RAPPORT DE PROJET

Sommaire :

I. Préambule :

Nous avons consacré beaucoup de temps à la réflexion d'implémentation des deux parties des plus importantes du projet : le générateur et le solveur. Ce premier nous a appris à bien visualiser les conditions d'existence d'une pièce ainsi que de son orientation pour éviter que la grille ne soit pas solvable. Nous nous sommes inspirés non seulement de ces idées pour commencer à coder notre solveur, mais également d'intuitions que nous avons développé grâce aux cours d'IA. Nous expliquerons cela en détail plus tard. C'est ce qui nous a permis de résoudre de manière optimale des grilles allant jusqu'à 200x100 en très peu de temps. Nous vous expliquerons ensuite le déroulement du projet et nos choix dans la structure du code et l'implémentation des méthodes.

NB : il n'est pas utile de regarder nos classes tests afin de vérifier le bon déroulement de l'exécution des consignes. Nous vous proposons d'exécuter selon les commandlines du main, nous vous expliquerons en détail comment vérifier de cette manière, grâce aux images GUI.

II. Répartition du travail :

Tout d'abord, il faut savoir que ce travail a été effectué sans aucune inégalité d'implication entre les deux membres du binôme. Nous avons certes choisi de ne pas utiliser Git qui aurait été très intéressant si nous n'avions pas les mêmes disponibilités de travail ainsi qu'une répartition différente des tâches. Cependant, nous avons toujours réfléchit, débattu nos idées et codé ensemble en appel sur la plateforme Discord, sur nos fichiers respectifs, et mis en commun à chaque fin de session de travail. Si l'on veut plus de précision, Salim a écrit les versions finales de GUI et Generator alors que Soufiane a codé celles de Solver et Components. Cependant, même si l'on écrivait des choses différentes, nous partagions toujours nos idées et nous entraidions en partageant un bout de code permettant de débloquer l'autre en cas de problème. Comme la mise en commun était constante, l'utilisation du Git nous aurait probablement posé un problème vu que nous ne travaillions jamais seuls, comme précisé plus haut.

III. Base du programme

Nous avons commencé par compléter le code donné en regardant quelles étaient les méthodes utilisées non implémentées. Il suffisait de lire les commentaires écrits signalant ce qu'il fallait implémenter sur telle méthode. Jusque-là il n'y avait pas de difficultés si ce n'est la cohérence avec les méthodes déjà implémentées. C'était d'ailleurs une aide plus qu'un obstacle, à notre sens.

IV. Generator

La première chose à laquelle nous avons réfléchit est la manière dont nous allions générer une grille de manière cohérente, pour qu'elle puisse absolument être résolue, car nous nous sommes rendu compte qu'en générant seulement des pièces aléatoires, nous tomberions toujours sur une grille insolvable. La stratégie était donc ici de ne pouvoir choisir qu'entre certaines pièces de manière aléatoire, tout en gardant de la cohérence. Le but ici sera donc de générer une grille déjà résolue avant d'aléatoirement mettre une nouvelle orientation à nos pièces pour bien sûr proposer une grille non résolue. Nous générons donc une grille de taille demandée en l'initialisant avec des pièces VOID. Pour chaque pièce que nous devons créer, nous regardons sa position, son voisin de gauche et son voisin du haut. En effet, en créant pièce par pièce, une pièce est assurée d'avoir un voisin à gauche et en haut (VOID inclus) sauf si elle se trouve en bordure ou en coin. Pour cela, nous avons implémenté les méthodes suivantes dans la class Grid:

- *allPieces* intialise une liste des 16 couples (pièce/orientation) possibles.
- *leftNeighborFilter, topNeighbotFilter* renvoient une liste des couples possibles en regardant si respectivement les voisins de gauches et du haut ont un connecteur vers notre pièce courante.
- *borderFilter, cornerFilter* renvoient une liste des couples possibles si on se trouve respectivement en bordure ou en coin.
- *interFilter* fait l'intersection des listes renvoyées par les filtres de voisin et de bordure/coin.

Exemple : nous sommes au coin haut / droite de la grille et nous avons comme voisin de gauche un ONECONN SOUTH. Sachant que nous sommes en coin haut / droite nous n'avons droit qu'à un VOID NORTH, LTYPE SOUTH, ONECONN WEST / SOUTH. Cependant, le voisin de gauche ne lui donne pas de connecteur, donc son filtre renvoie VOID NORTH, ONECONN NORTH / EAST / SOUTH, BAR NORTH, TTYPE EAST et LTYPE NORTH / EAST, car la pièce courante ne doit donc non plus pas lui donner de connecteur. En faisant l'intersection, il nous restera seulement VOID NORTH et ONECONN SOUTH. La pièce courante sur cette position sera ainsi choisie aléatoirement parmi les deux possibilités.

Ensuite nous nous sommes intéressés au Nbcc. Pour implémenter cette option, nous avons d'abord réfléchit à la façon de créer une grille formée d'un seul composant connexe. Pour cela, nous avons simplement fait en sorte que dans le *interFilter*, un appel d'une méthode visant à resserrer encore plus le filtre se fasse. Il s'agit de *nbcFilter*. Celle-ci prend en argument notre filtre final et retire le ONECONN EAST/NORTH et LTYPE WEST sous deux conditions qui doivent être vérifiées ensemble :

- Si nous avons le choix d'utiliser d'autres pièces, car sinon, nous nous retrouvons avec un filtre vide.
- Si le composant est susceptible d'être fermé (par exemple, si nous avons deux connecteurs, un LTYPE WEST est interdit si l'on a le choix car nous ne donnons pas la possibilité pour son futur voisin de droite de continuer le chemin de la composante connexe).

Pour finir, dans le generator, nous concaténons en quelques sortes le nombre de grilles nécessaires pour atteindre le nombre nbcc demandé par l'utilisateur (seulement s'il en demande un), grâce à une boucle qui travaille par portion de grille.
La génération du fichier n'est pas pertinente à expliquer, nous avions juste utiliser l'interface NIO2 et suivi le format demandé en implémentant une méthode *buildFile*.

V. Checker

Cette classe fut relativement simple à implémenter. Tout comme pour *buildFile*, les détails de code ne sont pas nécessaires. A la différence de cette dernière, nous lisons cette fois un fichier, encore à l'aide de l'interface NIO2, puis générons la grille correspondante. Nous vérifions à l'aide de la méthode statique *isSolved* qui prend en paramètre un fichier (à ne pas confondre avec la méthode de même nom sur la classe Grid qui elle n'est pas statique et ne prend pas de paramètre), appelle *buildGrid* pour générer la grille et vérifie si tous ses éléments sont connectés, c'est-à-dire si tous les éléments ont tous leurs connecteurs liés.

VI. Solver

Nous arrivons à la partie la plus intéressante. L'idée du générateur nous a beaucoup aidé car nous avons compris que parfois (même très souvent), une seule orientation est possible. Les différences sont qu'à la génération, nous avons le choix entre plusieurs pièces alors qu'à la résolution, le type de pièce est déjà choisi. De plus, nous avons implémenté deux nouveaux filtres, *rightNeighborFilter* et *bottomNeighborFilter* qui ont la même utilité que les deux autres filtres, mais cette fois en regardant respectivement les voisins de droite et du bas. Par exemple, un ONECONN entouré d'un voisin lui donnant un connecteur le forcera à être orienté vers celui-ci, et un TTYPE entouré d'un voisin ne lui donnant pas de connecteur le forcera d'être orienté à l'opposé de celui-ci.

Avant même de résoudre notre grille, nous appliquons une sorte de résolution partielle dans laquelle nous **fixons** toutes les pièces qui ont une orientation certaine sans erreur possible. Nous avons implémenté pour cela plusieurs méthodes :

- *fixPiece* qui cette fois fait appel à tous les filtres implémentés, mais en regardant seulement les voisins déjà fixés car l'orientation des voisins non fixés n'est pas certaine et donc ne doit pas être déterministe de l'orientation de notre pièce courante. Nous regardons les orientations possibles d'une pièce, et si une seule possibilité subsiste, nous fixons la pièce qui aura une orientation certaine et ne sera pas changée par le solveur. Si la pièce n'a plus d'orientation possible c'est que la grille n'est pas solvable et qu'il faut abandonner, vous en verrez l'utilité plus bas. Si plus d'une possibilité subsiste, nous lui mettons l'une de ses orientations possibles sans la fixer.
- *fixAllPieces* fixe toutes les pièces possibles de la grille, cependant nous utilisons sa deuxième version *fixAllPieces2* beaucoup plus efficace et rapide. La première réitère sur la grille tant qu'on réussit à fixer au moins une nouvelle pièce, car il y a partout de l'interdépendance. Cependant nous nous sommes rendu compte que cette dépendance n'existe que dans le voisinage d'une pièce. Pour la deuxième version, nous nous sommes inspirés de l'idée de l'algorithme AC-3 de notre cours d'IA sur les contraintes de domaines de variable car nous avons en fait que des contraintes binaires entre une pièce et son voisin (par rapport à si le voisin regardé donne un connecteur ou non). En fait, nous initialisons une queue comprenant toutes nos pièces de la grille, mises en couple avec une liste de leurs voisins (initialisée par notre méthode *initListNeighbors*, nous ne prenons par la méthode déjà écrite car nous avons même besoin des voisins à qui on ne donne pas de connecteur). Jusqu'à ce que la queue soit vide, à chaque itération nous essayons de fixer une pièce qu'on pop de la queue avec sa liste de voisins. Chaque fois que nous arrivons à fixer, nous ajoutons dans la queue tous les couples pièce/voisins des voisins de la pièce courante que nous venons de fixer. Dans les deux versions, si la grille n'est pas solvable à un moment donné, nous nous arrêtons car la résolution sera impossible.

Ainsi, avant même la résolution du solver, nous remarquons que même sur les grilles de grande taille, nous résolvons environ entre 80 et 90% de la grille de manière certaine et instantanée.

Maintenant, place à la résolution finale. Nous initialisons un stack avec la grille de départ. A chaque itération, jusqu'à ce que le stack soit vide, une grille est pop du stack et nous choisissons une pièce à fixer en choisissant nous-même l'orientations (cette fois, contrairement à la résolution partielle, l'orientation mène peut-être à quelque chose de faux). Pour le choix de cette pièce nous avons implémenté plusieurs méthodes. Mais avant, expliquons ce que fera le solver de cette pièce choisie. Celle-ci est fixée sur ses orientations possibles (déjà restreintes grâce au *fixPiece*) et pour chacune d'entre elles une copie de la grille est push dans le stack. A savoir que nous avons implémenté notre propre *copyGrid* pour être sûr de pouvoir copier aussi les champs isFixed de la pièce, solvable de la grille et fixedPieces de la grille qui compte le nombre de pièce fixées, car il sera utile plus tard et permet de ne pas avoir à itérer sur la grille chaque fois qu'on veut compter ce nombre.

Ce qui se passe ensuite est vraiment ce qui nous a permis d'optimiser notre solver le plus possible. Pour chaque copie, avant de push dans le stack, nous appliquons notre *fixAllPieces* sur la copie, à partir de la pièce que nous venons de fixer nous-même. Le fait d'avoir fixé cette pièce va faire que les voisins d'à côté vont pouvoir probablement être fixés, et grâce à la récursivité d'ajout de voisins dans notre *fixAllPieces*, nous pourrons fixer un nombre conséquent de pièces de manière sûre à partir d'un seul choix de pièce.

Notons que si la copie devient non solvable après l'appel de la méthode, il n'y aura pas de push de cette copie, cela permet d'économiser un nombre très conséquent d'espace mémoire (avant d'utiliser cette ruse, nous étions bloqués à la résolution d'une grille 30x30). Sinon, avons de push, nous vérifions si la grille est résolue. Si elle ne l'est pas, on pop la dernière copie à la prochaine itération et ainsi de suite.

Dans tous les cas, l'appel de *fixAllPieces* va nous permettre de :

- Libérer plus d'espace mémoire en éliminant plus vite les grilles non solvables.
- Nous rapprocher plus vite d'une solution en fixant le plus de pièces possibles avant de push

Voici maintenant les méthodes de sélection de pièce :

- *pieceSelection_first* prend en paramètre un booléen car en fonction de ce dernier nous utilisons deux versions différentes de cette méthode. La première (booléen à false) choisit la première pièce non fixée. La deuxième version quant à elle, après la sélection, trie la pièce de manière croissante en fonction de ses orientations en regardant combien de pièces va fixer la méthode *fixAllPieces* à partir de l'orientation choisie. Nous faisons cela car après le push, le pop prendra la dernière copie, donc celle dans laquelle on a fixé l'orientation classée en dernier, ce qui, théoriquement, permettrait de se rapprocher plus vite d'une solution ou d'une grille non solvable. Cependant il n'y a pas de grande différence de performances entre ces deux versions. Ce sont les meilleures de nos choix de pièce. La compilation est très rapide. Les grilles 100x100 sont résolues en moins de 3 secondes en moyenne.
- *pieceSelection_random* choisit de manière aléatoire une pièce non fixée.
- *pieceSelection_best* a également deux versions. Dans les deux cas nous choisissons une heuristique permettant d'estimer à quel point fixer une pièce impactera le nombre de pièces que *fixAllPieces* va fixer. Dans les deux cas nous prenons la pièce qui impactera le plus les autres, avant de trier les orientations de la même manière que plus haut. Dans la première version (booleen à false) nous prenons comme heuristique la moyenne de ce nombre de fixage de toutes les orientations de la pièce, alors que dans la deuxième, nous choisissons comme heuristique le maximum parmi les orientations de la pièce. Le problème de cette sélection et que, bien qu'elle permettrait d'optimiser l'espace mémoire encore plus, elle met beaucoup trop de temps en termes de complexité. Donc inutile sur les petites grilles car notre pieceSelection_first le fait déjà quasi instantanément, et non efficace sur les grosses grilles qui prennent beaucoup trop d'espace mémoire pour la première méthode de sélection (à partir de 200x200) car le temps d'exécution est beaucoup trop grand.

VII. GUI

Bien que nous nous soyons aidés de codes sources déjà existant pour notre classe GUI, nous avons quasi tout implémenté par nous même à l'aide de vidéos youtube et des documentations sur oracle sur les méthodes des différentes classes utilisées (JFrame, JPanel, JButton etc). Il y a certaines exceptions lorsque nous avions besoin d'un bout de code permettant de debugger une petite partie, exemple : petit bout de code *createTransparentImage* et *createTransparentIcon* qui ont permis de créer une image transparante pour l'icône VOID. Sinon, la plupart s'est fait grâce à notre propre conception et réflexion.

Il y a deux utilités principales à notre GUI. Pour le tester il suffit d'exécuter sur le main de la classe Main avec le generate et une grille s'affiche en nouvelle fenêtre (marche aussi avec l'option solve mais la fenêtre ouverte montre directement la grille résolue).

La première utilité est de faire jouer l'utilisateur. Il peut cliquer sur les pièces représentées par des icônes sur la fenêtre de jeu et celles-ci tournent grâce à un MouseListener en classe anonyme. S'il résout la grille en jouant, un message sur la console s'affiche pour prévenir qu'il gagne. Si le joueur rencontre des difficultés, il peut appuyer sur le bouton bleu, cela va fixer une pièce (si c'est possible) de manière certaine et la colorier en vert, ce qui revient à donner un indice à l'utilisateur. S'il touche à cette pièce, elle n'est plus fixée. Le joueur peut aussi faire un clic droit sur une pièce quand il est sûr de sa position, ce qui va la fixer et la colorier en bleu ciel (pour différencier le fixage fait par l'ordinateur et lui-même), il peut revenir sur son choix en refaisant un clic droit.

La deuxième utilité et de tester le solver. Il y a en effet un bouton rouge à côté de la grille. En faisant un clic gauche, elle se résout partiellement grâce à notre *fixAllPieces* en coloriant en vert les pièces fixées. Avec un clic droit, elle résout complètement la grille avec notre solver, sans qu'on ait besoin de jouer.

Notons qu'à chaque mouvement d'image, nous mettons à jour la pièce concernée de la grille afin de ne pas rencontrer de problème au niveau des fixages de pièce. Aussi, nous générons un fichier chaque fois que l'utilisateur tourne une pièce, ce fichier est examiné grâce aux Checker, et c'est ce qui permet d'écrire que l'utilisateur a gagné dès que c'est le cas.

VIII. Conclusion

Pour conclure, nous avons identifié les différents axes d'améliorations de notre projet. Premièrement, concernant la résolution, nous n'arrivons pas à résoudre des grilles de taille 200*200 ou plus, cela est dû à une saturation du heap space de notre stack. En effet, grâce à l'optimisation de notre algorithme de résolution, nous n'avons pas de problème de temps de résolution mais un problème de mémoire pour les grilles trop grandes. Nous avons donc compris que nous avons atteint les limites de mémoire de notre thread, et pour pouvoir résoudre ce problème, nous aurions dû implémenter une résolution avec plusieurs threads. Un autre axe d'amélioration concerne les images, malgré un effort considérable de notre part pour réaliser des images sachant que nous sommes des novices dans cet aspect de la programmation, l'interface des images peut certainement être amélioré avec plus de boutons et de panels, etc. Nous sommes globalement très satisfaits de notre projet et espérons que notre interface graphique vous plaira et vous amusera !