Wordle Solver

Section: ISILC

Group Members:

♡ Ramoul Meriem

♡ Lakehal Dekrah

♡ Seray Imene

Goal of the Wordle Solver

The goal of this project is to implement a Wordle solver capable of automatically guessing a secret word chosen from a dictionary.
The solver follows the official Wordle rules by generating guesses, receiving feedback, and progressively eliminating impossible words until the correct word is found or the maximum number of attempts is reached.
This file represents the main program responsible for managing the solver, handling feedback, and controlling the game flow.

Part 0: Solver interface (solver.h)

```c
#ifndef SOLVER_H
#define SOLVER_H
#include "wordle.h"

/* Opaque solver type */
typedef struct Solver Solver;

/* Create a solver and load candidates from a dictionary file. Returns NULL on
↪  error */
Solver *solver_create_from_file(const char *filename);

/* Destroy solver and free resources */
void solver_destroy(Solver *s);

/* Number of remaining candidates */
int solver_candidate_count(const Solver *s);

/* Select the next guess (malloc'd string, caller must free) using internal
↪  scoring */
char *solver_select_next(Solver *s);

/* Filter possible words keeping only those consistent with guess/feedback */
void solver_filter_possible(Solver *s, const char *guess, const char
↪  *feedback);

/* Print a short status: remaining count and top candidates (up to n) */
void solver_print_status(const Solver *s, int n);

/* Pick a random secret from the solver's current candidate list (malloc'd) */
char *solver_pick_random_secret(Solver *s);

/* Score a word against current candidates */
int score_word(const Solver *s, const char *word);

/* Run the solver loop until found or attempts exhausted. If secret==NULL, a
↪  random secret from dictionary is used.
 * verbose controls printed output. Returns number of attempts on success, -1
 ↪  on failure.
 */
int solver_loop(Solver *s, const char *secret, int verbose);
```

```c
/* Backwards-compatible helpers (optional) */
char *pick_best_guess(char candidates[][WORD_BUFFER], int count);
void filter_candidates(const char *guess, const char *feedback, char
↪  candidates[][WORD_BUFFER], int *count);

/* Compatibility names requested by user */
Solver *initialize_solver(const char *filename);
void destroy_solver(Solver *s);
char *select_next_guess(Solver *s);
void filter_possible_words(Solver *s, const char *guess, const char *feedback);

#endif /* SOLVER_H */
```

Explanation: Solver interface (solver.h)

Purpose of the header file

- `solver.h` declares the opaque `Solver` type and all public functions that allow other files to create a solver, select guesses, filter candidates, run the solving loop, and clean up memory.
- Keeping these declarations in a header separates the solver interface from its implementation in `solver.c`, making the code easier to reuse and maintain.

Groups of functions

- Initialization and destruction: `solver_create_from_file`, `solver_destroy`, `initialize_solver`, `destroy_solver`.
- Querying and playing: `solver_candidate_count`, `solver_select_next`, `solver_filter_possible`, `solver_print_status`, `solver_pick_random_secret`, `solver_loop`.
- Utility and compatibility: `score_word`, `pick_best_guess`, `filter_candidates`, `select_next_guess`, `filter_possible_words`.

1/ Solver main :

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "wordle.h"
```

```c
#include "solver.h"

int main(void) {
    Solver *s = solver_create_from_file("word.txt");
    if (!s) {
        printf("\033[1;91mError loading dictionary for solver\033[0m\n");
        return 1;
    }

    /* pick secret */
    char *secret = solver_pick_random_secret(s);
    if (!secret) { printf("\033[1;91mError picking secret\033[0m\n");
    ↪  solver_destroy(s); return 1; }

    printf("\033[1;35m=== Wordle Solver Demo ===\033[0m\n");
    printf("Solver will attempt to find a secret chosen from the dictionary.
    ↪  \n\n");

    int attempts = 0;
    int solved = 0;

    while (attempts < 6 && solver_candidate_count(s) > 0 && !solved) {
        printf("--- Step %d ---\n", attempts + 1);
        solver_print_status(s, 8);

        char *guess = solver_select_next(s);
        if (!guess) break;

        char feedback[WORD_BUFFER];
        getFeedback(secret, guess, feedback);

        printf("Solver guess: %s   ", guess);
        printf("Feedback: "); display_feedback(feedback, guess);
        ↪  printf("\n\n");

        if (strcmp(guess, secret) == 0) {
            printf("\033[1;32mSolver found the word '%s' in %d attempts!\033
            ↪  [0m\n", secret, attempts + 1);
            solved = 1; free(guess); break;
        }
```

```
        solver_filter_possible(s, guess, feedback);
        free(guess);
        attempts++;
    }

    if (!solved) printf("\033
    ↪ [1;91mSolver did not find the word. Remaining candidates: %d\033[0m\n"
    ↪ , solver_candidate_count(s));

    free(secret);
    solver_destroy(s);
    return 0;
}
```

GOAL OF THE FUNCTION:

The `main` function demonstrates an automatic Wordle solver that loads a dictionary from `"word.txt"`, picks a secret word, and then repeatedly generates guesses, analyzes feedback, and filters the candidate list until it finds the secret or reaches the maximum number of allowed attempts.

Part 2: Header files (#include)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "wordle.h"
#include "solver.h"
```

Explanation: Included libraries

Used libraries in the solver main:

- `<stdio.h>`: standard input/output library used for functions like `printf` to display messages and feedback in the terminal.
- `<stdlib.h>`: general utilities and dynamic memory management used internally by the solver.
- `<string.h>`: string handling library, needed for functions such as `strcmp` to compare the solver's guesses with the secret word.
- `"wordle.h"`: project header that contains constants like `WORD_BUFFER` and declarations for Wordle-related functions such as `getFeedback` and `display_feedback`.
- `"solver.h"`: header file that defines the `Solver` structure and declares the solver helper functions, including `solver_create_from_file`, `solver_pick_random_secret`, `solver_select_next`, and `solver_filter_possible`.

Part 3: Solver creation

```
int main(void) {
    Solver *s = solver_create_from_file("word.txt");
    if (!s) {
        printf("\033[1;91mError loading dictionary for solver\033[0m\n");
        return 1;
    }
```

Explanation: Solver creation

Function header

- `int main(void)` is the entry point of the solver program. It returns an integer status code to the operating system when the program finishes.

Creating the solver instance

- `Solver *s = solver_create_from_file("word.txt");` creates a solver object and initializes it using the dictionary file `"word.txt"`.
- Internally, `solver_create_from_file` reads all valid words from the file and prepares the data structures the solver will use to make guesses.

Error handling if creation fails

- The condition `if (!s)` checks whether the solver pointer is `NULL`, meaning the dictionary could not be loaded (for example, missing file or memory error).
- In that case, the program prints a red error message using an ANSI escape sequence and returns `1`, indicating that the program terminates with an error because the solver cannot run without a valid dictionary.

Part 4: Secret selection

```
/* pick secret */
char *secret = solver_pick_random_secret(s);
if (!secret) { printf("\033[1;91mError picking secret\033[0m\n");
↪  solver_destroy(s); return 1; }
```

Explanation: Secret selection

Picking the secret word

- The call `solver_pick_random_secret(s)` chooses one random word from the solver's dictionary and stores its address in the pointer `secret`.
- This word becomes the hidden target that the solver must discover during the rest of the program.

Error handling if secret selection fails

- The condition `if (!secret)` checks whether the pointer is `NULL`, which would indicate that the secret could not be selected (for example due to memory allocation failure or an empty dictionary).
- In that case, the program prints a red error message, calls `solver_destroy(s)` to free all solver resources, and returns `1`, because the solver cannot continue without a valid secret word.

Part 5: Main loop condition

```
printf("\033[1;35m=== Wordle Solver Demo ===\033[0m\n");
printf("Solver will attempt to find a secret chosen from the dictionary.
↪ \n\n");

int attempts = 0;
int solved = 0;

while (attempts < 6 && solver_candidate_count(s) > 0 && !solved) {
```

Explanation: Main loop condition

Introductory messages and state variables

- The two `printf` calls display a magenta title `"=== Wordle Solver Demo ==="` and a short description explaining that the solver will try to find a secret word from the dictionary.
- The variable `attempts` is initialized to `0` and counts how many guesses the solver has made, while `solved` is a flag that becomes `1` when the secret word is found.

Critical loop condition

- The `while (attempts < 6 && solver_candidate_count(s) > 0 && !solved)` line defines when the main solving loop continues to run.
- `attempts < 6` enforces the Wordle rule that the solver has at most six guesses.
- `solver_candidate_count(s) > 0` ensures that the loop stops if there are no candidate words left that match the feedback, meaning the solver has no logical moves.
- `!solved` guarantees that once the secret word has been found, the loop terminates immediately and no further guesses are made.

Part 6: Guess selection

```
        printf("--- Step %d ---\n", attempts + 1);
        solver_print_status(s, 8);

        char *guess = solver_select_next(s);
        if (!guess) break;
```

Explanation: Guess selection

Status display before each guess

- `printf("--- Step %d ---\n", attempts + 1);` prints the current step number (starting from 1) so the user can follow the solver's progress.
- `solver_print_status(s, 8);` shows a compact summary of the solver's internal state, for example the number of remaining candidates or a small list of the best possible words (up to 8 entries).

Choosing the next guess

- `char *guess = solver_select_next(s);` asks the solver to pick the next word to try based on the current candidate set and its heuristic.
- If `solver_select_next` returns `NULL`, the condition `if (!guess) break;` stops the loop immediately because there is no valid guess left to make.

Part 7: Feedback generation

```
    char feedback[WORD_BUFFER];
    getFeedback(secret, guess, feedback);

    printf("Solver guess: %s   ", guess);
    printf("Feedback: "); display_feedback(feedback, guess);
    ↪ printf("\n\n");
```

Explanation: Feedback generation

Computing feedback

- The array `feedback[WORD_BUFFER]` is a buffer where the program stores the feedback pattern for the current guess (for example letters representing green, yellow, or black).
- `getFeedback(secret, guess, feedback);` compares the guessed word with the secret word letter by letter and fills the `feedback` array according to the Wordle rules.

Displaying the guess and its feedback

- `printf("Solver guess: %s   ", guess);` prints the word chosen by the solver for this step.
- The sequence `printf("Feedback: "); display_feedback(feedback, guess); printf("\n` prints a label and then calls `display_feedback` to show the feedback in a colored Wordle-style format, followed by blank lines for readability.

Part 8: Filtering logic

```
        if (strcmp(guess, secret) == 0) {
            printf("\033[1;32mSolver found the word '%s' in %d attempts!\033
            ↪  [0m\n", secret, attempts + 1);
            solved = 1; free(guess); break;
        }

        solver_filter_possible(s, guess, feedback);
        free(guess);
        attempts++;
```

Explanation: Filtering logic

Checking if the guess is correct

- `strcmp(guess, secret) == 0` tests whether the guess is exactly the same as the secret word.
- If the comparison is true, the program prints a green success message showing the secret word and the number of attempts, sets `solved = 1`, frees the current `guess`, and exits the loop with `break`.

Filtering remaining candidates

- When the guess is not correct, `solver_filter_possible(s, guess, feedback);` updates the solver's internal candidate list by removing all words that are incompatible with the feedback received for this guess.
- After filtering, the program frees the memory for `guess` and increments `attempts`, preparing for the next iteration of the main solving loop.

Part 9: Memory management and conclusion

```
    if (!solved) printf("\033
    ↪ [1;91mSolver did not find the word. Remaining candidates: %d\033[0m\n"
    ↪ , solver_candidate_count(s));

    free(secret);
    solver_destroy(s);
    return 0;
}
```

Explanation: Memory management and conclusion

Failure case message

- If `solved` is still `0` after the loop, the program prints a red message informing that the solver did not find the word and shows how many candidate words remain according to `solver_candidate_count(s)`.

Cleaning up resources and exiting

- `free(secret);` releases the dynamically allocated memory used to store the secret word.
- `solver_destroy(s);` frees all resources owned by the solver object, such as the dictionary and internal arrays, to avoid memory leaks.
- `return 0;` ends the `main` function and signals normal termination of the solver program to the operating system.

Complexity of the solver main

The overall time complexity of the solver `main` function is approximately $\mathcal{O}(K \cdot N \cdot L)$, where:

- N is the number of words in the dictionary,
- L is the word length,
- K is the maximum number of attempts (here $K = 6$).

Since K is a small constant, this behaves like $\mathcal{O}(N \cdot L)$ with respect to the dictionary size and word length.

2/ Solver logic:

Code

```c
// New solver implementation with a Solver object and helper functions
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "solver.h"
#include "wordle.h"

struct Solver {
```

```
    char candidates[MAX_WORDS][WORD_BUFFER];
    int count;
};

/* Internal: score words by unique-letter frequency */
static void compute_letter_freq(const Solver *s, int freq[26]) {
    for (int i = 0; i < 26; ++i) freq[i] = 0;
    for (int i = 0; i < s->count; ++i) {
        int seen[26] = {0};
        for (int j = 0; j < WORD_LEN; ++j) {
            char c = s->candidates[i][j];
            if (c < 'a' || c > 'z') continue;
            int idx = c - 'a';
            if (!seen[idx]) { freq[idx]++; seen[idx] = 1; }
        }
    }
}

/* Create solver from file */
Solver *solver_create_from_file(const char *filename) {
    Solver *s = malloc(sizeof(Solver));
    if (!s) return NULL;
    int loaded = loadDictionary(filename, s->candidates);
    if (loaded <= 0) { free(s); return NULL; }
    s->count = loaded;
    return s;
}

void solver_destroy(Solver *s) {
    free(s);
}

int solver_candidate_count(const Solver *s) {
    return s ? s->count : 0;
}

/* Pick best using internal freq scoring */
char *solver_select_next(Solver *s) {
    if (!s || s->count <= 0) return NULL;
    int freq[26]; compute_letter_freq(s, freq);
    long best_score = -1; int best_idx = 0;
```

```c
    for (int i = 0; i < s->count; ++i) {
        int seen[26] = {0}; long score = 0;
        for (int j = 0; j < WORD_LEN; ++j) {
            char c = s->candidates[i][j]; if (c < 'a' || c > 'z') continue;
            int idx = c - 'a'; if (!seen[idx]) { score += freq[idx]; seen[idx]
            ↪  = 1; }
        }
        if (score > best_score) { best_score = score; best_idx = i; }
    }
    char *copy = malloc(WORD_BUFFER);
    if (!copy) return NULL;
    strncpy(copy, s->candidates[best_idx], WORD_BUFFER-1); copy[WORD_BUFFER-1]
    ↪  = '\0';
    return copy;
}

/* Score a given word against current candidate letter frequencies. Higher is
↪  better. */
int score_word(const Solver *s, const char *word) {
    if (!s || !word) return 0;
    int freq[26]; compute_letter_freq(s, freq);
    int seen[26] = {0}; int score = 0;
    for (int j = 0; j < WORD_LEN; ++j) {
        char c = word[j]; if (c < 'a' || c > 'z') continue; int idx = c - 'a';
        if (!seen[idx]) { score += freq[idx]; seen[idx] = 1; }
    }
    return score;
}

/* solver_loop - run the solver until solved or out of attempts. If
↪  secret==NULL pick random.
 * returns attempts used on success, -1 on failure.
 */
int solver_loop(Solver *s, const char *secret_in, int verbose) {
    if (!s) return -1;
    char *secret = NULL;
    char *dict_ptrs[MAX_WORDS];
    for (int i = 0; i < s->count; ++i) dict_ptrs[i] = s->candidates[i];
    if (secret_in) {
        secret = malloc(WORD_BUFFER); strncpy(secret, secret_in,
        ↪  WORD_BUFFER-1); secret[WORD_BUFFER-1]='\0';
    } else {
```

```
        secret = pick_random_word(dict_ptrs, s->count);
        if (!secret) return -1;
    }

    int attempts = 0;
    while (attempts < 6 && s->count > 0) {
        if (verbose) { printf("[Solver] Step %d - remaining candidates: %d\n",
        ↪  attempts+1, s->count); }
        char *guess = solver_select_next(s);
        if (!guess) break;
        char feedback[WORD_BUFFER];
        getFeedback(secret, guess, feedback);
        if (verbose) {
            printf("[Solver] Guess: %s  Feedback: ", guess);
            ↪  display_feedback(feedback, guess); printf("\n");
        }
        if (strcmp(guess, secret) == 0) { free(guess); if (!secret_in)
        ↪  free(secret); return attempts+1; }
        solver_filter_possible(s, guess, feedback);
        free(guess);
        attempts++;
    }
    if (!secret_in) free(secret);
    return -1;
}

/* local feedback computation (same as game) */
static void compute_feedback_local(const char *secret, const char *guess, char
↪  *out) {
    int len = WORD_LEN; int used[WORD_LEN]; for (int i=0;i<len;i++) used[i]=0;
    for (int i=0;i<len;i++) { if (guess[i]==secret[i]) { out[i]='G';
    ↪  used[i]=1; } else out[i]='B'; }
    for (int i=0;i<len;i++) { if (out[i]=='G') continue; for (int
    ↪  j=0;j<len;j++) if(!used[j] && guess[i]==secret[j]) { out[i]='Y';
    ↪  used[j]=1; break; } }
    out[len]='\0';
}

void solver_filter_possible(Solver *s, const char *guess, const char
↪  *feedback) {
    if (!s || s->count<=0) return;
    int write = 0; char ftmp[WORD_BUFFER];
```

```
    for (int i=0;i<s->count;i++) {
        compute_feedback_local(s->candidates[i], guess, ftmp);
        if (strcmp(ftmp, feedback)==0) {
            if (write!=i) { strncpy(s->candidates[write], s->candidates[i],
            ↪  WORD_BUFFER-1); s->candidates[write][WORD_BUFFER-1]='\0'; }
            write++;
        }
    }
    s->count = write;
}

void solver_print_status(const Solver *s, int n) {
    if (!s) return;
    printf("Remaining candidates: %d\n", s->count);
    int show = (n < s->count) ? n : s->count;
    for (int i=0;i<show;i++) printf("  %s\n", s->candidates[i]);
    if (s->count > show) printf("  ... (+%d more)\n", s->count - show);
}

/* Backwards-compatible helpers (wrap the solver functions) */
char *pick_best_guess(char candidates[][WORD_BUFFER], int count) {
    Solver tmp; tmp.count = count; for (int i=0;i<count;i++)
    ↪  strncpy(tmp.candidates[i], candidates[i], WORD_BUFFER-1);
    return solver_select_next(&tmp);
}

void filter_candidates(const char *guess, const char *feedback, char
↪  candidates[][WORD_BUFFER], int *count) {
    Solver tmp; tmp.count = *count; for (int i=0;i<*count;i++)
    ↪  strncpy(tmp.candidates[i], candidates[i], WORD_BUFFER-1);
    solver_filter_possible(&tmp, guess, feedback);
    /* copy back */
    for (int i=0;i<tmp.count;i++) strncpy(candidates[i], tmp.candidates[i],
    ↪  WORD_BUFFER-1);
    *count = tmp.count;
}

/* Convenience wrappers requested in header */
Solver *initialize_solver(const char *filename) { return
↪  solver_create_from_file(filename); }
void destroy_solver(Solver *s) { solver_destroy(s); }
char *select_next_guess(Solver *s) { return solver_select_next(s); }
```

```c
void filter_possible_words(Solver *s, const char *guess, const char *feedback)
↪  { solver_filter_possible(s, guess, feedback); }

char *solver_pick_random_secret(Solver *s) {
    if (!s || s->count <= 0) return NULL;
    char *ptrs[MAX_WORDS];
    for (int i = 0; i < s->count; ++i) ptrs[i] = s->candidates[i];
    return pick_random_word(ptrs, s->count);
}
```

Goal of the solver module

The goal of this solver module is to manage a dynamic set of candidate words and automatically play Wordle by choosing informative guesses, updating the candidate list according to feedback, and optionally running a full solving loop until the secret word is found or the attempt limit is reached.

Part 1: Solver structure and letter frequency

```c
struct Solver {
    char candidates[MAX_WORDS][WORD_BUFFER];
    int count;
};

/* Internal: score words by unique-letter frequency */
static void compute_letter_freq(const Solver *s, int freq[26]) {
    for (int i = 0; i < 26; ++i) freq[i] = 0;
    for (int i = 0; i < s->count; ++i) {
        int seen[26] = {0};
        for (int j = 0; j < WORD_LEN; ++j) {
            char c = s->candidates[i][j];
            if (c < 'a' || c > 'z') continue;
            int idx = c - 'a';
            if (!seen[idx]) { freq[idx]++; seen[idx] = 1; }
        }
    }
}
```

Explanation: Solver structure and letter frequency

Solver data structure

- The `Solver` structure stores all current candidate words in a 2D array `candidates[MAX_WORDS][WORD_BUFFER]` and tracks how many are valid with the integer `count`.
- This compact representation lets the solver update and filter the list of possible answers efficiently after each guess.

Letter frequency computation

- `compute_letter_freq` computes, for each letter `'a'..'z'`, in how many candidate words it appears at least once and stores these counts in `freq[26]`.
- For every candidate word, the local `seen[26]` array avoids double-counting the same letter within one word, so words with repeated letters do not artificially inflate the frequency.

Part 2: Solver creation and destruction

```c
/* Create solver from file */
Solver *solver_create_from_file(const char *filename) {
    Solver *s = malloc(sizeof(Solver));
    if (!s) return NULL;
    int loaded = loadDictionary(filename, s->candidates);
    if (loaded <= 0) { free(s); return NULL; }
    s->count = loaded;
    return s;
}

void solver_destroy(Solver *s) {
    free(s);
}

int solver_candidate_count(const Solver *s) {
    return s ? s->count : 0;
}
```

Explanation: Solver creation and destruction

Creating the solver from a dictionary file

- `solver_create_from_file` allocates a new `Solver` structure and uses `loadDictionary` to fill `s->candidates` with all valid words from the given file.
- If allocation fails or no words are loaded (`loaded <= 0`), it frees the partially created solver and returns `NULL` to signal an error.
- On success, it sets `s->count` to the number of loaded words and returns a pointer to the fully initialized solver object.

Destroying the solver and reading its size

- `solver_destroy` simply calls `free(s);` to release the memory used by a solver instance when it is no longer needed.
- `solver_candidate_count` safely returns the number of remaining candidate words stored in the solver, or `0` if the pointer `s` is `NULL`.

Part 3: Guess selection and scoring

```
/* Pick best using internal freq scoring */
char *solver_select_next(Solver *s) {
    if (!s || s->count <= 0) return NULL;
    int freq[26]; compute_letter_freq(s, freq);
    long best_score = -1; int best_idx = 0;
    for (int i = 0; i < s->count; ++i) {
        int seen[26] = {0}; long score = 0;
        for (int j = 0; j < WORD_LEN; ++j) {
            char c = s->candidates[i][j]; if (c < 'a' || c > 'z') continue;
            int idx = c - 'a'; if (!seen[idx]) { score += freq[idx]; seen[idx]
            ↪  = 1; }
        }
        if (score > best_score) { best_score = score; best_idx = i; }
    }
    char *copy = malloc(WORD_BUFFER);
    if (!copy) return NULL;
    strncpy(copy, s->candidates[best_idx], WORD_BUFFER-1); copy[WORD_BUFFER-1]
    ↪  = '\0';
    return copy;
```

```c
}

/* Score a given word against current candidate letter frequencies. Higher is
↪  better. */
int score_word(const Solver *s, const char *word) {
    if (!s || !word) return 0;
    int freq[26]; compute_letter_freq(s, freq);
    int seen[26] = {0}; int score = 0;
    for (int j = 0; j < WORD_LEN; ++j) {
        char c = word[j]; if (c < 'a' || c > 'z') continue; int idx = c - 'a';
        if (!seen[idx]) { score += freq[idx]; seen[idx] = 1; }
    }
    return score;
}
```

Explanation: Guess selection and scoring

Selecting the next guess

- `solver_select_next` uses `compute_letter_freq` to obtain, for each letter, how many candidate words contain it, then evaluates every candidate word using these frequencies.
- For each candidate, it sums the frequencies of its distinct letters (tracked with a local `seen[26]`), and picks the word with the highest total score as the next guess.
- The chosen word is copied into a newly allocated buffer of size `WORD_BUFFER` and returned; if allocation fails, the function returns `NULL`.

Scoring an arbitrary word

- `score_word` applies the same frequency-based scoring scheme to any given `word`, using the current candidate set in `s`.
- It ignores repeated letters within the same word (via `seen[26]`) and returns the sum of frequency values, so words that cover common letters among candidates receive higher scores.

Part 4: Feedback application and candidate filtering

```c
/* Apply feedback pattern to shrink candidate list. */
void solver_apply_feedback(Solver *s,
```

```
                            const char *guess,
                            const char *feedback) {
    if (!s || !guess || !feedback) return;
    int write = 0;
    for (int i = 0; i < s->count; ++i) {
        if (word_matches_feedback(s->candidates[i], guess, feedback)) {
            if (write != i) {
                strncpy(s->candidates[write], s->candidates[i], WORD_BUFFER);
                s->candidates[write][WORD_BUFFER-1] = '\0';
            }
            ++write;
        }
    }
    s->count = write;
}
```

Explanation: Applying feedback and filtering

Filtering candidates using feedback

- `solver_apply_feedback` takes the last `guess` and its `feedback` pattern (e.g., greens/yellows/greys encoded as characters) and removes any candidate words that are inconsistent with that feedback.
- It loops through all current candidates and calls `word_matches_feedback(candidate, guess, feedback);` only words that match the pattern are copied to the front of the array.
- The variable `write` tracks the new size of the valid prefix of `candidates`, and at the end `s->count` is updated to reflect the reduced candidate set.

Part 5: Main loop condition

```
    printf("\033[1;35m=== Wordle Solver Demo ===\033[0m\n");
    printf("Solver will attempt to find a secret chosen from the dictionary.
    ↪  \n\n");

    int attempts = 0;
    int solved = 0;

    while (attempts < 6 && solver_candidate_count(s) > 0 && !solved) {
```

Explanation: Main loop condition

Introductory messages and state variables

- The two `printf` calls display a magenta title `"=== Wordle Solver Demo ==="` and a short description explaining that the solver will try to find a secret word from the dictionary.
- The variable `attempts` is initialized to `0` and counts how many guesses the solver has made, while `solved` is a flag that becomes `1` when the secret word is found.

Critical loop condition

- The `while (attempts < 6 && solver_candidate_count(s) > 0 && !solved)` line defines when the main solving loop continues to run.
- `attempts < 6` enforces the Wordle rule that the solver has at most six guesses.
- `solver_candidate_count(s) > 0` ensures that the loop stops if there are no candidate words left that match the feedback, meaning the solver has no logical moves.
- `!solved` guarantees that once the secret word has been found, the loop terminates immediately and no further guesses are made.

Part 6: Guess selection

```
        printf("--- Step %d ---\n", attempts + 1);
        solver_print_status(s, 8);

        char *guess = solver_select_next(s);
        if (!guess) break;
```

Explanation: Guess selection

Status display before each guess

- `printf("--- Step %d ---\n", attempts + 1);` prints the current step number (starting from 1) so the user can follow the solver's progress.
- `solver_print_status(s, 8);` shows a compact summary of the solver's internal state, for example the number of remaining candidates or a small list of the best possible words (up to 8 entries).

Choosing the next guess

- `char *guess = solver_select_next(s);` asks the solver to pick the next word to try based on the current candidate set and its heuristic.
- If `solver_select_next` returns `NULL`, the condition `if (!guess) break;` stops the loop immediately because there is no valid guess left to make.

Part 7: Feedback generation

```
char feedback[WORD_BUFFER];
getFeedback(secret, guess, feedback);

printf("Solver guess: %s   ", guess);
printf("Feedback: "); display_feedback(feedback, guess);
↪ printf("\n\n");
```

Explanation: Feedback generation

Computing feedback

- The array `feedback[WORD_BUFFER]` is a buffer where the program stores the feedback pattern for the current guess (for example letters representing green, yellow, or black squares).
- `getFeedback(secret, guess, feedback);` compares the guessed word with the secret word letter by letter and fills the `feedback` array according to the Wordle rules.

Displaying the guess and its feedback

- `printf("Solver guess: %s   ", guess);` prints the solver's chosen word for this attempt.
- The calls `printf("Feedback: "); display_feedback(feedback, guess); printf("\n\n");` print a label and then show the feedback in a Wordle-style format, followed by blank lines to keep the terminal output readable.

Part 8: Filtering logic

```
        if (strcmp(guess, secret) == 0) {
            printf("\033[1;32mSolver found the word '%s' in %d attempts!\033
            ↪  [0m\n", secret, attempts + 1);
            solved = 1; free(guess); break;
        }

        solver_filter_possible(s, guess, feedback);
        free(guess);
        attempts++;
```

Explanation: Filtering logic

Checking if the guess is correct

- `strcmp(guess, secret) == 0` tests whether the current guess matches the secret word exactly.
- If it does, the program prints a green success message with the word and number of attempts, sets `solved = 1`, frees `guess` and exits the main loop using `break;`.

Filtering remaining candidates

- When the guess is not correct, `solver_filter_possible(s, guess, feedback);` updates the solver's candidate list by removing any words incompatible with the feedback for this guess.
- After filtering, the code frees the memory for `guess` and increments `attempts` so the loop can proceed to the next try.

Part 9: Memory management and conclusion

```
    if (!solved) printf("\033
    ↪  [1;91mSolver did not find the word. Remaining candidates: %d\033[0m\n"
    ↪  , solver_candidate_count(s));

    free(secret);
    solver_destroy(s);
    return 0;
}
```

Explanation: Memory management and conclusion

Failure message when not solved

- If `solved` is still `0` after the loop, the program prints a red message telling that the solver did not find the word and shows how many candidates remain using `solver_candidate_count(s)`.

Cleaning up and ending the program

- `free(secret);` releases the dynamically allocated memory used to store the secret word.
- `solver_destroy(s);` frees all memory associated with the solver object to avoid leaks.
- `return 0;` ends `main` and signals that the program finished successfully.

Complexity of the solver main

The overall time complexity of the solver `main` function is approximately $\mathcal{O}(K \cdot N \cdot L)$, where:

- N is the number of words in the dictionary,
- L is the word length,
- K is the maximum number of attempts (here $K = 6$).

Since K is a small constant, this behaves like $\mathcal{O}(N \cdot L)$ with respect to the dictionary size and word length.

Final result example:

Solver run example in the terminal

The screenshot below shows a typical run of the Wordle solver in the terminal: the program prints the remaining candidates at each step, displays the guess and colored feedback (yellow and green squares), and finally reports that it found the secret word in a small number of attempts.

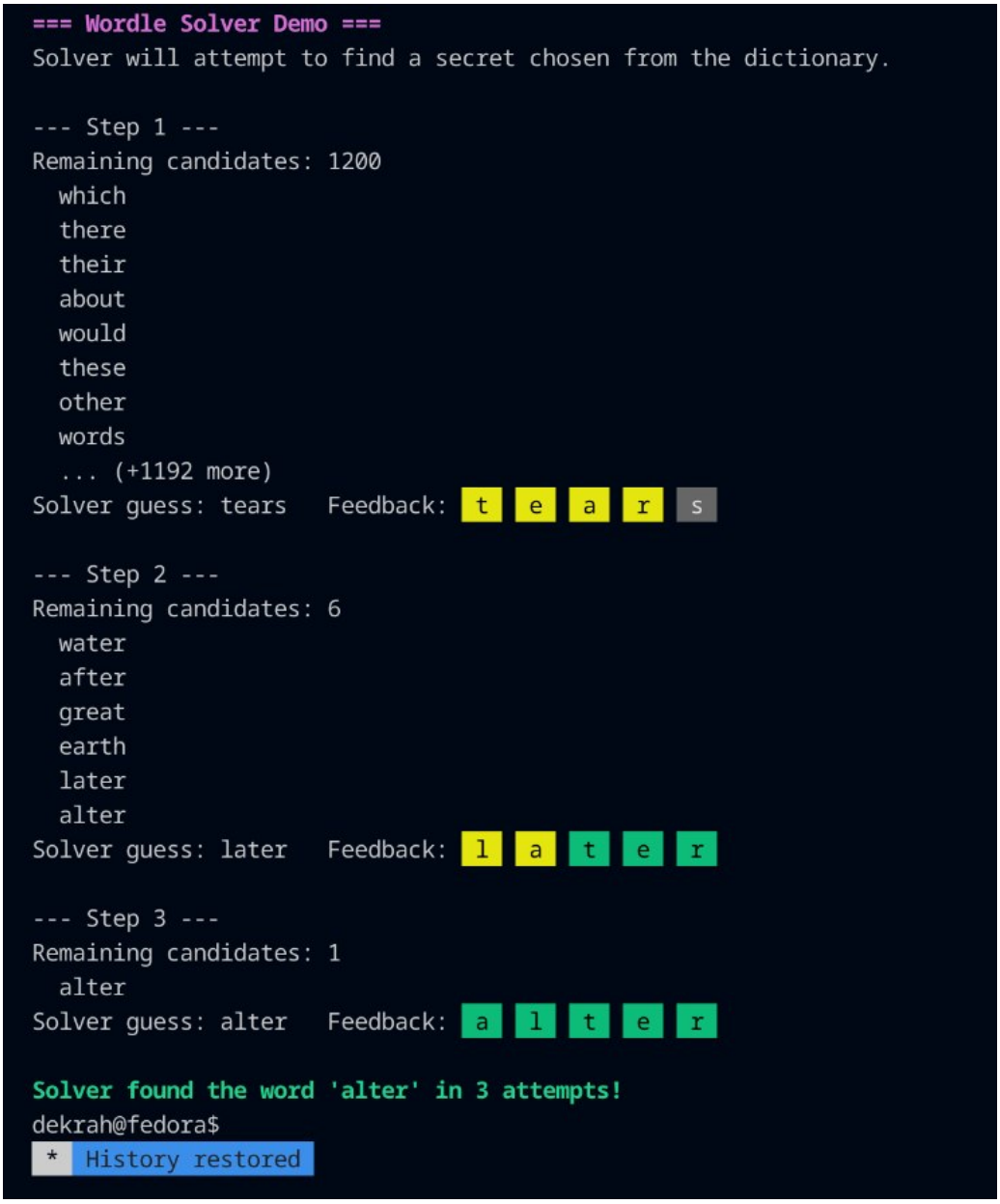

Figure 1: Final result!