TELECOM NANCY

# WORDCHAMP & SOLVCHAMP
## Wordle

*Auteurs :*
Malo DAMIEN
Louis-Vincent CAPELLI
Jules BRUNET
Tristan SMAGGHE

*Responsable de module :*
Olivier FESTOR

UNIVERSITÉ DE LORRAINE

8 juin 2022

# Table des matières


**1 Présentation du projet** **1**
1.1 Introduction au sujet . . . 1
1.2 Présentation de Wordchamp . . . 1

**2 État de l'art** **2**
2.1 Le jeu Wordle . . . 2
2.2 Les variantes du jeu . . . 2
2.3 Résoudre le Wordle . . . 2

**3 Application Web** **4**
3.1 Intégration en continue . . . 4
3.2 Concept et structure du site . . . 4
3.2.1 Nom et concept du site . . . 4
3.2.2 Structure du Site . . . 4
3.2.2.1 Frontend : VueJS . . . 5
3.2.2.1.1 VueJS 3 . . . 5
3.2.2.1.2 Vuetify . . . 5
3.2.2.1.3 Vuex Router . . . 5
3.2.2.1.4 Axios . . . 5
3.2.2.2 Backend : Flask . . . 5
3.2.2.2.1 JSON Web Tokens . . . 5
3.3 Partie Base de Données . . . 6
3.3.1 SQLAlchemy . . . 6
3.3.2 Particularités de certains enregistrements . . . 6
3.3.2.1 Dates . . . 6
3.3.2.2 Cas particulier des tables "game", "game_normal" et "game_carriere" . . . 6
3.4 Fonctionnalité du site . . . 7
3.4.1 Wordle . . . 7
3.4.1.1 Choix des paramètres . . . 7
3.4.1.2 Le jeu . . . 8
3.4.1.3 Gestion de la soumission de réponse . . . 8
3.4.1.4 Algorithme de vérification du mot . . . 8
3.4.2 Wordle - Partie Classée . . . 8
3.4.2.1 Présentation du mode de jeu . . . 9
3.4.2.2 Calcul de la difficulté d'un mot . . . 10
3.4.2.3 Explication du système d'élo . . . 11
3.4.3 Classement . . . 12
3.4.4 Historique . . . 12

**4 Solveur** **14**
4.1 Introduction . . . 14
4.2 Solveur avorté . . . 14
4.2.1 Arbre de force brute . . . 14
4.3 Wordle en C . . . 15
4.4 Solveur par minimisation d'entropie . . . 15
4.4.1 Notion d'entropie . . . 15
4.4.2 Principe et fonctionnement du solveur . . . 16
4.4.3 Implémentation . . . 16
4.4.3.1 Algorithme principal . . . 16

4.4.3.2 Première version du solveur . . . 16
4.4.3.3 Seconde version du solveur . . . 18
4.4.3.4 Troisième version du solveur . . . 18
4.4.4 Limites de ces implémentations . . . 19
4.5 Solveur naïf . . . 19
4.5.1 Motivations . . . 19
4.5.2 Structures de données . . . 19
4.5.3 Fonctionnement . . . 20
4.5.4 Abandon . . . 20
4.6 Solveur par analyse fréquentielle et rapprochement d'un mot idéal . . . 20
4.6.1 Principe et algorithme . . . 20
4.6.2 Structure . . . 20
4.6.3 Analyse Fréquentielle . . . 20
4.6.4 Calcul du Score . . . 20
4.6.5 Filtrage en fonctions des informations récupérées . . . 21
4.6.6 Vérification précision des mots comparaison avec premier solveur . . . 23
4.7 Solveur naïf avec beaucoup de structures . . . 23
4.7.1 Motivation . . . 23
4.7.2 Structures utilisées . . . 23
4.7.2.1 Mot . . . 23
4.7.2.2 Lettre . . . 23
4.7.2.3 Liste . . . 24
4.7.2.4 Table de hachage . . . 24
4.7.3 Fonctionnement . . . 24

**5 Testeur** **25**
5.1 Présentation . . . 25
5.2 Principe de fonctionnement . . . 25
5.3 Choix du language . . . 25
5.4 Problèmes rencontrée . . . 26
5.4.1 Concurrence . . . 26
5.4.2 Sortie en tampon . . . 26

**6 Test et Performance** **28**
6.1 Introduction . . . 28
6.2 Test et Performance . . . 28
6.3 Nos conclusions . . . 28

**7 Gestion de Projet** **31**
7.1 Équipe de projet . . . 31
7.2 Analyse de Projet . . . 31
7.2.1 Définition des objectif . . . 31
7.2.2 Analyse des risques : Matrices SWOT . . . 31
7.3 Organisation du projet . . . 32
7.4 Outils de travail . . . 32
7.4.1 IDE . . . 32
7.4.2 Logiciel de gestion de version . . . 32
7.4.3 Rédaction du rapport . . . 32
7.4.4 Communication . . . 32

**8 Bilan** **33**
8.1 Bilan du travail réalisé . . . 33
8.2 Ressentis personnels . . . 33
8.2.1 Jules BRUNET . . . 33
8.2.2 Louis-Vincent CAPELLI . . . 34
8.2.3 Malo DAMIEN . . . 34
8.2.4 Tristan SMAGGHE . . . 34
8.3 Taux horaires . . . 35

**Annexes** **37**

**Annexe 1** **37**

**Annexe 2** **40**

**Annexe 3** **41**

**Annexe 4** **48**

**Annexe 5** **49**

# Chapitre 1

# Présentation du projet

## 1.1 Introduction au sujet

WORDLE est un jeu en ligne basé sur l'émission Lingo (Motus en France) qui a connu un gain de popularité en Janvier 2022. Ce gain soudain est probablement dû au fait que le jeu est simple à comprendre, jouable partout et partageable sur les réseaux sociaux. Les règles sont simples : il faut deviner un mot via plusieurs tentatives et le jeu indiquera pour chacune d'entre-elles la position des lettres bien placées et mal placées. Ainsi, il est de notre devoir que de représenter le coq Français en reconstituant le jeu (à notre manière) et en proposant des mots de la langue de Molière.

## 1.2 Présentation de Wordchamp

Nous avons donc décidé de développer un site web dont le nom est : Wordchamp. Il proposera évidemment le jeu de base du Wordle en proposant à l'utilisateur de personnaliser sa partie (taille des mots et nombres d'essais) mais aussi une version compétitive du jeu qui propose des mots de plus en plus compliqués à l'utilisateur. Ce dernier aura le choix de lancer une partie d'entraînement ou une réelle partie compétitive (afin de prendre soin de son rang). Les utilisateurs pourront personnaliser leur profil et accéder au classement global.


FIGURE 1.1 – Logo du site

# Chapitre 2

# État de l'art

## 2.1 Le jeu Wordle

Wordle est un jeu assez simple, il faut deviner un mot avec un nombre limité de tentatives. À chaque tentative, le jeu nous retourne des informations en colorant les lettres correctement placées, les lettres présentes dans le mot mais mal placées et enfin en laissant sans couleur les lettres qui ne sont pas présentes dans le mot. Précisément, si le joueur entre un mot avec deux mêmes lettres, et que cette lettre apparaît une seule fois dans la solution, le jeu indiquera une seule des deux lettres comme étant bien ou mal placée. Ce système permet au joueur d'acquérir des informations sur la composition du mot au fur et à mesure de ses essais. Plusieurs stratégies plus ou moins efficaces ont été développées par les joueurs : notamment commencer chaque partie par les mêmes mots composés de lettres fréquentes de la langue (par exemple des voyelles), toutes différentes. D'autres essaient d'approcher le mot au fur et à mesure en ne proposant que des mots qui valident les informations récupérées par leurs précédentes tentatives. Suite au succès de ce jeu de nombreuses variantes sont apparues suivant le même mécanisme de "gagner de l'information" à chaque tentative.

## 2.2 Les variantes du jeu

Il y a tout d'abord les versions françaises du jeu : SUTOM et TUSMO. Un autre jeu nommé CÉMENTIX donne à chaque tentative un pourcentage de proximité avec le mot recherché, en se basant sur les sonorités du mot mais également sur le sens du mot (les synonymes et mots ayant des sonorités proches auront alors un haut pourcentage de corrélation). D'autres versions un peu plus exotiques existent également avec Heardle qui propose chaque jour une musique connue en ne laissant que la première seconde de celle-ci, il faut alors deviner le titre et l'artiste correspondant. Suivant le principe évoqué plus tôt, à chaque tentative la durée de l'extrait augmente. Une version pour les amateurs de géographie existe aussi : Worldle qui donne un pourcentage de proximité géographique avec le pays à chaque tentative. En clair : le principe sur lequel se base le jeu est déclinable de bien des manières avec pour seule limite l'imagination des développeurs.

## 2.3 Résoudre le Wordle

L'autre axe majeur de ce projet, c'est la mise en place d'un solveur qui a pour objectif de trouver la solution le plus vite possible (ou à minima trouver la solution tout court).

Cette démarche s'inscrit dans un courant bien plus large, celui de vouloir "résoudre" des jeux de réflexion créés par l'Homme. Plus précisément, cela consiste à trouver les manières les plus optimales de jouer pour gagner / atteindre le meilleur score possible. L'exemple le plus simple est le morpion, où même un enfant peut apprendre par coeur quelle est la meilleure option pour une position donnée. De nombreux autres jeux tombent dans cette catégorie, comme par exemple les échecs, le jeu de go, le puissance 4, ou encore le jeu de Nim pour ne citer que les plus connus.

On peut alors classer les jeux que l'on souhaite résoudre selon de nombreux critères, mais le plus important est le suivant : le jeu est-il déterministe ?

La notion de déterminisme est assez simple à comprendre : si, à un état donné, et pour une action donnée, l'état résultant du jeu est toujours le même, alors il est déterministe (en clair : il ne fait pas intervenir de hasard). C'est bien le cas du Wordle.

La théorie des jeux [10] fournit de nombreux autres critères de classification, mais cela s'éloigne de l'objectif que l'on se fixe ici (on s'intéresse à des solutions algorithmiques et non nécessairement mathématiques).

Puisque le Wordle est déterministe, il n'est pas nécessaire d'avoir une approche probabiliste pour évaluer quoi faire dans chaque état, ce qui facilite les différentes implémentations. Ainsi, c'est dans ces conditions que l'on va s'intéresser à différents solveurs qui ont déjà été imaginés :

— L'approche Greedy-min-max [5] : lorsque l'on souhaite trouver la solution d'un Wordle, cela revient en fait à réduire au fur et à mesure la liste des mots possibles pour qu'il n'en reste plus qu'un. Ainsi, on cherche à faire diminuer la longueur de cette liste de la manière la plus optimale possible. L'idée de cette approche, c'est pour chaque tentative, de tester l'ensemble des mots proposables pour ensuite choisir celui qui, dans le pire des cas, aura la liste de mots possibles la plus courte. Ainsi on avance vers la solution en s'assurant une marge de progrès minimum à chaque tentative.

— L'approche Average-minimize [5] : presque semblable à l'approche précédente, on effectue les mêmes calculs à chaque tentative, sauf que l'on sélectionne le mot qui en moyenne élimine le plus de possibilités, au lieu de vouloir avoir le "meilleur pire cas". Cette méthode donne, en moyenne, de meilleurs scores, même si elle peut avoir plus de mal sur certains mots spécifiques.

— L'approche de la "minimisation d'entropie" [2] : se basant sur la Théorie de l'information [9], et plus précisément sur la notion d'Entropie de Shannon [8], cette méthode permet de quantifier la quantité d'information que chaque proposition va apporter, et d'ensuite sélectionner le mot permettant de réduire au maximum la quantité d'information incertaine (entropie) restante. Pour bien se le représenter, on peut imaginer un exemple : si un Wordle peut avoir 16 mots différents comme solution, on dira que l'incertitude liée à la solution vaut $log_2(16) = 4$. En effet, pour s'assurer d'avoir trouvé le bon mot, il faut diviser 4 fois par 2 l'espace de départ pour n'avoir qu'une unique possibilité restante. Cet algorithme s'assure donc de trouver le mot qui permettra, à chaque étape, de diminuer au maximum l'entropie de l'ensemble.

— Une approche basée sur l'intelligence artificielle [6] : sans rentrer dans les détails, cet algorithme consiste à entraîner une intelligence artificielle selon le principe du Deep Learning de manière à ce qu'elle apprenne d'elle-même à battre le jeu. C'est une approche qui a déjà été envisagée sur de nombreux autres problèmes du même style, mais qui reste assez complexe à mettre en oeuvre.

— Une approche plus heuristique [4] : en se basant sur la fréquence d'apparition des lettres à chaque position parmi la liste des solutions possibles, on peut créer un mot "type" (par exemple, si presque tous les mots restants ont un e en dernière position, notre mot type finira par un e). Ensuite, pour chaque mot proposable, on va calculer son erreur par rapport à ce mot type, pour choisir celui qui minimise cette erreur. Cette approche, bien que simple à première vue, permet tout de même d'obtenir des résultats très satisfaisants.

Ainsi, comme nous l'avons vu, il existe de nombreuses manières d'envisager un solveur, au-delà même de celles présentées ici. Il en existe encore sûrement beaucoup d'autres (par exemple, on pourrait avoir une approche plutôt axée sur la science des données et s'amuser à analyser les parties jouées par des milliers de joueurs, pour ensuite travailler sur les fréquences de victoires selon les mots employés...). C'est un problème ouvert et il n'existe pas réellement de solution ni d'algorithme optimal : ils ont tous leurs propres avantages et inconvénients.

# Chapitre 3

# Application Web

## 3.1 Intégration en continue

Afin de pouvoir vérifier facilement les différentes versions du projet, et pour pouvoir vérifier depuis n'importe quelle plateforme, nous avons mis en place une intégration en continue. Elle utilise Docker, logiciel permettant de créer des conteneurs qui ne dépendent pas de la machine hôte (évite le "mais ça fonctionne sur ma machine"). Son fonctionnement est assez simple :

- Lorsqu'un membre du groupe décide de publier une nouvelle version il crée un "Tag" sur Gitlab (plateforme qui héberge le projet) ;
- L'intégration en continue se lance alors.
- Dans un premier temps, les fichiers sont envoyés à un "runner" (machine qui se charge de la compilation).
- Ensuite cette machine crée l'image du projet (permet de lancer plus tard un conteneur).
- Puis celui-ci publie cette image sur Gitlab (qui reste privée).
- Ensuite, le "runner" envoie un message à notre serveur.
- Ce dernier télécharge alors l'image, et relance le conteneur du projet.
- Le projet devient alors accessible sur `wordchamp.info`.

## 3.2 Concept et structure du site

### 3.2.1 Nom et concept du site

Nous avons décidé d'orienter notre version du Wordle vers la compétition afin qu'il sorte un peu de l'ordinaire. C'est ainsi que nous est venu son nom : WordChamp, composé du préfixe Word- en hommage au jeu original et pour rappeler l'univers des mots et du suffixe -Champ pour "champion" car c'est le but de notre jeu : devenir le champion. WordChamp reprend le principe de base du jeu Wordle en y instaurant un classement et un système d'elo pour classer les joueurs et leur proposer des challenges à leur hauteur.

### 3.2.2 Structure du Site

FIGURE 3.1 – Vue JS et Flask

#### 3.2.2.1 Frontend : VueJS

Nous avons décidé de diviser le site en deux pour faciliter la répartition des tâches. VueJS est un framework qui sert à construire des interfaces utilisateur. Il est aussi basé sur des composants qui permettent d'intégrer facilement plusieurs parties indépendantes. Ainsi, nous avons utilisé différents plugins pour le frontend.

##### 3.2.2.1.1 VueJS 3

Comme dit précédemment, VueJS nous permet de dynamiser les pages sans utiliser autant de Javascript que si nous voulions le faire nativement. L'exemple ci-dessous est un composant qui contient un bouton dont la valeur augmente lorsqu'il est cliqué.

```
<script setup>
import { ref } from 'vue'

const count = ref(0)

function implement() {
  count.value++
}
</script>

<template>
  <button @click="implement">count is: {{ count }}</button>
</template>
```

##### 3.2.2.1.2 Vuetify

Vuetify est un framework équivalent à Bootstrap mais qui fournit des composants pour VueJS.

##### 3.2.2.1.3 Vuex Router

Vuex Router est un plugin pour VueJS qui nous permet de facilemenet intégrer le stockage local à notre implémentation. Cela va nous permettre de sauvegarder la session de l'utilisateur que l'on verra aussi dans le backend.

##### 3.2.2.1.4 Axios

Axios est une librairie Javascript qui permet de faire des requêtes depuis le frontend. On l'utilise afin de faire les requêtes vers l'API car elle permet de simplement injecter le jeton de connexion.

#### 3.2.2.2 Backend : Flask

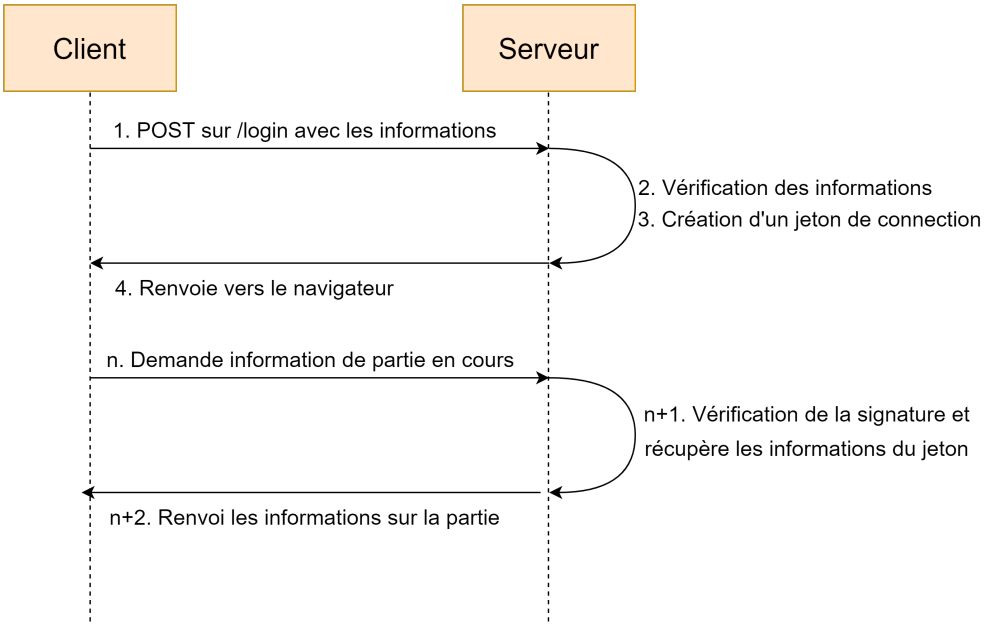
Puisque nous n'avons pas besoin de faire de rendu avec Flask, il sera utilisé ici afin de créer une API qui recevra des appels depuis le frontend. Le frontend étant côté client, il n'est pas "de confiance" et nous devons nous assurer que les bonnes personnes puissent faire les bonnes requêtes sans que les mauvaises personnes n'aient accès à leurs pages. Les seules particularités du backend sont les routes de connexion qui vont utiliser des JSON Web Tokens.

##### 3.2.2.2.1 JSON Web Tokens

Les JSON Web Tokens ou JWT sont des standards définis dans la RFC 7519 et permettent d'échanger des jetons sécurisés entre plusieurs paires. Ils sont composés en trois parties :

- En-tête : Décrit le jeton et la sécurité en place.
- Charge utile : Informations contenues, numéro de session et temps d'expiration (décodable par n'importe qui mais pas altérable).
- Signature numérique utilisant l'algorithme de l'en-tête.

Voici comment nous utilisons les JWT :




Nous utilisons aussi des jetons de rafraîchissement qui servent à rafraîchir la session lorsque le token de session a expiré.

## 3.3 Partie Base de Données

### 3.3.1 SQLAlchemy

Comme pour notre premier projet, nous avons décidé d'utiliser SQLAlchemy, qui est une librairie python permettant un accès efficace aux bases de données. En fait, on manipule des classes python pour traiter les données, et SQL Alchemy traduit ces classes en tables SQL, ce qui facilite l'intégration du traitement des données dans le code python.

Ci-joint l'exemple de la déclaration d'une table dans un fichier python utilisant SQLAlchemy.

```
class User(db.Model, UserMixin):
    __tablename__ = 'users'
    id = db.Column(db.String(40), primary_key=True)
    username = db.Column(db.String(80), unique=True, nullable=False)
    password_hash = db.Column(db.String(128), nullable=False)
    admin = db.Column(db.Boolean, nullable=False)
    points = db.Column(db.Integer, nullable=False)
    elo = db.Column(db.Float, nullable=False)
```

Listing 3.1 – Fichier python user.py (raccourci)

Comme on peut le remarquer, cela nous permet d'associer des méthodes à la classe User, par exemple on utilisera la méthode set_password() pour générer le hash du mot de passe. Cela permet une gestion plus claire des profils utilisateurs et un code plus propre ce qui est très appréciable pour un gros projet.

Voici l'exemple d'une requête SQL réalisée à l'aide de SQLAlchemy.

```
utilisateur = User.query.get(id='351464456645645')
utilisateur.username -> Contient le nom d'utilisateur
```

La liste de nos tables, des attributs, ainsi que le schéma entités-associations sont dans l'Annexe 1.

### 3.3.2 Particularités de certains enregistrements

#### 3.3.2.1 Dates

Afin d'enregistrer les dates, nous avons choisi d'enregistrer des horodatages en millisecondes écoulées depuis l'epoch UNIX (1er Janvier 1970). Ceci permet à la fois d'enregistrer un entier, mais permet aussi (si besoin) de pouvoir échanger ces horodatages facilement.

#### 3.3.2.2 Cas particulier des tables "game", "game_normal" et "game_carriere"

La gestion de ces trois tables doit se faire différemment, puisque game_normal et game_carriere sont des spécialisations de la table game. La manière dont SQLAlchemy va gérer cette particularité va tirer profit de la gestion des classes

en Python. Tout comme on l'a vu en Java, les classes game_normal et game_carriere vont simplement hériter de la classe game, et on va les construire à la fois avec leur constructeur propre, mais aussi via le super constructeur de game.

Cependant, comme ce comportement doit ensuite être "traduit en SQL", il faut un moyen de pouvoir faire des requêtes englobant cet héritage de type. C'est pourquoi on doit "mapper" un argument (ici, game_type) qui servira à l'identifier. [12]

```
class Game(db.Model):
    __tablename__ = 'games'
    id = db.Column(db.String(40), primary_key=True)
    game_type = db.Column(db.String(40))
    state = db.Column(db.Boolean, nullable=False)
    date = db.Column(db.BigInteger, nullable=False)

    __mapper_args__ = {
        'polymorphic_identity':'games',
        'polymorphic_on':game_type
    }
```

Listing 3.2 – Fichier python game.py (raccourci)

```
class Game_carriere(Game):
    __tablename__ = 'games_carriere'
    id = db.Column(db.String(40), db.ForeignKey("games.id"), primary_key=True)
    id_user = db.Column(db.String(40), db.ForeignKey("users.id"),nullable=False)
    solution = db.Column(db.String(40), nullable=False)
    maxtry = db.Column(db.Integer, nullable=False)
    length = db.Column(db.Integer, nullable=False)
    difficulty = db.Column(db.Float, nullable = False)
    won = db.Column(db.Boolean, nullable = True)
    elo_player =  db.Column(db.Float, nullable = True)
    ranked = db.Column(db.Boolean, nullable = False)

    __mapper_args__ = {
        'polymorphic_identity':'game_carriere',
    }
```

Listing 3.3 – Fichier python game_carriere.py (raccourci)

Enfin, la gestion des requêtes à ces tables est elle aussi modifiée. En effet, deux problèmes majeurs se posent alors en SQL :

— Les attributs de la table game ne sont pas enregistrés en clair dans les tables game_carriere et game_normale ;
— Les attributs des tables game_normale et game_carriere sont différents, il est donc difficile de récupérer les deux types d'objet en une seule requête ;

La manière dont SQLALchemy va régler ce problème est très simple : en définissant au préalable quelles sont les tables avec lesquelles on travaille (commande with_polymorphic() [11]), un nouveau type de table va temporairement être créé en réunissant tous les attributs possibles. On regroupe ensuite tous les objets avec un JOIN, où les attributs non possédés par un type précis seront mis à NULL (par exemple, game_normale ne possède pas d'attribut "ranked", son champ vaudra alors NULL dans cette requête). Bien entendu, cette procédure est gérée par SQLAlchemy, et voici comment faire une requête dans ce cas précis :

```
all_games_poly = with_polymorphic(game.Game, [game_normal.Game_normal, game_carriere.Game_carriere])
reqGames = db.session.query(all_games_poly).filter_by(state = True).order_by(Game.date.desc()).limit
    (20).all()
reqGames -> Contient les 20 parties finies les plus recentes, de tous types.
```

## 3.4 Fonctionnalité du site

### 3.4.1 Wordle

#### 3.4.1.1 Choix des paramètres

Pour commencer, notre application permet de jouer une partie classique de Wordle. Avant de commencer une partie on a le choix de la taille du mot et du nombre d'essais (ce choix se fait à l'aide de sliders VueJs). Un fois la demande faite une requête est effectuée dans le backend pour créer une partie à l'utilisateur. Si l'utilisateur possède déjà une partie il sera directement redirigé vers la partie en cours qui a été sauvegardée dans la base de données.


FIGURE 3.2 – Sliders

#### 3.4.1.2 Le jeu

Après cette étape nous arrivons sur la page de jeu celle-ci comporte : la grille de jeu et un clavier virtuel interactif dont les lettres changent de couleur en fonction des informations récoltées durant le jeu. Nous avons agrémenté cela d'animations esthétiques pour avoir une meilleure expérience de l'utilisateur.

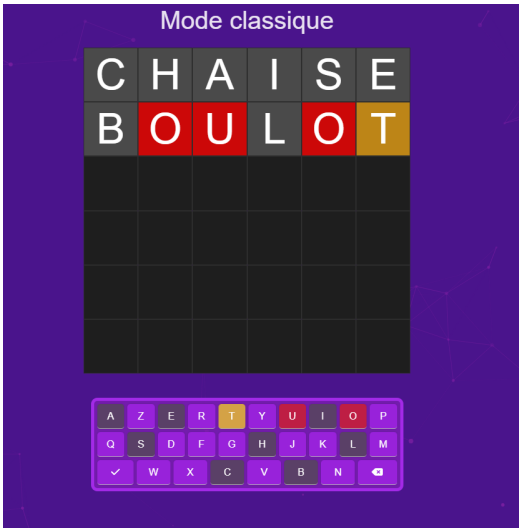



FIGURE 3.3 – Jeu classique

#### 3.4.1.3 Gestion de la soumission de réponse

Afin de rendre le site "compétitif" nous avons décidé que le traitement des réponses se ferait dans le backend. Ainsi l'utilisateur n'a accès à aucune information sur le mot qu'il est en train de chercher (en cherchant dans la console ou dans le code source frontend). Ci-dessous un schéma explicatif du fonctionnement de la soumission de réponses.

#### 3.4.1.4 Algorithme de vérification du mot

Afin de vérifier et récupérer les informations pour l'affichage des couleurs nous avons réalisé un algorithme de vérification de mot.

Tout d'abord nous allons vérifier les lettres bien placées et les marquer comme traitées, pour toutes les autres lettres, nous allons les placer dans une liste (un sac de lettres) puis on va itérer sur la solution privée des lettres bien placées, si on trouve une lettre du sac de lettres dans la solution, marquer cette lettre comme présente dans le mot, puis supprimer la lettre du sac de lettres.

On obtient ainsi toutes les informations sans erreurs.

### 3.4.2 Wordle - Partie Classée

En plus d'un mode de jeu classique, nous avons choisi d'implémenter une légère variante, un mode "classé" dans lequel les joueurs peuvent tenter de s'améliorer pour obtenir le meilleur élo (classement) possible. Dès lors qu'ils sont connectés, les utilisateurs ont le choix entre le mode de jeu classique, et le mode de jeu carrière (classé).

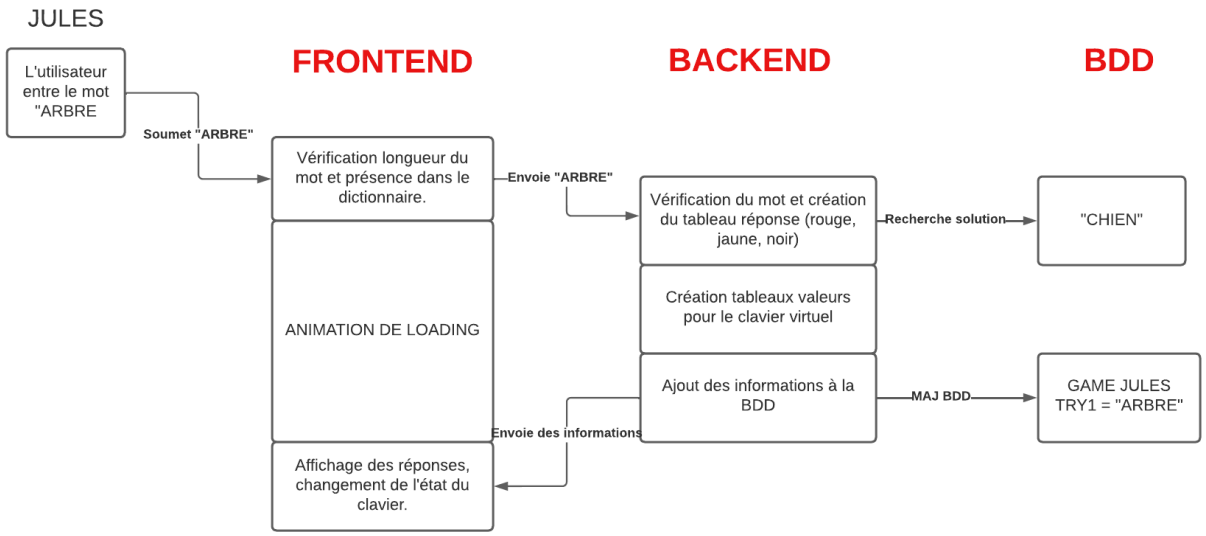



FIGURE 3.4 – Gestion réponse utilisateur

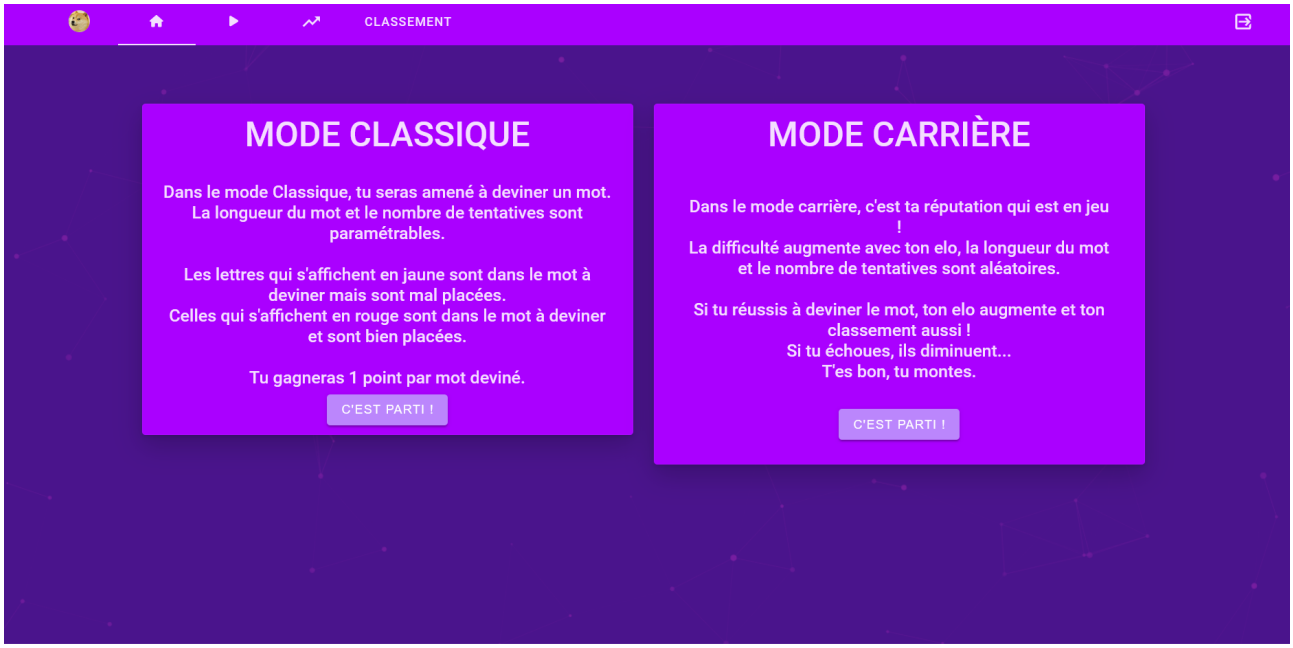



FIGURE 3.5 – Choix du mode de jeu depuis l'accueil du site

#### 3.4.2.1 Présentation du mode de jeu

Dans ce mode de jeu, le joueur est évalué. Il a un classement, que l'on nomme élo, qui diminue ou augmente selon sa performance en jeu. Ainsi, il n'a plus aucun contrôle sur les mots qui lui sont soumis : ils sont de tailles aléatoires (entre 5 et 10 lettres), et le nombre maximum d'essais est défini comme suit :

— Si la longueur de la solution est supérieure ou égale à 6, alors le nombre maximum d'essais est égal à cette longueur ;

— Sinon, le mot est de taille 5 et le nombre max d'essais vaut 6 ;

Cependant, les mots choisis ne sont pas totalement aléatoires : en effet, chaque mot se voit attribuer une difficulté, et on propose au joueur des mots dont la difficulté est proche de son élo. S'il gagne la partie, son élo monte, tandis que s'il perd, il descend. En jeu, il a possibilité de voir son élo actuel, la difficulté du mot qu'il cherche, ainsi que le gain (respectivement la perte) résultant(e) en cas de victoire (respectivement de défaite).

Le fonctionnement en lui-même du jeu est exactement le même que celui du mode classique, c'est pourquoi on ne s'attardera pas dessus ici. Notons cependant quelques points importants :


FIGURE 3.6 – Exemple de partie en mode carrière

— Si le joueur a déjà une partie en cours, elle sera aussi sauvegardée et automatiquement lancée s'il essaye de lancer une partie en mode carrière, et le joueur peut avoir à la fois une partie classique et classée en cours simultanément ;
— Lors du lancement de la partie, il peut choisir de jouer une partie en mode carrière "non classé" : il tombera sur des mots proches de son niveau de difficulté sans toutefois perdre ou gagner d'élo dans sa partie ;
— Le dictionnaire des mots proposés en solution est différent. Il se base sur la base de données Lexique [7] ; là où celui du mode classique est composée d'environ 1200 mots connus de tous. Ce nouveau dictionnaire a deux avantages : le fait d'être bien plus complet (140 000 mots environ), et beaucoup plus d'informations utiles sur les mots (fréquence d'apparition par exemple) ;

#### 3.4.2.2 Calcul de la difficulté d'un mot

Ainsi, comme dit précédemment, chaque mot se voit attribué une difficulté. Voyons comment nous avons décidé de le faire. Avant de s'intéresser aux algorithmes en eux-mêmes, précisons ce qui, selon notre expérience personnelle (à force de jouer au Wordle), rend un mot difficile à trouver :

— Si on connaît le mot ou non. C'est évident, mais un mot qu'on ne connaît pas, on ne peut pas le trouver instinctivement. Le seul moyen de gagner la partie est de trouver toutes les lettres qui le composent et leur position ;
— Si le mot possède beaucoup de lettres doublons ou non. En effet, lorsque l'on a trouvé une lettre présente dans la solution, on propose rarement par la suite un mot qui la contient encore une fois (voire qui la contient plusieurs fois), puisque l'on se concentre sur les lettres dont on ne sait pas encore si elles sont présentes ou pas ;
— Enfin, plus un mot a de lettres peu communes, moins on va le trouver vite (par exemple : OXYGENE). En effet, on cherche souvent à trouver les lettres les plus répandues (E,A,S,I,R,...) plutôt que des lettres comme Z,Y,W,X,... . Ainsi ces mots sont un peu plus durs à trouver.

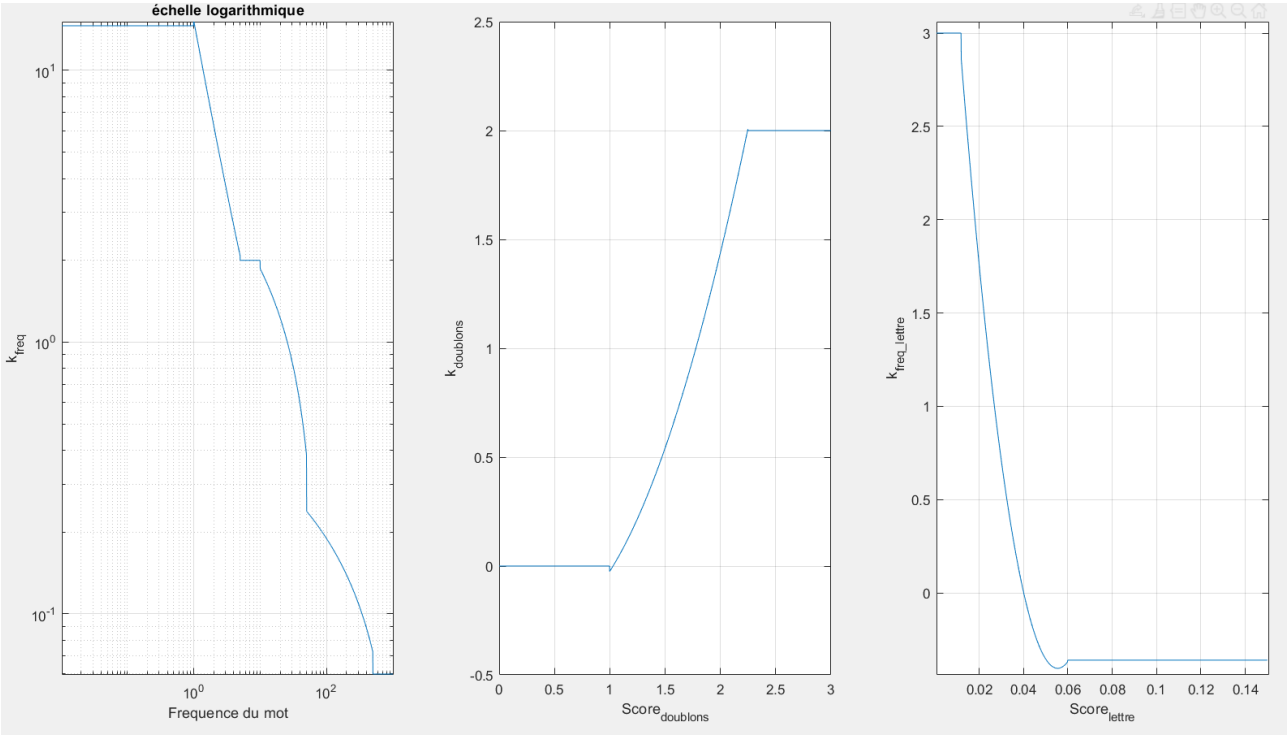
C'est en se basant sur ces trois points que le calcul de la difficulté d'un mot se fait. Ainsi :

$$difficulte = k_{freq} * (1 + k_{doublons} + k_{freq\_lettres})$$

Où chacun des coefficients est déterminé selon les 3 fonctions suivantes : (voir figures ci-dessous)

Notons que $Score_{lettre}$ correspond à la moyenne de la fréquence d'apparition dans la langue française des lettres du mot, et que $Score_{doublons}$ vaut 1 si le mot n'a pas de doublons, et vaut $\frac{\sum_{i=1}^{taille\_mot} d(l_i)}{taille\_mot}$ sinon (où $d(l_i)$ correspond à la multiplicité de la lettre $l_i$).

De ces expressions, il faut retenir que c'est la fréquence d'utilisation du mot qui joue le plus dans la difficulté obtenue, et que de toute manière les différentes fonctions en jeu ont été modifiées petit à petit à la main jusqu'à obtenir une courbe de difficulté à peu près cohérente.

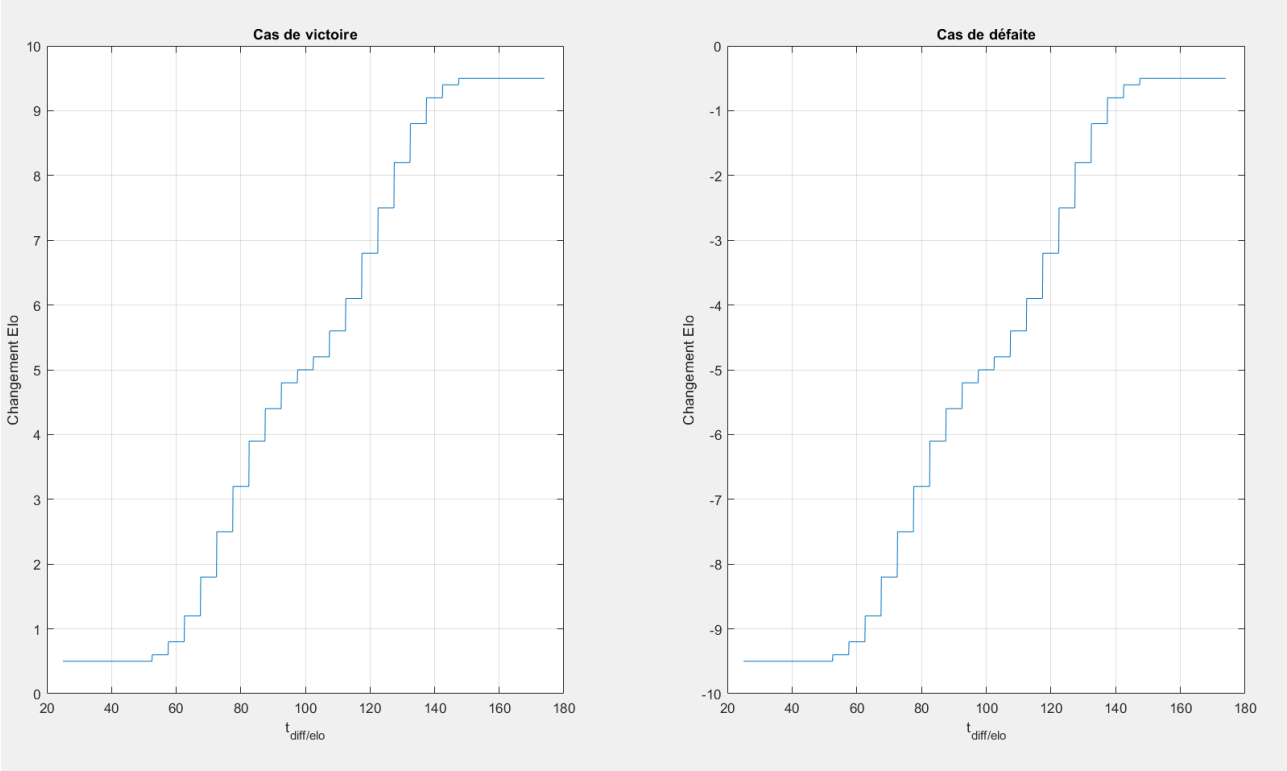



#### 3.4.2.3 Explication du système d'élo

Maintenant que la manière dont la difficulté est attribuée aux mots a été définie, reste à voir comment on attribue un niveau au joueur. On se donne pour objectif qu'un joueur d'un élo spécifique se voit proposer des mots de la même difficulté. Ainsi, on définit qu'un mot est proche de l'élo du joueur si sa difficulté appartient à :

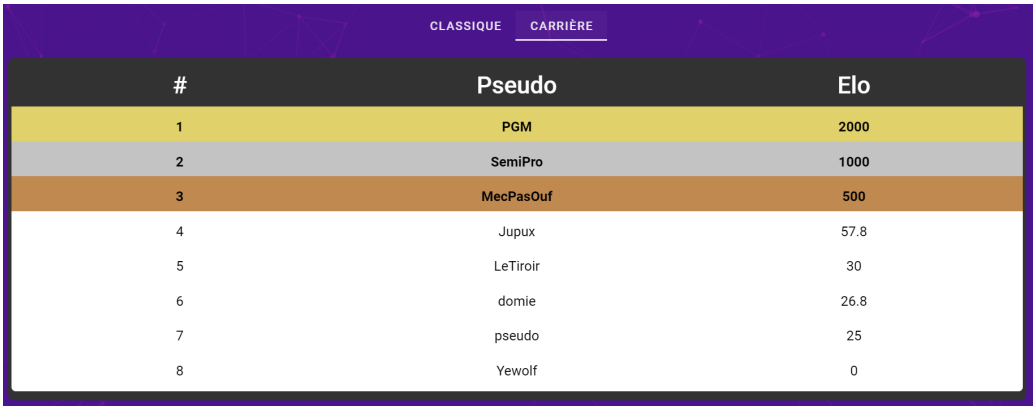
$$elo\_range = \begin{cases} (elo - 15, elo + 15) & \text{si } elo <= 120 \\ (0.75 * elo, 1.25 * elo) & \text{sinon.} \end{cases}$$

Pour ce qui est du gain (ou de la perte) d'élo, il dépend de la différence entre l'élo du joueur et la difficulté du mot, et se fait par paliers. Si on pose $t_{diff/elo}$ le taux $\frac{difficulte}{elo} * 100$, on a :




### 3.4.3 Classement

La spécificité de WordChamp, c'est son classement. Il y a en fait deux classements : le premier est un classement d'activité, les joueurs gagnent 1 point à chaque fois qu'ils gagnent une partie, peu importe le mode de jeu ; le second concerne seulement le mode classé et ordonne les joueurs en fonction de leur élo.




| # | Pseudo | Elo |
|---|---|---|
| 1 | PGM | 2000 |
| 2 | SemiPro | 1000 |
| 3 | MecPasOuf | 500 |
| 4 | Jupux | 57.8 |
| 5 | LeTiroir | 30 |
| 6 | domie | 26.8 |
| 7 | pseudo | 25 |
| 8 | Yewolf | 0 |

FIGURE 3.7 – Classement

### 3.4.4 Historique

Afin de garder un oeil sur sa progression et d'être en mesure de revoir ses meilleures (ou ses pires) parties chaque joueur a accès à un historique qui regroupe ses dernières parties dans les deux modes de jeu. Chaque partie est représentée par une carte (verte si c'est une victoire, rouge si c'est une défaite), sur laquelle apparaît un bref résumé de la partie : la date à laquelle elle a été réalisée, la solution, et son issue. En cliquant sur une entrée de l'historique, elle s'agrandit pour montrer les détails de la partie : le joueur voit alors les différentes tentatives qu'il a faites lors de cette partie.

Pour ce faire, les 20 dernières parties du joueur sont récupérées dans la base de données avec toutes leurs informations. Les parties en cours que le joueur n'a pas encore terminées n'y apparaissent pas.


FIGURE 3.8 – Historique

# Chapitre 4

# Solveur

## 4.1 Introduction

Afin de creuser plus en profondeur le sujet et apprendre à réaliser une application en C, le sujet nous invite à réaliser un solveur en C. Après avoir réfléchi, nous avons décidé de fournir 4 solveurs différents afin d'avoir un axe de comparaison des plus performants. Il y a un solveur naïf réalisé par Louis-Vincent qui va permettre d'imiter un comportement humain, un solveur se basant sur la théorie de l'information et la vidéo de 3blue1brown [3], un solveur se basant sur l'analyse fréquentielle réalisé par Malo et un solveur de Tristan qui donne une implémentation différente d'un solveur naïf en utilisant des structures de données différentes. Dans cette partie nous allons détailler la théorie derrière chacun des solveurs et l'implémentation en language C que nous avons effectué. Plusieurs des solveurs ont étés réalisé avant dans une optique de test rapide en Python.

## 4.2 Solveur avorté

### 4.2.1 Arbre de force brute

Le principe de ce solveur est assez simple, il ne s'agit pas de réfléchir mais bien de regarder tous les mots encore en lice et de regarder le nombre de possibilités moyen suivant tous les résultats possibles. Deux versions sont possibles : soit faire les calculs au fur et à mesure en élaguant les branches de l'arbre, soit en pré-calculant un graphe de la manière suivante :

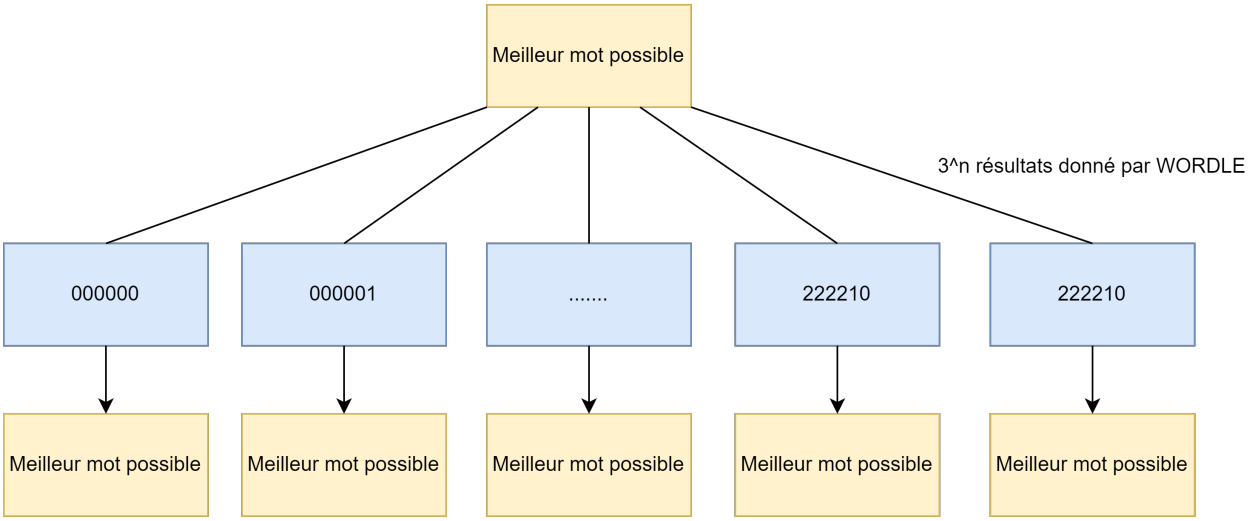



FIGURE 4.1 – Chêne de force brute

Il "suffit" de regarder pour chaque pattern le nombre de mots qui restent en lice à l'étape n+1 et de choisir le mot qui en moyenne en compte le moins. On peut alors stocker cet arbre en pré-calculant le premier mot (le plus long) et en stockant dans un très gros fichier JSON. Cette technique implémente indirectement la théorie de l'information mais est très lente, nous avons fait un essai sans aller plus loin puisque le calcul du premier mot (pour 5 lettres) prenait environ 2 heures.

## 4.3 Wordle en C

Afin de vérifier nos solveurs à la main sans avoir à retourner à chaque fois sur notre site. Nous avons développé un Wordle en C fonctionnant dans le terminal. Il possède le même algorithme de vérification de mot que notre application web. Et permet de lancer une partie en choisissant le dictionnaire et la taille de mot souhaité en argument.

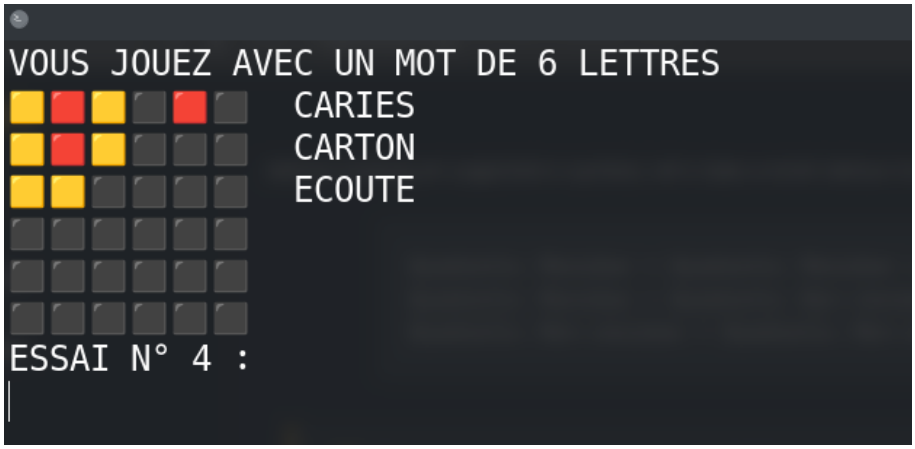



FIGURE 4.2 – Wordle en C

## 4.4 Solveur par minimisation d'entropie

Inspirés par la vidéo de 3blue1brown [3], nous avons décidé d'implémenter, à notre manière, un solveur fonctionnant par minimisation d'entropie.

### 4.4.1 Notion d'entropie

L'entropie permet de quantifier l'incertitude sur l'information qu'émet une source vis-à-vis de son récepteur. Ainsi, plus les possibilités de transmission sont grandes, plus l'entropie sera élevée. De la même manière, plus le récepteur a d'informations, en amont de la transmission, sur le message qui sera émis, plus l'entropie est faible. Prenons un exemple pour illustrer :

Supposons qu'un émetteur envoie une lettre de l'alphabet au hasard. Il y a donc 26 possibilités différentes, et chaque lettre représente 1 unité d'information pouvant être possiblement transmise.

On se dote d'une unité, le $bit$, pour pouvoir quantifier cette entropie. La manière dont il faut comprendre le bit est la suivante : si l'entropie d'un système est de $4\ bits$, alors l'incertitude sur l'information transmise est de $2^4 = 16$, c'est à dire qu'il y a 16 possibilités différentes. On peut aussi le voir autrement : si l'entropie $H$ d'un système vaut 4 $bits$, alors il faudrait réduire le champ des possibles de moitié 4 fois consécutives pour être certain du message transmis. Revenons à notre exemple : puisqu'il y a 26 possibilités différentes, l'entropie vaut $H = log_2(26) = 4.70\ bits$ Si on apprenait que la source ne pouvait émettre que des voyelles, alors l'entropie diminuerait puisque l'on a gagné des informations sur le message transmis, et alors $H = log_2(5) = 2.32\ bits$

Remarquons que ces calculs ont été effectués dans le cas où chaque lettre avait la même probabilité d'être transmise. Dans le cas où la probabilité est non-uniforme, on a la formule suivante :

$$H = \sum_{i=1}^{26} p(w_i) * log_2(\frac{1}{p(w_i)})$$

où $p(w_i) \neq 0$ correspond à la probabilité d'émission d'une lettre.

On s'aperçoit qu'avec une probabilité uniforme, on retombe bien sur la formule $H = log_2(n)$. La manière de comprendre cette formule est la suivante : on fait la somme pondérée de l'incertitude portée par chaque symbole $w_i$ pouvant être transmis par le système. La pondération se fait sur la probabilité de transmission du symbole, tandis que l'incertitude liée

au symbole vaut $log_2(\frac{1}{p(w_i)})\ bits$. Intuitivement, si par exemple $p(w_i) = \frac{1}{k}, k$ entier, alors l'incertitude de $w_i$ laisse un équivalent de k possibilités de transmission, soit $log_2(k) = log_2(\frac{1}{p(w_i)})\ bits$. Ainsi, plus la probabilité d'apparition d'un symbole est grande, moins son incertitude l'est (logique).

Pour visualiser ceci sur notre exemple, supposons qu'une voyelle a 2 fois plus de chances d'être transmise qu'une consonne. On a $p(voyelle) = \frac{2}{31}$ et $p(consonne) = \frac{1}{31}$. D'où :

$$H = 5 * p(voyelle) * (-log_2(p(voyelle))) + 21 * p(consonne) * (-log_2(p(consonne))) = 4.63\ bits.$$

On remarque que notre entropie a légèrement baissé. En effet, on a gagné en informations sur le message transmis grâce aux probabilités fournies, ce qui explique ce résultat.

### 4.4.2 Principe et fonctionnement du solveur

Commençons par cadrer les circonstances d'utilisation du solveur. Étant donné une liste de mots pouvant être solution, et une liste de mots étant proposables au jeu, le solveur cherche à trouver le mot proposable qui réduira au maximum l'incertitude liée à la solution. Pour se raccrocher à ce qu'on a vu précédemment, on définit un mot possiblement solution comme 1 unité d'information.

Dans notre cadre, comme on a, a priori, aucune information sur la probabilité d'un mot d'être solution par rapport à un autre, on est dans un cas de probabilité uniforme (nous reviendrons sur ce point un peu plus tard). Ainsi, si on pose $n$ le nombre de mots possiblement solution, on a l'entropie de la solution $H = log_2(n)$.

On va alors calculer, pour chaque mot proposable $w_i$, la quantité d'information $H_i$ qu'il permettra de connaître sur la solution. Reste à savoir comment calculer $H_i$. Pour cela, revenons au fonctionnement du wordle : lorsqu'un mot est proposé, le jeu nous renvoie un "pattern" $pat$ sous la forme "02011" par exemple. L'idée est, pour chaque mot proposable, de calculer la probabilité d'apparition de chaque pattern une fois ce mot proposé, et la quantité d'information apporté par ce pattern.

Cela tombe dans le cas du calcul d'entropie avec probabilité non uniforme, et on a :

$$H_i = \sum_j p(pat_j) * log_2(\frac{1}{p(pat_j)})$$

En sachant que $p(pat) = \frac{k}{n}$ où k correspond au nombre de mots possiblement solution pour lesquelles Wordle renverrait ce pattern pour le mot proposé. Par exemple, si je propose "MAISON", et que parmi mes 10 mots pouvant être solution, 4 me renvoient le pattern "012001", alors ce pattern a une probabilité $p(pat) = \frac{4}{10}$ pour le mot "MAISON".

Notons que l'entropie $H_i$ n'est pas exactement celle que le mot va nous donner : elle représente l'entropie $moyenne$ gagnée. On choisit donc le mot proposable qui maximise l'entropie moyenne gagnée sur la solution. Il ne reste qu'à répéter l'opération jusqu'à la fin du jeu.

Cependant, on le verra après mais ce solveur est assez lent, ce qui nous a poussé à pré-calculer la meilleur proposition possible en tant que première proposition pour gagner du temps. Ainsi, voici la structure de l'algorithme :

### 4.4.3 Implémentation

Avant de commencer, il est important de préciser que 3 implémentations différentes, pour des raisons de performances, ont été réalisées. Nous allons présenter les 3 et expliquer leurs particularités. Avant ça, présentons rapidement l'algorithme qui englobe ces implémentations.

#### 4.4.3.1 Algorithme principal

Dans le fichier solv_3b1b_main.c, il gère l'interaction utilisateur / solveur et s'occupe de précharger les mots ainsi que la meilleure solution pour ouvrir le jeu. Il ne s'occupe pas du calcul des $H_i$ ni du choix du meilleur mot. Simplement, selon la taille des mots en entrée, il choisira l'implémentation de ces calculs la plus performante.

#### 4.4.3.2 Première version du solveur

De loin la plus complexe, c'est l'implémentation la plus littéral de ce qui a été fait précédemment. On calcule le meilleur mot proposable de la façon décrite plus haut. Tout d'abord, elle charge les mots proposables ainsi que les mots possiblement solution dans des tables de hachage dont l'implémentation est similaire à ce qui a été vu en TP de SD. La justification de ce choix de structure se fera dans la partie suivante. On introduit aussi une structure $pattern\_t$ pour faciliter les calculs de pattern, définie comme suit :

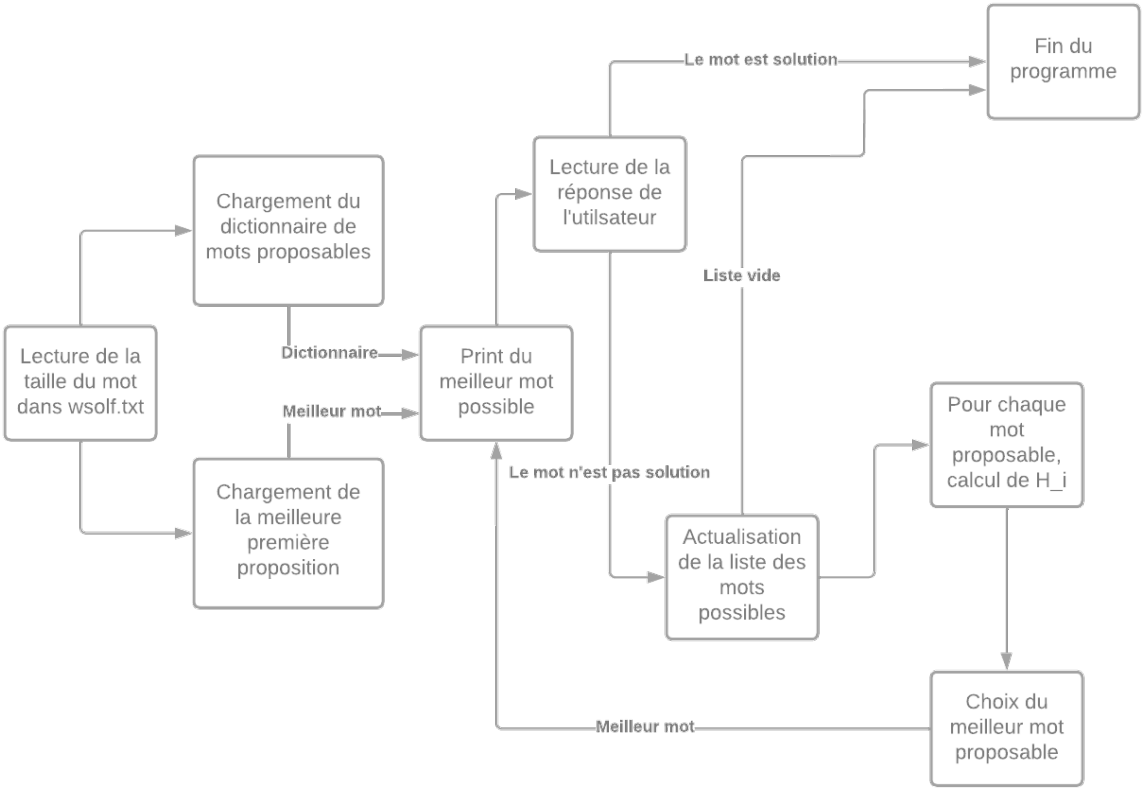



FIGURE 4.3 – Fonctionnement du solveur par minimisation d'entropie

```c
typedef struct pattern_t {

    int * val;
    char * guess;

} pattern_t;

pattern_t * create_pattern(int * val, char * guess);
void destroy_pattern(pattern_t * result);
bool possible(char * word, pattern_t * result, int * count_arr, bool * bool_arr);
void possible_pre(int ** count_arr, bool ** bool_arr, pattern_t * result);
int count_occ(char * string, char c);
void print_pattern(pattern_t * pattern);
bool pattern_is_possible(pattern_t * pattern);
pattern_t *** generate_all_pattern(char * word,int ** val_arr);
void destroy_pattern_arr(pattern_t *** pattern_arr );
void gen_val_array(int *** val_arr, int size);
void destroy_val_array(int *** val_arr, int size);
void update_possible_words(table_t * table, pattern_t * pattern);
double calculate_proba(pattern_t * pattern, table_t  * table, char *** all_words_arr,int arr_size);
void best_guess(table_t * table_dico, table_t * table_proposable,int taille);
```

L'idée de ce solveur est la suivante :

mots_proposables;
mots_possibles;
patterns;
meilleur_mot;
$\max_H \leftarrow 0$ ;
**for** *mot* $\in$ *mots_proposables* **do**
  $H_i \leftarrow 0$;
  **for** *pat* $\in$ *patterns* **do**
    p$\leftarrow$ $calculate\_proba(pat, mot)$ ;
    $H_i \leftarrow H_i + p * log_2(\frac{1}{p})$ ;
  **end**
  **if** $H_i >$ *max*$_H$ **then**
    $\max_H \leftarrow H_i$ ;
    meilleur_mot $\leftarrow$ mot ;
  **end**
**end**
return meilleur_mot ;

Bien que fonctionnelle en théorie, le problème majeur de cet implémentation est sa lenteur. En sachant que la complexité de $calculate\_proba(pat, mot)$ est en $\Theta(N)$, cet algorithme a une complexité globale en $\Theta(N^2 * 3^{taille})$ où $taille$ est la taille des mots (on a $3^{taille}$ patterns !). De plus, les tables de hachage ne sont pas pratiques lorsqu'il s'agit de parcourir tout leur contenu, ce qui ajoute à la lenteur. Enfait, les implémentations suivantes vont se passer de table de hachage et utiliser des astuces algorithmiques pour permettre d'améliorer les performances.

#### 4.4.3.3 Seconde version du solveur

La première modification majeure est la suppression des tables de hachage. On charge simplement les mots dans des tableaux qui sont bien plus rapides en terme de temps d'exécution. La seconde est la manière de calculer $H_i$. Plutôt que de calculer, pour chaque pattern possible, le nombre de mots possiblement solution qui le généreraient, on va créer un tableau d'entiers initialisés à 0 de taille $3^{taille}$, qui va en fait représenter tous nos patterns. Ensuite, pour chaque mot possible, on regarde le pattern généré et on fait +1 dans la case du tableau correspondant à ce pattern. Voici le nouvel algorithme :

$initialize(tab)$ ;
**for** *mot_prop* $\in$ *mots_proposables* **do**
  **for** *mot_poss* $\in$ *mots_possibles* **do**
    pat $\leftarrow$ $generate\_pattern(mot\_prop, mot\_poss)$ ;
    tab($pat\_to\_int(pat)$) $\leftarrow$ +1;
  **end**
  $H_i \leftarrow sum(tab)$;
  **if** $H_i >$ *max*$_H$ **then**
    $\max_H \leftarrow H_i$ ;
    meilleur_mot $\leftarrow$ mot ;
  **end**
**end**
return meilleur_mot ;

Précision : $sum(tab)$ renvoie la somme : $\sum_i p_i * log_2(\frac{1}{p_i})$ où $p_i = \frac{tab(i)}{len(mots\_possibles)}$

De plus, la manière dont fonctionne pat_to_int est la suivante : si pat = "21001", la fonction renvoie $2 * 3^4 + 1 * 3^3 + 0 * 3^2 + 0 * 3^1 + 1 * 3^0$. Cela permet une bijection entre le domaine des entiers et le domaine des patterns.

Rien ne change pour l'entropie calculée, mais la complexité passe en $\Theta(N * (N + 3^{taille}))$, ce qui équivaut à du $\Theta(N^2)$ pour $3^{taille} << N$.

Cependant, un autre problème apparaît pour des mots de taille >= 13 : à partir de ces mots, $3^{taille} > 10^6 >> N$, et le programme commence à sérieusement ralentir. De plus l'allocation mémoire devient bien trop grande (au delà de 20, le programme plante). C'est la raison pour laquelle on a crée une troisième et dernière version du solveur. Notons que ce même problème est rencontré pour la première implémentation, où la compléxité explose pour des mots de ces tailles.

#### 4.4.3.4 Troisième version du solveur

Maintenant, on souhaite régler le problème précédent sans pour autant perdre ses avantages. L'introduction du tableau de "patterns entiers" est ce qui permettait de gagner énormément en temps d'exécution, mais aussi ce qui pose soucis pour des mots trop long. On va donc le redéfinir. Après avoir initialisé un tableau de longint vide (tableau $tab$), mais aussi

un tableau d'entier vide (tableau $count$), on utilise pour chaque pattern la même bijection que précédemment. Si le long donné n'est pas dans $tab$, on l'ajoute dedans (via un realloc), sinon on incrémente la valeur du tableau $count$ qui a pour indice la position de ce pattern dans $tab$.

Tout le reste de l'algorithme reste le même. Ainsi, on ne sauvegarde que le nombre de patterns qui sont réellement générés (par exemple, "000000000000" pourrait ne pas être généré pour un mot donné). Le problème de cette méthode, c'est qu'on a besoin de parcourir $tab$ en entier pour savoir s'il contient ou non un pattern fraîchement généré. Cela implique que notre complexité théorique, dans le pire cas, est en $\Theta(N^3)$ (contre $\Theta(N^2)$ dans le meilleur). Le pire cas étant celui où chaque pattern n'est généré que par un unique mot possiblement solution. Cependant, il est important de noter deux points :

— Ce cas est très rare lorsque N est grand : on aura beaucoup de mot possiblement solutions qui généreront le même pattern ;
— On utilise cet algorithme pour des mots de taille assez grande (>=13), et il n'y a pas énormément de mots qui ont ces tailles ci.

En clair, même s'il est en moyenne plus lent que la seconde implémentation sur les mots courts, il sera plus rapide sur les mots plus longs. C'est ce qui explique pourquoi notre algorithme principal jongle entre ces deux dernières implémentations suivant la taille des mots.

### 4.4.4 Limites de ces implémentations

Alors finalement, tout n'est pas si mal ? Oui et non. Malgré les optimisations mises en place, le programme reste assez lent pour deviner le meilleur mot lors de la première proposition (environ 2h dans le pire cas (mots de 13-15 lettres)). C'est la raison pour laquelle on a stocké tous les calculs des meilleures ouvertures pour chaque longueur de mot. L'inconvénient, c'est que ce calcul est effectué sur notre dictionnaire, et qu'il devra être recommencé si on venait à le changer. (Toutefois, il y a en annexe 2 un résumé des résultats obtenus pour ces premières propositions, sur notre dictionnaire d'environ 400 000 mots).

Ensuite, nous nous sommes placés dans un cas bien particulier, celui où la probabilité d'un mot d'être solution par rapport à un autre est uniforme : c'est le cas simple lors du calcul d'entropie.

Mais supposons que l'on veuille tester notre solveur sur un Wordle dont nous ne connaissons pas la liste exacte des mots possibles. Déjà, il y aurait le problème de lenteur lié au calcul de la meilleur ouverture. Mais surtout, on ne pourrait pas, avec les deux dernières implémentations, exploiter de possibles informations sur la liste des mots possibles !

C'est ce que fait 3blue1brown dans sa vidéo : il part du principe que les mots possibles sont ceux qui sont connus du plus grand monde, et associe à chaque mot une probabilité d'être solution ou non, en fonction de sa fréquence d'utilisation. Or, avec des implementations utilisant des arrays pour stocker les mots, on ne peut pas vraiment faire de même. C'est pourquoi on a voulu créer une première version utilisant des tables de hachage : avec un coût d'accès à une donnée (ici la probabilité d'être solution) et d'ajout en temps constant, c'était la structure de données idéale. Ainsi, si les problèmes rencontrés précédemment (de performances) n'avaient pas été si importants, on aurait pu tenter de créer une quatrième implémentation qui tiendrait compte de ce nouveau paramètre, à l'aide des tables de hachage en combinaison avec des array.

Enfin, même si nous n'avons pas pu aller au bout de ce que l'on souhaitait faire avec cette proposition de solveur, il faut quand même noter que ses performances de jeu sont très bonnes, et que cela justifie la perte de performance temporelle.

## 4.5 Solveur naïf

### 4.5.1 Motivations

Lorsqu'il est question d'optimisation, de comparaisons de performances, il est toujours judicieux d'avoir des points de référence, des étalon qui permettent de vérifier la pertinence des solutions développées.

Ceci est d'autant plus vrai dans le cas d'un problème ouvert comme le nôtre, c'est donc tout naturellement, qu'au moment de décider des implémentations de solveur que nous allions réaliser, nous avons choisi d'en créer un très simple, qui simulerait les habitudes de jeu d'un débutant et qui permettrait de contrôler l'efficacité des solveurs plus compliqués.

### 4.5.2 Structures de données

Ce solveur utilise, pour mettre en place les filtres traités dans la partie suivante, la structure de tuple suivante :

```
typedef struct strInt_t {
    char* lettre;
    int position;
} strInt_t;
```

Elle permet de traiter facilement les informations données par le jeu, qui sont majoritairement sous la forme : "lettre X à la position X". Il utilise également, pour répertorier l'ensemble des mots encore en lice, une structure de liste chaînée de chaînes de caractères. Elle permet de parcourir la liste tout en déterminant pour chaque élément s'il est encore une solution possible ou non. Elle permet donc, en temps linéaire de réduire la liste, des mots possibles en fonction des nouvelles informations obtenues.

### 4.5.3 Fonctionnement

Ce solveur porte bien son nom : il détermine une liste de mots encore potentiellement solutions et il en choisit un au hasard parmi ceux-ci.

Il conserve les mots selon 4 critères :

— Si le mot contient une lettre qui est notée absente de la solution par le Worlde, il est supprimé.

— S'il ne contient pas toutes les lettres "jaunes" il est supprimé.

— S'il présente une lettre "jaune" à la position où le Worlde la note "jaune", il est supprimé.

— S'il ne présentes pas aux bons endroits toutes les lettres que le Worlde a notées comme "présentes et bien placées", il est supprimé.

Ce solveur joue donc comme le ferait un joueur inexpérimenté mais qui connaît parfaitement la langue française et qui n'aurait aucun scrupule à répondre "AMNIOS" plutôt que "MAISON"...

### 4.5.4 Abandon

Ce solveur faisant doublon avec l'autre solveur naïf présenté plus loin, il a été développé surtout dans le but d'utiliser des structures de données différentes et d'explorer d'autres implémentations. Nous avons ensuite réalisé les tests de performance sur le plus performant des deux (à savoir l'autre) afin de servir de point de référence pour les solveurs plus évolués.

## 4.6 Solveur par analyse fréquentielle et rapprochement d'un mot idéal

### 4.6.1 Principe et algorithme

Une autre approche du problème permet de le résoudre plus rapidement, l'objectif de ce solveur était de rester en complexité linéaire avec des calculs très rapides contrairement au premier solveur. On utilise toujours la théorie de l'information mais le meilleur mot trouvé n'est pas toujours le mot parfait comme on ne fait pas de recherche exhaustive (contrairement au premier algorithme) il s'agit d'une approximation du mot idéal. En résumé ce solveur est rapide mais fait des compromis sur la précision du meilleur mot possible. Voici un schéma de son fonctionnement dont nous allons détailler les différentes parties.

### 4.6.2 Structure

Pour ce solveur, aucune étape de l'algorithme ne nécessitait l'implémentation d'une structure complexe car elles n'apportaient ni simplification ni optimisation en runtime ou stockage. Nous avons simplement utilisé les array en C.

### 4.6.3 Analyse Fréquentielle

À chaque itération nous allons soumettre la liste des mots possibles à une analyse fréquentielle. Elle consiste en la création d'un tableau de $26 \times taillemot$. Par la suite on itère sur le nombre de mots de la liste et pour chaque lettre de chaque mot, on va ajouter 1 dans la ligne lettre et la colonne position de la lettre lue. Au final on se retrouve avec la répartition de la fréquence des lettres par position pour une liste de mots de taille donnée

### 4.6.4 Calcul du Score

Afin de pouvoir comparer les mots de notre liste et trouver celui qui permettra d'en éliminer le plus d'autres il faut créer un algorithme qui va calculer le score pour un mot donné. Pour faire cela nous allons créer un mot idéal (qui n'existe normalement pas dans le dictionnaire) composé des lettres les plus présentes à chaque position. Par exemple pour un mot de 6 lettres ce mot idéal est : CARAEE.

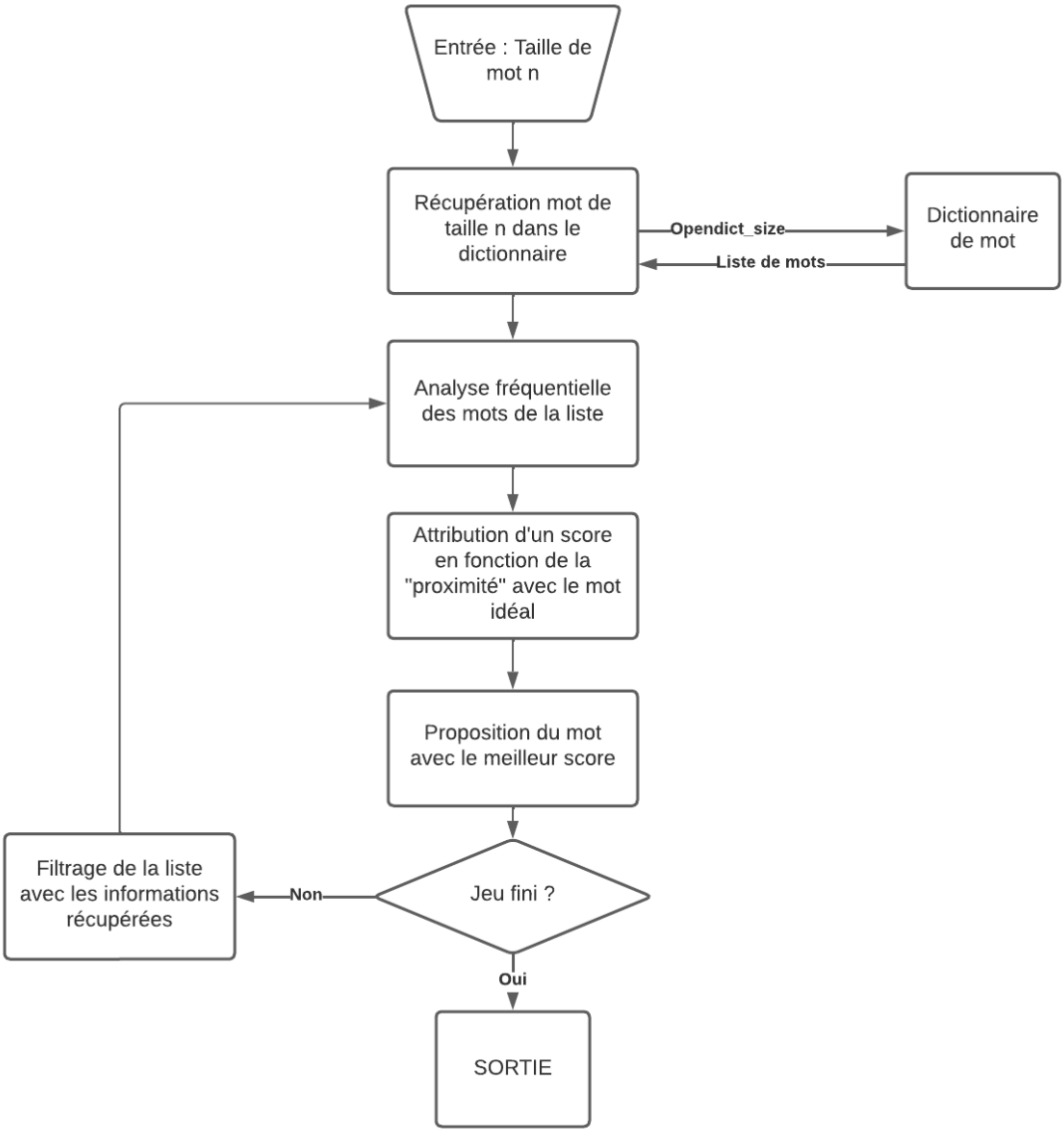



FIGURE 4.4 – Algorithme solveur

On note $\forall i \in [0; taillemot - 1], F_{mot}[i]$ comme étant la fréquence de la lettre en position i dans le mot et $F_{max}[i]$ la valeur max de la fréquence pour la position $i$. Il en découle donc la formule du score :

$$d = \prod_{i=0}^{taillemot-1} 1 + (F_{mot}[i] - F_{max}[i])^2$$

Plus le score est bas plus le mot est proche du mot idéal. En fait cette formule est similaire à la formule de la distance au carré entre le mot idéal et un mot de la classe des mots de la liste (en se basant sur la fréquence des lettres). Par exemple pour reprendre l'exemple de taille 6 le mot correspondant est CARIES

Le mot qui sera présenté sera donc le mot qui est le plus proche du mot idéal et ce mot va nous donner beaucoup d'informations sur le mot recherché car il est commun avec beaucoup d'autres mot.

### 4.6.5 Filtrage en fonctions des informations récupérées

Pour ce solveur un système de 5 filtres est mis en place. Pour simplifier la gestion des filtres à chaque proposition et réponse du wordle on créer trois listes : la liste des lettres bien placées, mal placées, et ne faisant pas partie du mot (non existante).

— Filtre 1 : Pour toutes les lettres appartenant à la liste des lettres non existante si le mot possède une lettre dans cette liste et que cette même lettre n'est pas dans bien placée ou mal placée, On enlève le mot de la liste des mots

possibles.

— Filtre 2 : Pour toutes les lettres bien placées on vérifie que le mot possède les lettres placées à la bonne place.
— Filtre 3 : Pour toutes les lettres mal placées on vérifie que le mot ne possède pas la lettre à la même position.
— Filtre 4 : Pour toutes les lettres bien placées et mal placées on vérifie qu'elles apparaissent dans le mot
— Filtre 5 : Pour les lettres apparaissant à la fois dans bien placées ou mal placées et dans la liste de non existant on vérifie que le nombre de fois où apparaît cette lettre est égal à son nombre d'apparition dans mal placées et bien placées.

La complexité de ce filtrage est en $O(n)$

### 4.6.6 Vérification précision des mots comparaison avec premier solveur

Dancs ce tableau on peut comparer l'efficacité des mots choisis par la solveur fréquentiel (Best s.2) par rapport au solveur exhaustif (Best s.1). Les scores des mots sont bons mais pas optimaux.

| Taille | Best s.1 | Mots restants | Best s.2 | Mots restants | Nb mots total |
|---|---|---|---|---|---|
| 5 | RAIES | 97.896 | PARES | 126.691 | 7 980 |
| 6 | TARIES | 77.654 | CARIES | 88.523 | 17 991 |
| 7 | TARINES | 59.019 | CARIEES | 69.124 | 32 230 |
| 8 | RATINEES | 43.259 | CARENEES | 69.403 | 48 039 |
| 9 | CERTAINES | 25.135 | VERIFIEES | 88.625 | 59 584 |
| 10 | CETONURIES | 14.172 | CERTIFIEES | 26.246 | 62 954 |

FIGURE 4.5 – Tableau comparatif du nombre de mots restants, en moyenne, après la première proposition

## 4.7 Solveur naïf avec beaucoup de structures

### 4.7.1 Motivation

Pour les raisons évoquées plus haut, nous avons développé un autre solveur naïf, afin de nous aider à appréhender les résultats des autres et de permettre à tous les membres du groupe d'implémenter leurs propres structures.

### 4.7.2 Structures utilisées

#### 4.7.2.1 Mot

Afin de stocker les mots plus facilement nous utilisons une structure mot avec les caractéristiques ci-dessous. Cette structure est utilisée pour la liste de mot définis dans la suite.

```
typedef struct word {
  int score;
  char * word;
} word_t;
word_t * word_create(char * word); // Initialise un objet
void word_destroy(word_t * word); // Detruit l'objet
int word_has(char * word, char letter); // Permet de voir si une lettre est dans le mot
int word_count(char * word, char letter); // Permet de compter le nombre d'occurrences d'une lettre
```

#### 4.7.2.2 Lettre

Cette structure sera utilisée afin de réaliser le filtre qui va enlever du dictionnaire les mots invalides. Elle sert a lister les lettres et, pour une lettre, à savoir combien d'occurrences elle doit avoir et où dans le mot.

```
typedef struct letter {
    int count;
    int ** places;
    int size;
} letter_t ;

letter_t * letter_create(); // Initialise
void letter_destroy(letter_t * letter); // Detruit
void letter_add_place(letter_t * letter, int place, char state); // Rajoute un endroit, etat = 0, 1,
    2
int letter_is_miss_placed(letter_t * letter, int place);
int letter_is_right(letter_t * letter, int place);
int letter_has_right(letter_t * letter);
int letter_has_miss_place(letter_t * letter);
int letter_has_wrong(letter_t * letter);
int letter_get_count(letter_t * letter);
```

#### 4.7.2.3 Liste

Nous utilisons une liste a taille dynamique pour stocker des pointeurs vers des éléments. L'utilisateur peut mettre n'importe quel type de pointeur et cela n'importe pas ici. Dans l'implémentation, la taille de la liste double lorsqu'elle est au maximum.

```
typedef struct element_t
{
    char * key;
    void * value;
} element_t;

struct list_t
{
    int size;
    int capacity;
    element_t ** data;
} list_t;

list_t *list_create();
void list_destroy(list_t *one_list);
bool list_is_empty(list_t *one_list);
int list_resize(list_t *one_list);
void list_add(list_t *list, size_t index, char *key, void * value);
void list_append(list_t *one_list, char *one_key, void * one_value);
void * list_get(list_t *one_list, size_t index);
void element_print(element_t *one_element);
void list_print(list_t *one_list);
bool list_contains(list_t *one_list, char *one_key);
void * list_find(list_t *one_list, char *one_key);
void list_iterate(list_t *one_list, void (*one_function)(char *, void *)); // Permet d'executer une
    fonction sur chacun des elements.
```

#### 4.7.2.4 Table de hachage

Nous utilisons les mêmes tables de hachage que vues en TP, en utilisant les listes définies ci-dessus.

```
typedef struct wordlist {
  word_t ** words;
  int size;
} wordlist_t ;
wordlist_t * wordlist_create();
void wordlist_destroy(wordlist_t * wl);
void wordlist_add(wordlist_t * wl, char * word);
void wordlist_open_file(wordlist_t * wl, char * filename, int size);
void wordlist_remove(wordlist_t * wl, int index);
word_t * wordlist_pick_random_word(wordlist_t * wl);
void wordlist_remove_words(wordlist_t *wl, char * guess, char * response); // Enleve les mots
    inutiles
int check_right_letters(table_t * all_letters, char * word); // Filtre "bonne lettre"
int check_miss_placed_letters(table_t * all_letters, char * word); // Filtre "mal placee"
int check_wrong_letters(table_t * all_letters, char * word); // Filtre "pas dans le mot"
```

### 4.7.3 Fonctionnement

Le solveur fonctionne très simplement. En tout premier lieu il va renvoyer un mot aléatoire. Ensuite avec la réponse, il va filtrer les mots qui ne sont plus valides et choisir un autre mot aléatoirement. Le filtre va d'abord regarder les lettres dont le compte est à 0 : elles ne sont pas présentes dans le mot. Puis pour les lettres mal placées, il regarde si elles sont toujours au même endroit (ce qui n'a pas lieu d'être) et le compte, il doit être au moins supérieur. Puis il regarde les lettres qui sont censées être bien placées dans le mot. Les mots qui ont un score de -1 sont ensuite enlevés. Cette opération est en O(n) par rapport à la taille de la liste de mots.

# Chapitre 5

# Testeur

## 5.1 Présentation

Afin de pouvoir tester, comparer, étudier différentes implémentations, nous avons décidé de créer un logiciel (TEST-CHAMP) qui ferait cela automatiquement à partir de fichier exécutable. Cela signifie que nous pouvons tester en python une implémentation avant de la réaliser en C et de tester ses performances ainsi que le gain en changeant de langage.

## 5.2 Principe de fonctionnement

Puisque le logiciel doit être capable de jouer avec un solveur, celui-ci doit intégrer un WORDLE minimal mais fonctionnel ainsi qu'un dictionnaire de mot. Pour simplifier, celui-ci disposera du dictionnaire Scrabble ODS 8 disponible sur *Github*. Ensuite, le logiciel doit être capable de pouvoir jouer avec un solveur quelconque, donc de lancer le solveur ainsi que de communiquer avec. Cela va se faire grâce à l'entrée et la sortie standard du solveur et grâce au protocole défini dans le sujet.

## 5.3 Choix du language

Pour réaliser ce testeur, nous avons choisi de le faire en *Go*, c'est un language compilé au même titre que le C. En revanche il propose plus de fonctionnalités par défaut comme une gestion de mémoire avec un garbage collector et une "parallélisation" (appelée concurrence) grâce à un token spécifique (celui-ci aura une importance dans la section "Problèmes rencontrés").

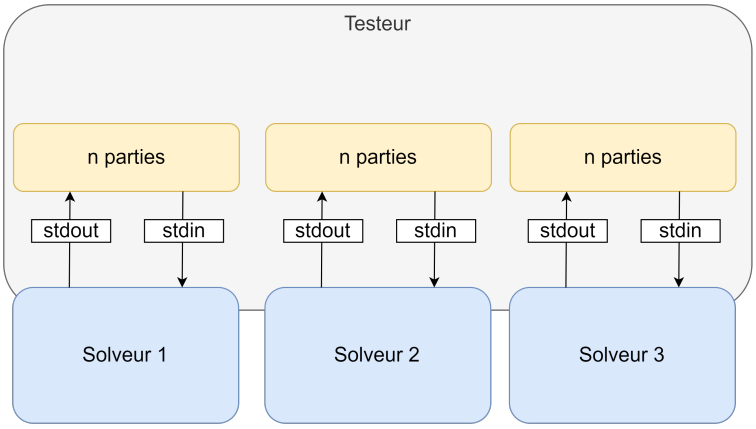



FIGURE 5.1 – Diagramme du fonctionnement du testeur

## 5.4 Problèmes rencontrée

### 5.4.1 Concurrence

Après avoir réalisé la toute première version du testeur nous n'avions pas encore de solveur, nous avons donc créé des faux solveurs qui ne font que répéter le même mot en boucle (pour tester le testeur). Ce qui fonctionnait assez bien mais était assez lent puisque les parties étaient jouées les unes à la suite des autres. Nous avons donc décidé de paralléliser la tâche et donc de lancer le solveur autant de fois que l'on peut. Ce qui ressemblerait plus au schéma suivant.

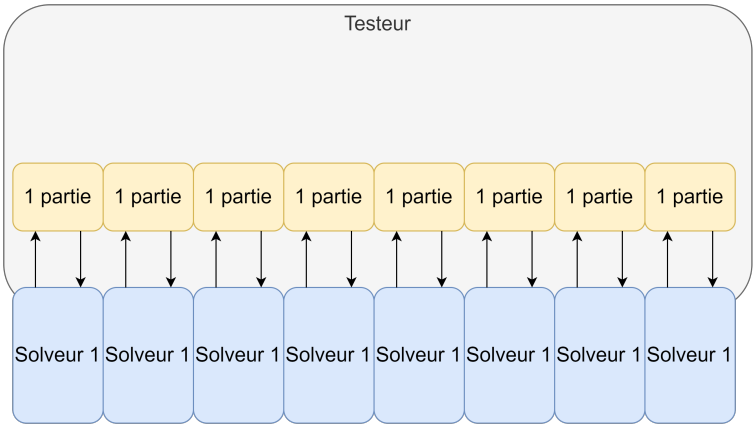



FIGURE 5.2 – Diagramme du fonctionnement du testeur (un peu plus réaliste)

### 5.4.2 Sortie en tampon

Après avoir réalisé le premier solveur nous nous sommes rendus compte que celui-ci ne fonctionnait pas avec le testeur : ce dernier ne lisait rien dans la sortie standard du solveur. La sortie standard fonctionne comme un fichier dans le sens où, du point de vue du programme, il ne fait qu'écrire dans un fichier pour afficher, ou lire un fichier pour l'entrée standard. En revanche, le fonctionnement en exécution normale n'est pas le même qu'en exécution "pipe" (puisque le testeur doit recevoir la sortie standard il crée une pipe). Dans ce mode, le tampon n'attend pas un caractère de fin de ligne mais la fin du programme afin d'afficher du texte. Pour pallier à ce problème on peut utiliser *stdbuf -oL*.

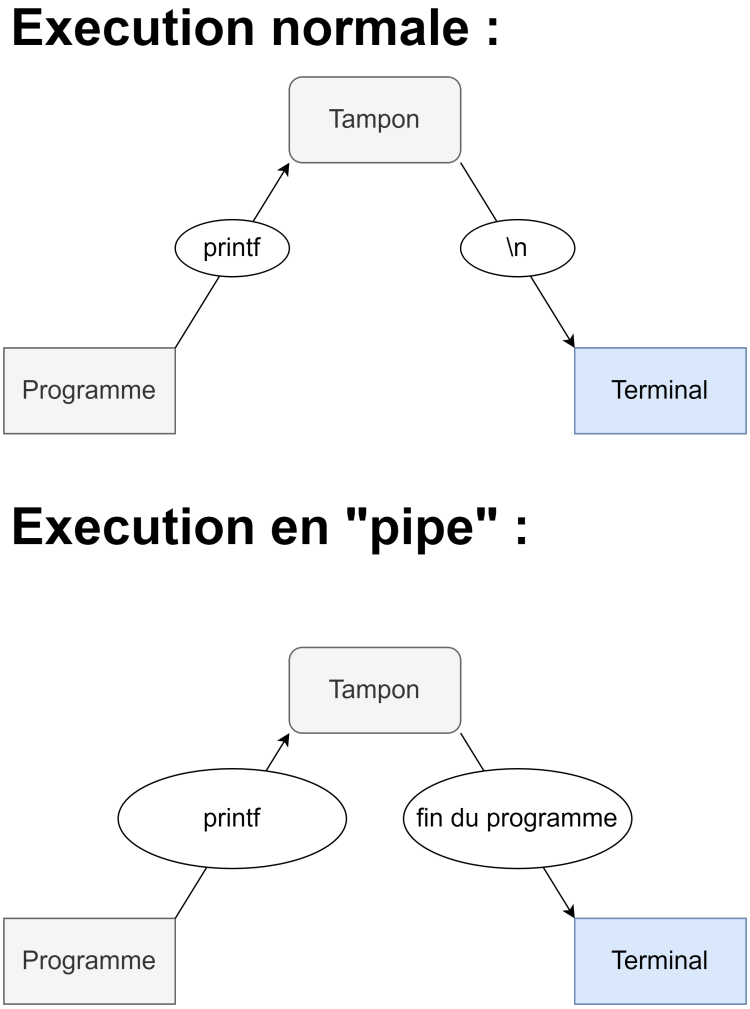



FIGURE 5.3 – Fonctionnement du tampon d'écriture

# Chapitre 6

# Test et Performance

## 6.1 Introduction

Afin de tester nos différents solveurs, nous avons utilisé l'utilitaire présenté ci-dessus.

## 6.2 Test et Performance

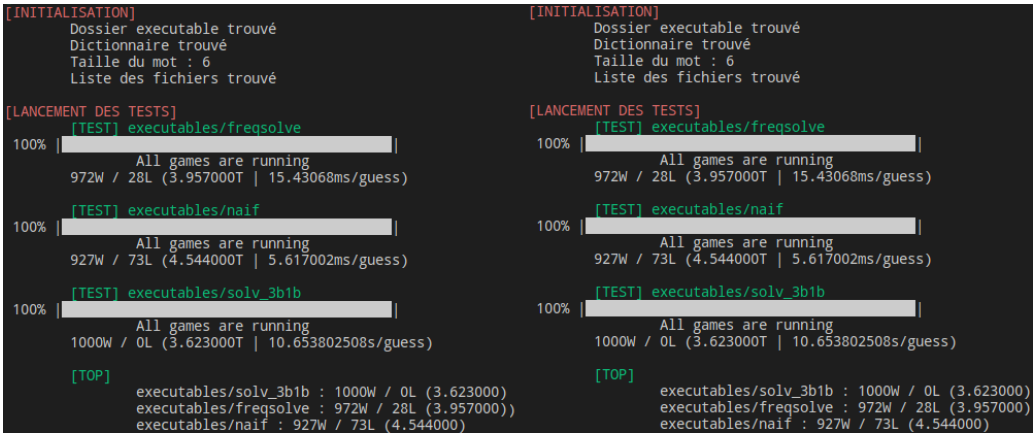



FIGURE 6.1 – Exemples d'essais réalisé

Voici les graphes résultants de nos tests :

## 6.3 Nos conclusions

Suite au développement de ces différents solveurs, nous sommes arrivés à plusieurs conclusions et avons appris beaucoup de choses sur la programmation en général.

Tout d'abord une solution naïve n'est pas forcément inefficace : les solveurs utilisant une méthode purement naïve peuvent donner une solution en un nombre d'essais acceptable et de plus ils sont très rapides dans leur traitement. Cela nous a permis de nous rendre compte à quel point la théorie de l'information apporte énormément, peu importe le mot choisi.

Deuxièmement : la perfection algorithmique a un coût non-négligeable : le premier solveur suivant la vidéo de 3Blue1Brown est une recherche exhaustive et nécessite de couvrir toutes les possibilités. Le résultat de l'algorithme est parfait (le mot recherché est le mot qui éliminera le plus de possibilités) mais le calcul prend du temps notamment pour les premiers mots. La solution parfaite et exhaustive n'est donc pas vraiment optimale dans le cadre d'un algorithme devant être performant en temps.

Enfin l'importance du rapport performance/résultats : le solveur fréquentiel semble être un bon compromis car il est performant et possède une qualité de résolution bien meilleure que le solveur naïf. Ce solveur nous a appris qu'en informatique la réponse à un problème n'est ni unique, ni idéale et que c'est l'utilisation finale du programme qui doit dicter la manière de le réaliser. L'objectif d'une personne qui développe et imagine des solutions informatiques doit alors être de trouver le meilleur compromis performance/résultats tout au long du développement.

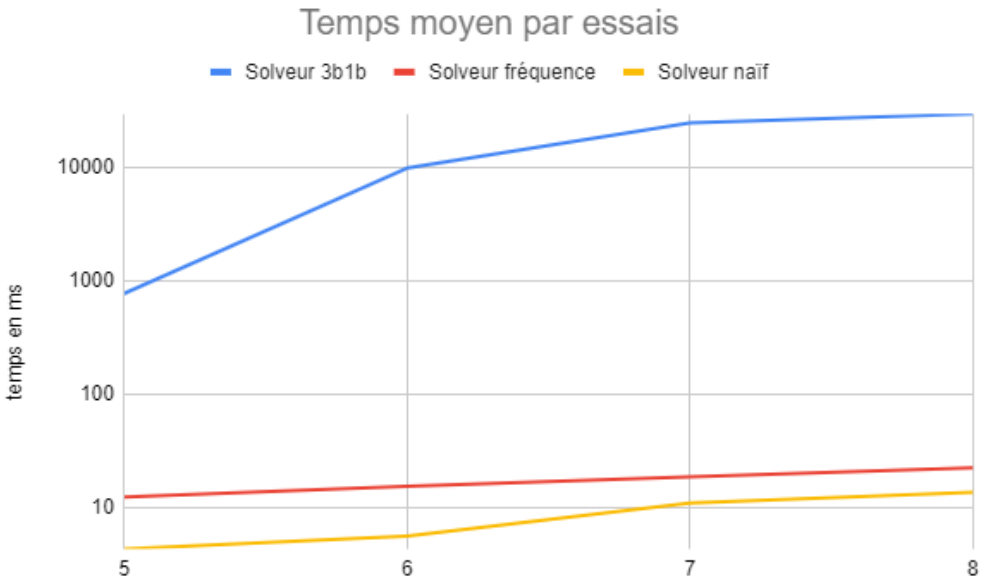



FIGURE 6.2 – Graphique montrant le temps moyen par réponse pour chaque solveur

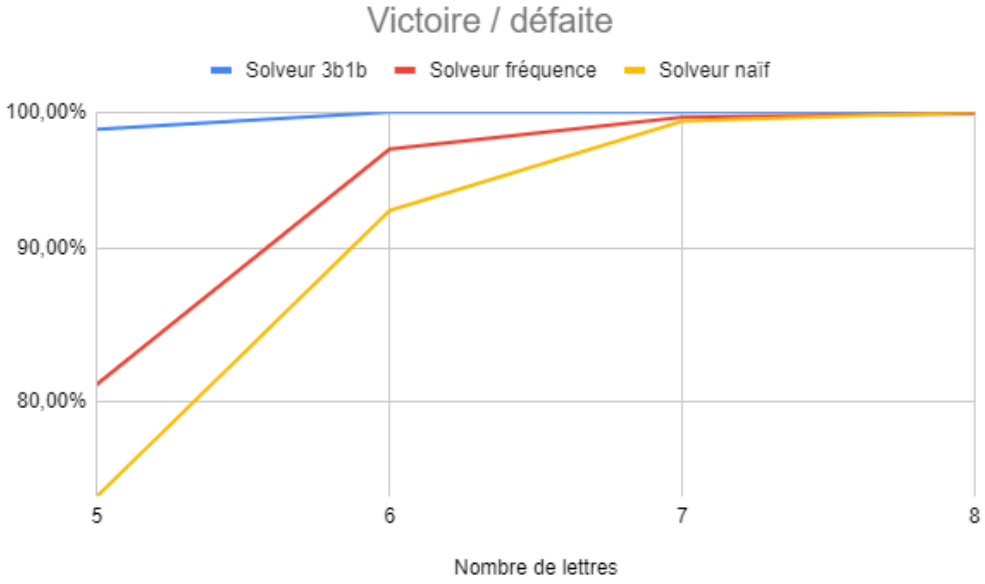



FIGURE 6.3 – Graphique montrant le pourcentage de victoire pour chaque solveur

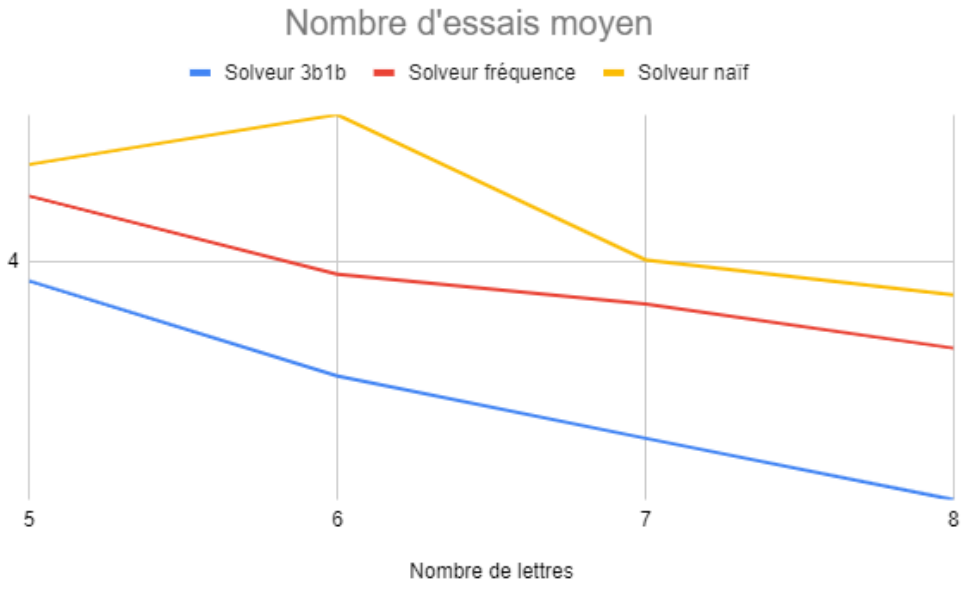



FIGURE 6.4 – Graphique montrant le nombre d'essais moyen pour chaque solveur

# Chapitre 7

# Gestion de Projet

## 7.1 Équipe de projet

L'équipe se compose de 4 étudiants en première année :
— Tristan SMAGGHE
— Jules BRUNET
— Louis-Vincent CAPELLI
— Malo DAMIEN

Tristan SMAGGHE a été désigné TechnoKing : il est le référent technique du groupe, celui à qui s'adresser en priorité en cas de problème ou de question. Louis-Vincent CAPELLI est le WordChamp : il rédigera les comptes-rendus de réunion et relira et corrigera les fautes dans le rapport avant le rendu final.

L'équipe est la même que celle de nos travaux dirigés et celle avec laquelle nous avions réalisé notre projet du premier semestre.

## 7.2 Analyse de Projet

### 7.2.1 Définition des objectif

Les objectifs ont été déterminés à l'aide de la méthode SMART : ils sont spécifiques, mesurables, atteignables, réalistes et temporellement définis.

### 7.2.2 Analyse des risques : Matrices SWOT

Nous avons évalué les forces et les faiblesses de l'équipe mais également les risques et les opportunités qui pourraient impacter la réalisation du projet. Nous avons résumé cela dans une matrice SWOT.

| FORCES | FAIBLESSES |
|---|---|
| - Tristan a beaucoup d'expérience en développement de site web avec base de données<br>- Équipe motivée et investie<br>- Tristan sait réaliser une intégration continue pour avoir accès à des versions intermédiaires<br>- | - Premier projet en C<br>- Faibles connaissances du groupe du langage JavaScript |
| **OPPORTUNITÉS** | **MENACES** |
| - Réutiliser le mode de fonctionnement de notre groupe qui a déjà fait équipe sur le premier projet et apprendre de nos erreurs | - Les partiels |

FIGURE 7.1 – Matrice SWOT

## 7.3 Organisation du projet

Afin que chaque membre de l'équipe puisse travailler sur le backend et le frontend, nous avons réalisé un découpage en lots de travail réalisables et individualisés (cf. WBS annexe 4) que nous avons ensuite distribués aux membres de l'équipe en fonction de leurs compétences, leurs affinités et pour chacun ait une quantité de travail équivalente grâce à une matrice RACI (cf. annexe 5). Pour ce qui est du solveur, nous avons décidé de laisser la création des lots de travail et leur répartition à plus tard, après avoir réalisé l'état de l'art. Nous avons finalement réparti le travail comme décrit dans les CR3 et CR4 (cf. annexe 3).

## 7.4 Outils de travail

### 7.4.1 IDE

L'ensemble de l'équipe a utilisé Visual Studio Code et notamment son extension SQLLite Reader.

### 7.4.2 Logiciel de gestion de version

Durant la totalité du projet nous avons utilisé le répertoire GitLab de l'école et notre dossier prévu à cet effet. Les clients git utilisés sont GitKraken, et le logiciel git. L'entretien de la branche a été géré par Tristan SMAGGHE : il a notamment organisé le projet et vérifié quotidiennement le bon fonctionnement du git.

### 7.4.3 Rédaction du rapport

Pour la rédaction du rapport nous avons utilisé le rédacteur et compilateur en ligne Overleaf (l'instance de l'école).

### 7.4.4 Communication

Pour la communication et l'échange entre les membres du groupe nous avons mis en place un serveur Discord. Sur ce dernier nous avons déployé un bot qui affichait sous forme de notifications chaque commit, ce qui nous permettait d'être au courant des dernières modifications. De plus lorsque nous n'étions pas disponibles pour une réunion en présentiel nous utilisions ce serveur pour faire nos réunions en distanciel.
L'utilisation de cet outil présente tout de même un inconvénient : beaucoup d'informations étaient échangées hors-réunions ce qui peut compliquer le compréhension des compte-rendus successifs.

# Chapitre 8

# Bilan

## 8.1 Bilan du travail réalisé

Nous avions prévu d'implémenter de plus nombreux modes de jeu alternatifs pour diversifier l'expérience de jeu sur WordChamp. Pour s'assurer de pouvoir rendre un produit fini satisfaisant, nous avions décidé de ne commencer à implémenter ces modes de jeux qu'après avoir fini la première version de WordChamp contenant toutes les fonctionnalités indispensables et après avoir commencé le solveur. Cependant, l'évaluation de l'application Web a finalement été effectuée à mis-parcours et nous avons donc abandonné ces modes de jeux qui n'auraient pas été évalués pour se concentrer sur le solveur.

## 8.2 Ressentis personnels

### 8.2.1 Jules BRUNET

| | |
|---|---|
| Points positifs | Même groupe de travail, même ambiance donc des conditions idéales de travail<br>Pas de rush de fin pour finir le projet<br>Un projet plutôt bien cadré du début à la fin |
| Difficultés rencontrées | Nouveaux outils (VueJS et le Javascript notamment)<br>Manque de motivation par moments (surtout au début du projet)<br>Très peu de temps libre pour la partie solveur |
| Expérience personnelle | Deuxième projet de ce type : beaucoup moins de pression que pour le premier<br>Gain d'expérience en gestion de base de donnée, VueJS et C |
| Axes d'améliorations | Gestion du temps de travail (comme toujours)<br>Phases de Test à améliorer<br>Plus de temps à passer sur le solveur |

### 8.2.2 Louis-Vincent CAPELLI

| | |
|---|---|
| Points positifs | - Nous avons choisi de conserveer la même équipe projet ce qui nous a permis d'être efficace immédiatement et de profiter des habitudes établies au premier semestre.<br>- M'a permis d'apprendre le JavaScript et de découvrir plus en détails le développement web.<br>- M'a permis de mettre en place les axes d'amélioration établis lors du projet précédent. |
| Difficultés rencontrées | - Se mettre dans le rythme au début du projet et travailler régulièrement dessus plutôt qu'en grosses sessions éparses.<br>- Prise en main de C au début du projet solveur et galère sous Windows. |
| Expérience personnelle | - J'ai aimé travailler sur ce projet qui m'a fait découvrir le développement web avec JS/HTML/CSS que j'ai particulièrement apprécié et qui m'a donné envie d'en apprendre plus sur tout ce qu'on peut faire facilement en CSS sans utiliser de framework.<br>- L'ambiance au sein du groupe était très agréable, chacun aidant les autres et partageant ses connaissances et découvertes. |
| Axes d'améliorations | - Se mettre dans l'ambiance plus tôt même si j'ai été moins en rush sur la fin que lors du projet précédent. |

### 8.2.3 Malo DAMIEN

| | |
|---|---|
| Points positifs | Equipe projet qui fonctionne bien en avance sur le rendu du projet<br>Expérience et découverte de Vue JS pour le web mais aussi de C pour le solveur<br>Recherche algorithmique satisfaisante<br>Équipe projet toujours disponible pour de l'aide |
| Difficultés rencontrées | Certaines difficultés de communication de mes problèmes<br>Difficulté de programmation en C langage très exigeant (mais satisfaction lorsque tout est bien fait)<br>Corrections de mes algorithmes souvent laborieuses |
| Expérience personnelle | Apprentissage C<br>Apprentissage de nouveaux algorithmes et découverte de la théorie de l'information<br>Satisfaction de voir les parties de chacun s'imbriquer ensemble (malgré un debug nécessaire) |
| Axes d'améliorations | Mélanger nos solveurs en prenant les parties les plus performantes de chacun<br>Travail plus régulièrement réparti |

### 8.2.4 Tristan SMAGGHE

| | |
|---|---|
| Points positifs | Garder la même équipe projet.<br>Intégration en continu.<br>Ouverture à de nouvelles technologies. |
| Difficultés rencontrées | Manque de temps pour faire plus que prévu sur la partie Web. |
| Expérience personnelle | Progression en algorithmie et informatique.<br>Progression en LaTeX. |
| Axes d'améliorations | Répartition des tâches. |

## 8.3 Taux horaires

| Personne | Temps sur Gestion de Projet | Temps de Travail | Total |
|---|---|---|---|
| Tristan SMAGGHE | 16h | 131h | 147h |
| Malo DAMIEN | 19h | 119h | 138h |
| Louis-Vincent Capelli | 24h | 104h | 128h |
| Jules BRUNET | 16h | 113h | 129h |

FIGURE 8.1 – Tableau des taux horaires des membres du groupe

# Annexes

# Annexe 1

**Schéma Entités-Associations de notre base de données.**

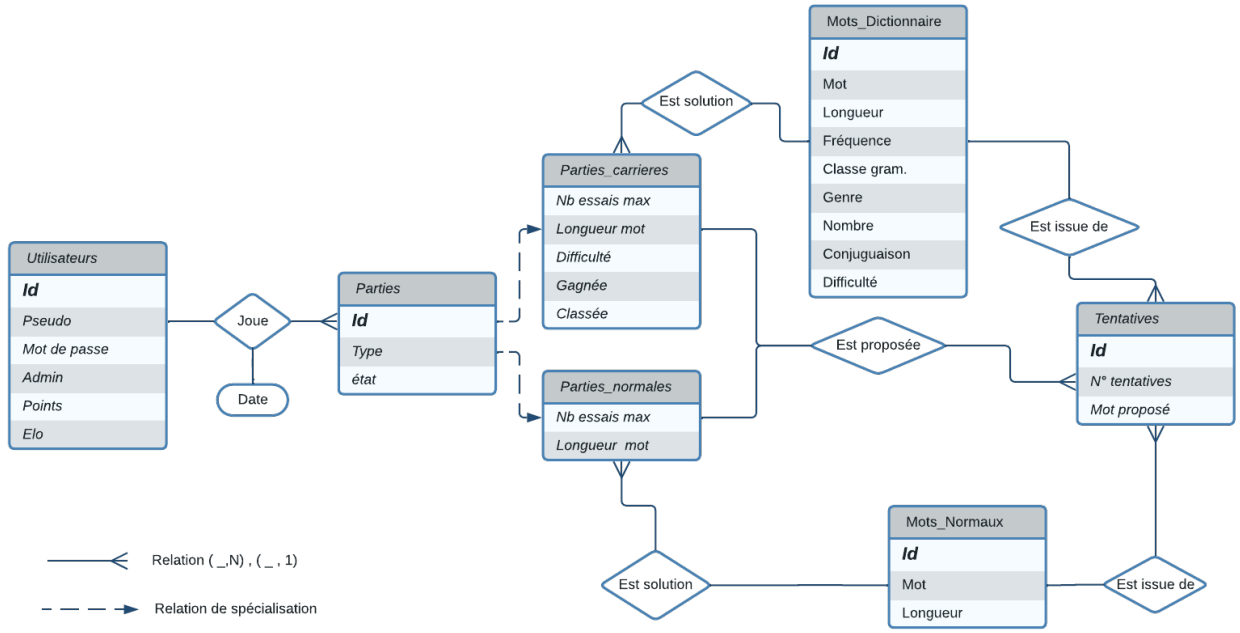



FIGURE 2 – Schéma entité association de notre base de données

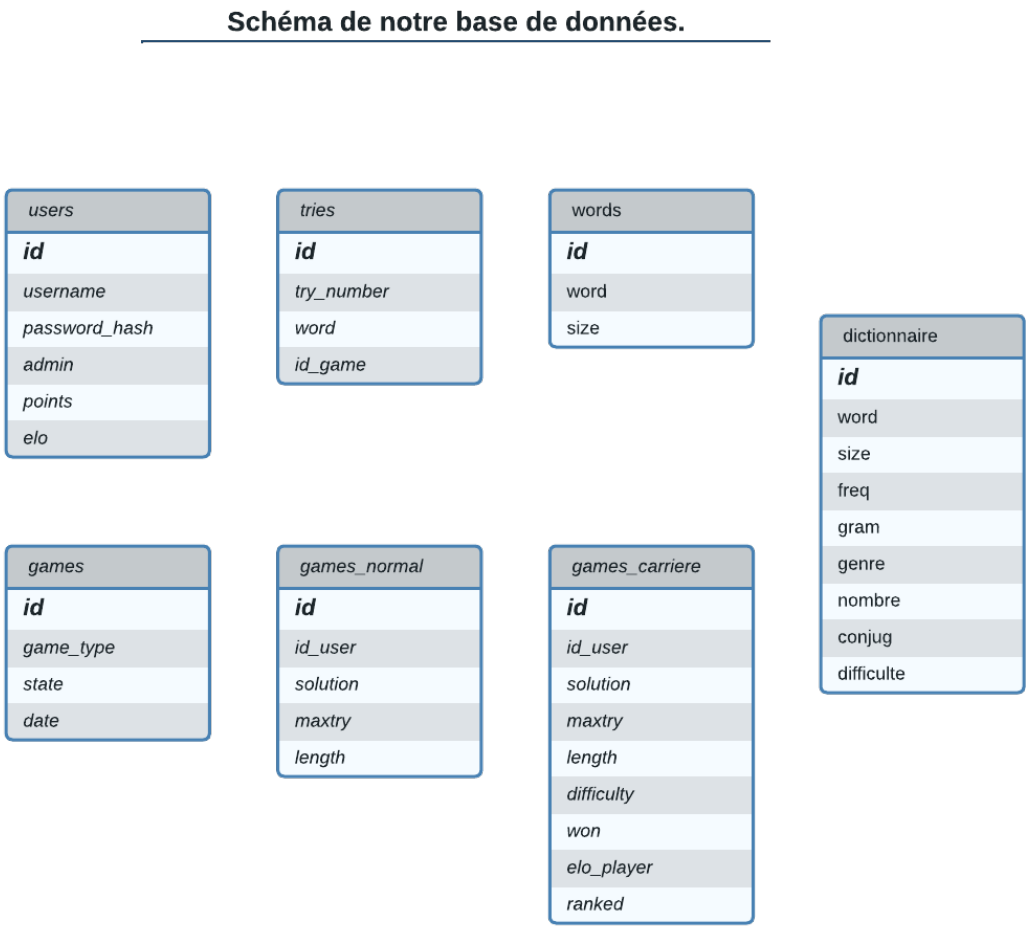



FIGURE 3 – Tables et attributs utilisés par la base de données

| Nom de la table | Nom de l'attribut | Description |
|---|---|---|
| Users | id | (String) Clé primaire |
| Users | username | (String) Nom d'utilisateur |
| Users | password_hash | (String) Mot de passe hashé |
| Users | admin | (Bool) Définini si l'utilisateur est admin ou non |
| Users | points | (Int) Nombre de points de l'utilisateur |
| Users | elo | (Float) Elo de l'utilisateur |
| Tries | id | (String) Clé primaire |
| Tries | try_number | (Int) N° de tentatives |
| Tries | word | (String) Mot proposé |
| Tries | id_game | (String) Clé étrangère de game |
| Words | id | (String) Clé primaire |
| Words | word | (String) Mot |
| Words | size | (Int) Longueur du mot |
| Dictionnaire | id | (String) Clé primaire |
| Dictionnaire | word | (String) Mot |
| Dictionnaire | size | (Int) Longueur du mot |
| Dictionnaire | freq | (Float) Fréquence d'utilisation du mot dans la langue française |
| Dictionnaire | gram | (String) Classe grammaticale du mot |
| Dictionnaire | genre | (String) Genre du mot |
| Dictionnaire | nombre | (String) Nombre du mot (S/P) |
| Dictionnaire | conjug | (String) Comment est conjugué le mot |
| Dictionnaire | difficulte | (Float) Difficulté du mot |
| Games | id | (String) Clé primaire |
| Games | game_type | (String) Type de partie |
| Games | state | (Bool) Indique si la partie est finie ou non |
| Games | date | (Int) Date de création de la partie |
| Games_normal | id | (String) Clé primaire et clé étrangère de games |
| Games_normal | id_user | (String) Clé étrangère de user, celui qui joue la partie |
| Games_normal | solution | (String) Mot solution |
| Games_normal | maxtry | (Int) Nombre d'essais max |
| Games_normal | length | (Int) Longueur du mot solution et donc des mots proposables |
| Games_carriere | id | (String) Clé primaire et clé étrangère de games |
| Games_carriere | id_user | (String) Clé étrangère de user, celui qui joue la partie |
| Games_carriere | solution | (String) Mot solution |
| Games_carriere | maxtry | (Int) Nombre d'essais max |
| Games_carriere | length | (Int) Longueur du mot solution et donc des mots proposables |
| Games_carriere | difficulty | (Float) Difficulté du mot solution |
| Games_carriere | won | (Bool) Indique si la partie est gagnée ou non |
| Games_carriere | elo_player | (Float) Elo du joueur au moment où il a lancé la partie |
| Games_carriere | ranked | (Bool) Indique si la partie est classée ou non |

# Annexe 2

| Taille | Nb mots total | H tot. | Meilleur mot | H mot | Mots restants (moyenne) |
|---|---|---|---|---|---|
| 3 | 633 | 9.31 | AIE | 3.016 | 78.25 |
| 4 | 2 623 | 11,36 | SAIE | 4.821 | 92.80 |
| 5 | 7 980 | 12.96 | RAIES | 6.349 | 97.90 |
| 6 | 17 991 | 14.14 | TARIES | 7.856 | 77.65 |
| 7 | 32 230 | 14.98 | TARINES | 9.093 | 59.01 |
| 8 | 48 039 | 15.55 | RATINEES | 10.117 | 43.26 |
| 9 | 59 584 | 15.86 | CERTAINES | 11.211 | 25.14 |
| 10 | 62 954 | 15.94 | CETONURIES | 12.117 | 14.17 |
| 11 | 57 784 | 15.82 | RACONTEUSES | 12.698 | 8.70 |
| 12 | 46 501 | 15.50 | RETICULAIRES | 13.187 | 4.99 |
| 13 | 32 962 | 15.01 | METICULOSITES | 13.406 | 3.04 |
| 14 | 20 886 | 14.35 | RENONCIATAIRES | 13.362 | 1.98 |
| 15 | 12 077 | 13.56 | CENTRALISATIONS | 13.014 | 1.46 |
| 16 | 5 690 | 12.57 | DEMORALISATRICES | 12.191 | 1.22 |

FIGURE 4 – Tableau exposant les résultats du solveur par minimisation d'entropie, pour la première proposition

# Annexe 3

## Compte-rendu de la réunion du 23/03/22

Rédigé le 27 mars 2022 par Louis-Vincent CAPELLI

### Présents

— Tristan
— Jules
— Malo
— Louis-Vincent

### Ordre du jour

— Se mettre enfin d'accord sur un nom pour le projet
— Attribuer à chacun un rôle dans l'équipe
— Prévoir le déroulement du projet

### Nom du projet

L'équipe a finalement décidé à l'unanimité du nom WordChamp (qui fait écho à l'aspect compétitif qui sera la particularité de notre Wordle-like). Le nom de domaine wordchamp.info a été acheté.

### Rôles au sein de l'équipe

Tristan sera notre "TeknoKing", il sera le référent technique du projet à qui il sera privilégié de s'adresser en cas de question ou de problème technique.

Louis-Vincent sera le "WordChamp", il prendra en note ce qui se dit en réunion, rédigera les CR et relira le rapport.

Les autres fonctions seront occupées par tous les membres du groupe.

### Déroulement du projet

Nous avons esquissé une première frise chronologique qui sera affinée une fois le WBS fait, sous la forme d'un Gantt. L'idéal serait d'avoir un Wordle fonctionnel avant les vacances, aux alentours du 10 mars. Nous aurions alors jusqu'au 30 avril (date de l'évaluation de mi-projet) pour peaufiner l'aspect graphique et commencer à implémenter le solveur. Une fois le solveur fini, nous devrions avoir le temps de rédiger le reste du rapport et de me mettre en place les spécificités de WordChamp à savoir : un aspect compétitif avec un classement et un mode entraînement avec possibilité d'avoir des indices.

### Pour la prochaine réunion le 03/04

| Personne affféctée | Tâche |
|---|---|
| Tout le monde | Commencer à réfléchir au WBS qui sera fait ensemble le 03/04 |

# Compte-rendu de la réunion du 5/04/22

Rédigé le 5 avril 2022 par Louis-Vincent CAPELLI

## Présents

— Tristan
— Jules
— Malo
— Louis-Vincent

## Ordre du jour

— Créer le WBS
— Répartir les tâches dans une matrice RACI

## WBS

L'équipe est arrrivé à ce WBS :


Nous avons décidé de laisser la répartition des tâches du solveur pour plus tard. En effet, nous ne sommes actuellement pas assez au courant de ce qui se fait et de comment nous allons faire pour nous projeter si loin.

## Matrice RACI

En tenant en compte des envies, des préférences et des compétences de chacun, voilà la matrice RACI qui a été établie :

| Lot | Tristan | LV | Jules | Malo |
|---|---|---|---|---|
| 1.1 | CI | I | RAI | I |
| 1.2.1 | C | | R | RAI |
| 1.2.2.1 | C | RAI | | |
| 1.2.2.2 | C | | RA | |
| 1.2.3 | RA | | | |
| 2.1.1.1 | C | RA | | |
| 2.1.1.2 | C | RA | | |
| 2.1.2.1 | RA | | | |
| 2.1.2.2 | RA | | | |
| 2.2.1.1 | C | | I | RA |
| 2.2.1.2 | C | | I | RA |
| (2.2.1.3) | C | | RA | |
| 3.1.1 | R | R | RA | R |

## Pour la prochaine réunion le 04/05

| Personne affféctée | Tâche |
|---|---|
| Tout le monde | Chacun doit implémenter ce qui lui a été attribué |
| Tout le monde | Commencer à réfléchir au solveur et à se renseigner<br>Regarder la vidéo de 3B1B |

# Compte-rendu de la réunion du 04/05/22

Rédigé le 4 mai 2022 par Louis-Vincent CAPELLI

## Présents

— Tristan
— Jules
— Malo
— Louis-Vincent

## Ordre du jour

— Mettre en commun les infos
— Choisir des implémentations
— Se repartir les tâches

## Mise en commun des infos

Nous avons comme prévu tous regardé la vidéo de 3Blue1Brown. LV propose également aux autres de regarder une autre vidéo qui explique une implémentation qui cherche à construire un profil de mot "parfait" en réalisant une analyse fréquentielle des mots de la langue.

## Choix des implémentations

Nous choisissons d'implémenter un solveur naïf qui servira à comparer ses performances avec les solveurs plus évolués, un solveur similaire à celui de la vidéo de 3B1B, et un qui réalise une analyse fréquentielle comme suggéré par LV.

## Répartition

Jules se propose pour réaliser le module qui permettra d'ouvrir les fichers qui contiennent les dictionnaires.

Malo réalisera un Wordle en C qui servira à tester nos solveurs.

LV réalisera le solveur naïf pour se familiariser avec le C.

Tristan mettra en place l'environnement de développement du projet pour faciliter le développement des différentes fonctions et structures en C.

## Pour la prochaine réunion

| Personne afféctée | Tâche |
|---|---|
| Tout le monde | Chacun doit implémenter ce qui lui a été attribué |

# Compte-rendu de la réunion du 20/05/22

Rédigé le 20 mai 2022 par Louis-Vincent CAPELLI

## Présents

— Tristan
— Jules
— Malo
— Louis-Vincent

## Ordre du jour

— Répondre aux questions de chacun
— Distribuer les implémentations

## Questions et discussions

On réfléchit à implémenter des tables de hashage pour une nouvelle implémentation mais pour cela ne semble pas nécessaire, peut-être plus tard.

Malo suggère l'implémentation d'un wrapper qui permettra de séléctionner l'implémentation de solveur choisie par l'utilisateur.

Tristan propose de faire un testeur en go.

Malo fait remarquer qu'il faudrait séparer les dictionnaires par nombre de lettres pour faciliter le traitement.

LV reçoit l'aide de Tristan sur le choix des structures de données à implémenter pour son solveur.

Tristan propose une implémentation à l'aide d'un arbre qui calculerait à l'avance toutes les réponses possibles du solveur pour chaque mot et choisirait comme tentative le mot qui permettrait d'en éliminer le plus d'autres.

## Répartition

— Malo commencera à implémenter la méthode de l'analyse fréquentielle.
— Jules commencera l'implémentation de la video de 3B1B.
— LV finit son implémentation naïve.
— Tristan commencera l'implémentation évoquée plus haut.

## Pour la prochaine réunion

| Personne afféctée | Tâche |
|---|---|
| Tout le monde | Chacun doit implémenter ce qui lui a été attribué |

# Compte-rendu de la réunion du 23/05/22

Rédigé le 23 mai 2022 par Louis-Vincent CAPELLI

## Présents

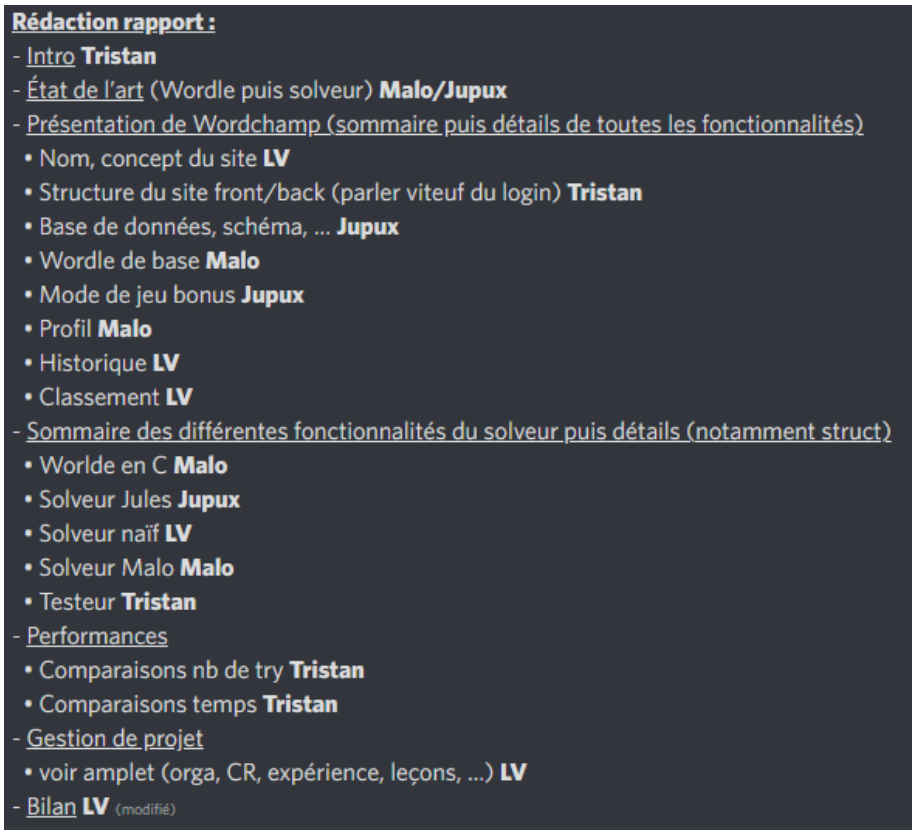
— Tristan
— Jules
— Malo
— Louis-Vincent

## Ordre du jour

— Créer la structure finale du rapport et se répartir sa rédaction

## Rapport

En s'inspirant en partie du premier rapport que nous avons rédigé et en donnant à rédiger à chacun les parties du rapport sur lesquelles il a travaillé, nous sommes arrivés à cette répartition :



# Compte-rendu de la réunion du 28/05/22

Rédigé le 28 mai 2022 par Louis-Vincent CAPELLI

## Présents

— Tristan
— Jules
— Malo
— Louis-Vincent

## Ordre du jour

— Faire un point sur l'avancement du rapport
— Rédaction de la section Bilan

## Avancement du rapport

Chacun évoque ce qu'il lui reste à faire pour s'assurer que nous pourrons rendre le rapport dans les temps :
— Tristan : rien
— Jules : partie sur son solveur et partie base de données et remettre en forme les Gantt
— Malo : fin de la partie sur le mode de jeu principal WordChamp, comparaison avec le solveur de Jules et commentaires dans tests et performances
— Louis-Vincent : Renuméroter les annexes, relire les parties des autres
Il faudra que tout soit fait avant le 05/06 à 23h59 pour laisser le temps à LV de tout relire pour corriger les fautes.

## Bilan

Chacun doit remplir sa partie du bilan dans le rappport, la template a été insérée par LV au préalable en s'inspirant de celle d'Amplet.

## D'ici le 05/06

| Personne afféctée | Tâche |
|---|---|
| Tout le monde | Chacun doit remplir sa partie du Bilan |
| Malo | Fin de la partie sur le mode de jeu principal WordChamp,<br>comparaison avec le solveur de Jules et commentaires dans tests et performances |
| Jules | Partie sur son solveur et partie base de données et remettre en forme les Gantt |
| Tristan | Refaire l'architecture de la partie solveur du projet |
| LV | Renuméroter les annexes, relire les parties des autres |

# Annexe 4

## WBS

### 1. Backend

1.1 Base de données

1.2 Interface de programmation d'application (API)

- 1.2.1 Traitement des réponses du joueur
- 1.2.2 Gestion des parties
  - 1.2.2.1 Historiques
  - 1.2.2.2 Partie en cours
- 1.2.3 Login

1.3 Modes de jeu additionnels

### 2. Frontend

2.1 Profil

- 2.1.1 Historiques
  - 2.1.1.1 Historiques sommaire (résumé des parties)
  - 2.1.1.2 Affichage d'une partie en particulier
- 2.1.2 Paramètres utilisateur
  - 2.1.2.1 Afficher les infos personnelles
  - 2.1.2.2 Pouvoir modifier les infos personnelles

2.2 Jeu(x)

- 2.2.1 Mode classique
  - 2.2.1.1 Grille interactive
  - 2.2.1.2 Clavier virtuel
  - (2.2.1.3 Sons)
- 2.2.2 Modes de jeu additionnels

### 3. Solveur

3.1 Recherches et documentation

- 3.1.1 Etat de l'art
- 3.1.2 Décisions du fonctionnement de notre solveur

3.2 Implémentation en C

- 3.2.1 Gestion des fichiers
- 3.2.2 Implémentation de la méthode choisie (découpe après 3.1)
- 3.2.3 Tests de correction
- 3.2.4 Tests d'efficacité

FIGURE 5 – WBS

# Annexe 5

| Lot | Tristan | LV | Jules | Malo |
|---|---|---|---|---|
| 1.1 | CI | I | RAI | I |
| 1.2.1 | C | | R | RAI |
| 1.2.2.1 | C | RAI | | |
| 1.2.2.2 | C | | RA | |
| 1.2.3 | RA | | | |
| 2.1.1.1 | C | RA | | |
| 2.1.1.2 | C | RA | | |
| 2.1.2.1 | RA | | | |
| 2.1.2.2 | RA | | | |
| 2.2.1.1 | C | | I | RA |
| 2.2.1.2 | C | | I | RA |
| (2.2.1.3) | C | | RA | |
| 3.1.1 | R | R | RA | R |

FIGURE 6 – Matrice RACI

# Bibliographie


[1] 3Blue1Brown. Solving wordle using information theory. https://www.youtube.com/watch?v=v68zYyaEmEA, 2022.

[2] Grant Sanderson (3Blue1Brown). Oh, wait, actually the best wordle opener is not "crane".... https://www.youtube.com/watch?v=fRed0Xmc2Wg, 2022. [En ligne ; vu le 25 avril 2022].

[3] Grant Sanderson (3Blue1Brown). Solving wordle using information theory. https://www.youtube.com/watch?v=v68zYyaEmEA/, 2022. [En ligne ; vu le 25 avril 2022].

[4] Sam Walker (The Dodgy Engineer). Solving wordle in under 3 guesses in python. https://www.youtube.com/watch?v=fVMlnSfGq0c/, 2022. [En ligne ; vu le 25 mars 2022].

[5] Yotam Gafni. Automatic wordle solving. https://towardsdatascience.com/automatic-wordle-solving-a305954b746e, 2022. [En ligne ; vu le 27 mars 2022].

[6] Andrew Ho. Solving wordle with reinforcement learning. https://wandb.ai/andrewkho/wordle-solver/reports/Solving-Wordle-with-Reinforcement-Learning--VmlldzoxNTUzOTc4, 2022. [En ligne ; vu le 26 mars 2022].

[7] Documentation Lexique. Lexique.org. http://www.lexique.org/, 2022. [En ligne ; vu le 20 avril 2022].

[8] Contributions multiples. Entropie de shannon. https://fr.wikipedia.org/wiki/Entropie_de_Shannon, 2022. [En ligne ; vu le 17 avril 2022].

[9] Contributions multiples. Théorie de l'inofrmation. https://fr.wikipedia.org/wiki/Théorie_de_l'information, 2022. [En ligne ; vu le 17 avril 2022].

[10] Contributions multiples. Théorie des jeux. https://fr.wikipedia.org/wiki/Théorie_des_jeux#Typologie, 2022. [En ligne ; vu le 17 avril 2022].

[11] Documentation SQLAlchemy. Loading inheritance hierarchies. https://docs.sqlalchemy.org/en/14/orm/inheritance_loading.html, 2022. [En ligne ; vu le 12 avril 2022].

[12] Documentation SQLAlchemy. Mapping class inheritance hierarchies. https://docs.sqlalchemy.org/en/14/orm/inheritance.html, 2022. [En ligne ; vu le 12 avril 2022].