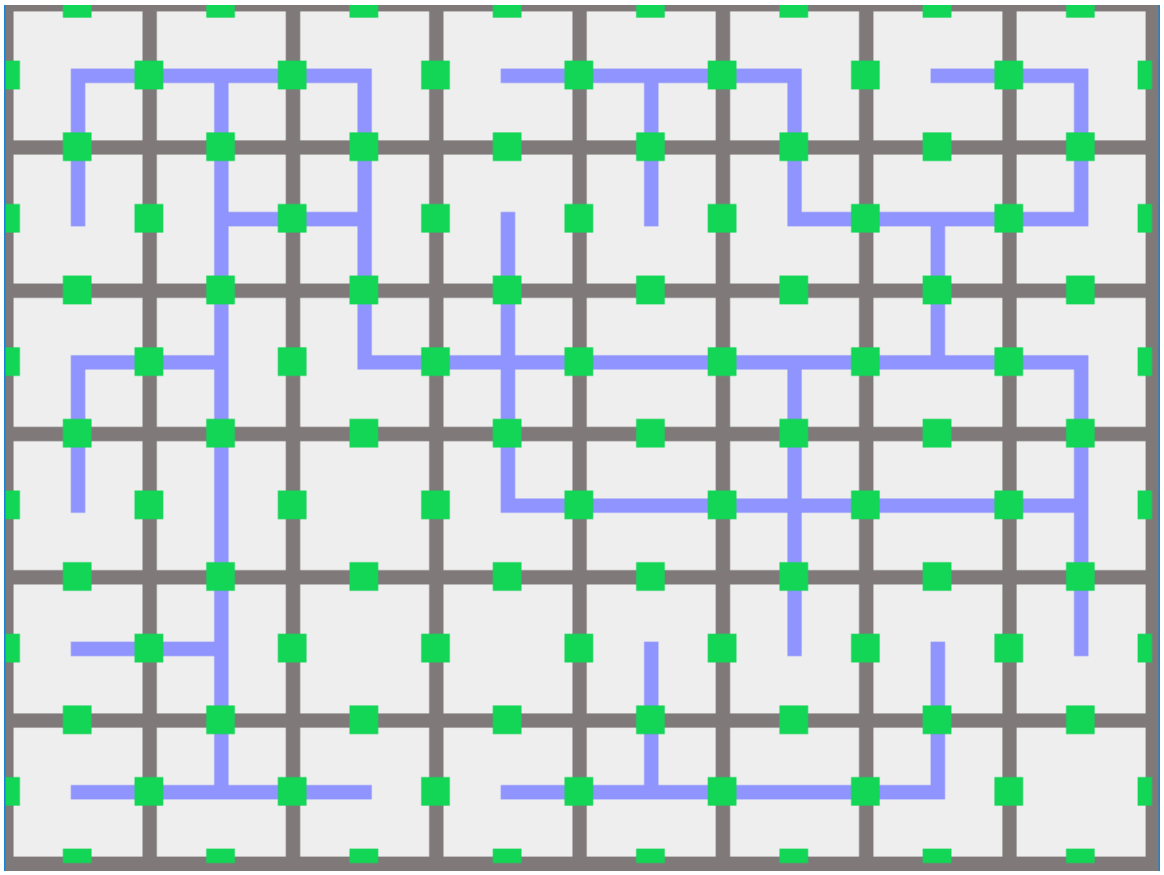
RAPPORT DE PROJET JAVA PHINELOOPS_CHAVA

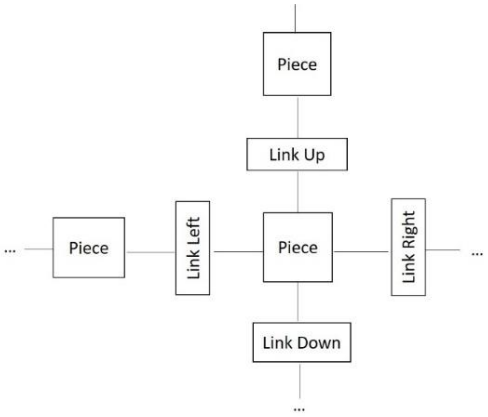


AZZAZ MYRIAM
DAUVERGNE SIDNEY
PACHOUD ALEXANDRE

Table des matières

I. Représentation d'un jeu

Premièrement, nous allons expliquer quels ont été nos choix en ce qui concerne la représentation d'un jeu. En effet, nous avons décidé de représenter un plateau de jeu grâce à des objets de types Piece et des objets de type Link.



Une Piece peut être de type Empty (n°0), End (n°1), Line (n°2), L (n°3), Plus (n°4), ou T (n°5), en fonction du nombre de connexions qu'elle a. Le tableau ci-dessous extrait du sujet montre les différentes possibilités d'orientation.

pièce		numéro-pièce	orientation-pièce
emplacement vide		0	0
pièce 1 connection nord	╹	1	0
pièce 1 connection est	╺	1	1
pièce 1 connection sud	╻	1	2
pièce 1 connection ouest	╸	1	3
pièce 2 connections nord-sud	┃	2	0
pièce 2 connections est-ouest	━	2	1
pièce 3 connections (T) nord-est-ouest	┻	3	0
pièce 3 connections (T) nord-sud-est	┣	3	1
pièce 3 connections (T) est-sud-ouest	┳	3	2
pièce 3 connections (T) nord-sud-ouest	┫	3	3
pièce 4 connections nord-sud-est-ouest	╋	4	0
pièce L nord-est	┗	5	0
pièce L est-sud	┏	5	1
pièce L sud-ouest	┓	5	2
pièce L ouest-nord	┛	5	3

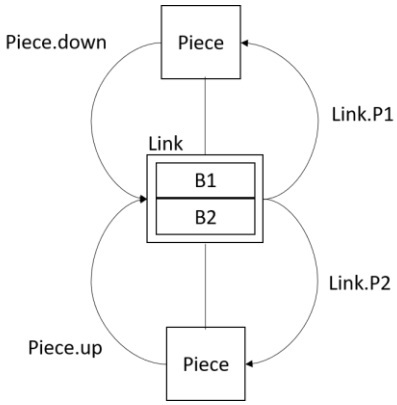
Une Piece a plusieurs attributs :

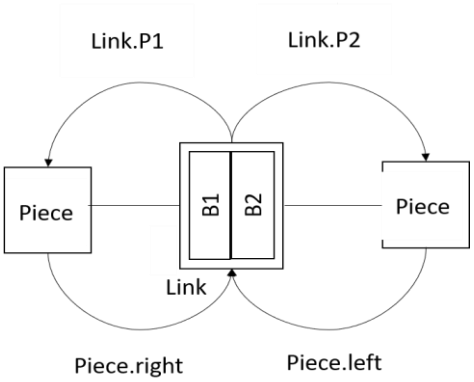
- int id (correspond au numéro-pièce dans le tableau).
- int column, line : correspondent aux coordonnées sur le plateau de jeu.
- ArrayList<Integer> possibilities : correspond aux différentes orientations possibles pour une Piece.
- Link up, down, right, left : correspondent au Link qui entourent la piece. Ils sont mis à jour lors d'une rotation ou d'un changement d'orientation quelconque. Ils permettent de savoir où sont les connexions en fonction du type de la pièce.

Un Link est un objet représentant la connexion entre deux pièces. Un Link a quatre attributs :

- Piece p1 : correspond à la pièce de gauche ou du dessus, en fonction de l'orientation du Link
- Piece p2 : correspond à la pièce de droite ou du dessous, en fonction de l'orientation du Link
- Boolean b1 : vrai si p1 a une connexion qui touche p2
- Boolean b2 : vrai si p2 a une connexion qui touche p1

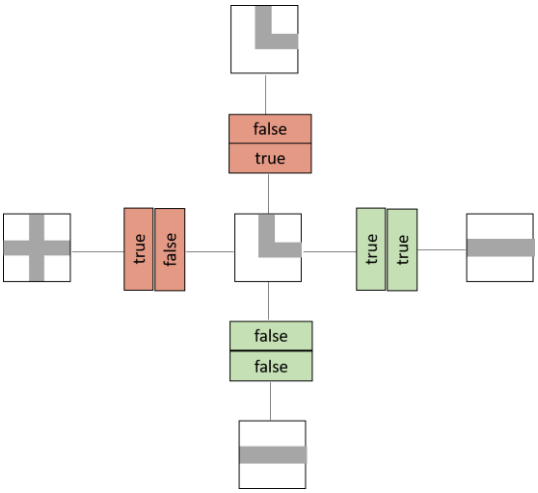
Voici comment on pourrait représenter les relations entre les objets de type Piece et ceux de type Link.





Un Link est valide si les valeurs de b1 et de b2 sont égales. En effet, si b1 est vrai et b2 est vrai, alors la piece p1 et la piece p2 se touchent. Si b1 est faux et b2 est faux alors les deux pieces ne se touchent pas. Au contraire, si b1 est vrai et b2 est faux (ou inversement), cela signifie qu'une piece touche l'autre, mais que l'autre non. Or, on cherche à connecter toutes les pièces entre elles. Ainsi, le jeu est valide lorsque tous les Links sont valides.

Voici un exemple concret de relation entre les objets Piece et les objets Link :

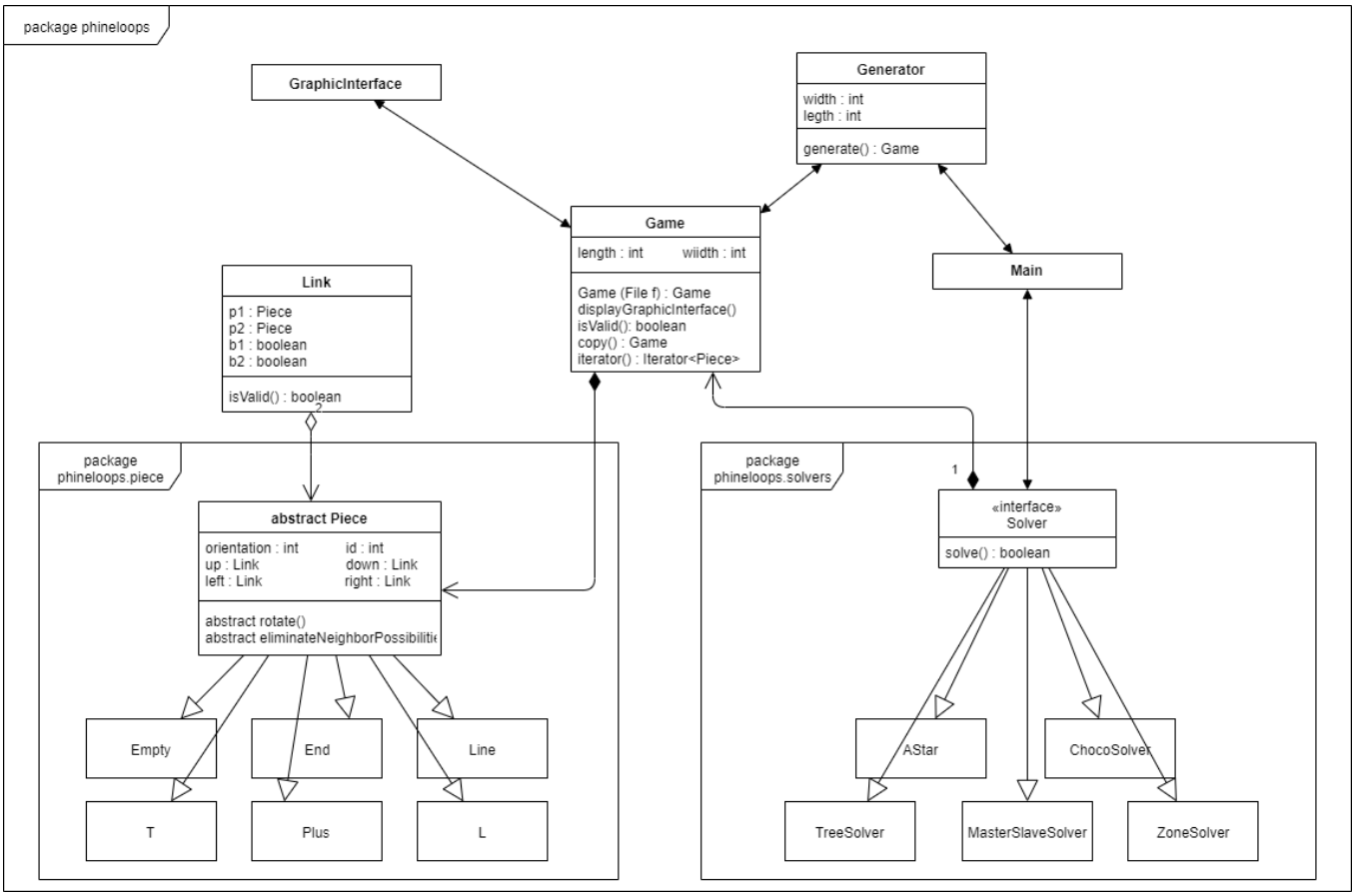


Les Link en vert sont des Link valides. En effet, b1 et b2 sont tous les deux vrais ou tous les deux faux. Au contraire, les Link en rouge ne sont pas valides.

Nous avons jugé plus simple de concevoir notre jeu ainsi, que ce soit à la vérification ou à la génération, dont le processus sera défini un peu plus tard. En effet, lorsque nous avons réfléchi à la manière de représenter un jeu, nous avons pensé qu'il serait judicieux de trouver un moyen rapide de vérifier si le game était valide, quitte à dépenser plus de mémoire, car nous en aurions souvent besoin, notamment pour implémenter le solveur.

II Architecture

Dans cette section nous allons présenter brièvement l'architecture générale à l'aide d'un diagramme de classes simplifié, puis nous verrons en détail les choix et avantages de cette architecture.



Pour plus d'informations, il existe un dossier JavaDoc dans notre projet pour consulter la JavaDoc.

Nous avons décidé de créer des classes pour chaque type de pièces qui étendent une classe Piece de type carré général. Nous avons pris cette décision pour avoir plus de clarté sur le code et pouvoir rajouter des types de pièce à volonté.

Nous avons fait en sorte que Game soit un itérable de Piece. Ainsi, lorsque dans le code on a besoin de parcourir toutes les pièces, inutile de faire deux boucles pour toutes les lignes, pour toutes les pièces de la ligne. Ici : « `for(Piece p : game){}` » va parcourir toutes les pièces dans le sens de lecture.

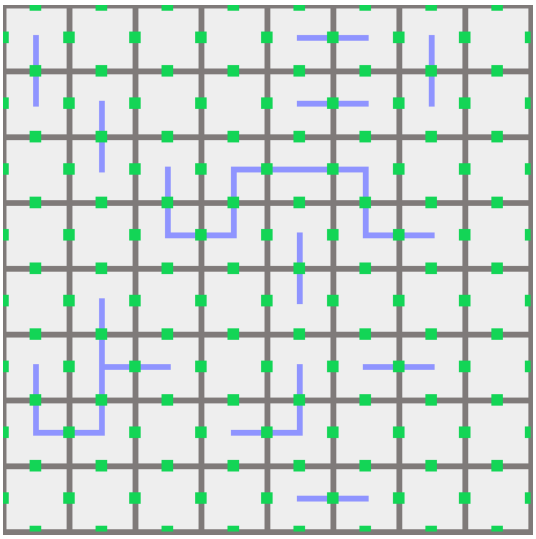
Par ailleurs, nous avons créé une interface `Solve` que chaque solveur implémente. Ainsi chaque solveur possède une fonction `solve()` qui retourne un booléen : vrai si le jeu être résolu, faux s'il est irrésolvable.

III. Générateur de niveaux

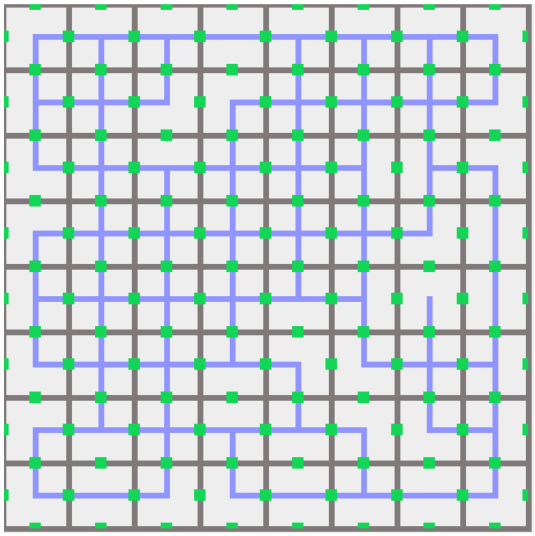
Une classe est dédiée au générateur de niveau. Il s'agit de la classe Generator. Nous procédons à la génération d'un niveau en passant par quatre étapes :

1. On va d'abord créer un jeu constitué de pièces vides.
2. Ensuite, on va parcourir chacun des links de ce jeu. S'il est en bordure du jeu, il sera défini comme faux, sinon on lui attribue aléatoirement la valeur de vrai ou faux selon une probabilité qui peut être donnée. Plus la probabilité est proche de 0, plus le jeu à de chance de former une unique composante connexe, et inversement, plus la probabilité est proche de 1, plus le jeu sera constitué de composantes connexes. Si l'on n'attribue pas de probabilité lors de la construction, la probabilité par défaut est 0,5.
3. Une fois que tous les links sont fixés, on va parcourir chacune des pièces du jeu, et compter combien de links sont true. C'est à partir du chiffre obtenu que l'on reconnaît la pièce dont il sera question et, ainsi, il est possible d'instancier la pièce dans le jeu. Prenons l'exemple d'une case. Si aucun Link n'est vrai, alors la pièce sera une pièce vide. S'il y a trois links vrais, alors c'est un T, etc. Puisque on génère un jeu, qui est mélangé, inutile de savoir dans quelle orientation construire la pièce.
4. L'étape finale est de mélanger aléatoirement toutes les pièces afin d'avoir un jeu à résoudre.

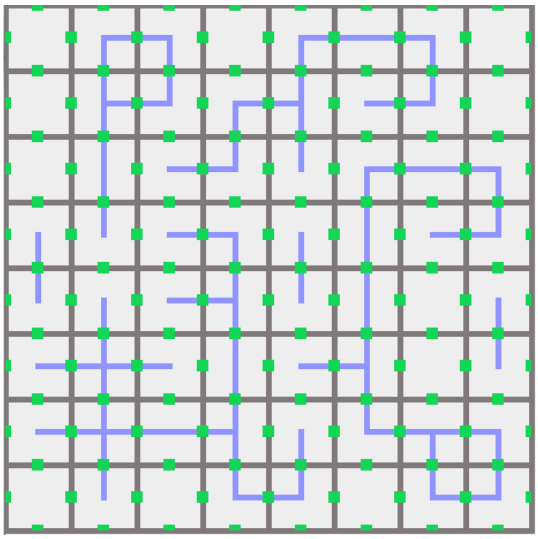
Voici quelques exemples de grilles générées. Les arguments du constructeur de la classe Generator sont le nombre de lignes et le nombre de colonnes que doit contenir la grille. L'argument de la méthode generate est la probabilité qu'un lien soit true ou false, comme expliqué précédemment.



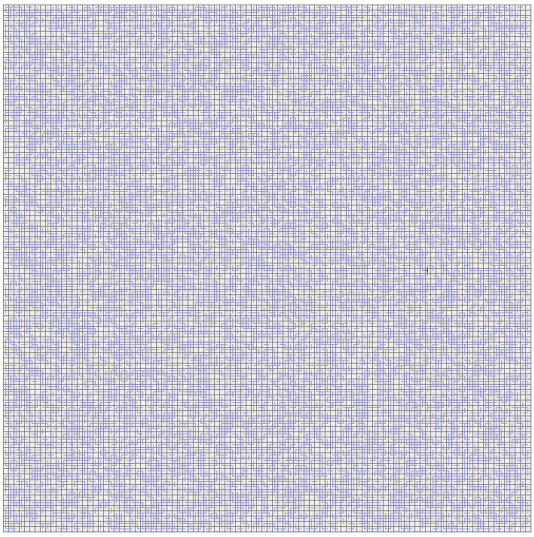
Generator(8,8).generate(0.8)



Generator(8,8).generate(0.2)



Generator(8,8).generate(0.5)



Generator(100,100).generate(0.5)

Cependant, on notera qu'il n'est pas possible de choisir le nombre de composantes connexes lors de la génération En effet, nous n'avons pas conçu notre jeu sous la forme d'un graphe. L'architecture que nous avons construite est donc peu propice pour cela.

IV. Solveurs

Dans cette section nous allons décrire l'évolution des méthodes de résolution. Nous présenterons les motivations à la création de chaque méthode, puis les avantages et inconvénients qu'elles présentent.

A/ AStarSolver

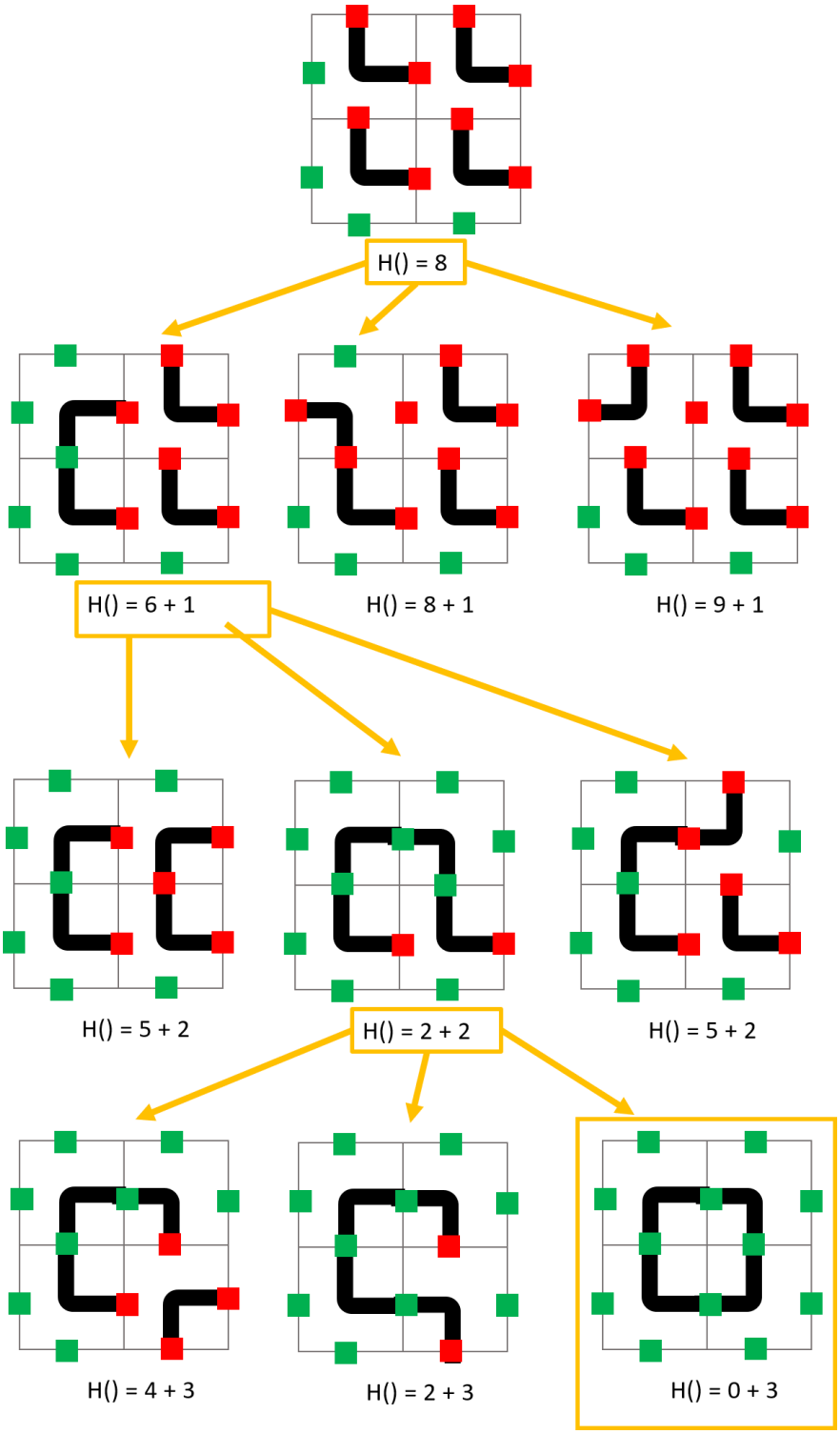
La première approche a été de nous inspirer de l'algorithme A*. L'idée est qu'on s'approche de la solution pas à pas, en faisant tourner les pièces.

Avec notre choix d'architecture, nous savons que si tous les liens entre les pièces sont valides, alors le jeu est résolu. De ce fait, nous avons défini notre heuristique comme le nombre de liens entre les pièces qui ne sont pas valides. Plus il y a de liens valides, plus nous nous approchons de la solution.

L'avantage de cette solution est qu'elle a été rapide à implémenter. De plus, elle fonctionne très bien sur des petites instances.

L'inconvénient de ce solveur est qu'il est trop lent. En effet, il teste beaucoup trop de possibilités et, d'un état à un autre, toutes les pièces peuvent être tournées.

Voici un exemple d'exécution avec ce solveur :



B/ MasterSlaveSolver

Après avoir compris que l'algorithme A* n'était pas le meilleur algorithme d'intelligence artificielle à appliquer, nous avons décidé d'appliquer un autre algorithme qui consiste à éliminer les orientations impossibles qu'une pièce peut prendre.

Cette approche est l'approche qui sera utilisée dans tous les autres solveurs. Dans cette section, nous expliquerons l'algorithme appliqué, et où interviennent les variations d'un solveur à un autre.

- **Initialisation** : Donner à toutes les pièces un ensemble d'orientations possibles.
  Si la pièce est sur un bord, il faut lui attribuer les orientations possibles, en conséquence. On initialise 2 listes (des listes chainées car elles sont vouées à être modifier régulièrement) : la liste des pièces non-fixées, remplie initialement de toutes les pièces, et une deuxième liste, celle des pièces fixées, initialement vide.
  Cette étape est commune à quelques solveurs, et est donc réutilisée.

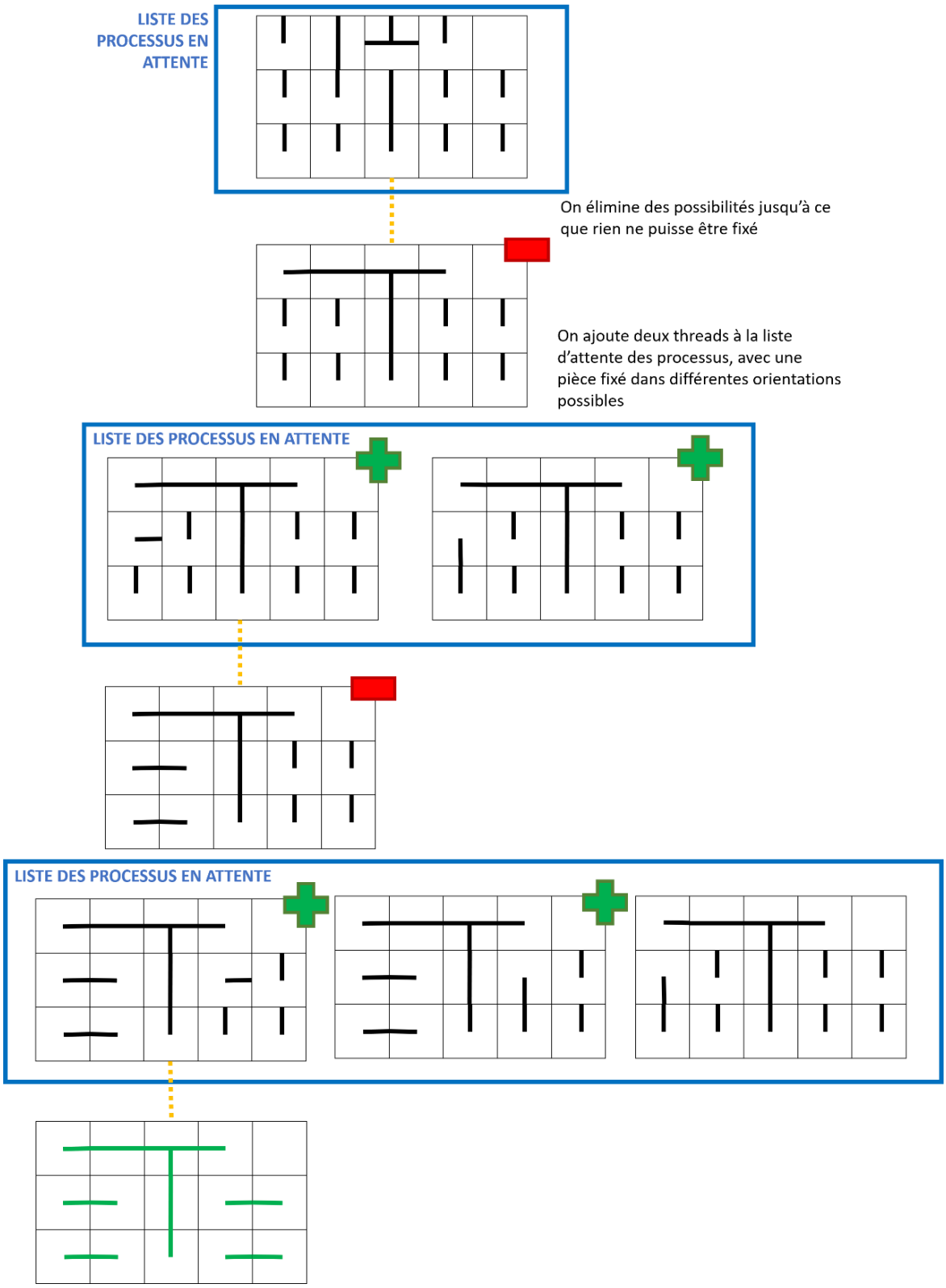
- Tant que le jeu n'est pas **valide** :
  - On parcourt toutes les pièces qui ne sont pas fixées
    - Si une pièce n'a **aucune possibilité**, arrêter l'algorithme : le jeu est insolvable.
    - Si une pièce n'a plus **qu'une possibilité** possible on la lui donne et on la rajoute à la liste des pièces fixées.
  - Pour toutes les pièces qui ont été fixées, on **élimine les possibilités** que les voisins ne peuvent plus prendre.
  - SI **aucune pièce vient d'être fixé**. Alors on doit prendre une décision d'orientation. C'est ce que le solveur fait dans ce cas, qui varie d'un solveur à un autre.

Pour le solveur appelé MasterSlaveSover, l'idée est, comme son nom l'indique, de résoudre de façon Multi-Threadée, avec un processus « maitre » qui donne des calculs à faire aux esclaves. Lorsqu'un Processus esclave a terminé son exécution et qu'il a trouvé une solution, il la transmet au processus maître. S'il a trouvé une impossibilité, il ne se passe rien. Si le processus ne peut rien fixer de plus, il transmet au maitre les possibilités à tester, dans des processus.

Pour ce qui est du maitre, lorsqu'un processus a fini, il lance un processus qui était en attente. Ainsi, on peut contrôler le nombre de processus en cours d'exécution, ce qui est le principal avantage de ce solveur.

L'inconvénient est que l'ordonnancement à un coût. Ce coût ralenti la recherche de solution.

Voici un exemple d'exécution avec ce solveur :



C/ ChocoSolver

L'idée de ce solveur était d'utiliser un solveur déjà construit. En effet, ChocoSolver est une librairie Java pour la programmation par contraintes.

Dans ce solveur, nous créons donc un modèle puis nous définissions nos variables et nos contraintes, puis ChocoSolver essaye de trouver une solution. Nos variables correspondent aux différentes orientations possibles pour une pièce. Ainsi, nous initialisons nos variables avec les différentes valeurs que peuvent prendre les orientations pour chaque pièce (exemple : orientation pour Empty ne peut prendre que la valeur 0, orientation pour Line ne peut prendre que les valeurs 0 ou 1, …). Puis, nous définissons nos contraintes. Pour ce faire, nous avons regardé les pièces une par une et, en fonction de la pièce qu'il y au-dessus (respectivement en dessous, à droite et à gauche), nous remarquons quelles orientations sont impossibles ou quelle orientation est obligatoire pour pouvoir résoudre le jeu.

L'avantage de ce solveur est qu'il est rapide d'expliquer les contraintes pour le programmeur.

En revanche, ce solveur est lent car il ne nous permet pas de maîtriser la façon dont les contraintes sont satisfaites et car il y a beaucoup de contraintes. Si nous devions réutiliser ce solveur, il conviendrait de diminuer le nombre de contraintes.

D/ TreeSolver

Ce solveur reprend l'idée du MasterSlaveSolveur, en essayant de l'optimiser. Ainsi, l'initialisation et les listes de pièces fixées et non fixées ont la même utilisation. Le but est ici de réduire l'ordonnancement.

On procède à la résolution comme précédemment. Lorsqu'il n'est plus possible de fixer une pièce mais qu'il est encore possible de résoudre, il faut prendre une décision, et c'est à partir de ce moment-là que le nom de « TreeSolver » prend tout son sens.

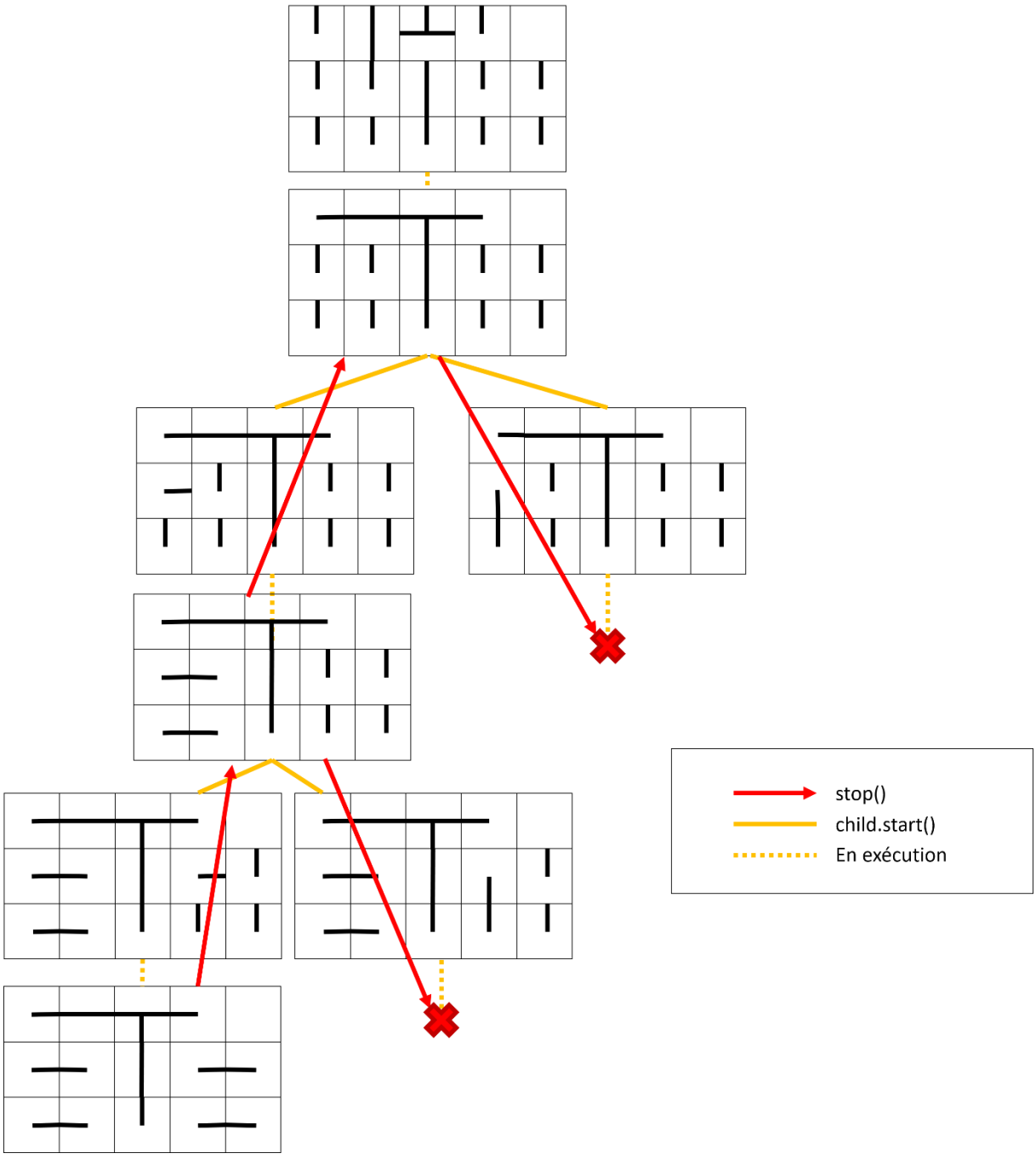
En effet, lorsqu'il n'est plus possible de fixer une pièce, il faut en choisir une et la fixer. Pour le moment, la pièce choisie est la première dans la liste des pièces non fixées. Si on voulait réutiliser ce solveur, il serait pertinent de modifier le choix de la pièce, de façon à réduire la complexité.

Après avoir choisi une pièce, on lance autant de processus qu'il y a d'orientations encore possibles pour cette pièce et on les teste, de façon à voir quelle possibilité permet de résoudre le jeu. Les Threads essayent alors de résoudre chacun avec ce qui a été résolu auparavant et à partir de la nouvelle pièce fixée. Si un Thread arrive à résoudre le jeu alors il renvoie la solution au père et arrête l'exécution des autres Threads. Au contraire, si un thread n'arrive pas à résoudre, il s'arrête mais les autres peuvent continuer.

Cette nouvelle approche permet un gain de temps considérable.

Cependant, il existe un inconvénient. De fait, cette méthode est assez difficile à implémenter pour le programmeur qui doit porter une attention particulière au problème ThreadSafe.

Voici un exemple d'exécution avec ce solveur :



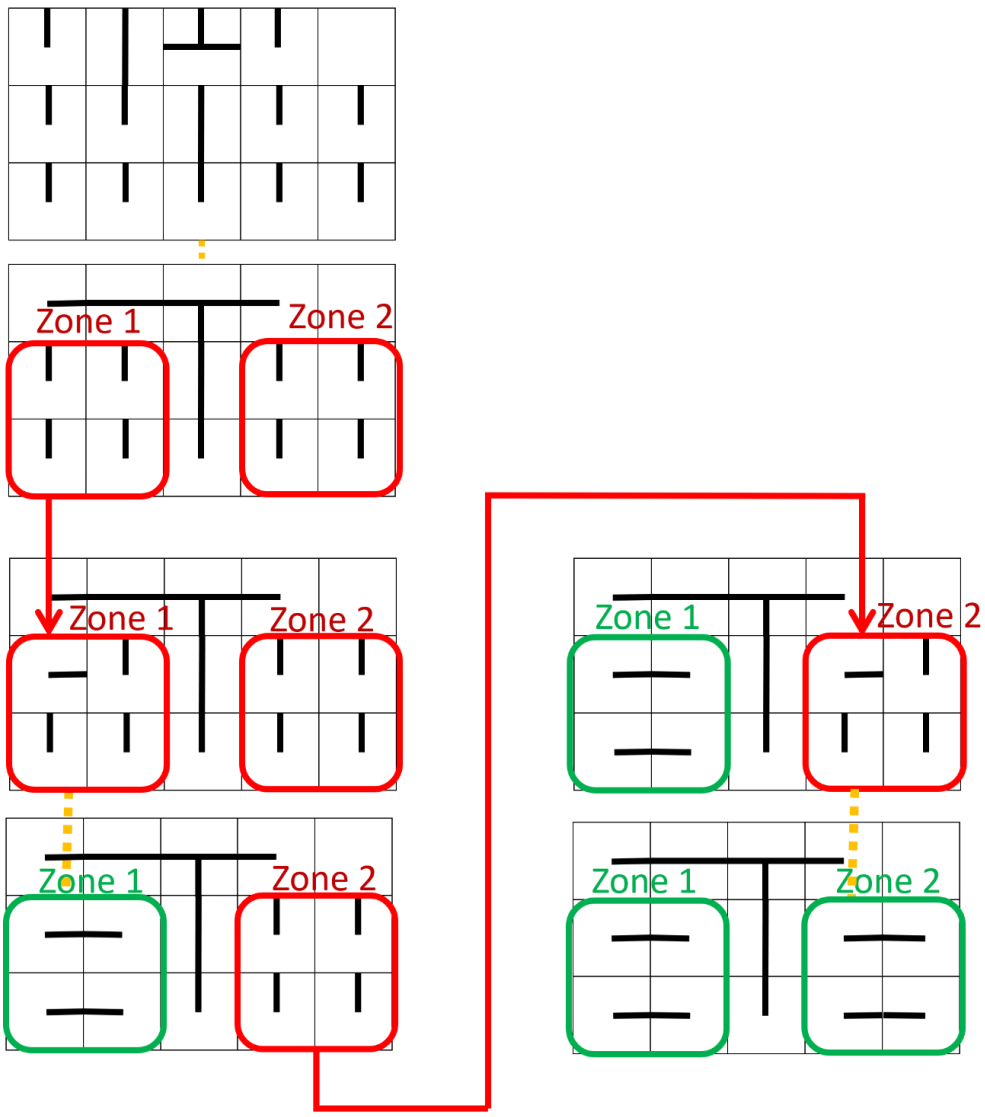
E/ ZoneSolver

Lorsque nous avons utilisé l'interface graphique pour observer la résolution de grosses instances, nous avons remarqué qu'il y avait des zones qui se dessinaient dans le jeu, et qui pouvaient être résolues indépendamment les unes des autres. L'idée qui nous est venue fut de résoudre ces zones indépendamment des unes des autres, afin de résoudre le jeu de manières récursive.

Un des avantages de ce solveur est de pouvoir détecter plus rapidement si le jeu est insoluble. En effet, il n'y a pas besoin de résoudre toutes les zones : si une zone est insoluble alors le jeu l'est. Un autre avantage est que l'algorithme utilise une seule instance du jeu en mémoire, ce qui réduit considérablement le besoin en espace de stockage.

Pour faire évoluer ce solveur rapide, il faudrait le rendre multithread, en résolvant les zones en parallèle.

Voici un exemple d'exécution avec ce solveur :

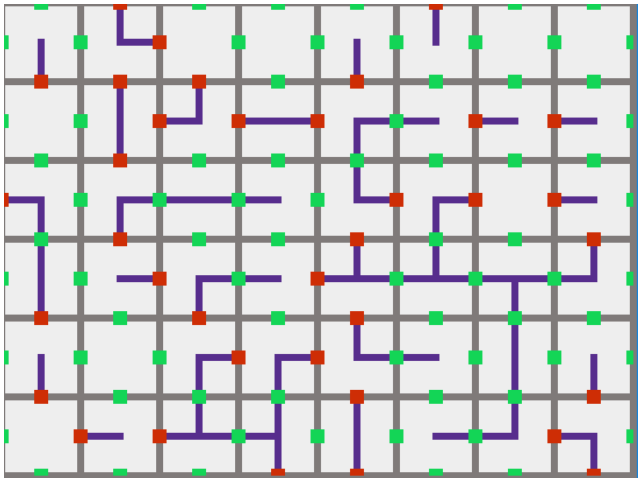


V. Interface graphique

L'interface graphique a été codée au début du projet, peu de temps après avoir établi l'architecture. Il nous a semblé important de l'implémenter au plus tôt afin d'observer la résolution en temps réel, repérer les bugs et améliorer les solveurs.

L'avantage majeur de notre interface graphique réside dans sa simplicité d'utilisation. Une simple ligne de commande suffit à afficher un jeu (`g.displayAdvanced()` pour un affichage qui montre la structure du jeu `ou g.displaySimple()` pour un affichage plus lisible).

Nous avons également établi un code couleurs sur les links et les pièces. Un Link est rouge lorsqu'il est faux, vert lorsqu'il est vrai. Une pièce est bleue par défaut, et rose pâle lorsqu'elle est fixée. Ce code couleur nous permet de repérer où est le problème, s'il y en a un, ou de vérifier que tout soit valide, lorsque tous les Links sont verts. Voici un exemple d'un jeu généré non résolu :



Generator(6,8).generate()

L'interface graphique est également responsive. Le quadrillage du jeu va s'adapter à la taille de la fenêtre mais également à la taille du jeu. Toutefois, nous ne sommes pas parvenus à la rendre cliquable et dynamique, afin qu'un joueur puisse tenter de résoudre le jeu par lui-même.

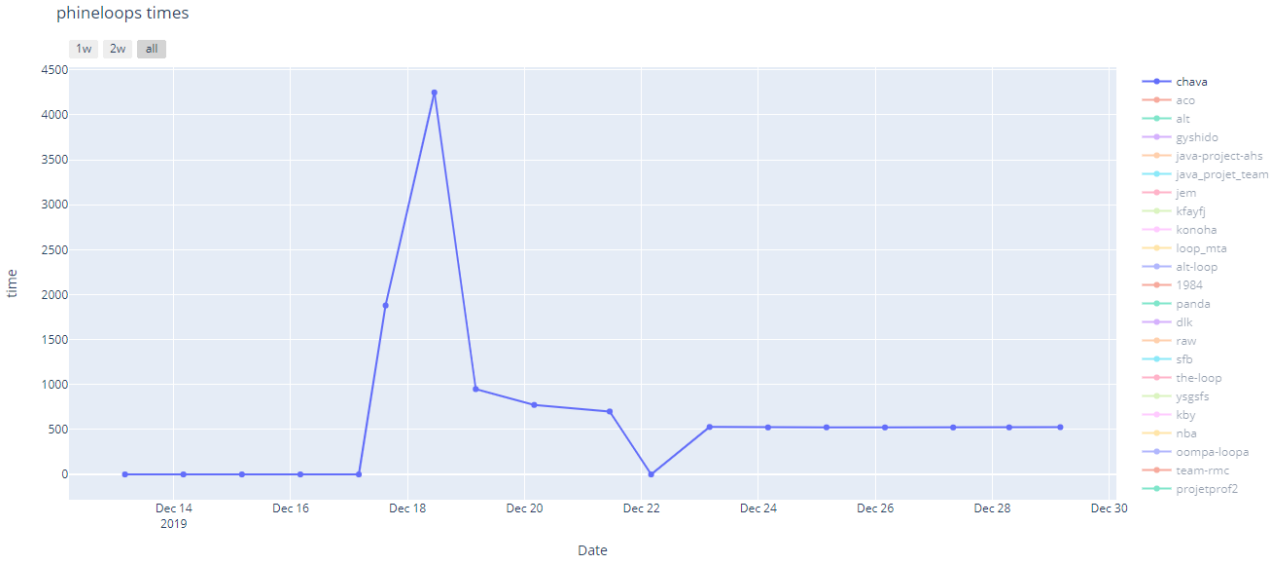
VI. Organisation du travail

En ce qui concerne la répartition du travail, nous avons commencé par une première réunion de préparation. En effet, il était important pour nous de tout organiser ensemble et de se mettre d'accord sur l'architecture choisie. Ainsi, chacun a pu donner son avis et émettre ses idées pour que le projet soit compris par tous et qu'il soit le fruit de nos efforts communs.

Dans le même temps, nous avons créé les pièces ensemble, autour d'un même ordinateur, de sorte à avoir une base commune comprise et apprivoisée par tous les membres du groupe. Puis, nous avons réparti le travail, de sorte à pouvoir avancer parallèlement et à pouvoir commencer le plus tôt possible.

Ainsi, nous pourrions fournir un travail complet et observer notre évolution dans un lapse de temps plus grand.

Cela a notamment pris son sens lorsque les tests du professeur ont commencé. Nous avons suivi avec attention tous les tests qui ont été effectués. En effet, nos tests personnels permettaient notamment de pouvoir comparer nos différents solveurs mais surtout de vérifier si les jeux étaient correctement résolus ou non et dans un intervalle de temps acceptable ou non. Avec les tests du professeur, nous avons pu nous comparer au cours du temps, de façon plus précise, mais également par rapport aux autres groupes.



Ce graphique, extrait des résultats du professeur nous montre l'évolution de notre projet, en représentant le temps mis pour résoudre les différentes instances en fonction du temps. Il est intéressant d'étudier ce graphique afin de le comprendre et, ainsi, de comprendre nos choix.

En effet, le pic le plus élevé représente le moment où les tests ont été effectués avec notre solveur ChocoSolver. On peut remarquer que le temps a considérablement augmenté par rapport au solveur précédent. Cela nous a donc amené à élaborer un nouveau solveur, TreeSolver, bien plus performant que ChocoSolver, mais également plus performant que le MasterSlaveSolver, que nous pouvons observer vers le 18 décembre. Enfin, on peut remarquer que notre dernier solver désormais en place, ZoneSolver, est le plus performant de tous puisque c'est celui qui résout en moins de temps les différentes instances données. Par ailleurs, on omettra les pics à 0 dans la mesure où, ces jours-là, nous n'étions pas présents dans la liste des résultats, ou avec une erreur corrigée par la suite par le professeur : « no jar file ».

Initialement, Myriam avait la charge d'implémenter le générateur de niveau, Sidney le solveur et Alexandre l'interface graphique et la gestion des fichiers (lecture et écriture). Nous avions réparti ces tâches en fonction des compétences de chacun.

Cependant, nous avons rapidement remarqué un déséquilibre, dans la mesure où le solveur représentait un travail particulièrement conséquent et qu'il formait le projet d'intelligence artificielle, comme nous l'avons appris par la suite. Ainsi, nous avons décider de travailler en collaboration sur les différentes parties.

Il est important de noter que nous avons multiplier les réunions. En effet, nous nous sommes souvent vus ou appelés pour parler du projet. Il était important pour nous d'avancer ensemble, d'expliquer au fur et à mesure ce que nous avions implémenter de notre côté pour être sûr que tout le monde comprenne et pour permettre à chaque individu de pouvoir ajouter ses idées dans les différentes parties.

Par ailleurs, lorsque les grèves ont commencé, nous avions finalement plus de temps libre et nous avons passé beaucoup de temps ensemble au CRIO de l'université ou sur l'application Teams, afin d'améliorer notre projet.