Rapport Technique

Moteur de Génération et de Validation Morphologique de la Langue Arabe

Réalisé par Eya Gharbi et Zaghouani Ayoub
Classe 1ING4

1. Présentation générale

Ce projet implémente un moteur de dérivation morphologique arabe capable de générer des formes dérivées à partir d'un radical trilitère et d'un schème, et de valider si un mot dérivé appartient à un radical donné.

2. Chargement des données

Au démarrage, le système charge deux fichiers texte :

- **roots.txt** : liste initiale des racines.
- **patterns.txt** : liste initiale des schèmes.

Ces fichiers jouent le rôle d'une petite base de données locale permettant de peupler les structures en mémoire avant tout traitement.

3. Structures de données utilisées

3.1. Arbre binaire de recherche

Les racines trilittérales (ex : ك-ت-ب) sont stockées sous forme compacte (كتب).

Chaque lettre arabe est représentée par son code Unicode. La comparaison lexicographique repose donc sur l'ordre Unicode des caractères, ce qui permet un tri naturel sans transformation supplémentaire.

Chaque nœud contient :

- La racine compacte,
- Une liste chaînée des mots dérivés,
- Un pointeur vers le sous-arbre gauche,

- Un pointeur vers le sous-arbre droit.

Normalisation en entrée : Toute racine est d'abord normalisée : suppression des diacritiques, unification des formes d'alef, vérification du format ب-ت-ك.

Encodage et ordre : Chaque racine est encodée en forme compacte (ex : كتب). Chaque lettre arabe est ramenée à un code ordonné (ordre Unicode/arabique), puis la clé compacte est comparée lexicographiquement pour positionner le nœud dans l'arbre.

Insertion : Descente récursive selon l'ordre lexicographique. Doublons refusés.

Suppression : La suppression suit les trois cas classiques : feuille, un enfant, deux enfants avec remplacement par le successeur.

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	$O(n)$	$O(n)$

3.2. Liste chaînée

Chaque nœud de racine contient une **liste chaînée** de dérivés validés (mot + fréquence). Cette structure permet de :

- éviter les recalculs et alléger la mémoire.
- insérer simplement des mots dérivés.
- conserver un historique morphologique.

3.3. Table de hachage

Les schèmes sont stockés dans une **table de hachage à chaînage**. Chaque cellule pointe vers une liste chaînée de nœuds portant le schème normalisé et sa règle de dérivation.

Fonction de hachage : Utilise une fonction polynomiale `_hash(key)` de type :

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

Note : La taille est fixée à $m = \mathbf{37}$, choisie comme nombre premier pour réduire les collisions.

Type d'entrée. Les schèmes sont validés :

- présence des lettres ف ع ل,
- lettres arabes uniquement,
- conservation de la shadda (si présente).

Normalisation : Chaque schème est normalisé (diacritiques supprimés sauf shadda, variantes d'alef unifiées). Les schèmes doivent contenir ف ع ل et respecter une longueur minimale.

Règle de dérivation : Elle est identique au schème normalisé : la dérivation consiste uniquement à substituer ف ع ل par les lettres du radical => Approche `rule = pattern`

Chaînage : Chaque case de la table pointe vers une **liste chaînée** de schèmes. Si deux schèmes ont le même hash, ils sont ajoutés à la liste du même bucket. La recherche parcourt cette liste jusqu'à trouver la clé exacte.

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 de schèmes et m la capacité de la table.

4. Algorithmes de génération et validation morphologique

4.1. Génération

1. Normaliser le radical.
2. Vérifier l'existence du schème dans la table.
3. Substituer ف ع ل par les lettres du radical.
4. Retourner le mot dérivé et le stocker dans la liste chaînée.

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

1. **Normalisation de la racine donnée** : suppression des diacritiques, unification des formes d'alef, vérification du format trilitère. Coût : $O(k)$ où k est la longueur de la racine (typiquement 3 lettres).
2. **Validation de la racine** : recherche dans le BST – $O(\log n)$ en moyenne.
3. **Validation du schème** : recherche dans la table de hachage – $O(1)$ amorti.
4. **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).
5. **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.

4.2. Validation

1. Vérifier l'existence du radical.
2. Générer les mots pour chaque schème.
3. Comparer au mot donné (normalisé).
4. Valider et stocker si correspondance.

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 :

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

5. Choix et justification des algorithmes

ABR pour racines : L'objectif était d'obtenir une recherche en moyenne rapide avec un coût mémoire faible. Un ABR permet un accès **logarithmique en moyenne** si l'arbre est équilibré. Dans ce projet, aucun AVL/Red-Black n'est imposé, donc le pire cas reste $O(n)$. Cette différence est explicitée dans la section complexité et respecte la contrainte théorique du projet en moyenne, même si le pire cas n'est pas logarithmique.

Justification mathématique du $O(\log n)$ moyen : Dans un ABR issu d'une insertion d'éléments en ordre aléatoire, la hauteur moyenne est proportionnelle à $\log n$. Plus précisément, l'espérance de la hauteur est $O(\log n)$ et la profondeur moyenne d'un nœud est $O(\log n)$. Ainsi, les opérations de recherche et d'insertion ont un coût moyen $O(\log n)$, même si le pire cas dégénéré reste $O(n)$.

Table de hachage pour schèmes : Le nombre de schèmes est fixe et relativement petit : un accès en $O(1)$ moyen est optimal. Le chaînage a été choisi pour éviter la réallocation et simplifier les suppressions.

Validation exhaustive : La validation par génération sur l'ensemble des schèmes garantit la cohérence linguistique, sans heuristique ni approximation.

Normalisation : Une normalisation stricte évite des collisions sémantiques (ex. variantes d'alef, diacritiques), ce qui stabilise le comportement des algorithmes.

6. Complexité algorithmique

Soient n le nombre de racines, P le nombre de schèmes, k la longueur du schème.

Opération	**Meilleur cas**	**Pire cas**
Recherche ABR	$O(1)$	$O(n)$
Insertion ABR	$O(1)$	$O(n)$
Suppression ABR	$O(1)$	$O(n)$
Recherche/insertion Hash	$O(1)$	$O(P)$
Génération	$O(k)$	$O(k)$
Validation	$O(k)$	$O(P \times k)$

Approche d'optimisation. Les structures choisies minimisent la complexité moyenne : ABR pour l'ordre, table de hachage pour l'accès, et stockage des dérivés pour éviter des recalculs.

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	$O(p)$	$O(p^2/m)$

7. Difficultés rencontrées

- Normalisation cohérente entre racines et schèmes.
- Gestion des collisions dans la table de hachage.
- Contrôle des duplicatas dans ABR et hash.
- Équilibrage implicite du ABR sans auto-balancement.

8. Point d'originalité

Le système est exploité comme un **dictionnaire morphologique** : chaque schème est associé à une définition, et chaque dérivé validé peut être affiché avec son sens linguistique. Cette exploitation donne une valeur appliquée aux structures de données. Cette exploitation démontre l'application concrète des structures algorithmiques développées.

9. 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.