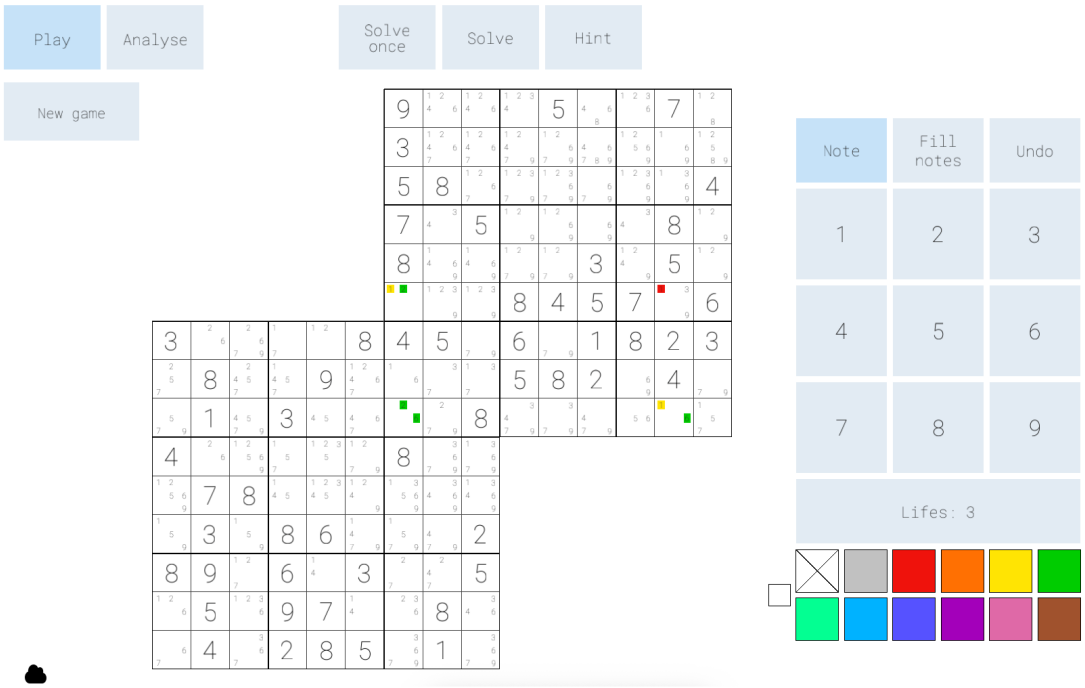
Rapport de projet HAI606I
Génération de grille de Carpet sudoku

Hugo Lopez
Léo Hafdane
Matis Bégot
Rémi Bonnafé

09/05/2025



L3 Informatique
Faculté des Sciences
Université de Montpellier.

Table des matières

1 Introduction

Notre Travail Encadré de Recherche pour cette Unité d'Enseignement *HAI606I* a d'abord consisté à résoudre puis générer des grilles de sudokus simples et d'analyser la difficulté de celles-ci. Le but a ensuite été de générer des grilles de formes spéciales comme un sudoku samuraï (5 sudokus imbriqués en forme de croix), ou un carpet sudoku. Enfin, nous nous sommes demandés s'il était possible de réaliser un pavage infini de sudokus avec une éventuelle périodicité sur un plan réellement infini ou un tore qui donnerait une impression d'infini.

Le suivi de cette unité d'enseignement nous a permis d'améliorer notre capacité de travail en équipe, nos compétences de programmation en Rust ainsi qu'à gérer toutes les étapes de développement d'un projet. L'UE nous a rappelé l'importance de bien structurer notre code pour qu'il soit plus facile à maintenir.

Nous avions aussi à cœur de créer la meilleure application de sudoku, car la plupart des sites étaient soit trop spécifiques dans les variantes et les formes de sudoku, soit incomplets dans les fonctionnalités. Nous voulions donc créer une application proposant un large panel de formes et variantes tout en préservant une expérience utilisateur plus intuitive.

En premier lieu, nous allons aborder comment nous nous sommes organisés lors de cette unité d'enseignement. Puis, nous verrons comment nous avons programmé la résolution et la génération d'un simple sudoku, ainsi que la création de sudokus imbriqués. Finalement, nous parlerons de notre base de données ainsi que de l'application développée.

2 Organisation du groupe

Le début du projet a commencé par le choix d'un langage de programmation, et l'utilisation de **Rust** fut évidente, car c'est un langage avec un typage fort qui mélange performance et fonctionnalités de haut niveau (vecteurs, itérations sur objets...).

L'utilisation de de **Cargo** (gestionnaire de projet de Rust) nous a permis de gérer les règles de compilation ainsi que les dépendances.
Puis, nous avons utilisé Git pour versionner notre projet. De nombreuses branches ont été créées pour chaque fonctionnalité afin de travailler en parallèle, puis tout a été enregistré sur un dépôt distant sur **Github**.

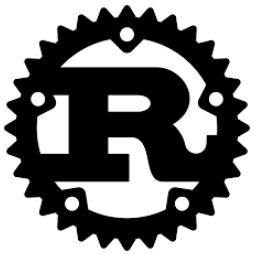


FIGURE 1 – Rust

FIGURE 2 – Cargo

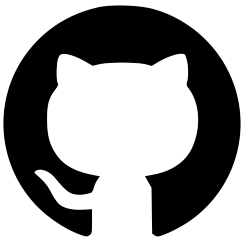


FIGURE 3 – GitHub

Le projet a débuté en janvier et nous avons travaillé au moins cinq heures ensemble durant les créneaux de projet ; chacun a ensuite apporté des contributions individuelles en dehors de ces horaires.

L'objectif étant de réaliser différents types de grilles de sudoku, nous avons commencé par la réalisation d'un sudoku simple, en travaillant sur la structure de ce dernier dans notre code source. Nous avons par la suite réalisé les règles de résolution logiques des sudokus. Une fois la base d'un sudoku établie, nous avons effectué la répartition des tâches suivante :

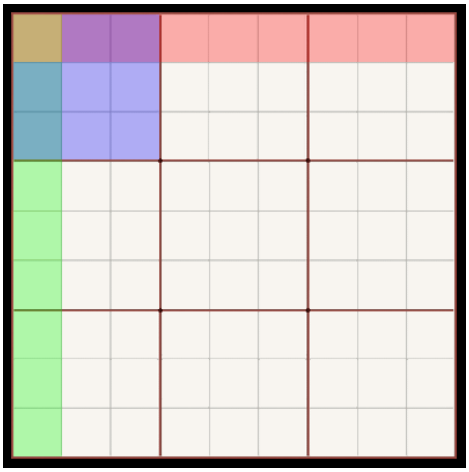
- Matis Begot et Rémi Bonnafe se sont occupés de la partie interface graphique pour pouvoir illustrer nos grilles de sudoku, et ensuite pouvoir réaliser un jeu avec nos grilles générées
- Léo Hafdane et Hugo Lopez ont réalisé les méthodes de résolution et de génération de grilles des différents formats de sudoku.

Des réunions fréquentes ont été effectuées avec notre encadrant pour continuer à nous guider vers la bonne direction et nous faire des retours sur les prototypes fournis. La rédaction du rapport fut tardive, mais a permis à chacun d'entre nous de consacrer du temps à celle-ci.

3 Le simple sudoku

3.1 Comment fonctionne un sudoku

Une grille de sudoku consiste en un carré de taille n^2 où n est généralement égal à 3. Le but d'un sudoku est de remplir entièrement la grille de nombres allant de 1 à n^2, la contrainte étant que tout nombre ne doit être présent qu'une unique fois dans chaque groupe. Un groupe représente ici une ligne de taille n^2, une colonne de taille n^2 ou un carré de taille $n*n$.



Par exemple, si on met le nombre 3 en haut à gauche de la grille, toutes les cases en couleur ne peuvent pas contenir 3.

Une grille à remplir n'est bien évidemment pas vide ; le but est de trouver quels chiffres mettre dans quelles cases à partir d'une grille de départ partiellement remplie (une grille de départ valable appartient à une unique grille pleine) en utilisant différentes règles logiques pour éliminer les possibilités jusqu'à ce qu'il n'en reste plus qu'une seule. La difficulté de la grille initiale est définie par la difficulté des différentes méthodes nécessaires à la résolution du sudoku. Une grille difficile aura généralement plus de cases vides (moins d'informations) mais ce n'est pas forcément lié.

Concernant l'implémentation d'un simple sudoku dans notre code source, voici un diagramme de classes qui l'illustre :

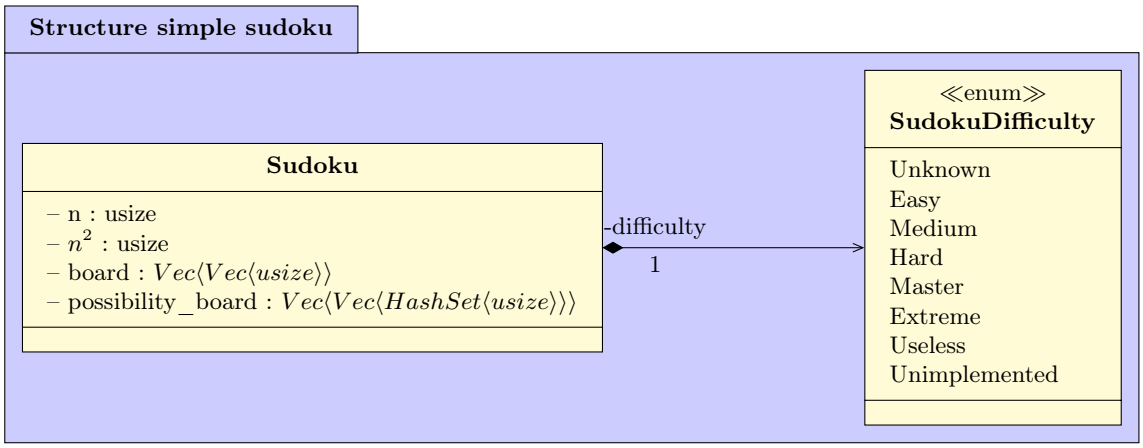


FIGURE 4 – Diagramme de classe de Sudoku.

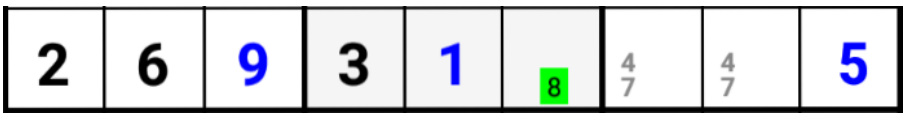
Les variables *board* et *possibility_board* sont des tableaux représentant le contenu et les possibilités de chaque case. Les possibilités d'une case sont représentées par un *HashSet* qui est une liste de valeurs unique. Afin de maintenir le *possibility_board* cohérent, les possibilités sont mises à jour à chaque ajout ou suppression de valeur.

3.2 Les méthodes de résolution

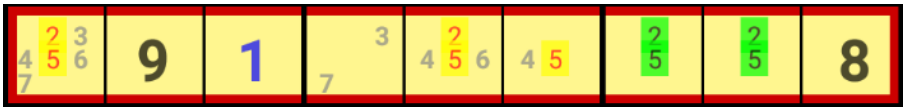
Afin de résoudre un sudoku, il existe un grand nombre de règles logiques plus ou moins difficiles à comprendre et à utiliser. Il est important de noter que les règles et leur difficulté ne sont pas les mêmes partout. Nous avons utilisé comme source de règles le site taupierbw.be/SudokuCoach/ car les règles étaient nombreuses et bien expliquées.

Les images suivantes viennent du site Taupierbw ; dans ces images, les possibilités remarquables sont surlignées en vert et les possibilités à supprimer sont surlignées en jaune.

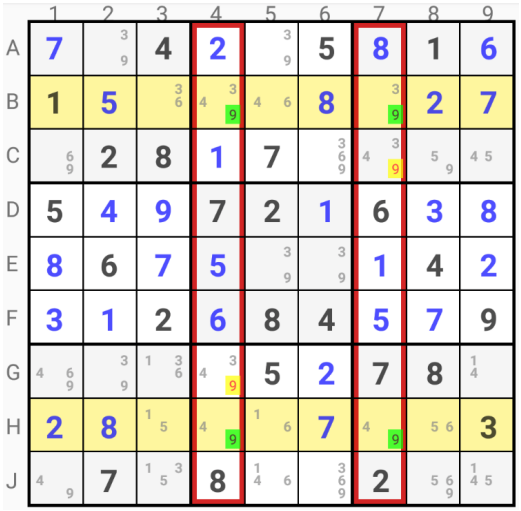
Par exemple, prenons la règle la plus simple *Naked Single* : Si une case ne contient qu'une seule possibilité, alors ce chiffre est forcément dans cette case.



Cette règle permet de mettre un nombre dans une case, mais la plupart des règles permettent seulement de réduire les possibilités, telles que *Naked Pair* : Si dans un groupe il y a deux cases qui possèdent une paire de possibilités identiques, alors ces possibilités peuvent être enlevées des autres cases du groupe.



Ces règles pourraient être trouvées par des débutants uniquement grâce à la logique, mais il existe des règles plus complexes, comme la règle *X-Wing* : Si dans une ligne il existe deux cases qui ont une possibilité commune qui n'apparaît pas dans le reste de la ligne, et que dans une autre ligne, cette même possibilité apparaît dans deux cases des mêmes colonnes que les deux cases de la ligne précédente (de manière à former un rectangle) alors on peut enlever cette possibilité du reste des deux colonnes. Cette règle est aussi applicable en inversant les lignes et les colonnes.



Nous n'avons pas implémenté toutes les règles du site **Taupierbw** car beaucoup d'entre elles sont trop spécifiques et presque jamais utilisées. Nous avons donc implémenté un total de vingt règles. Il n'existe pas de consensus sur le classement de ces règles donc nous avons établi notre propre hiérarchie. À noter l'importance de l'ordre des règles dans une même difficulté, car une règle jugée plus simple qu'une autre sera au-dessus.

Easy	Naked Singles
	Hidden Singles
	Naked Pairs
	Naked Triples
Medium	Hidden Pairs
	Hidden Triples
	Pointing Pair
	Pointing Triple
	Box Reduction
	Naked Quads
Hard	Hidden Quads
	X Wing
	Finned X Wing
	Swordfish
Master	Skyscraper
	Y Wing
	Simple Coloring
Extreme	W Wing
	Franken X Wing
	Finned Swordfish

Pour résoudre un sudoku, nous avons réalisé une fonction qui itère sur les règles dans un ordre de difficulté croissant. l'itération revient à la première règle à chaque modification. Ainsi, la résolution n'utilise pas de règles plus complexes que nécessaire. Cette boucle continue jusqu'à ce que le sudoku soit résolu ou qu'aucune règle ne puisse le modifier.

Nous gardons en mémoire toutes les règles utilisées. La difficulté du sudoku est évaluée comme étant la difficulté de la règle la plus complexe utilisée. Le sudoku se voit également attribué un score de difficulté, qui est la somme des identifiants de toutes les règles utilisées.

3.3 Canonisation d'une grille

Soit deux grilles de sudoku ; on considère que ces grilles sont **équivalentes** si l'on peut transformer l'une en l'autre par une combinaison de ces opérations :

- Permuter des nombres
- Échanger les lignes et les colonnes (rotation 90°).
- Permuter les lignes au sein d'un même bloc.
- Permuter les blocs de lignes.

Nous avons défini un grille canonique comme étant de la forme suivante :

- La première ligne du sudoku est strictement croissante

Avant :

7	9	5	8	4	6	3	2	1
8	1	2	3	7	5	9	4	6
4	6	3	9	2	1	7	5	8

Après :

1	2	3	4	5	6	7	8	9
4	9	8	7	1	3	2	5	6
5	6	7	2	8	9	1	3	4

- Les lignes au sein d'un même bloc sont triées par ordre croissant en fonction de leur case de gauche
- Les blocs de ligne sont triés par ordre croissant en fonction de leur case en haut à gauche

Non canonisé

4	9	5
2	1	8
3	6	7
6	8	2
1	3	4
7	5	9
9	7	6
5	2	1
8	4	3

lignes triées au sein de chaque blocs

2	1	8
3	6	7
4	9	5
1	3	4
6	8	2
7	5	9
5	2	1
8	4	3
9	7	6

blocs triés par première valeure

1	3	4
6	8	2
7	5	9
2	1	8
3	6	7
4	9	5
5	2	1
8	4	3
9	7	6

Ce processus permet d'éliminer toute symétrie horizontale (grâce à la permutation des nombres), et verticale (grâce au tri des lignes au sein d'un même bloc et des blocs entre eux). Il permet donc d'obtenir une seule et même grille à partir de deux grilles à première vue différentes mais en fait équivalentes.

3.4 La génération d'une grille

La première étape est de générer une grille pleine. Plutôt que de créer une grille de manière purement aléatoire puis de la canoniser (ce qui pourrait régulièrement donner la même grille), nous générons dans un premier temps la première ligne et la première colonne déjà canonisées. Nous remplissons ensuite le reste grâce à une méthode de backtrack.

À partir d'une grille de sudoku complète, on retire une case au hasard. Si la grille reste résoluble et que sa difficulté ne dépasse pas le niveau souhaité, on continue à retirer d'autres cases. En revanche, si la grille devient trop difficile ou impossible à résoudre, on revient à l'étape précédente et on choisit une autre case à supprimer.

Ce processus brut étant beaucoup trop long, nous utilisons différents stratagèmes pour accélérer la génération :

- **Passer les redondances :** Puisque la génération découle d'une seule et même grille pleine, le seul critère qui caractérise une possibilité de la génération est le placement des cases vides. On peut exprimer une possibilité de la génération comme étant un nombre à n^4 bits où le *i-ème* bit représente la réponse à la question : "la *i-ème* case est-elle vide ?". Sachant cela, enregistrer la liste des cases vides pour chaque étape de la génération qui n'a pas aboutie et ne pas traiter chaque étape dont la liste de case vide a déjà été vue, évite de traiter plusieurs fois la même possibilité.

  Le nombre de possibilités brute est de $n^4!$. Après cette optimisation, on passe à 2^{n^4}. Le premier nombre s'explique par le fait que la stratégie brut prend en compte l'ordre (enlever la première puis la quatrième case n'est pas la même chose). Le deuxième s'explique par le fait qu'un ensemble de n^4 bits représente 2^{n^4} nombres.

- **Passer les grilles impossible :** Jusqu'à présent, aucune grille de sudoku 9x9 valable avec moins de 17 cases remplies n'a été trouvée *(source)* et il est probable qu'il soit impossible d'en trouver. Donc, dès que nous atteignons cette limite, nous passons à la branche suivante. Après cette optimisation, on passe donc à $2^{n^4} - 2^{17}$ possibilités.

- **Limiter arbitrairement le nombre de fils par nœud :** Nous nous sommes rendu compte que diminuer le nombre maximal de cellules à supprimer à chaque étape rendait la génération parfois bien plus rapide. Nous avons donc posé la limite arbitraire de n^2 fils maximum par nœud de l'arbe des possibilités.

  En théorie, cela rends possible un sénario où la génération ne trouverait pas de solutions. Nous avons donc fait en sorte que la génération explore dans un premier temps avec une limite de n^2 puis avec une limite de $n^2 + 1$ etc... En pratique, notre algorithme n'a jamais repoussé cette limite.

- **Paralléliser l'exploration :** Enfin, chaque branche de possibilité étant indépendante, on peut paralléliser l'exploration.

  Après cette optimisation, supposons qu'on soit sur une machine à K threads, on divise donc le temps théorique par K.

Cette implémentation génère fréquemment des sudokus dont la résolution suit le schéma suivant :
Plusieurs règles très simples sont appliquées, suivies d'une règle de la difficulté voulue puis à nouveau de plusieurs règles simples.

Voici quelques statistiques sur le temps de génération selon la difficulté souhaitée. Elles résultent du temps mis à générer 100 sudokus et ont été effectuées sur une machine à 12 threads :

	Donnée	Brut	Sans redondance	+ Limite minimale	+ n^2 fils	+ Multithread
EASY	Minimum	19ms	21ms	19ms	7ms	31ms
	Maximum	43ms	49ms	53ms	7ms	46ms
	Moyenne	28ms	32ms	32ms	14ms	37ms
	Médiane	27ms	32ms	31ms	14ms	37ms
MEDIUM	Minimum	Immensurable	70ms	52ms	16ms	52ms
	Maximum		3m 10.399s	12m 27.918s	4m 8.406s	1.042s
	Moyenne		13.344s	15.291s	20.243s	148ms
	Médiane		2.572s	3.325s	1.869	107ms
HARD	Minimum	Immensurable	149ms	93ms	37ms	76ms
	Maximum		21m 38.567s	1h 25m 38.835s	1h 3m 28.503s	5.067s
	Moyenne		1m 13.540s	2m 21.197s	2m 52.202s	420ms
	Médiane		23.954s	20.349s	6.359s	166ms
MASTER	Minimum	Immensurable	143ms	131ms	51ms	85ms
	Maximum		1m 54.182s	7m 44.950s	5m 32.069s	702ms
	Moyenne		12.288s	21.266s	20.681s	217ms
	Médiane		4.623s	4.430s	1.603s	160ms
EXTREME	Minimum	Immensurable	263ms	135ms	71ms	116ms
	Maximum		4h 47m 56.848s	21m 15.653s	1h 11m 2.371s	8.566s
	Moyenne		4m 8.136s	1m 44.170s	2m 52.424s	1.113s
	Médiane		28.628s	30.414s	21.607s	480ms

Nous pouvons constater que l'optimisation Limite minimale est négligeable. En effet, lors de la génération, nous arrêtons d'explorer une branche dès que le sudoku actuel n'est pas résoluble. Ainsi, explorer un sudoku ayant autour de 17 cellules pleines est extrêmement rare. De ce fait, l'utilisation de l'optimisation "Limite minimale" l'est tout autant. Les différences de temps observées entre les méthodes Sans redondances et Limite minimale sont donc probablement dues au hasard ou à la taille des échantillons, et non à un effet réel de l'optimisation.

Nous pouvons également voir que les résultats après l'optimisation multithread ont été divisés par plus de 12 (le nombre de threads). Cela s'explique par le fait que puisque l'algorithme de génération consiste en une énorme exploration aléatoire d'arbre, paralléliser cette exploration nous permet aussi d'avoir plus de chances d'explorer une branche présentant une solution "plus évidente".

4 Carpet sudoku

4.1 Qu'est-ce qu'un Carpet sudoku ?

Nous avons défini un *Carpet sudoku* comme étant des sudokus imbriqués les uns dans les autres. Notre implémentation consiste en une liste de sudokus agrémentée d'une liste de liens. Les liens sont dans un premier temps générés comme étant un ensemble de paires $((sudoku_1, carre_1), (sudoku_2, carre_2))$, tel que le lien $((0,0),(1,6))$ signifie que le carré 0 du sudoku 0 est lié au carré 6 du sudoku 1.

À noter que l'indice i du sudoku correspond à sa position dans la liste des sudokus, et qu'un carré d'indice $j = y \times n + x$ désigne une cellule à la position (x, y) dans le sudoku concerné.

Nous trouvons cette implémentation justifiée car elle est suffisamment flexible pour permettre tout type d'assemblage de grilles, y compris des topologies complexes comme des anneaux ou des structures 2D.

4.2 Comment maintenir un ensemble de sudokus cohérent ?

La principale difficulté de cette structure réside dans la cohérence des cellules liées. Une cellule liée existe dans plusieurs sudokus différents, mais elle représente conceptuellement une seule et même cellule. Il est donc crucial que sa valeur et ses possibilités soient synchronisées dans tous les sudokus où elle apparaît.

Pour rendre cela plus compréhensible, nous avons introduit plusieurs concepts :

- **Cellules jumelles** : deux cellules sont dites jumelles si elles sont connectées par un lien. Elles représentent la même entité logique, même si, dans notre structure, elles existent dans plusieurs sudokus différents.
- **Groupe global** : dans un sudoku simple, une cellule est dans trois groupes (ligne, carré ou colonne). Dans un carpet sudoku, les groupes globaux d'une cellule sont l'union des groupes de ses cellules jumelles.

À l'aide de ces concepts, nous avons mis en place les mécanismes suivants :

- Lors de l'ajout d'une valeur dans une cellule, si celle-ci possède des jumelles, la valeur est propagée dans toutes les cellules jumelles. De la même manière, dès qu'une possibilité est supprimée d'une cellule, on la supprime de toutes ses jumelles.
- Lors de la suppression d'une valeur dans une case I, cette valeur est ajoutée à chaque case J des groupes globaux de I si aucun groupe global de J ne possède de cases ayant cette valeur fixée.

4.3 Résolution et génération d'un carpet

La résolution d'un carpet consiste à itérer sur chaque sudoku et à appliquer une règle de résolution, et ce jusqu'à ce que la grille soit complète. Dans notre structure, appliquer une règle logique ne se fait que sur un seul sudoku sans s'occuper des liens. Donc, après chaque étape, nous préservons la cohérence en selon deux principes :

- Les possibilités de chaque cellule jumelle sont réduites à leur intersection.
- Pour chaque cellule, si une de ses jumelles est fixée, alors elle prend la même valeur.

La génération commence un peu comme un simple sudoku. Dans un premier temps, on remplit les premières lignes et colonnes dans un ordre le plus croissant possible (presque canonisation). Puis, on peut générer les grilles pleines à partir de ces ensembles de lignes et colonnes. Durant toutes ces étapes, la cohérence est préservée comme expliqué dans *le point précédent*.

Enfin, nous rajoutons un test de validité avant de soummettre une solution. Pour la pertinence d'un jeu de carpet, on vérifie que chaque sous-carpet n'est pas résoluble. Un sous-carpet étant un sous-ensemble de sudokus liés entre eux :

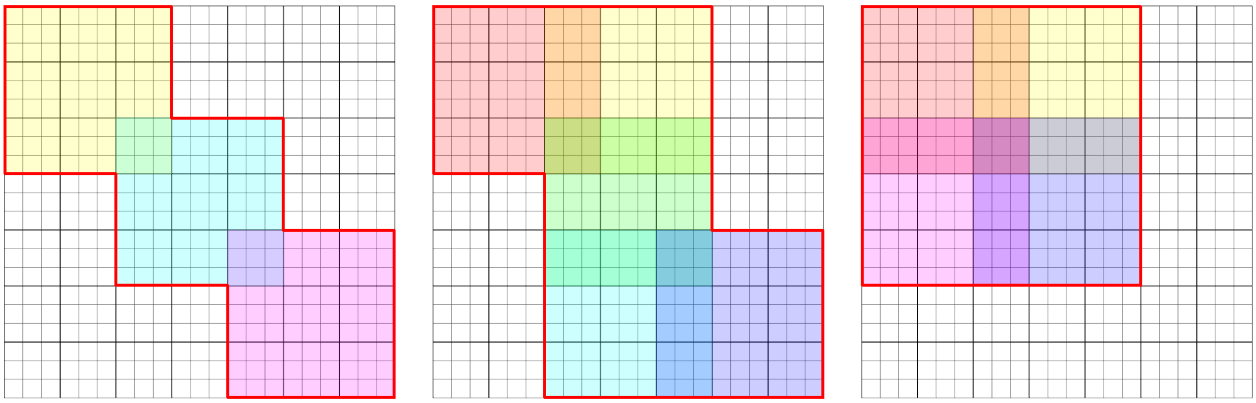


FIGURE 5 – Exemple de sous-carpet dans un carpet de taille 3

4.4 Les différentes formes

Bien que notre implémentation supporte toutes les formes possibles, nous avons introduit certaines formes prédéfinies pour le jeu. Nous pouvons aussi définir une variable N qui augmente la taille d'une forme donnée. Pour chacun de nos carpets, nous avons deux versions : une version normale et une version "dense" (à l'exception du Samuraï). La version dense du carpet imbrique davantage les sudokus entre eux, et crée donc plus de liens. Voici un exemple de la forme **diagonale** :

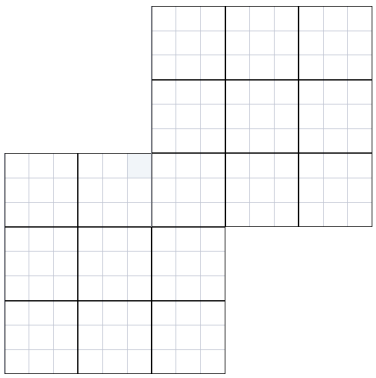


FIGURE 6 – Diagonal taille 2

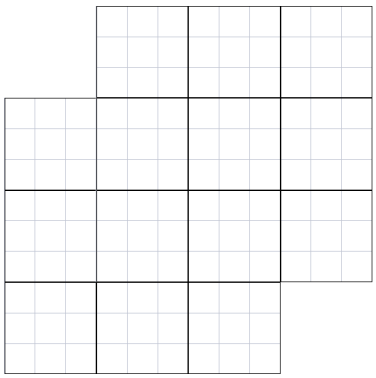


FIGURE 7 – Diagonal dense taille 2

Nous avons aussi implémenté la forme **carpet**, qui réalise un pavage carré de sudokus : ici chaque sudoku est coloré pour une meilleure visualisation.

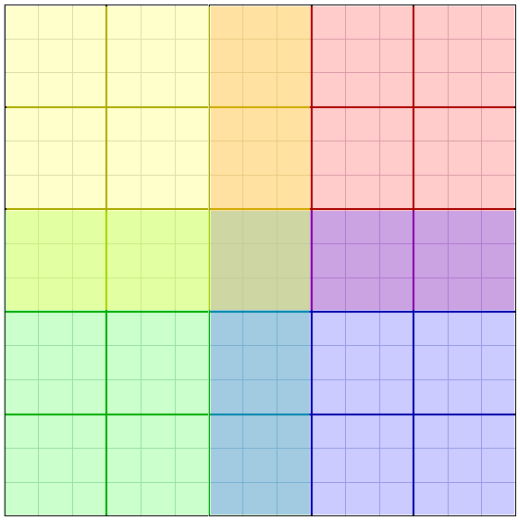


FIGURE 8 – Carpet taille 2 avec ses 4 Sous-sudokus

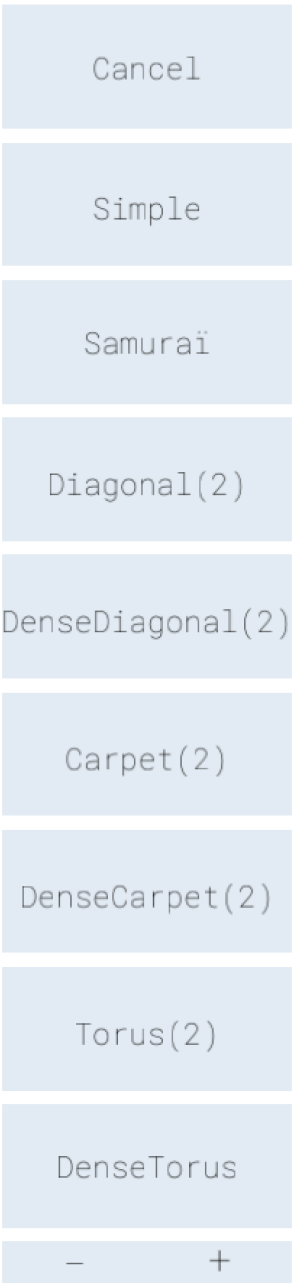
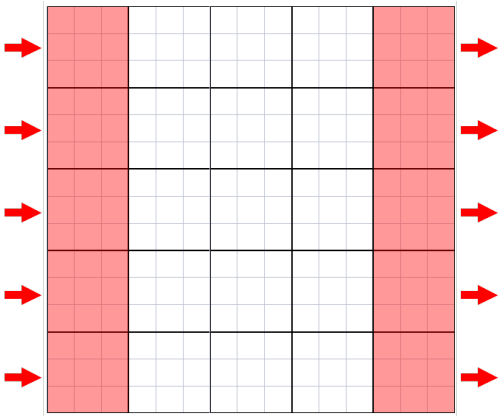


FIGURE 9 – Liste des formes de carpet sudoku

Enfin, nous avons ajouté la forme **Tore**. Cela ressemble à un carpet, à la différence que les sudokus en bordure extérieure ont un lien avec les sudokus de la bordure opposée, ce qui réalise un pavage qui forme une boucle. Nous l'avons réalisé ainsi car replier un rectangle de cette manière donne un tore.



Dans cet exemple, les deux bandes rouges sont en fait équivalentes et forment un lien. Cet exemple de Tore est de taille 2, mais il peut évidemment y avoir plus de sudokus dans la boucle. Cette liaison apparaît aussi pour les deux bandes horizontales supérieure et inférieure. Il existe aussi en variante dense.

La forme de tore dense a quelques spécificités.

- Il est logique qu'un tore dense ne puisse pas avoir de taille $N < n$:
  Posons deux sudokus avec $n = 3$ ayant respectivement a, b, c et b, c, f comme carrés du haut. Supposons que ces sudokus soient liés ? Nous pouvons créer un premier lien dense entre les carrés b, c et d, e mais ne pouvons pas en créer un faisant la boucle qui permettrait le tore où $N = 2$ car il faudrait que $(a, b) = (c, d)$. Or c'est impossible car par définition des sudokus $a \neq c$.

- Il est impossible de créer un tore dense où $N \not\equiv 0[n]$:
  On définit le prédicat $EQ(a, b)$: "les carrés a et b ont les mêmes chiffres dans leur ligne du haut".
  Pour tout $N > n$, la ligne globale des carrés du haut est :
  $s_1, s_2, .., s_n, \quad \{t_1, ..., t_k\}, \quad s_1, s_2, .., s_{n-1}$. avec $k = N - n$
  Les carrés du haut du sudoku tout à droite sont $t_k, s_1, s_2, ..., s_{n-1}$.
  Puisque ceux du sudoku tout à gauche sont $s_1, s_2, .., s_n$, on peut dire par élimination que $EQ(t_k, s_n)$.
  En raisonnant de cette manière de proche en proche on a $EQ(t_k, s_n)$, puis $EQ(t_{k-1}, s_{n-1})$ etc...

  Cela nous donne un sudoku avec $(s_2, ..., s_n, t_1)$ comme carrés du haut.
  — Si $N \equiv 0[n]$ alors $EQ(t_1, s_1)$. Ce qui rend ce sudoku valable.
  — Sinon $N \equiv m[n]$ on a $EQ(t_1, s_{n+1-m})$ Ce qui rend ce sudoku impossible.

  Cette logique s'applique également à toutes les lignes et de même pour les colonnes.

Les tore dense sont donc une suite répétitive de $N + n - 1$ carrés, à quelques permutations de colonnes au sein d'un même carré près. Ils sont donc inutiles. En effet tous les sudokus individuels sont *équivalents*. Un tore dense ne change pas grand chose comparé à un sudoku simple.

5 La base de données

La génération pouvant être très longue, nous avons décidé de mettre en place une base de données afin d'avoir accès à tout type de carpet et simples sudokus en temps réel. Nous avons, dans un premier temps, cherché un moyen efficace et pratique de communiquer avec une base de données en Rust. Notre choix s'est porté sur *Diesel* parce qu'il permet l'interaction avec une base de données sans avoir à écrire du SQL dans Rust et de traduire automatiquement les tables SQL en struct Rust.

Après avoir choisi Diesel, nous avions le choix parmi PostgreSQL, MySQL et SQLite. Nous avons choisi *PostgreSQL* car c'est celui conseillé par diesel et qu'il est utilisé par la majorité de la communauté.

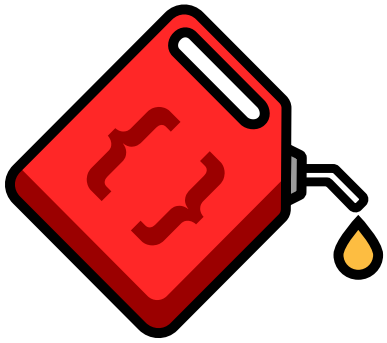


FIGURE 10 – Diesel

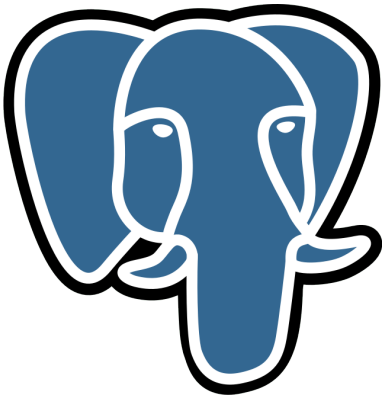


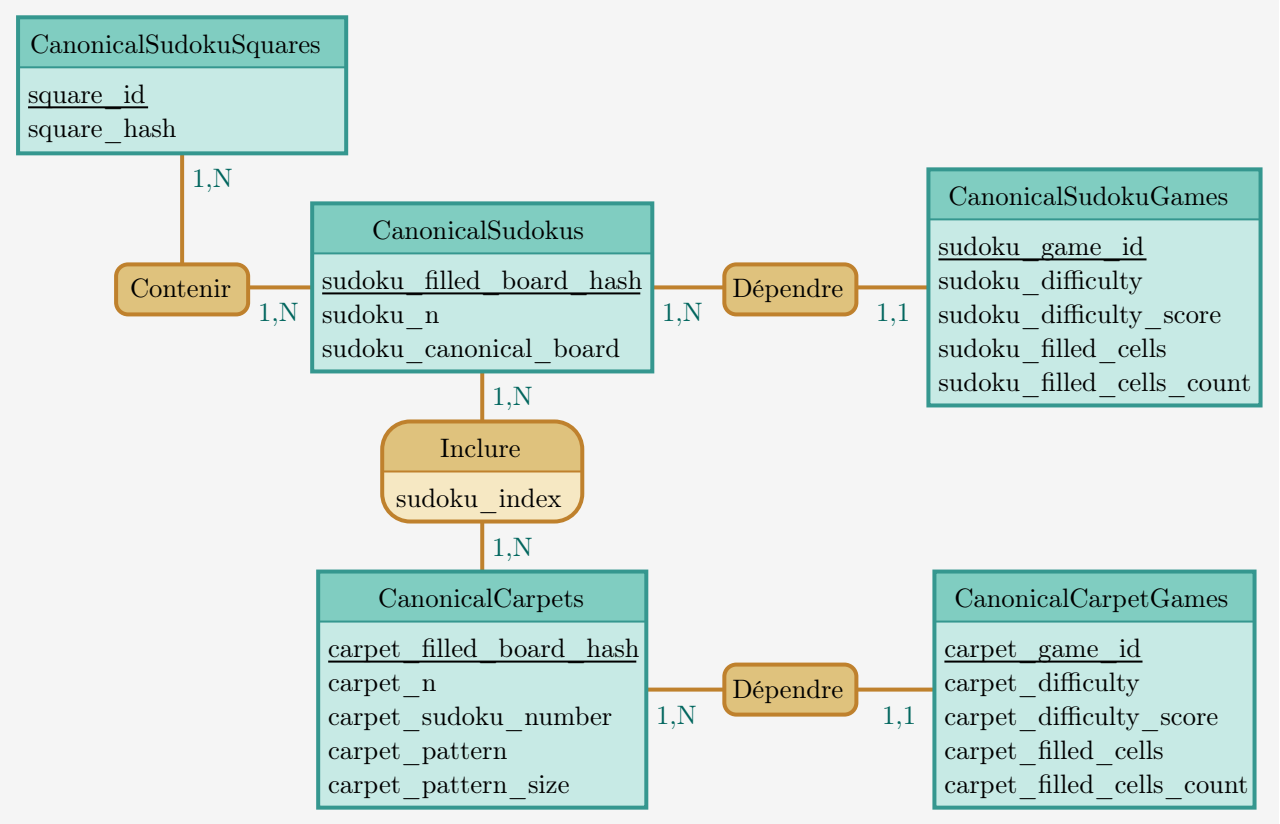
FIGURE 11 – PostgreSQL

Le modèle entité-association et le schéma relationnel ont été générés avec *Mocodo online*.

5.1 Schéma relationnel

- CanonicalCarpetGames (<u>carpet_game_id</u>, carpet_difficulty, carpet_difficulty_score, carpet_filled_cells, carpet_filled_cells_count, #*carpet_filled_board_hash*)
- CanonicalCarpets (<u>carpet_filled_board_hash</u>, carpet_n, carpet_sudoku_number, carpet_pattern, carpet_pattern_size)
- CanonicalSudokuGames (<u>sudoku_game_id</u>, sudoku_difficulty, sudoku_difficulty_score, sudoku_filled_cells, sudoku_filled_cells_count, #*sudoku_filled_board_hash*)
- CanonicalSudokus (<u>sudoku_filled_board_hash</u>, sudoku_n, sudoku_canonical_board)
- CanonicalSudokuSquares (<u>square_id</u>, square_hash, #*sudoku_filled_board_hash*)
- Inclure (#<u>*sudoku_filled_board_hash*</u>, #<u>*carpet_filled_board_hash*</u>, sudoku_index)

5.2 Modèle Entité-Association



5.3 Dictionnaire de données

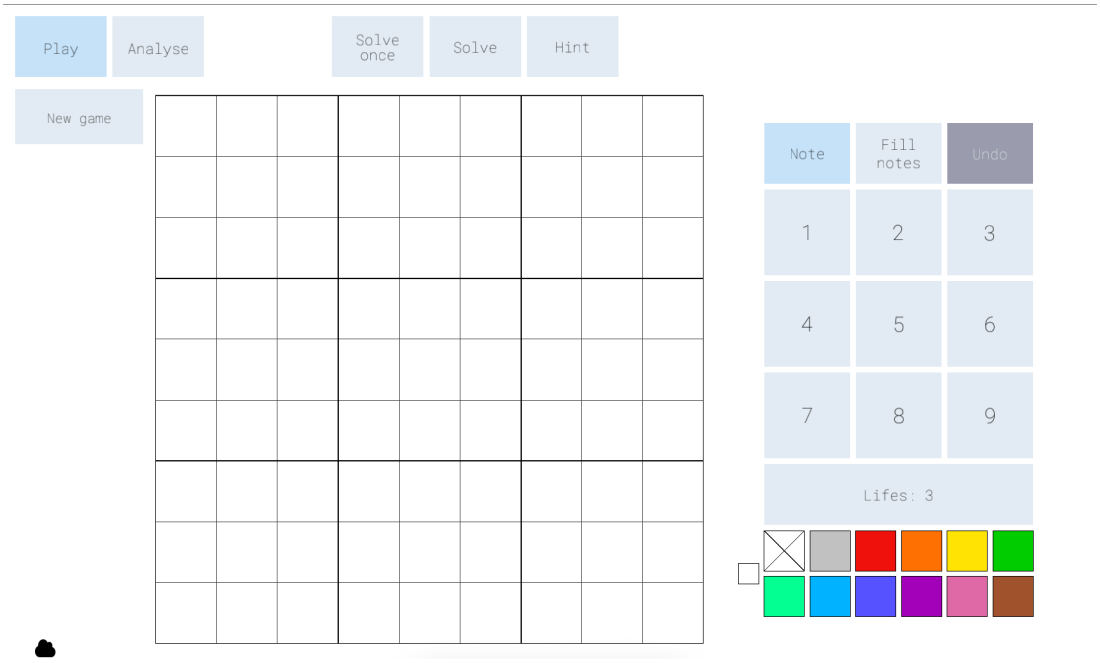
Nom Attribut	Type	Description	Règle de calcul	Contraintes
sudoku_filled_board_hash	BIGINT	hash de la grille pleine canonisée		NOT NULL
sudoku_n	SMALLINT	n du sudoku		NOT NULL
sudoku_canonical_board	BYTEA	buffer de la grille pleine canonisée		NOT NULL
sudoku_game_id	SERIAL	id de la partie	auto incremental	
sudoku_difficulty	SMALLINT	difficulté évaluée		NOT NULL
sudoku_difficulty_score	SMALLINT	score de difficulté		NOT NULL
sudoku_filled_cells	BYTEA	buffer de booléens décrivant quelle cellule est pleine		NOT NULL
sudoku_filled_cells_count	SMALLINT	nombre de cellules pleines		NOT NULL
square_id	SMALLINT	identifiant du carré dans son sudoku		NOT NULL
square_hash	BIGINT	hash du carré		NOT NULL
carpet_filled_board_hash	BIGINT	cf sudoku_filled_board_hash		NOT NULL
carpet_n	SMALLINT	cf sudoku_n		NOT NULL
carpet_sudoku_number	SMALLINT	cf sudoku_sudoku_number		NOT NULL
carpet_pattern	SMALLINT	cf sudoku_pattern		NOT NULL
carpet_pattern_size	SMALLINT	cf sudoku_pattern_size		NOT NULL
sudoku_index	SMALLINT	position du sudoku dans la liste de son carpet		NOT NULL
carpet_game_id	SERIAL	cf sudoku_game_id	auto incremental	
carpet_difficulty	SMALLINT	cf sudoku_difficulty		NOT NULL
carpet_difficulty_score	SMALLINT	cf sudoku_difficulty_score		NOT NULL
carpet_filled_cells	BYTEA	cf sudoku_filled_cells		NOT NULL
carpet_filled_cells_count	SMALLINT	cf sudoku_filled_cells_count		NOT NULL

6 L'affichage graphique avec Macroquad

Pour la partie affichage, nous avons choisi la librairie Macroquad. Initialement prévue pour faire des petits jeux 2D, Macroquad permet néanmoins de faire efficacement des interfaces basiques grâce à ses fonctions qui permettent simplement de dessiner les éléments d'interface.

6.1 Présentation de l'interface

Lorsque l'application est lancée, voici l'interface qui s'affiche :



L'interface est constituée d'une grille de jeu, d'un pavé numérique et de boutons permettant de changer de mode. Les boutons **Play** et **Analyse** servent à alterner entre ces modes.

Le bouton **Solve once** permet d'effectuer des étapes de résolution jusqu'à ce qu'une valeur soit fixée. Il est utilisable dans les deux modes mais en mode Analyse, il affiche quelles règles ont été utilisées sur quel sudoku. Le bouton **Solve** résout entièrement le sudoku. Le bouton **Hint** n'est utilisable qu'en mode *Play* ; il affiche la règle la plus simple utilisable dans un des sudokus actuellement visible.

Pour lancer une nouvelle partie, on clique sur le bouton **New Game** puis on choisit dans l'ordre, la forme et la taille du carpet suivie de la difficulté ou une grille vide. Enfin on choisit entre générer localement la grille ou bien parcourir dans notre base de données si une grille avec ces paramètres existe dans cette dernière.

Le bouton **Note** permet d'activer ou non le mode Note. Dans le mode Note, une palette de couleur apparaît. Cette palette permet de choisir la couleur des chiffres qui seront ajoutés aux notes. La couleur blanche correspond à aucune couleur et permet de les désactiver. Pour ajouter une note colorée, il faut d'abord choisir la couleur, puis cliquer sur le nombre dont on souhaite changer la couleur. Si on clique sur un nombre déjà présent de la même couleur que celle qui est sélectionnée, cela efface la note.

Le bouton **Fill Notes** permet de remplir automatiquement les notes de l'utilisateur avec les possibilités actuelles, lui permettant un gain de temps. Il peut alors se concentrer directement sur la résolution avec les

règles au lieu de passer 15 minutes à remplir les notes des 81 cellules à la main !

Le bouton **Undo** restaure les notes de l'utilisateur à l'état précédent ; cela est très utile si l'utilisateur a fait une erreur, ou si son chat/hamster a marché sur son clavier !

Au-delà de l'affichage, Macroquad prend aussi en charge les interactions au clavier et à la souris. Pour la souris, on dispose de fonctions qui vérifient la position du curseur sur la fenêtre ainsi que les clics effectués. On s'en sert pour colorer la case ou le bouton survolé, ainsi que pour sélectionner une case, ce qui colore ses groupes.

Mais la souris n'est pas le seul périphérique surveillé par Macroquad. En outre, on peut également surveiller les interactions au clavier. Ce dernier permet divers raccourcis, que ce soit pour interagir avec les boutons, déplacer la cellule sélectionnée ou placer des valeurs / notes.

Les flèches directionnelles du clavier permettent de naviguer sur le carpet. La touche Échap permet de désélectionner la case courante afin d'éviter d'éventuelles erreurs. Le pavé numérique et la barre de chiffres au-dessus des lettres permettent d'interagir avec le pavé numérique de l'application.

Les formes **Tore** et **DenseTore** sont un peu particulières. Bien qu'elles contiennent plusieurs sudokus, un seul est affiché. En appuyant sur les flèches directionnelles alors qu'une touche *Alt* est pressée, la vision du joueur se déplace pour afficher le sudoku voisin dans la direction souhaitée.

Quand l'utilisateur clique sur un chiffre, il y a deux cas de figure :

- En mode Note, le jeu va ajouter/retirer la valeur aux notes de la cellule sélectionnée ainsi que la couleur sélectionnée.
- En mode normal, le jeu vérifie que la valeur mise corresponde bien à la valeur de la correction. Si c'est le cas, le jeu met à jour les cellules. Si la valeur proposée est erronée, l'utilisateur perd une vie, la case est colorée en rouge et il n'est plus possible de proposer de valeur pendant un court instant afin d'éviter d'éventuelles fautes de frappe en cascade (et permettre une introspection et à l'utilisateur de méditer sur ses erreurs dans son sudoku et dans sa vie).

6.2 Fonctionnement technique

Pour notre interface, nous avons choisi de faire une classe SudokuDisplay qui prend des attributs utiles à l'expérience utilisateur, tels que :

- Une instance de CarpetSudoku sur laquelle le jeu va se baser
- Une grille de notes et de couleurs que l'utilisateur pourra modifier à souhait grâce à l'interface
- Un historique des notes/couleurs afin que l'utilisateur puisse revenir en arrière s'il a fait une erreur dans ses notes
- Une grille cachée du sudoku résolu afin de vérifier si la proposition de l'utilisateur est valide
- Une variable qui représente le nombre de vies restantes à l'utilisateur
- Un tableau contenant les informations de tous les boutons
- Un dictionnaire qui associe chaque bouton à son action.

Pour afficher une grille nous avons créé une fonction qui génère l'affichage des différentes parties d'un sudoku dans cet ordre :

1. La grille
2. Les couleurs ajoutées par l'utilisateur
3. La cellule sélectionnée ainsi que les autres cellules des groupes auxquels elle appartient
4. La cellule survolée
5. Le texte des cellules (valeur ou notes/possibilités)

L'ordre d'affichage est important car lorsqu'on dessine un nouvel objet et qu'il y a déjà un élément à cet endroit de l'affichage, le nouvel objet sera affiché par-dessus et masquera ce qu'il y a en dessous.

Pour enregistrer les notes, nous avons créé un tableau de profondeur 3 de dictionnaire. Les profondeurs correspondent respectivement au sudoku, à la ligne, à la colonne et au dictionnaire, associant les possibilités de chaque case à leur couleur.

L'historique des notes est composé d'un tableau de notes. Lorsqu'on apporte une modification aux notes, on duplique ce tableau et on le ajoute dans le tableau d'historique. Lorsque l'utilisateur clique sur Undo, on vérifie que ce tableau ne soit pas vide et on "pop" la dernière valeur (on la récupère et on la retire du tableau), puis on l'assigne aux notes actuelles.

Pour le Solve Once, on appelle la méthode de résolution du CarpetSudoku pour modifier les possibilités jusqu'à ce que l'on ait ajouté une valeur. Ensuite, on récupère un tableau qui indique quelles règles ont été appliquées sur quel sudoku, ce qui permet de faire un affichage de cette résolution en mode Analyse.

7 Conclusion

Au bout de ce projet, nous avons globalement répondu à nos attentes initiales. Nous avons réussi à générer des sudokus de nombreuses formes et tailles, et tous ces sudokus sont des grilles jouables sur une application ergonomique, ouverte aussi bien aux amateurs qu'aux experts.

Avec plus de temps, nous aurions pu mettre en place plusieurs mécanismes supplémentaires :

Nous avons réfléchi au plan infini de sudoku ; nous pouvions prolonger un carpet de forme **Carpet**, mais au détriment de la suppression de la partie devenue invisible (cela rendait parfois la prolongation impossible). Nous aurions pu garder en mémoire les parties non visitées et ainsi pouvoir se déplacer sur un plan infini, mais le développer aurait pris trop de temps, donc nous nous sommes concentrés sur d'autres objectifs.

Nous aurions également pu améliorer la génération de sudokus avec, par exemple, la méthode de *look-ahead*, qui - comme son nom l'indique - examine les possibilités des branches à une certaine profondeur et évalue si le *score* du sudoku est intéressant. Nous avons pu implémenter la notion de score, mais pas la technique de look-ahead. Cette dernière aurait permis de générer plus souvent des sudokus présentant plusieurs règles de la difficulté voulue.

Nous tenons à remercier notre encadrant, M. Éric Bourreau, pour nous avoir concocté un TER passionnant, aussi ludique que complexe, ainsi que pour son accompagnement tout au long du développement.

8 Annexe

- **Github du projet :** https://github.com/Leottaro/hai606i_sudoku
- **Manuel d'utilisation :** https://github.com/Leottaro/hai606i_sudoku/blob/main/README.md

Table des figures