Ecole Nationale Supérieure des Techniques Avancées

Rapport de Projets

IN104

Bonnet Thomas et Valiergue Charles

19/05/2022

1 Introduction

Ce projet est articulé autour du jeu wordle. Ce jeu est l'adaptation du jeu télévisé Motus. Nous devons dans un premier temps coder le jeu afin qu'un utilisateur puisse y jouer. Dans un second temps nous devons créer un programme qui trouve à chaque fois le bon mot en un minimum de tour de jeu.

2 Développement du jeu

2.1 Objectif

Il faut coder le jeu "WORDLE" afin qu'un utilisateur puisse y jouer.

2.2 Schéma fonctionnel du jeu

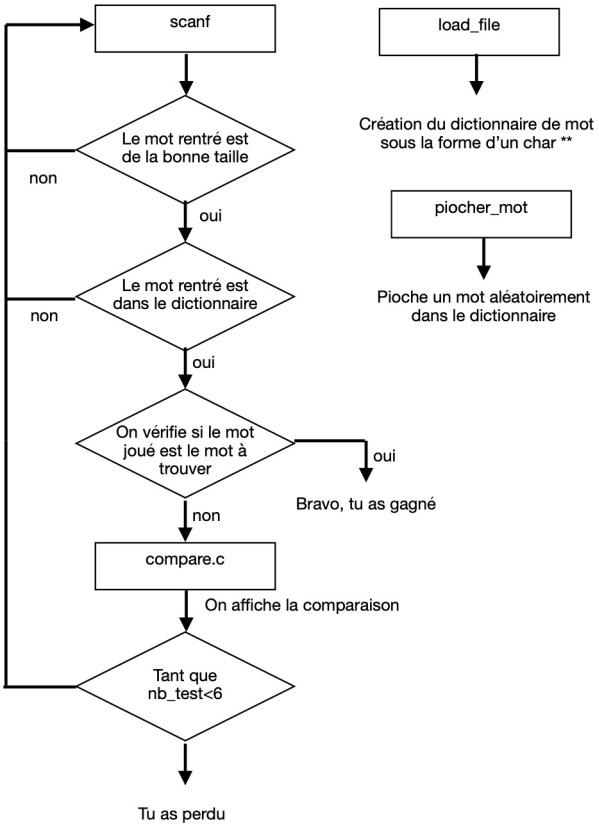

FIGURE 1 – Schéma fonctionnel du jeu

2.3 Explication de l'algorithme

Nous travaillons sur un dictionnaire trouvé sur internet en recherchant "dictionnaire français 5 mots". Nous commençons tout d'abord à mettre notre dictionnaire de mots dans un tableau de "string" afin d'y accéder plus simplement et d'éviter de lire dans un fichier. De plus, un mot est pioché aléatoirement dans le dictionnaire à l'aide de la fonction "piocher_mot". Ce mot est le mot à trouver.

L'algorithme est organisé autour d'une boucle "while" sur le nombre de tour de jeu.

L'utilisateur rentre un mot avec "scanf" ; le mot qu'il joue. Si ce mot n'est pas de la bonne taille ou alors n'est pas dans le dictionnaire (la recherche se fait par recherche dichotomique), l'utilisateur peut rentrer à nouveau un mot sans que le nombre de tour soit incrémenté de 1. Pour cela on a choisi deux conditions différentes, correspondant à 2 fonctions distinctes, qui prennent en entrée un booléen, ce booléen doit passer à True pour sortir de la boucle. Si le mot rentré est de la bonne taille et est dans le dictionnaire, on regarde si c'est le mot à trouver à l'aide de la fonction "strcmp". Si c'est le même mot, le jeu est gagné, sinon on affiche la corrélation entre les deux mots à l'aide de la fonction "compare.c". Cette fonction renvoie un tableau de 5 entiers où, si la lettre est à la bonne place c'est un 2, si elle est dans le mot mais mal placée c'est un 1 et c'est un 0 si la lettre n'est pas dans le mot. Nous avons choisi de travailler avec des tableaux de int. Le choix des tableaux a été fait afin de garder une forme compacte et réutilisable pour chaque comparaison et de pouvoir accéder plus facilement à châque comparaison interne au mot. Nous avons choisi de travailler avec des int arbitrairement, on aurait pu choisir {a,b,c}. Un tableau de cinq int est initialisé à 0, c'est le tableau qui sera renvoyé. Nous regardons premièrement si les lettres du mot sont bien placées car c'est la condtion la plus forte. Ensuite nous regardons les lettres qui ne sont pas bien placées si elles sont dans le mot ou non. Nous gérons le problème des doublons en utilisant un int[5] initialisé à [00000] pour chaque mot. Si le i-ème chiffre est modifié, cela signifie que la i-ème lettre du mot à trouver a déjà été utilisée dans une comparaison valant 1 ou 2. Si le nombre de tours de jeu est inférieur à 6, on retourne au début de la boucle, sinon, l'utilisateur a perdu.

2.4 Commentaires

Notre programme affiche simplement les corrélations entre les mots joués et le mot à trouver si le mot joué n'est pas le bon. De plus, il affiche si l'utilisateur doit rentrer à nouveau un mot suite à une entrée non conforme au dictionnaire utilisé. Il affiche aussi si l'utilisateur a gagné ou a perdu à l'issue de sa partie.

Cet affichage est suffisant puisqu'il nous est seulement demandé de créer une interface pour un utilisateur.

Nous n'avons jamais eu d'erreur avec notre programme de jeu, il renvoie à chaque fois ce qu'il faut.

Nous avons choisi un dictionnaire sur internet contenant une liste non exhaustive des mots français de cinq lettres.

3 Programmation d'un ordinateur afin qu'il joue le meilleur mot possible à chaque étape

3.1 Objectif

L'objectif de cette partie est de faire jouer un ordinateur de la meilleure manière possible.

3.2 Schémas fonctionnels

L'algorithme final comprend la fonction ia.c et la fonction new word.c. Nous allons donc décrire l'algorithme final et ces deux fonctions.

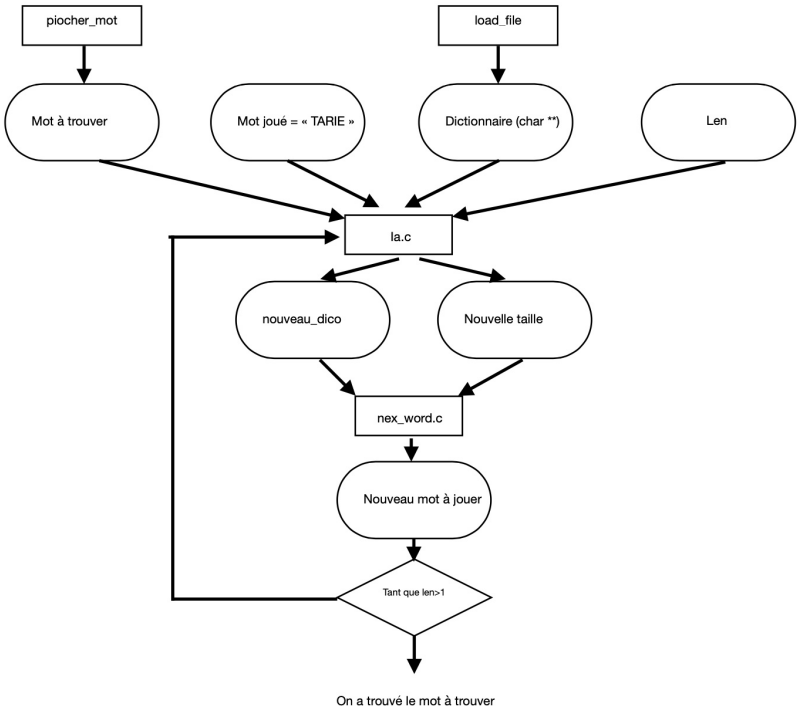

FIGURE 2 – Schéma fonctionnel de l'algorithme final

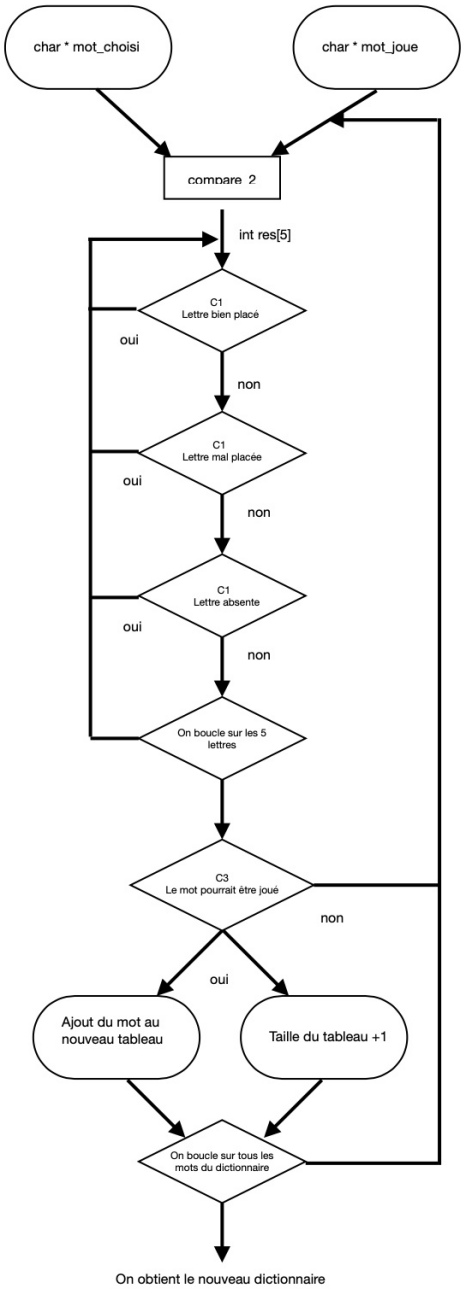

FIGURE 3 – Schéma fonctionnel de ia

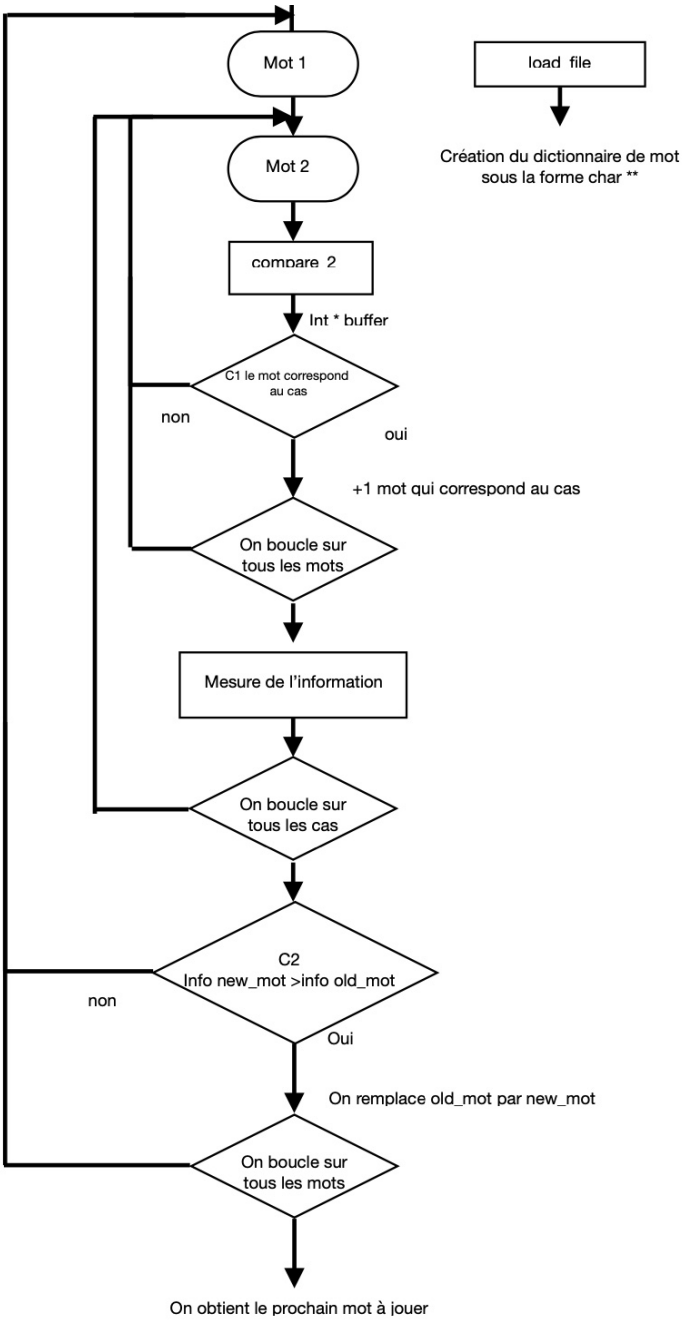

FIGURE 4 – Schéma fonctionnel de next_word

3.3 Explication de l'algorithme

Tout d'abord, il faut trouver le premier mot à jouer. Pour cela, nous avons créé une fonction qui calcule l'occurence des lettres dans notre dictionnaire. Nous avons pris les lettres les plus présentes et nous avons construit un mot contenu dans le dictionnaire. Nous avons trouvé "TARIE". Initialement nous avions choisi de procéder par calcul de l'information, (d'appliquer la même fonction que next_word) mais à tout le dictionnaire, seulement sur les 7980 boucles, la fonction s'arrêtait au bout de 76 tours. On a remarqué que cela nécessitait trop de mémoire vive, c'est pourquoi, malgré l'efficacité certaine de cette technique, nous avons du nous rabattre sur une méthode moins coûteuse, mais renvoyant un résultat très concluant.

Le dictionnaire choisi est toujours le même. Nous mettons aussi notre dictionnaire dans un tableau de mots afin d'y accéder facilement et de le parcourir avec deux indices simultanément.

Ensuite, il nous faut le mot à trouver. Donc nous piochons un mot aléatoirement dans notre tableau de mot.

L'algorithme final prend en entrée le tableau de mots et sa taille. Il repose sur une boucle $while(taille > 1)$. En effet , nous ne pouvons pas savoir à l'avance combien de tours de jeu notre algorithme va faire. De plus, nous avons choisi de ne pas jouer au jeu (c'est à dire de stopper la recherche au bout de 6 tours) afin de tester l'efficacité de notre algorithme et d'effectuer une moyenne sur le nombre de tours nécessaire pour trouver le mot. Le mot joué est initialisé à "TARIE". On a décidé de baser son fonctionnement autour de 2 fonctions détaillées ci-dessous, ia.c qui renvoie les mots pouvant encore être éligible, et next_word.c qui donne le prochain mot à jouer.

La fonction "ia.c" prend en entrée le mot à trouver, le mot joué, le tableau de mot et la taille du tableau et renvoie un tableau de mot, contenant uniquement les mots pouvant encore être joué. Dans un premier temps, on compare le mot joué avec le mot à trouver avec la fonction compare2, qui est identique à "compare" mais renvoie le tableau d'entiers au lieu de l'afficher. Nous avons ensuite décidé de boucler sur tous les mots du dictionnaire en regardant s'ils peuvent toujours être joués. Pour cela nous avons décidé d'utiliser un compteur réinitialisé à chaque nouveau mot. Si ce compteur vaut 5 une fois chaque lettre parcourue, c'est que le mot peut être joué et il est ajouté au tableau.

Nous nous sommes heurtés à 2 problèmes dans cette fonction. On s'est aperçu qu'en affichant notre nouveau dictionnaire, certains mots ne pouvaient pas être la solution, mais étaient quand même dans la liste. Cela survenait dans le cas où le mot à trouver avait une lettre en double voire plus. Dans ce cas en effet, comme lorsqu'on regardait une lettre mal placée on balayait nécessairement

tous le mot, le compteur pouvait s'incrémenter de 2 en une seule fois. Pour y remédier, on a mis une condition d'arrêt afin de sortir de la boucle dès que qu'une lettre est trouvée.

La résolution de ce problème en a mis en avant un second. Celui-ci survient lorsqu'une lettre, qu'on attend mal placé dans le mot, est bien placée dans le mot sélectionné, le compteur va quand même s'incrémenter de 1, alors qu'on ne souhaite pas ce mot. On a donc ajouté une condition supplémentaire, qui est de ne pas regarder la lettre positionnée au même endroit.

Exemple : Le mot à trouver est "HABIT"
Le mot que je joue est "TARTE"
La combinaison renvoyée par compare2 est donc [12000] Dans ce cas, sans la 2e condition, un mot comme "TACLE" serait dans mon nouveau dictionnaire car il possède bien un "A" bien placée et les lettres "C,L,E" n'appartiennent pas au mot cherché. Seulement ce mot ne peut pas être le bon puisque le mot recherché à certe un "T" dans son mot, mais pas en première postion.

Maintenant, la fonction "next_word.c" est appliquée à ce nouveau dictionnaire. Cette fonction prend en entrée un dictionnaire sous la forme d'un tableau de mots, sa taille et renvoie le meilleur mot à jouer de ce dictionnaire. On se base sur la théorie de l'information. Nous avons recensé 243 cas possibles pour une comparaison entre deux mots de cinq lettres (3^5). Chaque cas est représenté en écriture ternaire afin de faciliter les comparaisons entre nos cas et les résultats de "compare_2.c". Nous avons décidé de représenter ces cas à l'aide d'un int[243][5] afin de pouvoir accéder simplement à chaque cas. Cette fonction boucle sur tous les mots du dictionnaire. Ensuite elle boucle sur tous les cas de comparaison. Enfin elle reboucle sur tous les mots du dictionnaire (on a donc besoin de deux pointeurs sur notre dictionnaire ce qui montre bien qu'on avait besoin de le représenter sous la forme d'un tableau). Pour chaque mot, elle va donc calculer la fréquence d'appartion de chaque cas en comparant ce mot avec tous les mots du dictionnaire. Ensuite, elle nous donne l'information apportée pour chaque cas. Si la moyenne de ces informations est supérieure à la moyenne précédentes alors le mot courant devient le meilleur mot à jouer. Ici nous décidons de remplacer le mot à jouer à chaque fois par le nouveau et non pas de stocker toutes les informations dans un tableau et de prendre le meilleur afin d'économiser de l'espace de stockage. En effet, dans cette fonction l'économie de mémoire vive est importante, nous avions eu des problèmes de mémoire avec cette fonction.

Dans l'algorithme final, tant que la taille du tableau est plus grande que 1 alors on reboucle, sinon, la taille du tableau vaut 1, dans ce cas le tableau contient le mot trouvé et on sort de la boucle.

3.4 Commentaires

Notre algorithme final affiche le mot à trouver, le mot final et le nombre de tour de jeu que l'algorithme fait. Parfois, le nombre de tour de jeu est supérieur à 6, notre algorithme n'est donc pas parfait puisqu'il ne gagne pas à chaque fois. Sur 100 tentatives nous avons une moyenne de 3,6 tours de jeu, ce qui est plutôt correct. Or sur ces 100 tentatives nous avons eu 4 segfault. Nous n'avons pas réussi à corriger ce problème. Si plus de temps nous était alloué, nous aurions essayé de pallier à ce dernier.

Parfois, le temps d'exécution de notre algorithme est long, cela est surement du à la mauvaise complexité spatiale de "next word.c". En effet, si le mot "TARIE" ne réduit pas énormément le dictionnaire initial, "next word.c" prend en entrée un tableau de taille conséquente.

Nous pensons donc que la principale source d'amélioration est notre fonction "next word.c" qui a une complexité temporelle en $O(n^2)$. De plus, la fonction "ia.c" peut aussi être améliorée. En effet, nous comparons à chaque fois les cinq lettres alors que si une lettre une lettre ne corrspond pas à ce qu'on attend, on pourrait sortir de la boucle en utilisant un break. Cela améliorerait les complexités spatiales et temporelles de "ia.c". On a identifié que les "segfault" provenaient de cette fonction, qui renvoie pour certains mots un dictionnaire vide, nous n'avons pas réussi à résoudre ce problème.

4 Conclusion

Finalement, les objectifs principaux du projet ont été réalisés. En effet on a bien dans un premier temps un jeu fonctionnel et dans un deuxième temps un programme qui renvoie le mot à trouver à tous les coups. Comme piste d'amélioration, nous pourrions rendre l'interface du jeu plus ludique et agréable, ou par exemple faire en sorte de prendre en compte le code des lettres majuscules et de donner une condition que si le code des lettres rentrées est décalé de 26 (qu'on saisi un mot en miniscule) alors le jeu reconnait le mot. Afin d'améliorer notre algorithme de résolution du jeu, nous pourrions essayer d'écrire un algorithme necessitant moins de mémoire vive, donc de diminuer le nombre de calcul.