Projet de programmation

Algorithmes, choix techniques, outils utilisés

Albanel Clément, Alves Nathanael, Auzi Emma, Bornes Yohan, Coudougnan Sylvain, Devidas Hugo

Mars 2022

Table des matières

1 Outils utilisés

1.1 Langages

D'après l'article de thèse [1] qui préconise d'utiliser un tableau de bits pour la manipulation (et le stockage) de la base de données de nos mots et définitions associées; il est conseillé d'utiliser le langage **Cpp** qui permet une manipulation simple de ces derniers. Le language C est également conseillé, mais nous avons néammoins choisi le langage **Cpp** afin d'avoir une plus grande liberté sur la structure du code (utilisation de Design Pattern).

1.2 Interface Graphique

Il ressort de nos recherches sur les meilleures interfaces graphiques en Cpp que le framework Qt [2] semble être le plus performant, documenté et simple d'utilisation. C'est celui que nous allons utiliser.

1.3 Choix du dictionnaire

Comme stipulé dans les articles mentionnés dans le cahier des besoins, afin d'obtenir de bonnes performances (et obtenir une complexité de recherche / intersection de domaines **linéaire**) nous avons recherché une base de données contenant des mots issus de mots croisés, et classés par pertinence. Nous l'avons trouvé sur ... (mettre référence bibliographique). En utilisant la bibliothèque **numpy** de python, nous lui avons ensuite appliqué un traitement (*mettre ta fonction python ici*) afin de la trier par taille et récursivement par ordre alphabétique. Ainsi, nous avons obtenu une base de données de **31316** mots. Bien entendu, chacun des mots est associé à une définition.

2 Représentation du jeu des mots croisés

2.1 Choix de conception

Nous avons modélisé notre grille de mots croisés par un objet *grid*, celui-ci représente la grille avec laquelle l'utilisateur interragit. Cette grille est constituée d'un deuxième type d'objet important que nous avons nommés *variables*. Ce choix permet d'apporter la sémantique des lettres constituant un mot. Ces *variables* peuvent être vues comme des champs ou espaces pouvant accueillir un mot. Ainsi la *grid* générée contient autant de *variables* que de mots que l'utilisateur devra trouver.

La *grid* est également constituée de cases noires, c'est grâce à ces cases que les *variables* sont délimitées. Par exemple, une même ligne ou colonne pourra avoir plusieurs *variables*. Cette configuration précise de cases noires est enregistrée dans ce que nous appelons des *pattern* : petits fichiers contenant le modèle précis de la *grid* à utiliser. Ces *patterns* sont invisibles pour l'utilisateur et seront chargés aléatoirement selon trois niveaux de difficultés. Notre choix a été de définir la difficulté comme des dimensions de *grid*:

- Facile: grille de 6x6 cases
- Moyen: grille de 8x8 cases
- Difficile: grille de 10x10 cases

Dans le processus de génération de la grille, on manipule un objet *database* que l'on associe à la *grid*, il contient l'ensemble des mots qui pourront être proposés dans une grille prête à jouer. A la fin de ce processus, une fois que la grid a pu remplir chaque variable avec un mot valide, une fonction indépendante permet d'associer à chaque mot, une définition. L'utilisateur peut à présent tenter de déterminer les mots de la grille grâce à notre interface adéquate.

2.2 GRID - Grille de jeu

Tableau à deux dimensions d'une taille définit par le niveau de difficulté et contient des cases pouvant accueillir ou non (cases noires) des lettres. Plus précisément elle contient les élements suivants:

- une *database* : dictionnaire et l'ensemble des fonctions le manipulant.
- un *pattern* : configuration initiale de la grille, seules la position des cases noires et les dimensions de la grille y sont présentes.
- un tableau des *variables* de la grille.
- le nombre de ces variable.

Dans un premier temps la *grid* est initialisée comme non remplie, on lui associe un dictionnaire pour sa résolution future, un pattern pour obtenir sa structure, et on définit un ensemble de variables grâce à notre **Algorithme d'initialisation des Variables**.

La *grid* est initialisée mais ne contient pas encore de mots dans ses *variables*, il suffit alors de lui appliquer le solver à l'aide de l'**Algorithme d'attribution de mots aux Variables**.

2.3 VARIABLE - Champ acceuillant un mot

Afin de représenter le problème de la génération des mots croisés sous format de **CSP** (problème de satisfaction de contraintes), il convient de définir plus précisément ici les *variables*.

Pour cela, chaque variable contiendra les élements suivants :

- le mot qu'elle contient et sa taille (à l'initialisation ce mot est définis à *null*).
- les autres variables avec lesquelles elles s'intersectent (ainsi que les positions d'intersection).

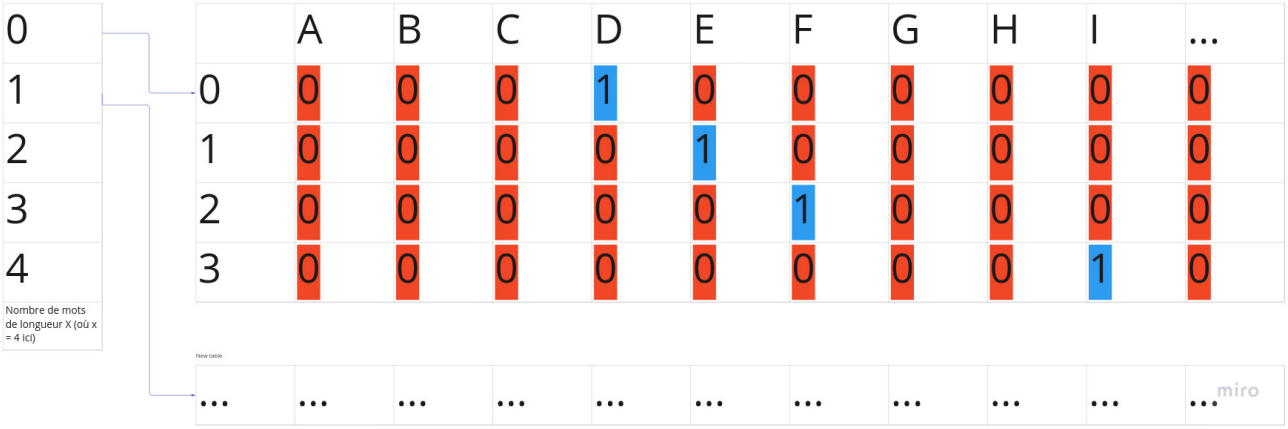
- sa position absolue dans la grille.
- un booléen pour savoir si le mot stocké est vertical ou horizontal.
- le *domaine* : ensemble des mots que peut accueillir la variable, ainsi que sa cardinalité.

Le *domaine* est un sous ensemble du dictionnaire *database*. Initialement il contient tous les mots de *database* d'une longueur égale à l'espace occupé par la *variable*.

2.4 DATABASE - Stockage des mots

Comme énoncé en section langages, nous allons utiliser des tableaux de bits pour stocker nos mots. C'est en effet sous cette représentation que l'on trouve les meilleures performances en terme de recherche algorithmiques de mots selon des critères précis d'après [1]. Par exemple, ce stockage permet de trouver un mot de longueur 5 dont la première lettre est un a et la 3ème un c en $\mathbf{O}(d/32)$ avec $\mathbf{d}$ les valeurs que peuvent prendre chacune des variables (énoncé en section précédente).

Ainsi, cela permet de trouver un mot avec une complexité **linéaire**. De plus, la représentation de cette base de données de mots sous forme de tableaux de bits permet de minimiser l'espace mémoire utilisé. En effet, pour un dictionnaire de **20 000** mots, seuls 2.4 Mb d'espace seront utilisés. Concentrons nous maintenant sur l'implémentation de ce tableau de bits. Pour simplifier, nous allons donner un exemple de la gestion globale des mots dans notre base de données :



Sur la figure ci-présente, nous présentons comment le mot "**defi**" est stocké. Ici, le tableau en vert représente le tableau où chaque indice correspond à des mots de longueurs 4. Il y a ainsi un tableau de pointeurs (menant à des tableaux 2D pour chaque mot) par longueur de mot. Ce stockage nous permet de trouver très rapidement un mot de telle longueur possédant telle lettre à tel indice, et donc de l'intersecter éventuellement aussi avec un autre mot possédant d'autre caractéristiques dans le but de construire la grille.

On aura donc :

- Pour chaque longueur de mot de taille X, un tableau d'entier de longueur *nbMotTailleX*.
- Pour chaque mot un tableau de taille *nbLettre*
- Pour chaque lettre un tableau de bits de taille 26

Cela se fait grâce à notre **Algorithme d'encodage des mots en tableau de bits**.

3 Algorithmes

3.1 Initialisation des Variables

Algorithm 1 Algorithme d'initialisation des Variables

```
case_precedente ← case_noire
case_actuelle ← case_noire
taille_mot ← 0
for chaque colonne i do
    for chaque ligne j do
        case_precedente ← case_actuelle
        case_actuelle ← case[i][j]
        if (case_precedente == case_noire ET case_actuelle == case_lettre) then
            taille_mot ++
            retient la position
        end if
        if (case_precedente == case_lettre ET case_actuelle == case_lettre) then
            taille_mot ++
        end if
        if (case_precedente == case_lettre ET case_actuelle == case_noire) then
            taille_mot ← 0
            if (taille_mot ≥ 3*) then
                Ajout variable (avec position, direction horizontale et taille_mot)
            end if
        end if
    end for
    if (case_actuelle == case_lettre ET (taille_mot ≥ 3*)) then
        Ajout variable (avec position, direction et taille_mot)
    end if
    taille_mot ← 0
    case_actuelle ← case_noire
end for
                                ▷ Ré appliquer l'algorithme une seconde fois avec un parcours vertical
for chaque variable v do
    initialiser le domaine de v
    for chacun de ses voisins w do
        définir w comme voisin de v
    end for
end for
```

* Plus petit mot possible que nous avons décidé d'utiliser.

3.2 Attribution des mots aux Variables - Solver

Pour générer une grille qui a une solution, l'objectif est d'associer un mot à chaque *variable* tout en respectant les contraintes de celles-ci.
* Renvoie la variable dont la cardinalité associée au domaine est minimale.
** Retourne le premier mot du domaine de la Variable.

Par un processus de randomisation du *domaine* des *variable*s, deux appels sur une même *grid* pourra produire plusieurs grilles differentes.

Si une solution à la grille peut-être trouvée, alors elle sera proposée à l'utilisateur avec les définitions associées et les mots non révélés.

Algorithm 2 Algorithme d'attribution des mots aux Variables

```
while Toutes les variables ne sont pas assignées do
    Calculer la taille du domaine de chaque Variable de la grille
    Variable v = chooseVariable() *
    char* word = v -> getWordFromVarDomain() **
    if (word != NULL) then
        Valeur de v ← word
        Mettre à jour les domaines des voisins de v
    else
        On retourne en arrière et on teste la valeur suivante pour la variable précédente
    end if
end while
```

3.3 Encodage des mots en tableau de bits

Ci-après l'algorithme que nous utiliserons pour encoder notre base de données de mots en tableau de bits :

Algorithm 3 Encodage base de donnée du lexique

```
listDeMots ← chargemotdepuisnotrelexique
dBWordLength2 ← allocation[nbTotalWord2][2][26]      ▷ création des tableaux pour chacune des longueurs de mot
for mot dans listDeMots do ajouterMot(dbWordLenght(lenghtMot))
end for
```

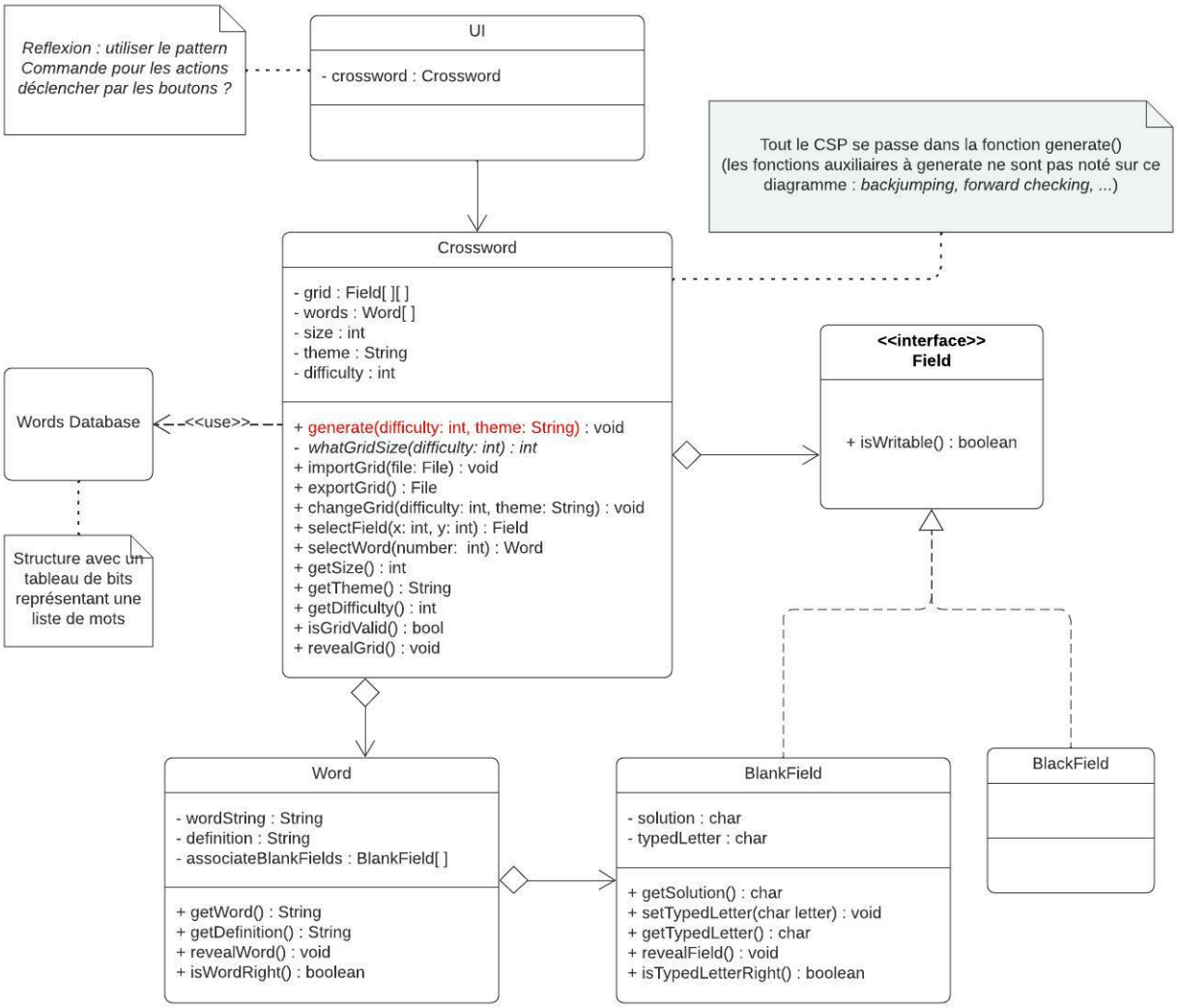
Nous considérons que cet algorithme est utilisé dans notre constructeur de l'objet "DataBase" et que les dBWordLength sont ses attributs. Nous allons maintenant voir comment ajouter un mot dans un sous-tableau (fonction que nous avons appelé dans la fonction précédente) :

Algorithm 4 Ajouter Mot

```
for lettre dans mot do
    index ← lettre −′ a′
    dbWord[currentWordIndex][currentWordSize][index] ← 1
end for
```

4 Diagramme de classes

Ci-après une première version du diagramme **UML** tel que nous l'imaginons actuellement.



Une case de la grille de mot croisés est représentée par l'interface Field, une case peut être une case blanche à remplir: classe BlankField, ou une case noire: classe BlackField.

BlankField est composé de deux attributs:
- solution : sera null avant la génération de la grille, lors de la génération de la grille, il contiendra la lettre attendue.
- typedLetter : il contiendra la lettre saisie par l'utilisateur, il sera null si l'utilisateur n'a rien rentré dans la case.
Il y a une fonction pour révéler la solution et une qui permet de savoir si la lettre rentrée (typedLetter) correspond à la lettre attendue (solution).

La Classe Word correspond à un mot de nos mots croisés, elle est composée de 3 attributs:
- wordString : chaine de caractère representant le mot.
- definition : definition du mot, qui sera affichée dans notre jeu de mots croisés.
- associateBlankFields : tableau pointant vers les cases blanches du jeu de mots croisés qui sont associés au mot. En vue de vérifier la validité d'un mot saisie par l'utilisateur (et aussi afficher la

solution) plus facilement, nous aurons seulement à parcourir ce tableau.
Il y a une fonction pour révéler le mot et une qui permet de savoir si le mot saisie correspond au mot attendue (typiquement ces fonctions devraient parcourir le tableau associateBlankFields et appeler les méthodes correspondantes pour chacune des cases).

La classe Crossword est la grille de mots croisés, composé de 5 attributs:
- grid : un tableau en 2 dimensions de type Field, représente la grille de mots croisés.
- words : liste des mots retenus dans notre grille.
- size : taille de la grille (par exemple taille 8 signifie que la grille est en 8x8).
- theme : thème de la grille.
- difficulty : difficulté de la grille.

La méthode generate() contient l'algorithme de génération d'une grille décrit plus tôt dans le document, generate() manipulera une base de données de mots : cf. stockage des données.
Il y a une méthode d'importation de grille et une autre d'exportation de grille, l'importation et l'exportation de grille n'étant pas des besoins essentiels, les algorithmes n'ont pas encore été décrits.
La méthode changeGrid() est là pour repondre au besoin utilisateur "Changer de grille", elle se contentera de nettoyer les instances en places puis rapellera generate().
La méthode selectField() nous renvoie la case correspondante à la position (dans la grid) passée en paramètre. selectWord() nous renvoi le mot correspondant à l'index (dans le tableau words) rentré en paramètre.
Il y a aussi une méthode pour vérifier si la grille est valide et une méthode pour révéler la grille.

References

[1] John Duchi James Connor and Bruce Lo. "Crossword Puzzles and Constraint Satisfaction". In: 2005. URL: https://dept-info.labri.fr/~narbel/PdP/Subjects21-22/Crosswords/crossword_writeup.pdf.

[2] Qt. *Qt Cross-platform*. https://www.qt.io/. Accessed: 14-03-2022. Last update in 2022.