

University of Science and Technology Houari  
Boumediene

Faculty of Computer Science

# Project Report

## Wordle Game

Directed by:

Souidi Ichrak 232331500812  
Lasledj Noura 242431386417

Section: Isil c

Groupe: 3

December 19, 2025

## Abstract

This document analyses a small Wordle program written in C. It contains a playable human mode and an automatic solver mode. The solver uses a simple heuristic (maximizing unique letters) and deterministic filtering to maintain candidate words. The report covers strategy, data structures, complexity, code documentation plus execution screenshots.

## Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Introduction</b>   | <b>3</b>  |
| <b>2</b> | <b>Strategy Description</b>   | <b>3</b>  |
| 2.1      | Word / Candidate Selection Strategy . . . . .                                     | 3         |
| 2.2      | Use of Feedback to Eliminate Possibilities . . . . .                              | 4         |
| 2.3      | Why This Approach Is Effective . . . . .  | 4         |
| <b>3</b> | <b>Data Structure Justification</b>   | <b>4</b>  |
| 3.1      | Data Structures Used . . . . .  | 4         |
| 3.2      | Alternatives Considered . . . . .   | 4         |
| 3.3      | How Choices Support Strategy . . . . .  | 5         |
| <b>4</b> | <b>Complexity Analysis</b>  | <b>5</b>  |
| 4.1      | Time Complexity . . . . .   | 5         |
| 4.2      | Space Complexity . . . . .  | 5         |
| 4.3      | Experimental Scenarios: Solver Behavior at 100,683,and 2315 Words . . .           | 6         |
| <b>5</b> | <b>Code Documentation</b>   | <b>8</b>  |
| 5.1      | Major Functions . . . . .   | 8         |
| 5.2      | Example commented snippet . . . . .   | 8         |
| <b>6</b> | <b>Implementation examples</b>  | <b>9</b>  |
| <b>7</b> | <b>Conclusion</b>   | <b>10</b> |
| <b>8</b> | <b>Optional Extension: Alternative Solver Strategy and Performance Comparison</b> | <b>10</b> |
| 8.1      | Description of the Alternative Strategy . . . . .                                 | 10        |
| 8.2      | Time Complexity Analysis . . . . .  | 10        |
| 8.3      | Space Complexity Analysis . . . . .   | 11        |

|     |   |    |
|-----|---|----|
| 8.4 | Experimental Results and Comparison . . . . . | 12 |
| 8.5 | Conclusion . . . . .                          | 13 |

# 1 Introduction

This project implements a simple Wordle game in C:

- Loads a dictionary of 5-letter words from `words.txt`.
- Provides an interactive (human) game mode.
- Provides an automatic solver mode that picks guesses and updates candidates based on feedback (G/Y/B).

# 2 Strategy Description

## 2.1 Word / Candidate Selection Strategy

The solver's selection strategy picks the candidate word that has the highest number of unique letters. This heuristic is implemented by computing a `unique_score()` for each candidate and choosing the word with the maximum score. The method favors guesses that are likely to reveal the presence or absence of many letters.

Pseudocode:

```
best_score = -inf
best_word = NULL
for each index i in candidates:
    word = solver_words[i]
    s = unique_score(word)
    if s > best_score:
        best_score = s
        best_word = word
return best_word
```

## 2.2 Use of Feedback to Eliminate Possibilities

Feedback format: 5-character string composed of:

- **G** = Green (correct letter position)
- **Y** = Yellow (letter present but different position)
- **B** = black/Gray (letter absent in remaining positions)

Filtering logic (implemented in `matches_feedback()` and `solver_feedback_and_update()`):

1. Confirm all **G** positions match the candidate.
2. Build counts of remaining letters in the candidate (excluding greens).
3. For each **Y**: ensure the guessed letter exists in the remaining counts; decrement.
4. For each **B**: ensure that, after removing greens, the guessed letter does not occur in remaining letters.

## 2.3 Why This Approach Is Effective

- The unique-letter heuristic is cheap ( $O(L)$  per candidate) and tends to test many different letters early.
- Filtering matches Wordle's semantics, so it correctly reduces the candidate set.
- Not optimal globally (doesn't run minimax/entropy), but fast and practical for typical dictionaries.

# 3 Data Structure Justification

## 3.1 Data Structures Used

- **solver\_words**: `char solver_words[MAX_WORDS][WORD_LENGTH+1]` — contiguous storage of dictionary words.
- **candidates**: `int candidates[MAX_WORDS]` — stores indices into `solver_words`.
- **cand\_count**: number of current candidates.
- **Queue**: circular buffer of fixed size to store recent guesses (history).

## 3.2 Alternatives Considered

- **Linked list / dynamic arrays**: would support faster removals but add pointer overhead and complexity.
- **Bitsets**: fast set intersections; good for very large dictionaries but required generating masks per feedback.
- **Hash table**: would speed up word-validity checks for human input (currently linear search).

### 3.3 How Choices Support Strategy

Arrays + index-based candidate lists are cache-friendly and match the solver's  $O(N)$ -scan style: both selection and filtering scan the candidate set, so contiguous arrays are efficient.

## 4 Complexity Analysis

Let  $N$  be the dictionary size,  $L = 5$  word length,  $G = 6$  max guesses.

### 4.1 Time Complexity

- **Initialization** (`init_solver`):  $O(N * L)$  to copy words.
- **Selection** (`solver_suggest`):  $O(\text{cand\_count} * L)$  per guess (compute unique score for each candidate).
- **Filtering** (`solver_feedback_and_update`):  $O(\text{cand\_count} * L)$  per feedback (for each candidate call `matches_feedback()`).
- **Per guess total**:  $O(N * L)$  worst-case; across  $G$  guesses:  $O(G * N * L)$ .

### 4.2 Space Complexity

Major arrays:

$$\text{solver\_words} \approx N * (L + 1) \text{ bytes}, \quad \text{candidates} \approx N * \text{sizeof}(\text{int})$$

Approx total  $14 * N$  bytes (with 1-byte chars and 4-byte ints), so  $N=5000 \rightarrow 70$  KB.

### 4.3 Experimental Scenarios: Solver Behavior at 100,683,and 2315 Words

```
Loaded 2315 words.

Choose mode:
1 - Play (human)
2 - Solver (auto)
3 - Exit
Enter choice: 2
Solver mode: target (hidden) chosen.
Solver guess 1: abhor
a b h o r
-> Candidates left: 39
Solver guess 2: decoy
d e c o y
-> Candidates left: 3
Solver guess 3: demon
d e m o n
-> Candidates left: 2
Solver guess 4: depot
d e p o t
-> Candidates left: 1
Solver guess 5: detox
d e t o x
Solver found the word in 5 attempts!
Solver total time: 0.007000 seconds
```

Figure 1: Solver performance on a dictionary of 2315 words

```
Loaded 683 words.

Choose mode:
1 - Play (human)
2 - Solver (auto)
3 - Exit
Enter choice: 2
Solver mode: target (hidden) chosen.
Solver guess 1: abhor
a b h o r
-> Candidates left: 38
Solver guess 2: clone
c l o n e
-> Candidates left: 1
Solver guess 3: clown
c l o w n
Solver found the word in 3 attempts!
Solver total time: 0.004000 seconds
```

Figure 2: Solver performance on a dictionary of 683 words

```
Loaded 100 words.

Choose mode:
1 - Play (human)
2 - Solver (auto)
3 - Exit
Enter choice: 2
Solver mode: target (hidden) chosen.
Solver guess 1: fable
f a b l e
-> Candidates left: 25
Solver guess 2: fight
f i g h t
-> Candidates left: 1
Solver guess 3: foggy
f o g g y
Solver found the word in 3 attempts!
Solver total time: 0.003000 seconds
```

Figure 3: Solver performance on a dictionary of 100 words

## 5 Code Documentation

### 5.1 Major Functions

- `init_solver(words, total)`: copy dictionary, initialize candidates.
- `solver_suggest(out_guess)`: pick the candidate with maximum unique letters.
- `solver_feedback_and_update(guess, feedback)`: filter candidates using feedback.
- `get_feedback(guess, target, out_feedback)`: compute G/Y/B string.
- `play_game(...)`: interactive human mode.
- `run_solver_auto(...)`: run solver automatically against a random target.
- `initQueue/enqueue/dequeue`: queue helpers for guess history.

### 5.2 Example commented snippet

Listing 1: `matches_feedback` helper (from `solver.c`)

```
1  /* Check whether 'word' (candidate) is consistent with feedback
2     produced
3     when guessing 'guess'. Returns 1 if consistent, 0 otherwise. */
4  static int matches_feedback(const char* word, const char* guess, const
5     char* feedback) {
6     int i;
7     int used_target[WORD_LENGTH];
8     for (i = 0; i < WORD_LENGTH; ++i) used_target[i] = 0;
9
10    /* 1) Greens must match */
11    for (i = 0; i < WORD_LENGTH; ++i) {
12        if (guess[i] == word[i]) {
13            if (feedback[i] != 'G') return 0;
14            used_target[i] = 1;
15        } else {
16            if (feedback[i] == 'G') return 0;
17        }
18    }
19
20    /* 2) Count remaining letters and verify Y / B */
21    int counts[26] = {0};
22    for (i = 0; i < WORD_LENGTH; ++i) {
23        if (!used_target[i]) counts[word[i] - 'a']++;
24    }
25    for (i = 0; i < WORD_LENGTH; ++i) {
26        if (feedback[i] == 'Y') {
27            if (counts[guess[i] - 'a'] <= 0) return 0;
28            counts[guess[i] - 'a']--;
29        } else if (feedback[i] == 'B') {
30            if (counts[guess[i] - 'a'] > 0) return 0;
31        }
32    }
33    return 1;
34 }
```

## 6 Implementation examples

```
Loaded 2315 words.

Choose mode:
1 - Play (human)
2 - Solver (auto)
3 - Exit
Enter choice: 1
Game started you have 6 attempts to guess a 5-letter word.
Attempt 1 enter your 5-letter guess: cloud
c l o u d
Attempt 2 enter your 5-letter guess: mourn
m o u r n
Attempt 3 enter your 5-letter guess: furor
f u r o r
Attempt 4 enter your 5-letter guess: outer
o u t e r
Congratulations! You found the word in 4 attempts.
```

Figure 4: Example of a winning player

```
Choose mode:
1 - Play (human)
2 - Solver (auto)
3 - Exit
Enter choice: 1
Game started you have 6 attempts to guess a 5-letter word.
Attempt 1 enter your 5-letter guess: smile
s m i l e
Attempt 2 enter your 5-letter guess: cloud
c l o u d
Attempt 3 enter your 5-letter guess: bread
b r e a d
Attempt 4 enter your 5-letter guess: sleep
s l e e p
Attempt 5 enter your 5-letter guess: human
h u m a n
Attempt 6 enter your 5-letter guess: turbo
t u r b o
Sorry! You couldn't find the word. The secret word was: guilt
```

Figure 5: Example of a losing player

## 7 Conclusion

- The implemented solver is simple, robust and scales linearly with dictionary size.
- Choice of static arrays and linear filtering offers excellent practical performance for dictionary sizes up to several thousands of words.
- The documented code is modular: solver and data structures can be improved independently while keeping the interface unchanged.

## 8 Optional Extension: Alternative Solver Strategy and Performance Comparison

In addition to the primary solver implementation, an alternative solver strategy was developed using a linked list data structure instead of the queue-based approach used in the original solution.

### 8.1 Description of the Alternative Strategy

In the original solver, candidate words are stored and processed using a queue-like structure. After each guess, all remaining candidates are scanned sequentially, and invalid words are filtered out based on the feedback received.

In the alternative implementation, candidate words are stored in a linked list. After each guess:

- The solver traverses the linked list.
- Nodes corresponding to words that do not match the feedback are removed dynamically.
- The remaining nodes represent the updated set of valid candidate words.

This approach emphasizes dynamic memory usage and pointer manipulation, which are key concepts covered in the course.

### 8.2 Time Complexity Analysis

For both implementations, the dominant operation is filtering candidate words after each guess.

Let  $N$  be the number of remaining candidate words.

Each filtering step requires checking all candidates against the feedback.

Therefore, the time complexity per guess is  $O(N)$  for both approaches.

However, practical differences exist:

The queue-based solver benefits from contiguous memory storage, leading to better cache locality and slightly lower constant-time overhead.

The linked list solver requires pointer dereferencing during traversal and deletion, which introduces a small overhead compared to the queue approach.

Despite these differences, experimental results show that execution times remain very small (on the order of milliseconds), even with large dictionaries.

### 8.3 Space Complexity Analysis

Both solvers require memory to store all candidate words:

- Queue-based solver: Requires  $O(N)$  memory for storing candidate words.

```
Loaded 2315 words.

Choose mode:
1 - Play (human)
2 - Solver (auto)
3 - Exit
Enter choice: 2
Solver mode: target (hidden) chosen.
Solver guess 1: abhor
a b h o r
-> Candidates left: 39
Solver guess 2: decoy
d e c o y
-> Candidates left: 3
Solver guess 3: demon
d e m o n
-> Candidates left: 2
Solver guess 4: depot
d e p o t
-> Candidates left: 1
Solver guess 5: detox
d e t o x
Solver found the word in 5 attempts!
Solver total time: 0.007000 seconds
```

- Linked list solver: Requires  $O(N)$  memory for storing words plus additional memory for pointers, making it slightly more memory-intensive.

```

Loaded 2315 words.

Solver (auto) starting...
Solver guess 1: aback
a b a c k
-> Candidates left: 15

Solver guess 2: drank
d r a n k
-> Candidates left: 2

Solver guess 3: frank
f r a n k
-> Candidates left: 1

Solver guess 4: prank
p r a n k
Solver found the word in 4 attempts!
Solver total time: 0.008000 seconds
Loaded 2315 words.

```

Thus, while both approaches have the same asymptotic space complexity, the linked list implementation has a higher constant memory cost.

## 8.4 Experimental Results and Comparison

Several experiments were conducted using different dictionary sizes (2315 words, 683 words, and 100 words). The solver was run in automatic mode, and the total execution time and number of attempts were observed.

The screenshots included in the report show that:

Both solvers successfully find the target word within a small number of attempts.

Execution times are consistently very low in all cases.

The queue-based solver is marginally faster in practice, although the difference is negligible for the tested input sizes.

Table 1: Comparative Summary: Queue vs Linked List

| Aspect           | Queue         | Linked List            |
|------------------|---------------|------------------------|
| Time Complexity  | $O(N)$        | $O(N)$                 |
| Space Complexity | $O(N)$        | $O(N)$                 |
| Memory Overhead  | Lower         | Higher (pointers)      |
| Dynamic Deletion | Less flexible | Very flexible          |
| Practical Speed  | Very fast     | Very fast (comparable) |

## 8.5 Conclusion

The alternative linked list-based solver demonstrates a valid and functional alternative strategy. While it does not significantly outperform the queue-based solver in terms of execution time, it provides a different perspective on data structure usage and reinforces key concepts such as dynamic memory management and pointer operations.

This comparison highlights that, for this problem size, algorithmic strategy and heuristic choice have a greater impact on performance than the specific linear data structure used.