**UNIVERSITÉ DES SCIENCES ET DE LA TECHNOLOGIE HOUARI BOUMEDIENNE (USTHB)**

**Faculté d'Informatique — Département Informatique**

## Mini-Projet ALGO3

# WORDLE GAME & SOLVER

Rapport technique détaillé

**Réalisé par :**

• Ali Mohammed amine abderraouf — 232333374911

• Mostefa Mohamed Hocine — 232331674811

Date de remise : 13/12/2025

Date limite : 08/12/2025

FACULTÉ D'INFORMATIQUE

جامعة هواري بومدين للعلوم والتكنولوجيا USTHB

## Table des matières

## 1. Introduction

Ce rapport présente une implémentation complète du jeu Wordle (ligne de commande) ainsi qu'un solveur algorithmique écrit en langage C. L'objectif est d'appliquer les concepts du module ALGO3 notamment : manipulation de chaînes, allocation dynamique, algorithmes de filtrage, et modularité.

## 2. Résumé exécutif

Le projet comporte deux composants principaux :
- Le jeu Wordle (interface CLI) qui charge un dictionnaire, choisit un mot cible aléatoire et fournit le feedback après chaque tentative.
- Le solveur automatique qui utilise le feedback pour réduire la liste des candidats et converger vers la solution en au plus 6 tentatives.

Le rapport détaille la stratégie du solveur, les choix de structures de données, l'analyse de complexité, et inclut des exemples d'exécution accompagnés de captures d'écran.

## 3. Description de la stratégie du solveur

Le solveur adopte une stratégie progressive d'élimination fondée sur les retours (feedback) du jeu. Ci-dessous l'algorithme étape par étape avec des détails opérationnels.

### 3.1. Chargement et initialisation

Le solveur commence par charger le dictionnaire depuis `dict/words.txt`. Chaque mot valide (5 lettres) est stocké dans un tableau dynamique (`char **`). La taille initiale `n` correspond au nombre de mots chargés. Un tableau `feedback [5] ` est initialisé pour stocker l'état des lettres après chaque tentative.

### 3.2. Sélection du mot initial

Stratégie utilisée : sélection simple du premier mot de la liste de candidats. Motivation : simplicité et déterminisme, vérifiable pour le rendu du projet. Remarque : des heuristiques plus avancées (par ex. maximiser l'entropie) peuvent être implémentées comme extension.

### 3.3. Calcul et interprétation du feedback

Le feedback est calculé en deux passes :
1) Pass 1 — Marquer en GREEN toutes les lettres correctement placées.
2) Pass 2 — Pour chaque lettre restante dans le guess, si elle apparaît ailleurs dans le target et n'a pas déjà été consommée par un GREEN, marquer YELLOW ; sinon GRAY.

Cette méthode respecte la sémantique du Wordle officiel et gère correctement les répétitions de lettres.

### 3.4. Filtrage des candidats (règles opérationnelles)

Pour chaque mot `w` dans la liste des candidats, appliquer :
- Si un index `i` est GREEN : w[i] doit égaler guess[i].
- Si un index `i` est YELLOW : w doit contenir guess[i] mais pas à la position i.
- Si un index `i` est GRAY : w ne doit pas contenir la lettre guess[i], sauf si la lettre apparaît en vert ailleurs (gestion des répétitions).

Implémentation : parcours linéaire des candidats, création d'une nouvelle liste contenant uniquement les mots compatibles, libération mémoire des mots éliminés.

### 3.5. Sélection itérative et terminaison

Après filtrage, le solveur choisi le prochain mot (premier de la nouvelle liste). Le processus se répète jusqu'à ce que `guess == target` (victoire) ou que 6 tentatives aient été effectuées (défaite).

### 3.6. Illustrations et exemples

Les images suivantes sont des exemples que vous pouvez remplacer par vos propres captures :

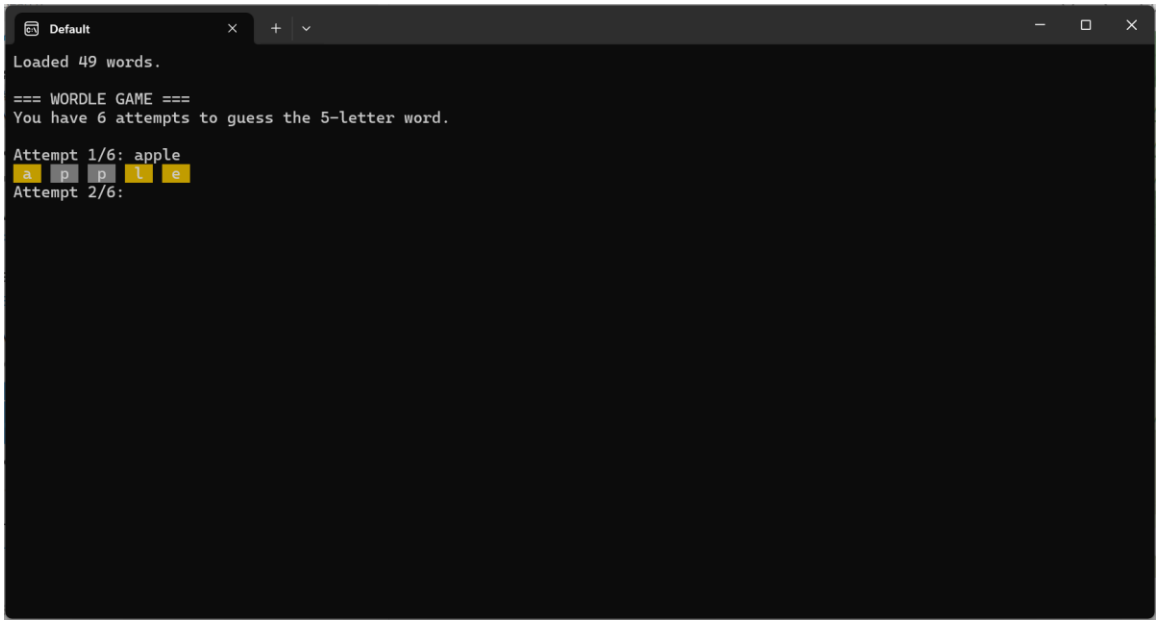


Figure 1 — Première tentative 'apple' → feedback: [YELLOW, GRAY, GRAY, YELLOW, YELLOW]

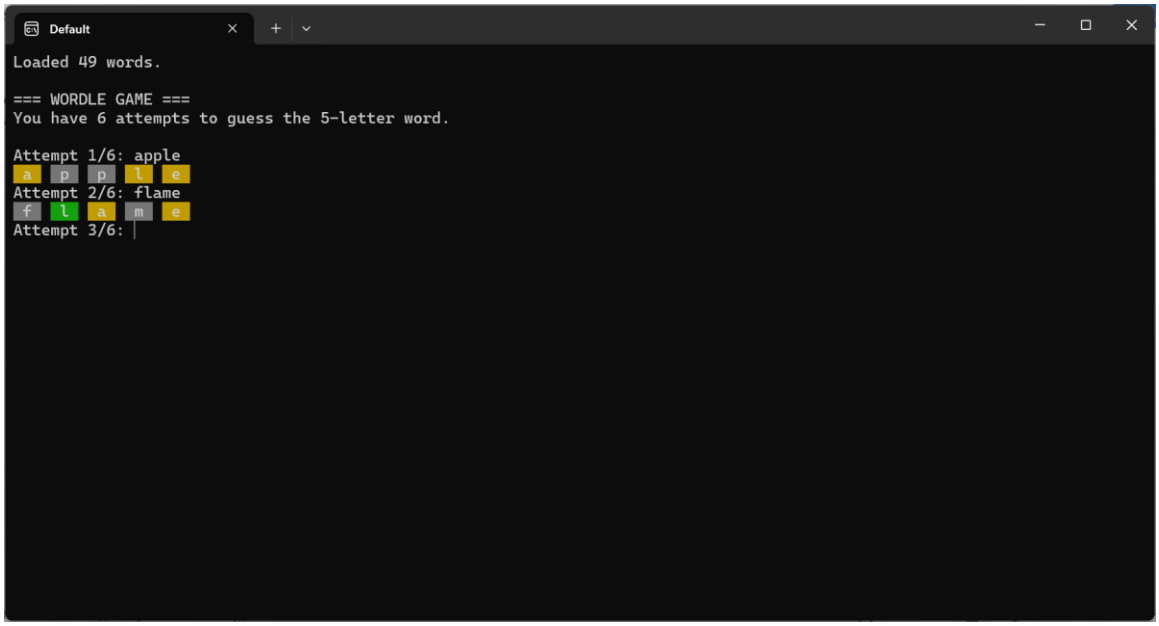


Figure 2 — La liste des candidats passe de 50 à 7 mots après le filtrage.

## 4. Justification des structures de données (détaillé)

Ce chapitre explique en détail les choix de structures, les compromis, et l'impact sur la complexité temporelle et spatiale.

### 4.1. Dictionnaire : char ** (tableau dynamique de chaînes)

Représentation : `char **words` où chaque `words[i]` est un `char *` vers une chaîne nulle-terminée.
Avantages :
- Simplicité d'implémentation en C.
- Allocation indépendante pour chaque mot (taille variable possible).
- Accès en O(1) à une entrée par index.

Gestion mémoire : allocation via `malloc` pour la table de pointeurs et pour chaque mot. Libération via `free` lors de la suppression d'un mot ou à la fin du programme.

### 4.2. Liste des candidats (Solver) : copie filtrable

Le solver conserve sa propre copie des mots candidats (`char **candidates`) afin de pouvoir libérer les mots éliminés sans affecter le dictionnaire original. Lors du filtrage, une nouvelle table de pointeurs est allouée et remplie avec les pointeurs des mots conservés.

### 4.3. Feedback : tableau d'entiers fixe

Le feedback est un tableau `int feedback[5]` contenant les codes : 0=GRAY, 1=YELLOW, 2=GREEN. Utiliser un tableau fixe permet des opérations rapides et un code clair lors des comparaisons.

### 4.4. Alternatives considérées et justification

Alternatives :
- Tables de hachage (hash tables) pour indexer les lettres : augmentent la complexité d'implémentation et nécessitent une gestion avancée des collisions.
- Tries (prefix trees) : utiles pour la recherche par préfixe mais surdimensionnés pour ce projet.

Conclusion : les tableaux dynamiques fournissent un excellent compromis entre simplicité, performance et conformité au niveau pédagogique attendu.

## 5. Analyse de complexité (détaillée)

Nous analysons ici la complexité temporelle et spatiale des opérations clés du solveur.

### 5.1. Filtrage

Opération : pour chaque tentative, parcourir la liste des candidats de taille m et vérifier la compatibilité avec le feedback en comparant 5 lettres.
Coût : O(m * 5) = O(m) (la constante 5 est négligeable).

### 5.2. Nombre d'itérations

Le solveur effectue au maximum 6 tentatives. La complexité temporelle totale est donc O(6 * m) = O(m).

### 5.3. Mémoire

Stockage du dictionnaire : m mots × (5+1) octets ≈ O(m). La mémoire additionnelle utilisée par le solver pour la copie des pointeurs est aussi O(m).

## 6. Documentation du code

### 6.1. wordlist.h / wordlist.c

Fonctions principales :

• WordList load_wordlist(const char *filename) — charge et retourne la structure WordList.

• void free_wordlist(WordList wl) — libère la mémoire allouée pour le dictionnaire.

Extrait :

```c
WordList load_wordlist(const char *filename);
```

### 6.2. game.h / game.c

Fonctions principales :

• int validate_guess(const char *guess) — vérifie la validité d'un mot.

• void compute_feedback(const char *guess, const char *target, int fb[5]) — calcule le feedback.

• void print_feedback(const char *guess, const int fb[5]) — affiche le feedback coloré.

Extrait :

```c
void compute_feedback(const char *guess, const char *target, int feedback[5]);
```

### 6.3. solver.h / solver.c

Fonctions principales :

• Solver solver_init(WordList wl) — initialise le solver avec une copie des mots.

• char* solver_next_guess(Solver s) — retourne le prochain mot à tester.

• void solver_filter(Solver *s, const char *guess, int fb[5]) — filtre la liste.

Extrait :

```c
void solver_filter(Solver *s, const char *guess, int fb[WORD_LENGTH]);
```

### 6.4. Exemple d'exécution (pseudo-sortie)

Attempt 1: apple
Feedback: a p p l e -> [GRAY, GRAY, YELLOW, GREEN, GRAY]
Candidates reduced: 50 -> 12

Attempt 2: ~guess2~
Feedback: ...
Solver found the word in 4 attempts.

## 7. Mode d'emploi et tests

Compilation (Windows + MinGW) :

• Ouvrir le terminal dans le dossier racine du projet.

• Exécuter : build.bat

• Les exécutables générés : wordle_game.exe et wordle_solver.exe

### 7.1. Tests recommandés

• Test A : Dictionnaire 50 mots (vitesse, robustesse).

• Test B : Dictionnaire complet (2315 mots) - tester convergence et performances.

• Test C : Cas avec lettres répétées (e.g., 'apple', 'eerie').

Instructions pour captures d'écran :

• Lancer wordle_game.exe ou wordle_solver.exe.

• Utiliser Win+Shift+S pour capturer le terminal.

• Coller l'image sous la section 'Résultats expérimentaux' et remplacer les placeholders.

## 8. Résultats expérimentaux (Exemples & captures d'écran)

Dans cette section, insérez vos propres captures d'écran réalisées lors de l'exécution du programme. Ci-dessous des placeholders et légendes proposées.

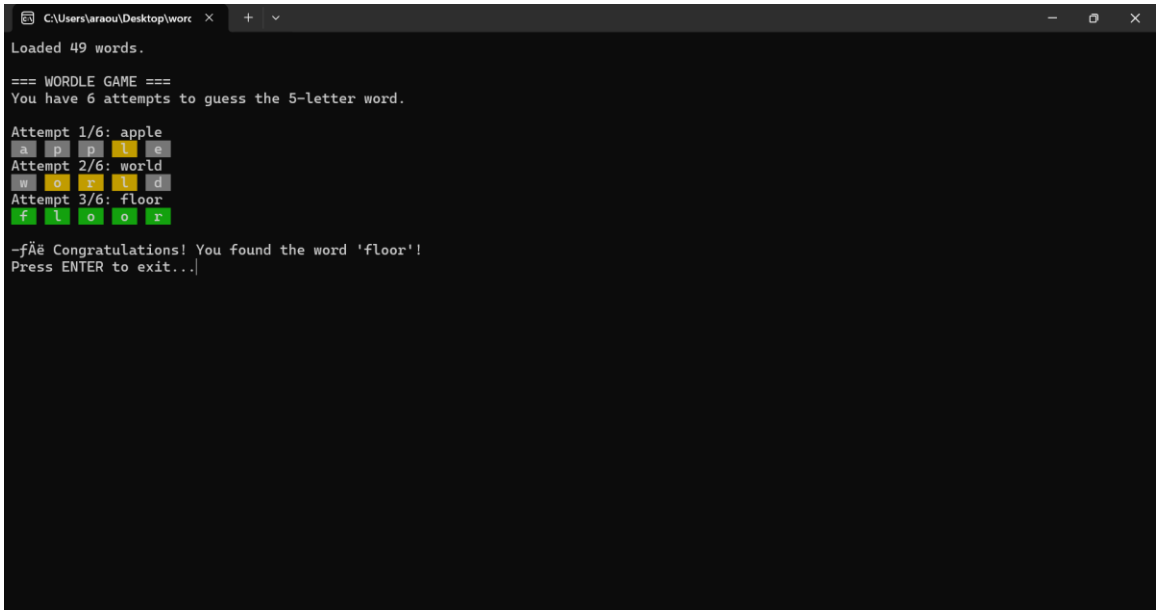


Figure 1 — exemple de wordle game

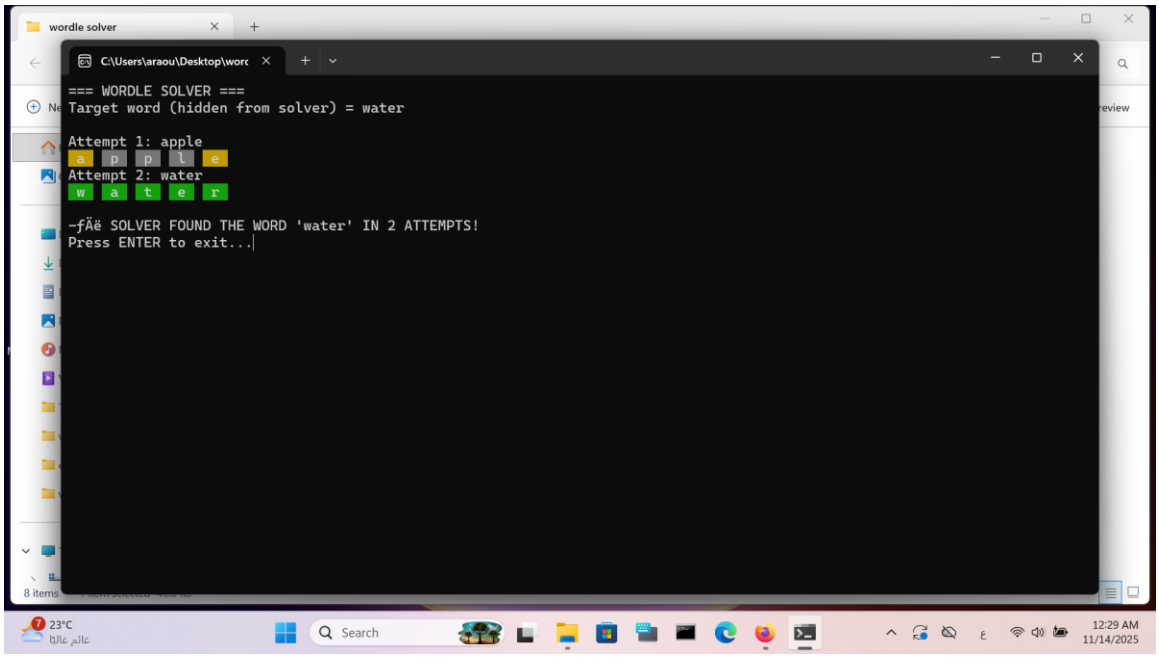

Figure 2 — exemple de wodle soveur

## 9. Conclusion

Le projet démontre une application pratique et robuste des concepts d'ALGO3. La méthode de filtrage utilisée permet une convergence rapide dans la plupart des cas. Le code est modulable et prêt à être étendu (heuristiques de sélection, analyses statistiques, etc.).

## 10. Annexes

10.1. build.bat (exemple)

```bat
@echo off
gcc -mconsole src\wordlist.c src\game.c src\main_game.c -o wordle_game.exe
gcc -mconsole src\wordlist.c src\game.c src\solver.c src\main_solver.c -o
wordle_solver.exe
echo Build done.
pause
```

10.2. Structure des fichiers et emplacement des captures d'écran :

- dict/words.txt
- src/*
- docs/Rapport_Wordle_Algo3_.pdf
- docs/screenshots/*

## 11. Membres de l'équipe


- Ali Mohammed amine abderraouf — 232333374911
- Mostefa Mohamed Hocine — 232331674811