IN104:

Rapport du projet Wordle

Anthony Kalaydjian, Lucas Duhem

Professeur encadrant : Marine Saint Mézard mars-mai 2022

Table des matières

Introduction

Ce projet a pour but de coder et de résoudre le jeu Wordle. Ce dernier étant un jeu où le joueur doit deviner un mot de cinq lettres en plusieurs tentatives. Lors de chaque tentative, le jeu indique au joueur si les lettres de sont présentes ou non, bien placées ou non par rapport au mot à trouver. Le projet se divise donc en deux grandes parties. Dans la première, nous cherchons à coder le jeu afin que n'importe qui puisse y jouer comme s'il y jouait en ligne. Dans la seconde, nous cherchons à implémenter un algorithme qui puisse trouver la solution du jeu de manière optimale.

I Implémentation du jeu en C

Structure du programme

La version que nous avons implémentée possède plus de spécificités que la version originale. En effet, nous avons laissé le choix à l'utilisateur de personnaliser sa partie en choisissant la langue dans laquelle il veut jouer (anglais ou français), le nombre de tours qu'il souhaite jouer ainsi que le nombre de lettres pour le dictionnaire de jeu. La partie s'effectue entièrement sur le terminal, un peu plus de temps nous aurait permis de créer une interface graphique…

Le programme commence ainsi par demander à l'utilisateur le nombre de tours qu'il souhaite jouer dans sa partie, ainsi que le nombre de lettre, *word_size,* qu'il souhaite pour les mots de la partie.

On peut ainsi créer une liste des mots ayant ce nombre de lettres. Puis choisir un mot au hasard parmi cette liste, comme étant le mot à trouver.

On fait ensuite une boucle sur le nombre de tours en demandant à chaque fois le mot que l'utilisateur aura deviné.

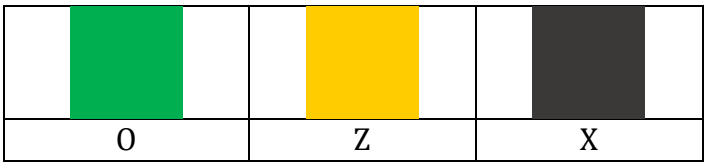
Ce mot est ensuite comparé au mot à trouver, à aide de la fonction *letter_check*, qui affichera également la combinaison associée aux deux mots en question, combinaison indiquant si les lettres du mot qu'il a choisi sont dans le mot à trouver, et à la bonne place.

Structures de données utilisées

Il est nécessaire, pour pouvoir représenter notre jeu et pouvoir effectuer des opérations sur les différents éléments de ce dernier et donc de définir un ensemble de structures qui représenteront ces différents éléments.

Pour le dictionnaire de mots à dispositions, le plus naturel est d'utiliser une liste de mots, de type char**, que l'on pourra créer à partir d'un fichier texte.

Du côté de la représentation des configurations d'un mot, on choisira, tout aussi naturellement des tableaux de lettres (char*) dont chaque lettre représente une couleur qui est affichée dans le jeu original, suivant le codage suivant :



Si la lettre du mot deviné est au même endroit que celle du mot à trouver, le caractère associé sera 'O'. Si la lettre est dans le mot à trouver, mais à la mauvaise place, on utilisera un 'Z'. Finalement, si la lettre n'est pas dans le mot, on aura un 'X'.
Ainsi, le codage ''XXOZX'' représente la combinaison
le codage ''ZZXOO'' représente la combinaison

Importation du dictionnaire

L'ouverture du dictionnaire se fera à l'aide d'un fichier texte contenant, sur chaque ligne, un mot du dictionnaire. Fichier disponible gratuitement en ligne.

Pour l'ouverture du dictionnaire, ne connaissant pas à l'avance le nombre de mots de *word_size* lettres, il partait naturel d'effectuer cette ouverture à l'aide d'une liste chaînée. Ainsi, en parcourant le fichier texte, on ajoute de manière dynamique dans notre liste les mots de *word_size* lettres. On garde également en mémoire le nombre de mots ainsi importés à l'aide d'un compteur.

Une fois que l'on a parcouru tout le fichier, on peut créer une liste de mots (*char***) à la bonne taille, dans laquelle on va introduire chacun des mots de notre liste chaînée. On peut ainsi retourner au programme parent le pointeur vers la liste contenant tous les mots de *word_size* lettres du dictionnaire, ainsi que la taille de cette liste.

Combinaison associée à deux mots

Il faut à présent réaliser la fonction *letter_check* qui prend en argument la solution (le mot à trouver) et le mot proposé par le joueur. La fonction affiche en sortie une combinaison de trois caractères: 'X' 'O' 'Z'. Par exemple: ''OXZXX'' signifie que la première lettre du mot du joueur correspond avec la première lettre de la solution tandis que la deuxième, quatrième et cinquième lettre ne sont pas présentes dans la solution. Enfin, la troisième lettre est présente dans la solution, mais pas à la même position.

Pour commencer la fonction vérifie que les deux mots on bien le même nombre de lettres, dans le cas contraire la fonction renvoie un message d'erreur.

Ensuite on crée un tableau *seen* de cinq booléens initialisés à *false*. Ce tableau a pour but de garder en mémoire les lettres qui auront déjà été traitées dans la fonction plus tard.

On crée également un tableau de caractères *affichage* qui contiendra les caractères à afficher.

On commence alors par regarder si les lettres du mot du joueur sont présentes à la bonne position. Si cela est le cas pour la lettre à la position *i* alors: *affichage[i]='*O' et *seen[i]=true*. Cette lettre est traitée, il ne faut plus la prendre en compte dans le futur.

Puis pour chaque lettre du mot du joueur on la compare avec toutes les lettres de la solution. Si on a *seen[i]==false && seen[j]==false && solution[j]=mot_joueur[i]* (avec *solution[j]*: lettre de la solution en position j), alors affichage[i]=Z. La lettre est donc à la mauvaise place mais est bien présente dans la solution. (Boucle 1)

Par ailleurs si on n'avait pas créé le tableau *seen* le code ne fonctionnerait pas. Nous allons illustrer pourquoi avec un exemple:
Si le mot à trouver est ALLER et le mot du joueur est ABATS, on devrait afficher: OXXXX. Cependant le code afficherait: OXZXX. En effet sans le tableau seen dans la boucle (1) le code comparerait la troisième lettre de ABATS avec le A de ALLER et afficherait Z car *solution[1]==mot_joueur[3]* (*'A'=='A'*).

Un autre problème qui nous avait bloqué quelque peu était de ne pas avoir posé la condition *seen[j]==false*, ce qui posait des problèmes avec les mots dont les lettres se répétaient, par exemple avec ALLEE.

II Implémentation d'un solveur de Wordle

Théorie de l'information

L'objectif du solveur de Wordle est de deviner le mot de la partie le plus rapidement possible. Une méthode efficace pour modéliser le problème est d'utiliser la théorie de l'information. Ce domaine permet de quantifier de manière probabiliste l'information que contient une partie d'un message, sur l'ensemble de tous les messages possibles.

Faisons une analogie avec l'informatique, en prenant pour espace l'ensemble des nombres en base 2 de *n* bits. On peut ainsi définir un bit d'information comme l'un des bits du nombre. Obtenir un bit d'information revient donc à déterminer l'un des bits du nombre. Cette connaissance divise l'ensemble de nombres possibles par 2. On le comprend facilement, notre espace de nombres possibles est initialement l'ensemble des nombres qui peuvent être codés en *n* bits, ce qui représente 2^n nombres. Si on connait l'un des bits du code binaire, il ne reste plus qu'à connaître les n-1 bits restants, ce qui représente 2^{n-1} bits, soit 2 fois moins de nombres possibles !

On peut raccorder cette description à notre cas. Dans le jeu Wordle, l'espace de travail est l'espace des mots d'un certain nombre lettre *word_size*. Les messages sont alors les singletons de cet espace, i.e. les différents mots qu'il contient. L'objectif du solveur sera donc de trouver, à chaque tour, le mot qui maximise l'information que l'on obtient sur le mot à trouver, suite à notre choix. On représentera l'information ainsi obtenue sous la forme de bits, comme dans l'analogie informatique.

L'acquisition de cette information se fait grâce au retour que le jeu nous fait sur un mot qu'on lui soumet. Comme vu précédemment, on appellera combinaison ce retour, qui prend la forme d'un *word_size*-uplet des trois caractères 'O', 'Z', 'X'.

En reprenant ce que l'on a expliqué plus haut, on peut écrire que :

$$N_k = \frac{1}{2^{I_k}} \cdot N_{k-1}$$

Où N_k représente le nombre de mots qui pourraient correspondre au mot à trouver, en connaissant l'information que l'on a à disposition.

I_k le nombre de bits acquis au tour k, grâce au choix de l'utilisateur.

On notera par la suite Ω_k l'espace de choix possibles au rang k.

On peut écrire ceci de manière équivalente :

$$p_k = \frac{1}{2^{I_k}}$$

Où $p_k = \frac{N_k}{N_{k-1}}$ est la fraction représentant le rétrécissement de notre espace de choix possibles au tour k, par rapport au tour k-1, après l'acquisition de la nouvelle information.

En prenant le logarithme en base 2, il vient :

$$I_k = -\log_2 p_k$$

Cette information I_k est en fait une fonction du mot proposé par l'utilisateur, de la combinaison renvoyée par le jeu, et de l'espace de mots possibles au tour *k-1*.

C'est-à-dire $I_k = I_k(mot, x, \Omega_{k-1})$

Une propriété intéressante de cette information est que contrairement aux probabilités qui se multiplient d'un tour à un autre, le logarithme en fait une fonction additive, c'est-à-dire qu'à un tour quelconque, l'information globale que l'on possède sera la somme des informations recueillies aux tours précédemment.

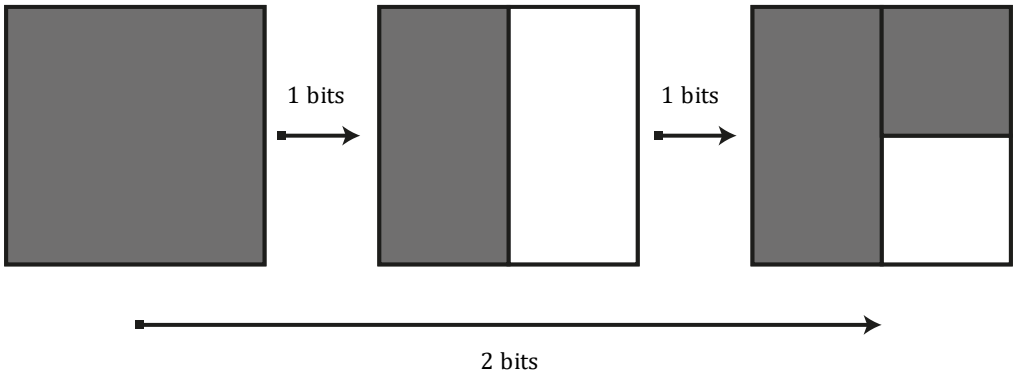


Figure 1 : *Additivité de l'information*

L'utilisateur n'a d'influence que sur le mot qu'il peut choisir au prochain tour. Pour calculer le meilleur choix possible, on peut calculer l'Esperance de l'information d'un mot au tour *k*, selon toutes les combinaisons possibles que l'ordinateur peut nous renvoyer, puis choisir le mot qui a la meilleure espérance.

En théorie de l'information, cette espérance est appelée entropie.

En supposant que p_k représente une probabilité, par rapport aux combinaisons possibles, on peut calculer l'entropie d'un certain mot :

$$E[mot] = \sum_x p_k(x) \cdot I_k(mot, x, \Omega_{k-1})$$

$$= -\sum_x p_k(x) \cdot \log_2 p_k(x)$$

Où *x* représente une combinaison possible retournée par l'ordinateur.

Pour résumer, l'objectif de l'algorithme à mettre en place sera, à chaque tour, de trouver le mot dont l'information minimise en moyenne le plus le nombre de mots restant de l'espace. On distingue deux espaces utiles : l'espace total de mots de *word_size* lettres, Ω_0 et l'espace de mots possibles au tour *k* Ω_k, sachant toute l'information que l'on a obtenu aux tours précédents. Ces espaces sont tels que $\Omega_k \subset \Omega_{k-1}$, au sens strict.

Implémentation en C

Structure du solveur

Pour le solveur, l'objectif est de donner le meilleur mot possible à chaque tour. Pour ce faire, on commence par demander à l'utilisateur les caractéristiques de la partie (langue du dictionnaire, nombre de tours et nombre de lettres par mot). On effectue ensuite une boucle sur le nombre de tours.

A chaque tour, on demande au joueur le mot qu'il a deviné, puis la configuration que lui a retournée l'ordinateur. On peut ensuite tronquer l'espace des mots pour ne garder que ceux qui sont compatibles avec cette information, puis trouver le mot (dans le dictionnaire entier) qui rapporte la plus grande entropie sur l'ensemble tronqué.

Calcul de l'entropie

Imaginons que nous soyons à un tour k quelconque.

Le calcul de l'entropie d'un mot se fait en suivant le protocole énoncé dans la partie précédente. On va ainsi sommer les entropies partielles $p_k(x) \cdot \log_2 p_k(x)$, sur l'ensemble des combinaisons *x*.

Pour ce faire, on fait une boucle sur l'ensemble des combinaisons possibles (on appellera mot original, le mot itéré). Cela revient à compter en ternaire de 0 à $3^{word\_size} - 1$ et à utiliser le caractère 'X' pour les 0, 'Z' pour les 1 et 'O' pour les 2.

Pour chaque combinaison, on peut compter le nombre de mots qui sont compatibles avec le mot original, en bouclant sur tous les mots de Ω_k et à l'aide de la fonction *is_similar* (présentée après). On peut ensuite diviser ce nombre par la taille de l'espace Ω_k pour obtenir le nombre $p_k(x)$ associé à la combinaison x, puis de calculer l'entropie instantanée $p_k(x) \cdot \log_2 p_k(x)$ associée à la combinaison *x*.

Mais il y a un problème, associé au fait que $p_k(x)$ n'est pas une probabilité. En effet, deux mots différents peuvent être compatibles avec un unique troisième mot pour une même combinaison.

Par exemple :

ARBRE est compatible avec ARMON pour la combinaison OOXXX
ARISA est compatible avec ARMON pour la combinaison OOXXX

Cette propriété implique que $\sum_x p_k(x) \neq 1$ et donc que p_k n'est pas une probabilité, ce qui discrédite le calcul de l'entropie d'un mot.

Pour pallier ce problème, on peut forcer cette sommation à 1 en utilisant une liste de mémoire booléenne qui garde en mémoire si le mot sur lequel on itère dans Ω_k a déjà été compatible (dans le passé) avec le mot original, pour une autre combinaison. On peut ainsi calculer de manière plus rigoureuse l'entropie d'un mot.

Mais cette entropie est dégénérée puisque l'utilisation de la liste de mémoire dépend de l'ordre dans lequel on traite les combinaisons, ainsi que de l'ordre des mots dans le dictionnaire, ce qui va causer d'autres problèmes…

Fonction *is_similar*

C'est sur cette fonction que repose tout l'algorithme du solveur, c'est-à-dire pour le calcul de l'entropie d'un mot.

La fonction prend en argument:

- Deux mots *(char*)*
- Une combinaison *(char*)*
- La taille des mots *word_size*

La fonction vérifie si les deux mots sont compatibles pour une combinaison donnée et renvoie *true*, dans ce cas et *false* sinon.

Cette fonction est très similaire à la fonction *letter_check*, mais possède tout de même des spécificités qui nous posé des problèmes longs à résoudre.

De la même manière que dans la fonction *letter_check*, on commence par crée le tableau *seen*. Et on traite d'abord les 'O' (correspondant au vert). En itérant sur le nombre de lettres de la combinaison, si on a un 'O' en indice *i*, on vérifie si les deux mots possèdent la même lettre à cet indice. Si ce n'est pas le cas, la fonction retourne directement *false*. Si c'est le cas alors on continue la vérification et on pose *seen[i]=true*, pour ne plus traiter ces lettres par la suite.

On réitère sur le nombre de lettres de la combinaison en cherchant les 'Z' (correspondant aux jaunes). On procède comme dans la fonction *letter_check* à la seule différence près que l'on introduit un compteur *count* qui nous permet de savoir si l'on a parcouru toute la liste. Si *count* vaut *word_size* alors on a parcouru toute la liste sans trouver la lettre en question, la fonction retourne donc false.

Dans un dernier temps, on reparcourt les lettres de la combinaison en cherchant cette fois les 'X' (correspondant au gris). Pour ceci, on parcourt les deux mots avec le même indice d'itération et on vérifie que si on trouve deux lettres identiques et qui n'ont toutes deux pas été traitées par le passé (*seen[i]==false && seen[j]==false*), dans ce cas, cet indice de la combinaison ne peut pas être 'X', on renvoit *false*. Si on parvient à parcourir tout le mot sans trouver cette contradiction, toutes les étapes étant validées, on renvoit *true*.

Problème de convergence

L'entropie que nous avons calculée, en suivant les ressources à notre disposition n'est ainsi pas l'entropie théorique. L'existence de cette dernière paraît discutable, suivant la construction que nous en avons faite. En effet, existe-t-il une bijection idéale reliant deux mots par une unique combinaison, et nous permettant de calculer l'entropie théorique ? Comment choisir convenablement cette bijection parmi toutes celles qui existent ?

N'ayant pas trouvé la réponse à cette dernière question, nous nous sommes rendu compte, en implémentant le solveur à l'aide de l'entropie dégénérée que cet algorithme ne convergeait pas. Il existe néanmoins un moyen facile pour forcer la convergence de l'algorithme. Au lieu de chercher le mot possédant la meilleure entropie dans tout le dictionnaire, on peut le chercher dans Ω_k. De plus, on peut rajouter l'option de supprimer le mot deviné de la liste tronquée une fois qu'il a été choisi à chaque tour. Ainsi, la taille de la liste tronquée est strictement décroissante, et minorée par 0. On va donc converger!

Tests de vitesse

Maintenant que notre solveur est au point, nous pouvons tester son efficacité. Après de nombreux tests, nous avons pu constater qu'il convergeait toujours, mais que la vitesse de convergence n'était pas optimale ! Bien que les meilleurs solveurs puissent trouver la solution en 3-4 tours, le notre atteint la solution en 6-7 tours, en moyenne sur le dictionnaire français et 4-5 tours sur l'anglais. Mais ceci peut être expliqué de plusieurs façons, d'abord le problème théorique que nous avons relevé, mais aussi et surtout le dictionnaire que nous prenons en compte ! Le dictionnaire que nous avons utilisé est très grand et possède de nombreux mots incongrus, ce qui augmente le nombre de tours nécessaire pour obtenir l'information nécessaire sur le mot à trouver. Un autre problème que nous avons trouvé est qu'en forçant la convergence de l'algorithme, en ne devinant que les mots dans Ω_k, il arrive assez souvent que les seuls mots restants dans Ω_k ne diffèrent que d'une seule lettre. Arrivé à ce stade, l'algorithme ne fait que deviner au hasard son mot jusqu'à trouver le bon.

Du point de vue du temps de calcul du meilleur mot à chaque étape, ce temps est relativement faible, de l'ordre de la dizaine de secondes pour les premiers tours (sans compter le meilleur mot qui sera calculé de dans un programme à part entière, après) et diminue plus on avance dans les tours.

Nous avons également comparé notre algorithme avec un algorithme plus naïf qui choisit un mot au hasard dans Ω_k. Le résultat est que, comme envisagé, ce deuxième algorithme est beaucoup plus imprédictible. Bien qu'il soit parfois plus efficace, il l'est aussi souvent moins, pouvant parfois aller jusqu'à 15 tours...

Quel est le meilleur mot ?

Pour trouver le meilleur mot, il nous suffit de calculer les entropies de chaque mot du dictionnaire par rapport à tout le dictionnaire et de choisir le mot qui a l'entropie maximale.

Dans notre cas, le meilleur premier coup est TARES, pour une entropie de 3.11. C'est-à-dire qu'en moyenne, ce mot nous permet diviser le nombre de choix possibles par $2^3 = 8$.

On peut trouver le meilleur mot possible, pour des mots comportant un autre nombre de lettres, et même pour d'autres dictionnaires :

Nombre de lettres	Meilleur mot	Entropie	Temps de calcul
3	TAS	1.54	0.155 s
4	CARS	2.3	2.98 s
5	TARES	3.1	1min 5s
6	TARIES	3.7	15min 50s

Figure 2 : *Table des meilleurs mots du dictionnaire français pour quelques nombres de lettres*

Nombre de lettres	Meilleur mot	Entropie	Temps de calcul
3	SAT	1.64	0.16 s
4	BOAS	2.21	2.79 s
5	CARES	3.0	22.8s
6	CARIES	3.6	3min 6s

Figure 3 : *Table des meilleurs mots du dictionnaire anglais pour quelques nombres de lettres*

Solutions alternatives

Si nous avions eu peu plus de temps, il y a quelques pistes parallèles que nous aurions voulu explorer.

- Calcul de l'entropie

Une idée pour résoudre le problème d'entropie dégénérée qui dépend de l'ordre des mots dans le dictionnaire aurait été de calculer l'entropie comme la moyenne des entropies dégénérées, sur plusieurs dictionnaires mélangés(contenant les mêmes néanmoins les mêmes mots). C'est une solution plausible à notre problème, mais elle engendrerait un temps de calcul multiplié par le nombre de dictionnaires ainsi utilisés.

- Autre critère d'information

On pourrait également penser à un autre critère d'information. Ayant la liste tronquée Ω_k, de tous les mots possibles au tour k, on pourrait créer une table comptant l'apparition de chaque lettre dans tous les mots de la liste tronquée. Le meilleur choix, serait alors celui qui arrive à avoir le plus de lettres qui ont le plus gros compteur dans la table.

Conclusion

Bien que ce projet ait pu paraître assez facile à réaliser dans un premier temps, la liberté dans les choix de programmation et les différents problèmes techniques et théoriques en ont fait un vrai défi. La première partie était plutôt directe dans son implémentation, hormis la fonction *letter_check*. Et on a constaté que l'ajout de quelques options de la partie était très simple à implémenter. La création d'un solveur du jeu était quant à elle beaucoup plus complexe et chronophage, que ce soit du côté théorique, avec la théorie de l'information, mais aussi du côté de l'implémentation avec la fonction *is_similar* et le calcul de l'entropie. Nous sommes tout de même parvenus à créer un programme qui fonctionne et qui battrait un humain lambda sur les dictionnaires que l'on a utilisé. Le défi a donc été relevé.