Compléments d'informatique

Projet 3 : Wordle

30 novembre 2022

Dans ce projet, on vous propose d'implémenter le jeu populaire `Wordle`[1]. Vous devrez implémenter à la fois une version classique du jeu pour un joueur humain et également un algorithme de résolution automatique du jeu. L'objectif pédagogique est de vous faire manipuler des fichiers et des chaînes de caractères et surtout d'utiliser des structures de données de type liste et dictionnaire.

Ce projet est à réaliser par **groupe de deux étudiants maximum**. La date limite de remise du projet sur Gradescope est indiquée sur Ecampus.

1 Principe du jeu

L'objectif du jeu `Wordle` est de trouver un mot caché d'exactement 5 lettres. Pour trouver ce mot, vous avez droit à 6 propositions maximum. Pour chacune de ces propositions, vous obtenez une information sous la forme d'un **pattern** indiquant les lettres de votre proposition qui sont présentes dans le mot caché. Plus précisément, dans le jeu officiel, chaque lettre du mot proposé est colorée selon trois couleurs :

— gris si la lettre n'apparaît pas dans le mot caché,
— vert si la lettre apparaît dans le mot caché exactement à la même position
— jaune si la lettre apparaît dans le mot caché mais à une autre position

Si une lettre du mot caché apparaît plusieurs fois dans le mot proposé, une seule de ses apparations dans le mot proposé sera colorée en vert ou en jaune, les autres étant grisées. Si une de ces apparitions est à la même position que dans le mot caché, c'est celle-ci qui sera colorée en vert, sinon seule la première apparition de cette lettre dans le mot caché sera colorée en jaune. Dans ce projet, le pattern de couleurs sera représenté par une chaîne de 5 caractères (+ le caractère nul de fin de chaîne) : `_` indiquera un caractère grisé, `*` indiquera un caractère jaune, et `o` indiquera un caractère vert. Des exemples de patterns correspondant à différentes paires de mots caché et proposé sont illustrés à la figure 1.

1. `https://www.nytimes.com/games/wordle/index.html`

TABLE 1 – Exemples de pattern pour différentes mots cachés et proposés

Mot caché	Mot proposé	Pattern
abbey	opens	__*__
abbey	babes	**oo_
abbey	kebab	_*o**
kebab	abbey	**o*_
abbey	keeps	_*___
abbey	algae	o___*

On considérera le jeu en anglais dans le contexte de ce projet. Deux listes de mots de 5 lettres sont exploitées dans le jeu :

— une liste S_a de 2309 mots qui constituent les mots cachés potentiels (dénommés *"answers"* dans le code)

— une liste S_g de 12953 mots qui constituent les mots proposés valides (dénommés *"guesses"* dans le code) incluant évidemment les 2309 mots cachés.

2 Résolution automatique

On vous demande également d'implémenter un solveur du jeu, c'est-à-dire un algorithme proposant une séquence de mots permettant d'arriver le plus rapidement possible au mot caché. Soit l'ensemble S_a des N_a mots cachés potentiels et S_g l'ensemble des N_g mots proposés valides. L'idée d'un solveur est typiquement d'associer un score à chaque mot de S_g évaluant son potentiel comme mot proposé et à utiliser le mot de S_g de score le plus élevé comme proposition. Différents solveurs plus ou moins efficaces peuvent être définis sur base de différentes fonctions de score. Le solveur qu'on vous demande d'implémenter associe comme score à un mot w_g de S_g le nombre moyen de mots de S_a qui ne seront plus des candidats potentiels pour le mot caché si on utilise w_g comme proposition.

Plus précisément, soit $N(w_a, w_g)$ le nombre de mots de S_a qui ne sont plus des candidats potentiels pour le mot caché une fois connu le pattern entre le mot proposé w_g et le mot caché qu'on suppose être w_a. Si on note $P(w_a, w_g)$ le pattern correspondant à w_g si w_a est le mot caché, on peut se convaincre que $N(w_a, w_g)$ est donné par :

$$N(w_a, w_g) = |\{w'_a \in S_a | P(w'_a, w_g) \neq P(w_a, w_g)\}|. \tag{1}$$

En effet, tous les mots de S_a qui forment un pattern différent de $P(w_a, w_g)$ avec w_g ne peuvent plus être le mot caché. Le score $Score(w_g)$ d'une mot w_g est alors défini par :

$$Score(w_g) = \frac{1}{N_a} \sum_{w_a \in S_a} N(w_a, w_g).$$

Si $Score(w_g) = 0$, cela signifie qu'utiliser w_g ne réduira pas le nombre de candidats potentiels pour le mot cachés et donc n'apporte pas d'information sur ce mot. Au contraire, si $Score(w_g)$ est proche de N_a, l'utilisation de w_g a de grandes chances de réduire fortement l'ensemble de mots candidats.

Soit le mot caché w_a^*. L'idée du solveur est de répéter les étapes suivantes :

1. Si S_a ne contient qu'un seul mot, alors on choisit ce mot, sinon ;
2. Calculer le score $Score(w_g)$ pour tous les mots de S_g ;
3. Déterminer le mot w_g^* de score maximum et le jouer pour obtenir le pattern $P^* = P(w_a^*, w_g^*)$;
4. Enlever de S_a tous les mots w_a tels que $P(w_a, w_g^*) \neq P^*$.

La partie se poursuit ainsi jusqu'à ce que l'ensemble S_a se réduise à un seul mot, auquel cas la partie est gagnée, ou que 6 mots aient été proposé, auquel cas la partie est perdue.

3 Implémentation

Vous devez implémenter deux modules :

— Un module constitué des fichiers `wordle.c` et `wordle.h` implémentant les fonctions de base permettant d'implémenter le jeu.

— Un module constitué des fichiers `solver.c` et `solver.h` implémentant le solveur.

Nous vous fournissons pour votre implémentation :

— Un fichier `main.c` implémentant la boucle de jeu

— Une implémentation générique de liste liée dans les fichiers `LinkedList.c` et `LinkedList.h`

— Une implémentation de dictionnaire dans les fichiers `dict.c` et `dict.h`

3.1 Fichiers `wordle.h` et `wordle.c`

Ce module définit les fonctions de base permettant d'implémenter le jeu. Un type opaque `Wordle` permet de stocker des informations sur la partie, principalement la liste des mots valides et le mot caché.

Fonctions de l'interface. Les fonctions à implémenter dans le fichier `wordle.c` sont les suivantes :

`Wordle *wordleStart(char *answers_file, char *guesses_file, char *answer)` : initialise une partie. `answers_file` est le nom du fichier contenant l'ensemble S_a des mots cachés candidats et le fichier `guesses_file` l'ensemble S_g des mots proposés valides. Si l'argument `answer` vaut `NULL`, la fonction doit choisir un mot au hasard dans le fichier `answers_file` comme mot caché. Sinon, `answer` doit être forcé comme le mot caché.

`void wordleFree(Wordle *game)` : libére la mémoire prise par le structure en argument.

`char *wordleComputerPattern(char *guess, char *answer)` : renvoie le pattern correspondant au mot proposé `guess` si `answer` est le mot caché.

`char *wordleCheckGuess(Wordle *game, char *guess)` : renvoie la pattern entre `guess` et le mot caché si `guess` est un mot valide, `NULL` sinon.

`char* wordleGetTrueWord(Wordle* game)` : renvoie le mot caché. Cette fonction est introduite pour faciliter le débuggage du code mais ne doit pas être appelée par le solveur.

3.2 Fichiers `solver.h` et `solver.c`

Ce module implémente les différentes fonctions relatives au solveur. Un type opaque `Solver` permet de stocker les informations nécessaires au solveur, principalement la liste courante des mots cachés candidats et la liste des mots acceptés.

Fonctions de l'interface. Les fonctions à implémenter dans le fichier `solver.c` sont les suivantes :

`Solver *solverStart(char *answers_file, char *guesses_file)` : initialise une structure de type `Solver` à partir des fichiers de mots candidats potentiels et de mots proposés valides.

`void solverFree(Solver* solver)` : libére la mémoire prise par la structure de type solveur.

`int solverGetNbAnswers(Solver *solver)` : renvoie le nombre courant de mots cachés candidats.

`int solverUpdate(Solver *solver, char *guess, char *pattern)` : met à jour la liste de mot cachés candidats en fonction du mot proposé et du pattern correspondant (correspond à l'étape 4 du solveur ci-dessus).

`double solverBestGuess(Solver* solver, char *guess)` : calcule le mot proposé de score maximum étant donné l'état courant du solveur (correspond aux étapes 1 à 3 du solveur ci-dessus). L'argument `guess` pointe vers un espace pré-alloué (de taille 6) dans lequel le mot doit être recopié. La fonction doit renvoyer le score (maximum) correspondant à ce mot.

3.3 Autres fichiers

Un fichier `main.c` vous est fourni qui permet de créer un exécutable appelé `wordle` pour tester votre code. Lancez cet exécutable sans argument en ligne de commande pour voir les options proposées. Le premier argument permet de lancer soit le jeu manuel ou soit le solveur. Deux listes de mots, `possible_answers.txt` et `possible_guesses.txt`, vous sont fournies qui correspondent aux ensembles S_a et S_g décrits plus haut.

Les fichiers `LinkedList.c/h` contiennent une implémentation de liste liée pouvant contenir des valeurs de type `void *`. Les fichiers `dict.c/h` contiennent une implémentation de dictionnaire à clé de type `char *` et à valeur de type `double`. Consultez les fichiers header dans les deux cas pour voir les fonctions disponibles.

4 Conseils d'implémentation

Commencez par implémenter les fonctions du fichier `wordle.c` pour pouvoir jouer manuellement d'abord. L'idée de la fonction `wordleStart` est d'abord de lire les mots du fichier `guesses_file` et de les stocker dans un dictionnaire qui permettra ensuite d'implémenter efficacement la vérification faite dans `wordleCheckGuess`. Pour le chargement des mots du fichier, il y a plusieurs exemples dans le cours théorique. `wordleStart` devra ensuite choisir un mot au hasard dans le fichier `answers_file`. Une manière de faire ça est de d'abord lire l'entièreté du fichier pour connaître le nombre de mots, tirer ensuite une position au hasard et enfin aller lire le mot à cette position dans le fichier. La fonction `wordleComputePattern` est la plus délicate à implémenter. Faites en particulier attention à la manière de prendre en compte les lettres répétées.

Une fois que vous êtes sûrs que les fonctions de `wordle.c` fonctionnent correctement, passez à l'implémentation du solveur. La structure de type `Solver` est destinée à stocker une liste liée des mots du fichier `guesses_file` et une liste liée des mots du fichier `answers_file` (qui devront être parcourues dans la fonction `solverBestGuess`). Pour implémenter `solverUpdate` en quelques lignes seulement, nous vous recommandons d'utiliser la fonction `llFilter` de `LinkedList.c`.

Terminez par la fonction `solverBestGuess` qui est la plus compliquée du projet. Conformément à la description ci-dessus du solveur, l'idée de cette fonction est de réaliser une boucle sur les mots de S_g et de calculer pour chacun d'eux $Score(w_g)$. Le calcul de ce score demandera de boucler sur les mots de S_a et pour chacun d'eux de calculer $N(w_a, w_g)$ tel que défini dans l'équation 1. Pour ce calcul, vous pouvez utiliser à nouveau la fonction `llFilter` en exploitant l'argument `dryRun` (qui permet de compter les éléments de liste pour lesquels le test est vrai sans les supprimer de la liste). L'appel à cette fonction est très similaire à l'appel que vous ferez dans `solverUpdate`.

Si vous vous arrêtez à cette implémentation, vous constaterez qu'elle est extrêmement lente. Une manière d'accélérer significativement les temps de calcul est d'exploiter le fait que $N(w_a, w_g)$ a la même valeur pour tous les mots w_a qui forment le même pattern $P(w_a, w_g)$

avec w_g. L'idée pour accélérer le calcul de $Score(w_g)$ est de stocker les valeurs de $N(w_a, w_g)$ au fur et à mesure dans un dictionnaire avec comme valeur de clé le pattern $P(w_a, w_g)$. Avant de calculer un nouveau $N(w_a, w_g)$, on calculera $P(w_a, w_g)$ et on ne fera le calcul de $N(w_a, w_g)$ que s'il n'y a pas déjà une valeur associée à ce pattern dans le dictionnaire. Le cas échéant, on stockera la valeur nouvellement calculée dans le dictionnaire. Bien que beaucoup plus efficace, cette implémentation restera relativement lente en particulier pour le choix du premier mot pour lequel la liste des mots cachés candidats n'a pas encore été filtrée [2]. Néanmoins comme ce premier mot est le même quel que soit le mot caché, une option de ligne de commande permet de préciser ce mot sans devoir relancer les calculer pour pouvoir tester votre implémentation en un temps plus raisonnable.

5 Soumission

Le projet doit être soumis via Gradescope. Seuls les fichiers suivant doivent être soumis :

— `wordle.c`

— `solver.c`

— Le fichier `rapport.txt` complété avec vos réponses.

Dans ce fichier `rapport.txt` vous sont uniquement demandés les contributions de chacun au projet.

Vos fichiers seront compilés et testés en utilisant le fichier `Makefile` fourni via la commande `make all` qui utilise les flags de compilation habituels (`-pedantic -Wall -Wextra -Wmissing-prototypes`), qui ne devront déclencher aucun avertissement lors de la compilation, sous peine d'affecter négativement la cote.

Toutes les soumissions seront soumises à un programme de détection de plagiat. En cas de plagiat avéré, l'étudiant (ou le groupe) se verra affecter une cote nulle à l'ensemble des projets.

Bon travail !

2. Ce calcul du premier meilleur mot prend 20 minutes avec notre implémentation.