Maxime Boyer
Marie Magny
Mickaël Nabet

Rapport du projet : Infinity Loop

Table des matières

Les components

L'énumération Orientation

L'énumération Orientation nous permet d'associer une direction d'une boussole à une pièce. Il y a 4 orientations possible :

- NORTH ;
- EAST ;
- SOUTH ;
- WEST.

A chacune de ces orientations est associé un entier nommé compassDirection, respectivement 0, 1, 2 et 3. Outre un getter sur compassDirection, il y a plusieurs fonctions disponibles dans cette énumération.

Tout d'abord, la fonction getOrifromValue(int orientationValue) permet de récupérer une orientation comme Orientation.NORTH en fonction de l'entier entre 0 et 3 passé en paramètre.

La fonction turn90() permet quant à elle de récupérer l'orientation qui est celle d'après en tournant à 90 degrés dans le sens des aiguilles d'une montre. Par exemple turn90() alors que l'orientation était Orientation.NORTH à la base nous retournera Orientation.EAST.

Enfin, la fonction getOpposedOrientation() permet de récupérer l'orientation opposée à celle actuelle à 180°. Par exemple, pour Orientation.NORTH la fonction retournera Orientation.SOUTH.

L'énumération PieceType

L'énumération PieceType permet d'associer un type de pièce à une instance de la classe Piece. Il y a 6 types possibles avec chacun un nombre de connecteurs associés et un entier intValue qui est un identifiant :

- VOID : 0 connecteur, intValue = 0
- ONECONN : 1 connecteur, intValue = 1
- BAR : 2 connecteurs, intValue = 2
- TTYPE : 3 connecteurs, intValue = 3
- FOURCONN : 4 connecteurs, intValue = 4
- LTYPE : 2 connecteurs, intValue = 5

Cette énumération implémente quelques fonctionnalités assez intéressantes comme la méthode getListOfPossibleOri() qui retourne une liste d'Orientation possibles étant donné le type de la pièce actuelle.

Une autre méthode qui est setConnectorsList(Orientation orientation) permet de générer une liste de d'orientations qui contiendra les directions vers lesquelles pointent les connecteurs de cette pièce en fonction de son type et de l'orientation passée en paramètre.

Enfin, la fonction getTypeFromValue(int typeValue) permet de récupérer le type de la pièce correspondant à l'entier (entre 0 et 5) passé en paramètre. Par exemple, si 1 est passé en paramètre, la fonction retournera PieceType.ONECONN.

Nous avons réalisé des tests JUnit des méthodes que nous avons implémentées dans ces deux classes : getOrientation(), getIntValueTest(), getListOfPossibleOriTest(), setConnectorsListTest(), getNbConnectorsTest(), getTypefromValueTest() pour celles de l'énumération PieceType dans la classe PieceTypeTest. De même, des tests des fonctions getOpposedPieceCoordinates(), getCompassDirection(), getOrifromValue(), turn90(), getOpposedOrientation() ont été implémentées pour l'énumération Orientation dans la classe OrientationTest.

Le Générateur

La classe Generator possède plusieurs fonctions.

Tout d'abord il y a la fonction generateLevel(String filename). Cette fonction va appeler generateGrillFilled() pour obtenir une grille solvable et qui est donc dans l'état résolu. Elle va ensuite redonner aléatoirement une orientation à chaque pièce avant d'écrire cette pièce dans un fichier .txt portant comme nom filename. Le fichier contiendra finalement la hauteur et la largeur de la grille solvable ainsi que les caractéristiques de chaque pièce : son type ainsi que son orientation.

Ensuite, la fonction generateGrillFilled() va s'occuper de la création de chaque pièce. Elle va commencer par regarder les voisins en haut et à gauche de cette nouvelle pièce pour voir s'il faut ou non que cette pièce soit connectée avec elles. On utilise pour cela 2 entiers left et top qui prendront la valeur 1 s'il faut se connecter en haut ou à gauche, 0 sinon. Cela va ensuite nous permettre d'utiliser une autre fonction qui nous retournera une liste de types de pièces possible avec ces contraintes : possiblePieceType(Piece p, int top, int left).

Cette fonction va retourner une liste de type(s) de pièces possibles pour cette pièce compte tenu du nombre minimum et maximum de connecteurs qu'elle doit avoir. Elle va pour cela regarder si la pièce est sur la dernière ligne et/ou la dernière colonne ou si c'est une autre pièce de la grille. En effet, pour la dernière ligne/colonne, il n'y a pas de variable (comme left ou top) pour indiquer qu'il ne faut pas de connecteur vers le bas / la droite.

Enfin, de retour dans generateGrillFilled() avec la liste de type(s) de pièces possibles, on va appeler une dernière fonction pour choisir le type et l'orientation de la pièce, qui est choosePieceTypeAndOrientation(Piece p, ArrayList<PieceType> possiblePieceType). Cette fonction va tester dans un ordre aléatoire les types de pièces et orientations possibles jusqu'à en trouver une qui fonctionne grâce à la fonction isValidOrientationForGridCreation(int line, int column). Cette dernière va vérifier que la pièce ne se connecte pas dans le vide en haut et à gauche, ou qu'elle n'oublie pas de se connecter à un voisin de gauche ou d'en haut qui veut se connecter. Elle va laisser la pièce avoir un connecteur vers la droite sauf si c'est la dernière colonne, et va également laisser la pièce avoir un connecteur vers le bas sauf si c'est la dernière ligne.

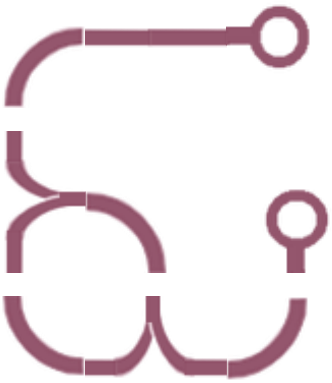
Prenons un exemple de déroulement pour la génération d'une grille 3x3. Tout d'abord une instance de Generator sera créée avec une grille de la classe Grid associée de taille 3x3.

Ensuite, on va parcourir la grille comme on lirait un texte pour y mettre des pièces. On commence donc par le coin en haut à gauche, qui n'a pas de voisin en haut et à gauche donc left et top seront égaux à 0. On va tester tous les types de pièces possible et ne garder uniquement ceux ayant entre 0 et 2 connecteurs, soit VOID, ONECONN, LTYPE et BAR. Ensuite, comme on est sur un coin on va retirer BAR car c'est impossible d'avoir une barre dans un coin. On va donc tester dans un ordre aléatoire VOID, ONECONN et LTYPE. On va tester LTYPE en premier. On va maintenant tester toutes les orientations jusqu'à en avoir une qui marche. Il se trouve que seule l'orientation EAST marchera avec 2 connecteurs au sud et à l'est donc on gardera cette pièce pour le coin en haut à gauche.

Pour la deuxième pièce de cette première ligne, il faut donc entre 1 et 3 connecteurs. On va donc garder comme type de pièces ONECONN, LTYPE, BAR et TTYPE. On va aléatoirement commencer par examiner BAR et choisir la seule Orientation possible qui est EAST avec 2 connecteurs à l'est et à l'ouest. S'il n'y avait pas eu d'orientation possible pour le type BAR, la fonction aurait examiné un autre type (pris aléatoirement parmi les types restants) ainsi que ses orientations.

Pour la troisième pièce de première ligne, il faut entre 1 et 2 connecteurs. On va donc garder comme type de pièces ONECONN, LTYPE et BAR. On va ensuite supprimer BAR de cette liste car on est dans un coin. Et on va tester dans un ordre aléatoire les types de pièces possible en commençant par ONECONN. La seule orientation possible pour ONECONN est WEST car le voisin de gauche a un connecteur droit.

Et on continue avec la même méthode pour les deuxièmes et troisièmes lignes. A la fin, on pourrait obtenir une grille comme celle-ci :



Nous avons réalisé un test Junit dans la classe GeneratorTest, sur la méthode possiblePieceType() : possiblePieceTypeTest() pour vérifier que chaque élément de l'arraylist retournée est correct.

Le Solveur

Le solveur a été implémenté comme un arbre de recherche grâce à une fonction récursive : SolveGrid() de la classe Solve. Cette fonction prend en paramètre les indices i et j qui correspondent à la ligne et à la colonne de départ. Pour lancer l'algorithme, il faut donc passer en paramètre les indices 0, 0. La grille est ensuite parcourue dans le sens de lecture. La pièce correspondant à la ligne et à la colonne étudiée est récupérée.

On commence par enlever les orientations de la pièce qui ne sont pas possibles par rapport aux voisins : c'est le rôle de la fonction eliminatePossibleOrientation(). On ne regarde que les voisins du haut et de gauche car ce sont ceux dont l'orientation a déjà été déterminée. Pour les voisins de droite et du bas, on regarde juste s'il y a une pièce ou pas (pièce VOID) et qu'il n'y ait donc pas de connecteur(s) dans ce(s) direction(s). Les pièces VOID des voisins de gauche et de haut sont aussi vérifiées. Si l'orientation d'une pièce implique d'avoir un connecteur en haut ou en bas alors que le voisin correspondant n'a pas de connecteur, alors l'orientation est supprimée de l'attribut « possibleOrientations » de Piece. De même si, à l'inverse, une orientation n'implique pas de connecteur en haut ou à gauche alors que son voisin en a un vers lui. Il ne doit pas y avoir non plus de connecteur vers un bord de la grille : par exemple, un connecteur vers le haut alors que la pièce se situe sur la première ligne. Toutes ces vérifications sont effectuées par la fonction isValidOrientationSolve().

Plusieurs cas peuvent alors se présenter :

- Il ne reste qu'une position possible : dans ce cas, la pièce n'a pas d'autre choix que de prendre cette position. La pièce est donc déclarée comme fixée.
- Il reste plusieurs positions possibles : la pièce prend une orientation parmi celles qui sont possibles -> utilisation de la fonction turnFromPossibleOrientation().
- Il ne reste aucune position possible : dans ce cas c'est que l'une des pièces précédentes a été mise dans une mauvaise orientation. On retourne donc en arrière vers les pièces précédentes jusqu'à trouver une pièce non fixée. Chaque pièce parcourue dans ce retour en arrière est

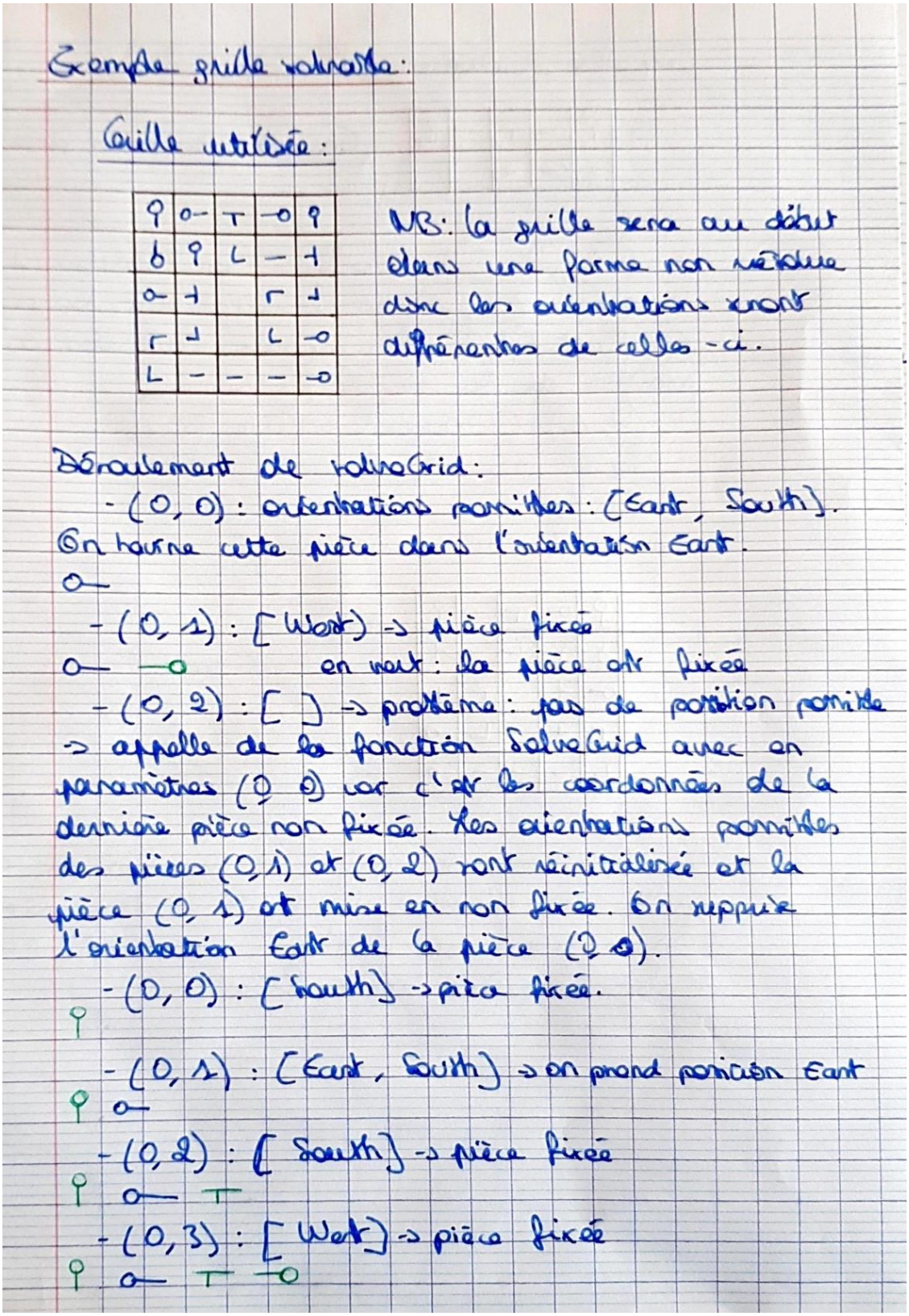
réinitialisée au niveau des attributs possibleOrientations et isFixed. Une nouvelle fois, deux cas sont possibles :

- On retourne en arrière jusqu'à la première pièce (la pièce se trouvant à la position (0,0)) sans trouver de pièce non fixée. Dans ce cas, cela signifie que la grille ne peut pas être solvée. On retourne donc false.
- On trouve une pièce non fixée. Cette pièce non fixée n'est pas réinitialisée comme les précédentes. Cependant, on enlève de son attribut possibleOrientations l'orientation dans laquelle cette pièce était car elle ne permettait pas de résoudre la grille. On relance alors la fonction SolveGrid en passant en paramètre les indices de la position de la pièce non fixée.

Si une grille a été résolue par la fonction, alors la solution est revérifiée une dernière fois avec la fonction isSolution() de la classe Checker. Si cette fonction confirme que la grille est bien résolue, alors on génère un fichier correspondant à cette solution grâce à la fonction GenerateFileFromGrid() de la classe Grid. La fonction retourne true.

Dans la classe SolverTest, Il y a deux tests Junit qui permettent de tester la fonction solveGrid() : SolveGrillNonSolvableTest() pour tester que la fonction retourne bien false pour une grille non solvable et SolveGrilSolvableTest() pour tester que la fonction retourne bien true pour une grille solvable. Il y a aussi un test Junit permettant de tester eliminatePossibleOrientation() : eliminatePossibleOrientationTest().

Voici un exemple d'implémentation de la fonction SolveGrid sur une grille solvable :



Exemple grille solvable :

Grille utilisée :

⚲	o-	T	-o	⚲
b	⚲	L	—	T
a	T		r	⌟
r	⌟		L	-o
L	—	—	—	-o

NB : la grille sera au début dans une forme non résolue donc les orientations seront différentes de celles-ci.

Déroulement de SolveGrid :

- (0, 0) : orientations possibles : [East, South].
On tourne cette pièce dans l'orientation East.

o—

- (0, 1) : [West] -> pièce fixée

o— —o en vert : la pièce est fixée

- (0, 2) : [ ] -> problème : pas de position possible
-> appelle de la fonction SolveGrid avec en paramètres (0, 0) car c'est les coordonnées de la dernière pièce non fixée. Les orientations possibles des pièces (0, 1) et (0, 2) sont réinitialisée et la pièce (0, 1) est mise en non fixée. On regarde l'orientation East de la pièce (0, 0).

- (0, 0) : [South] -> pièce fixée.

⚲

- (0, 1) : [East, South] -> on prend position East

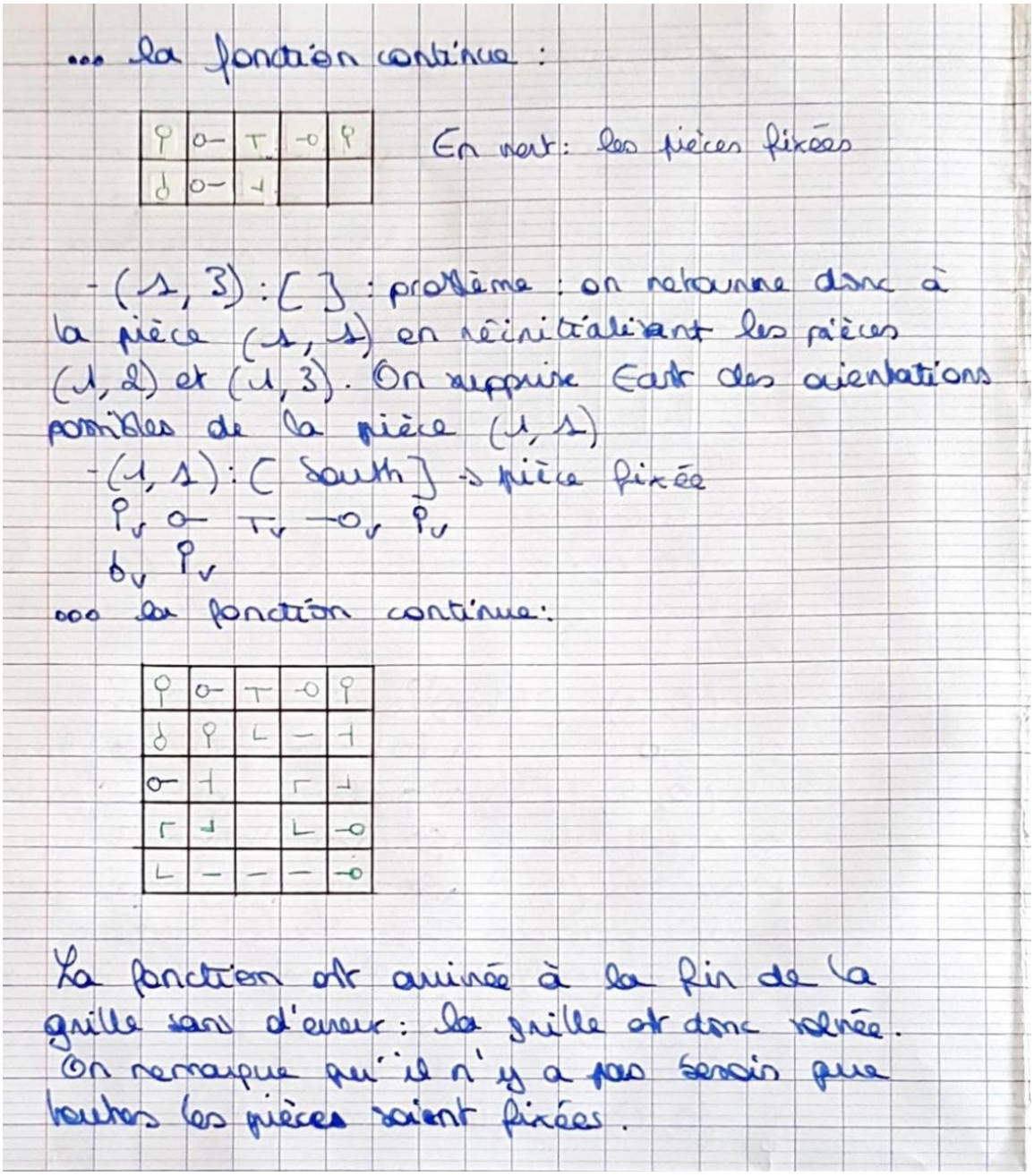
⚲ o—

- (0, 2) : [South] -> pièce fixée

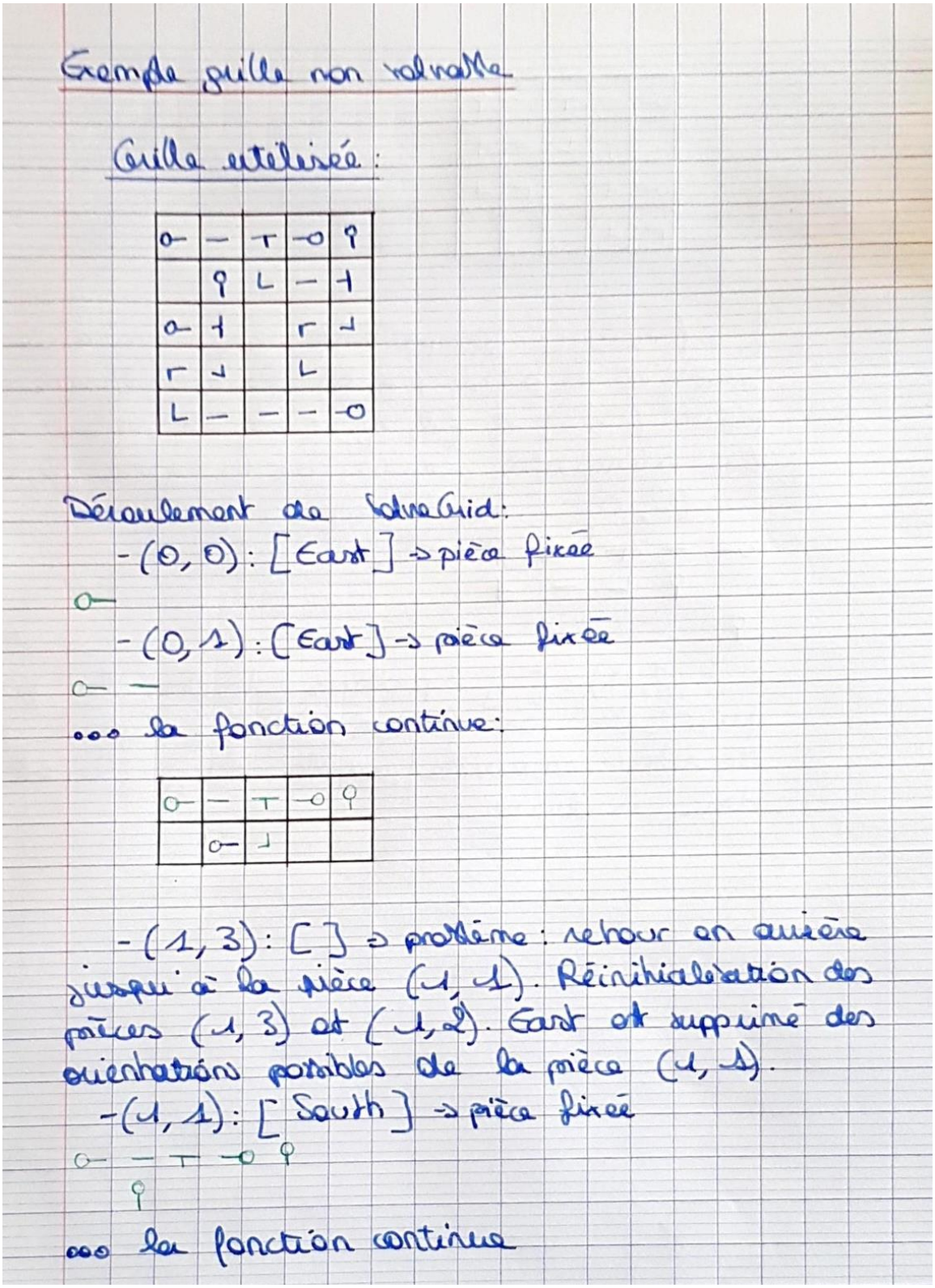
⚲ o— T

- (0, 3) : [West] -> pièce fixée

⚲ o— T —o


... la fonction continue :
En vert : les pièces fixées
- (1, 3) : [ ] : problème : on retourne donc à la pièce (1, 1) en réinitialisant les pièces (1, 2) et (1, 3). On supprime East des orientations possibles de la pièce (1, 1)
- (1, 1) : [South] => pièce fixée
... la fonction continue :
La fonction est arrivée à la fin de la grille sans d'erreur : la grille est donc valide.
On remarque qu'il n'y a pas besoin que toutes les pièces soient fixées.

Voici un autre exemple d'implémentation sur une grille non solvable :



Exemple grille non solvable

Grille utilisée :

○	─	┬	─○	○
	○	└	─	┤
○	┤		┌	┘
┌	┘		└	
└	─	─	─	─○

Déroulement de solveGrid :

- (0, 0) : [East] → pièce fixée

○─

- (0, 1) : [East] → pièce fixée

○──

... la fonction continue :

○	─	┬	─○	○
	○	┘		

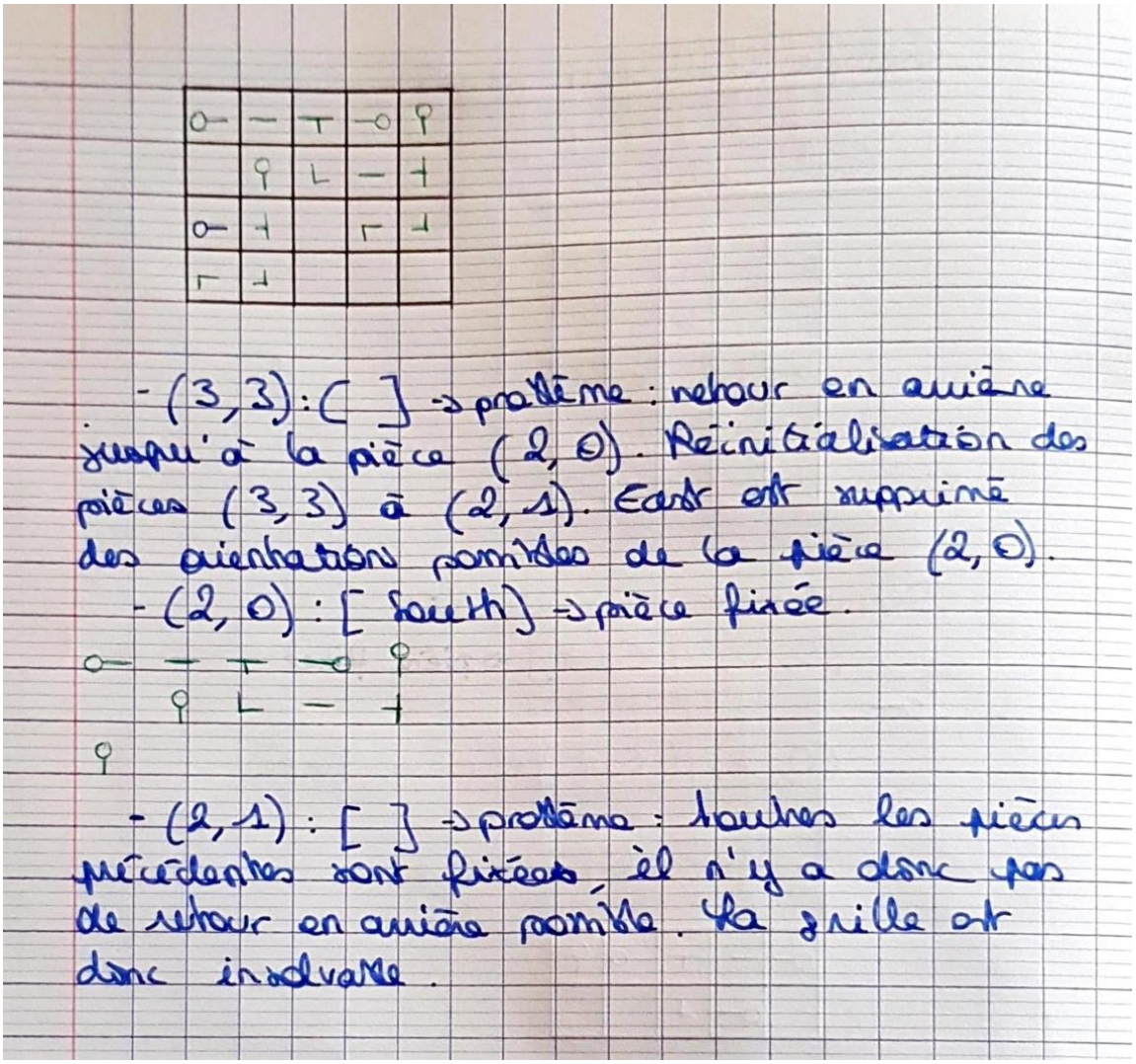
- (1, 3) : [ ] → problème : retour en arrière jusqu'à la pièce (1, 1). Réinitialisation des pièces (1, 3) et (1, 2). East est supprimé des orientations possibles de la pièce (1, 1).

- (1, 1) : [South] → pièce fixée

○── ┬ ─○ ○

○

... la fonction continue



Checker

Le checker permet de vérifier si une grille est bien solvée. Dans cette classe se trouve donc la fonction isSolution(). Elle va se servir de la fonction isValidOrientation() de la classe Grid. En effet, si une grille est solvée, alors chacune de ses pièces aura une orientation correcte, c'est-à-dire que tous ses connecteurs seront bien placés et orientés. Le test JUnit testIsSolution() de la classe CheckerTest permet de vérifier que la fonction fonctionne correctement.

Interface graphique

Pour faire fonctionner l'interface graphique, plusieurs fonctions devaient être implémentées : getImageIcon() et initialize() de la classe GUI qui comporte en plus d'autres fonctions. Elle possède comme attributs un JFrame, une image, une hauteur et une largeur. La fonction startGUI() permet,

grâce à un thread, de lancer l'interface graphique. Pour cela, elle appelle le constructeur de la classe GUI. Ce dernier va initialiser le frame et appeler la fonction initialize() qui va paramétrer le frame. Ce dernier doit avoir comme dimensions les mêmes dimensions que la grille mais avec une hauteur et une largeur multipliées par 70 car chaque image de pièce a une dimension de 70*70. La grille va ensuite être parcourue dans le sens de lecture. La fonction getImageIcon() permettra de récupérer l'image correspondant à une pièce. Chaque image de pièce est ensuite ajoutée dans le frame grâce à la fonction add() qui fait appelle à la fonction paintComponent(). Cette dernière permettra d'afficher l'image aux bonnes coordonnées dans le frame.

Quand nous avons essayé d'afficher l'interface graphique, nous avons rencontré des problèmes à l'exécution. En faisant des recherches, nous nous sommes rendu compte que ce problème provenait souvent d'un problème dans la bibliothèque qui implémente notre fonction qui récupère les images. Voici un lien pour plus d'explications : https://bugs.openjdk.java.net/browse/JDK-6945174. Faute de temps, nous n'avons malheureusement pas pu nous pencher plus longtemps sur la question et modifier notre implémentation.

Ce problème de temps nous a aussi impacté pour d'autres aspects de notre implémentation d'Infinity Loop. En effet, nous aurions aimé avoir le temps de réfléchir à une manière de gérer le nombre de nbcc dans la génération d'une grille. Nous avions commencé à réfléchir à différents moyens de l'implémenter comme, par exemple, générer une grille grâce à la fonction generateGrillFilled() (qui retourne une grille sous sa forme résolue), compter le nombre de nbcc puis regénérer une grille jusqu'à ce que le nombre de nbcc corresponde à ce que l'utilisateur avait demandé. Notre deuxième idée a été de rajouter ou retirer quelques connecteurs (donc de changer le type ou l'orientation d'une pièce) à une grille déjà générée pour avoir un bon nombre de nbcc.