Aix*Marseille université
Socialement engagée
Année 2021-22

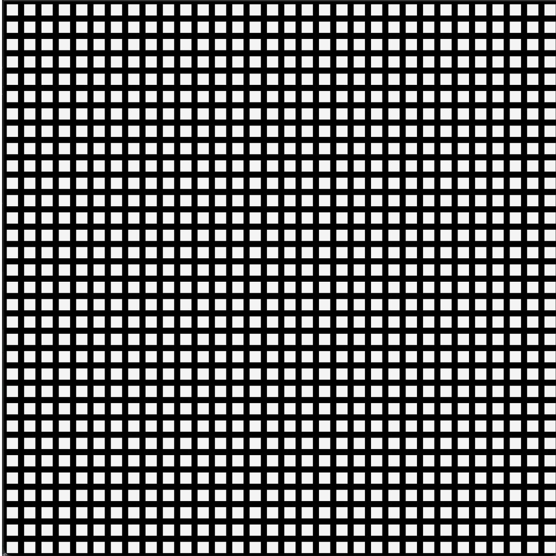
Domaine Sciences et Technologies
Licence informatique

Programmation 2 : TP 11 (licence informatique)
Code UE : SIN3U02L

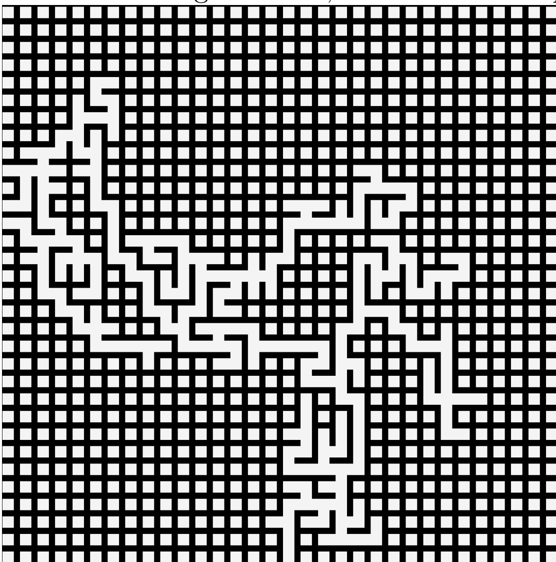
Génération de labyrinthes

1 Labyrinthe

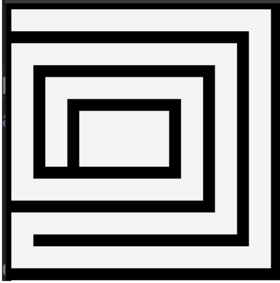
Nous allons nous intéresser à la génération de labyrinthes à l'aide d'algorithmes simples randomisés. Dans ce projet, il s'agit de générer des labyrinthes à partir d'une grille rectangulaire. Chaque cellule case de la grille est initialement séparée de chacune ses voisines par un mur :



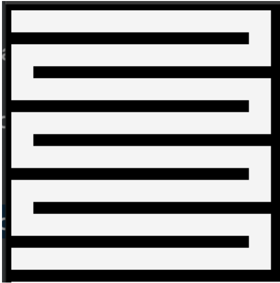
Au cours de la génération, des murs sont supprimés de façon à obtenir des chemins entre cases de la grille :



Un labyrinthe est dit *parfait* s'il est complètement connecté, c'est-à-dire si il existe un chemin entre n'importe quelle paire de cases. Dans l' exemple ci-dessous, il s'agit d'un labyrinthe imparfait : la zone centrale est isolée du reste des cases.



Ce second exemple présente par contre un labyrinthe parfait.



Nous allons dans la suite implémenter des générateurs de labyrinthes parfaits.

1.1 Principe

La génération de labyrinthes se déroule par *étapes* (*step*). Initialement, la grille est complètement murée : chaque case est séparée de ses voisine par un mur. À chaque étape, une ou plusieurs paires de cases voisines sont connectées en supprimant le mur qui les sépare. Des étapes sont effectuées jusqu'à ce que toutes les cases soient connectées. Ainsi, les algorithmes de génération doivent répondre à deux objectifs apparemment contradictoires :

- Connections de toutes les cases. Ceci peut se faire très simplement en supprimant tous les murs ;
- Difficulté du labyrinthe. Le labyrinthe obtenu ne doit pas être trop simple à résoudre. On supposera que le point de départ est la case dans le coin supérieur gauche, et le point de sortie la case dans le coin inférieur droit.

1.2 Interfaces et classes fournies

Nous vous fournissons des classes qui permettent de manipuler la grille, ses cases ainsi que les murs. Certaines classes vous sont familières puisque issues du TP sur les tuiles de Wang.

1.2.1 Grille : Interface `Grid` et implémentation `ArrayGrid`

Une grille est représentée par une instance d'une classe implémentant l'interface `Grid`. La classe `ArrayGrid`, qui est fournie, implémente le type `Grid` et représente une grille rectangulaire. On accède aux dimensions de la grille via les méthodes `getNumberOfRows()` et `getNumberOfColumns()`.

1.2.2 Cases de la grille : Interface `Square` et implémentation `MapSquare`

Les cases de la grille sont représentées par des instances de classes implémentant l'interface `Square`. Une implémentation `MapSquare` est fournie. Un case est repérée par son index de colonne `columnIndex` et son index de ligne `rowIndex`.

- La méthode `getSquare(int columnIndex, int rowIndex)` du type `Grid` retourne un objet de type `Square` qui représente la case situées à la colonne `columnIndex` et à la ligne `rowIndex`.

— A partir de d'un objet de type `Square`, il est possible d'accéder directement à ses voisins via la méthode `getNeighbor(CardinalDirection direction)`. `CardinalDirection` est un `Enum` dont les valeurs sont `NORTH`, `EAST`, `SOUTH` et `WEST`. Ainsi, si l'objet `square` représente la case de coordonnées `(columnIndex,rowIndex)`, `square.getNeighbor(NORTH)` retourne l'objet de type `Square` qui correspond à la case de coordonnées `(columnIndex,rowIndex-1)`. Si `square` est placé sur un bord, il possède quand même un voisin, de type `EmptySquare` dans la direction de ce bord, ceci afin de faciliter la gestion de ces cas limites. Par exemple, si `square` est placé sur la dernière colonne, un appel à `square.getNeighbor(EAST)` retournera une instance d' `EmptySquare`.

— Le constructeur de la classe `ArrayGrid` prend en charge la création et le placement d'une instance de `MapSquare` sur chacune des cases de la grille. Il s'assure aussi que les voisins sont cohérents : si `square1` et `square2` sont côte à côte sur une même ligne, `square1.getNeighbor(EAST)` retourne `square2` et inversement `square2.getNeighbor(WEST)` retourne `square1`.

1.2.3 Murs entre cases : Classe `Wall` et Enum `WallType`

Un mur (ou l'absence de mur) entre deux murs est modélisé par la classe `Wall`. Chaque case possède 4 murs, un dans chaque direction cardinale `NORTH`, `EAST`, `SOUTH` et `WEST`. Pour représenter cela, chaque instance de `MapSquare` possède quatre instances de `Wall`, une pour chaque direction.

— Les instances sont partagées avec les cases des directions correspondantes. Ceci signifie que si par exemple `square1` et `square2` sont l'un à coté de l'autre sur la même ligne avec `square1` à l'est de `square2`, `square1` et `square2` ont la même instance de `Wall` dans les directions `EAST` et `WEST` respectivement. Soit `wall` cette instance commune. Toute modification de `wall` par `square1` sera donc immédiatement visible par `square2`.

— Un objet `wall`, instance de `Wall` peut avoir différents états, qui sont définis dans l'enum `WallType`. Cette énumérations a 3 valeurs possibles : `EMPTY`, `BREAKABLE` et `UNBREAKBLE`.
  — Un mur est présent si il est `BREAKABLE` ou `UNBREAKBLE`.
  — `EMPTY`correspond à un mur absent, qui a donc été supprimé.
  — Un mur `UNBREAKABLE` ne doit pas être supprimé. A l'initialisation de la grille, les murs sur les bords sont dans l'état `UNBREAKLE`.

Les méthodes suivantes permettent de manipuler les murs et leur état :

— `setWallType(WallType wallType, CardinalDirection direction)` dans la classe `MapSquare` : change l'état du mur du coté `direction` à `wallType` ;

— `WallType getWallType(CardinalDirection direction)` dans la classe `MapSquare` retourne l'état du mur du coté `direction`.

— Les méthodes `getWallType()` et `setWallType(WallType wallType)` de la classe `Wall` qui permettent d'obtenir et de modifier l'état d'une instance de `Wall`

— Les méthodes `isModifiable()`et `isEmpty()` qui permettent de déterminer si une instance de `WallType` est modifiable (i.e., dont la valeur est `BREAKABLE` ou `EMPTY`) ou vide (i.e., dont la valeur est `EMPTY`).

Ainsi, si l'on veut **supprimer le mur entre deux cases**, par exemple les cases (4,2) et (5,2) (4 et 5 ième colonne, 2ième ligne) il faut :

1. récupérer la case aux coordonnées (5,2) (méthode `getSquare()`) ;
2. obtenir l'état du mur dans la bonne direction, ici `EAST` (méthode `getWallType()`) ;
3. s'assure que le mur est bien modifiable (méthode `isModifiable()`) ;
4. enfin, changer l'état du mur à `EMPTY` (méthode `setWallType()`).

Puisque les cases aux coordonnées (4,2) et (5,2) partagent la même instance de `Wall` pour les cotés `EAST` et `WEST` respectivement, il n'est donc pas nécessaire de modifier l'instance de `Wall` de la case (5,2) du coté `WEST`. On peut bien sûr partir de la case (5,2) en remplaçant `EAST` par `WEST`. Encore une fois, supprimer un mur **ne consiste pas à** supprimer l'instance de `Wall` correspondante mais **à positionner l'attribut WallType** de cette instance à **EMPTY**.

1.3 Tâches à accomplir

Vous allez écrire plusieurs implémentations de l'interface `MazeGenerator`. Cette interface contient les méthodes suivantes :

— `setGrid(Grid grid)`, qui prend en paramètre l'instance de `Grid` sur laquelle le labyrinthe est généré. Cette instance est supposée correctement initialisée : les cases (instances de `MapSquare`) et les murs (instances de `Wall`) ont été crées et correctement initialisées. En particulier, les murs sur les bords sont de type `WallType.UNBREAKABLE` et les murs internes, `WallType.BREAKABLE`.
— `nextStep()`, qui effectue une étape de l'algorithme de génération.
— `hasNextStep()`, qui indique si l'algorithme de génération a terminé.

1.3.1 Pour démarrer

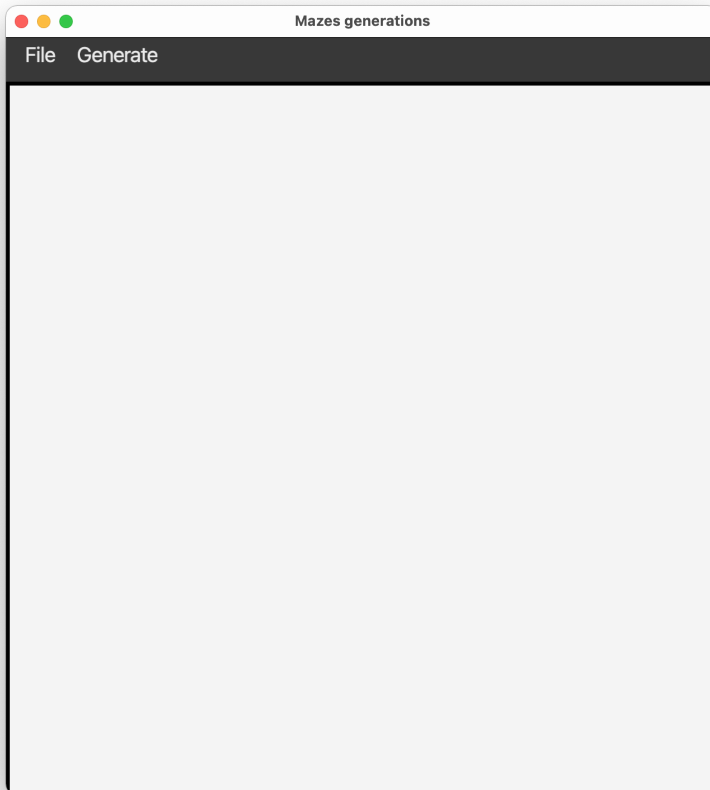
TODO
forker blaabla gitlab/github classroom

1.3.2 Tâche 1 : Classe `EmptyMazeGenerator`

Il s'agit de générer un labyrinthe vide, c'est-à-dire dans lequel tous les murs, à l'exception de ceux des bords, ont été enlevés. La méthode `nextStep()` enlèvera les murs restant d'une nouvelle case, en mettant leur type à `WallType.EMPTY`. On fera attention à ne pas modifier les murs externes, dont le type est `WallType.UNBREAKABLE`. La méthode `nextStep()` retournera `true` une fois l'ensemble des murs internes supprimés, et `false` sinon.

Notez que l'interface `Grid` étend l'interface `Iterable<Square>`. Vous pouvez donc parcourir les cases de la grille à l'aide d'un itérateur. La méthode `setGrid(Grid grid)` initialisera un attribut de type `Grid` avec la grille passée en paramètre. Elle créera aussi un itérateur de type `Iterator<Square>` qui sera utilisé par `nextStep()`et `hasNextStep()`.

Testez votre code en lançant l'application (`Gradle -> Tasks -> application -> run`), puis générer un labyrinthe vide (`Generate -> empty`). Vous devriez obtenir l'affichage suivant :

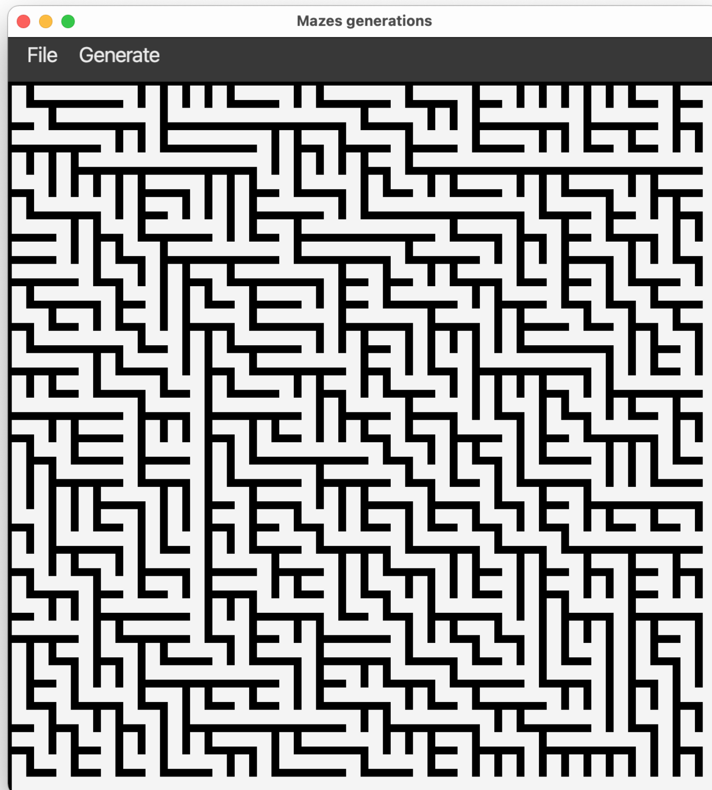


1.3.3 Tâche 2 : Classe `BinaryMazeGenerator`

Un algorithme générant des labyrinthes plus intéressant procède de la manière suivante : Pour chaque case de la grille, sélectionner aléatoirement une direction parmi {`EAST`, `SOUTH`} et supprimer le mur sur le coté correspondant à la direction sélectionnée, *si ce mur est modifiable*.

Compléter le code de la classe `BinaryMazeGenerator` qui met en oeuvre cet algorithme. Le constructeur prendra en paramètre un objet `random`, instance de la classe `Random` qui servira pour la sélection aléatoire. Pour sélectionner un élément d'un tableau ou d'une liste de façon aléatoire, on pourra utiliser les méthodes static de la classe `util.RamdomUtil` qui est fournie.

Testez votre code en lançant l'application (`Gradle -> Tasks -> application -> run`), puis générer un labyrinthe (`Generate -> from random binary tree`). Vous devriez un affichage similaire à l'image suivante :



1.3.4 Tâche 3 : Classe `RandomWalkGenerator`

L'un des inconvénient de l'algorithme précédent est que les labyrinthes générés contiennent des couloirs qui courent le long des bords *est* et *sud*, ce qui en facilite grandement leur résolution. Nous allons remédier à cela en effectuant des marches aléatoires.

Dans une marche aléatoire, une case est initialement choisie au hasard dans la grille. A chaque *étape*, une case voisine de la case courante est sélectionnée au hasard. Cette voisine devient la nouvelle case courante. Si on laisse ce processus se dérouler suffisamment longtemps, toutes les cases de la grilles seront visitées.

Pour générer un labyrinthe, on effectue les actions suivantes à chaque *étape* de la marche aléatoire :

— Si la case sélectionnée n'a pas été encore visitée (c'est la première fois que la marche passe par cette case), alors supprimer le mur entre la case courante et la case sélectionnée.

— Sinon, on ne modifie pas les murs de la grille.

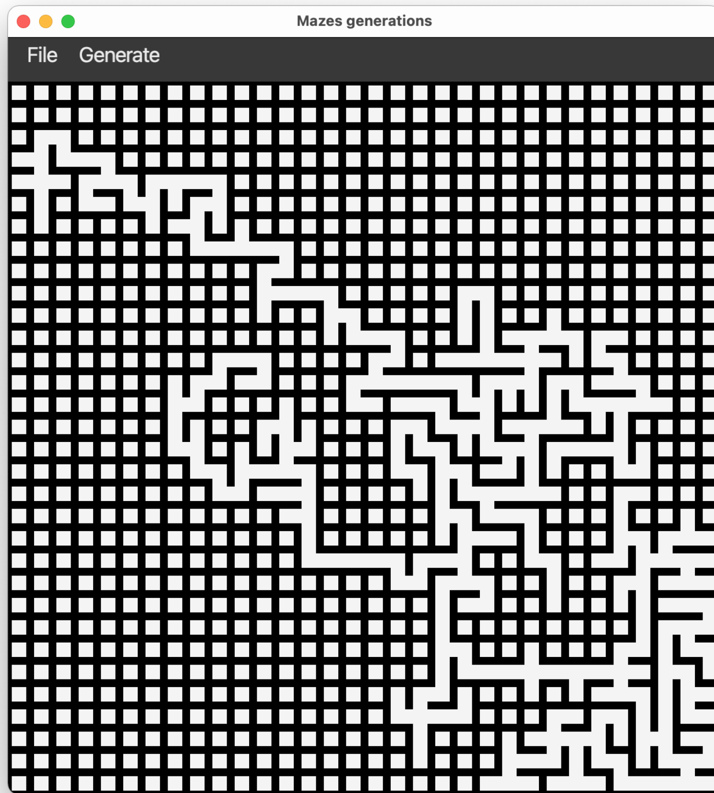
L'algorithme s'arrête quand toutes les cases ont été visitées. Au cours de la marche, il faut faire attention à ne pas sortir des cases de la grille. On pourra pour cela faire appel aux méthodes publiques de la classe `SelectDirection`. Il faut aussi se rappeler des cases déjà visitées par la marche. Vous utiliserez pour cela une instance de la classe `Marker` qui est fournie, ainsi que l'énunmération `MarkType`.

Implémentez la classe `RandomWalkGenerator`. Votre classe aura les attributs suivants, ainsi que peut-être d'autres que vous déterminerez :

— `marker` de type `Marker<Square,Marktype>`

— `random` de type `Random`, initialisé par le constructeur.

Testez votre code en lançant l'application (`Gradle -> Tasks -> application -> run`), puis générer un labyrinthe (`Generate -> from random walk`). Vous devriez obtenir des affichages du labyrinthe en formation similaire à l'image suivante :



1.3.5 Tâche 4 : Classe `DFSMazeGenerator`

L'inconvénient de la méthode précédente est son long temps de terminaison. Un grand nombre d'étapes peut-être nécessaire avant de passer sur une case qui n'a pas encore été visitée. Pour pallier à cela, nous allons mettre en oeuvre une autre stratégie fondée sur un parcours en profondeur. Les principaux éléments sont les suivants :

- Comme pour la marche aléatoire, les cases visitées sont marquées (à l'aide d'une instance de `Marker<Square,MarkType>`)
- De plus, chaque case lorsqu'elle est visitée pour la première fois est placée dans une pile. Cette structure de données va permettre de retrouver plus facilement les cases non visitées voisines de cases déjà visitées.

L'algorithme procède ainsi :

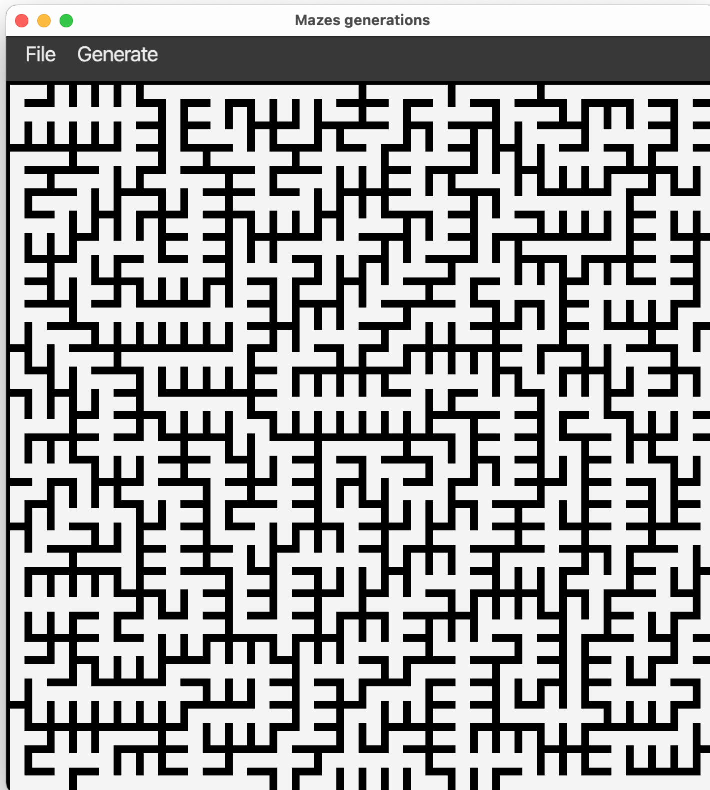
- Initialement, une case est choisie (par exemple celle dans le coin supérieur gauche). Elle est marquée comme visitée et placée dans la pile.
- A chaque étape, la case au sommet de la pile est dépilée. Chacune de ses voisines est examinée *dans un ordre aléatoire.*
  - Si elle n'a pas été visitée, le mur entre cette voisine et la case dépilée est supprimé, la voisine est marquée comme visitée et ajoutée à la pile.
  - Sinon elle est ignorée.

L'algorithme se termine quand la pile est vide.

Mettre en oeuvre cet algorithme dans la classe `DFSMazeGenerator`. Le constructeur prendra en paramètre un objet `random` de type `Random`. Les structures de données nécessaires seront (ré)initialisées dans la méthode `setGrid(Grid grid)`. On pourra utiliser le type `java.util.Stack<E>`. La méthode

`java.util.Collections.shuffle(List<?> list, Random random)` permet d'ordonner aléatoirement les éléments d'une liste. On fera attention à la gestion des bords.

Testez votre code en lançant l'application (`Gradle -> Tasks -> application -> run`), puis générer un labyrinthe (`Generate -> from random DFS`). Vous devriez obtenir un affichage similaire à l'image suivante :



1.3.6 Tâche 5 : Classe `PrimMazeGenerator`

Le point commun des trois méthodes précédentes est de supprimer des murs entre cases visitées et cases non visitées. Dans cette nouvelle approche, on va marquer les cases comme *visitées* ou *non-visitée*.

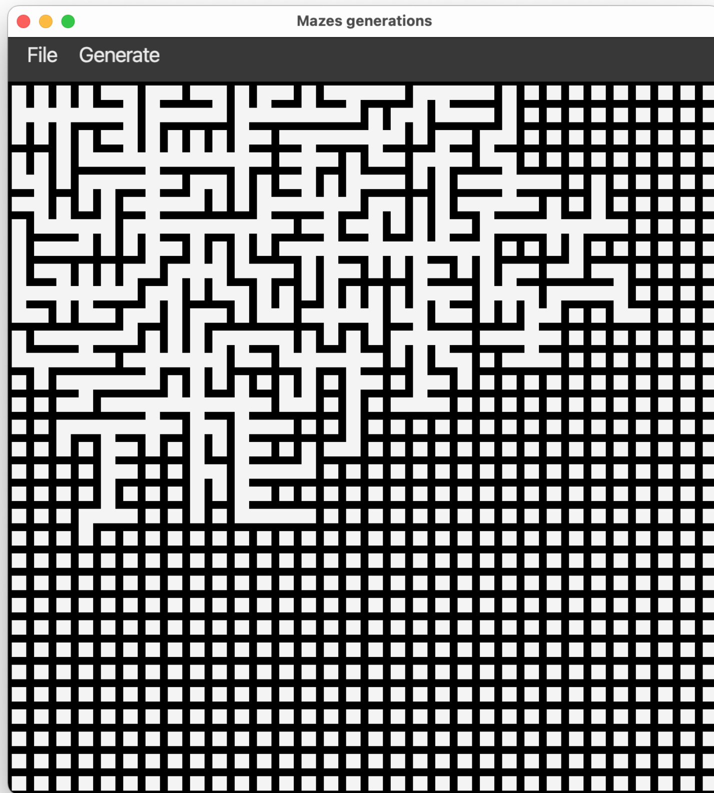
— A chaque étape, une case *non-visitée* est sélectionnée au hasard. Ensuite, l'une des ses voisines marquée *visitée* est sélectionnée au hasard et le mur les séparant est supprimé. La case devient *visitée* et ses voisines, si elles ne sont pas marquées, deviennent *non-visistées*.

— Initialement, une case est choisie aléatoirement et marquée *visitée*. Ses voisines sont marquées *non-visitées*.

— L'algorithme se termine lorsque toutes les cases sont *visitées*.

Implémenter cette stratégie dans la classe `PrimMazeGenerator`. Vous ferez attention à la gestion des bords. Vous utiliserez une instance de type `Marker` pour marquer les cases, ainsi que des méthodes des classes `RandomUtil` et `SelectDirection`.

Testez votre code en lançant l'application (`Gradle -> Tasks -> application -> run`), puis générer un labyrinthe (`Generate -> from random Prim's algorithm`). Vous devriez obtenir des affichages du labyrinthe en formation similaire à l'image suivante :



1.4 Tâches avancées

1.4.1 Tâche 6 : Export

Réaliser l'export de labyrinthe au format `.maz` :

— Un labyrinthe est décrit lignes par lignes, chaque ligne représentant une ligne complète de murs horizontaux ou verticaux.

— Les lignes de murs horizontaux alternent avec les lignes de murs verticaux.

— un mur horizontal est décrit par `---` si il est plein et (3 espaces) s'il est vide. Les murs horizontaux sont séparés par `o` (o minuscule)

— un mur vertical est décrit par `|` s'il est plein, (1 espace) s'il est vide. Les murs verticaux sont séparés par (3 espaces).

Des exemples (`snake.maz`, `snail.maz`, `unperfect.maz`) sont fournis ans le répertoire `app/` du projet. Vous pouvez les ouvrir avec n'importe quel éditeur, y compris avec IntelliJ.

Compléter la classe `MatrixOfWallsWriter` qui implémente l'interface Iterable<String>. Chaque appel à `next()` renverra sous forme de chaîne de caractères une ligne complète de murs soit horizontaux, soit verticaux. Ces lignes *ne devront pas* se terminer par un retour à la ligne `\n`. Vous pourrez utiliser les méthodes statiques de la classe `util.WriteUtil` qui est fournie.

Testez votre code en lançant l'application (`Gradle -> Tasks -> application -> run`), puis générer un labyrinthe (`File -> Export`). Vous pouvez ensuite afficher le labyrinthe exporté (`File -> Open`) ou ouvrir le fichier `.maz` généré dans un éditeur.

1.4.2 Tâche 7 : Classe `SpanningTreeMazeGenerator`

Dans cette tâche, nous adoptons la stratégie suivante : Nous maintenons sous forme d'arbres les cases qui sont connectées entre elles. Deux cases sont connectées s'il existe un chemin entre elles qui ne traverse que des murs vides.

- Initialement, il y a autant d'arbres que de cases. La grille étant en effet initialement complètement murée, il n'existe pas de paires de cases connectées. Chacun des murs internes est marqué *non-visité*
- A chaque étape, un nouveau mur *non-visité* est choisi au hasard. Les cases qu'il sépare sont examinées. Si elles sont déjà connectées entre elles, le mur est laissé tel quel. Sinon, le mur est supprimé. Dans tout les cas, le mur est marqué *visité*.
- L'algorithme s'arrête lorsque l'ensemble des cases est connecté.

1.4.3 Tâche 7.1 : Classe interne `Node`

Comme indiquer ci-dessus, on va utiliser des arbres pour représenter des ensembles de cases connectées. Créer une classe interne statique `Node` qui a pour attributs :

- `parent`, de type `Node`
- `hasParent` de type `boolean`
- `square` de type `Square`

Cette classe a aussi un constructer qui prend un paramètre de type `Square` et les méthodes évidentes `getParent()`, `setParent()`, `hasParent()` et `getSquare()`.

1.4.4 Tâche 7.2 : Union/find

Pour savoir si deux cases sont connectées, on va déterminer si les instances de `Node` qui les contiennent ont la même racine. Ajouter une méthode `Node findRoot()` qui retourne la racine de l'arbre contenant `this`.

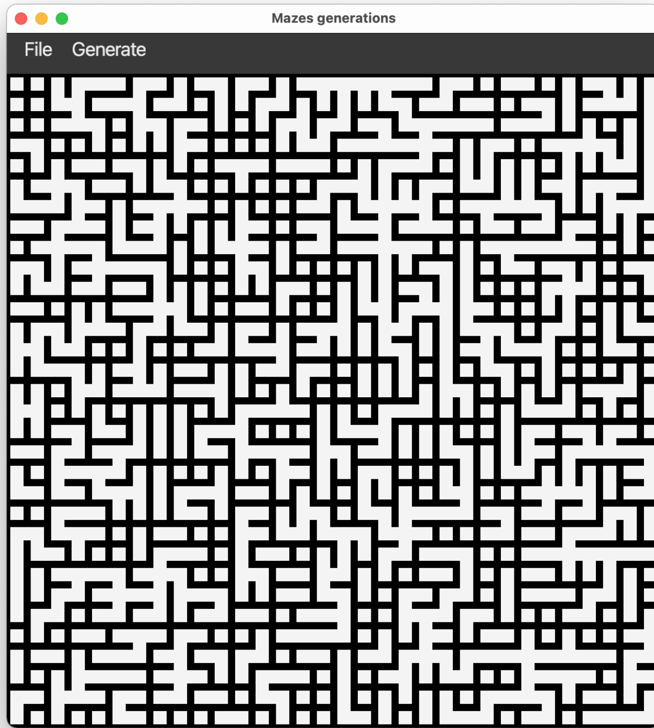
Enfin, la suppression d'un mur va entraîner la connection de cases d'arbres disjoints. Pour refléter cela, ces arbres doivent être fusionnés. Ajouter une méthode statique `union(Node rootA, Node rootB)` qui ne renvoie rien et positionne le parent de `rootB` à `rootA`. Écrire des tests et tester !

1.4.5 Tâche 7.3 : Terminer le code de la classe `SpanningTreeMazeGenerator`

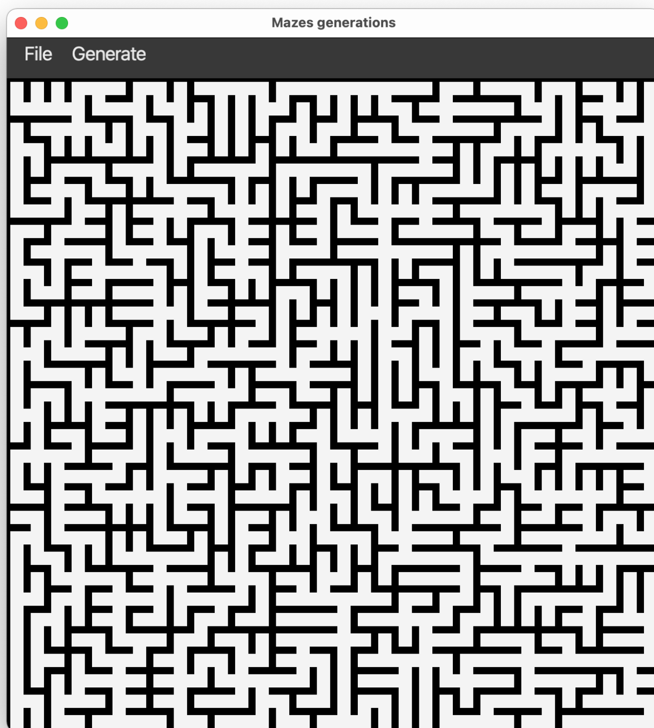
Pour terminer, implémenter l'algorithme de génération dans la classe `SpanningTreeMazeGenerator`. Utiliser une instance de `Marker` pour marquer les murs internes, qui peuvent être récupérées au moyen de la méthode `getInternalWalls()` de la classe `Grid`. Les cases qu'un mur sépare peuvent être accédées au moyen de la méthode `getExtremities()`de la classe `Wall`.

Vous aurez aussi probablement besoin de pouvoir retrouver à partir d'un objet `square`, instance de `Square`, l'instance de `Node` qui le contient. Ceci peut se faire à l'aide d'un dictionnaire de type `Map<Square,Node>`.

Testez votre code en lançant l'application (`Gradle -> Tasks -> application -> run`), puis générer un labyrinthe (`Generate -> from spanning tree`). Vous devriez obtenir des affichages du labyrinthe en formation similaire à l'image suivante :



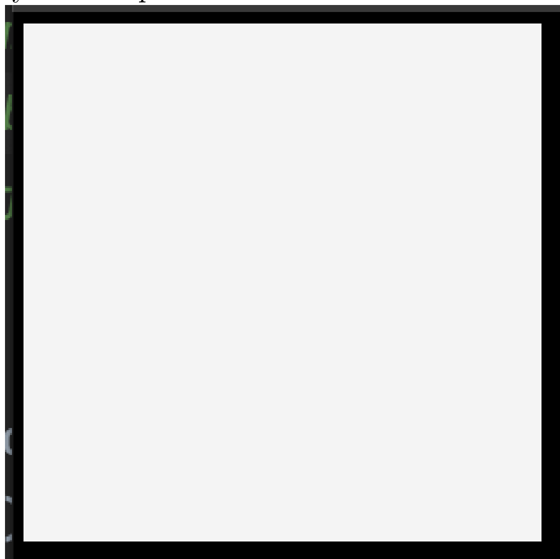
Le résultat final devrait être similaire à l'image suivante :



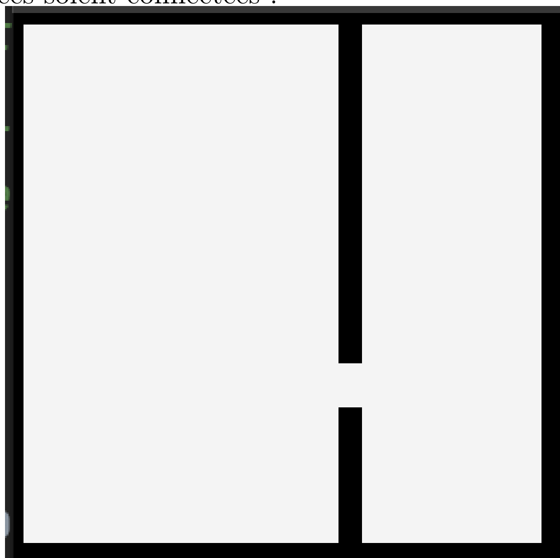
1.4.6 Tâche 8 : Classe `RandomDivisionMazeGenerator`

Un dernière stratégie consiste à récursivement divisé des zones vides de la grille au moyen d'une série de murs verticaux ou horizontaux, en laissant une *porte* entre les zones nouvellement créées.

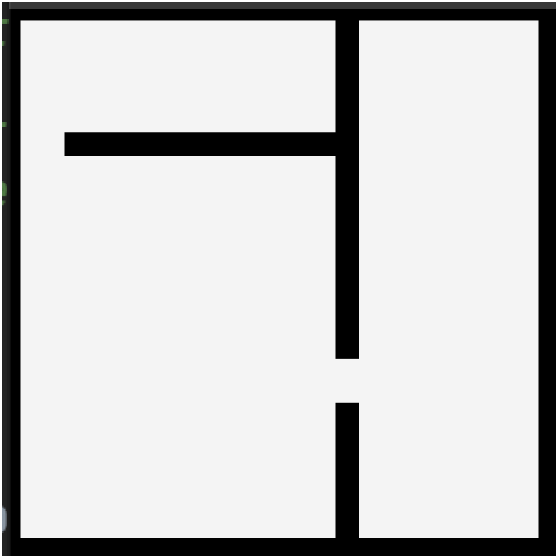
On appellera *chambre* une partie rectangulaire de la grille dépourvue de murs internes. Initialement, on démarre d'une grille dans laquelle tous les murs internes ont été supprimés (les murs des bords sont conservés). Il n'y a donc qu'une seule chambre :



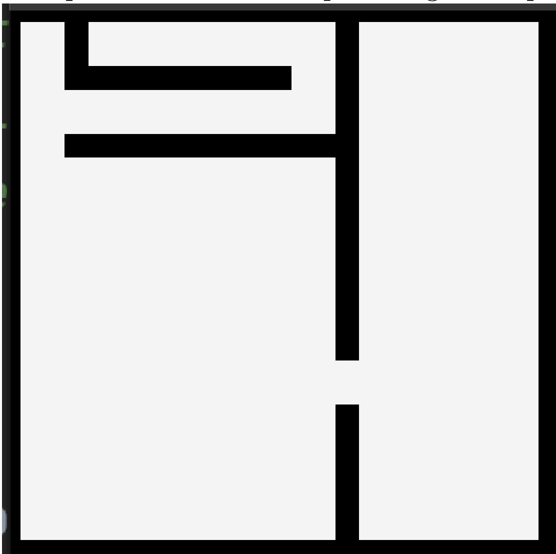
Ensuite, une colonne est choisie au hasard et des murs verticaux sont construits le long de cette colonne. Un mur (choisi au hasard) de cette colonne est alors supprimé de façon à ce que les deux chambres nouvellement créées soient connectées :



La procédure est appliquée récursivement sur les deux nouvelles chambres, ici d'abord la chambre de gauche, puis sur celle de droite.
Elles sont divisés par des murs cette fois horizontaux, toujours en laissant un passage de façon à assurer que l'ensemble des chambres soient connectées :



Lorsqu'une chambre n'a qu'une ligne ou qu'une colonne, elle n'est pas divisée et la récursion s'arrête.

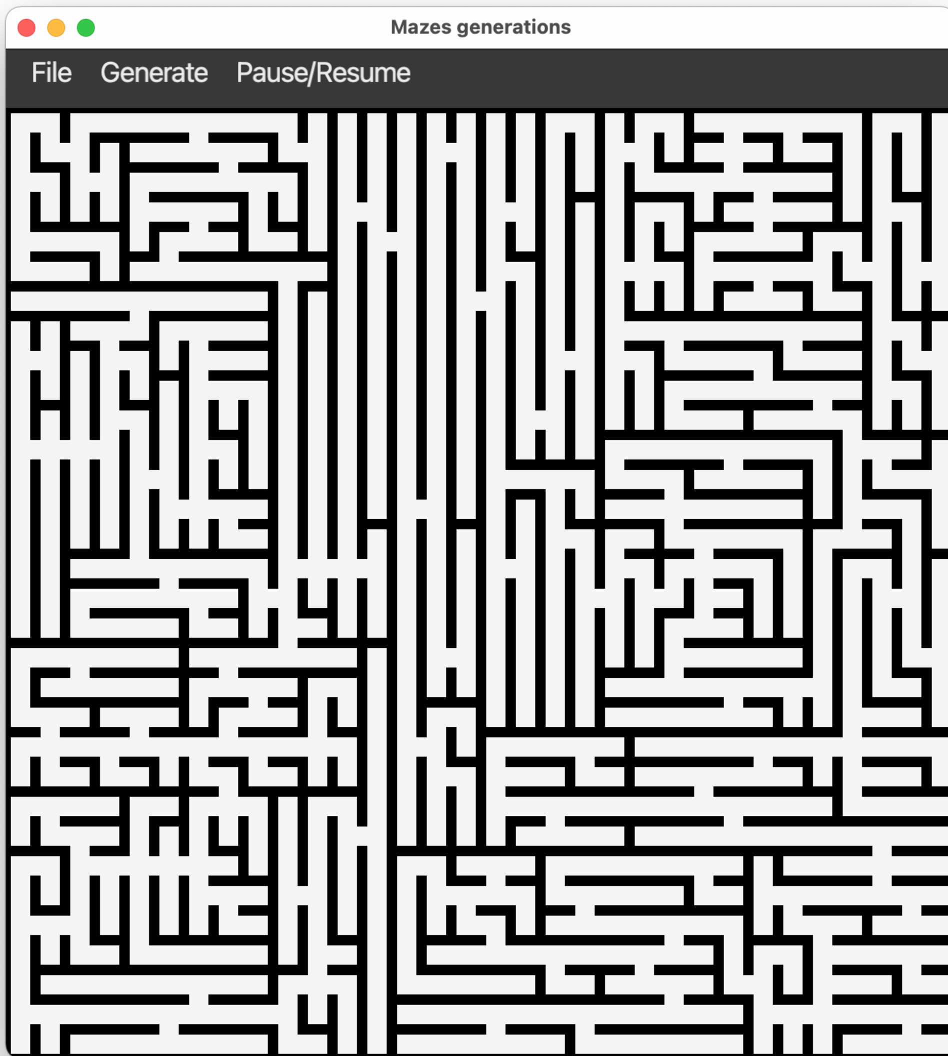


Vous pouvez visualiser une exécution complète de cet algorithme sur une grille plus grande ici.

A faire : Implémenter l'algorithme de division aléatoire dans la classe `RandomDivisionMazeGenerator`. Voici quelques indications :

- Vous pouvez implémentez une classe interne `Room` qui représentant une chambre vide à diviser. Cette classe aura des attributs qui décrivent la chambre, comme par exemple les coordonnées de la case dans son coin supérieur gauche, son nombre de colonnes et son nombre de lignes. D'autres attributs seront peut-être nécessaire.
- Pour simuler les appels récursifs, la méthode `nextStep()` pourra placer les chambres nouvellement créées et donc à diviser ultérieurement dans une pile
- L'algorithme terminera donc lorsque cette pile est vide.

Testez votre code en lançant l'application (`Gradle -> Tasks -> application -> run`), puis générer un labyrinthe (`Generate -> from random division`). Vous devriez obtenir des affichages du labyrinthe en formation similaire à l'image suivante :


Mazes generations
File
Generate
Pause/Resume