

Projet de programmation

Mémoire - Mots croisés

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

Avril 2022

Table des matières

| | | |
|-----------|--|-----------|
| 1 | Objet de la demande | 5 |
| 1.1 | Utilisateurs et parties prenantes du projet | 5 |
| 1.1.1 | En interne | 5 |
| 1.1.2 | En externe | 5 |
| 1.2 | Description des objectifs généraux | 5 |
| 2 | Analyse de l'existant | 6 |
| 2.1 | Introduction | 6 |
| 2.2 | Outils en ligne | 6 |
| 2.3 | Logiciel nécessitant une installation | 10 |
| 2.4 | Synthèse | 11 |
| 3 | Besoins fonctionnels | 12 |
| 3.1 | Avant-propos | 12 |
| 3.2 | Diagramme des cas d'utilisation | 12 |
| 3.3 | Besoins utilisateurs | 13 |
| 3.4 | Besoins système | 31 |
| 4 | Besoins non fonctionnels | 32 |
| 4.1 | Besoins d'utilisabilité | 32 |
| 4.2 | Besoins de performances | 32 |
| 4.3 | Besoins de sécurité | 32 |
| 5 | Parallèle entre notre analyse initiale et les choix faits lors du développement | 33 |
| 5.1 | Maquettes | 33 |
| 5.2 | Patterns | 34 |
| 5.3 | Format de données pour l'importation et l'exportation | 34 |
| 6 | Représentation du jeu des mots croisés et algorithmes | 36 |
| 6.1 | SOLVER - Coeur de l'algorithme de génération | 36 |
| 6.2 | VARIABLE - Champ accueillant un mot | 36 |
| 6.3 | Algorithme d'initialisation des Variables | 36 |
| 6.4 | Algorithme d'attribution des mots aux Variables (Solver) | 38 |
| 6.5 | Lexique (base de données) - Stockage des mots | 38 |
| 6.5.1 | Algorithme d'encodage des mots en tableau de bits | 39 |
| 7 | Techniques utilisées | 40 |
| 7.1 | Langages | 40 |
| 7.2 | JSON | 40 |
| 7.3 | Interface Graphique | 40 |
| 7.4 | Choix du dictionnaire | 40 |
| 8 | Architecture de l'application | 42 |
| 8.1 | Démarche | 42 |
| 8.2 | Model-View-Controller | 45 |
| 8.3 | Design patterns | 45 |
| 8.4 | Diagramme de classes UML détaillé | 45 |
| 8.5 | Diagramme de paquetages UML | 47 |
| 9 | Documentation du code | 47 |
| 10 | Résultat final | 49 |

| | |
|---|-----------|
| 11 Tests de génération de grilles sur des grilles de tailles variables | 53 |
| 11.1 Sur des grilles carrées | 53 |
| 11.1.1 Avec cases noires | 53 |
| 11.1.2 Grilles vides | 54 |
| 11.2 Sur des grilles rectangulaires | 54 |
| 11.2.1 Avec cases noires | 54 |
| 11.2.2 Grilles vides | 55 |
| 11.3 Bilan du test de génération | 55 |
| 12 Un peu de combinatoire | 56 |
| 12.1 Complexité algorithmique | 56 |
| 12.1.1 Quelques chiffres | 57 |
| 12.2 Tests avec une base de données de mots de taille 6 simplifiée | 57 |
| 12.3 Un premier bilan de complexité | 58 |
| 12.4 Tests avec des mots "parasites" | 59 |
| 13 Combinatoire, une fatalité sans issue ? | 60 |
| 13.1 Quelques améliorations apportées | 60 |
| 13.1.1 Coût de l'application d'une contrainte | 60 |
| 13.1.2 Mots appliquant la même contrainte sur une variable bloquée | 61 |
| 13.1.3 Les résultats suites aux modifications apportées | 61 |
| 14 Tests aux frontières - limites de l'application | 62 |
| 14.1 Conditions d'expériences | 62 |
| 14.2 Limites en temps | 62 |
| 14.3 Limites en mémoire | 63 |
| 15 Tests concernant la base de données | 64 |
| 16 Tests unitaires | 66 |
| 16.1 Choix de l'outil de test | 66 |
| 16.2 Implémentation | 66 |
| 17 Gestion de projet | 67 |
| 17.1 Outils utilisés | 67 |
| 18 Conclusion | 68 |
| 18.1 Bilan technique | 68 |
| 18.2 Ce qu'on a fait, pas fait | 68 |
| 18.3 Perspectives et améliorations | 68 |
| 18.4 Bilan global | 69 |
| 19 Nos vraies motivations pour la réalisation de ce solver | 71 |

Table des figures

| | | |
|----|---|----|
| 1 | Interface Application "The Teacher's Corner : puzzle and activities" | 6 |
| 2 | Interface Application "Sport Cérébral : mots croisés" | 7 |
| 3 | Interface Application "Education.com : Crossword Puzzle" | 8 |
| 4 | Interface Application "Puzzel" | 9 |
| 5 | Interface Application "Eclipse Crossword" | 10 |
| 6 | Interface Application "Mots croisés pro" | 11 |
| 7 | Diagramme de cas d'utilisation | 12 |
| 8 | Maquette - Accueil | 13 |
| 9 | Maquette - Menu d'importation | 14 |
| 10 | Maquette - Grille précédemment remplie importée | 14 |
| 11 | Maquette - Choix de la difficulté | 16 |
| 12 | Maquette - Choix du thème | 17 |
| 13 | Maquette - Paramétrage de la grille | 18 |
| 14 | Maquette - Grille vide | 19 |
| 15 | Maquette - Partie en cours | 21 |
| 16 | Maquette - Menu fichier | 23 |
| 17 | Maquette - Menu export | 24 |
| 18 | Maquette - Aides | 26 |
| 19 | Maquette - Solution | 28 |
| 20 | Maquette - Grille remplie | 30 |
| 21 | Maquette - Validation | 31 |
| 22 | Maquette finale - Grille vide | 33 |
| 23 | Maquette finale - Grille remplie | 33 |
| 24 | Exemple de patterns pour chaque difficulté | 34 |
| 25 | Exemple de fichier d'importation | 35 |
| 26 | Classe "Crossword" | 42 |
| 27 | Classe "Difficulty" | 42 |
| 28 | Diagramme de classes simplifié | 44 |
| 29 | Schéma de notre implémentation du MVC | 45 |
| 30 | Diagramme de classes UML détaillé de notre application | 46 |
| 31 | Diagramme de paquetages UML de notre application | 47 |
| 32 | Page <i>annotated.html</i> de la documentation | 48 |
| 33 | Menu de l'application | 49 |
| 34 | Grille niveau facile vide | 49 |
| 35 | Grille niveau normal vide | 50 |
| 36 | Grille niveau difficile vide | 50 |
| 37 | Révéler un mot avec l'interface | 51 |
| 38 | Grille niveau difficile révélée | 51 |
| 39 | Début du lancement en mode terminal | 52 |
| 40 | Fin du lancement en mode terminal | 52 |
| 41 | Temps de génération de grille avec des cases noires pré-placées | 55 |
| 42 | Temps de génération de grille sur des grilles vides aux dimensions variables | 56 |
| 43 | Résultat avec une grille de 6 x 6 sur la base de données de test - possibilité 1 | 58 |
| 44 | Résultat avec une grille de 6 x 6 sur la base de données de test - possibilité 2 | 58 |
| 45 | Résultat avec une grille de 9 x 8 sur la base de données de test | 58 |
| 46 | Bilan de performances avec la base de données de test | 59 |
| 47 | Bilan de performances avec la base de données de test (avec mots "parasites") | 60 |
| 48 | Bilan de performances en temps sur des grandes grilles | 62 |
| 49 | Bilan de performances en mémoire sur des grandes grilles | 63 |
| 50 | Grille témoin utilisée pour les mesures de variation de taille de la base de données | 64 |
| 51 | Comparaison de performances sur la grille témoin - variation de la taille de la base de données | 65 |

| | | |
|----|--|----|
| 52 | Etude algorithmique en milieu aquatique | 71 |
| 53 | Des performances largement inférieures à celles de notre solver | 72 |

1 Objet de la demande

Le projet à réaliser consiste en la réalisation d'un programme capable de générer automatiquement des grilles de mots-croisés, dont les difficultés sont variables. Outre la génération, l'utilisateur doit pouvoir y jouer.

1.1 Utilisateurs et parties prenantes du projet

1.1.1 En interne

Le projet est réalisé dans le cadre de l'UE "Projet de Programmation" en Master 1 à l'Université de Bordeaux. Le commanditaire est Monsieur Philippe Narbel et le projet est suivi par Madame Wahiba Larbi. L'équipe de développement du projet est composée de Albanel Clément, Alves Nathanael, Auzi Emma, Bornes Yohan, Coudougnan Sylvain et Devidas Hugo.

1.1.2 En externe

Les clients concernés par l'application développée sont des enseignants chercheurs travaillant au Laboratoire Bordelais de Recherche pour l'Informatique.

1.2 Description des objectifs généraux

Le but premier de l'application à développer est de construire un programme capable de générer automatiquement des grilles de mots croisés. Il s'agira d'un jeu traditionnel de mots croisés et nous accorderons une grande importance aux techniques algorithmiques de génération de mots croisés. Ainsi, nous focaliserons nos recherches selon ces 4 références principales :

- L'article universitaire [1] publié à l'Université de Stanford traite des problèmes appartenant à la classe algorithmique nommée "Problème de satisfaction de contraintes". Il s'agit de problèmes mathématiques où l'on cherche des états qui satisfont un ensemble de critères ou de contraintes données. Le problème des mots croisés en est un. En effet, générer une grille de jeu implique évidemment qu'elle soit résoluble, et donc il s'agira de notre contrainte principale (bien sûr, elle implique de nombreuses sous-contraintes). Ce papier universitaire expose des algorithmes et techniques utilisées pour la résolution des problèmes de satisfaction de contraintes, appliquée à la génération de mots croisés.
- Le second article [2] issu de l'Université Nationale Australienne est directement lié au précédent, puisqu'il présente les mots croisés en tant que problème de satisfaction de contraintes. Plus précisément, à travers l'article, les 2 auteurs présentent des nouvelles techniques de génération de mots croisés, bien plus efficaces que les techniques utilisées jusqu'alors. Le modèle présenté repose sur deux points de vue différents : l'un contenant des variables de cellules, et l'autre contenant des variables d'emplacement de mots. Ce modèle hybride permet d'exploiter au maximum les avantages de chacune des 2 techniques, indépendamment.
- Par ailleurs, le papier universitaire [3] se base sur la recherche de solutions via les arbres de décision. Cela permet, combiné à des heuristiques performantes d'aiguiller l'exploration de l'arbre et donc de trouver une solution plus rapidement à une grille donnée. Il a été montré que l'arbre de décisions influence largement l'efficacité de l'algorithme de recherche. L'idée est ici de montrer comment il est possible de construire l'arbre de décision, indépendamment de l'heuristique et ainsi d'obtenir de meilleures performances algorithmiques. Pour cela, le jeu des mots croisés est utilisé comme modèle.

- Enfin l'article de journal [4] écrit par Matthew L. Ginsberg aborde un solveur particulier, qui résout les grilles de mots croisés (variante Américaine). L'idée est de convertir le jeu des mots croisés en problème de satisfaction de contraintes et de ce fait de pouvoir utiliser des nouvelles techniques pour trouver des solutions.

Bilan des 4 articles : bien qu'il s'agisse essentiellement de problèmes abordants des solveurs, ils font néanmoins partie intégrante de la génération de grille, puisqu'une fois la grille générée, il s'agit de vérifier si cette dernière est résoluble.

2 Analyse de l'existant

2.1 Introduction

La plupart des outils que nous avons analysés sont des applications web et utilisent les technologies associées. Notre logiciel s'installera sur machine, cependant la plupart de ces outils gratuits et très faciles d'utilisation restent une source d'inspiration conséquente pour notre projet. Certains logiciels plus poussés sont également payants ou disposent d'options payantes. Ces outils sont plutôt similaires, mais des aspects comme la personnalisation de la grille ou l'interface utilisateur pourront être un peu plus poussées chez certains. Nous allons en présenter les principaux.

2.2 Outils en ligne

- The Teacher's Corner : puzzle and activities [5]

Il permet de générer des grilles pré-faites sur différents thèmes comme les fêtes, la géographie, les langues etc.. Ou bien de générer soi-même sa propre grille en rentrant nous même nos propres mots et leurs définitions associées. Il permet de grandes options sur l'affichage de la grille, comme le choix des polices, changement de la taille et des couleurs des textes ou des cellules, ajouter du texte ou des images. Une fois que le projet est prêt on peut choisir de l'exporter avec ou sans la solution inscrite dans la grille. Bien que l'on puisse directement remplir les champs, l'usage de cet outil semble destiné à exporter la grille en image ou en pdf dans le but d'une éventuelle impression. Cet outil est à destination d'une cible anglophone si l'usage principal est d'utiliser les templates existants.

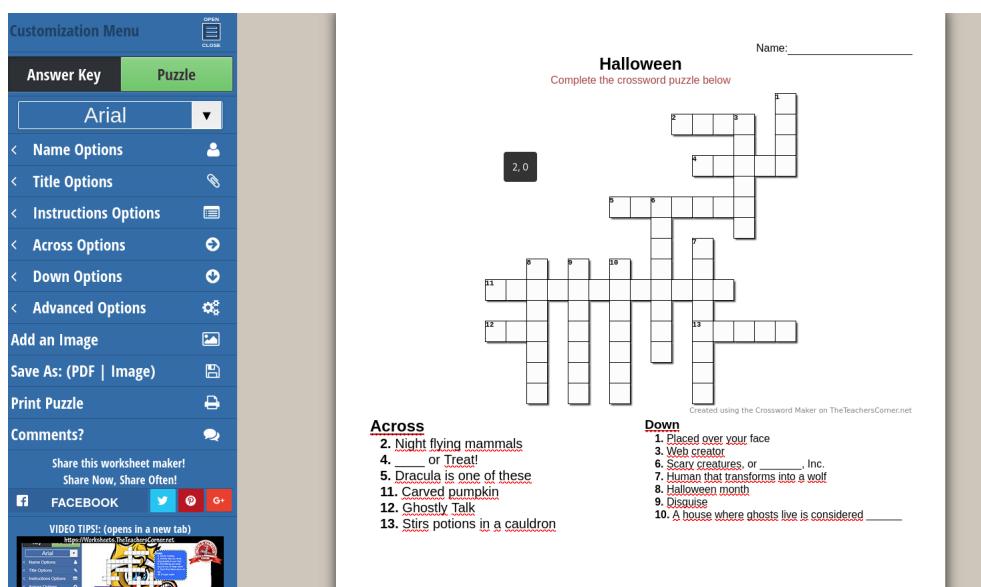


Figure 1: Interface Application "The Teacher's Corner : puzzle and activities"

- Sport Cérébral : mots croisés [6]

Cet outil permet de remplir une unique grille chaque jour dans le but de se divertir. Il ne permet aucune option de génération ou d'exportation. Cependant dans les fonctionnalités on a accès à des boutons indices et des options ergonomiques tels que le zoom, la possibilité d'effacer nos mots, etc pour enfin valider notre grille. D'un point de vue design et besoins utilisateurs, cette application est en adéquation avec le projet.

The screenshot shows the 'SPORT CÉRÉBRAL' website with a crossword puzzle interface. At the top, there's a navigation bar with links for Revues, Abonnements, Jeux en ligne, Concours, Informations, a search bar, and user account icons. Below the navigation is a toolbar with a magnifying glass icon for zoom, a trash can for clear, and a shopping cart icon for purchases.

The main area features a crossword grid with numbered squares (1-11 across, 1-17 down) and blacked-out squares. A status bar at the top of the grid says "Pas fait pour attacher." (Not made for attachment). The grid has numerical labels 1 through 11 along the bottom and 1 through 17 along the right side.

To the right of the grid, there are two sections: "Horizontal" and "Vertical".

Horizontal:

1 Pas fait pour attacher. 2 On lui serre la vis. Ne fleures pas bon. 3 Pièges. Système antiblocage. 4 Il attire par tous les moyens. 5 Distinction. Pour dans. 6 Vida l'éuelle. Malaria. 7 As ton origine. Ennuyer. 8 Chaîne à Rome. Oukase du tsar. 9 Direction. Démonstratif. 10 Famoux inspecteur. Fromage des Pays-Bas. 11 Pièces pour quêtes. 12 Partie de la Grande-Bretagne. Rejeté. 13 Grignotage. 14 Rêvassent. Avec un air d'interdit. 15 Su.

Vertical:

1 Trop tard. 2 Porta à la bouche. Teinte. 3 Valeur étonnant. Eclos. 4 Mesure thermique. 5 Personnage à genoux. Perdu. 6 Balancées en hauteur. 7 Manouche. Un point sur la carte. 8 Il prend place sur le lit. 9 Roi de Hongrie. Effaré, hébété. 10 Entre deux mots. Qui a vécu. 11 Il nous fait suer. 12 Documents. Réponse positive. 13 Parent. Pont de Paris. 14 Lieu de rencontres sportives. Point fort. 15 Fleur ou déchiffre. Elles se retrouvent dans un CV. 16

At the bottom of the grid, there are three buttons: Effacer (Clear), Indice (Hint), and Zoom.

Figure 2: Interface Application "Sport Cérébral : mots croisés"

- Education.com : Crossword Puzzle [7]

Outil permettant de générer ses propres mots croisés, avec une très légère option de personnalisation du design (bordures) et possibilité d'export/import de nos propres grilles.

Crossword Puzzle

Reading worksheet generator

Options

Theme

Enter crossword puzzle content

See instructions below. Under 50 words is best. More help: [help](#)

Title

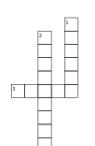
Titre test

Words

alligator, a reptile that looks a little like a crocodile
tiger, a big orange cat with black stripes
spider, an eight-legged bug that spins webs

WORKSHEET
ANSWER KEY

Titre test



Down:

- 1. an eight-legged bug that spins webs
- 2. a reptile that looks a little like a crocodile

Across:

- 3. a big orange cat with black stripes

[Scramble](#)

[Download Worksheet](#)

[Download Answer Key](#)

Save worksheetSave

Figure 3: Interface Application "Education.com : Crossword Puzzle"

- Puzzel [8]

Outil à l'interface soignée, beaucoup d'options ne sont malheureusement accessibles qu'avec une participation financière.

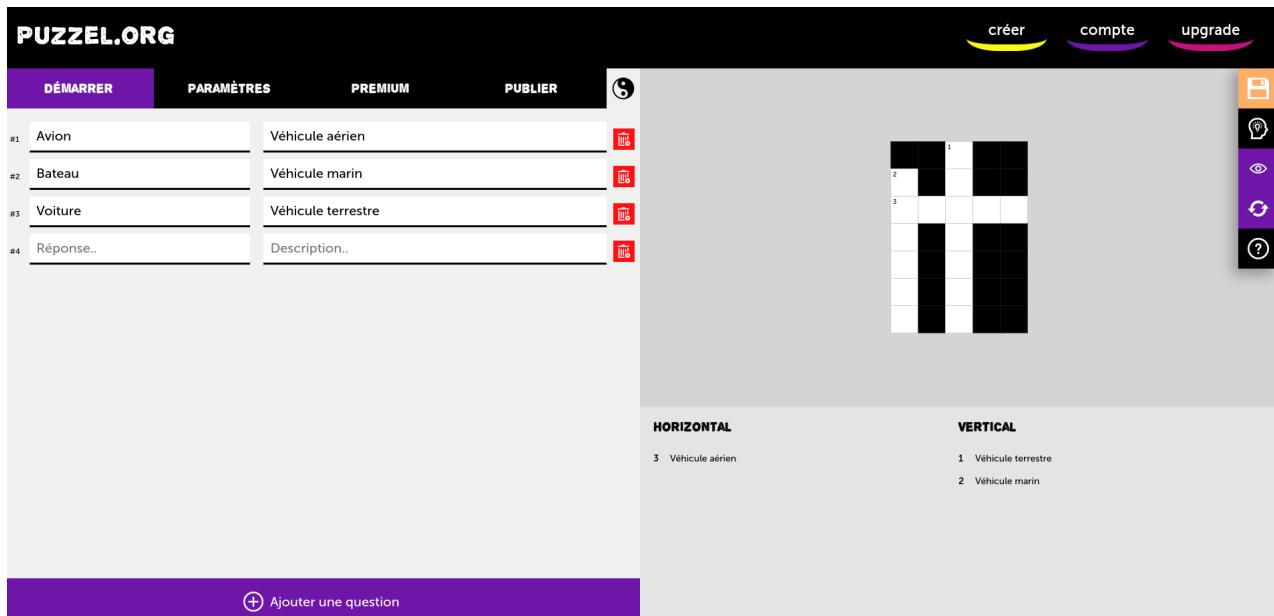


Figure 4: Interface Application "Puzzel"

2.3 Logiciel nécessitant une installation

- EclipseCrossword[9]

Disponible sur le catalogue d'applications Microsoft. Ce logiciel est gratuit et disponible uniquement sur Windows.

Logiciel complet de création de grille. Possibilité d'importer une liste de mots/indices. Possibilité d'enregistrer sa grille dans le format du logiciel (.ecw) et sa "word list", d'imprimer avec plusieurs options (présence des réponses dans la grille, afficher les indices, etc...) et partager sa grille en ligne. On trouve également des options plus avancées comme des exports html, rtf, wmf, eps et "across lite text format".

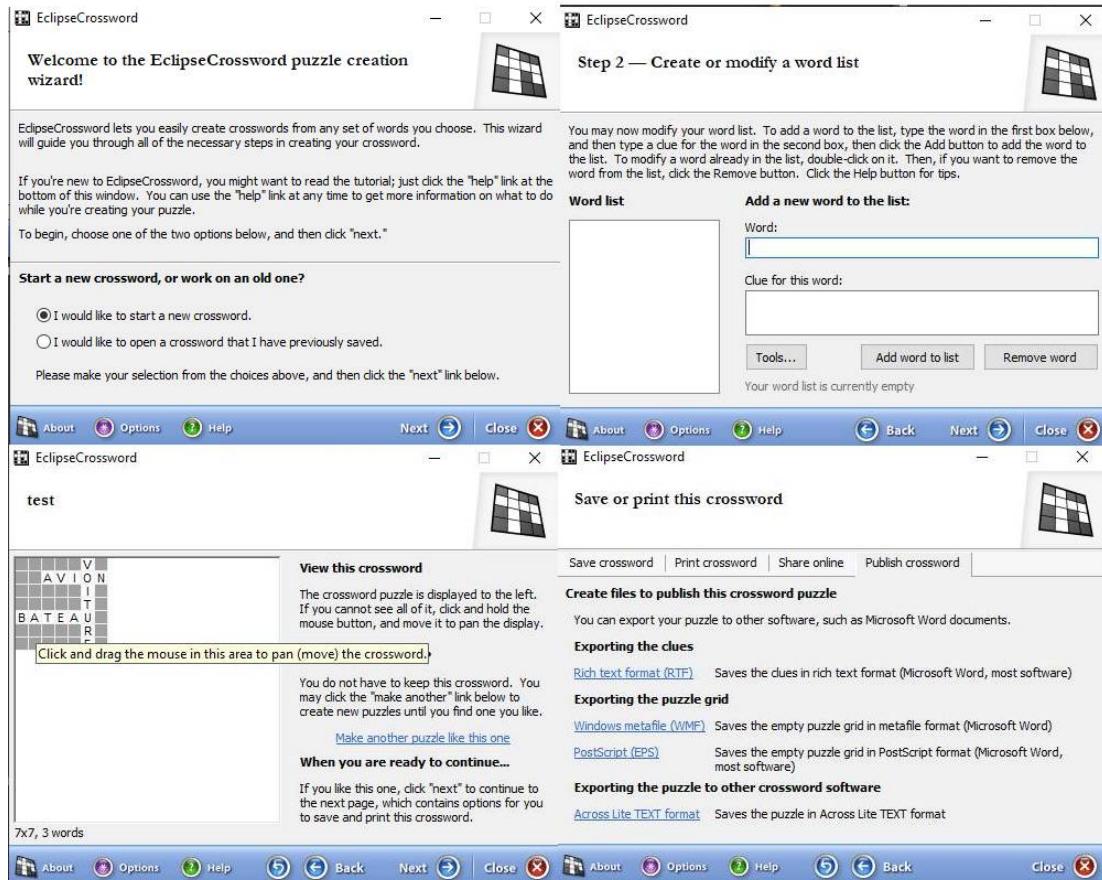


Figure 5: Interface Application "Eclipse Crossword"

- Mots-croisés-Pro [10]

Logiciel payant disponible uniquement sur Windows au prix de 59 euros. Une version gratuite d'essai est disponible.

Ce logiciel permet la création de grilles de mots croisés ou fléchés jusqu'à 400 cases. Il permet de créer des grilles de manière automatique, semi-automatique ou manuelle. Outil extrêmement complet, mais malheureusement limité dans sa version gratuite.

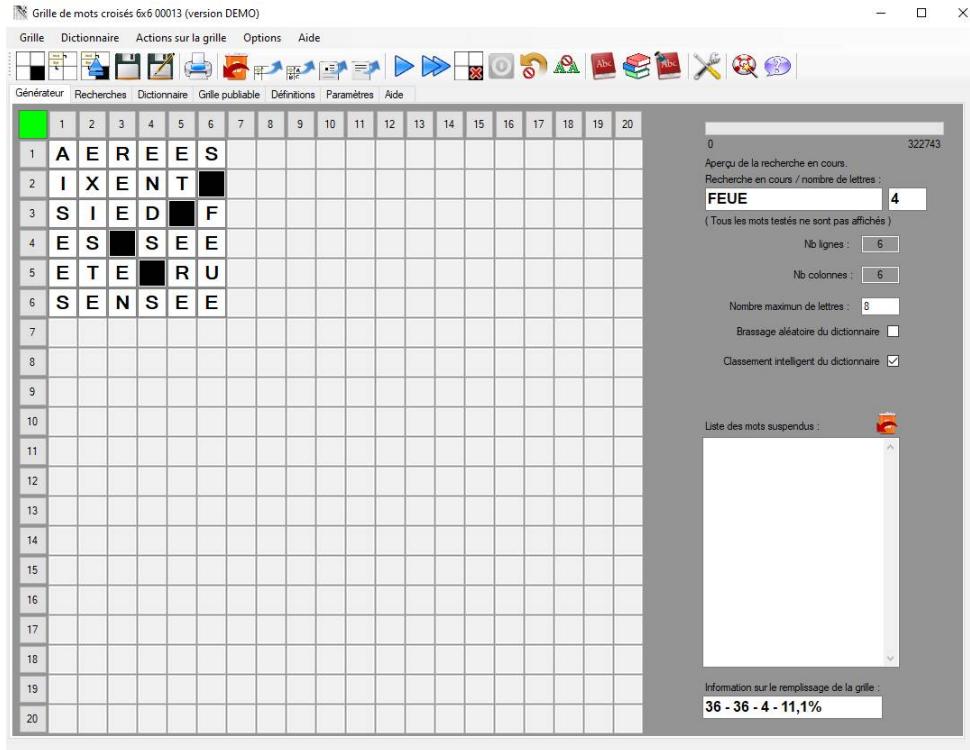


Figure 6: Interface Application "Mots croisés pro"

2.4 Synthèse

La plupart des générateurs facilement trouvables en ligne permettent de créer des grilles très simples, la cible de ces générateurs sont probablement des enfants ou un public non expert des mots-croisés, l'utilisation de tels outils semble adaptée à un but éducatif. En revanche aucun des ces outils ne semble proposer de solutions plus avancées telles que l'on pourrait trouver dans des journaux ou magazines spécialisés. En effet si l'on cherche à construire une grille complexe avec peu de cases noires (absence de lettres) et des normes précises comme une taille de grille particulière ou le respect de la symétrie (voir figure plus bas), il faut nous pencher vers des logiciels plus avancés et souvent payants, comme le fait Mots-croisés-Pro. De plus ces logiciels sont davantage une aide à la construction manuelle de grille, qu'une solution toute faite automatisée.

Dans notre application, les options de personnalisation poussées ne seront pas prioritaires, elles viendront enrichir notre logiciel que dans un second temps. Nous voulons nous inspirer des options d'aide à la résolution de grille pour l'utilisateur comme ceux présents dans Sport Cérébral par exemple. Nous ne prévoyons pas d'utiliser le web pour notre application (pour ne pas nous préoccuper des conventions du web telles que les requêtes HTTP), ainsi l'interface utilisateur devrait être bien différente des générateurs en ligne que l'on a pu étudier ici. Le partage et la publication de grilles ne nous concerne pas particulièrement, les options d'imports/exports à l'usage d'un seul utilisateur nous suffisent.

3 Besoins fonctionnels

3.1 Avant-propos

Nous utiliserons les codes de correspondance suivants au sein des besoins :

- **(E)** : Essentiel → le logiciel ne sera pas acceptable sans que ce besoin soit réalisé.
- **(C)** : Conditionnel → besoin qui étend et améliore le logiciel, sans que ce besoin soit nécessaire pour rendre le logiciel acceptable.
- **(O)** : Optionnel → besoin dont la valeur n'est pas encore assurée.

Pour chacun des besoins présentés, nous écrirons ici 3 différents tests :

- test **positif** (ou test d'affirmation) qui vérifie le bon fonctionnement du besoin dans des conditions d'utilisation correctes (qui ne "stressent" pas les cas d'utilisation du logiciel).
- test **négatif** qui vérifie le comportement hors des cas d'utilisation définis par les besoins du logiciel.
- test **aux frontières**, ils s'appliquent aux limites/frontières de fonctionnement normal. Ils sont donc situés à la frontière entre les cas d'utilisation entraînant un test positif, et ceux entraînant un test négatif. A noter que ce test ne peut pas s'appliquer à tous les besoins, nous ne le mettrons donc pas toujours (comme parfois les tests négatifs).

Pour chacun des tests énoncés ci-après, nous énoncerons d'abord le nom du besoin tel qu'exprimé précisément dans le cahier des besoins, puis nous écrirons les 3 tests selon l'ordre expliqué en introduction. Nous séparerons chacun des 3 tests selon la charge graphique suivante : fond bleu = test positif / fond rouge = test négatif / fond gris = test aux frontières. A noter que sur certains besoins, nous n'avons pas trouvé de tests négatifs et aux frontières pertinents.

3.2 Diagramme des cas d'utilisation

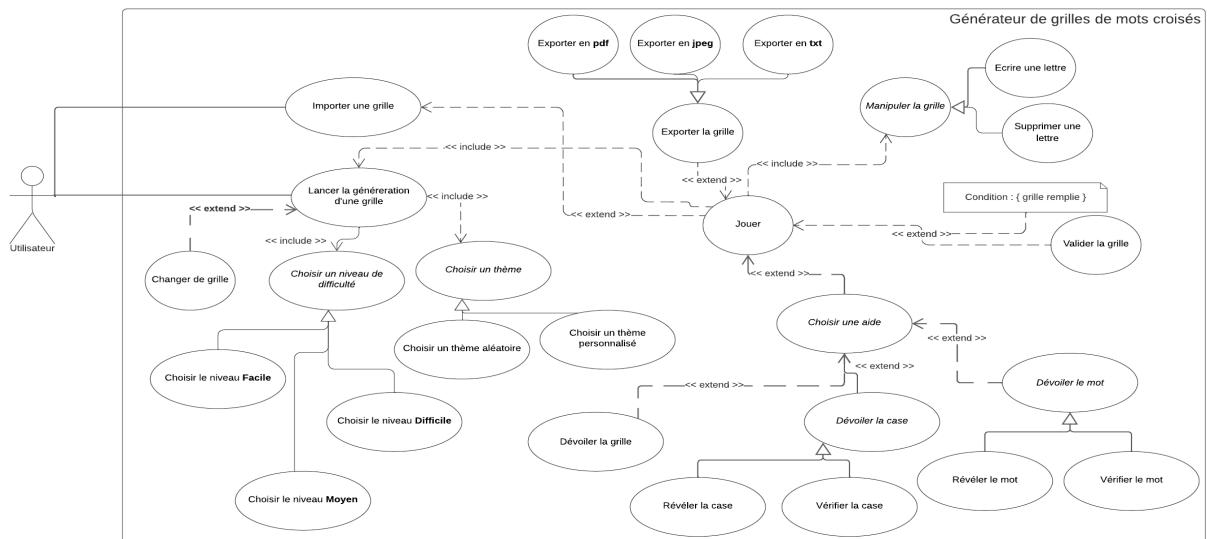


Figure 7: Diagramme de cas d'utilisation

Le besoin principal de notre application est naturellement de pouvoir générer une grille en y appliquant un niveau de difficulté et de permettre à l'utilisateur d'y jouer. Aussi, l'utilisateur doit impérativement pouvoir bénéficier d'aides diverses l'aidant à la résolution de la grille, il pourra révéler ou vérifier une case ou un mot ou dévoiler la grille entièrement. Lorsque la grille est remplie il pourra valider ses réponses. Il est enfin important dans notre application que l'on puisse arrêter à tout moment une partie en cours et pouvoir la reprendre quand l'utilisateur le souhaite : import et export d'une grille. C'est d'ailleurs un des points qui nous a semblé manquant sur l'existant étudié.

3.3 Besoins utilisateurs

(O) Importer une grille

Acteur : Utilisateur.

Résumé : L'utilisateur souhaite importer une grille spécifique sur l'application pour y jouer.

Préconditions : L'application est correctement lancée.

Post conditions : L'utilisateur peut jouer sur la grille qu'il a importée.

Enchaînement nominal :

| Étapes | Utilisateur | Système | Maquette correspondante |
|--------|--------------------------------------|---|---|
| 1 | Clique sur "File/Import" | | Maquette - Accueil |
| 2 | Sélectionne le fichier correspondant | | Maquette - Menu d'importation |
| 3 | | Vérifie la validité du fichier (cf. codage d'une grille) | / |
| 4 | | Affiche la grille ou une erreur d'importation | Maquette - Grille précédemment remplie importée |

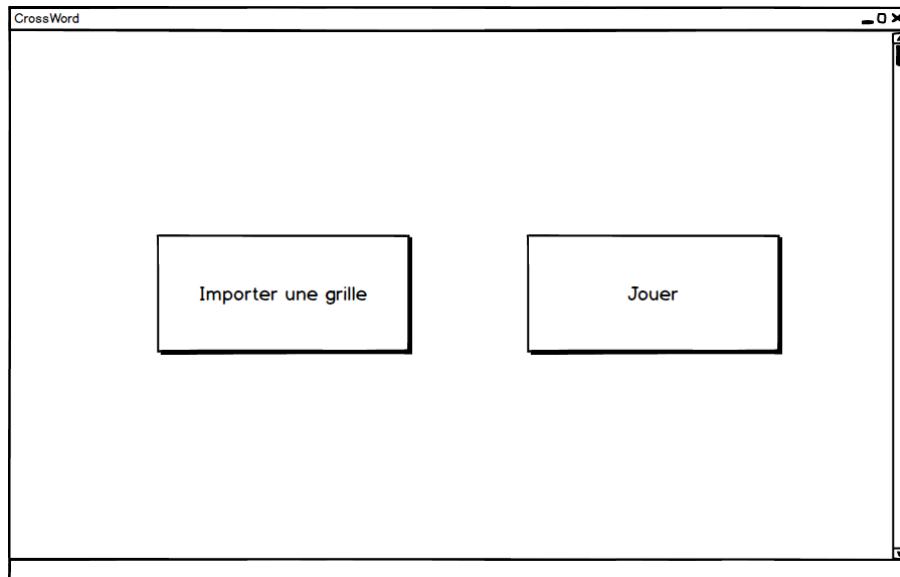


Figure 8: Maquette - Accueil

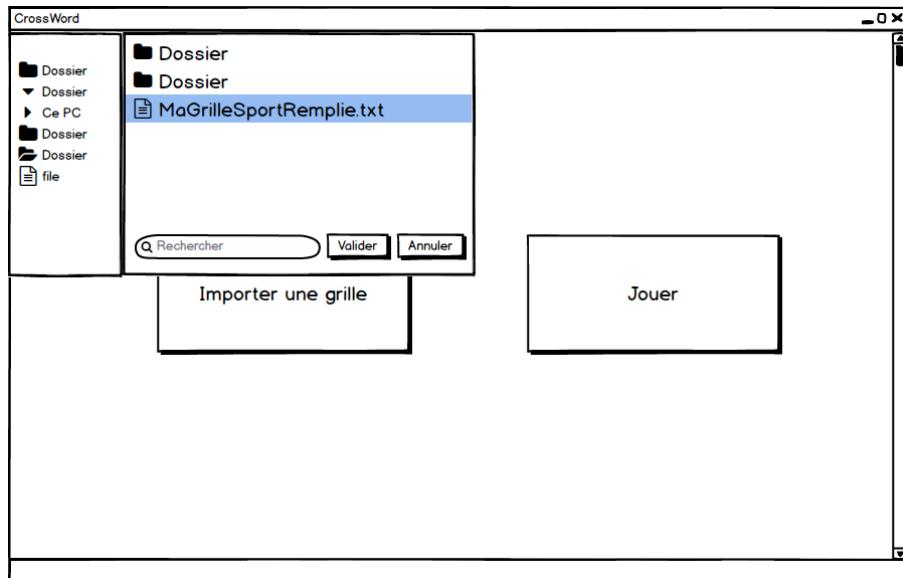


Figure 9: Maquette - Menu d'importation

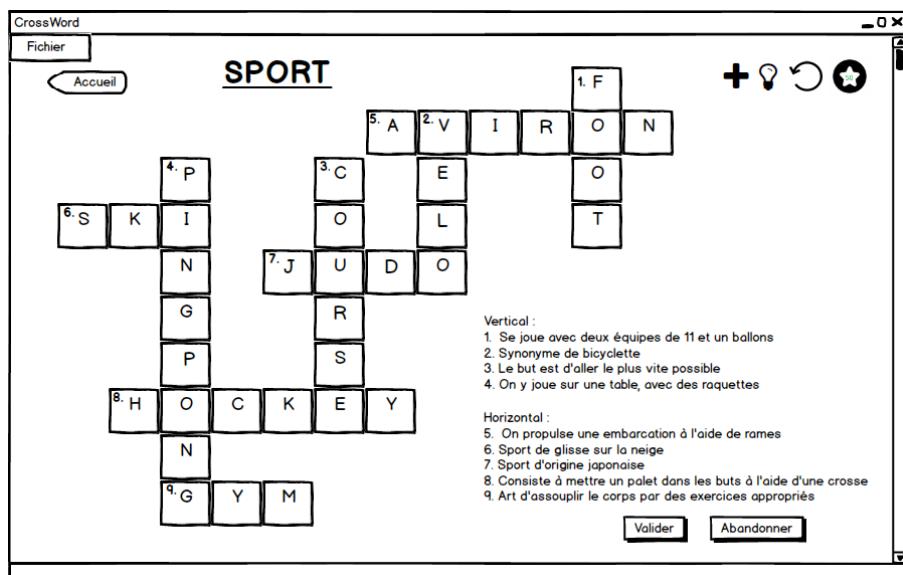


Figure 10: Maquette - Grille précédemment remplie importée

Codage d'une grille : La grille sera codée dans un fichier .txt, nous devrons stocker les différents mots et définitions pour que le système arrive à régénérer correctement la grille. Le format précis du codage sera précisé lors des spécifications techniques, si le fichier .txt ne respecte pas le codage, le système retournera une erreur lors de l'importation.

Tests associés :

1. *Description et but du test* : Importation d'une sauvegarde valide d'une grille.
2. *Cas de test* : On utilise la sauvegarde d'un grille "témoin" valide. On importe cette grille sauvegardée et on vérifie que l'importation de cette dernière a bien été effectuée.
3. *Déroulement et scénario du test* : On sauvegarde une grille sur notre machine, on l'importe et on vérifie que la grille importée est la même que la grille fraîchement sauvegardée.
4. *Analyse du test* : Si la grille importée est bien celle attendu alors le test est valide, invalide sinon.

1. *Description et but du test* : Importation d'une sauvegarde non valide d'une grille (format de fichier non conforme).
2. *Cas de test* : Un fichier de sauvegarde invalide d'une grille. Il est attendu que le logiciel ne se charge pas et qu'un message d'erreur s'affiche.
3. *Déroulement et scénario du test* : On importe la sauvegarde invalide dans notre logiciel avec l'aide du bouton "Importer une grille".
4. *Analyse du test* : Si un message d'erreur indiquant que le fichier est invalide s'affiche alors le test est valide, sinon il est échoué.

(E) Choisir un niveau de difficulté

Acteur : Utilisateur.

Résumé : L'utilisateur peut choisir une difficulté pour la grille de mots croisés (il se rend dans "play/select difficulty/difficulty wanted").

Préconditions : L'application est correctement lancée et l'utilisateur a choisi de commencer à jouer.

Post conditions : L'utilisateur a la liberté de jouer selon différentes difficultés.

Maquette :

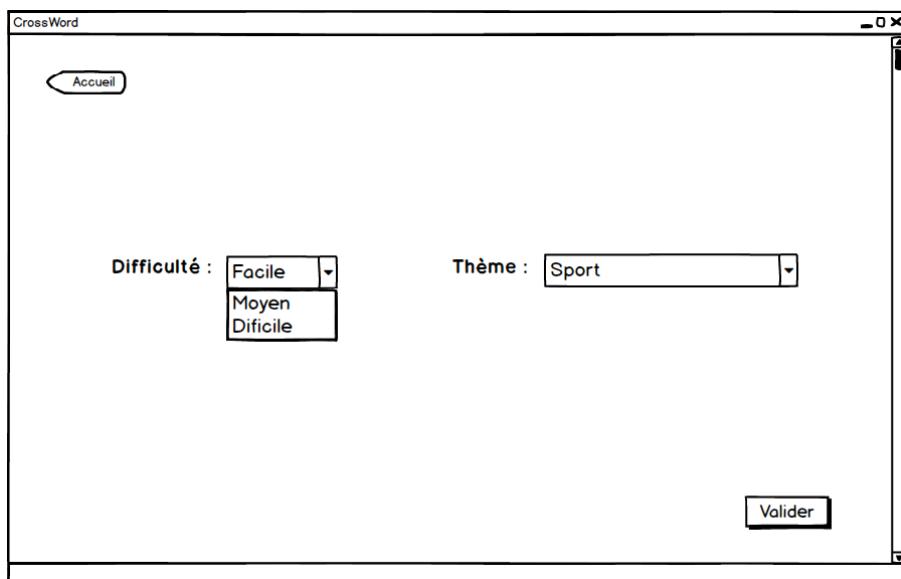


Figure 11: Maquette - Choix de la difficulté

Les différentes difficultés : La difficulté d'une grille sera définie par la taille de la grille et le nombre d'aides autorisées au joueur. Nous aurons 3 niveaux de difficultés : Facile (grille 6x6), Moyen (grille 8x8) et Difficile (grille 10x10).

Test associé :

- Description et but du test* : Vérifier que la génération des grilles est bien modifiée en fonction du niveau de difficulté.
- Cas de test* : Utilisation du logiciel témoin (sans condition de stress). Le résultat attendu est que la grille soit générée avec la méthode qui représente la difficulté sélectionnée.
- Déroulement et scénario du test* : On génère la grille avec chaque difficulté possible, à chaque génération on vérifie que la grille sélectionnée est générée avec la méthode reliée correspondant à sa difficulté.
- Analyse du test* : Si chaque grille a été générée avec la méthode correspondant à son niveau de difficulté alors le test est validé sinon il a échoué.

(O) Choisir un thème

Acteur : Utilisateur.

Résumé : L'utilisateur peut choisir un thème pour la grille de mots croisés.

Préconditions : L'application est correctement lancée et l'utilisateur à choisi de commencer à jouer.

Post conditions : L'utilisateur a la liberté de jouer sur des grilles de différents thèmes.

Maquette :

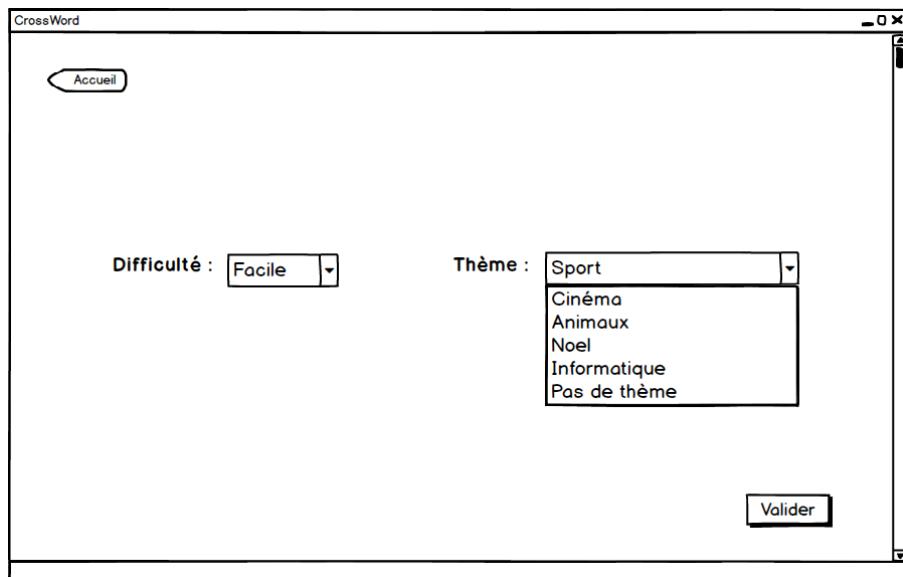


Figure 12: Maquette - Choix du thème

Les différents thèmes : Les thèmes influenceront les mots qui composent la grille, par exemple un thème de sport donnera une grille avec des mots du lexique du sport. Les thèmes auxquels nous avons pensé sont : Sport, Cinéma, Animaux, Noël, Informatique.

Test associé :

1. *Description et but du test* : Vérifier que les mots utilisés pour créer la grille viennent de la liste de mots correspondant au thème.
2. *Cas de test* : Utilisation du logiciel témoin (sans condition de stress) et des listes de mots des différents thèmes. On s'attendait à ce que les grilles générées avec un thème n'utilisent que des mots de la liste de son thème.
3. *Déroulement et scénario du test* : On génère la grille avec chaque thème et pour chaque grille on vérifie que seuls les mots de leur thème ne composent la grille.
4. *Analyse du test* : Si chaque grille n'est composée que des mots correspondant à son thème alors le test est validé, sinon le test échoue.

(C) Changer de grille

Acteur : Utilisateur.

Résumé : L'utilisateur n'est pas satisfait d'une grille et veut donc changer de grille.

Préconditions : L'application est correctement lancée, l'utilisateur est en train de jouer sur une grille.

Post conditions : La nouvelle grille est affichée, l'utilisateur peut maintenant jouer.

Enchaînement nominal :

| Étapes | Utilisateur | Système | Maquette correspondante |
|--------|--|---------------------------------|-------------------------------------|
| 1 | Clique sur "Changer de grille" | | (Bouton + de l'écran de jeu) |
| 2 | | Affiche les différentes options | Maquette - Paramétrage de la grille |
| 3 | Choisi une difficulté (cf. "Choisir une difficulté") | | (cf. "Choisir une difficulté") |
| 4 | Choisi un thème (cf. "Choisir un thème") | | (cf. "Choisir un thème") |
| 5 | Valide sa sélection | | Maquette - Paramétrage de la grille |
| 6 | | Génère la grille | / |
| 7 | | Affiche la grille | Maquette - Grille vide |

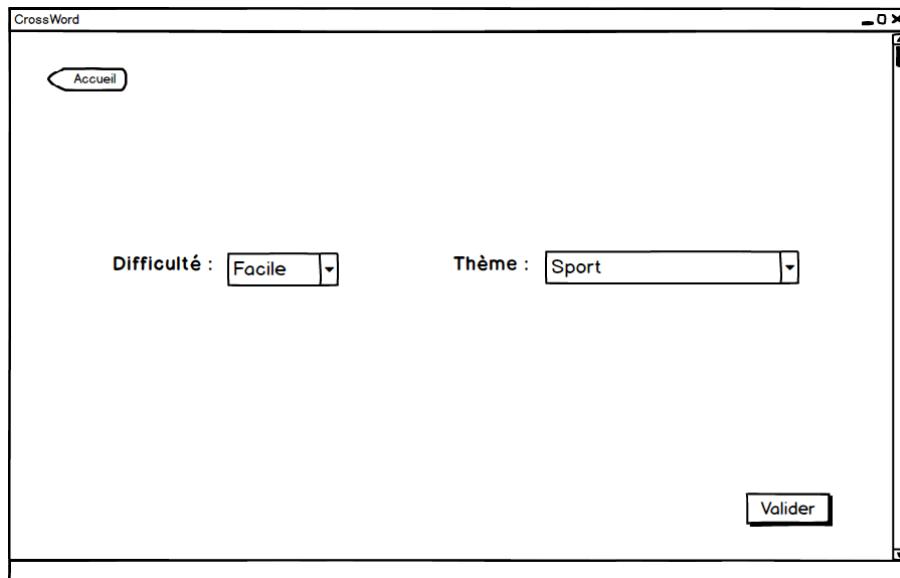


Figure 13: Maquette - Paramétrage de la grille

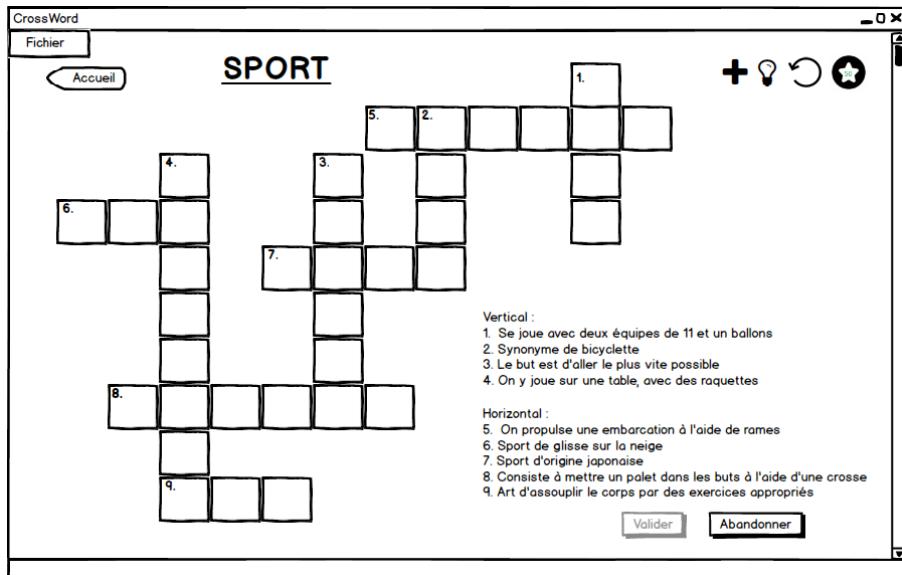


Figure 14: Maquette - Grille vide

Tests associés :

1. *Description et but du test* : Vérification que le bouton changer de grille génère bien une nouvelle grille.
2. *Cas de test* : Utilisation du logiciel témoin (sans condition de stress) . On s'attend à ce que le logiciel génère une nouvelle grille.
3. *Déroulement et scénario du test* : On clique sur le bouton "Changer de grille".
4. *Analyse du test* : On génère une nouvelle grille. On vérifie que cette dernière n'est composée que de mots appartenant au thème de l'ancienne grille.

1. *Description et but du test* : Vérification que le bouton changer de grille n'est pas disponible si aucune grille n'est générée.
2. *Cas de test* : Utilisation du logiciel témoin (sans condition de stress) . On s'attend à ce que le bouton de changement de grille ne soit pas affiché.
3. *Déroulement et scénario du test* : On lance l'application et on vérifie que le bouton n'est pas présent.
4. *Analyse du test* : Le test est un succès si le bouton n'est pas affiché.

(E) Lancer la génération d'une grille

Acteur : Utilisateur.

Résumé : L'utilisateur lance la génération d'une grille de mots croisés en vue d'y jouer (détailé sur le PDF Algorithmes, choix techniques, outils utilisés).

Préconditions : L'application est correctement lancée.

Post conditions : La grille est affichée, l'utilisateur peut maintenant jouer.

Enchaînement nominal :

| Étapes | Utilisateur | Système | Maquette correspondante |
|--------|--|---------------------------------|---|
| 1 | Clique sur "Jouer" | | (cf. Maquette - Accueil) |
| 2 | | Affiche les différentes options | (cf. Maquette - Paramétrage de la grille) |
| 3 | Choisit une difficulté (cf. "Choisir une difficulté") | | (cf. "Choisir une difficulté") |
| 4 | Choisit un thème (cf. "Choisir un thème") | | (cf. "Choisir un thème") |
| 5 | Valide sa sélection | | (cf. Maquette - Paramétrage de la grille) |
| 6 | | Génère la grille | / |
| 7 | | Affiche la grille | (cf. Maquette - Grille vide) |

Test associé :

- Description et but du test* : Vérifier que la génération de la grille fonctionne.
- Cas de test* : Génération de la grille en fonction du thème et de la difficulté choisie par l'utilisateur.
- Déroulement et scénario du test* : Sélection d'un thème et d'une difficulté, valider la sélection.
- Analyse du test* : Vérifier que la grille est bien solvable par l'utilisation de notre solveur. Vérification de la bonne appartenance des mots de la grille au thème et à la difficulté sélectionnée préalablement.

(E) Jouer (remplir les cases)

Acteur : Utilisateur.

Résumé : L'utilisateur veut jouer à une grille de mots croisés.

Préconditions : L'application est correctement lancée.

Post conditions : La grille est affichée, l'utilisateur peut maintenant jouer.

Enchaînement nominal :

| Étapes | Utilisateur | Système | Maquette correspondante |
|--------|---|---------------------------------------|------------------------------|
| 1 | Sélectionne une case | | (cf. Maquette - Grille vide) |
| 2 | | Prend en compte la case sélectionnée | (cf. Maquette - Grille vide) |
| 3 | Rentre une nouvelle valeur (au clavier) | | / |
| 4 | | Affiche la nouvelle valeur de la case | Maquette - Partie en cours |

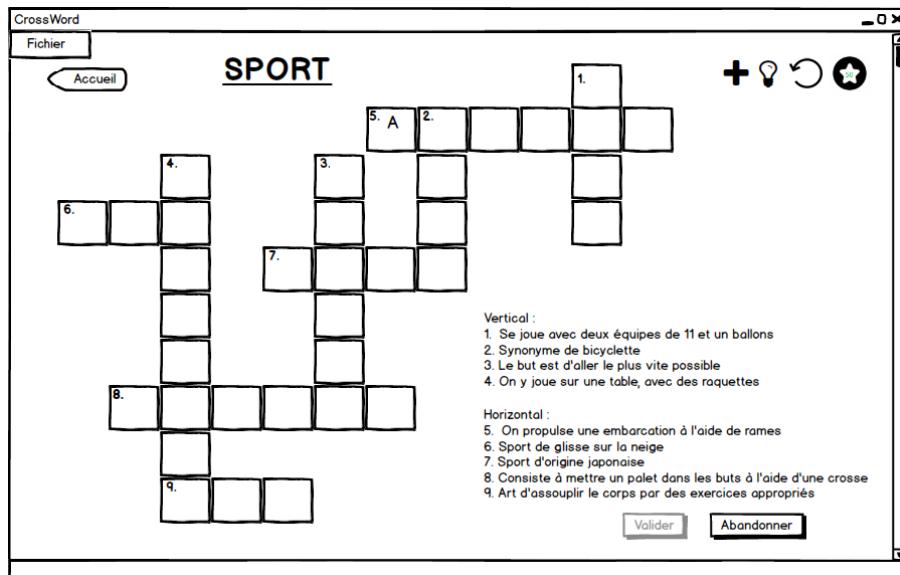


Figure 15: Maquette - Partie en cours

Test associé :

1. *Description et but du test* : L'utilisateur doit pouvoir remplir les cases de la grille par des lettres $\in [a - zA - Z]$.
2. *Cas de test* : Ecrire dans une case avec différents caractères.
3. *Déroulement et scénario du test* : A partir d'une nouvelle grille, essayer de remplir des cases avec des caractères acceptés par le programme et par des caractères spéciaux ou majuscules (non acceptés)
4. *Analyse du test* : Vérifier que les caractères spéciaux et majuscules ne sont pas entrés dans la case sélectionnée. Seuls les caractères $\in [a - zA - Z]$ seront acceptés.

(O) Exporter la grille

Acteur : Utilisateur.

Résumé : L'utilisateur veut exporter une grille de mots croisés.

Préconditions : L'application est correctement lancée, une partie est lancée, la grille est correctement affichée.

Post conditions : L'utilisateur possède la grille sous un format externe (pdf, jpeg ou txt).

Enchaînement nominal :

| Étapes | Utilisateur | Système | Maquette correspondante |
|--------|--|--|-------------------------|
| 1 | Selectionne "Export" dans le menu File | | Maquette - Menu fichier |
| 2 | Choisit le format (pdf, jpeg ou txt) | | Maquette - Menu export |
| 3 | | Génère le fichier au format demandé (cf. ci-dessous explication des différents formats) | / |
| 4 | Enregistre le fichier | | / |

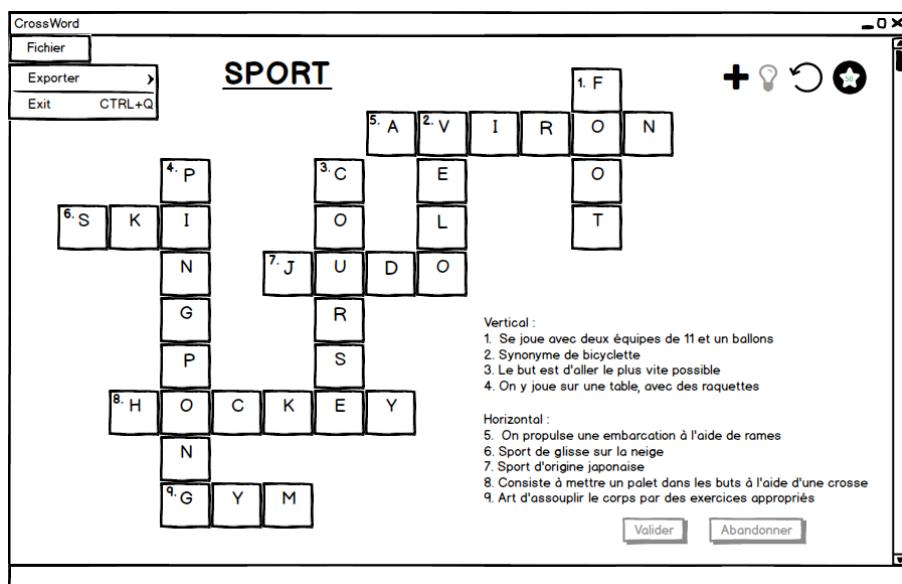


Figure 16: Maquette - Menu fichier

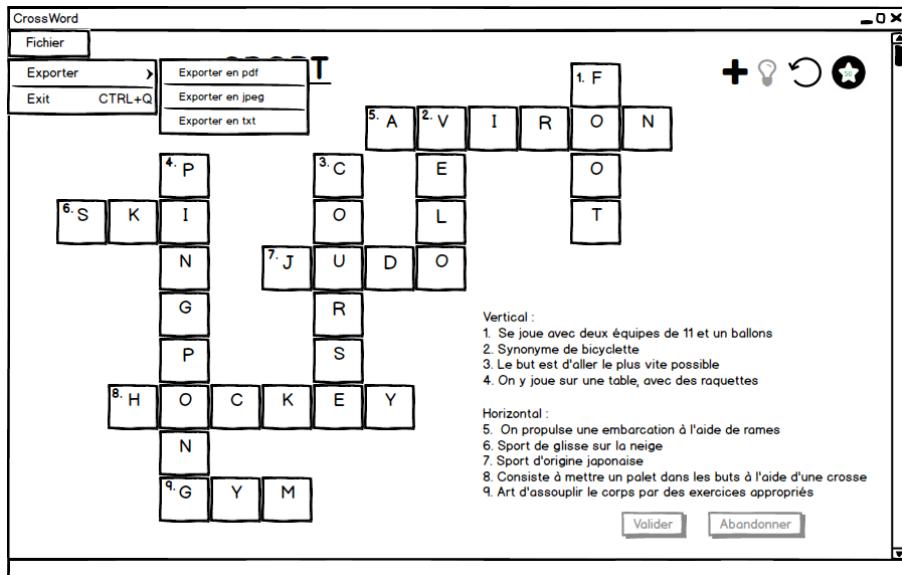


Figure 17: Maquette - Menu export

Format pdf : Le format pdf est mis à disposition en vue d'une impression, la grille est affichée de manière classique tel qu'elle était au moment de l'importation.

Format jpeg : Le format jpeg est mis à disposition en vue d'un partage, la grille est affichée de manière classique telle qu'elle était au moment de l'importation.

Format txt : Le format txt est mis à disposition en vue d'une importation future de la grille. La grille est codée (cf. "Importer une grille" pour la description du codage).

Test associé :

1. *Description et but du test* : Sauvegarde de la grille en cours avec une possibilité de choix de trois extensions différentes : .PDF, .JPG, .TXT .
2. *Cas de test* : Avec l'utilisation d'une grille "témoin", cette dernière doit être sauvegardée sous l'extension choisie par l'utilisateur, à l'emplacement choisi par l'utilisateur grâce à l'ouverture de l'explorateur. Cette grille doit pouvoir être rechargée par notre application afin que l'utilisateur puisse poursuivre sa partie.
3. *Déroulement et scénario du test* : L'utilisateur clique sur le "Fichier" puis "Exporter" et il choisit l'extension ".TXT" parmi les trois proposées. L'explorateur de fichier s'ouvre afin que l'utilisateur puisse sélectionner son emplacement de sauvegarde. Une fois la grille sauvegardée, il devra pouvoir la recharger(cf. voir besoin "Importer une grille").
4. *Analyse du test* : Une comparaison de fichier entre une grille temoin sauvegardée en format .TXT par l'utilisateur à un moment défini X du jeu et une grille temoin (en guise de test Oracle) définie préalablement. Les deux fichiers comparés ne doivent montrer aucune différence.

(C) Choisir une aide

Acteur : Utilisateur.

Résumé : L'utilisateur choisit d'être aidé pour se débloquer.

Préconditions : L'application est correctement lancé (affichage du menu de selection des paramètres), une partie est commencée et la grille est correctement affichée.

Post conditions : Une aide a été affichée dans la grille.

Enchaînement nominal "Révéler la case" :

| Étapes | Utilisateur | Système | Maquette correspondante |
|--------|-------------------------------|--|-------------------------|
| 1 | Selectionne une case | | / |
| 2 | | Prend en compte la case sélectionnée | / |
| 3 | Clique sur "Choisir une aide" | | / |
| 4 | | Affiche les différentes aides | Maquette - Aides |
| 5 | Choisit "Révéler la case" | | / |
| 6 | | Affiche la lettre de la case correspondante | / |
| 7 | | Actualise les points d'aide de l'utilisateur (cf. explication des points d'aide) | / |

Enchaînement nominal "Révéler le mot" :

| Étapes | Utilisateur | Système | Maquette correspondante |
|--------|-------------------------------|--|-------------------------|
| 1 | Sélectionne une définition | | / |
| 2 | | Prend en compte la définition sélectionnée | / |
| 3 | Clique sur "Choisir une aide" | | / |
| 4 | | Affiche les différentes aides | Maquette - Aides |
| 5 | Choisit "Révéler le mot" | | / |
| 6 | | Affiche le mot correspondant dans la grille | / |
| 7 | | Actualise les points d'aide de l'utilisateur (cf. explication des points d'aide) | / |

Enchaînement nominal "Vérifier la case" :

| Étapes | Utilisateur | Système | Maquette correspondante |
|--------|-------------------------------|--|-------------------------|
| 1 | Sélectionne une case | | / |
| 2 | | Prend en compte la case sélectionnée | / |
| 3 | Clique sur "Choisir une aide" | | / |
| 4 | | Affiche les différentes aides | Maquette - Aides |
| 5 | Choisit "Vérifier la case" | | / |
| 6 | | Affiche si la lettre est bonne ou mauvaise | / |
| 7 | | Actualise les points d'aide de l'utilisateur (cf. explication des points d'aide) | / |

Enchaînement nominal "Vérifier le mot" :

| Étapes | Utilisateur | Système | Maquette correspondante |
|--------|-------------------------------|--|-------------------------|
| 1 | Sélectionne une définition | | / |
| 2 | | Prend en compte la définition sélectionnée | / |
| 3 | Clique sur "Choisir une aide" | | / |
| 4 | | Affiche les différentes aides | Maquette - Aides |
| 5 | Choisit "Vérifier le mot" | | / |
| 6 | | Affiche si le mot est bon ou mauvais | / |
| 7 | | Actualise les points d'aide de l'utilisateur (cf. explication des points d'aide) | / |

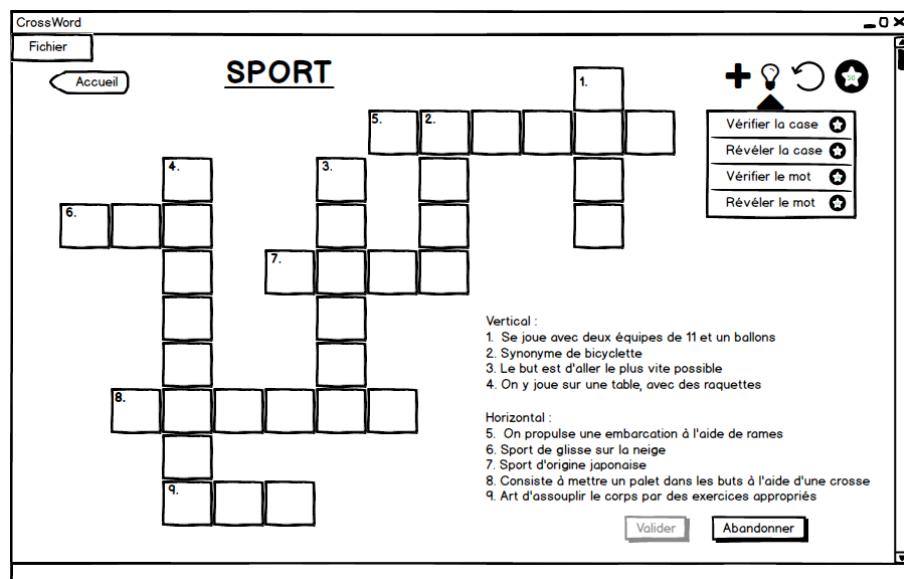


Figure 18: Maquette - Aides

Explication des points d'aide : Lors d'une partie, l'utilisateur aura tant de points (ce nombre de points sera fixé par la difficulté de la grille). A chaque utilisation d'une aide, l'utilisateur perdra des points, selon la "force" de l'aide plus ou moins de points seront enlevés : Vérifier la case < Révéler la case < Vérifier le mot < Révéler le mot. Si l'utilisateur n'a plus de points, il ne peut plus utiliser d'aides.

Tests associés :

1. *Description et but du test* : Bon fonctionnement des options d'aide.
2. *Cas de test* : A partir d'une grille non finie, l'utilisateur doit pouvoir sélectionner les différentes aides proposées par notre programme si son nombre de point le permet.
3. *Déroulement et scénario du test* : L'utilisateur choisit d'utiliser l'aide "réveler le mot", "révéler la case", "vérifier la case", "vérifier le mot".
4. *Analyse du test* : Vérifier que l'aide sélectionnée fonctionne correctement et que les points associés à l'utilisation de cette aide sont bien décomptés aux points restants à l'utilisateur.

1. *Description et but du test* : Utilisation d'une aide avec un nombre de points insuffisant.
2. *Cas de test* : L'utilisateur souhaite utiliser une aide mais n'a pas assez de points. Son utilisation doit être refusée
3. *Déroulement et scénario du test* : Cliquer sur l'utilisation d'une aide avec un nombre de points inférieur à celui demandé pour utiliser l'aide
4. *Analyse du test* : Vérifier que l'aide n'ait pas été exécutée.

(C) Dévoiler la grille - abandonner

Acteur : Utilisateur.

Résumé : L'utilisateur veut voir la solution complète, même s'il n'a pas commencé à remplir la grille.

Préconditions : L'application est correctement lancée, la grille a été générée et affichée.

Post conditions : La solution de la grille est affichée, et l'utilisateur n'aura plus la possibilité de valider.

Enchaînement nominal :

| Étapes | Utilisateur | Système | Maquette correspondante |
|--------|-----------------------------------|-----------------------------------|---|
| 1 | Clique sur le bouton "Abandonner" | | (cf. Maquette - Grille vide, Bouton "Abandonner") |
| 2 | | Affiche la grille remplie | Maquette - Solution |
| 3 | | Bloque la validation de la grille | |

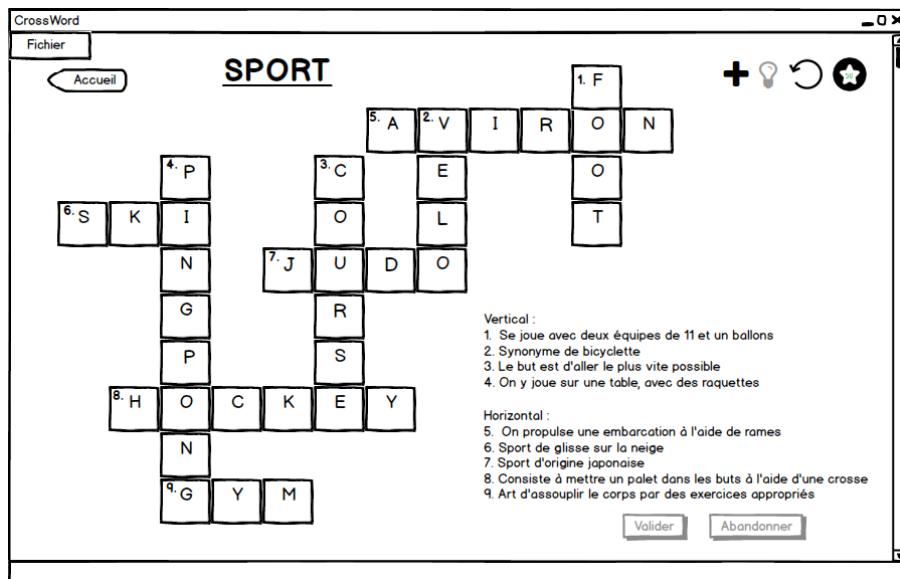


Figure 19: Maquette - Solution

Tests associés :

1. *Description et but du test* : Dévoiler toutes les cases restantes de la grille .
2. *Cas de test* : A partir d'une grille non finie, l'utilisateur doit pouvoir dévoiler l'entièreté de la grille. Chaque case non dévoilée sera remplacée par la lettre associée.
3. *Déroulement et scénario du test* : L'utilisateur clique sur le bouton "Abandonner". Chaque case est remplacée par la lettre associée.
4. *Analyse du test* : Vérifie que toutes les cases de la grille ont bien une lettre associée. Comparaison de la grille dévoilée "temoin" et une grille construite préalablement (test Oracle).

1. *Description et but du test* : Dévoiler la grille sur une grille finie.
2. *Cas de test* : A partir d'une grille finie et correctement complétée, l'utilisateur a toujours accès au bouton "Abandonner".
3. *Déroulement et scénario du test* : L'utilisateur a correctement complété la grille mais le bouton "Abandonner" est toujours visible.
4. *Analyse du test* : Vérifier que le bouton "Abandonner" n'est plus disponible lorsqu'une grille est finie et correctement remplie.

(E) Valider la grille

Acteur : Utilisateur.

Résumé : L'utilisateur valide sa solution après avoir rempli la grille en entier. Il saura alors combien de mots il a correctement deviné et le nombre d'erreurs effectuées.

Préconditions : L'application est correctement lancée, la grille a été générée, affichée et l'utilisateur l'a remplie.

Post conditions: L'utilisateur a fini, les mots correctement devinés et le nombre d'erreurs sont affichés.

Enchaînement nominal :

| Étapes | Utilisateur | Système | Maquette correspondante |
|--------|--------------------------------|---------------------------------------|---------------------------|
| 1 | Clique sur le bouton "Valider" | | Maquette - Grille remplie |
| 2 | | Affiche les mots correctement devinés | |
| 3 | | Affiche le nombre d'erreurs | Maquette - Validation |
| 4 | | Met à jour le score | |

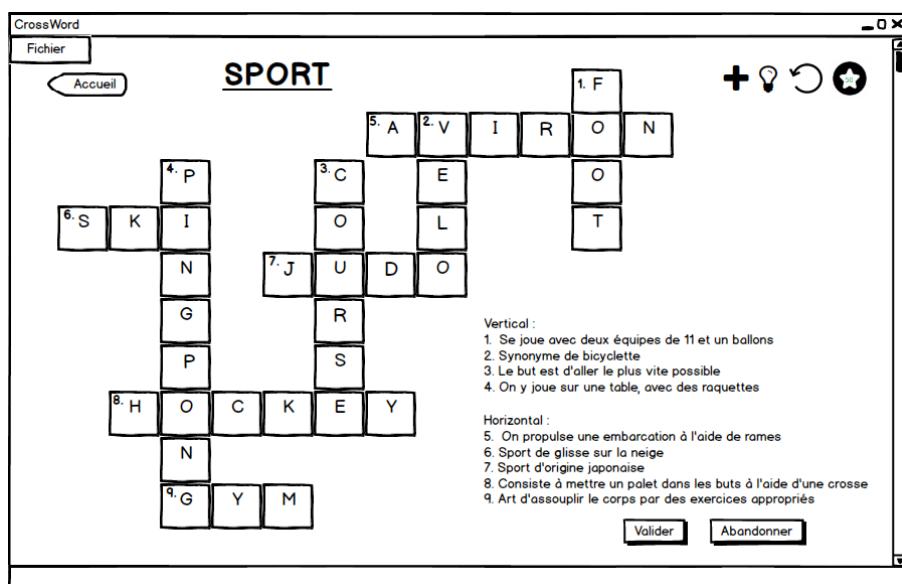


Figure 20: Maquette - Grille remplie

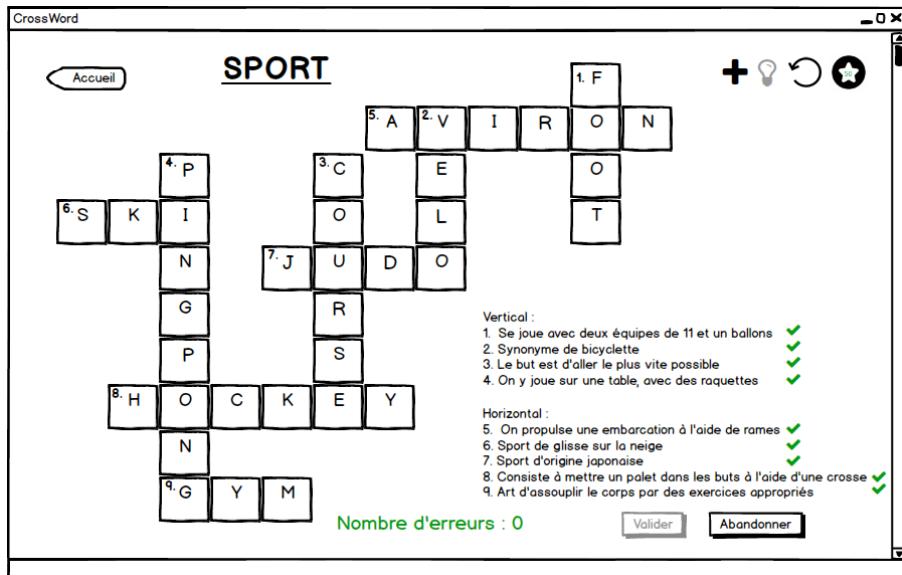


Figure 21: Maquette - Validation

3.4 Besoins système

Le besoin système principal concerne la génération de la grille. Nous devons être sûrs que la grille affichée à l'utilisateur a une solution. Pour cela nous devons donc coder un solver de grille qui trouvera la solution pour une grille donnée (si elle existe). Grâce à ce solver nous pourrons stocker la solution de la grille ce qui permettra également à l'utilisateur de jouer en vérifiant ses réponses et en utilisant les aides. Pour cela nous allons représenter la génération de grille de mots croisés comme un problème de satisfaction de contraintes. L'encodage sous forme de CSP est décrit en section "6.2 VARIABLE - Champ accueillant un mot". L'algorithme du solver est quant à lui décrit plus loin en section "6.4 Algorithme d'attribution des mots aux Variables - Solver".

4 Besoins non fonctionnels

4.1 Besoins d'utilisabilité

(E) Interface graphique

Notre grille de mots croisés sera jouable sur une simple interface graphique, disposant de la grille et d'un menu. Cette interface devra être simple d'utilisation.

(E) Manuel d'utilisation

Notre programme disposera d'un court manuel d'utilisation afin de préciser les différents points sur l'utilisation de notre grille de mots croisés et sur l'utilisation des différents menus de notre application.

(E) Langue du programme

Tout le programme (interface, mots de la grille et menu) devra être écrit en Anglais.

4.2 Besoins de performances

(E) Temps de génération de la grille de mots croisés

Toute génération de grille de mots croisés (quelque soit les paramètres choisis par l'utilisateur) doit se faire dans l'ordre de la seconde. Notons que cela concerne **uniquement** les grilles disponibles à l'utilisateur. Une attention particulière sur la complexité de l'algorithme de génération ainsi que sur les tests de génération de la grille devra être apportée.

4.3 Besoins de sécurité

(E) Sécurité

L'utilisation de notre programme ne requiert aucune connexion internet (sauf pour l'installation des librairies nécessaires à l'interface graphique), de ce fait aucune donnée personnelle propre à l'utilisateur ne sera sauvegardée ou utilisée ultérieurement.

5 Parallèle entre notre analyse initiale et les choix faits lors du développement

L'analyse des besoins présentée plus haut a été réalisée au début du projet avant de commencer son implémentation. Nous allons maintenant présenter et argumenter les différences entre cette première analyse et les choix qui ont été effectivement mis en place lors du développement.

5.1 Maquettes

Dans un premier temps nous nous sommes rendu compte que les grilles de mots croisés sont des grilles carrées ou rectangles contenant des cases noires et ne sont pas seulement composées des mots qui s'intersectent. Voici une mise à jour de l'interface actuelle :

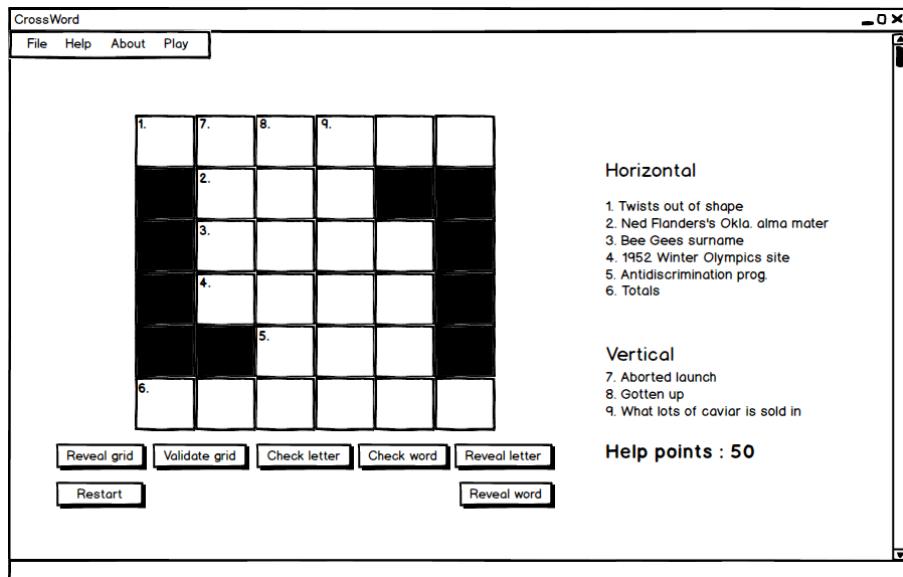


Figure 22: Maquette finale - Grille vide

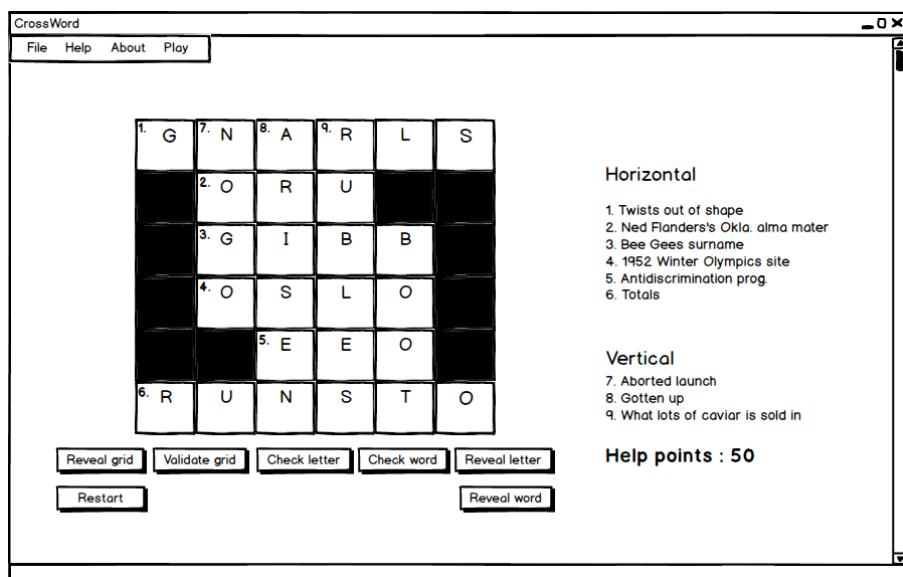


Figure 23: Maquette finale - Grille remplie

5.2 Patterns

Nous avons aussi fait le choix d'appliquer notre algorithme sur des grilles ayant des cases noires déjà placées. Pour cela nous avons créé une dizaine de motifs de grille pour chaque niveau de difficulté (facile, normal, difficile). Lors de la génération, nous choisirons une grille du bon niveau de difficulté et nous lui appliquerons notre solver (cf. "6.5 Algorithme d'attribution des mots aux Variables - Solver") afin d'être sûrs que la grille ait une solution. Ce choix des cases noires déjà placées au lieu d'un algorithme qui les placerait au fur et à mesure sur la grille vide s'il ne trouve pas de mots nous a permis d'éviter d'avoir à effectuer un parcours sur une liste de champs qui change constamment.

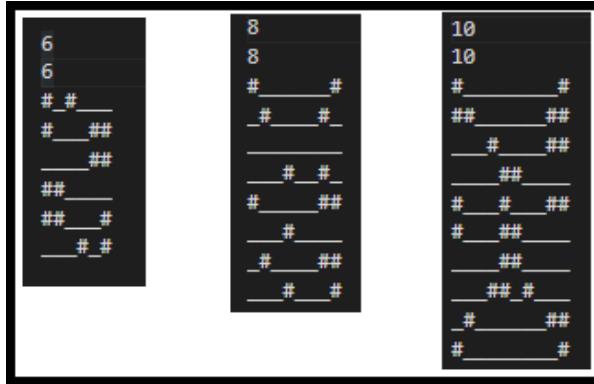


Figure 24: Exemple de patterns pour chaque difficulté

5.3 Format de données pour l'importation et l'exportation

Lors de notre conception nous pensions que nous allions créer nos fichiers d'importations et d'exportations avec des fichiers au format *txt*. Or, lors du développement de ces fonctionnalités, nous avons remarqué que manipuler ces fichiers *txt* devenait fastidieux (problème encodage entre différentes architectures). Nous avons donc réfléchi à une alternative et nous avons choisi le format de données *JSON*.

Le fichier d'importation regroupe l'état du jeu :

- la difficulté
- le nombre de points d'aides restants
- la grille du joueur
- la solution générée par notre algorithme
- les définitions/indices associés aux mots de la grille

```
{
  "current_grid": [
    "#_____###",
    "#_#_#_____",
    "#_____###",
    "###_##D___",
    "____F_F##",
    "###_###",
    "##_#_____",
    "____###_###",
    "_____",
    "#####"
  ],
  "difficulty": 2,
  "help_points": 50,
  "solution_grid": [
    "#ANDRO#####",
    "#GO##XXXXX",
    "#RUDY#####",
    "##E##MIND",
    "ABERDEEN##",
    "##MOEN##",
    "BUC#ORTEGA",
    "FRD##A##",
    "BLINDALLEY",
    "E#####
  ],
  "words": [
    {
      "associate_def": "Bodybuilding drug used by many home run hitters in the '90s \r",
      "horizontal": true,
      "number": 1,
      "pos": {
        "x": 1,
        "y": 0
      },
      "word": "ANDRO"
    },
    {
      "associate_def": "\"Swans Reflecting Elephants\" and others \r",
      "horizontal": true,
      "number": 2,
      "pos": {
        "x": 5,
        "y": 1
      },
      "word": "XXXXX"
    },
    {
      "associate_def": "Healthily red \r",
      "horizontal": true,
      "number": 3,
      "pos": {
        "x": 1,
        "y": 2
      },
      "word": "RUDY"
    }
  ]
}
```

Figure 25: Exemple de fichier d'importation

6 Représentation du jeu des mots croisés et algorithmes

6.1 SOLVER - Coeur de l'algorithme de génération

Le *solver* contient tous les éléments nécessaires à la résolution d'une grille. Plus précisément il contient les éléments suivants:

- un *pattern* : configuration initiale de la grille, seule la position des cases noires et les dimensions de la grille y sont présentes.
- une liste des *variables* de la grille (cf. section "VARIABLE - Champ accueillant un mot").
- le nombre de ces variable.
- les mots qui ont déjà été placés
- la grille sur laquelle on travaille

L'ensemble des variables sera initialisé grâce à notre **Algorithme d'initialisation des Variables** (cf section 6.3).

Ensuite, la résolution se fera en suivant l'**Algorithme d'attribution de mots aux Variables** (cf section 6.4).

6.2 VARIABLE - Champ accueillant 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 éléments suivants :

- le mot qu'elle contient et sa taille (à l'initialisation ce mot est défini 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 possibles pour une variable (en tenant compte des contraintes) ainsi que sa cardinalité.

Pour une variable les contraintes sont : la taille du mot que l'on doit trouver ainsi que des lettres précises qui doivent se trouver à certaines positions en fonction des mots qui ont déjà été placés.

Le *domaine* est un sous ensemble du **lexique**, i.e la base de données (Cf Section 6.5). Initialement il contient tous les mots du *lexique* d'une longueur égale à l'espace occupé par la *variable*. Par la suite il sera régulièrement mis à jour afin d'enlever les mots qui ne respectent plus les contraintes.

6.3 Algorithme d'initialisation des Variables

Cet algorithme permet d'initialiser le tableau de variable de la grid en prenant en entrée cette même grille initialisée à partir d'un pattern de grille. En sortie, chacune des variables du tableau sera initialisée et disposera de :

- son mot (initialisé à *null*)
- son *domaine*
- sa taille
- son orientation

- sa position (case où débutera le mot)
- ses voisins (autres variables qui ont des cases en commun)

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_blanche) then
            taille_mot ← taille_mot + 1
            Retient la position
        end if
        if (case_precedente = case_blanche ET case_actuelle = case_blanche) then
            taille_mot ← taille_mot + 1
        end if
        if (case_precedente = case_blanche ET case_actuelle = case_noire) then
            if (taille_mot ≥ 3*) then
                Créer variable (avec position, direction horizontale et taille_mot)
            end if
        end if
    end for
    if (case_actuelle = case_blanche ET (taille_mot ≥ 3*)) then
        Créer variable (avec position, direction horizontale et taille_mot)
    end if
    taille_mot ← 0
    case_actuelle ← case_noire
end for

for Chaque ligne i do
    for Chaque colonne j do
        De la même façon, créer les variables verticales
    end for
end for

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. Pour simplifier la lisibilité de l'algorithme le calcul de la position a volontairement été omis.

6.4 Algorithme d'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 celle-ci.

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 = variable dont la cardinalité associée au domaine est minimale
    word = premier mot du domaine de v
    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

```

Par un processus de randomisation du *domaine* des *variables*, deux appels sur une même *grid* pourront produire plusieurs grilles différentes.

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.

6.5 Lexique (base de données) - Stockage des mots

Notre solver (décrit via l'algorithme 2) nécessite une base de données de mots. Nous allons présenter ici son implémentation abstraite. Nous allons pour cela 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 [11]. 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 $O(d/32)$ avec **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 :

| 0 | A | B | C | D | E | F | G | H | I | ... |
|---|---|---|---|---|---|---|---|---|---|-----|
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 3 | 2 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 4 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| Nombre de mots de longueur X (où x = 4 ici) | | | | | | | | | | |
| ... miro | | | | | | | | | | |

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'autres 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** :

6.5.1 Algorithme d'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(dbWordLength(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 appelée dans la fonction précédente) :

Algorithm 4 Ajouter Mot

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

7 Techniques utilisées

7.1 Langages

D'après l'article de thèse [11] 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 néanmoins choisi le langage **Cpp** car c'est un langage objet, il nous permet un plus grand contrôle sur la structure du code et facilite l'utilisation des Designs Patterns.

7.2 JSON

On peut aussi noter l'utilisation du JSON comme format de données pour nos fichiers d'importation et d'exportation. Pour manipuler ce format de données avec du **Cpp** nous avons utilisé la library **nlohmann_json** [12]

7.3 Interface Graphique

Il ressort de nos recherches sur les meilleures interfaces graphiques en Cpp que le framework Qt [13] semble être le plus performant, documenté et simple d'utilisation. C'est celui que nous allons utiliser. De plus, certains du groupe l'avaient déjà utilisé, bien entendu ce paramètre est rentré en compte.

7.4 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 trouvée sur ce site web [14]. En utilisant la bibliothèque **numpy** (et **pandas**) de python, nous lui avons ensuite appliqué un traitement afin de la trier par taille et récursivement par ordre alphabétique. Initialement, la base de données contenait 6 millions de mots. Nous avons jugé cela trop important (rapelons que l'article [1] préconisait une taille d'environ 20 000 mots pour des performances optimales). Ainsi, nous avons obtenu via nos traitements une base de données de **31316** mots. Ce résultat a été obtenu en enlevant tous les mots doublons et en gardant ceux les plus pertinents (nous gardons les mots qui apparaissent au moins 25 fois). Bien entendu, chacun des mots est associé à une définition.

Pour trier et ne garder que ces 31316 mots, notre script python est le suivant :

```
1 import pandas
2 import numpy
3 import json
4
5
6 df = pandas.read_csv("xd/clues.tsv", sep='\t') #le fichier de base
7 df.pop("pubid") #on enlève les colonnes qui ne nous intéressent pas
8 df.pop("year")
9 rep=df["answer"].value_counts()
10 common=rep[rep>=25] #on garde seulement les mots les plus communs (qui apparaissent + de 25 fois)
11 l=sorted(common.keys())
12 l=sorted(l,key=len)
13 with open('jsonrep/result.txt', 'w') as f:
14     f.write('\n'.join(l)) #puis on écrit les mots qui nous intéressent dans notre fichier
```

Afin de récupérer et d'encoder le fichier "mot - définition" de manière adéquate à notre application, nous avons écrits le script python suivant :

```
1      #!/usr/bin/python
2      fichierMot = open("data/word_database.txt") #mot de notre base
3      FichierDef = open("data/clues.tsv") #les définitions
4      fichierOut = open("data/word_with_def.txt", "a") #le fichier à produire
5
6      lignesWords = fichierMot.readlines()
7      lignesClues = FichierDef.readlines()
8
9      for mot in lignesWords: #parcours chacun des mots du fichier mot
10         fichierOut.write(mot) #on l'écrit dans le fichier de sortie
11         for clues in lignesClues: #parcourt tous les indices
12             words = clues.split()
13             try:
14                 if(words[2] + "\n" == mot):
15                     for i in range(3,len(words)):
16                         fichierOut.write(words[i] + " ")
17                         fichierOut.write("\n \n")
18                     break
19             except IndexError:
20                 a = 5 #ne fait rien ...
21
22
23     fichierMot.close()
24     FichierDef.close()
25     fichierOut.close()
```

8 Architecture de l'application

Il a fallu réfléchir à une architecture qui nous permette de répondre à tous les besoins décrits plus haut et qui puisse accueillir notre générateur de grille.

8.1 Démarche

Notre démarche a été de penser à un découpage de classes qui respecte les bonnes pratiques inculquées en génie logiciel. Chacune de nos classes ont un rôle précis, nous avons notamment voulu que tout ce qui concerne le générateur d'une grille (les algorithmes décrits dans la section "6 Représentation du jeu des mots croisés et algorithmes") ne soit pas dépendant du jeu.

L'avantage à en tirer, au-delà des bonnes pratiques, est de permettre, dans un premier temps, de programmer le générateur de grille jusqu'à le rendre fonctionnel sans avoir à se soucier du jeu. Et une fois le générateur entièrement fonctionnel le jeu aura juste à l'utiliser pour récupérer des grilles.

Un point assez important pour être à l'aise avec l'architecture est de comprendre ce qu'est l'état de notre jeu de mots croisés, cet état est contenu dans la classe **Crossword**:

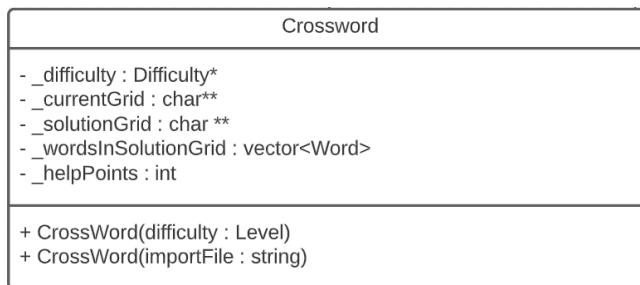


Figure 26: Classe "Crossword"

Nous y retrouvons:

- la difficulté des mots croisés
- la grille courante : la grille sur laquelle le joueur joue
- la grille de solution : c'est la grille remplie qui a été générée par notre générateur de grille
- la liste de mots de la grille : cette liste contient toutes les informations associées à chaque mot de la grille (comme par exemple leur numéro et leur définition associée)
- le nombre de points d'aide disponibles pour le joueur

Le jeu de mots croisés peut être construit à partir d'une difficulté : dans ce cas là on sélectionnera un pattern (lié à la difficulté choisie) au hasard et le générateur de grille générera une grille en se reposant sur le pattern sélectionné.

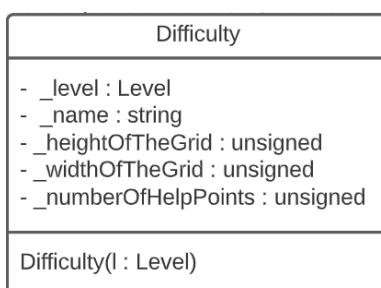


Figure 27: Classe "Difficulty"

Lors de la construction du jeu, il récupère toutes les données liées à la difficulté à l'aide de la classe **Difficulty**. Cette classe est construite à partir d'un "niveau" (EASY, NORMAL, HARD) et stocke toutes les données correspondantes à cette difficulté:

- le niveau (EASY, NORMAL ou HARD)
- le nom littéral de la difficulté («easy», «normal», «hard»)
- la longueur de la grille associée à cette difficulté
- la largeur de la grille associée à cette difficulté
- le nombre de point d'aides de départ associé à cette difficulté

Le jeu de mots croisés peut aussi être construit à partir d'un fichier d'importation, dans ce cas là l'état stocké dans le fichier d'importation est totalement rétabli.

Toutes les méthodes pour manipuler l'état du jeu de mots croisés sont réunies dans une interface, dont nous allons parler dans la section "8.3 Design patterns".

Pour y voir plus clair par la suite, voici le diagramme de classes UML *simplifié* de notre application :

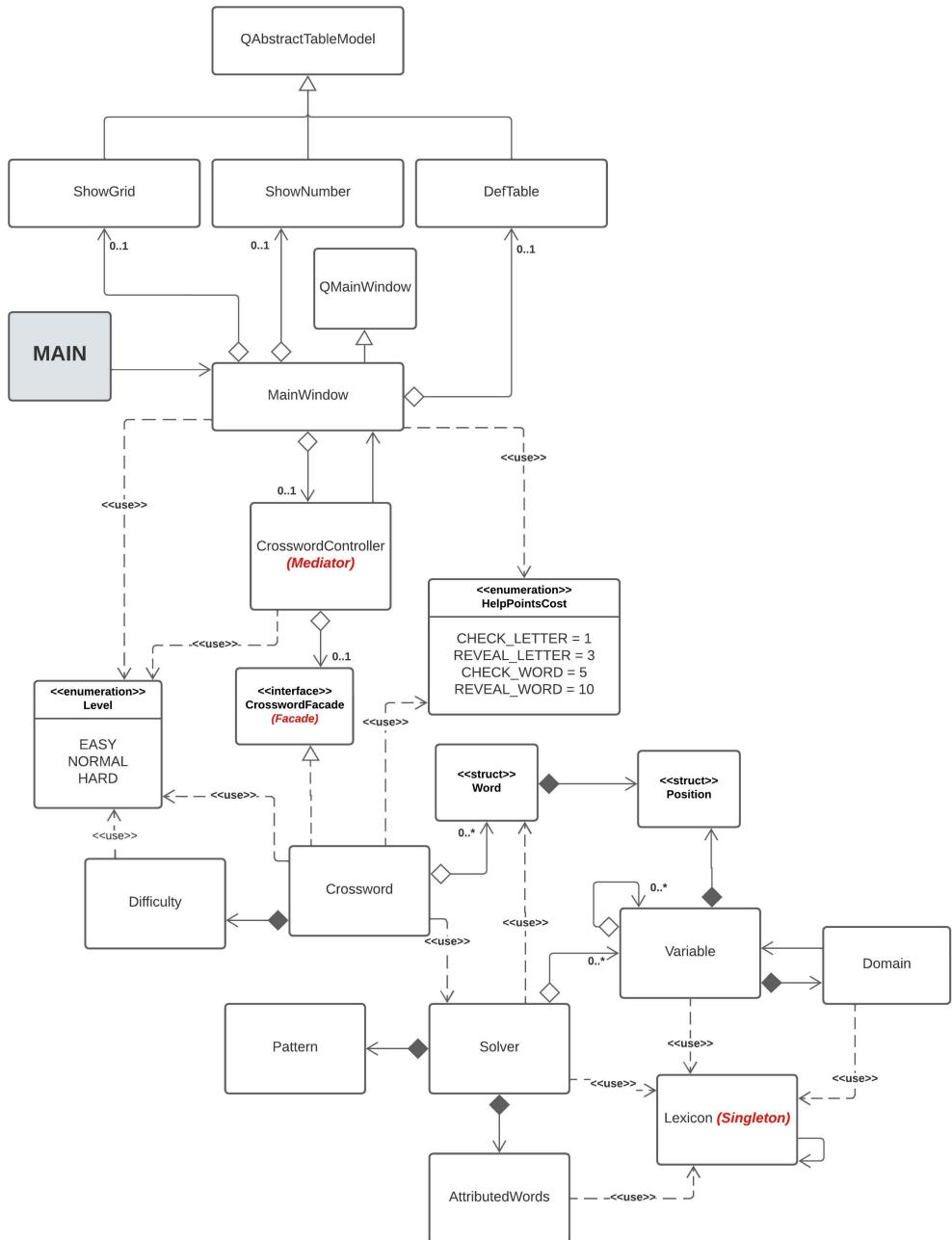


Figure 28: Diagramme de classes simplifié

En bas à droite du diagramme ci-dessus, on peut y voir la classe **Solver** : cette classe représente notre générateur de grille de mots croisés. Nous remarquons, comme il a été expliqué plus haut, que le jeu (**Crossword**) utilise le générateur mais que lui n'est pas dépendant du jeu.

Les classes **Variable**, **Domain**, **Lexicon**, **Pattern**, **AttributedWords**, dont le générateur dépend rappellent tous les éléments nécessaires à la génération d'une grille, décrits dans la section "6 Représentation du jeu des mots croisés et algorithmes".

Autre point concernant le découpage de nos classes : notre application possède une interface utilisateur, nous avons donc fait le choix d'utiliser le modèle **MVC** (*Model-View-Controller*), pour que ni le jeu ni l'interface ne soit dépendant l'un de l'autre.

8.2 Model-View-Controller

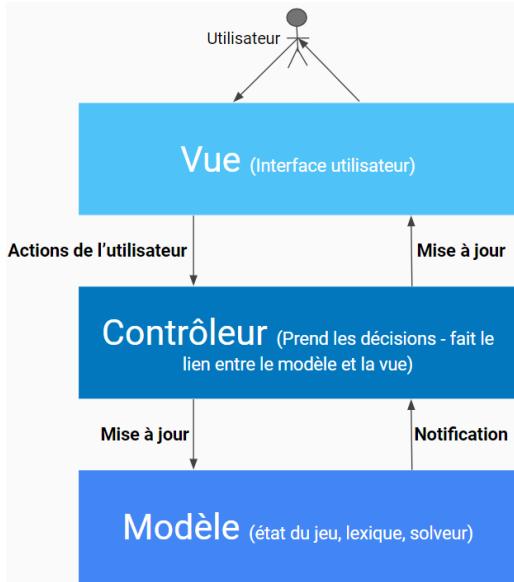


Figure 29: Schéma de notre implémentation du MVC

Dans notre application, l'utilisateur interagit essentiellement avec la vue. La vue dans notre architecture correspond à la classe **MainWindow** et ses agrégations : **ShowGrid** a pour rôle d'afficher la grille de jeu du joueur, **ShowNumber** d'afficher les numéros des mots dans la grille, **DefTable** d'afficher le tableau des définitions qui sont associées aux mots de la grille. **MainWindow** les instancie quand une grille a été générée.

MainWindow regroupe des déclencheurs qui communiquent au contrôleur chaque action de l'utilisateur. Ensuite le contrôleur, ici **CrosswordController**, manipule l'état de jeu (**Crossword**) puis communique à la vue le nouvel état à afficher.

8.3 Design patterns

Quand la solution s'y prêtait nous avons souhaité utiliser un design pattern. Le contrôleur du modèle MVC est d'ailleurs un design pattern, il joue le rôle de **médiateur**.

Le contrôleur utilise l'interface **CrosswordFacade** pour manipuler le jeu, cette classe est aussi un design pattern : **facade**, il nous a permis en amont de réunir dans une interface toutes les fonctionnalités dont l'extérieur aurait besoin pour manipuler le jeu. L'implémentation de cette classe a été réfléchie avant de fabriquer l'état du jeu, ce qui nous a donné comme premier avantage d'avoir les idées claires sur toutes les fonctionnalités à implémenter dans la classe **Crossword**. Le second avantage a été de pouvoir avancer sur le développement du contrôleur (et par conséquence de la vue aussi) sans se soucier de l'avancée du développement de la classe **Crossword**.

Nous pouvons aussi parlé de la classe **Lexicon** qui, elle, est un **singleton**. En effet, nous avons besoin d'accéder au lexique tout au long de la génération. De plus, le lexique est unique nous voulons nous assurer qu'une seule instance sera créée. Le pattern **singleton** nous a donc permis de fournir au générateur un point d'accès et aussi de nous assurer qu'il y a une unique instance du lexique.

8.4 Diagramme de classes UML détaillé

Si vous ressentez le besoin de voir l'architecture de l'application de manière détaillée voici le diagramme de classes UML avec les méthodes et attributs de chaque classe. Les seules méthodes qui n'y apparaissent pas sont les getters/setters et certaines méthodes qui appartiennent au framework Qt :

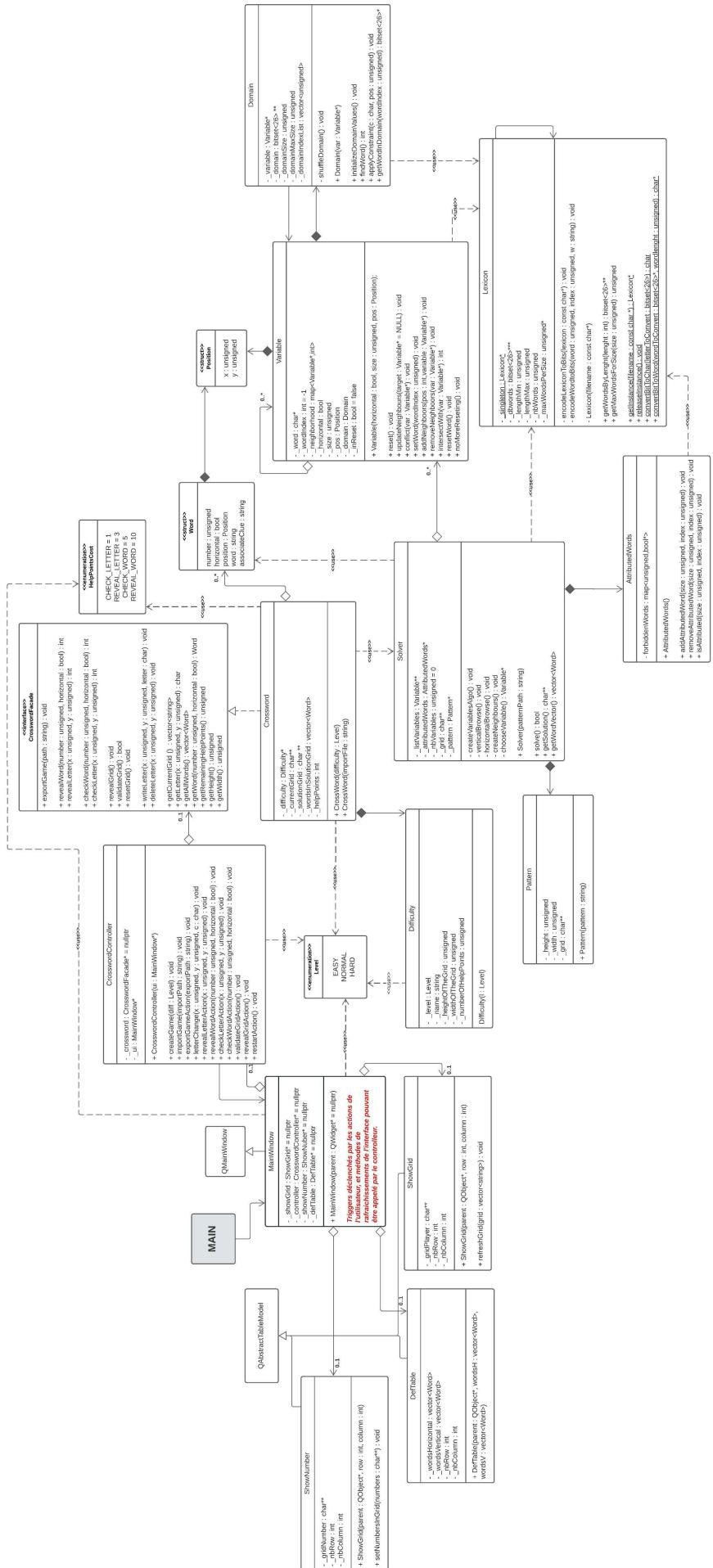


Figure 30: Diagramme de classes UML détaillé de notre application

8.5 Diagramme de paquetages UML

Un autre point intéressant de notre architecture est la répartition des classes dans les différents paquetages, en effet il y a beaucoup de classes donc une bonne répartition nous a semblé nécessaire.

Voici le diagramme de paquetages UML de notre application :

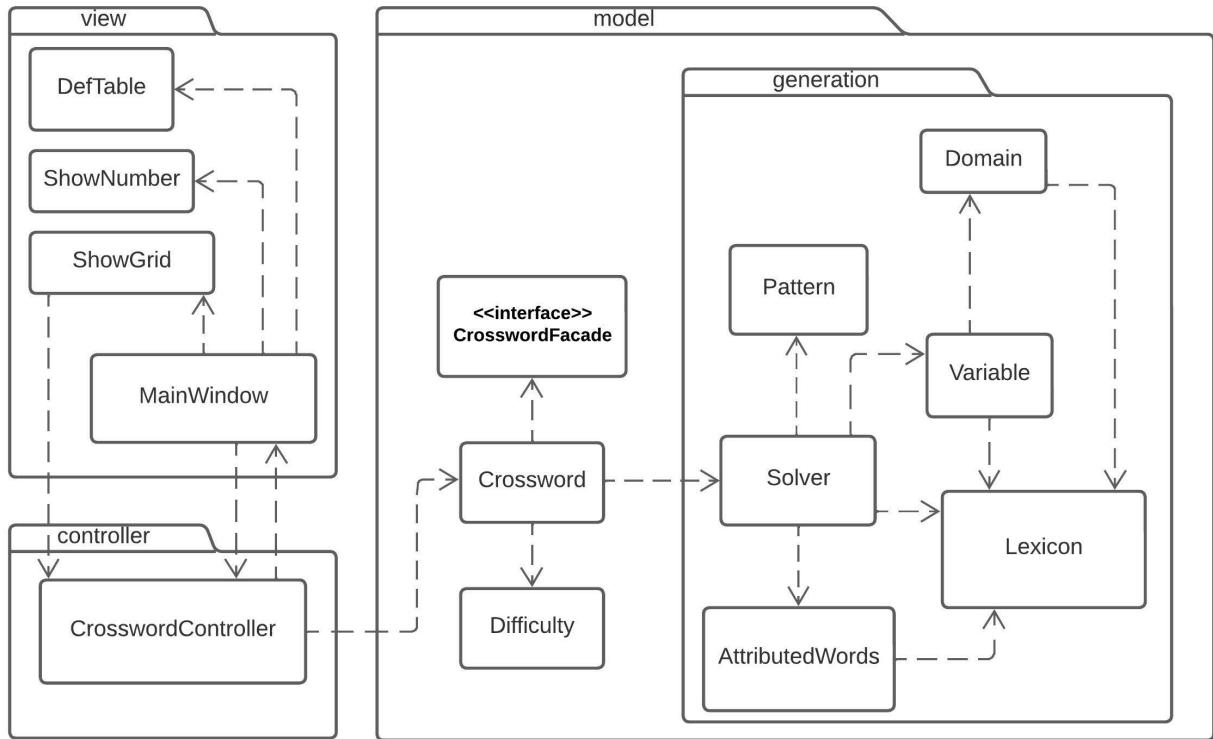


Figure 31: Diagramme de paquetages UML de notre application

Nous avons réparti les classes dans 3 paquetages principaux en se reposant sur notre utilisation du modèle MVC : un paquetage pour la ***view***, un pour le ***controller*** et un pour le ***model***. Dans le ***model***, nous avons aussi voulu séparer le jeu de mots croisés, de la génération de grille en créant un paquetage ***generation*** dans lequel nous avons mis toutes les classes concernant la génération d'une grille.

9 Documentation du code

Lors de la programmation, nous avons pris soin d'entièrement documenter notre code. Pour cette documentation nous avons utilisé le générateur de documentation **Doxygen** [15], cela à impliqué de commenter toutes nos classes, attributs et méthodes dans un format particulier qui permet donc aussi de lire la documentation directement sur le code.

Pour générer la documentation html que **Doxygen** propose nous avons utilisé la commande :
"doxygen Doxyfile" dans le répertoire "**src/build**". **Doxyfile** étant un fichier de configuration fournis par **Doxygen**, que nous avons modifié selon nos souhaits.

Sur le dépôt de notre application, cette documentation se trouve dans le répertoire :

"data/doxygen_documentation/html"

En ouvrant le fichier **annotated.html** vous ouvrez la page qui répertorie toutes les classes de notre implémentation, en cliquant sur une classe vous aurez accès à sa documentation (sous forme de page

html).

Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

| | |
|--|--|
| C AttributedWords | Class that makes an inventory of the words that are already placed in the grid |
| C Crossword | Crossword is the class that holds the state of our game |
| C CrosswordBadAccessException | Exception thrown when trying to access to a letter or word that is not in the crossword |
| C CrosswordController | This controller acts as a mediator between the graphical interface and the crossword game |
| C CrosswordFacade | Facade used to group together all the functions that the outside would need to play our crossword games |
| C CrosswordGenerationException | Exception thrown when the crossword generation failed |
| C DefTable | Table in the interface that displays the definitions associated with the words of the crossword |
| C Difficulty | Difficulty is the class that contains all the data associated with a difficulty of our crossword |
| C DifficultyException | Exception thrown when a non-existent difficulty is given |
| C Domain | Represents the domain of values for a Variable |
| C Lexicon | The application's lexicon, contains words encoded in binary |
| C LexiconException | Exception thrown when trying to access to a word that is not in the lexicon |
| C MainWindow | Class of the main window of the graphic interface |
| C Pattern | Used to create, from a txt file, a grid pattern that can be manipulated by our Solver |
| C PatternException | Exception thrown the pattern file is not in the good format or corrupted |
| C Position | Our crossword is a 2 dimension grid we use this type to store the position in the grid |
| C ShowGrid | Player's grid that is displayed on the graphical interface |
| C ShowNumber | Class that display numbers associated to the words in the grid |
| C Solver | This class contains the core of the algorithm for generating a crossword puzzle |
| C SolverLexiconException | Exception thrown when there is a LexiconException during the solving |
| C SolverPatternException | Exception thrown when there is a PatternException during the solving |
| C Variable | Class representing a variable in our CSP |
| C Word | "Crossword" word type. Use to represent what is a word in a Crossword game |

Figure 32: Page *annotated.html* de la documentation

10 Résultat final

Depuis l'écran d'accueil ci-dessous l'utilisateur peut sélectionner une difficulté ou importer une grille précédemment exportée :

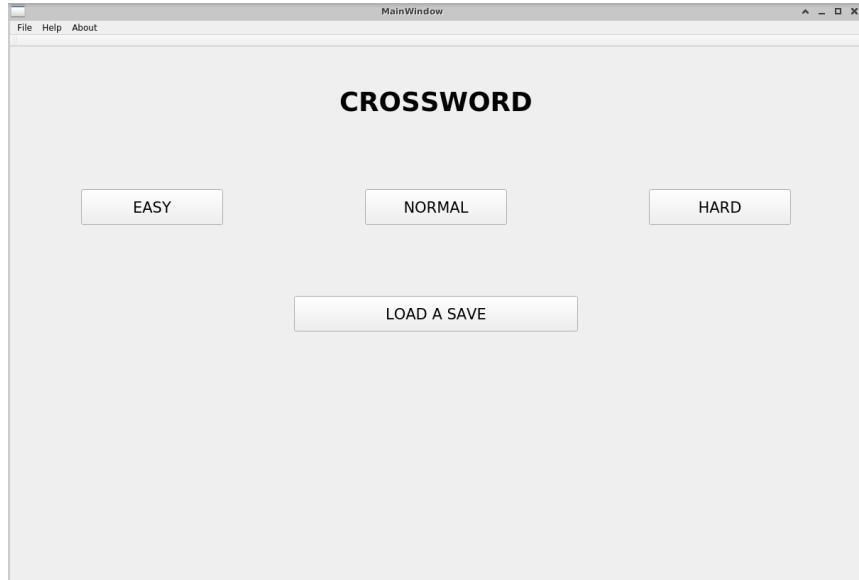


Figure 33: Menu de l'application

Voici des exemples de résultats pour les trois niveaux de difficulté :

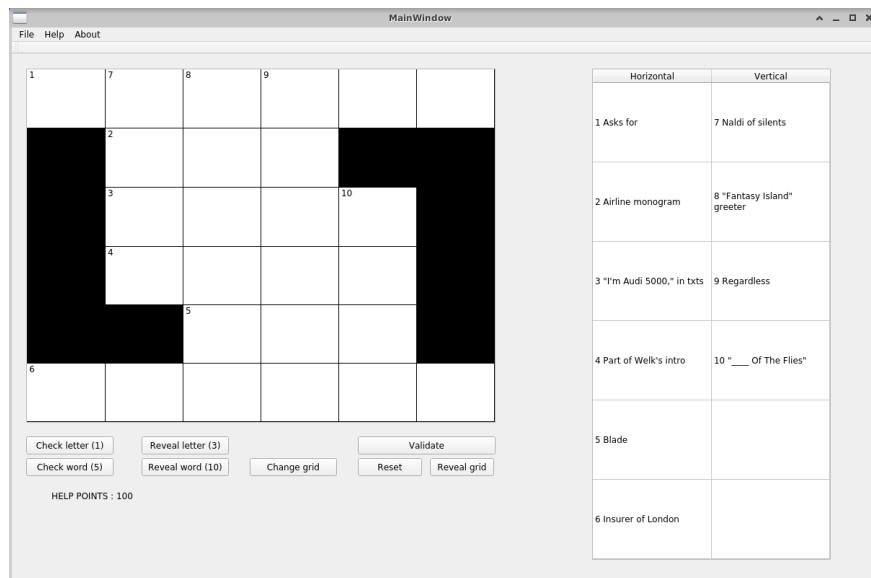


Figure 34: Grille niveau facile vide

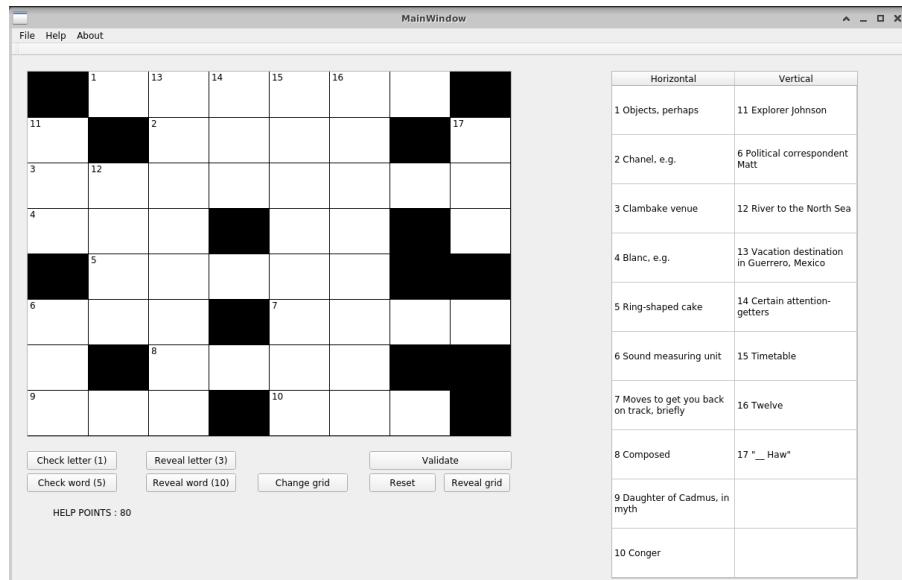


Figure 35: Grille niveau normal vide

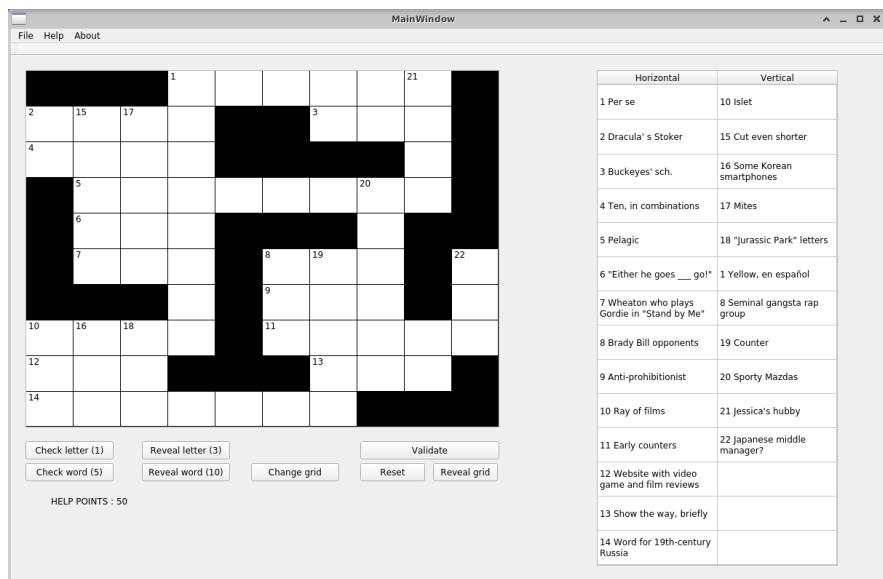


Figure 36: Grille niveau difficile vide

Lors de la partie l'utilisateur bénéficie de plusieurs aides. Il peut vérifier ou révéler une case ou un mot. Chaque aide coûte un certain nombre de points. L'utilisateur commence la partie avec un nombre de points d'aide plus ou moins élevé suivant le niveau de difficulté (cf. captures). Pour vérifier ou révéler un mot, le joueur peut simplement sélectionner la définition du mot choisi et ensuite cliquer sur le bouton correspondant. Par exemple pour révéler un mot :

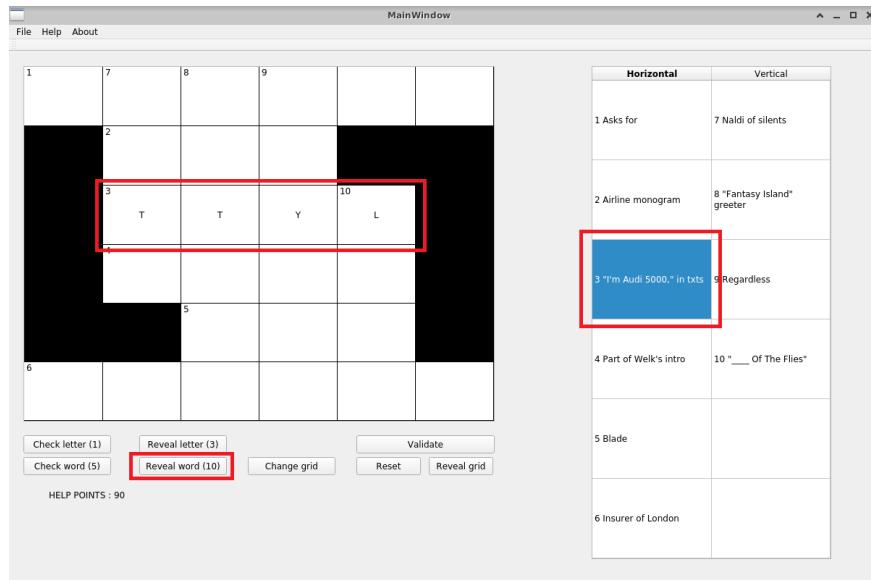


Figure 37: Révéler un mot avec l'interface

On voit que le nombre de points a été mis à jour et est maintenant de 90. Pour vérifier ou révéler une lettre on procède de la même façon en sélectionnant cette fois-ci la case.

L'utilisateur peut choisir de dévoiler la grille à n'importe quel moment s'il souhaite abandonner. Il pourra aussi vérifier sa grille pour savoir si elle est correcte ou non.

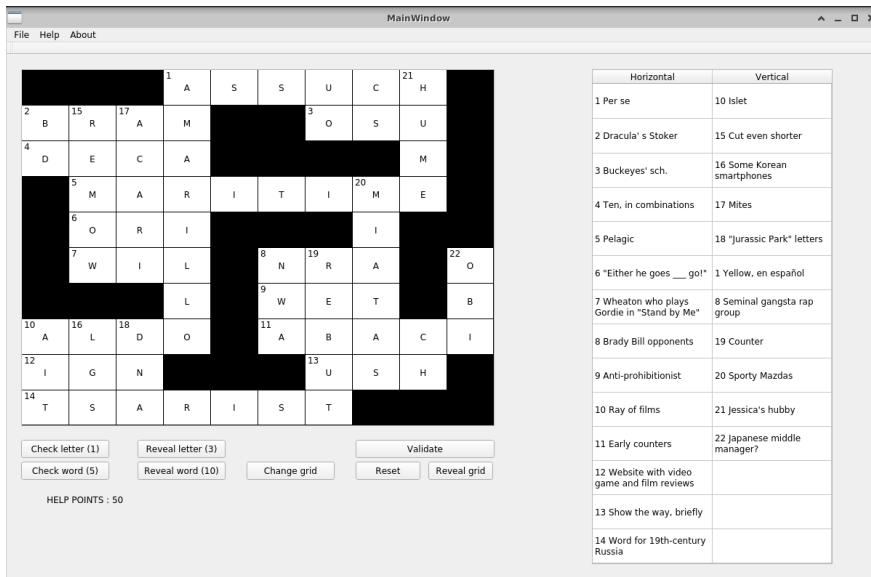


Figure 38: Grille niveau difficile révélée

Nous pouvons également suivre l'exécution du solver en mode terminal et voir la grille s'afficher à chaque mot placé :

```
emma@LAPTOP-8A26AH5Q:~/motscroises/src/build$ ./CrossedWord term easy
Solving the grid...

C T N # # #
- # # - - -
- - - - - -
- - - - # - -
# - # # - - -
# # # - - -

C T N # # #
I # # - - -
G - - - - -
N - - - - -
A - - # - -
# # # - - -

C T N # # #
I # # - - -
G - - - - -
N - - - - -
A R R # # -
# # # - - -
```

Figure 39: Début du lancement en mode terminal

Figure 40: Fin du lancement en mode terminal

Cela nous a principalement servi durant le développement afin de surveiller le déroulement des exécutions.

11 Tests de génération de grilles sur des grilles de tailles variables

L'idée va être maintenant de faire une analyse la plus complète possible du logiciel. Pour cela, nous allons ici faire une description des tests que nous avons effectué. Ceux-ci ont pour objectif de convaincre que le logiciel développé se comporte correctement dans diverses situations (jugées "hors du commun").

Dans cette partie, nous allons confronter notre algorithme à des grilles aux dimensions variables, possédant pour certaines des cases noires, et pour d'autres uniquement des cases blanches (i.e des grilles vides). Pour chacune des générations, nous intégrerons une capture d'écran de la génération issue du solver, puis nous noterons le temps associé. A noter que les conditions de tests sont un ordinateur fixe tournant sous Ubuntu 20.04 avec 64 GO de RAM.

11.1 Sur des grilles carrées

11.1.1 Avec cases noires

| Affichage terminal | Paramètres |
|---|--|
| <pre>Grid completed in : 265.185ms # K W A I # O # # B S S R E A R U P C H E A P O A U R # # T # D O M S #</pre> | dimension 6*6 Pattern : easy2.txt |
| <pre>Grid completed in : 711.745ms # T E A S E D # N O T I # V O A H R H # # E G O # R E Z O N E # # E R R A T A # T N N # # I R T A T E S # D E P # S T A G E D #</pre> | dimension 8*8 Pattern : medium1.txt |
| <pre>Grid completed in : 3082.77ms # O C A R I N A S # # # A T A B A L # # C W T # B O N A # # C O B H # # E R I A # O U I # A T M # # # E R Y # # T C B Y A R G O # # E L A N A S L # # B # O M E U # A L A R I C # # # S R I L A N K A #</pre> | dimension 10*10 Pattern : hard1.txt |

11.1.2 Grilles vides

| Affichage terminal | Paramètres |
|---|---------------|
| Grid completed in : 130.224ms S T S U I E B E A | dimension 3*3 |
| Grid completed in : 764.164ms Y A N K O T E A L E A L O M R I | dimension 4*4 |
| Grid completed in : 121114ms S W O R E C H R I S R O B O T A S I D E M E T E R | dimension 5*5 |

11.2 Sur des grilles rectangulaires

11.2.1 Avec cases noires

| Affichage terminal | Paramètres |
|--|--------------------------------------|
| Grid completed in : 2034.52ms G I N G E R S N A P P G A # # E T A P E S O B S # F A T E D # # # A M I D S T # | dimension 10*4 Pattern : 10_4.txt |
| Grid completed in : 6990.07ms C R O C O D I L E T E A R S P A Z # # I R O N S I D E S O Z Z Y # O I L E R S # E G S O I # A N N A M # E A S T # R E S T E A S Y # N A E S | dimension 14*5 Pattern : 14_5.txt |

11.2.2 Grilles vides

| Affichage terminal | Paramètres |
|---|---------------|
| Grid completed in : 160.601ms G R O O D S D A H S S A | dimension 3*4 |
| Grid completed in : 120.723ms P U P A E A R N U R S A | dimension 4*3 |
| Grid completed in : 2666.62ms S A R A H S P E L T T A C E T S L U R P | dimension 5*4 |
| Grid completed in : 4552.55ms G R A P P A R E T R I M A L T I M A N O U G A T | dimension 6*4 |

11.3 Bilan du test de génération

Résumons ce que nous avons observé jusqu'ici, à travers les 2 histogrammes suivants :

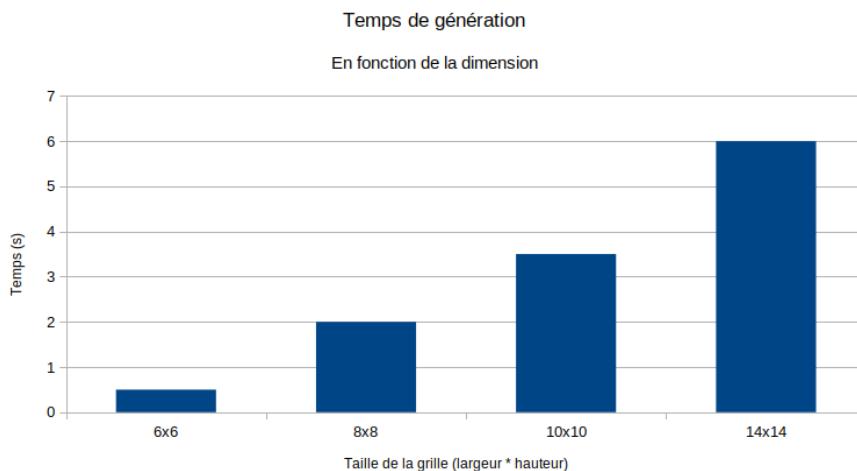


Figure 41: Temps de génération de grille avec des cases noires pré-placées

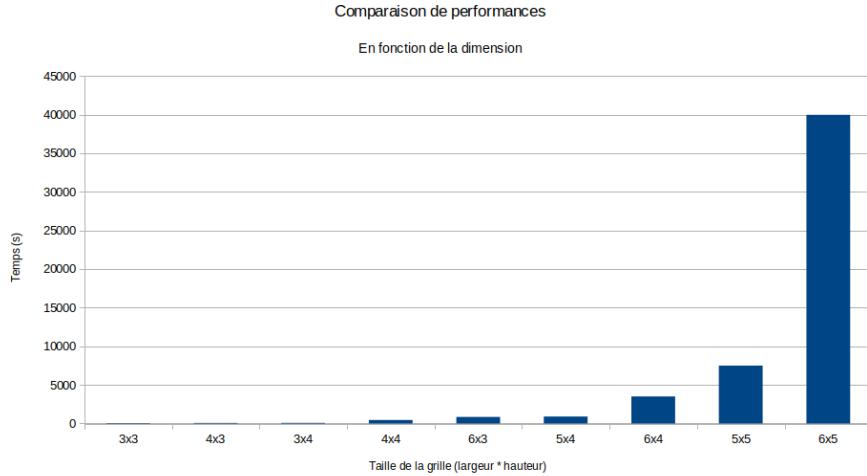


Figure 42: Temps de génération de grille sur des grilles vides aux dimensions variables

Au préalable, il est important de préciser que ces temps peuvent être globalement assez variables. En effet, notre algorithme intègre une part de **randomization** en ce qui concerne la recherche des mots. Afin d'être le plus précis possible, nous avons répété les expériences 3 fois et effectué une moyenne. Nous aurions pu faire le choix de désactiver cette randomisation pour les tests mais cela n'est pas représentatif de ce qu'il se passe réellement dans le cœur de l'algorithme et cela fausserait les résultats (on prendrait en effet toujours les premiers mots qui conviennent).

Nous remarquons que sur des grilles où des cases noires sont pré-placées, les temps de génération de grilles sont largement acceptables puisqu'ils n'excèdent pas les 4 secondes. C'est ce qu'atteste la figure 41. En revanche, sur des grilles vides, jusqu'à une taille de 5 x 5 nous avons un temps de génération convenable (de l'ordre de 30 secondes comme nous pouvons voir sur la figure 42). Par contre, à partir de 6 x 6, le temps de génération de la grille dépasse les 5 heures (nous n'avons réussi à terminer sa génération). Le temps de génération croît de façon exponentielle. Cela justifie le fait que nous avons préféré limiter le temps d'attente en ajoutant des cases noires sur la grille, comme énoncé précédemment.

Aussi, notons que nos tests ont débuté avec une taille de 3 x 3 au minimum ; cela se justifie simplement par le fait que notre base de données ne possède pas de mots de taille inférieure. En revanche, l'utilisation d'une base de données possédant des mots de taille 1 et 2 ne poseraient pas de soucis à notre algorithme principal.

12 Un peu de combinatoire

Nous allons dans cette section parler de combinatoire dans l'objectif d'expliquer le comportement que nous avons observé en figure 42. En effet, il est important de comprendre en quoi la génération d'une grille de mots-croisés à partir d'une grille vide de taille 6 x 6 est si longue. Est-ce fatallement dû à la combinatoire et donc aux nombres de combinaisons exponentielles à explorer ? Ou est-ce dû à la base de donnée. C'est la réponse à laquelle nous allons essayer de répondre dans cette section.

12.1 Complexité algorithmique

Restons sur le cas de la génération de la grille vide de taille 6 x 6, c'est d'abord celui-ci que nous devons élucider.

12.1.1 Quelques chiffres

Sur une telle grille, sont présents :

- **36** lettres, dont 20 sont situées à *l'extérieur* de la grille et sont donc en relation avec 2 autres lettres (pour les 4 lettres situées sur les coins supérieurs droits et gauche (respectivement inférieur droits et gauches) ou en relation avec 3 autres lettres pour les autres. Les 16 autres lettres situées au milieu ont toutes 4 voisines.
- **12** mots à placer qui bien sûr possèdent pour chacune de leur lettre une relation grammaticalement logique avec les 4 lettres qui les entourent → 12 variables à remplir selon notre définition des **variables**
- **7823** mots de taille 6 dans notre base de données

Ainsi, sans introduire d'aléas dans notre recherche algorithmique (en enlevant le mélange des mots que le solver introduit initialement), l'algorithme se comportera ainsi : pour chacune des variables de la grille, il y aura potentiellement **7823** possibilités. Aussi, pour chacune de ces variables, il faudra vérifier que chacune de ses lettres s'intersecte correctement avec :

- 6 autres lettres pour les mots situés sur les extrémités
- 12 autres lettres pour les autres variables

Simplifions les calculs : pour chacune des 12 variables, après placement d'un mot il faudra chercher parmi les $7823 - n$ (n étant le nombre de mots déjà placés sur la grille) mots de la base de données (en supposant qu'à chaque fois on doit parcourir tous les mots de la base de données de taille 6) un mot qui intersecte et respecte toutes les contraintes de voisinage. Ainsi, nous sommes à ce stade déjà sur une complexité de recherche très importante ($> O(n^2)$). Rajoutons à cela que plus nous plaçons de mots sur la grille, plus les contraintes sont importantes, intuitivement nous le voyons simplement.

Ainsi, à de très nombreux moments de l'algorithme, le solver place tous les mots sauf un puis se "rend compte" que le dernier mot à placer n'existe pas dans la base de données. Ainsi, il revient en arrière (par la technique utilisée, le **backtracking**) pour des fois revenir à une grille vide. Combien de combinaisons potentielles existent ? Nous avons un choix de 12 mots parmi 7823, un simple calcul nous donne une combinatoire égale à **108762005423896952721591306484241558583** (10^{38} environ, rappelons que le nombre d'atomes dans l'univers est évalué à 10^{80}) combinaisons possibles. Inutile de préciser que cela est plus qu'exponentiel. Bien sûr, il est illusoire de penser qu'on testera tout cela, mais potentiellement cela peut arriver et cela donne tout de même un ordre d'idée sur la complexité moyenne (même s'il est difficile de l'évaluer précisément) en présence d'une telle base de mots.

A ce stade, nous commençons à intuiter le fait que le temps de génération très important de la grille **6 x 6** ne vient pas de notre algorithme, mais plutôt de notre base de données. Pour le confirmer, nous allons effectuer des tests.

12.2 Tests avec une base de données de mots de taille 6 simplifiée

L'idée va être ici d'utiliser une base de donnée de mots de taille 6, très simple, dont nous sommes sûrs que si notre algorithme est performant, il trouvera une solution (au moins). Dans notre cas, nous utiliserons la base de donnée composée des mots suivants (nous savons préalablement qu'ici le solver est censé trouver 2 grilles différentes) : ACAPER, GAVOTE, IVERON, TINTIN, ETIOLE, SERRES, AGITES, CAVITE, AVENIR, PORTOR, ETOILE, RENNES. Et sans trop de surprise, nous obtenons les 2 résultats suivants :

```

A C A P E R
G A V O T E
I V E R O N
T I N T I N
E T I O L E
S E R R R E S

Grid completed in : 5.1958ms

```

Figure 43: Résultat avec une grille de 6 x 6 sur la base de données de test - possibilité 1

```

A G I T E S
C A V I T E
A V E N I R
P O R T O R
E T O I L E
R E N N E S

Grid completed in : 9.11654ms

```

Figure 44: Résultat avec une grille de 6 x 6 sur la base de données de test - possibilité 2

Commande pour effectuer ce test :

```
./CrossedWord term test -p ../../data/patterns/tests/9_8.txt
-bd ../../data/database/word_database_test.txt.
```

12.3 Un premier bilan de complexité

D'abord, cela nous conforte à dire que notre algorithme est robuste et que le résultat observé en figure 42 était directement **corrélé** à la base de données dont était nourrie le solver. Ensuite, cela nous pousse à faire des tests sur des grilles vides de tailles plus importantes que 6 x 6, afin de poursuivre les tests de notre application. Ces tests, jusqu'alors impossibles (du moins dans un temps raisonnable) seront réalisés avec le même principe d'une base de donnée simplifiée, pour confronter la robustesse du logiciel à différentes tailles.

Réalisons par exemple un test sur une grille (toujours vide), de taille **9 x 8**, avec la base de donnée simplifiée suivante : DERAPERA, ECOTAMES, CONTRAIS, REFRACTE, OULIPIEN, CRASHERA, HANTERAS, ENTERAIS, STERASSE, DECROCHES, ECOEURANT, RONFLANTE, ATTRISTER, PARAPHERA, EMACIERAS, REITERAIS, ASSENASSE dont nous savons qu'il existe une solution. Le solver arrive effectivement à résoudre la grille (dans un temps raisonnable) :

```

D E C R O C H E S
E C O E U R A N T
R O N F L A N T E
A T T R I S T E R
P A R A P H E R A
E M A C I E R A S
R E I T E R A I S
A S S E N A S S E

Grid completed in : 17.8091ms

```

Figure 45: Résultat avec une grille de 9 x 8 sur la base de données de test

Commande pour effectuer ce test:

```
./CrossedWord term test -p ../../data/patterns/tests/9_8.txt
-bd ../../data/database/word_database_test.txt.
```

Ainsi, nous comprenons que notre algorithme est robuste. Il est par contre directement lié à la base de données : l'intuition nous le faisait savoir, nous en avons la certitude à travers cette batterie de

tests effectuée ici. Néanmoins, notre choix ne s'est pas orienté vers ce type de base de données. Bien qu'il permette une génération de grilles vides, il ne permet pas d'obtenir différentes grilles. D'un point de vue utilisateur, cela est moins attractif. Il serait possible de trouver un compromis en utilisant une base de donnée de taille inférieure mais les possibilités de grilles différentes seraient forcément réduites par rapport à notre base de donnée privilégiée.

Par ailleurs, à des fins expérimentales nous avons laissé la base de données de mots de tests utilisée ici, située à l'emplacement **data/database/word_database_test.txt**.

Nous allons enfin généraliser cela sur des grilles vides plus grandes, toujours sur la base de données de tests, en faisant varier la taille des grilles. Notons que cette base de mots n'est utilisée qu'à des fins expérimentales : elle ne possède pas de définitions associées. De plus, les grilles supérieures à une taille de 10 seront réalisées à partir de grilles de mots mêlés ; les mots n'auront ainsi aucune signification, néanmoins nous avons l'assurance qu'ils s'entremêlent tous, les uns avec les autres. Les résultats obtenus sont les suivants :

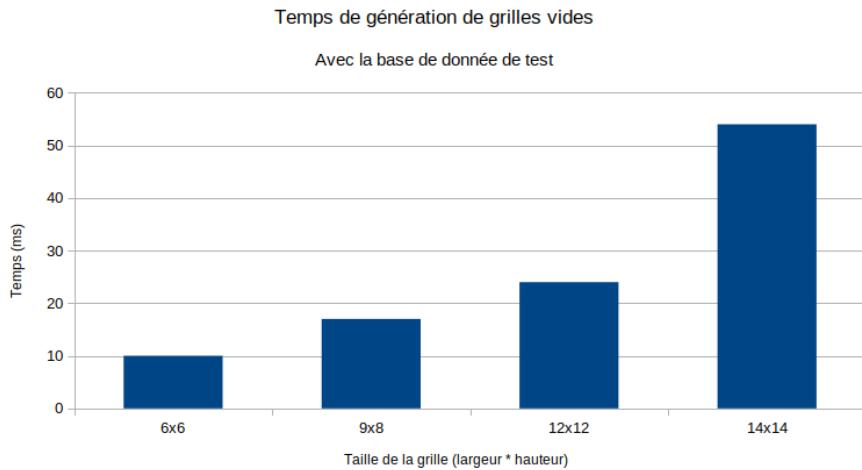


Figure 46: Bilan de performances avec la base de données de test

La figure 46 vient conclure cette partie, qui nous a permis de nous conforter dans la robustesse et la performance de notre algorithme, conjointement à la base de mots, tous deux intimement liés. Notons pour finir que l'algorithme se comporte également bien sur des grilles de mots de taille supérieures. Etant donné leur faible fréquence d'apparition lexicale, nous avons décidé d'écartier ces mots.

12.4 Tests avec des mots "parasites"

Bien sûr, les conditions expérimentales dans lesquelles nous avons effectué les tests ci-avant sont idéales. Idéales dans le sens où pour chacune des tailles de grilles, elles ne contiennent que des mots qui peuvent former une solution. Nous allons à ce stade, toujours dans le but de tester notre application (mais cette fois en condition un peu plus réaliste) rajouter quelques mots parasites pour chacune des tailles et observer l'impact sur le temps de génération de la grille. Ainsi, créons une nouvelle base de données, située dans **data/database/word_database_test_parasite.txt**. Pour chacune des dimensions étudiées sur des grilles vides (cf figure 6, taille 6x6, 9x8, 12x12 et 14x14), nous allons ajouter 500 mots parasites (issus de la base de donnée principale). Observons ci-après les résultats :

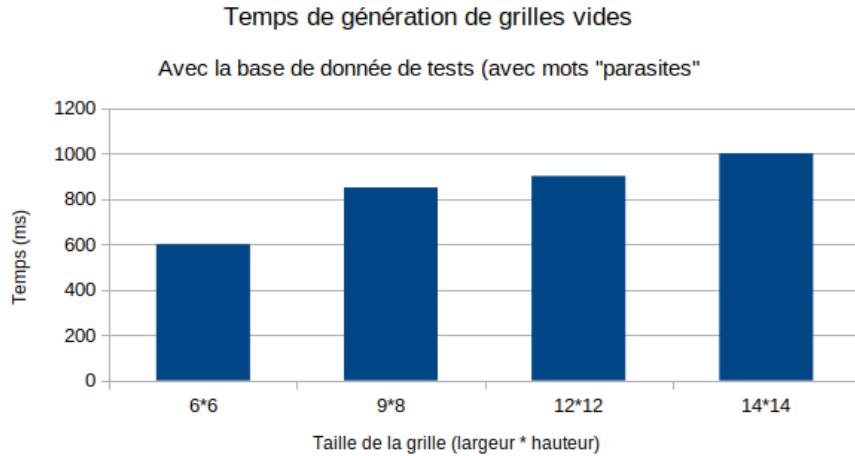


Figure 47: Bilan de performances avec la base de données de test (avec mots "parasites")

Nous observons que les temps sont plus importants que sans mots "parasites" (cf figure 27), en revanche ils restent tout à fait raisonnable (de l'ordre de la seconde). De plus, ils permettent de générer plusieurs grilles, ce qui était manquant avec la base de donnée de test. Cela est ainsi plus réaliste, mais les possibilités de grilles en sorties restent toutefois limitées.

13 Combinatoire, une fatalité sans issue ?

Nous avons exposé en section précédente les tenants et aboutissants de la complexité et (surtout) de la combinatoire associée à nos algorithmes. Finalement, cette phase de tests nous a apporté de l'intuition. Bien entendu, nous ne pourrons pas violer un problème **NP-Complet** en espérant une complexité linéaire, ce serait illusoire. En revanche, nous pouvons accélérer certains traitements. C'est ce que nous avons du moins intuité en observant notre algorithme se dérouler durant cette importante partie de tests.

13.1 Quelques améliorations apportées

13.1.1 Coût de l'application d'une contrainte

Pour rappel, une contrainte sur une variable dans notre problème correspond formellement à: «la variable doit avoir la **lettre** n à la **position** x ».

La méthode que nous avions donc utilisée jusqu'ici se déroulait en plusieurs étapes:

- Parcourir le lexique en regroupant dans un **ensemble** N tous les mots qui ont la **lettre** n à la **position** x ;
- Considérons D : le domaine d'une variable. Nous allons ensuite effectuer une intersection $N \cap D$;
- Cette intersection devient le nouveau domaine de la variable, nous avons uniquement gardé les mots du domaine qui ont la **lettre** n à la **position** x ;

Nous avions remarqué que cette méthode étais très coûteuse, il y a d'abord un parcours du lexique entier (dans notre cas 30 000 opérations) puis l'intersection de deux domaines qui, dans notre contexte, demande d'appliquer l'intersection sur chaque lettre de chaque mot (ce qui donne un nombre d'opération d'environ: **nombre de mots** de taille k dans le lexique * la **taille** k). Nous arrivions très vite à des centaines de milliers d'opérations à chaque fois que nous souhaitions appliquer une contrainte à une variable, cette opération étant le cœur de notre algorithme cela coûte très cher à la performance de notre générateur.

L'optimisation que nous avions mis en place pour réduire ce coût est de ne plus passer par le lexique pour appliquer les contraintes. Nous parcourons directement le domaine de la variable concernée en regardant sur chaque mot directement la lettre qui se trouve à la **position** x et si cette lettre n'est pas la **lettre** n , le mot est enlevé du domaine. Ce qui nous donne, pour une variable de **taille** k , un nombre d'opération qui est d'environ le nombre de mots de **taille** k dans le lexique. Ce qui dans notre cas est au maximum $\cong 6500$.

13.1.2 Mots appliquant la même contrainte sur une variable bloquée

Pour rappel, lors de la génération d'une grille, le backtracking «se déclenche» lorsqu'une variable n'a plus de valeur possible: on revient donc en arrière et change le mot précédent en espérant que les contraintes appliquées par le nouveau mot ne bloquent plus la variable concernée.

Lors des tests nous avons remarqué que beaucoup de mots étaient placés pour rien, par exemple : le mots MANGER vient d'être placé et la variable qui passe par le 'M' du mot MANGER n'a plus de valeur possible, l'algorithme revient donc en arrière et change MANGER par MANIER. La variable concernée se voit réappliquer la contrainte (la lettre 'M' à la **position** x) qui l'a bloqué précédemment. A ce moment là nous perdions beaucoup de temps à tester des mots lors du backtracking que nous savions déjà «mauvais».

La solution a donc été de détecter au moment du backtracking le cas où le mot qui vient d'être placé à appliqué une contrainte qui a bloqué une variable : si c'est le cas nous récupérons la **position** x et la **lettre** n du mot qui a bloqué la variable victime, et nous enlevons tous les mots qui ont cette lettre à cette position, dans le domaine de la variable qui a posé la contrainte problématique.

13.1.3 Les résultats suites aux modifications apportées

Note : Pour cette partie, nous utiliserons la base de données d'origine, celle de taille 31316.

Nous espérons des résultats améliorés en comparaison avec ceux obtenus en figure 23. Afin de mieux se rendre compte des (éventuels) changements, nous réaliserons les expériences sur les grilles où les temps étaient les plus importants, à savoir :

- 6*4 : 5000 ms
- 5*5 : 7500 ms
- 6*5 : 40 000 ms

Les nouveaux résultats sont les suivants :

- 6*4 : 1500 ms
- 5*5 : 3500 ms
- 6*5 : 17 000 ms

Nous constatons qu'avec les quelques modifications apportées, nous gagnons un facteur 2 à 3 en moyenne pour les grilles étudiées ce qui n'est pas négligeable. A ce stade, nous ne voyons plus d'améliorations significatives qui pourraient nous faire gagner du temps sans changer l'architecture globale du projet. Nous pensons avoir mené l'efficacité à bout des algorithmes.

14 Tests aux frontières - limites de l'application

14.1 Conditions d'expériences

Notre programme est capable de générer des grilles de très grandes tailles, nous avons voulu mesurer les limites de ces grilles en temps et en mémoire. Pour cela, nous avons sélectionné une grille difficile, il s'agit de hard5.txt, que nous avons dupliquée jusqu'à la dimension désirée. Une grille générée de 100*100 est disponible dans "data/misc/screen_test_grilles".

Concernant les valeurs obtenues dans ces tests, elles sont une approximation grâce à un calcul de moyenne des valeurs obtenues. En effet de par le processus de randomisation, il est difficile d'obtenir des résultats fiables. Retirer cette randomisation ne nous permettrait pas de résoudre certaines grandes grilles ou alors bien moins efficacement.

Il faut également prendre en compte que les résultats suivants dépendent essentiellement du pattern qui a été choisi ici. Il est possible par exemple de pouvoir générer des grilles bien plus grandes sur des patterns différents où par exemple, on aurait un nombre plus faible de Variables et de contraintes. De plus, ces tests utilisent la base de données de notre application, on pourrait donc imaginer de meilleures performances sur ce type de grille avec une base de données plus adaptée.

14.2 Limites en temps

Pour mesurer le temps pris par notre application, nous avons cette fois-ci fait le choix d'y inclure également le temps pris par notre algorithme de création des variables en plus de celui du solveur. En effet, lorsque nous ne traitons pas des grilles "blanches", le nombre de contraintes et de variables n'est pas aussi important et ainsi le temps pris par l'algorithme de création de variable n'est alors plus si négligeable. Il reste néanmoins plutôt court, mais sur des grandes grilles, il est plus aisé de constater sa proportion par rapport au temps total.

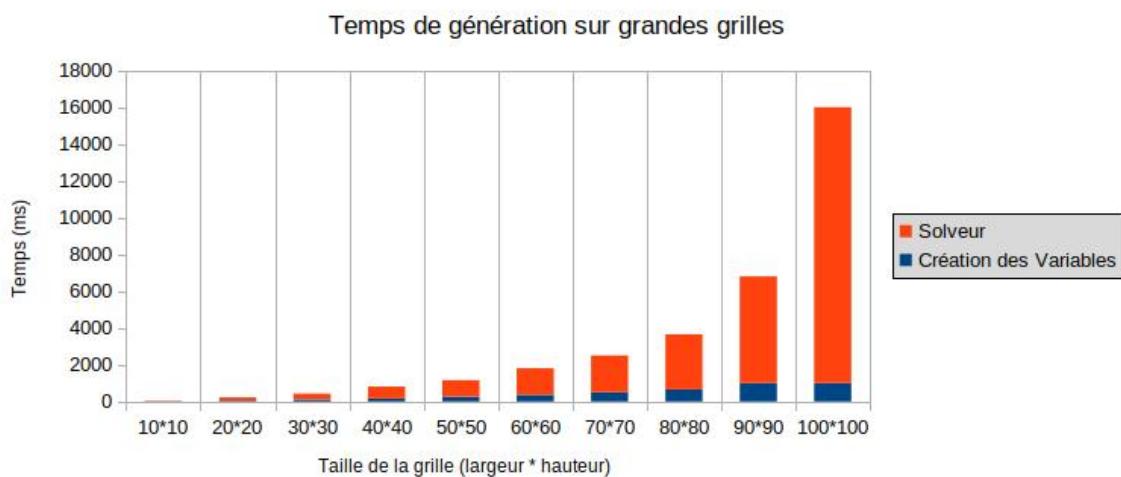


Figure 48: Bilan de performances en temps sur des grandes grilles

On remarque que le temps pris par notre algorithme augmente bien de façon exponentielle, ainsi au bout d'une certaine dimension, il devient complètement impossible de pouvoir générer une grille dans un temps de l'ordre de la minute. Nous avons pu obtenir quelques grilles de dimensions 110*110 grâce à un comportement favorable de la randomisation en une trentaine de secondes, mais ce côté aléatoire de l'algorithme nous empêche de pouvoir obtenir des mesures fiables. La plupart des autres essais ont échoué en dépassant des dizaines de minutes.

14.3 Limites en mémoire

Dans le cas théorique où nous pourrions générer des grilles plus grandes que celles des tests précédents, nous aurions eu une autre limite : celle de la mémoire occupée par les variables que nous manipulons dans notre algorithme. Dans le graphique suivant, on a associé à ces mêmes grilles leur nombre de variables, cela représente le nombre de mots que peut contenir la grille sur de telles dimensions. Nous avons alors pu lancer le processus de génération de grilles en l'arrêtant juste après la phase de création des variables pour étudier l'espace mémoire occupé.

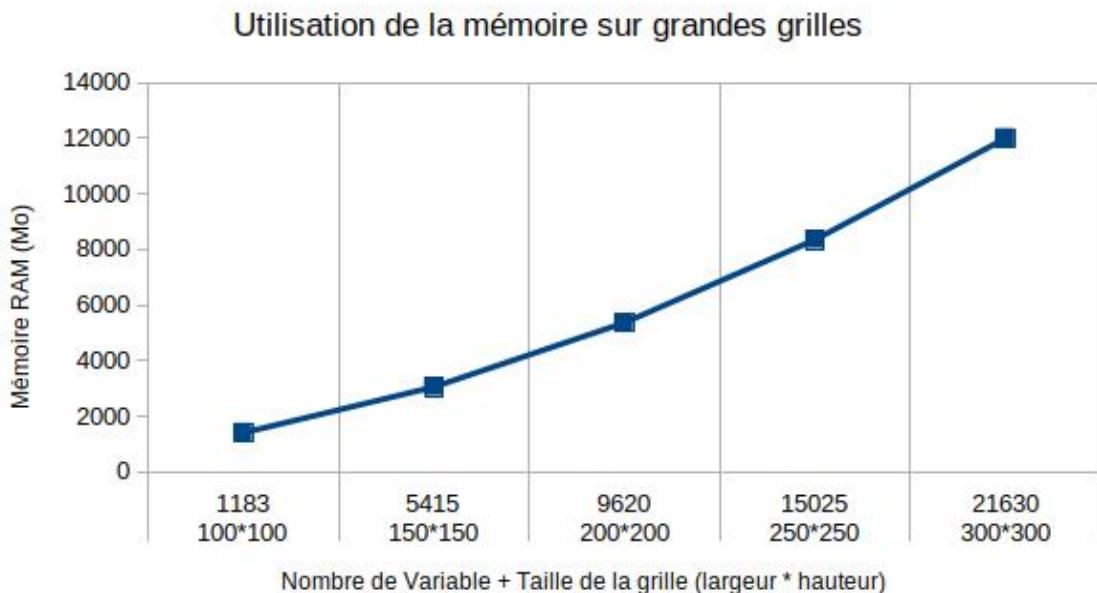


Figure 49: Bilan de performances en mémoire sur des grandes grilles

Sur la machine de test, cette grille de 300*300 est très proche de la limite de ce qu'on pourrait être amené à générer. En effet, une telle grille nécessiterait une quantité de 12 Go destinée uniquement au stockage de ces variables. D'après les calculs, nous estimons qu'une variable nécessite en moyenne 0,5 Mo. Cela est conséquent, car nous avons besoin de sauvegarder de nombreux mots au sein d'une variable. Cependant, lors d'une utilisation classique que nous proposons, on ne manipule pas plus d'une trentaine de variables et ainsi, nous avons besoin de seulement 10 à 15 Mo au maximum pour le bon fonctionnement du programme. Nous aurions pu travailler sur une version différente de notre algorithme permettant ainsi d'allouer et de libérer ces variables au fur et à mesure de leurs utilisations, cela afin de pouvoir traiter des grilles plus conséquentes. Cependant, au vu des priorités de notre application, nous n'avons pas jugé cela utile, car dans l'utilisation de notre programme, ces limites étant bien hautes, nous ne sommes jamais bloqués par cette contrainte d'espace mémoire.

15 Tests concernant la base de données

Notre algorithme d'encodage de la base de données permet une intégration facile de toute autre base de données dans notre logiciel. En effet, il suffit de transformer toute autre base de donnée en un fichier d'extension `.txt` en classant les mots par taille et récursivement au sein de chaque taille par ordre alphabétique (en utilisant par exemple les librairies `numpy` et `pandas` fournies par `Python`). Ainsi, toute autre base de donnée présentée sous ce format là serait intégrée de manière transparente à notre logiciel. Pour rendre notre logiciel plus complet, nous pourrions intégrer un outil d'encodage pour convertir des bases de données existantes en des bases de données **compatibles** avec notre logiciel. Cela pourrait faire d'ailleurs partie des ajouts complémentaires à réaliser.

Par contre, il est de la responsabilité de l'utilisateur de la base de données que sa base soit la plus complète possible. En effet, rappelons que notre base de données est issue d'une vraie base de mots croisés contenant des mots très courants et les plus faciles possibles à placer. C'est en effet de la diversité de la base de données que provient l'efficacité algorithmique résultante. En effet, une base de données contenant peu de mots ou des mots difficiles à placer (contenant par exemple peu de **voyelles**) entraînera un ralentissement important de l'algorithme voir une non résolution de la grille.

A ce propos, nous allons effectuer des tests utilisant diverses tailles de bases de données, afin de montrer le comportement de notre logiciel selon ce point. Pour ce faire, nous allons utiliser la grille suivante :

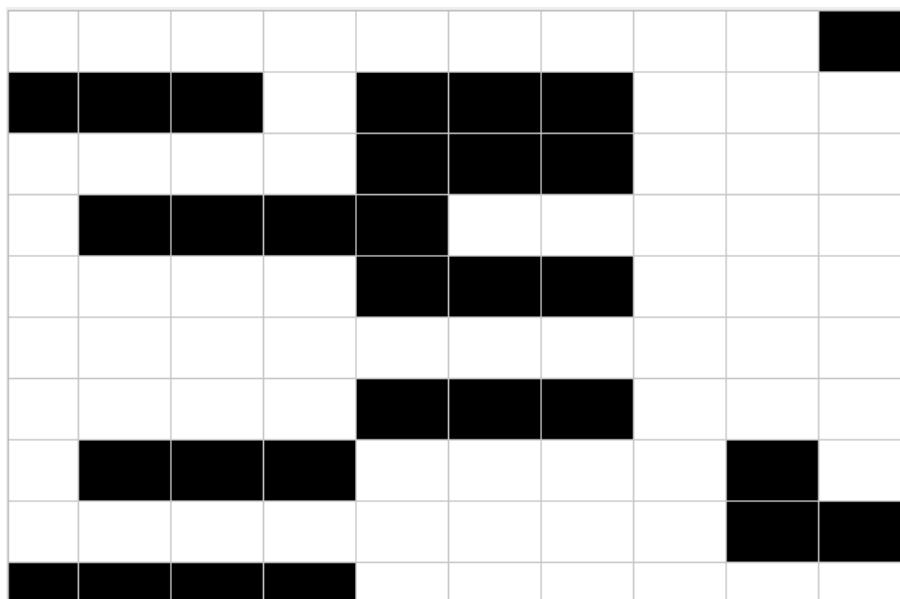


Figure 50: Grille témoin utilisée pour les mesures de variation de taille de la base de données

Ce choix de grille pour ce test a été fait car elle présente des mots de tailles variables s'entremêlant à beaucoup de positions variées.

Voici les résultats :

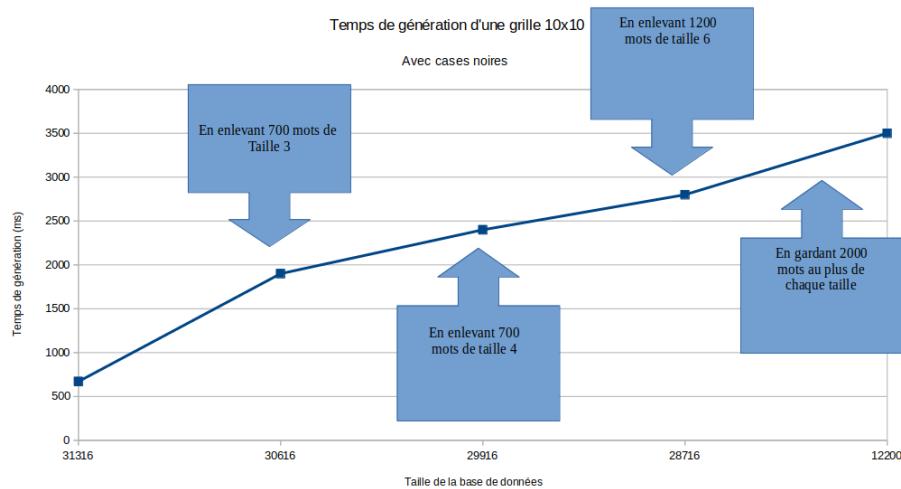


Figure 51: Comparaison de performances sur la grille témoin - variation de la taille de la base de données

Nous observons des résultats conformes à ce que l'intuition nous disait. Ceux-ci sont valables dans le cas où la randomization reste activée, ce qui est le cas ici, autrement nous aurions au contraire des résultats meilleurs. Ce test a pour seule vocation de nous conforter à garder une diversité des mots la plus importante, tout en conservant des performances convenables d'un point de vue utilisateur.

Le bilan que nous faisons ici est que nous avons fait un compromis entre performance et diversité de génération des grilles. Nous avons préféré garder une base de donnée la plus diversifiée soit-elle mais ne pas proposer des grilles vides pour l'utilisateur. C'est ce qui nous a semblé être plus réaliste. En effet avec des grilles vides et dans un temps raisonnable nous n'aurions pu proposer que des grilles de dimensions 5 x 5, ce qui n'utilise pas au mieux notre base de données et qui n'est pas très agréable pour l'utilisateur en terme de diversité produite et donc d'expérience de jeu. Rappelons qu'initialement le but est bien sûr de produire une application logicielle pour un utilisateur, elle doit donc être la plus attractive possible. Néanmoins, cette phase de tests nous a permis de confronter notre application à divers comportements et à diverses bases de données et cela nous a permis de voir (notamment) qu'il est possible de produire des grilles vides en temps tout à fait correct avec une base de données de mots plus adéquate. Les grilles sont par contre moins diversifiées, raison pour laquelle notre application native ne l'utilisera pas. Néanmoins, un développeur pourra intégrer cette fonctionnalité très facilement.

16 Tests unitaires

16.1 Choix de l'outil de test

Après s'être renseignés sur internet sur les différents outils de test compatibles avec le langage C++, nous avons décidé d'utiliser la librairie de GoogleTest [16], qui dispose d'une grande popularité et d'une bonne maintenabilité.

16.2 Implémentation

Chacun de nos tests est appliquée à une fonctionnalité précise de notre application et permet de s'assurer du bon fonctionnement de nos fonctions.

Pour leurs implementations, nous avons réutilisé les tests initialement associés aux besoins tout en ayant adapté leur utilisation à l'implementation finale de notre application.

Voici la liste des tests que nous avons implementé :

Tests positifs :

- Bon fonctionnement de la difficulté choisie (paramètres de dimension et du nombre de points).
- Bon fonctionnement des boutons ("Reveal grid", "Check letter", "Check word", "Validate", "Reset grid").
- Bon fonctionnement du décompte du système de points lors de l'utilisation des aides.
- Ecriture d'une lettre appartenant à [a-zA-Z].
- Importation d'une grille (sera utilisé en tant qu' Oracle dans la plupart des tests).
- Exportation d'une grille.

Tests négatifs :

- Importation d'un fichier non existant ou corrompu.
- Ecriture d'une lettre n'appartenant pas à [a-zA-Z].

17 Gestion de projet

Afin de gérer ce projet, il a été important d'adopter une coopération d'équipe la plus importante possible. De manière générale, nous nous sommes organisés par subdivisions, souvent en sous-groupes de 2 ou 3, sur des tâches bien précises. Nous avons fait attention à ce que les tâches réalisées en parallèles n'aient pas besoin les unes des autres afin d'éviter tout conflit lors des regroupements de code.

Pour la partie technique, bien sûr, nous avons utilisé un dépôt Git (fourni par le Cremi, **GitLab**). Nous avons régulièrement organisé des sessions de code durant lesquelles nous avons pu travailler tous ensemble grâce à Visual Studio Live Share.

Aussi, nous avions un groupe Discord nous permettant de faire des réunions de manière régulière, soit planifiée, soit dès lors que le besoin s'en faisait ressentir. Cela nous a permis d'avancer de manière concertée vers ce que nous souhaitions.

Nous avons commencé l'implémentation du code assez tardivement dans le calendrier : nous avons en effet préféré prendre un peu plus de temps pour bien comprendre les différents PDF et ainsi avoir une idée la plus claire possible avant de commencer à coder. Cela nous a semblé important d'avoir une idée claire concernant le cœur algorithmique et sur le mode d'encodage de la base de données, qui sont les deux points primordiaux pour que le projet se déroule le mieux possible.

17.1 Outils utilisés

Voici les différents outils que nous avons utilisés tout au long du projet :

- Discord pour communiquer entre membres du groupe
- Balsamiq Mockup pour le maquettage
- Lucidchart pour les diagrammes
- Overleaf pour la rédaction commune des documents
- LibreOffice Calc pour la réalisation des graphiques
- Trello pour suivre les différentes tâches à réaliser et les répartir
- Google Drive pour le partage de documents
- Git/GitLab pour partager et versionner le code lors du développement de l'application
- Visual Studio Live Share pour coder en groupe

18 Conclusion

18.1 Bilan technique

Dans l'ensemble notre application atteint les objectifs que nous nous étions fixés lors de la description des besoins. Nous sommes satisfaits de nos résultats, notre application est fonctionnelle. Comme prévu l'utilisateur peut jouer depuis une interface graphique qui est simple d'utilisation et il existe un mode terminal à l'intention des développeurs. La génération des grilles est rapide et notre programme ne requiert aucune connexion internet. Tout a été écrit et codé en anglais.

Il reste cependant quelques fuites de mémoire que nous n'avons pas réussi à identifier et régler.

Nous pouvons aussi noter que nos fichiers base de données qui sont au format txt rendent l'utilisation de l'application sur windows compliquée, si le projet est cloné sur windows : le fichier de la base de données n'est plus utilisable par notre application.

Enfin, notre utilisation de Qt est peu désirable car nous avons utilisé qu'une seule fenêtre sur laquelle on cache et/ou affiche tous les composants tour à tour.

18.2 Ce qu'on a fait, pas fait

En reprenant notre diagramme des cas d'utilisation initial : les besoins "Lancer la génération d'une grille", "Changer de grille", "Choisir un niveau de difficulté", "Jouer", "Exporter en txt", "Choisir une aide", "Valider la grille" et "Manipuler la grille" ont été implémentés. En revanche, les besoins "Choisir un thème", "Exporter en jpeg" et "Exporter en pdf" n'ont pas été implémentés par manque de temps. De plus, ces besoins avaient été classés comme des besoins conditionnels ou optionnels lors de la rédaction de notre cahier des besoins. Nous avons tout de même réalisé l'export en fichier JSON car permettre au moins un moyen d'export nous paraissait important.

Contrairement à ce que nous avions imaginé au début nous n'affichons pas non plus le nombre d'erreurs lorsque l'utilisateur valide la grille, il saura informé si la grille qu'il a remplie est correcte ou non.

Tous les besoins que nous avions classés comme essentiels ont été implémentés. Nous avons donc globalement atteint notre vision initiale de l'application.

18.3 Perspectives et améliorations

Comme décrit plus haut nous avons créé des fichiers texte à la main pour représenter les grilles contenant les cases noires déjà placées. Cela nous a permis notamment d'éviter de laisser la possibilité d'avoir des mots de taille deux sur la grille (notre base de données de mots possibles contient des mots de taille entre trois et quatorze). Des améliorations possibles seraient d'ajouter des mots de deux (et pourquoi pas de 1) à notre base de données ainsi que de pouvoir générer aléatoirement les grilles avec les cases noires qui seront ensuite résolues.

Notre façon de créer les variables pourrait aussi être améliorée car actuellement sur des grandes grilles qui contiennent donc beaucoup de variables nous passons un temps assez conséquent pour la création des variables. Il est même arrivé sur certaines résolutions de prendre plus de temps pour cette création qu'à résoudre la grille.

Enfin, l'utilisation de Qt pourrait être améliorée : actuellement il n'existe qu'une seule vue qui efface et ré affiche tous les composants à chaque changement. Il serait préférable de créer plusieurs vues, une pour chaque partie de l'application.

18.4 Bilan global

Ce projet nous a appris à fonctionner en groupe de taille assez conséquente puisque nous avons du travailler à six pour la première fois. Nous avons beaucoup appris autant en informatique qu'en organisation, conduite de projet et communication au sein d'un groupe. Nous avons bien fonctionné ensemble, nous avons beaucoup communiqué et bien réparti les tâches. Nous nous sommes tous investis dans ce projet et sommes contents de notre travail même si nous avons encore en tête des choses que l'on pourrait améliorer.

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] Ámbulagan and Adi Botea. "Crossword Puzzles as a Constraint Problem". In: 2008. URL: <https://dept-info.labri.fr/~narbel/PdP/Subjects21-22/Crosswords/CP08.pdf>.
- [3] Xinguand Chen Ádam Beacham and al. "Crossword Puzzles as a Constraint Problem". In: 2008. URL: <https://dept-info.labri.fr/~narbel/PdP/Subjects21-22/Crosswords/CP08.pdf>.
- [4] Matthew L. Ginsberg. "Crosswords and an Implemented Solver for Singly Weighted CSPs". In: *Journal of Artificial Intelligence Research* 42 (2011), pp. 851–886. URL: <https://dept-info.labri.fr/~narbel/PdP/Subjects21-22/Crosswords/1401.4597.pdf>.
- [5] The Teacher's Corner. *Crossword Puzzle Maker*. <https://worksheets.theteacherscorner.net/make-your-own/crossword/lang-fr/>. Accessed: 11-03-2022. Last update in 2022.
- [6] *Mots croisés - Sport Cérébral*. <https://www.sportcerebral.com/mots-croises-gratuit>. Accessed: 03-02-2022. Last update in 2022.
- [7] Education.com. *Crossword Puzzle*. <https://www.education.com/worksheet-generator/reading/crossword-puzzle/>. Accessed: 11-03-2022. Last update in 2022.
- [8] Puzzel.org. *Crossword Create*. <https://puzzel.org/fr/crossword/create>. Accessed: 11-03-2022. Last update in 2022.
- [9] Green Eclipse. *EclipseCrossword*. <https://www.microsoft.com/en-us/p/eclipsecrossword>. Accessed: 11-03-2022. Last update in 2016.
- [10] Aragon-Technologies. *Mot croises online*. <https://www.mots-croises-online.com/fr/motscroises/logiciel.php>. Accessed: 11-03-2022. Last update in 2012.
- [11] 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.
- [12] nlohmann. *JSON for Modern C++*. <https://github.com/nlohmann/json>. Accessed: 19-04-2022. 2022.
- [13] Qt. *Qt Cross-platform*. <https://www.qt.io/>. Accessed: 14-03-2022. Last update in 2022.
- [14] Saul Pwanson. *Database sources*. <https://xd.saul.pw/data/>. Accessed: 18-03-2022. Last update in 2022.
- [15] Dimitri van Heesch. *Doxygen: Doxygen*. <https://www.doxygen.nl/index.html>. Accessed: 21-04-2022. 2022.
- [16] GoogleTest. *GoogleTest*. <https://google.github.io/googletest/>. Accessed: 21-04-2022. Last update in 2022.

19 Nos vraies motivations pour la réalisation de ce solver

Pour finir sur une touche humoristique, nous avons réalisé cette application pour aider un des plongeurs de l'équipe, spécialisé en résolution de mots croisés sous-marins. Le voici, en pleine action, début avril, dans les profondeurs du bassin d'Arcachon :

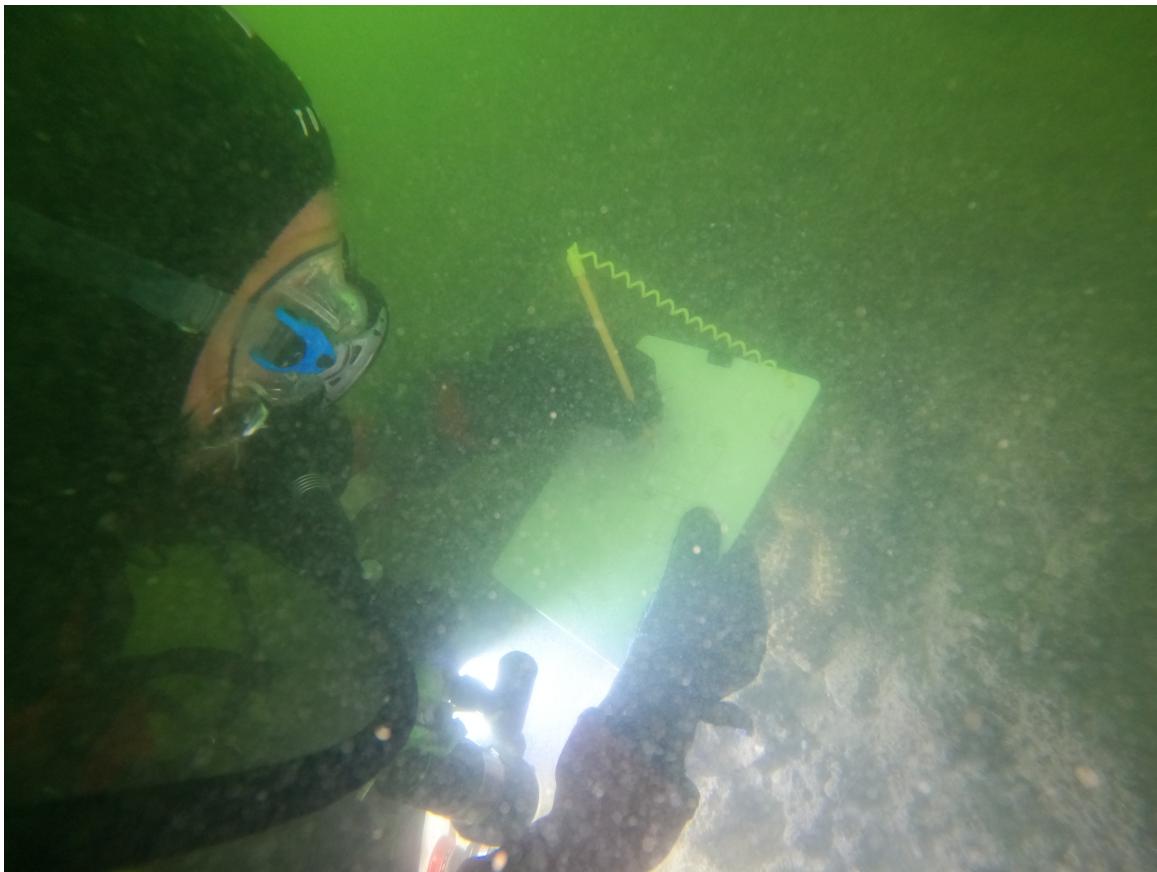


Figure 52: Etude algorithmique en milieu aquatique

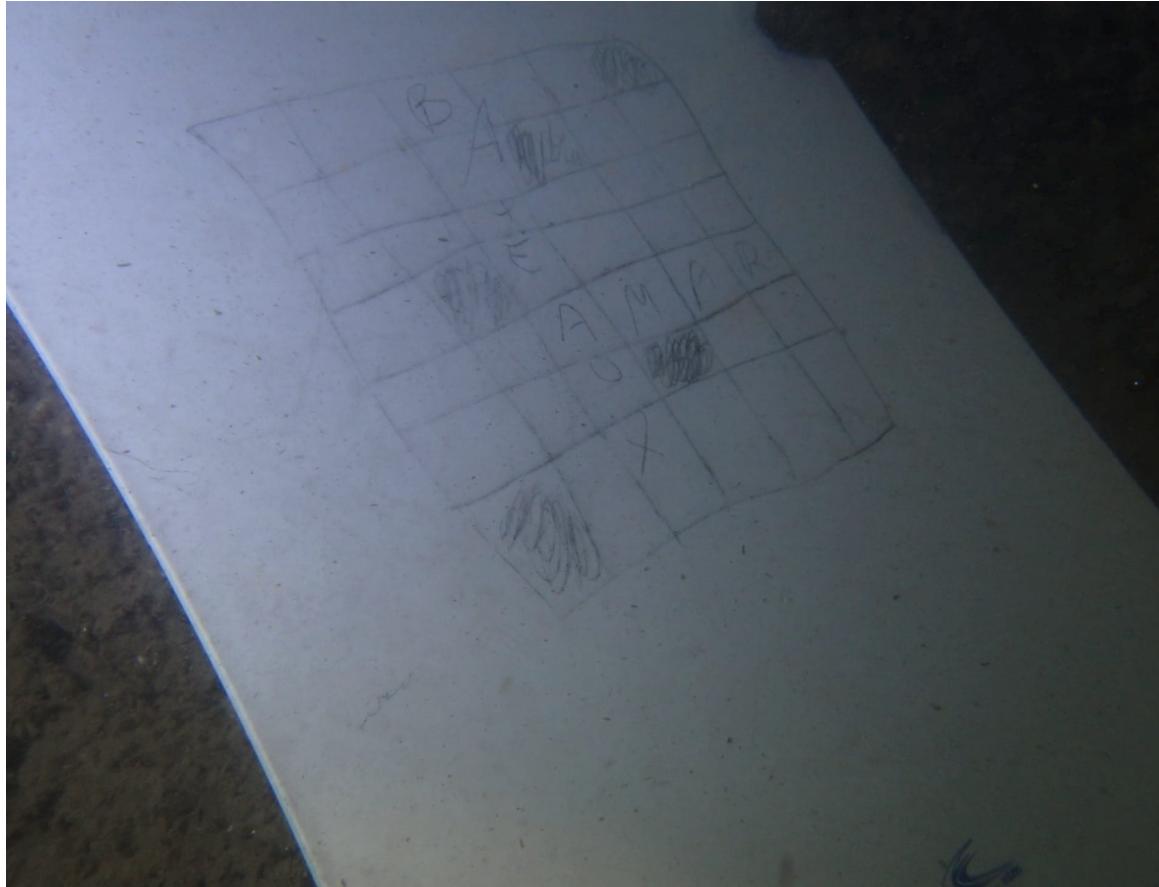


Figure 53: Des performances **largement** inférieures à celles de notre solver

Pour découvrir l'aventure en vidéo, rendez-vous [ici](#).