les schèmes, et des *listes chaînées* pour les mots dérivés ainsi que pour la résolution de collisions.

2. Structures de données utilisées

2.1. Arbre binaire de recherche (BST) – Racines

Les racines arabes trilitères (ex. `k-t-b`, كتب) sont stockées dans un **arbre binaire de recherche** (`RootBST`). Chaque nœud contient :

- la racine sous forme compacte (3 lettres Unicode),
- un pointeur vers une liste chaînée de mots dérivés validés,
- des pointeurs `left` / `right`.

L'ordre est défini par la comparaison lexicographique Unicode native de Python. Les opérations proposées sont l'insertion, la recherche, la suppression (avec gestion du successeur in-order), le parcours in-order et le calcul de la hauteur.

Complexité.

Opération	Cas moyen	Pire cas
Insertion	$O(\log n)$	$O(n)$
Recherche	$O(\log n)$	$O(n)$
Suppression	$O(\log n)$	$O(n)$
Parcours in-order	$O(n)$	$O(n)$
Hauteur	$O(n)$	$O(n)$

Le pire cas $O(n)$ survient lorsque l'arbre dégénère en liste (insertions triées). Le projet ne recourt pas à un arbre équilibré (AVL, rouge-noir) ; ce choix est volontaire afin de garder la structure simple tout en restant performant pour un nombre modéré de racines ($n < 10\,000$). En moyenne, les racines arabes, chargées depuis un fichier non trié, produisent un arbre raisonnablement équilibré, d'où une complexité pratique en $O(\log n)$.

2.2. Table de hachage – Schèmes morphologiques

Les schèmes sont stockés dans une **table de hachage à chaînage** (`PatternHashTable`) de taille fixe $m = 37$ (nombre premier). Chaque cellule pointe vers une `PatternRuleChain`, liste chaînée de nœuds `PatternRuleNode` portant le schème normalisé et sa règle de dérivation.

Fonction de hachage. La fonction utilise un hachage polynomial :

$$h(k) = \left(\sum_{i=0}^{|k|-1} \text{ord}(k_i) \cdot 131^i \right) \bmod 37$$

La base 131 (nombre premier) assure une bonne dispersion des caractères Unicode arabes. La taille 37 offre un compromis entre occupation mémoire et taux de collision pour un nombre de schèmes de l'ordre de quelques dizaines.

Complexité.

Opération	Cas moyen	Pire cas
Insertion	$O(1)$	$O(p)$
Recherche	$O(1)$	$O(p)$
Suppression	$O(1)$	$O(p)$
Itération	$O(m+p)$	$O(m+p)$

où p est le nombre total de schèmes et $m = 37$ la capacité de la table. En pratique, avec $p \approx 30$ schèmes, le facteur de charge $\alpha = p/m \approx 0{,}81$ reste modéré et les chaînes sont courtes, garantissant un accès quasi-constant.

2.3. Listes chaînées – Mots dérivés et chaînage

Les listes chaînées interviennent à deux niveaux :

1. **DerivedWordList** : rattachée à chaque nœud racine du BST, elle stocke les mots dérivés validés avec un compteur de fréquence. L'insertion vérifie l'unicité ($O(d)$ pour d dérivés existants) et incrémente le compteur en cas de doublon.
2. **PatternRuleChain** : utilisée pour le chaînage des collisions dans la table de hachage.

Complexité. L'insertion avec vérification d'unicité et la recherche sont en $O(d)$ dans le pire cas, d étant la longueur de la liste. La conversion en liste Python (`to_list`, `to_items`) est en $O(d)$.

3. Algorithmes de génération et de validation

3.1. Génération morphologique

L'algorithme de génération (`MorphologicalGenerator`) prend en entrée une racine et un schème, puis produit un mot dérivé. Le processus se décompose en :

1. **Validation de la racine** : recherche dans le BST – $O(\log n)$ en moyenne.
2. **Validation du schème** : recherche dans la table de hachage – $O(1)$ amorti.
3. **Dérivation** : la règle associée au schème est extraite. Les lettres-repères (ف, ع, ل) sont remplacées par les 3 lettres de la racine. Le parcours du schème est en $O(|s|)$ où $|s|$ est la longueur du schème (typiquement 4–7 caractères).
4. **Stockage** : le mot dérivé est ajouté à la `DerivedWordList` du nœud racine.

Complexité totale d'une génération unitaire :

$$T_{\text{gen}}(n, p) = O(\log n) + O(1) + O(|s|) = O(\log n)$$

La génération d'une famille complète (tous les schèmes pour une racine, `generate_family`) itère sur les p schèmes :

$$T_{\text{family}}(n, p) = O(\log n) + O\left(\sum_{i=1}^{p}(1 + |s_i|)\right) = O(\log n + p \cdot \bar{s})$$

où $\bar{s}$ est la longueur moyenne d'un schème.

3.2. Validation morphologique

L'algorithme de validation (`MorphologicalValidator`) détermine si un mot donné peut être dérivé d'une racine. La stratégie adoptée est une **validation par ré-génération exhaustive** :

1. **Recherche de la racine** dans le BST – $O(\log n)$.
2. **Itération** sur tous les schèmes de la table de hachage.
3. Pour chaque schème, **génération** du mot dérivé correspondant via le générateur.
4. **Comparaison** du mot généré (normalisé) avec le mot saisi (normalisé).
5. Si une correspondance est trouvée : résultat `OUI` + schème identifié. Sinon : `NON`.

Complexité totale d'une validation :

$$T_{\text{val}}(n, p) = O(\log n) + O(p \cdot \bar{s}) = O(\log n + p)$$

4. Choix et justification des algorithmes

- **BST non équilibré pour les racines** : la morphologie arabe comporte environ 6 000 à 10 000 racines trilitères. Un BST non équilibré offre une implémentation simple et reste efficace si les données ne sont pas insérées dans un ordre parfaitement trié, ce qui est le cas en pratique.
- **Table de hachage à taille fixe pour les schèmes** : le nombre de schèmes morphologiques est restreint (quelques dizaines).
- **Validation par ré-génération** : stratégie viable car p est petit (constant en pratique).
- **Listes chaînées pour les dérivés** : le nombre de dérivés par racine est faible.

5. Analyse globale de la complexité

5.1. Complexité spatiale

- **BST** : $O(n)$.
- **Table de hachage** : $O(m + p)$, avec $m = 37$.
- **Listes dérivées** : $O(D)$ au total.
- **Espace total** : $O(n + p + D)$.

5.2. Complexité temporelle – Récapitulatif

Opération	Cas moyen	Pire cas
Insertion d'une racine	$O(\log n)$	$O(n)$
Recherche d'une racine	$O(\log n)$	$O(n)$
Suppression d'une racine	$O(\log n)$	$O(n)$
Insertion d'un schème	$O(1)$	$O(p)$
Recherche d'un schème	$O(1)$	$O(p)$
Génération unitaire	$O(\log n)$	$O(n)$
Génération familiale	$O(\log n + p)$	$O(n + p)$
Validation d'un mot	$O(\log n + p)$	$O(n + p)$
Chargement racines fichier	$O(n \log n)$	$O(n^2)$
Chargement schèmes fichier	$O(p)$	$O(p^2/m)$

6. Principales difficultés rencontrées

1. **Normalisation Unicode de l'arabe** : gestion des variantes et diacritiques.
2. **Correspondance racine-schème** : mapping ف-ع-ل.
3. **Dégénérescence du BST** : possible avec un fichier trié.
4. **Validation sans analyseur morphologique** : ré-génération exhaustive.

7. Conclusion

Le moteur morphologique proposé combine trois structures de données classiques de manière cohérente. L'analyse de complexité montre que le système est efficace pour un usage pratique : $O(\log n)$ pour les opérations courantes sur les racines et $O(1)$ amorti pour les schèmes.