



PROJET PPII2
PROJET PLURIDISCIPLINAIRE D'INFORMATIQUE INTÉGRATIVE

Rapport de projet :
Wordle

Auteurs :

Thomas KIEFFER

Julien DE TOFFOLI

Nathan IORI-GINGEMBRE

Pierre PASQUIER

Table des matières

Introduction	1
1 Conception et implémentation de la partie Web	2
1.1 Conception du site	2
1.2 Template et hub	3
1.3 Inscription et connexion	4
1.4 partie libre	5
1.5 partie journalière	5
1.6 partie survie	6
1.7 historique	6
1.8 classement	7
2 Conception et théorie de l'information	8
2.1 Le jeu en ligne Wordle	8
2.2 Théorie de l'information	9
2.3 Conception du Solveur	11
3 Implémentation et spécificités du Solveur	12
3.1 Implémentation	12
3.1.1 Structure d'arbre préfixe pour stocker les mots	12
3.1.2 Parcours de l'arbre et calcul des paternes des mots	14
3.1.3 Stockage des informations	20
3.2 Spécificités et interaction	24
3.2.1 Arbres motif - meilleur mot	24
3.2.2 Interaction utilisateur	28

4	Tests et performances du solveur	30
4.1	Tests - Méthode du Right-BICEP	30
4.1.1	Fonction <i>paterne</i>	30
4.1.2	Fonction <i>mot_suivant</i>	33
4.2	Analyse en complexité des fonctions majeures	36
4.3	Performances et comparaison des algorithmes	36
4.3.1	Comparaison temps de calcul avec algorithme de parcours dans un arbre et algorithme de parcours dans un tableau pour la partie prétraitement	36
4.3.2	Temps d'exécution du solveur	38
5	Gestion de Projet	40
5.1	Gestion de Projet : Application Web	40
5.1.1	Forces et faiblesses de notre projet : matrice SWOT	40
5.1.2	Gestion des risques : évaluation de la fréquence et de la gravité des événements	42
5.1.3	Découpage des lots de travail	42
5.1.4	Répartition des responsabilités	43
5.2	Gestion de Projet : Solveur en C	44
5.2.1	Forces et faiblesses de notre projet : Matrice SWOT	44
5.2.2	Gestion des risques : évaluation de la fréquence et de la gravité des événements	45
5.2.3	Découpage en lots de travail	45
5.2.4	Responsabilités sous-jacentes	46
	Conclusion	48
	Annexes	48
	Annexe 1 : Comptes rendus	49
	Annexe 2 : Diagramme de Gantt Web	72
	Annexe 3 : WBS Web	72
	Annexe 4 : Gestion des risques WEB	73

Annexe 5 : Diagramme de Gantt Web	75
Annexe 6 : WBS Solveur	75
Annexe 7 : Gestion des risques Solveur	76

Introduction

Depuis le lundi 31 janvier, le célèbre jeu Wordle rejoint désormais la section des jeux du célèbre journal américain le New York Times après son rachat à plusieurs millions de dollars. Ainsi c'est dans cette vague de popularité du Wordle que se situe notre sujet, tout d'abord de créer notre propre jeu Wordle avec nos idées, puis de créer un solveur permettant de résoudre n'importe quel Wordle, avec nos stratégies et structures de données.

Nous nous focaliserons dans ce rapport de projet sur le solveur, que nous avons dû à réaliser en complément de l'application Web.

Nous avons commencé par réfléchir à la structure de donnée à utiliser, afin d'optimiser stockage et performances, avant de décider de quelle stratégie privilégier. Se focaliser sur le stockage, les performances ou équilibrer les deux ?

C'est dans cette optique de réflexion que nous nous sommes renseignés sur diverses structures ainsi que sur la théorie de l'information, permettant l'optimisation du calcul du meilleur mot.

Chapitre 1

Conception et implémentation de la partie Web

1.1 Conception du site

Lors de la conception du site, nous avons de base comme objectifs de faire une partie libre paramétrable ainsi qu'un historique des parties du joueurs. Après nos réunions nous nous sommes fixés sur les fonctionnalités suivantes :

- Un système de template pour que les pages aient le même design
- Un système d'inscription et de connexion afin d'y associer un historique au joueur
- Un mode de partie libre
- Un mode de partie journalier
- Un mode de partie survie
- Un historique lié au joueur
- Un classement prenant le temps des meilleurs joueurs du mode survie

1.2 Template et hub

Afin de suivre nos idées d'améliorations du premier projet, nous avons donc utilisé le concept de « blocks » avec *Flask*, permettant de créer une page Template qui sera un parent des pages fils qui seront dans leur affichage incluse dans le père. Cette méthode a permis également d'en incluant les paramètres dans l'affichage du fils, de faire afficher sans soucis le template père. Pour le hub de notre site, nous avons décidé d'afficher les 3 modes possibles, libre, journalier et survie, ainsi que la possibilité d'accéder à la connexion / inscription. Le template contenant les bannières supérieurs et inférieurs de la page sont bien visibles.

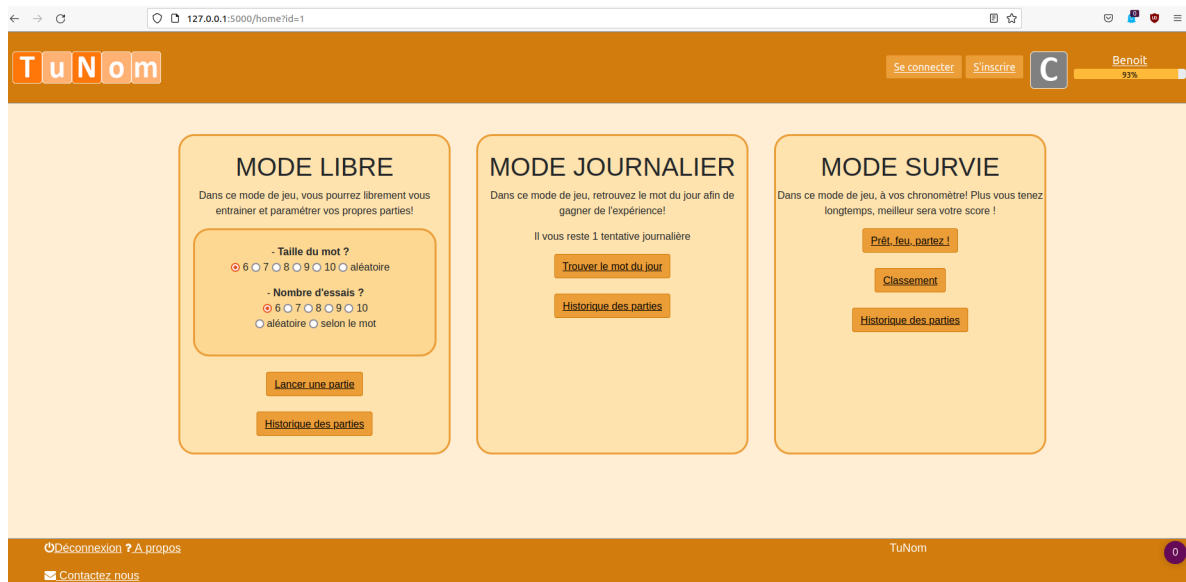


FIGURE 1.1 – Page d'accueil

Les images comme les badges et le logo ont été faites via Gimp et sont chargées lors du chargement de la page. Pour un utilisateur connecté, on peut voir en haut à droite sa jauge d'expérience ainsi que son badge reflétant son niveau sur le site, niveau augmentant grâce aux parties survie et journalière, et si un utilisateur dépasse le niveau maximal de notre base de donnée, son badge restera le dernier mais son pourcentage vers le niveau supérieur continuera de monter, permettant à l'utilisateur s'il le souhaite de continuer afin de viser le pourcentage le plus haut possible.

1.3 Inscription et connexion

Pour la partie inscription et connexion, nous voulions que l'utilisateur soit anonyme sur le site. Nous avons donc décidé de ne pas lui demander son nom ou son prénom mais un pseudo ce qui lui permet quand même d'être reconnaissable par les autres utilisateurs. Ce pseudo doit donc être unique pour ne pas confondre les utilisateurs. Nous demandons également l'adresse mail de l'utilisateur afin de pouvoir sécuriser le site car nous envoyons un code à l'adresse mail donnée pour s'assurer que des personnes ne s'amuse pas à utiliser l'adresse mail d'autres personnes ou ne s'inscrivent pas avec de nombreux comptes différents. Ainsi, il a fallut transmettre les informations de la page d'inscription, où l'utilisateur a entré ses données, jusqu'à la page de vérification de l'adresse mail. Pour cela, nous avons fait passer ces informations par l'url mais, vu qu'elle contenait des informations telles que l'adresse mail, le mot de passe ou le numéro de téléphone de l'utilisateur, nous avons décidé de chiffrer ces données. Se connecter donne à l'utilisateur accès à son historique, au mode journalier ainsi qu'à sa présence dans le classement de survie. Cela lui permet aussi de voir son badge et son expérience en haut à droite de la page.

Il nous fallait aussi un moyen simple et unique d'identifier les utilisateurs. Nous avons donc associé un id unique à chaque utilisateur dès son inscription.

1.4 partie libre

Pour la partie libre, il s'agissait de pouvoir paramétrer le nombre de lettres du mot à deviner ainsi que le nombre d'essais requis pour le trouver. Pour ce faire, nous avons procédé par passage de paramètre à l'aide de formulaire sur la page home. Nous avons implémenté en *Javascript* un programme qui permet de suivre le déroulé de la partie, en restreignant notamment les inputs claviers n'étant pas des lettres de l'alphabet. Ce programme utilise également *Jquery* et *Jquery UI* afin d'utiliser les animations qui sont fournies avec (changement de couleurs, etc...). Nous avons aussi utilisé les sons originaux du jeu télévisé Motus lors du changement des cases, que nous avons stockés en ressources statiques sur le site. Pour permettre de mémoriser l'historique de la partie, à la fin de la partie (victoire ou défaite) un formulaire caché sur la page permettait de recenser les informations nécessaires au remplissage de notre base de données.

1.5 partie journalière

Pour la partie journalière, il suffisait pour la page de reprendre la page de la partie libre, et les vérifications et paramètres spécifiques seraient juste entrés en paramètres afin de pouvoir les réutiliser en sortie. Niveau concept, la partie journalière est une partie faisable une fois par jour, sur un mot fixe commun pour tout les utilisateurs, et cela implique donc deux choses, tout d'abord de vérifier que le joueur n'a pas déjà participé au mot du jour, et que le mot du jour reste commun aux utilisateurs quoi qu'il arrive. Concernant la vérification, nous avons associé à chaque utilisateur la date de sa dernière partie journalière, ce qui permet de vérifier son accès au mode de jeu, date que l'on mettra à jour dès le lancement de sa partie, afin d'éviter de retourner en arrière et de revenir sur la page pour recommencer la partie. DU côté du mot commun, il a fallu enregistrer la partie du joueur en avance, et bien que cela paraisse bizarre plutôt que de l'enregistrer après, le soucis de cette méthode est que si un utilisateur ne termine pas sa partie du jour, cela n'enregistrerait pas le mot du jour et donc il serait à nouveau généré et non global à tous les utilisateurs. Ainsi, en pré enregistrant la partie du joueur, cela permet pour la création du mot du jour de rechercher parmi les parties d'utilisateurs une partie d'aujourd'hui, en mode "daily", permettant 1) si aucun résultat n'est trouvé, aucune partie n'a été faite et il faut générer un mot et le pré enregistrer pour le premier utilisateur souhaitant y jouer, et 2) si un résultat est trouvé, il suffit dans le pré enregistrement de mettre ce mot et de l'envoyer comme mot à trouver dans la partie. Enfin, ce

mode ajoutant de l'expérience à l'utilisateur, il a suffi de mettre à jour la quantité d'expérience de l'utilisateur en lui ajoutant une fois celle associée au mode journalier.

1.6 partie survie

La partie survie reprend une partie du mode libre en terme de *Javascript*, à la différence du timer et de la gestion des évènements se déroulant côté client (changement de couleurs, changement de grille). Les vraies difficultés que nous avons pu rencontrer au cours de l'implémentation de ce mode ont été d'une part le timer, et d'autre part la gestion des timings en parallèle du timer. Il s'agissait aussi de récupérer les temps de survie et les mots donnés, chose que nous avons pu faire grâce à un formulaire caché sur la page pour garder l'esthétique du placement des blocs et ne pas trop surcharger la page (étant donné que théoriquement, une partie peut durer à l'infini). Du côté de la base de données, il s'agissait de réfléchir à une façon de gérer cet infini, chose que nous avons faite en limitant le remplissage des mots donnés dans la base de données, ce qui avait comme défaut de ne pas proposer un historique fiable des parties survie aux limites du modèle. Par ailleurs, comme le fait de deviner des mots de longueur plus élevée est plus compliqué et prend plus de temps, nous avons implémenté une récompense en temps supplémentaire sur le timer en fonction du nombre d'essais (de 60s à 0s, avec une décroissance de 6s par essai raté) mis à deviner le mot ainsi que la longueur du mot elle même. Pour modérer l'apparition des mots de longueur élevée, nous avons modifié la répartition des chances de tomber sur un mot d'une longueur fixée en partant de x% et en décrémentant de 5% à chaque longueur de mot.

1.7 historique

Pour l'historique, nous voulions pouvoir stocker toutes les informations des parties jouées dans la base de donnée afin de pouvoir afficher tous les mots entrés par l'utilisateur pour toutes les parties qu'il a joué. Pour cela, étant donné que les parties libres et journalières sont très similaires, nous avons créé une table Historique pour stocker les informations liées à ces types de parties. Il nous suffisait donc de stocker les mots entrés par l'utilisateur séparés chacun par une virgule et on recalculait ensuite les paternes associés pour afficher les bonnes couleurs dans l'historique. Pour les parties de type survie, nous avons créé une table Historique survie. Nous avons aussi stocké les mots entrés par l'utilisateur mais cette fois, vu que chaque partie contenait plusieurs mot à

deviner, il nous fallait un autre moyen de stocker les mots donnés par l'utilisateur. Pour cela, nous avons séparé les mots donnés par une virgule et à chaque fois que le mot à deviner changeait, nous avons mis un point virgule. Cela nous permettait donc de savoir quels mots ont été donnés pour quel mot à deviner et donc de reconstruire la partie jouée. Enfin, pour que l'affichage ne soit pas trop lourd pour les parties survie, nous avons permis à l'utilisateur de faire défiler les grilles pour chaque partie survie plutôt que d'afficher les grilles les unes à la suite des autres. Cela à été fait à l'aide d'un programme *Javascript*.

1.8 classement

Concernant finalement le classement, toute les informations nécessaires sont dans la base de données, dans l'historique de survie des joueurs. Nous avons donc pensé à afficher d'une part le classement général des meilleurs temps, et d'ajouter au dessus un extrait du classement, montrant la position du joueur (si connecté) ainsi que la personne avant lui et celle après lui du classement. Ainsi il a fallut réussir à faire une requête *SQL* permettant de récupérer les temps des joueurs, leur id et pseudo, dans l'ordre croissant, et en ne conservant par joueur que son meilleur temps afin de ne pas l'afficher deux fois dans le classement.

Chapitre 2

Conception et théorie de l'information

2.1 Le jeu en ligne Wordle

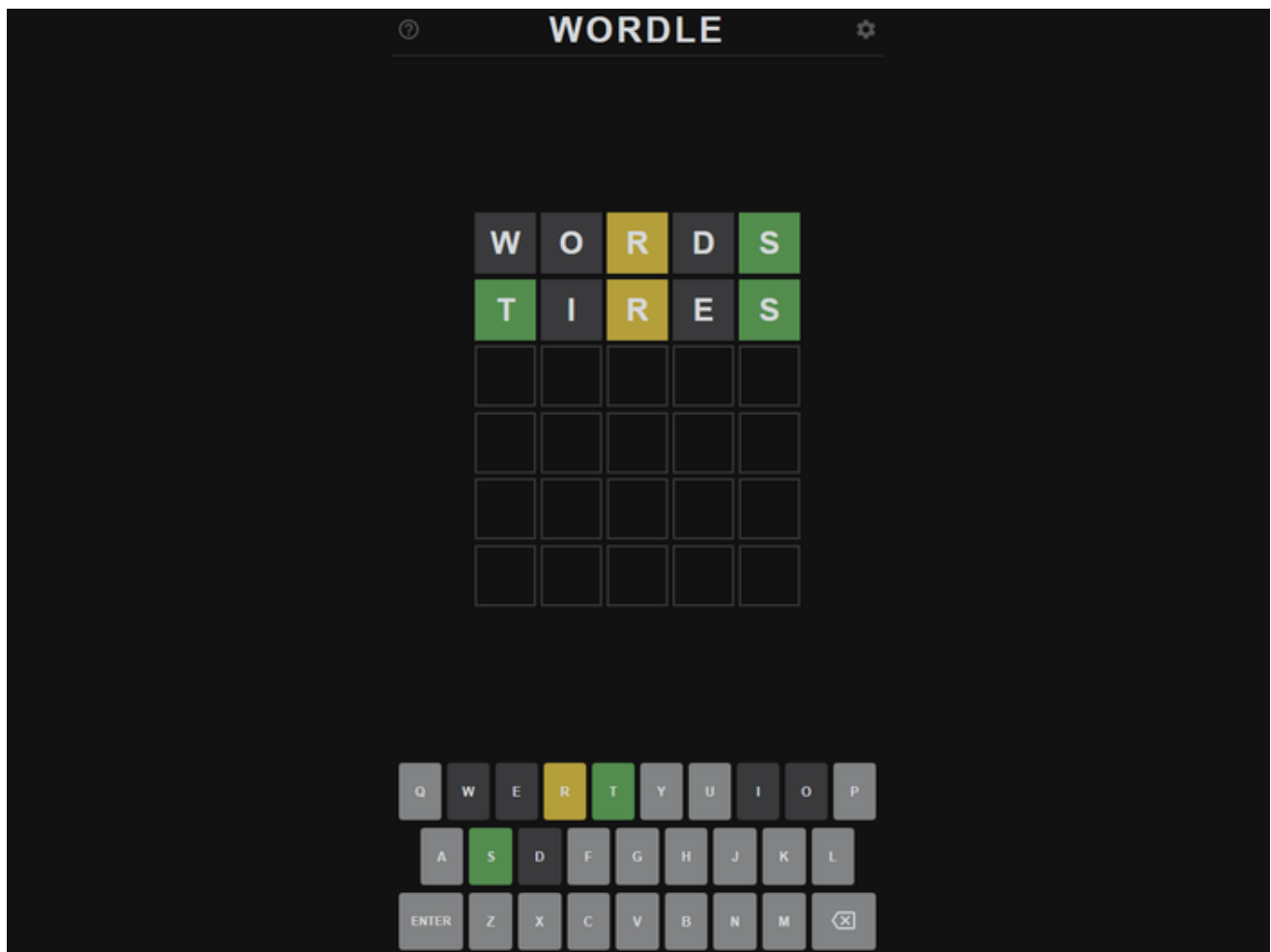


FIGURE 2.1 – Image d'une partie de Wordle

Le jeu en ligne *Wordle* (en anglais), est un jeu inspiré du jeu télévisé français *Motus* diffusé entre 1990 et 2019. Le principe du jeu repose sur le procédé suivant :

- Il faut deviner un mot, donc un mot est choisi parmi l'ensemble des mots proposés
- L'utilisateur doit saisir un mot, il ne peut que saisir des mots parmi l'ensemble des mots saisissables
- Après saisie, les cases se colorent pour signifier l'appartenance ou non d'une lettre au mot à deviner, c'est à dire « vert » lorsque la lettre est dans le mot au bon endroit, « jaune » lorsque la lettre est dans le mot mais pas au bon endroit et « gris » quand la lettre n'est pas dans le mot.
- Le procédé recommence à partir de la saisie de l'utilisateur, jusqu'à ce que son nombre d'essai soit épuisé

Nous pouvons remarquer que le jeu *Wordle* distingue mot saisissable et mot proposé, chose que nous ne feront pas. On parle ici d'aléatoire mais en vérité, l'ensemble des mots proposés est stocké dans une variable fixe et si l'on regarde le code source de la page, nous pouvons déjà savoir le mot à deviner du lendemain.

Nous avons tenté d'une part avec l'application Web de créer notre propre application la plus proche possible de *Wordle* en apportant des modes de jeu supplémentaires, et d'autre part avec le solveur, d'implémenter une résolution d'une grille de mot avec la théorie de l'information.

2.2 Théorie de l'information

La théorie de l'information a été introduite la première fois par Claude Edwood Shannon en 1948, qui désigne notamment l'information comme une quantité « mesurable ». On considère ici l'information comme le nombre de fois où on divise notre espace de recherche en deux. Dans le cadre du *Wordle*, il s'agit donc de trouver les mots qui détiennent l'information la plus grande afin de restreindre dès les premiers mots devinés, le nombre de mots restants possibles. Mais comme les mots à deviner sont tirés de manière aléatoire, cette grandeur ne suffit pas en terme de cohérence pour un meilleur mot. Considérant cet aléatoire, il faut donc prendre une moyenne, sur les tirages des mots, des informations, avec la probabilité correspondante de tomber sur tel ou tel pattern de couleurs après la saisie d'un mot. Dans l'ensemble du projet solveur, nous n'avons fait le calcul que sur le premier essai, c'est à dire le mot qui donne à chaque fois le plus d'information sur l'ensemble

des mots possibles dès le premier essai et nous avons répété ce procédé, comme si nous étions toujours au premier essai. Si on suppose un univers de possibles Ω , muni d'une probabilité p , alors on se ramène à un problème de la forme :

$$E(f(x)) = \sum_{x \in \Omega} (p(x) * f(x))$$

Avec $f(x)$ la fonction donnant, selon un élément x l'information qui lui est associée. La seule donnée que nous avons et qui est comparable pour chaque mot dans *Wordle* est la grille de couleurs apparaissant à la saisie du mot. En outre, ce sont les patterns que nous considérons comme élément nous fournissant de l'information. Comme l'information correspond à la division en une puissance de deux de l'espace de recherche, nous avons en notant $f(x)$ l'information relative à un pattern :

$$\left(\frac{1}{2}\right)^{f(x)} = p(x) \iff f(x) = -\log(p(x))$$

Donc nous avons :

$$H = \sum_{\text{pattern possible}} -p(\text{pattern}) * \log(p(\text{pattern}))$$

en désignant avec H l'entropie d'un mot. Nous avons donc cherché, au cours du calcul des meilleurs mots à maximiser cette entropie, en considérant ces mots équiprobables. La probabilité s'obtenait alors facilement selon :

$$p(\text{pattern}) = \frac{\text{Nombre de mots restants}}{\text{Nombre de mots total}}$$

2.3 Conception du Solveur

Nous avons conçu notre Solveur en C de telle sorte que :

- le déroulé de la partie soit rapide quitte à sacrifier du temps en pré-traitement
- le dictionnaire soit changeable à condition de respecter les conventions d'écriture des fichiers

Ainsi nous nous sommes ramenés à deux problèmes, l'un où l'on générerait les meilleures parties au sens de l'entropie pour chaque réponse possible de l'utilisateur via le terminal, et l'autre où il s'agissait de lire les données fournies par tout le pré-traitement. Nous nous sommes donc intéressés à la structure de données des arbres préfixes, puisqu'elle permet de ranger les mots et de savoir si un mot possède un préfixe commun avec un autre mot, et un autre type d'arbre, permettant, à la manière de l'arborescence des parties possibles, de se déplacer selon l'entrée de l'utilisateur dans le terminal.

Notons que le but de notre solveur n'est pas d'être utilisé pour résoudre les grilles de mots comme dans l'application *Sutom*, car le choix revient à l'utilisateur de mettre le mot qu'il souhaite. Il a plutôt comme but de deviner des mots choisis par l'utilisateur dans un dictionnaire chargé dans des fichiers textes, changeables en respectant l'ordre lexicographique et en mettant le nombre de mots en première ligne de fichier. Ici nous avons déjà généré en amont les parties possibles avec les meilleurs mots à chaque fois. Donc si jamais il y avait cette possibilité de changer le mot proposé, toute l'arborescence se révélerait fausse.

Chapitre 3

Implémentation et spécificités du Solveur

3.1 Implémentation

3.1.1 Structure d'arbre préfixe pour stocker les mots

Comme dit précédemment nous avons créé une structure d'arbre préfixe nous permettant de stocker les mots d'une manière à reconnaître les mots de même préfixe.

```
1 struct _element_t{
2
3     char value;
4     struct _element_t* pere;
5     struct _element_t *fils [26];
6     bool* char_is_in;
7     bool terminal;
8 };
9
10 struct _arbre_t {
11     int nbr_mots;
12     element_t* racine;
13 };
```

FIGURE 3.1 – Structure d'arbre préfixe

Nous avons donc séparé les structures d'arbres et de noeuds en nous servant :

- de la valeur du noeud *val* : c'est le caractère stocké au noeud courant
- du père du noeud, nous y reviendront lors de l'explication du calcul des patterns des mots
- du tableau des fils, 26 pour les 26 lettres de l'alphabet (à noter que beaucoup de parties seront vides du fait que toutes les combinaisons de lettres ne forment pas de mots de la langue française)
- *char is in* qui est un tableau de *nbr mots* éléments permettant de savoir si le caractère courant du noeud est dans tel ou tel mot (les mots sont supposés numérotés selon l'ordre lexicographique)
- de *terminal* permettant de savoir si le caractère courant marque la fin d'un mot
- du nombre de mots dans l'arbre afin de borner nos boucles sur les mots et d'instancier les tableaux *char is in*

Nous avons stocké dans des arbres les mots de même longueur pour essayer au maximum de diviser le problème. Nous nous sommes servis de cette structure pour nous permettre de calculer plus facilement la séquence (convertie de la base 3 en base 10). En effet, en se plaçant dans l'arbre nous avons la possibilité de ne pas calculer une partie de la séquence, notamment dans le cas où deux mots partagent un même préfixe. C'est cette possibilité que nous voulions exploiter pour augmenter les performances en temps du pré-traitement, comme nous réduisons le nombre de comparaisons caractère par caractère.

Nous sommes partis du principe dans la suite, que les mots étaient attribués d'un numéro selon l'ordre lexicographique, ce qui nous a permis d'implémenter notre algorithme de parcours qui se base sur l'essence même d'un arbre préfixe, c'est à dire un arbre qui permet de classer les mots selon leur plus grand préfixe commun.

3.1.2 Parcours de l'arbre et calcul des paternes des mots

Il nous fallait ensuite un moyen efficace de parcourir cet arbre en calculant les patterns associés à chaque couple de mot. Pour cela, nous avons mis au point la fonction "mot suivant" (ci-dessous) qui permet de trouver tous les mots de l'arbre grâce à la méthode du backtracking et qui appelle la fonction "parcours" à chaque mot trouvé.

```
1
2 void mot_suivant(arbre_t *arbre, element_t * elem, char *prefixe, int len_mot, int *
  num_mot_cherche, int **matrice, int nb_mot){
3     if (elem->terminal){ // cas de base, si on est dans une feuille
4         int *l = parcours(arbre, len_mot, num_mot_cherche, matrice, elem, prefixe, nb_mot
  ); //on parcourt tous les autres mots qui sont à droite
5         int int_c = num_mot_cherche[0];
6         for (int k=0; k<nb_mot; k++){
7             matrice[int_c][k] = l[k];
8         }
9         free(l); //On libère l'espace mémoire
10        num_mot_cherche[0] = num_mot_cherche[0] + 1;
11    }
12    for (int i=0; i<26 ; i++){ //pour tous les fils
13        if ((elem->fils)[i] != NULL){ //si le fils est non-vide
14            char new_prefixe[strlen(prefixe)+2]; // on commence à écrire le
  mot
15            if (elem->value == '\0'){
16                memcpy(new_prefixe, prefixe, strlen(prefixe));
17                new_prefixe[0] = elem->fils[i]->value;
18                new_prefixe[1] = '\0';
19            }
20            else{
21                memcpy(new_prefixe, prefixe, strlen(prefixe)); //copie pour
  ajouter la valeur du noeud courant
22                new_prefixe[strlen(prefixe)] = elem->fils[i]->value; //ajout
  de la valeur du noeud courant
23                new_prefixe[strlen(prefixe)+1] = '\0';
24            }
25            mot_suivant(arbre, elem->fils[i], new_prefixe, len_mot, num_mot_cherche,
  matrice, nb_mot); //on descend dans l'arbre pour trouver le suite du mot
  jusqu'à la feuille
26        }
27    }
28 }
```

FIGURE 3.2 – Programme Mot suivant

Une fois le mot suivant trouvé, la fonction `parcours` (ci-dessous) est appelée pour trouver les patterns de tous les autres mots par rapport à ce dernier. Ces patterns sont sous la forme d'une suite de 0 de 1 et de 2 où le k^{ieme} élément est un 2 si la k^{ieme} lettre est la bonne, un 1 si elle est dans le mot mais mal placée et un 0 si elle n'est pas dans le mot. Nous avons ensuite converti cette suite de 0 et de 1 en base 10 pour plus de simplicité dans les calculs et dans le stockage. Enfin, nous avons stocké cette valeur dans la variable `score`.

```

1  int *parcours(arbre_t *arbre, int len_mot, int *num_mot_cherche, int **matrice,
    element_t *position, char *mot_cherche, int nb_mot){
2      element_t *tmp = position;          //copie de la position
3      int *l = calloc(nb_mot, sizeof(int));
4      for (int i=0 ; i<nb_mot; i++){      //pour tous les mots à droite
5          if (i != num_mot_cherche[0]){
6              bool flag = true;
7              char *suffixe_mot_cherche = calloc(strlen(mot_cherche)+1, sizeof(char));
            //initialisation du suffixe du mot cherche
8              char *suffixe_mot_donne = calloc(strlen(mot_cherche)+1, sizeof(char));
            //initialisation du suffixe du mot donne
9              int profondeur = len_mot;
10             tmp = position;          //on réinitialise la position
11             int score;
12             score = (int) pow(3.0, (double) len_mot) - 1;          //initialisation du
score (= paterne en base 10)
13             while (profondeur > 0 && !(tmp->char_is_in)[i]){          //tant que la
lettre du noeud courant n'est pas dans le mot que l'on veut (mot_donne)
14                 score -= 2*(int)pow(3.0, (double) len_mot-profondeur);          //vu
que la lettre n'est pas dans le mot on enlève la valeur associé du score (repré
senté par un 2 dans le pattern en bas 3)
15                 if (flag == true){
16                     flag = false;
17                     suffixe_mot_cherche[0] = tmp->value;
18                     suffixe_mot_cherche[1] = '\\0';
19                 } else{
20                     char *interm = cat_d(tmp->value, suffixe_mot_cherche);          //on
ajoute la valeur au suffixe (Attention, reconstitution du suffixe à l'envers)
21                     for (int k=0; k<strlen(interim); k++){
22                         suffixe_mot_cherche[k] = interim[k];
23                     }
24                     free(interim);
25                 }
26                 profondeur--;          //on met à jour la profondeur
27                 tmp = tmp->pere;          //on remonte au noeud d'au dessus
28             }
29             flag = true;
30             while (tmp != NULL && !(tmp->terminal)){          //une fois arrivé à une
lettre dans le prochain mot, on commence la descente
31                 for (int k=0; k<26 ; k++){          //pour tous les fils
32                     if ((tmp->fils)[k] != NULL && ((tmp->fils)[k]->char_is_in)[i]){
            //si la lettre est dans le prochain mot
33                         tmp = (tmp->fils)[k];          //on se deplace dans ce fils
34                         if (flag == true){
35                             flag = false;
36                             suffixe_mot_donne[0] = tmp->value;
37                             suffixe_mot_donne[1] = '\\0';
38                         } else{
39                             char *interm = cat_f(suffixe_mot_donne, tmp->value);
            //on ajoute la valeur au suffixe (Attention, reconstitution du suffixe à l'
envers)
40                             for (int k=0; k<strlen(interim); k++){
41                                 suffixe_mot_donne[k] = interim[k];
42                             }
43                             free(interim);

```

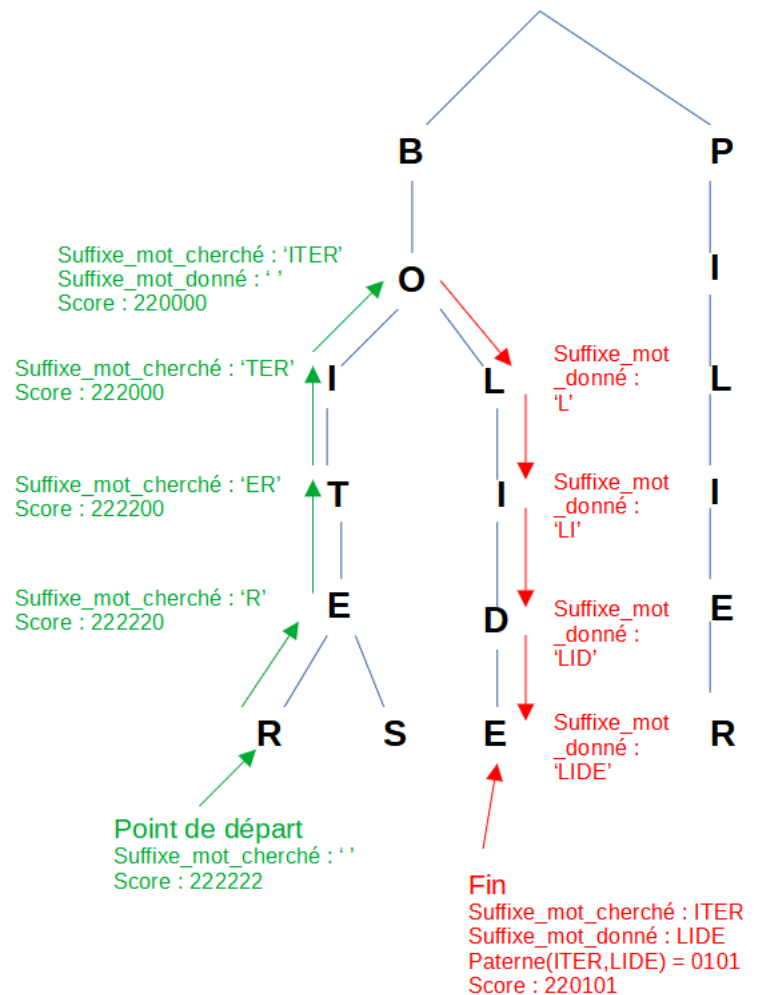
```

1         }
2         break;
3     }
4 }
5
6     score += paterne(suffixe_mot_cherche, suffixe_mot_donne, len_mot);
7     // on compare les suffixe des 2 mots pour avoir le score du pattern et on met
8     // à jour le score
9     l[i] = score;
10    free(suffixe_mot_cherche);
11    free(suffixe_mot_donne);
12 }
13 else {
14     l[i] = (int) pow(3.0, (double) len_mot) - 1;
15 }
16 }
17 return l;
18 }

```

FIGURE 3.3 – Fonction parcours

Son fonctionnement s'appuie sur la structure d'arbre préfixe pour faire le moins de calculs possible au niveau de la séquence. Pour simplifier la compréhension, nous allons appeler le mot par rapport auquel nous calculons les paternes "mot cherché" et les autres mots seront les "mots donnés". Lorsque que cette fonction est appelée, la variable position nous permet de nous positionner dans la feuille du mot cherché, dont la valeur est donc la dernière lettre de ce mot (voir point de départ dans le schéma ci-contre). Là nous initialisons la valeur du score à celle du score maximal. Ensuite, nous allons remonter dans l'arbre jusqu'au premier noeud en commun entre le mot cherché et le mot donné. Cela signifie donc que toutes les lettres au dessus du noeud courant forme un préfixe commun au mot cherché et au mot donné.



Mot cherché : BOITER, Mot donné : BOLIDE

Cette remontée est possible grâce au tableau "char is in" qui nous permet de savoir si la valeur du noeud courant est dans le mot donné. En remontant, nous allons ajouter les valeur de chaque noeud parcouru dans la variable "suffixe mot cherché". De plus, nous allons décrémenter la valeur du score en fonction de la profondeur à laquelle nous nous trouvons. Là, nous allons descendre dans l'arbre afin de former le mot donné. Chaque valeur des noeuds dans lesquels nous allons descendre est ajouté à la variable "suffixe mot donné". Une fois dans la feuille, nous allons appliquer la fonction pattern (ci-dessous) sur les variables "suffixe mot cherché" et "suffixe mot donné".

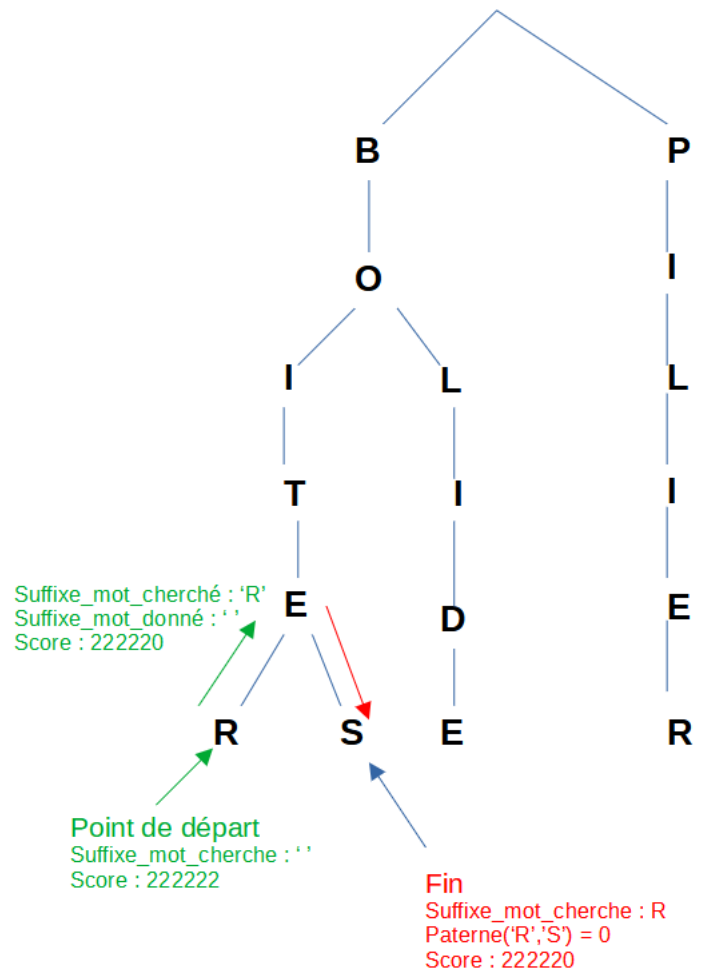
```

1  int pattern(char *mot_cherche, char *mot_donne, int len_mot){
2      int x = 0;
3      int n;
4      if (strlen(mot_cherche) > len_mot){
5          n = len_mot;
6      }
7      else{
8          n = strlen(mot_cherche);
9      }
10     int tab_c[n];
11     int tab_d[n];
12     for (int k=0;k<n;k++){
13         tab_c[k] = 0;
14         tab_d[k] = 0;
15     }
16     for (int i=0 ; i<n ; i++){
17         if (mot_cherche[i] == mot_donne[i]){
18             x += (int) 2*pow(3.0,(double) (n-1-i));
19             tab_c[i] = 1;
20             tab_d[i] = 1;
21         }
22     }
23     for (int i=0 ; i<n ; i++){
24         if (is_in(mot_donne[i], mot_cherche, tab_c) && tab_d[i] != 1){
25             tab_c[index(mot_cherche, mot_donne[i])] = 1;
26             x += (int) pow(3.0,(double) (n-1-i));
27         }
28     }
29     return x;
30 }
31 }
```

FIGURE 3.4 – Fonction pattern

Cette fonction nous renvoie le pattern entre le suffixe mot cherché et le suffixe mot donné que nous allons ajouter à notre score. Ce score est ensuite ajouté à une liste où l'indice représente le numéro du mot donné. Cette liste est ensuite renvoyée par la fonction parcours et est ajoutée à la ligne d'indice le numéro du mot cherché de la matrice où sont stockés tous les patrons.

Cette étape est ensuite appliquée pour tous les couples de mots de la liste. On peut voir sur le schéma ci-contre que dans certains cas, ce parcours évite beaucoup de calculs.



Mot cherché : BOITER, Mot donné : BOITES

3.1.3 Stockage des informations

Mais, pour faire le lien avec le solveur, nous avons eu besoin de stocker les résultats de nos fonctions, pour ensuite les réutiliser plus tard. Nous avons donc choisi de « stocker » les arbres de patterns dans des fichiers texte. Voici leur implémentation :

```
1 struct node {
2     struct node** fils; //[nombre_fils]
3     int nombre_fils;
4     char* mot;
5     int pattern;
6 };
7 struct arbre_pat {
8     int len_mots;
9     node* root;
10 };
```

FIGURE 3.5 – Structure de l’arbre des patterns

On stockait donc a chaque noeud le nombre de fils qu’il avait, pour allouer dynamiquement un tableau de fils de la taille correspondante, ainsi que le mot du noeud et le pattern qui lui était attribué (obtenu via la fonction de calcul expliquée dans la section précédente)

Mais, la simplicité du format ne laissait pas, de manière évidente, apparaître la structure d’arbre qui en découlait. C’est pour cela que nous avons opté pour une convention d’écriture nous permettant de distinguer les fils d’un même père, et les fils de deux pères distincts. Nous avons choisi « , » pour distinguer les fils d’un même père et « ; » pour les fils de pères distincts et nous avons considéré que une ligne dans le fichier représentait l’ensemble des fils à la même profondeur. Ceci nous paraissait suffisant pour décrire un arbre dans un fichier texte, mais des difficultés sous-jacentes sont apparues, notamment, comment régler le cas du fils *NULL*? Nous avons décidé de distinguer le fils *NULL* par le caractère vide.

Ainsi, nous avons pu écrire notre fonction d’écriture dans un fichier selon la ligne à laquelle nous voulions écrire :

```

1 void write_ligne_rec(FILE* file , int ligne , node* current , int profondeur , node*
  pere , int nb_fils_pere , int indice_boucle){
2   if (profondeur==ligne){
3     if (current!=NULL){
4       fprintf(file , "%d" , current->pattern);
5       fprintf(file , " ");
6       fprintf(file , "%s" , current->mot);
7       fprintf(file , " ");
8       fprintf(file , "%d" , current->nombre_fils);
9     }
10    if (indice_boucle==nb_fils_pere-1){
11      fprintf(file , ";");
12    }
13    else {
14      fprintf(file , ",");
15    }
16  }
17  else if (current!=NULL){
18
19    for (int k=0;k<current->nombre_fils;k++){
20      write_ligne_rec( file , ligne , current->fils[k] , profondeur+1 , current ,
21      current->nombre_fils , k);
22    }
23  }

```

FIGURE 3.6 – Code

Et nous appelions cette fonction autant de fois que nécessaire pour écrire toutes les lignes du fichier, c'est à dire jusqu'à la profondeur de l'arbre donné en paramètre.

Mais, nous devons remplir ces arbres afin de les « écrire » dans des fichiers texte, chose que nous avons faite avec l'algorithme suivant :

```

1  node* remplissage_arbre_rec(arbre_t* prev_mots, int** matrice_1, int len_mots, char*
    start_mot, int prev_pattern){ //matrice_1 est la matrice des patterns associée à
    //prev_mots
2      node* current=NULL;
3      //Calcul du nombre de fils;
4      arbre_t* temp;
5      int nb_mots;
6      int num_motd=get_num_mot(prev_mots, start_mot);
7      int nombre_fils=0;
8      int current_pattern;
9      int tab_check[prev_mots->nbr_mots];
10     //On initialise à 0 le tableau
11     for (int i=0; i<prev_mots->nbr_mots; i++){
12         tab_check[i]=0;
13     }
14     for (int i=0; i<prev_mots->nbr_mots; i++){
15         current_pattern=matrice_1[i][num_motd];
16         for (int k=i+1; k<prev_mots->nbr_mots; k++){
17             if (tab_check[k]!=1 && matrice_1[k][num_motd]==current_pattern){
18                 tab_check[k]=1;
19             }
20         }
21         if (tab_check[i]!=1){ //On met à jour les patterns déjà rencontrés
22             nombre_fils++;
23             tab_check[i]=1;
24         }
25     } //On insère le mot courant
26     current=cree_node(nombre_fils, prev_pattern, start_mot, len_mots);
27     //Si le nombre de mots dans l'arbre vaut 1 on s'arrête en retournant le noeud
    créé
28     //Sinon on calcule les matrices des meilleurs fils et on insère dans l'arbre //
    récursivement
29     if (prev_mots->nbr_mots>1){
30         int *num_mot_cherche = calloc(1, sizeof(int));
31         int** matrice_2;
32         int fils=0;
33         for (int i=0; i<prev_mots->nbr_mots; i++){
34             tab_check[i]=0;
35         };
36         for (int i=0; i<prev_mots->nbr_mots; i++){
37             current_pattern=matrice_1[i][num_motd];
38             //Les mots donnés sont stockés en colonne
39             for (int k=i+1; k<prev_mots->nbr_mots; k++){
40                 if (tab_check[k]!=1 && matrice_1[k][num_motd]==current_pattern){
41                     tab_check[k]=1;
42                 }
43             }
44             if (tab_check[i]!=1){ //On prend tous les mots de même pattern, en
    faisant //attention à ne pas prendre deux fois le même.
45                 char* temp_best_mot=calloc((len_mots+1), sizeof(char));
46                 num_mot_cherche[0]=0;
47                 nb_mots=same_pattern(prev_mots, matrice_1, current_pattern, num_motd);
48                 temp=create_arbre_mots(nb_mots);

```

```

1      //Fonction qui prend les mots de même pattern pour les insérer dans
      un //arbre
2      insert_same_pattern(prev_mots,temp,matrice_1,current_pattern,
num_motd);
3      //Création de la matrice pour les fils de même pattern
4      matrice_2=calloc(temp->nbr_mots,sizeof(int*));
5      for (int i=0;i<temp->nbr_mots;i++){
6          matrice_2[i]=calloc(temp->nbr_mots,sizeof(int));
7      }
8      //Remplissage de la matrice
9      mot_suivant(temp,temp->racine,"",len_mots,num_mot_cherche,matrice_2
,temp->nbr_mots);
10     //Calcul du meilleur mot de la matrice
11     strcpy(temp_best_mot,best_mot(temp,matrice_2,len_mots));
12     temp_best_mot[len_mots]='\0';
13     //Appel récursif
14     current->fils[fils]=remplissage_arbre_rec(temp,matrice_2,len_mots,
temp_best_mot,current_pattern);
15     //Libération de la mémoire
16     for (int i=0;i<temp->nbr_mots;i++){
17         free(matrice_2[i]);
18     }
19     free(matrice_2);
20     destroy_arbre(temp);
21     tab_check[i]=1;
22     fils++;
23     free(temp_best_mot);
24 }
25 }
26 free(num_mot_cherche);
27 } else {
28     return current;
29 }
30 return current;
31 }

```

FIGURE 3.7 – Fonction qui permet le remplissage de l’arbre des patterns

Notons que nous calculons à chaque fois le meilleur mot comme résultant d’un arbre contenant de moins en moins de mots à chaque appel récursif.

Nous avons ainsi écrit des fichiers pour chaque longueur de mot, en appliquant les algorithmes expliqués dans cette partie.

3.2 Spécificités et interaction

Après avoir pré-traité notre dictionnaire de mots, c'est là que le programme principal peut être utilisé. Le pré-traitement créant un fichier texte par longueur de mots, il faut maintenant accéder à ces fichiers dans le solveur et les utiliser afin de répondre au motif fourni par l'utilisateur. Nous avons divisé ce programme principal en plusieurs sous-programmes :

- On lit le fichier wself ou on ajoute un paramètre dans le terminal au lancement de l'application, afin de pouvoir récupérer n , la longueur du mot proposé.
- Après ça, on lit le fichier contenant l'arbre des mots de longueur n , afin d'en initialiser un arbre normal, contenant à ses noeuds pattern et meilleur mot suivant associé.
- Enfin, on affiche la racine de l'arbre, soit le meilleur mot à entrer pour le dictionnaire actuel.
- On boucle ensuite entre demande du pattern suivant de l'utilisateur et proposition du meilleur mot suivant, jusqu'à trouver le mot recherché.
- On libère l'espace mémoire de l'arbre.

3.2.1 Arbres motif - meilleur mot

Comme dit précédemment, notre programme commence par son lancement en terminal avec ou sans l'addition d'un paramètre n compris entre 6 et 10. Si un paramètre a été ajouté, le programme fera en sorte d'ouvrir le fichier wself afin de l'y inscrire. Sinon, le programme ouvrira wself afin d'y tenter de lire le paramètre n . Bien évidemment toute erreur de paramètre résultera en une sortie d'application. Ensuite on crée un arbre à partir de n ouvrira le fichier txt contenant l'arbre des meilleurs mots de taille n selon les motifs de l'utilisateur. La racine de cet arbre contiendra uniquement le meilleur mot de début de partie, tandis que les branches contiendront un motif (passé de base 10 à base 3), un nombre de fils et un mot associé à rentrer afin de continuer la partie.

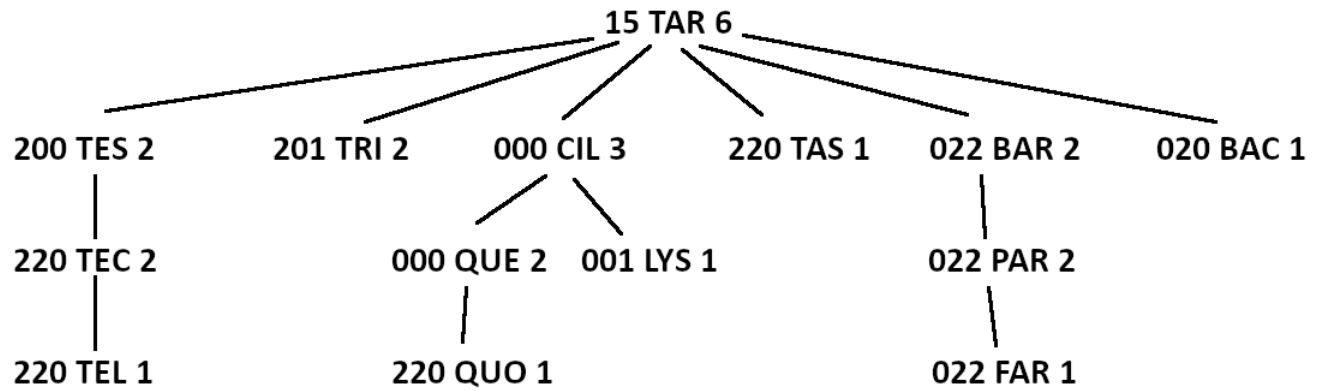


FIGURE 3.8 – Exemple d’arbre de parcours de pattern

La difficulté ici résidait dans la conversion des .txt en arbre, car nous avons décidé de compléter nos fichiers par un parcours en largeur, impliquant donc de lire cet arbre en largeur et en C. Nous avons donc dû lire les fichiers récursivement, causant de nombreuses difficultés qui ont su être évitées ou traversées.

```

1   for (int i = 0; i < line_size; i++)
2   {
3       strncpy(car, line, 1);
4       car[1] = '\0'; // afin de stocker le caractère lu actuellement
5
6       if (nbrpvir2 == nbrpvir) { // cas où dans la section des fils
7           if (strcmp(car, ",") == 0) {
8               if (strcmp(mot, "") != 0) { // vérif mot non vide
9                   sscanf(mot, "%d %s %d", &pattern, val, &nbrfils);
10                  noeud_t *newnoeud = create_noeud(nbrfils, pattern, val);
11                  noeud_pere->fils[nbrvir] = newnoeud;
12                  memset(mot, 0, 1);
13
14                  lectfils(newnoeud, fptr, pvir, pos2); // appel récursif
15                  fseek(fptr, pos2, SEEK_SET); // on se repositionne dans le
fichier parce que lectfils déplace indirectement
16                  nbrvir++;
17                  pvir++;
18              }
19          }
20          if (strcmp(car, ";") == 0) {
21              // vérif cas où mot fini par ; à ajouter et vérif non vide
22              if (strcmp(mot, "") != 0) {
23                  sscanf(mot, "%d %s %d", &pattern, val, &nbrfils);
24                  noeud_t *newnoeud = create_noeud(nbrfils, pattern, val);
25                  noeud_pere->fils[nbrvir] = newnoeud;
26                  memset(mot, 0, 1);
27                  lectfils(newnoeud, fptr, pvir, pos2);
28                  fseek(fptr, pos2, SEEK_SET); // on se repositionne dans le
fichier parce que lectfils déplace indirectement
29                  pvir++; // noeud non vide lu
30              }
31              nbrpvir2++;
32
33          }
34      } // cas hors section, on vérifie si on lit des noeuds
35      else if (strcmp(car, ",") == 0 || strcmp(car, ";") == 0) {
36          if (strcmp(mot, "") != 0) { pvir++; }
37          if (strcmp(car, ";") == 0) { nbrpvir2++; }
38          memset(mot, 0, 1);
39      } // puis si non, ou ; on ajoute au mot
40      if (strcmp(car, ";") != 0 && strcmp(car, ",") != 0) {
41          strcat(mot, car, 1);
42      }
43      line++; // avance dans la lecture de la ligne
44  }

```

FIGURE 3.9 – Boucle de lecture des fichiers .txt

Un problème qui aura été intéressant dans notre implémentation est le cas suivant :

```
1    ... //supposons à la fin du fichier , les deux lignes n et n+1 suivantes
2    ;;;QUE;;;FAR;
3    QUO;SAR; //fin du fichier
```

FIGURE 3.10 – Exemple de soucis d’implémentation 1

Ici, nous souhaitons :

- Avoir QUO mot fils de QUE
- Avoir SAR mot fils de FAR

Cependant, en utilisant la méthode virgule/point-virgule, nous voyons que les fils de QUE sont après 3 points virgules et que ceux de FAR sont après 8 point-virgules. Le problème est donc clairement évident dans un cas comme celui-ci, qui dans un dictionnaire de milliers de mots, est très présent.

La méthode initiale était de compter le nombre de point-virgules et de virgules afin de savoir à quelle section de la ligne suivante il fallait récupérer les fils du mot. En revanche si le mot n’a qu’un seul fils c’est à dire lui même, cela causait donc à la ligne suivante un double point-virgule, signifiant que le mot n’a pas d’autres fils. Mais ce qui posait réellement problème est la conséquence de ce double point-virgule, qui implique qu’à la ligne suivante ce double point-virgule disparaît, et ainsi cette méthode initiale ne coïncidait pas avec la position des fils. La lecture des caractères uniquement dans la section ne permet pas de vérifier la présence de noeuds vides puisque dans les deux cas aucun mot n’était récupéré. Ainsi la deuxième méthode a été de concaténer à chaque fois le caractère lu au mot, et de faire les mêmes vérifications que dans la section afin de remettre à vide le mot et nous avons pu vérifier les noeuds vides et donc pu compter le nombre de noeuds non vides, et donc pu savoir le nombre de point-virgules à traverser à la ligne suivante.

```
1    ... //supposons à la fin du fichier
2    ;;;QUE;;;FAR; //nous lisons aucun noeud non vide avant QUE et un avant FAR
3    QUO;SAR; //fin du fichier
```

FIGURE 3.11 – Exemple de l’implémentation 2

3.2.2 Interaction utilisateur

Une fois notre arbre créé, l'utilisateur reçoit son premier mot à rentrer dans sa partie de wordle. Ce premier mot a été calculé dans le pré traitement afin d'être le premier mot apportant le plus d'information sur le mot recherché. Une fois que l'utilisateur a pris connaissance du mot, il peut le rentrer dans sa partie, tandis que le programme demandera le motif résultant du mot proposé. A ce stade là, l'utilisateur peut rentrer soit 1) le motif résultant, 2) quit pour quitter l'application, 3) le motif résultant suivi de -i afin de voir des statistiques associées au déroulement de sa partie. Ceci continue tant que l'utilisateur n'a pas trouvé le mot. Par construction, le solveur trouvera toujours le mot tant qu'on ne modifie pas les valeurs qu'il est sensé recevoir, ainsi, si l'utilisateur rentre un motif qui ne fonctionne pas, le programme lui renverra une erreur et lui demandera de rentrer à nouveau le motif (il est possible de quitter si l'utilisateur le veut).

Pour l'ajout des statistiques du motif, il a suffi de sachant le nombre de mot total, récupérer le nombre de noeud dans le sous-arbre dans lequel nous sommes, permettant de pouvoir comparer les deux. La première méthode était de renvoyer un pourcentage, mais le nombre de branche étant tellement grand, le pourcentage était inférieur à 1 ce qui n'était pas intéressant pour l'utilisateur. Nous avons donc opté pour un $X \text{ mots} / N \text{ restants}$ afin de pouvoir effectivement remarquer l'efficacité des informations apportées par les mots choisis.

```
eleve@TNCY-Linux: ~/Documents/PPII2/project2...
eleve@TNCY-Linux:~/Documents/PPII2/project2-E2/Solveur$ ./solveur 6
Longueur 6, meilleur premier mot: CASIER

Entrez le motif du résultat du mot :
(2 = bien placé ; 1 = dans le mot ; 0 absent du mot ; ex 01211)
(ajoutez -i après le motif pour avoir plus d'informations)
121020 -i
Avec ce mot, il reste 12 / 2231 mots dans le dictionnaire !

Le mot à rentrer est: TACLES
Entrez le motif suivant :
022022

Le mot à rentrer est: FACHES
Entrez le motif suivant :
022222 -i
Avec ce mot, il reste 4 / 2231 mots dans le dictionnaire !

Le mot à rentrer est: GACHES
Entrez le motif suivant :
2222222
le motif ne correspond pas à un mot de cette longueur.
Entrez le motif suivant :
-1
-1
Arrêt du programme...
eleve@TNCY-Linux:~/Documents/PPII2/project2-E2/Solveur$
```

FIGURE 3.12 – Interactions possibles avec l'utilisateur

Chapitre 4

Tests et performances du solveur

4.1 Tests - Méthode du Right-BICEP

Pour tous les tests que nous avons effectués, nous avons utilisé la librairie *Snow*, qui se situe dans le projet. Les tests sont exécutables grâce à la commande `make tests`.

4.1.1 Fonction *paterne*

Voici le code utilisé pour vérifier les principes du Right - BICEP (tout n'a pas pu être testé)

```

1  #include<snow/snow.h>
2  describe(pattern){
3      it("Test de pattern : égalité de mot"){
4          char* mot1="BONJOUR";
5          char* mot2="BONJOUR";
6          int len=7;
7          assert(paterne(mot1,mot2,len)==(int)pow(3.0,len)-1,"Vérification de l'é
galité de mot");
8      }
9      it("Test de pattern : vérification valeur"){
10         char* mot1="BONJOUR";
11         char* mot2="HELLOOO";
12         int len=7;
13         assert(paterne(mot1,mot2,len)==21,"Vérification de valeur");
14         assert(paterne(mot2,mot1,len)==18+(int)pow(3.0,5.0),"Vérification de valeur
");
15     }
16     it("Test des limites : B "){
17         char* mot1="BONJOUR";
18         char* mot2="BONJOU";
19         int len=6;
20         int a=paterne(mot1,mot2,len);
21         assert(a==(int)pow(3.0,len)-1);
22         len=7;
23         int b=paterne(mot1,mot2,len);
24         assert(b==(int)pow(3.0,7)-3,"Vérification de limite de mots n'étant pas de
même longueur");
25         mot2="BO";
26         int c=paterne(mot1,mot2,len);
27         assert(c==2*(int)pow(3.0,6)+2*pow(3.0,5),"Vérification des limites");
28         //paterne attendu : 2200000
29         //On renvoie le paterne avec les lettres présentes dans le mot
30         int d=paterne(mot2,mot1,len);
31         assert(d==8,"Vérification de limite de mots n'étant pas de même longueur");
32         //paterne attendu : 0000022 <-> 8
33         mot2="";
34         len=0;
35         int e=paterne(mot1,mot2,len);
36         assert(e==0,"Vérification du mot vide");
37         int f=paterne(mot2,mot1,len);
38         assert(f==0,"Vérification du mot vide");
39     }
40     //On ne peut pas trouver un mot spécifique avec un paterne donné, même en
fixant le mot cherché, il peut avoir plusieurs mots qui correspondent
41     //Mais on peut au moins vérifier qu'il appartient à la classe des mots de ce
paterne donné
42

```

```

1  it("Test inverse : I"){
2      arbre_t* arbre=construct_arbre(6);
3      int** matrice=calloc(arbre->nbr_mots, sizeof(int*));
4      for (int i=0;i<arbre->nbr_mots;i++){
5          matrice[i]=calloc(arbre->nbr_mots, sizeof(int));
6      }
7      int *num_mot_cherche = malloc(sizeof(int));
8      num_mot_cherche[0] = 0;
9      mot_suivant(arbre, arbre->racine, "", 6, num_mot_cherche, matrice, arbre->
nbr_mots);
10     int d=paterne("ABATTU", "CASIER", 6); // L'arbre des paternes est fait de
telle sorte que le paterne d'un noeud correspond à
11     //paterne(mot_du_noeud, mot_du_père)
12     int nb_mots=same_pattern(arbre, matrice, d, get_num_mot(arbre, "CASIER"));
13     arbre_t* temp=create_arbre_mots(nb_mots);
14     insert_same_pattern(arbre, temp, matrice, d, get_num_mot(arbre, "CASIER"));
15     print_arbre(temp);
16     bool t=insert_arbre(temp, "ABATTU", 0);
17     //On vérifie que le mot est déjà dans l'arbre
18     assert(!t, "Vérification inverse");
19     //Libération de la mémoire
20     for (int i=0;i<arbre->nbr_mots;i++){
21         free(matrice[i]);
22     }
23     free(matrice);
24 }
25 //Nous n'avons pas trouvé de façon de cross-check
26 //Pour forcer l'erreur, les paramètres n'étant pas du bon type, le programme ne
compile pas
27 //Niveau performances, le programme met beaucoup de temps avec "peu" de mots
(6000), nous forçant à rester sur notre dictionnaire actuel
28 }

```

FIGURE 4.1 – Tests Right-BICEP paterne

Avec ce code, en compilant et en exécutant nous avons eu les résultats suivants :

```

Testing pattern:
✓ Success: Test de pattern : égalité de mot (10.99µs)
✓ Success: Test de pattern : vérification valeur (2.93µs)
✓ Success: Test des limites : B (4.15µs)
? Testing: Test inverse : I: ./Mots/mot6.txt
Mots bien ajoutés dans l'arbre
MOT : ABATTU
MOT : ANOMAL
MOT : APLOMB
MOT : BOYAUX
MOT : FUMANT
MOT : GLOBAL
MOT : JOYAUX
MOT : LOYAUX
MOT : TOTAUX
MOT : TUYAUX
MOT : TYMPAN
MOT : VOLANT
MOT : VOYANT
✓ Success: Test inverse : I (12.62s)
pattern: Passed 4/4 tests. (12.62s)

```

FIGURE 4.2 – Résultat des tests

Nous pouvons remarquer que la fonction *mot_suivant* possède des problèmes de performance, ce que nous avons explicité plus haut.

4.1.2 Fonction *mot_suivant*

Nous avons également pu tester la fonction *mot_suivant* avec le code ci-dessus :

```

1  describe(motsuivant){
2      //On fera majoritairement des tests sur des mots de longueur 3
3      int lenmot=3;
4      char* mot4="POT";
5      char* mot1="BUT";
6      char* mot2="FUT";
7      char* mot3="MAT";
8      arbre_t* arbre=create_arbre_mots(4);
9      insert_arbre(arbre,mot1,0); //On insère dans l'ordre lexicographique
10     insert_arbre(arbre,mot2,1);
11     insert_arbre(arbre,mot3,2);
12     insert_arbre(arbre,mot4,3);
13     //Création de la matrice
14     int **matrice=calloc(4,sizeof(int*));
15     for (int i=0;i<4;i++){
16         matrice[i]=calloc(4,sizeof(int));
17     }
18     int *num_mot_cherche = malloc(sizeof(int));
19     num_mot_cherche[0] = 0;
20     it("Test : Vérification des résultats , Right"){
21         mot_suivant(arbre,arbre->racine,"",lenmot,num_mot_cherche,matrice,4);
22         for (int i=0;i<4;i++){
23             assert(matrice[i][i]==(int)pow(3.0,lenmot)-1,"Vérification de la
diagonale , mot_c = mot_d");
24         }
25         //Quelques vérifications
26         assert(matrice[1][3]==paterne(mot2,mot4,lenmot),"Vérif");
27         assert(matrice[3][0]==paterne(mot4,mot1,lenmot),"Vérif");
28     }
29     it("Test des limites : nb mot=1 ,len mot=1000"){
30         char promp[1001];
31         memset(promp,'A',1001);
32         promp[1000]='\0';
33         int** matrice2=calloc(1,sizeof(int*));
34         matrice2[0]=calloc(1,sizeof(int));
35         int *num_mot_cherche=calloc(1,sizeof(int));
36         num_mot_cherche[0]=0;
37
38         arbre_t* t1=create_arbre_mots(1);
39         insert_arbre(t1,promp,0);
40         mot_suivant(t1,t1->racine,"",100,num_mot_cherche,matrice2,1);
41     }
42     it("Test des limites B: nb mot = 1, len mot=2000"){
43         char promp2[2001];
44         memset(promp2,'A',2001);
45         promp2[2000]='\0';
46         int** matrice3=calloc(1,sizeof(int*));
47         matrice3[0]=calloc(1,sizeof(int));
48         arbre_t* t2=create_arbre_mots(1);
49         insert_arbre(t2,promp2,0);
50         int *num_mot_cherche=calloc(1,sizeof(int));
51         num_mot_cherche[0]=0;
52         mot_suivant(t2,t2->racine,"",200,num_mot_cherche,matrice3,1);
53     }
54
55

```



```

1  it ("Test des limites B: nb_mots = 2000, len=6"){
2      //On prend les mots du dictionnaire
3      arbre_t* arbre=construct_arbre(6);
4      int** matrice=calloc(arbre->nbr_mots, sizeof(int*));
5      for (int i=0;i<arbre->nbr_mots;i++){
6          matrice[i]=calloc(arbre->nbr_mots, sizeof(int));
7      }
8      int *num_mot_cherche = malloc(sizeof(int));
9      num_mot_cherche[0] = 0;
10     mot_suivant(arbre, arbre->racine, "", 6, num_mot_cherche, matrice, arbre->
11     nbr_mots);
12 }
13 it ("Test des limites B: nb mots=4000, len=7"){
14     //On prend les mots du dictionnaire
15     arbre_t* arbre=construct_arbre(7);
16     int** matrice=calloc(arbre->nbr_mots, sizeof(int*));
17     for (int i=0;i<arbre->nbr_mots;i++){
18         matrice[i]=calloc(arbre->nbr_mots, sizeof(int));
19     }
20     int *num_mot_cherche = malloc(sizeof(int));
21     num_mot_cherche[0] = 0;
22     mot_suivant(arbre, arbre->racine, "", 7, num_mot_cherche, matrice, arbre->
23     nbr_mots);
24 }
25 //Il n'y a pas de relation inverse
26 //Le cross checking a pu être effectué avec l'implémentation en tableau
27 //Les erreurs de types ne permettent pas de compiler, par contre on provoque
    des erreurs de lecture dans la mémoire quand on change les valeurs en paramètre
    de len_mots mais aussi de nombre_mots
28 //Comme montré les performances en temps ne sont pas trop dérangeantes du fait
    du prétraitement, mais si nous avons à effectuer ces calculs à chaque lancement
    du solveur, les temps de calculs seraient vraiment longs
29 }

```

FIGURE 4.3 – Tests de la fonction mot_suivant

Et nous avons obtenu les résultats suivants :

```

Testing motsuivant:
✓ Success: Test : Vérification des résultats, Right (37.11µs)
✓ Success: Test des limites : nb mot=1 ,len mot=1000 (435.79µs)
✓ Success: Test des limites B: nb mot = 1, len mot=2000 (1.08ms)
? Testing: Test des limites B: nb_mots = 2000, len=6: ./Mots/mot6.txt
Mots bien ajoutés dans l'arbre
✓ Success: Test des limites B: nb_mots = 2000, len=6 (11.85s)
? Testing: Test des limites B: nb mots=4000, len=7: ./Mots/mot7.txt
Mots bien ajoutés dans l'arbre
✓ Success: Test des limites B: nb mots=4000, len=7 (58.28s)
motsuivant: Passed 5/5 tests. (70.14s)

```

FIGURE 4.4 – Caption

4.2 Analyse en complexité des fonctions majeures

	Complexité Temporelle de la fonction dans le pire des cas
Fonction <i>paterne</i>	$\Theta(l^2)$
Fonction <i>parcours</i>	$\Theta(n * l)$
Fonction <i>mot_suivant</i>	$\Theta(n^2 * l^2)$
Fonction <i>remplissage_arbre_rec</i>	$\Theta(n)$
Fonction <i>write_ligne_rec</i>	$\Theta((3^l)^{\log(n)})$

TABLE 4.1 – Complexités temporelle dans le pire des cas de nos fonctions majeures

- l la longueur des mots
- n le nombre de mots (de même longueur l)

Pour l'analyse de *remplissage_arbre_rec*, le pire des cas consiste à considérer une distribution de mots, et un nombre de mots tels que l'arbre à générer ait à chaque noeud un nombre de fils de $3^l - 1$, et comme l'arbre des paternes doit reprendre toutes les parties possibles, il doit avoir $2 * n$ noeuds (on affiche un mot au maximum deux fois, une fois lorsque le mot est le mot suivant, une autre fois quand c'est le mot gagnant), ce qui nous donne une complexité en $\Theta(n)$ avec n tel que :

$$\exists i \in \mathbb{N}, n = (3^l - 1)^i$$

Comme nous travaillons avec des mots de la langue française, l ne pourra jamais être plus grand que 27 (plus long mot de la langue française), mais ici nous nous sommes restreints à un dictionnaire de 6 à 10 lettres.

Nous n'avons pas les complexités des fonctions utilisées provenant de *<string.h>*, mais ces fonctions ont eu un impact assez négatif sur la complexité temporelle de nos fonctions.

4.3 Performances et comparaison des algorithmes

4.3.1 Comparaison temps de calcul avec algorithme de parcours dans un arbre et algorithme de parcours dans un tableau pour la partie prétraitement

Même si nous sommes partis sur une implémentation d'un arbre préfixe pour le stockage des mots, nous voulions constater si notre structure était bel et bien plus performante que l'implémen-

tation avec un tableau.

Nous avons testé avec *Snow* le temps d'exécution de notre algorithme muni d'un arbre et avec un algorithme muni d'un tableau avec le code suivant :

```
1 describe(implementTableau){
2     it("Test de performance en temps avec implémentation dans un tableau"){
3         FILE* fptr;
4         int len_mot=7;
5         ///ouverture fichier et lecture pour mettre dans un tableau
6         char link[100];
7         sprintf(link, "./Mots/mot%d.txt", len_mot);
8         puts(link);
9         fptr = fopen(link, "r");
10        if (fptr==NULL){
11            perror ("Error reading file");
12        }
13        char line[256];
14        fgets(line, sizeof(line), fptr);
15        int nb_mot;
16        sscanf(line, "%d",&nb_mot);
17        char* tabmots[nb_mot];
18            if (fptr==NULL)
19        {
20            perror ("Error reading file");
21        }
22        else
23        {
24            for (int k=0 ; k<nb_mot ; k++){
25                char mot[20];
26                fgets(line, sizeof(line), fptr);
27                sscanf(line, "%s", mot);
28                tabmots[k]=mot;
29            }
30        }
31        fclose(fptr);
32        ///le tableau contient tous les mots de longueur 6
33        ///Création de la matrice
34        int **matrice=calloc(nb_mot, sizeof(int*));
35        for (int i=0; i<nb_mot; i++){
36            matrice[i]=calloc(nb_mot, sizeof(int));
37        }
38        char* mot_cherche;
39        char* mot_donne;
40        ///Calcul des patterns
41        for (int i=0; i<nb_mot; i++){
42            mot_cherche=tabmots[i];
43            for (int j=0; j<nb_mot; j++){
44                mot_donne=tabmots[j];
45                matrice[i][j]=paterne(mot_cherche, mot_donne, len_mot);
46            }
47        }
48    }
```

```

1      //Tests
2      for (int i=0;i<nb_mot;i++){
3          assert(matrice[i][i]==(int)pow(3.0,len_mot)-1,"Vérification du
remplissage de la matrice");
4      }
5  }
6  }

```

FIGURE 4.5 – Test pour l’implémentation avec un tableau, mots de longueur 7

Et ensuite, le résultat de ce test :

```

✓ Success: Test des limites B: nb mots=4000, len=7 (58.20s)
motsuivant: Passed 5/5 tests. (70.23s)

Testing implemTableau:
? Testing: Test de performance en temps avec implémentation dans un tableau: ./Mots/mot7.txt
✓ Success: Test de performance en temps avec implémentation dans un tableau (14.37s)
implemTableau: Passed 1/1 tests. (14.37s)

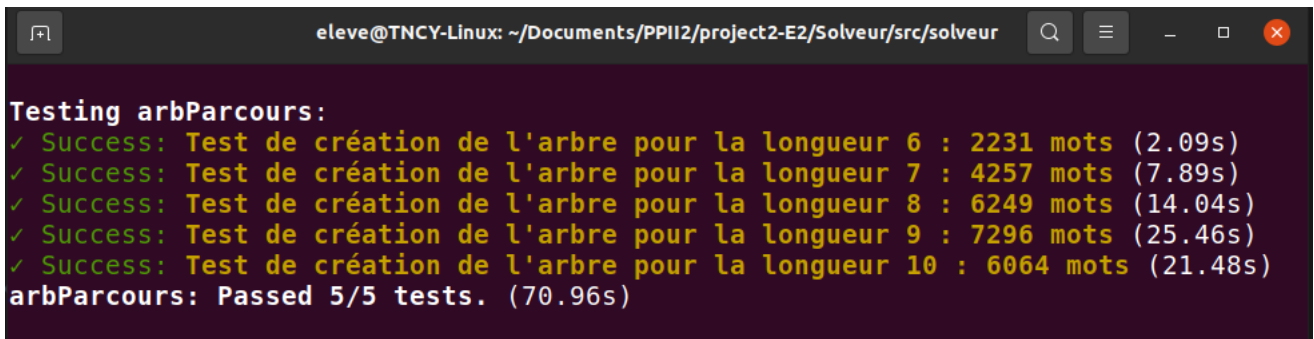
```

FIGURE 4.6 – Résultat du test de performance

Ainsi, comme explicité par ce test, nous n’avons pas réussi à réduire le temps de calcul par rapport à l’implémentation avec un arbre, faute aux nombreuses opérations sur les chaînes de caractères que nous n’avons pas su optimiser.

4.3.2 Temps d’exécution du solveur

Notre programme principal de solveur consiste d’abord à créer un arbre de pattern à parcourir à partir de fichiers txt issus du pré traitement, puis de parcourir cet arbre afin de trouver le mot de l’utilisateur. Ainsi, la deuxième partie dépendant de l’utilisateur car nous parcourons juste un arbre, nous visualiserons plutôt la création de l’arbre, qui est plus longue que son parcours. Nous testerons également sur chaque longueur de mots, mais la quantité de mot impactera davantage les tests que leur longueur.

A terminal window with a dark background and light-colored text. The window title is 'eleve@TNCY-Linux: ~/Documents/PPII2/project2-E2/Solveur/src/solveur'. The output shows a series of test results for 'arbParcours' with lengths 6 through 10. Each line indicates a successful test, the number of words, and the time taken in seconds. The final line shows that all 5 tests passed with a total time of 70.96s.

```
Testing arbParcours:
✓ Success: Test de création de l'arbre pour la longueur 6 : 2231 mots (2.09s)
✓ Success: Test de création de l'arbre pour la longueur 7 : 4257 mots (7.89s)
✓ Success: Test de création de l'arbre pour la longueur 8 : 6249 mots (14.04s)
✓ Success: Test de création de l'arbre pour la longueur 9 : 7296 mots (25.46s)
✓ Success: Test de création de l'arbre pour la longueur 10 : 6064 mots (21.48s)
arbParcours: Passed 5/5 tests. (70.96s)
```

FIGURE 4.7 – Résultat du test de durée des créations d’arbres

On remarque ainsi que tout d’abord le nombre de mots impacte la durée de création des arbres, mais également la longueur des mots, comme on peut voir entre 8 et 10, avec une différence de 7 secondes. En comparant à la durée de pré traitement avoisinant les quelques minutes pour les longueurs 8 9 et 10, on remarque que la création de l’arbre ne prend que peu de temps, bien que dans notre implémentation nous n’avons pas ajouté la fonctionnalité pour recommencer une partie, on peut d’ors et déjà imaginer avec une telle fonctionnalité le gain de temps sur le long terme. Nous pourrions donc diviser ce temps de création d’arbre par le nombre de parties jouées, ainsi rendant quasi négligeable ce temps sur le long terme. De même, nous pourrions implémenter un choix au début du programme, demandant si l’utilisateur souhaite faire plusieurs longueurs différentes ou non, ainsi chargeant un ou tout les arbres afin de ne pas avoir à les recréer et donc assurant de ne pas avoir à les recharger, et permettant sur une grande quantité de partie de quasi nullifier ce temps de création d’arbre. Enfin, c’est là dessus que nous pouvons observer la réussite de notre stratégie, le pré traitement, qui nous permet donc en moyenne un faible temps de calcul sur une grande quantité de parties jouées.

Chapitre 5

Gestion de Projet

5.1 Gestion de Projet : Application Web

5.1.1 Forces et faiblesses de notre projet : matrice SWOT

Après analyse des forces et faiblesses, internes et externes de notre projet, nous en avons déduit notre matrice SWOT :

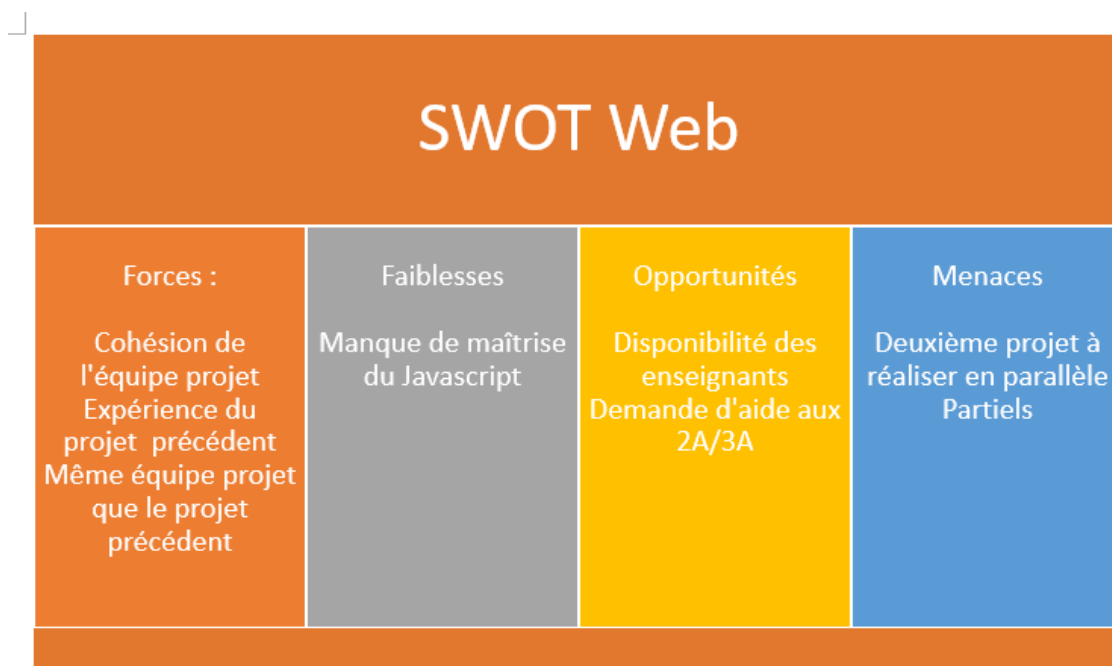


FIGURE 5.1 – Matrice SWOT

Ainsi, pour la gestion de projet, il s'agissait de bien prendre en compte les menaces et les faiblesses inhérentes au projet. En particulier, pour le manque de maîtrise du langage *Javascript*, nous nous sommes organisés afin de suivre des cours en ligne sur certains aspects essentiels du langage (balises HTML vues comme des objets) ce qui nous a permis notamment de faire un parallèle avec nos cours de POO. De plus les partiels ont pu être en partie anticipés en mettant au clair les conceptions des différentes parties (Web, Base de données, Algorithme). Quand bien même ces parties ont été sujettes à des modifications ultérieures (en particulier la base de données), la conception a permis de bien comprendre les besoins de l'application.

Par ailleurs, la possibilité de demander de l'aide aux 2A/3A nous a fortement aidés, en particulier pour les bases de données.

La force de notre projet, c'est à dire la cohésion, a permis au projet d'avancer, sans conflits. Les réunions et la communication, malgré certains problèmes, ont permis de mettre rapidement les choses au clair et de résoudre les problèmes assez rapidement. Nous n'avons pas souvent eu affaire à « l'effet tunnel ». De plus, comme notre équipe est constituée des mêmes membres que notre premier projet, nous avons pu bénéficier de notre connaissance des membres de l'équipe pour accélérer le développement, et améliorer notre gestion du temps.

La véritable question était de gérer le temps afin de commencer le plus tôt possible la deuxième partie du projet : le solveur en C.

5.1.2 Gestion des risques : évaluation de la fréquence et de la gravité des événements

Les faiblesses du projet nous ont permis d'établir le diagramme de gestion des risques, en estimant les fréquences et les gravités :

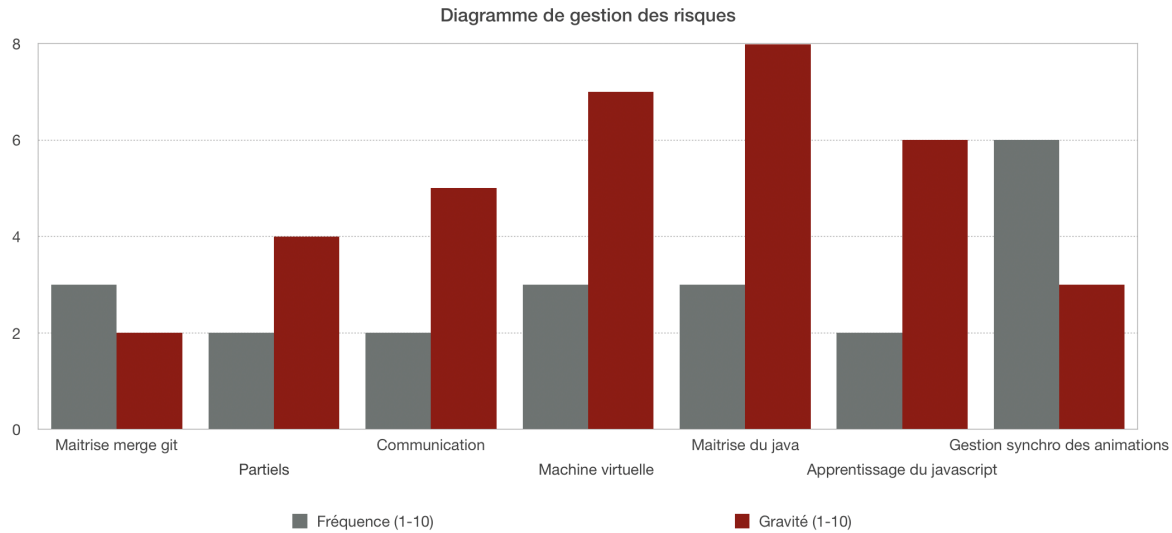


FIGURE 5.2 – Diagramme de fréquence et de gravité pour des risques éventuels

Certains problèmes n'ont pas pu être évités (cf Annexe 5 pour plus de visibilité), comme par exemple les problèmes des types de machines, c'est à dire la machine virtuelle de certains qui ne marchait pas à l'instant t. Mais cette analyse nous a permis, en estimant les fréquences et les gravités, de prendre conscience de ce qui pouvait être amélioré, comme la maîtrise des outils.

5.1.3 Découpage des lots de travail

Suite à cette évaluation, nous avons construit le WBS :

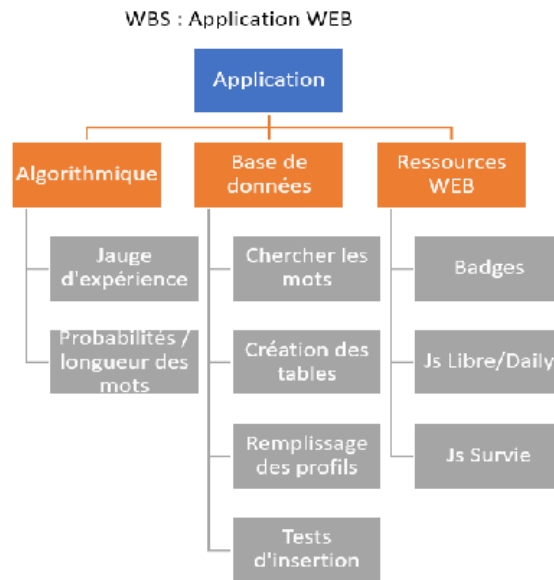


FIGURE 5.3 – Division du projet en lots de travaux

5.1.4 Répartition des responsabilités

Suite à cette division, nous avons établi la répartition des tâches :

	Nathan	Pierre	Julien	Thomas
Expérience des joueurs	I/C	R	R	A
Répartition des probabilités d'apparition	I	C	A	R
Création des tables	C	R	A	I
Recherche des mots	R	C/I	A	R
Remplissage des profils	C	R	I	A
Tests d'insertions	R	R	A	C/I
Template principal (header)	C	I	R	A
Home de l'application	A	C	R	I
Pages d'historique	A	R	C	I
Page de classement	A	C	R	I
Page de survie	R	A	C/I	R/A
Fonctions spécifiques au mode survie	R	A	C/I	R
Fonctions spécifiques au mode libre/daily	C	I	R/A	R
Badges des joueurs	A	C	R	I

TABLE 5.1 – Matrice RACI Web

- R - Réalise
- A - Autorité
- C - Conseille

5.2 Gestion de Projet : Solveur en C

5.2.1 Forces et faiblesses de notre projet : Matrice SWOT

Les forces, opportunités et menaces restant les mêmes pour le projet, nous avons juste dû à analyser les faiblesses de notre équipe vis à vis du développement en C. Nous avons dans cette partie, opté pour du développement en binômes afin de tester et de développer sur les mêmes parties en parallèle, et d'appliquer le principe d'amélioration continue.

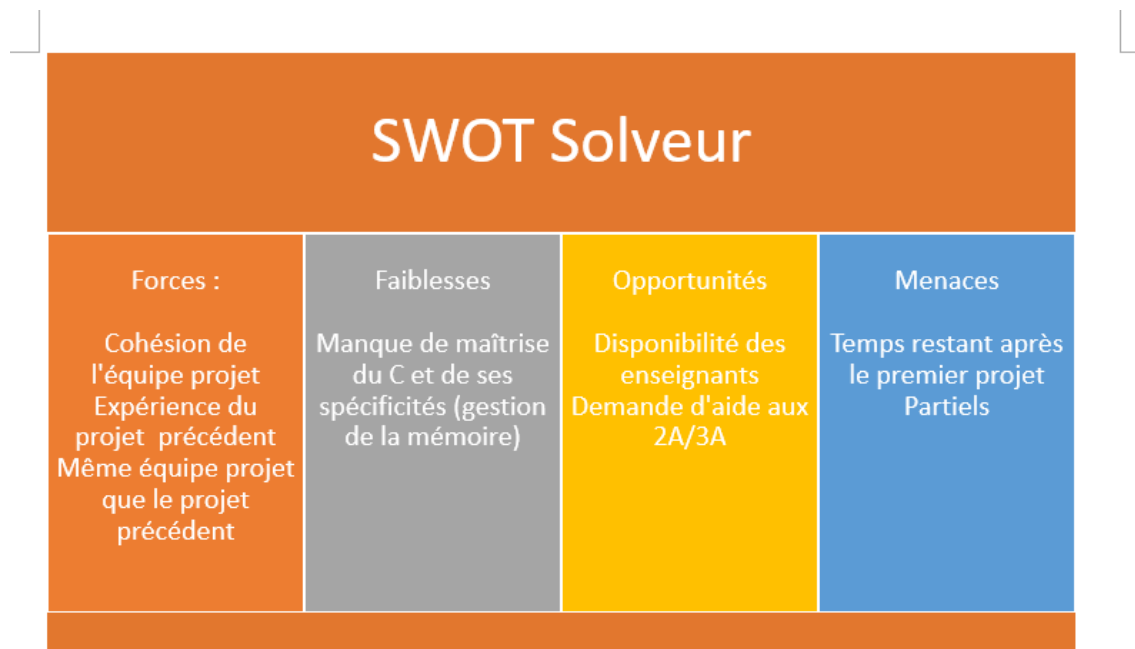


FIGURE 5.4 – Matrice SWOT

5.2.2 Gestion des risques : évaluation de la fréquence et de la gravité des événements

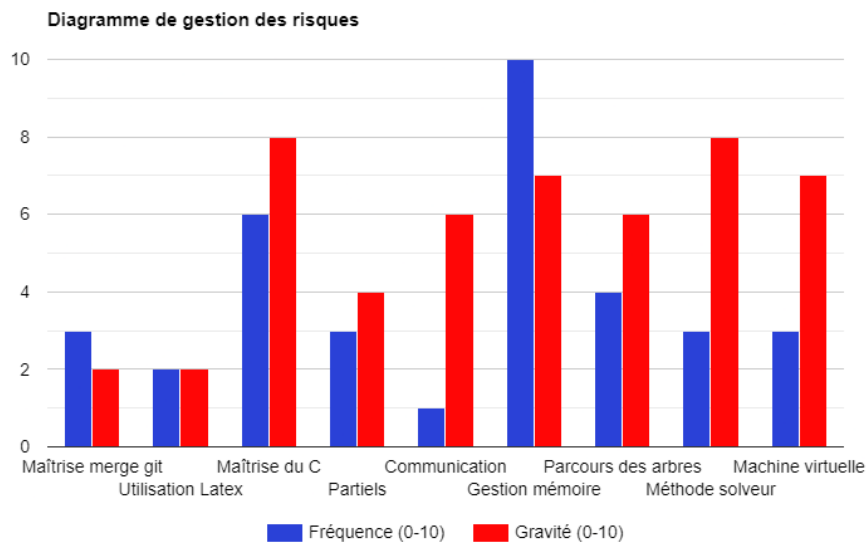


FIGURE 5.5 – Diagramme de fréquence et de gravité pour des risques éventuels

5.2.3 Découpage en lots de travail

Comme expliqué dans la partie conception de notre solveur, nous avons découpé le développement en deux parties.

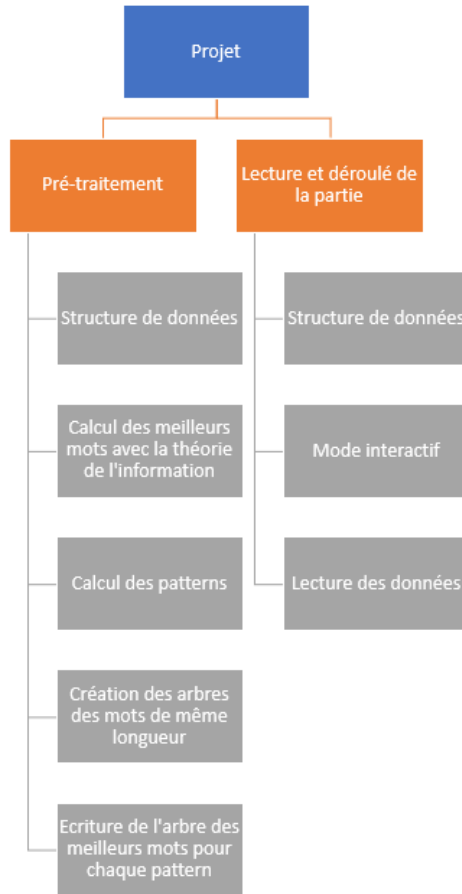


FIGURE 5.6 – Division du projet en lots de travaux

5.2.4 Responsabilités sous-jacentes

Nous nous sommes également répartis les responsabilités au sein de notre projet de solveur selon la matrice RACI suivante :

- R - Réalise
- A - Autorité
- C - Conseille
- I - Informe

	Nathan	Pierre	Julien	Thomas
Structure de données pour le prétraitement	I	A	C	R
Calcul des meilleurs mots avec la théorie de l'information	I	A	C	R
Calcul des patterns des mots et algorithme de remontée de l'arbre préfixe	I	R	C	A
Création des arbres des mots de même longueur	C	R	I	A
Création de l'arbre des meilleurs mots pour chaque pattern possible	C	A	I	R
Structure de données pour le solveur	R	A	I	C
Gestion du mode interactif	A	R	C	I
Lecture des données dans les fichiers texte	R	R/A	C	I

TABLE 5.2 – Matrice RACI du Solveur

Conclusion

A travers notre projet de Wordle, nous avons pu créer tout d’abord une application web orientée sur la création d’un jeu vidéo, nous faisant réfléchir aux notions de game design ainsi qu’au design web, en nous faisant chercher l’aspect pratique et attirant pour les utilisateurs avec des mécaniques aujourd’hui à la mode dans les jeux vidéos, comme les classements, l’expérience, des badges, ainsi que des parties quotidiennes pour inciter à jouer quotidiennement. Bien que l’apport web aura été moindre en programmation, nous avons néanmoins pu appliquer des volontés d’améliorations que nous avions souhaités en fin du premier projet, et qui ont pu porter leur fruit dans les template par exemple. A travers la deuxième partie du projet, nous avons donc dû nous confronter au C, qui est difficile à satisfaire en terme de rigueur quant aux notions de pointeurs et de mémoire. Nous avons cependant pu voir un aspect beaucoup plus théorique de la programmation avec la réflexion autour des structures de données, mais également via des fonctions bien plus complexe que dans la partie web. La coopération aura été plus que nécessaire afin de venir à bout des nombreux problèmes d’exécution, mais elle a été bien présente grâce à un groupe déjà soudé par la réussite du premier projet. Enfin, nous avons pu réussir notre stratégie de faire un solveur orienté par du pré traitement, ce qui au vu de nos tests nous a permis de conclure que cette méthode fonctionne comme supposée préférable sur une grande quantité de parties. Finalement, c’est ce travail de réflexion et en équipe qui aura permis à se projet de se terminer dans les temps, chaque membre ayant été d’une importance capitale dans la complétion de ce projet ardu.

Annexe 1 : Comptes Rendus

Compte rendu 17 Mars 2022

Nous sommes le 17/03/22, Julien, Pierre, Nathan et Thomas sont présents. L'objectif de cette réunion est de discuter sur les débuts du projet, ainsi que les fonctionnalités de l'application web à implémenter. La réunion a lieu sur Discord à 17h le 17/03/22.

Répartition au sein du groupe

- - Tout le monde développe
- - Nathan et Pierre seront testeurs
- - Julien sera responsable design
- - Thomas sera chef de projet

Décisions principales

- - Discussion sur les fonctionnalités de l'application à implémenter
- - Elaboration de la base de donnée
- - Réflexion sur un moyen de classer les mots par difficulté

To Do List

- - Description : Réfléchir à l'implémentation des algorithmes
- - Responsable : Tout le monde
- - Délai : pour la prochaine réunion le 17/03/2022

— - Validé par : Tout le monde

Prochaine Réunion : 20/03/2022

— - Faire le WBS pour l'application

— - Commencer à distribuer les tâches à effectuer pour l'implémentation de l'application

Compte rendu 20 Mars 2022

Nous sommes le 20 mars 2022, Julien, Pierre, Nathan et Thomas sont présents. L'objectif de cette réunion est de discuter des avancées sur le remplissage du WBS et de la matrice RACI en développant ce qui sera nécessaire au wordle, ainsi que la répartition du travail. La réunion a lieu sur Discord à 17h.

Décisions principales

- - Besoin de design de badges + logo + nom de l'appli
- - Fonctionnalités d'inscription et de connexion
- - Voir pour affichage des utilisateurs dans classement
- - Chercher pour ajouter du son lors de vérification des lettres
- - Réflexion sur le fonctionnement du mode survie

To Do List

- - Description : Voir intégration de son + création base de données
- - Responsable : Pierre Pasquier
- - Délai : Dimanche 27/03/2022
- - Validé par : Tout le monde
- - Description : Recherche affichage chronomètre
- - Responsable : Nathan Iori-Gingembre
- - Délai : Dimanche 27/03/2022
- - Validé par : Tout le monde
- - Description : designs + essais quotidiens
- - Responsable : Julien De Toffoli
- - Délai : Dimanche 27/03/2022
- - Validé par : Tout le monde
- - Description : Couleur chronomètre + vibration interface
- - Responsable : Thomas Kieffer
- - Délai : Dimanche 27/03/2022

— - Validé par : Tout le monde

Prochaine Réunion : 27/03/2022

— - Voir avancement des recherches

Compte rendu 03 Avril 2022

Nous sommes le 03 Avril 2022, Julien, Pierre, Nathan et Thomas sont présents. L'objectif de cette réunion est de répartir des différents éléments d'avancement ainsi que se mettre d'accord sur l'implémentation des différentes pages.

Décisions principales

- - Problème de partie libre où tout les effets apparaissent en même temps
- - Template sur toute les pages
- - Problème avec le chrono en 0 pour réduire le temps
- - Réflexion sur les éléments identification et connexion

To Do List

- — - Description : Mise en place de l'historique
 - - Responsable : Pierre Pasquier
 - - Délai : Dimanche 10/04/22
 - - Validé par : Toute l'équipe
- — - Description : Incrémenter le chronomètre et implémentation
 - - Responsable : Nathan Iori-Gingembre
 - - Délai : Dimanche 10/04/22
 - - Validé par : Toute l'équipe
- — - Description : Classement et finir hub + template
 - - Responsable : Julien De Toffoli
 - - Délai : Dimanche 10/04/22
 - - Validé par : Toute l'équipe
- — - Description : Synchroniser effets et succéder les mots
 - - Responsable : Thomas Kieffer
 - - Délai : Dimanche 10/04/22
 - - Validé par : Toute l'équipe

Prochaine Réunion : 10/04/2022

— - Voir avancement des recherches

Compte rendu 10 Avril 2022

Nous sommes le 10/04/2022, Julien, Pierre, Nathan et Thomas sont présents. L'objectif de cette réunion est de voir l'avancement de chacun. La réunion a lieu sur Discord à 20h.

Décisions principales

- - Choix du nom de l'appli : TuNom
- - Problème décalage de la saisie
- - Badges terminés ainsi que logo

Compte rendu 10 avril 2022

Nous sommes le 10/04/2022, Julien, Pierre, Nathan et Thomas sont présents. L'objectif de cette réunion est de voir les avancées de tout le monde. La réunion a lieu sur Discord à 20h30.

Décisions principales

- - Complétion du chronomètre
- - problème survie : impossible d'enchaîner 2 mots
- - avancement sur l'historique et la BD
- - complétion du design du template et du hub, plus qu'à ajouter paramétrage et liaisons inter pages
- - Reflexion sur la forme des motifs dans l'historique

To Do List

- - Description : Compléter l'app serveur
- - Responsable : Toute l'équipe
- - Délai : Jeudi 14/04/22
- Validé par : Toute l'équipe
- - Description : Finir survie et intégrer chronomètre
- - Responsable : Nathan, Thomas
- - Délai : Jeudi 14/04/22
- Validé par : Toute l'équipe
- - Description : Ajouter paramètres dans hub et liaisons pages
- - Responsable : Julien
- - Délai : Jeudi 14/04/22
- Validé par : Toute l'équipe
- - Description : Compléter historique
- - Responsable : Pierre
- - Délai : Jeudi 14/04/22
- Validé par : Toute l'équipe

Prochaine réunion : 14/04/2022

— - Voir l'avancement des recherches de chacun

Compte rendu 14 avril 2022

Nous sommes le 14/04/22, Julien, Pierre, Nathan et Thomas sont présents. L'objectif de cette réunion est d'observer l'avancement mutuel et de tenter de résoudre les problèmes survenus. La réunion a lieu sur Discord à 20h10.

Décisions principales

- - Barre d'xp implémentée
- - Problème d'équilibrage de récompense d'expérience
- - Problème base de donnée (classement)

Prochaine réunion : 21/04/2022

- - Observer l'avancement
- - Voir les difficultés

Compte rendu 21 avril 2022

Nous sommes le 21/04/2022, Julien, Pierre, Nathan et Thomas sont présents. L'objectif de cette réunion est d'observer l'avancement et de résoudre les problèmes. La réunion a lieu sur Discord à partir de 21h.

Décisions principales

- - Avancées sur les paramètres de libre, de home et classement
- - Discussion sur manière de récupérer données et de les stocker
- - Avancées sur historique et survie

To Do List

- - Description : readme + gestion de l'xp
- - Responsable : Nathan
- - Délai : Dimanche 24/04/2022
- Validé par : Toute l'équipe
- - Description : Finir classement et daily
- - Responsable : Julien
- - Délai : Dimanche 24/04/22
- Validé par : Toute l'équipe
- - Description : Finir historique et BD
- - Responsable : Pierre
- - Délai : Dimanche 24/04/22
- Validé par : Toute l'équipe
- - Description : Récupérer données de la partie survie
- - Responsable : Thomas
- - Délai : Dimanche 24/04/22
- Validé par : Toute l'équipe

Prochaine réunion : 24/04/2022

— - Finir l'application Web et commencer à préparer l'oral

Compte rendu 24 avril 2022

Nous sommes le 24/04/22, Julien, Pierre, Nathan, Thomas sont présents. L'objectif de cette réunion est d'observer l'avancement et de résoudre les problèmes. La réunion a lieu sur Discord à partir de 20h20.

Décisions principales

- - Thomas a fait des tests sur le site et a trouvé des bug
- - Pierre a corrigé la BD, ajouté inscription et connexion
- - Nathan a fini le readme et l'ajout d'xp du mode survie
- - Julien a fini daily et le classement

To Do List

- - Description : Compléter l'app serveur
- - Responsable : Toute l'équipe
- - Délai : 30/04/22
- Validé par : Toute l'équipe
- - Description : Corriger bugs et finir classement
- - Responsable : Julien
- - Délai : 30/04/22
- Validé par : Toute l'équipe
- - Description : Liaisons inter pages
- - Responsable : Julien
- - Délai : 30/04/22
- Validé par : Toute l'équipe
- - Description : Compléter historique
- - Responsable : Pierre
- - Délai : 30/04/22
- Validé par : Toute l'équipe

A voir après la soutenance

— - Voir préparatifs du solveur en C

Compte rendu 05 mai 2022

Nous sommes le 05/05/2022, Julien, Pierre, Nathan et Thomas sont présents. L'objectif de cette réunion est de choisir la structure de données pour le solveur. La réunion a lieu sur Discord à partir de 21h45.

Echange :

- - Réflexion sur l'utilisation des arbres pour stocker les données
- - Réflexion sur différence arbre normal et préfixe afin de stocker les parties en amont

To Do List

- - Description : Réfléchir à implémentation en C
- - Responsable : Toute l'équipe
- - Délai : Mardi 10/05/2022
- Validé par : Toute l'équipe

Prochaine réunion : 11/05/2022

- - Réfléchir à implémentation des arbres

Compte rendu 11 mai 2022

Nous sommes le 11/05/2022, Julien, Pierre, Nathan et Thomas sont présents. L'objectif de cette réunion est de décider de la manière d'implémenter le solveur. La réunion a lieu sur Discord à 21h.

Décisions principales

- - Un fichier par longueur de mot contenant les mots dans un arbre (‘,’ pour séparer les mots d'un même père et ‘;’ pour séparer les mots de pères différents). Chaque mot sauf la racine est suivi d'un espace et de son patrone en base 10 par rapport à son père.
- - Pour la structure de l'arbre, chaque liste de fils est stockée dans un tableau pour que la lecture soit le plus vite possible.
- - Pour le pré-traitement : mots stockés dans un arbre préfixe. Il y a un tableau de booléen associé à chaque lettre de l'arbre (défini dans la structure élément) qui donne dans quel mot est la lettre (1 si elle est dans le mot, 0 sinon)

To Do List

- - Description : Réfléchir aux fonctions à implémenter
- - Responsable : Toute l'équipe
- - Délai : Vendredi 12/05/22
- Validé par : Toute l'équipe

Prochaine réunion : 12/05/2022

- - Distribuer les fonctions à implémenter

Compte rendu 12 mai 2022

Nous sommes le 12/05/2022, Julien, Pierre, Nathan et Thomas sont présents. L'objectif de cette réunion est de voir les avancées de tout le monde. La réunion a lieu sur Discord à 16H30.

Décisions principales

- - Façon de parcourir la structure en arbre
- - Répartition en un duo arbre + parcours afin de prétraiter le dictionnaire et d'un duo solveur qui lira le fichier final et s'occupera de l'interface utilisateur
- - Fixer le fonctionnement de la structure de donnée et répartition du travail

To Do List

- - Description : Avancer sur la structure d'arbre
- - Responsable : Pierre Pasquier, Thomas Kieffer
- - Délai : Jeudi 15/05/22
- Validé par : Toute l'équipe
- - Description : Avancer sur le parcours de prétraitement de l'arbre
- - Responsable : Pierre Pasquier
- - Délai : Jeudi 15/05/22
- Validé par : Toute l'équipe
- - Description : Avancer sur le solveur
- - Responsable : Julien De Toffoli, Nathan Iori-Gingembre
- - Délai : Jeudi 15/05/22
- Validé par : Toute l'équipe
- - Description : Avancer sur le rapport
- - Responsable : Toute l'équipe
- - Délai : Jeudi 15/05/22
- Validé par : Toute l'équipe

Prochaine réunion : 15/05/2022

— - Observation de l'avancement des duos

Compte rendu 15 mai 2022

Nous sommes le 15/05/2022, Julien, Pierre, Nathan et Thomas sont présents. L'objectif de cette réunion est de voir les avancées de tout le monde. La réunion a lieu sur Discord à 20h.

Décisions principales

- - Observer l'avancement de chaque parties et résoudre les potentiels problèmes rencontrés.
- - Fonction récursive permettant de résoudre la suppression de l'arbre.
- - Problème sur un while dans une fonction qui renvoie des résultats inexpliqué.
- - Mise en accord sur l'imbrications des différentes parties.

To Do List

- - Description : Lecture du fichier et réalisation de Lecture finale
- - Responsable : Julien De Toffoli, Nathan Iori-Gingembre
- - Délai : Jeudi 19/05/22
- Validé par : Toute l'équipe
- - Description : Regarder à comment écrire les résultats dans le fichier texte
- - Responsable : Pierre Pasquier, Thomas Kieffer
- - Délai : Jeudi 15/05/22
- Validé par : Toute l'équipe
- - Description : Exploitation des pattern pour naviguer dans l'arbre
- - Responsable : Julien De Toffoli, Nathan Iori-Gingembre
- - Délai : Jeudi 15/05/22
- Validé par : Toute l'équipe

Prochaine réunion : 19/05/2022

- - Observation de l'avancement des duos

Compte rendu 19 mai 2022

Nous sommes le 19/05/2022, Julien, Pierre, Nathan et Thomas sont présents. L'objectif de cette réunion est de choisir la méthode de reconnaissance d'éléments pour la construction de l'arbre des paternes. La réunion a lieu sur Discord à 21h30.

Décisions principales

- - Se mettre d'accord sur le raccord entre pré-traitement et post-traitement (interaction solveur)
- - Sur la façon de « lire » le fichier texte, choix de la syntaxe pour différencier les fils d'un même père
- - Sur les problèmes du pré-traitement

To Do List

- - Description : Régler les problèmes de mémoire
- - Responsable : Pierre Pasquier
- - Délai : Jeudi 22/05/22
- Validé par : Toute l'équipe
- - Description : Régler le problème d'écriture dans le fichier texte pour le traitement
- - Responsable : Thomas Kieffer
- - Délai : Jeudi 22/05/22
- Validé par : Toute l'équipe
- - Description : Création structure d'arbre adéquate pour la lecture
- - Responsable : Julien De Toffoli, Nathan Iori-Gingembre
- - Délai : Jeudi 22/05/22
- Validé par : Toute l'équipe

Prochaine réunion : 22/05/2022

- - Voir l'avancement des recherches de chacun

— - Mettre en commun la lecture // écriture du fichier de pré-traitement

Compte rendu 22 mai 2022

Nous sommes le 22/05/2022, Julien, Pierre, Nathan et Thomas sont présents. L'objectif de cette réunion est de voir les avancées des deux groupes. La réunion a lieu sur Discord à 20h.

Décisions principales

- - Mise en corrélation des avancements des fonctions des 2 groupes
- - Difficulté sur la bonne fonction à utiliser afin qu'elle réponde à toutes nos attentes (lectures ligne par ligne ..)
- - Reflexion sur le fichier texte transmis pour le passage sous forme d'arbre
- - Manque la partie recursive pour remplir l'arbre

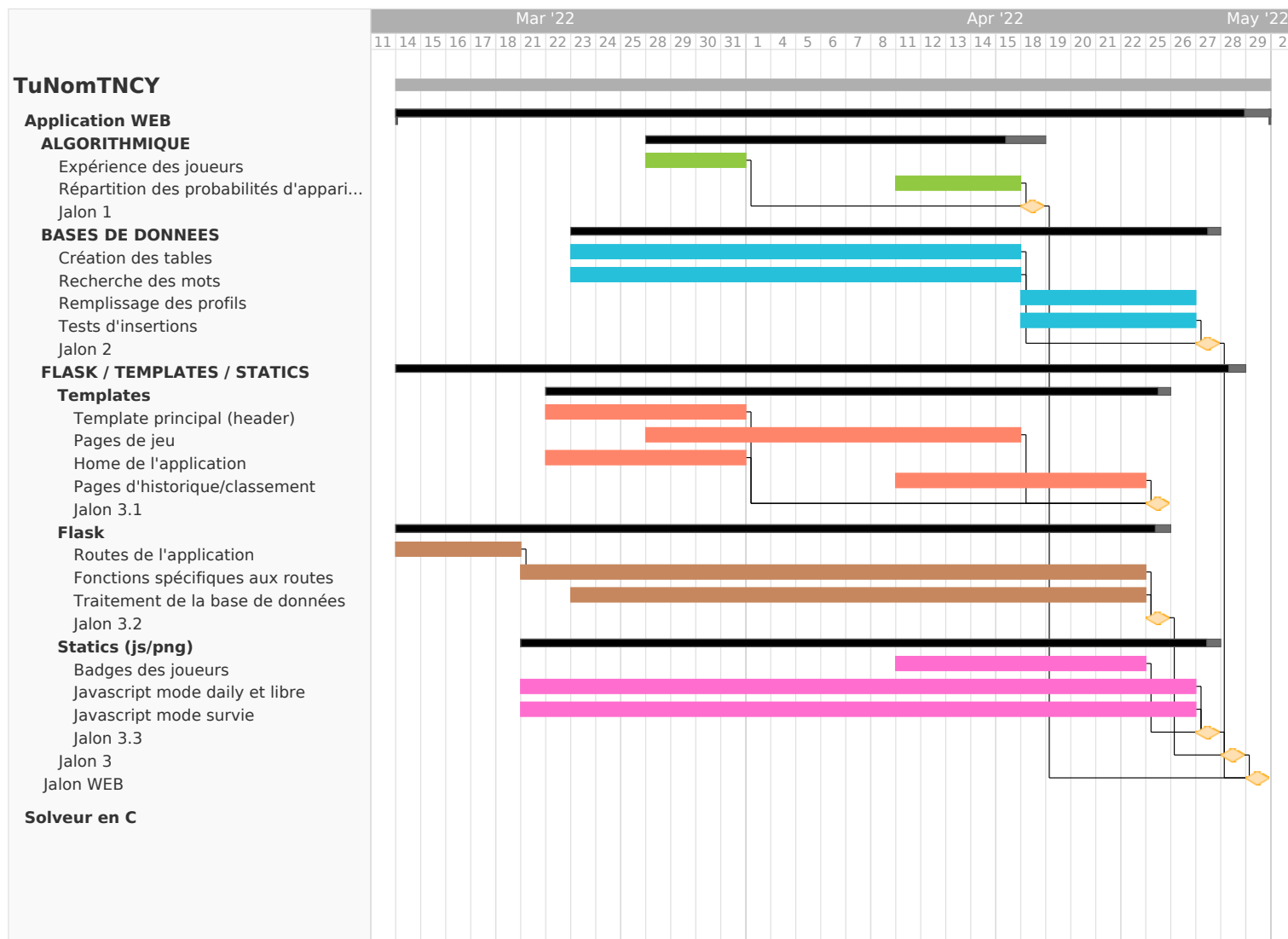
To Do List

- - Description : Fonction pour remplir l'arbre récursivement
- - Responsable : Pierre Pasquier, Thomas Kieffer
- - Délai : Jeudi 26/05/22
- Validé par : Toute l'équipe
- - Description : Finir la fonction de création d'arbre et le main
- - Responsable : Julien De Toffoli, Nathan Iori-Gingembre
- - Délai : Jeudi 26/05/22
- Validé par : Toute l'équipe

Prochaine réunion : 26/05/2022

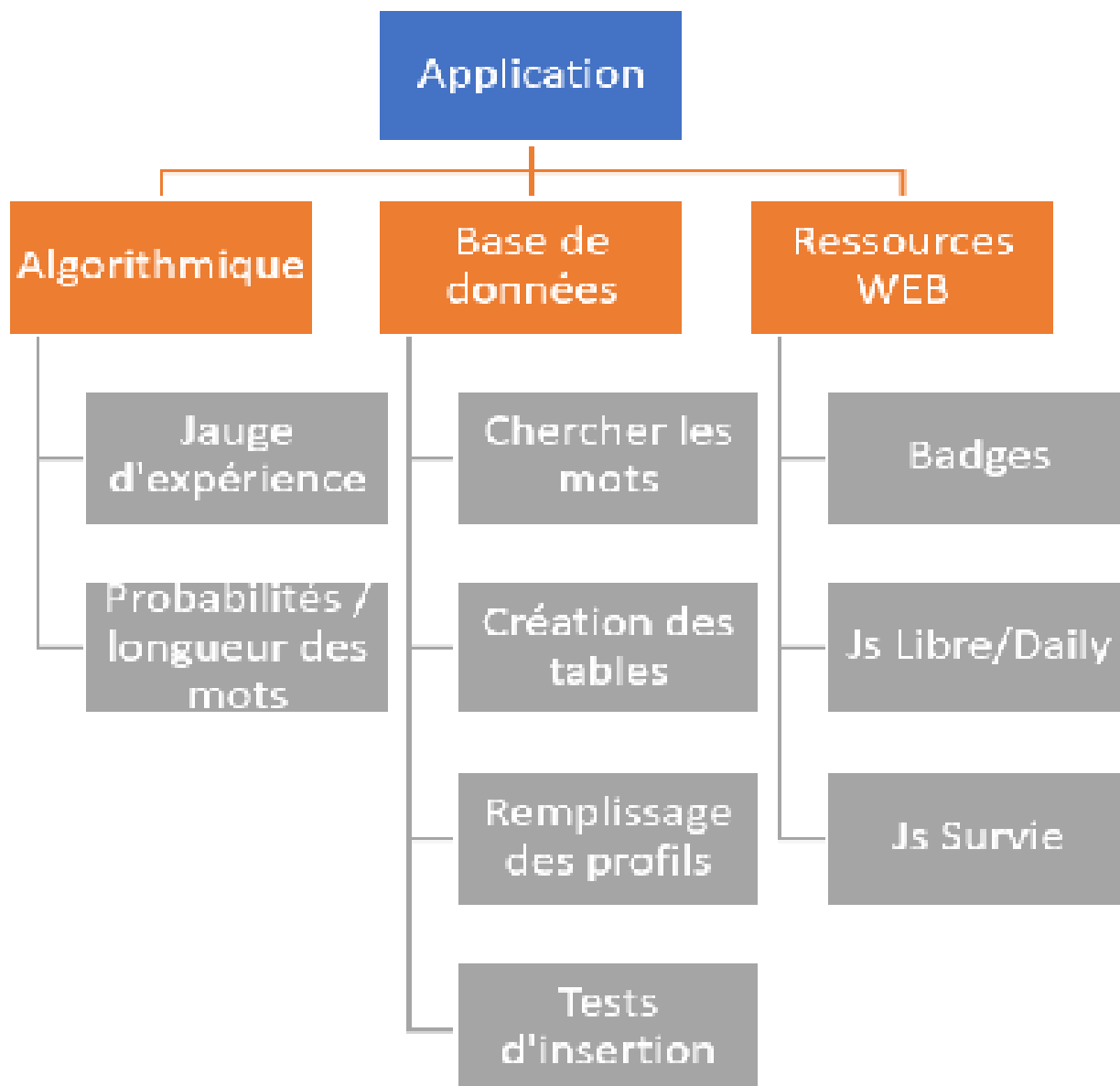
- - Voir l'avancement des recherches de chacun
- - Mettre en commun la lecture // écriture du fichier de pré-traitement

Annexe 2 : Diagramme de Gantt Web

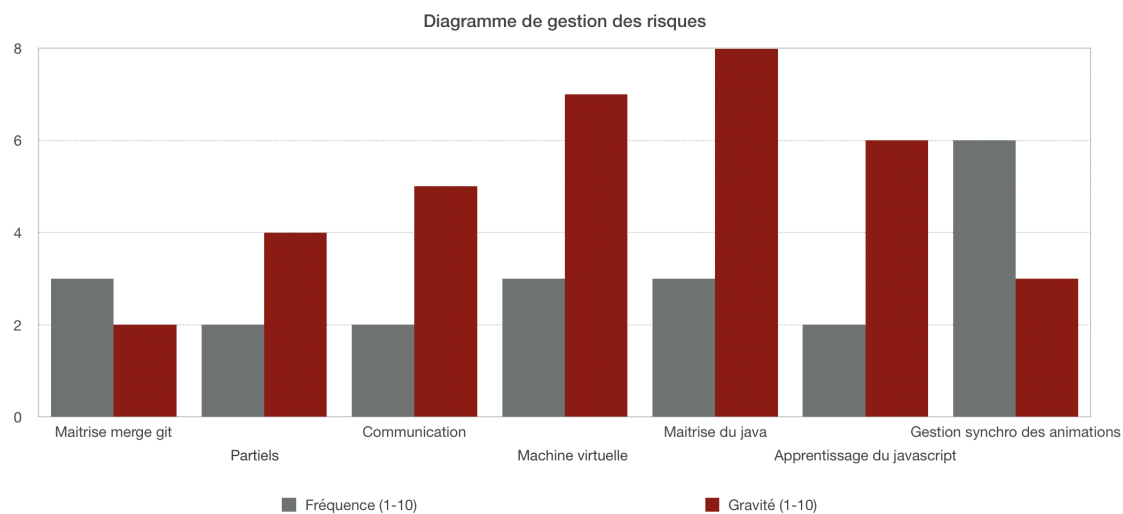


Annexe 3 : WBS Web

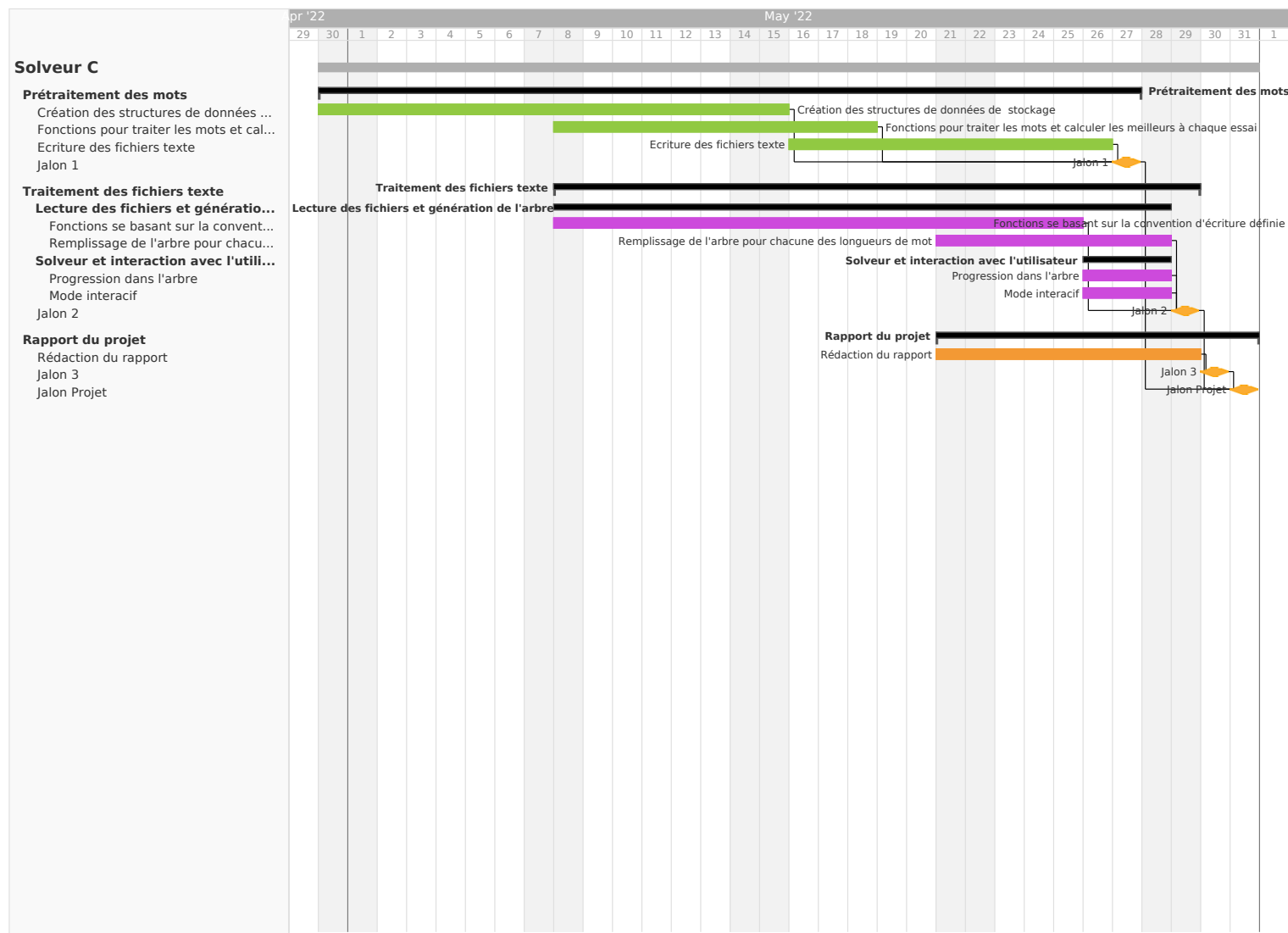
WBS : Application WEB



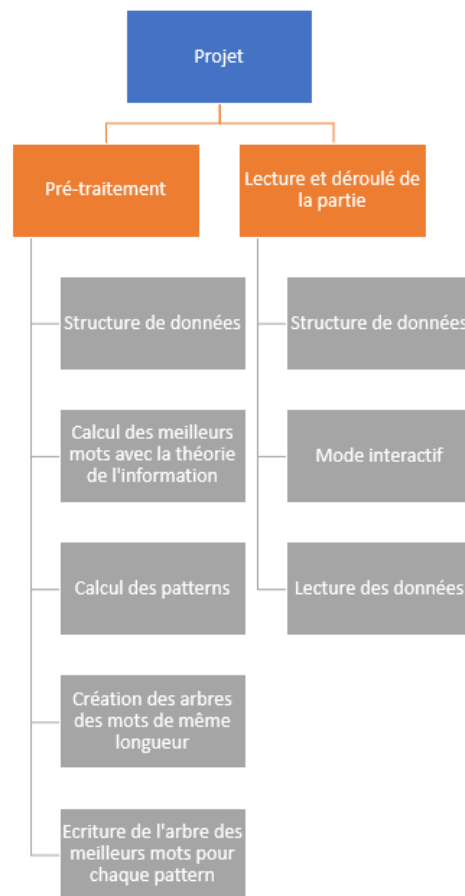
Annexe 4 : Gestion des risques WEB



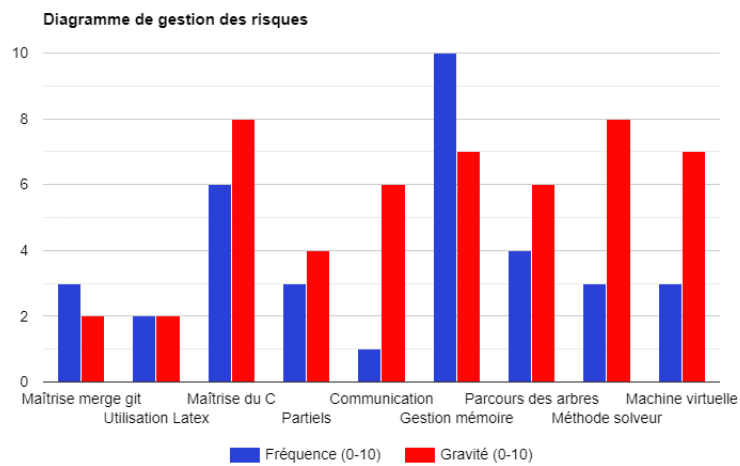
Annexe 5 : Diagramme de Gantt Solvreur



Annexe 6 : WBS Solveur



Annexe 7 : Gestion des risques Solveur



Bibliographie

- Le github de Snow : <https://github.com/mortie/snow/>
- Le dictionnaire pour le solveur : <https://www.jeuxdujardin.fr/sites/dujardin/files/notices/MOTUS.pdf>
- Le dictionnaire pour le site web à trouver dans le jeu téléchargé : <https://scripts.eggdrop.fr/details-Motus-s2.html>
- Le site pour l'autoformation en Javascript : <https://developer.mozilla.org/fr/docs/Web/JavaScript>
- Le site pour la réalisation des Gantt en ligne : <https://www.teamgantt.com/>
- Explications sur la structure d'arbre préfixe et la gestion de la mémoire : Chaîne Youtube de Jacob Sorber
- Introduction sur la théorie de l'information : <https://www.youtube.com/c/3blue1brown/videos>