

TELECOM SudParis

Projet Informatique 1<sup>ère</sup> Année

PRO3600\_21\_HEN\_09

# PATHFINDING

LE BERRE Jean

LE GAC Tangi

LORGE Denzel

RANDRIA Joshua

Enseignant responsable : **HENNEQUIN Pascal**



INSTITUT  
POLYTECHNIQUE  
DE PARIS

# Tables des Matières

<b>Introduction</b>	<b>3</b>
<b>Cahier des charges</b>	<b>4</b>
Génération des labyrinthes :	4
Visualisation de la mise en oeuvre d'algorithmes de pathfinding :	4
<b>Développement</b>	<b>5</b>
Analyse des problématiques du projet	5
Conception préliminaire	5
Conception des algorithmes	9
Tests	11
<b>Manuel utilisateur</b>	<b>13</b>
<b>Travail de groupe</b>	<b>16</b>
<b>Conclusion</b>	<b>17</b>
<b>Annexes</b>	<b>18</b>

# 1. Introduction

Dans le cadre du projet 3600, nous avons choisi de nous intéresser au pathfinding. Ce domaine consiste à trouver le chemin pour se déplacer dans un espace entre deux points. Il est utilisé à la fois dans les jeux vidéo pour le déplacement des personnages dans leur monde, dans l'optimisation des transferts de données en télécommunication mais aussi dans des systèmes de navigation ou de modélisation 3D pour la mécanique. Le pathfinding peut s'avérer extrêmement complexe. Nous nous proposons donc d'explorer un peu ce sujet dans le but d'en apprendre davantage sur l'algorithmique. La disposition des points de départ et d'arrivée, l'espace choisi permet à l'un des algorithmes connus de se démarquer des autres suivant les cas. Cependant certains algorithmes de recherche du plus court chemin sont plus performants que d'autres. La plupart de ceux-ci sont déterministes même si on commence à voir apparaître du machine learning dans certains projets de recherche. Une application qui permettrait de visualiser ces algorithmes permettra de comprendre ces algorithmes plus facilement en se le représentant plus instinctivement. Le tout dans des labyrinthes variables et générés aléatoirement suivant des critères.

## 2. Cahier des charges

Le but de cette application est de visualiser les algorithmes de pathfinding dans un labyrinthe plan en deux dimensions. Les fonctionnalités qui devront être présentes dans l'application graphique sont:

### 1. Génération des labyrinthes :

Paramètres réglables de création des labyrinthes :

- Taille (largeur, hauteur)
- Choix de l'algo de génération
- Vitesse du tracée
- Choix de l'entrée/ sortie de manière manuelle et aléatoire

### 2. Visualisation de la mise en oeuvre d'algorithmes de pathfinding :

Paramètres réglables de résolution des labyrinthes :

- Vitesse d'exécution
- Choix de l'algorithme ou affichage de tous

Ces fonctionnalités sont à implanter dans un premier temps avec une interface mode "terminal", puis avec une interface graphique.

Solutions choisies pour la visualisation :

- JavaFx
- Affichage du nombre d'opérations
- Affichage du nombre de cases visités / nombre de cases totales
- Classement des algorithmes à la fin de chaque résolution

Dans un deuxième temps nous aimerions implémenter les fonctionnalités suivantes:

- Ajouter un nombre de voisins variant de 2 à 7 et donc changer la forme des cellules.
- IA de résolution, apprentissage par héritage (Sans doute trop ambitieux)  
<https://www.youtube.com/watch?v=s8MT-5X7cyk&t=300s>

## 3. Développement

### 3.1. Analyse des problématiques du projet

Les premiers problèmes qui nous sont apparus ont été de gérer l'affichage. Nous avons choisi JavaFx car nous avions un cours de prévu dessus. Des problématiques de mise en page, d'interactions hommes-machines et de changement de page nous ont tout d'abord concerné. Mais le cours dispensé en PRO3600 et la documentation sur Internet à suffit à créer un squelette graphique de notre application. Ensuite, nous nous sommes posées la question de la manière de représenter le labyrinthe et de comment le stocker tout au long de son cycle de vie. Puis enfin s'est posé la question de comment passer des données à l'affichage, de l'animation via JavaFx.

### 3.2. Conception préliminaire

Nous avons choisi l'architecture logicielle Model-View-Controller pour ce projet. Le programme se décompose donc ainsi :

- `main.java` : le fichier de départ du logiciel, il permet aussi de changer de fenêtre et l'interaction entre les parties du programme.
- `model` : le package qui s'occupe de toute la partie intelligente du programme. Les algorithmes de génération et de résolution se trouvent dans cette partie ainsi que les structures de données permettant le stockage des labyrinthes et des différents paramètres.
- `controller` : le package qui s'occupe de l'interaction homme-logiciel par l'intermédiaire de widgets présents sur les différentes interfaces graphiques.
- `view` : le package qui contient les interfaces graphiques faites à partir de fichier `.fxml`.

Le stockage du labyrinthe et de ses propriétés se fait à partir d'une nouvelle instance `static` dans le fichier `Main`. Cette instance, de classe `informationLaby`, comporte des attributs privés et les `getter` et `setter`. Ainsi, on peut y avoir accès et la mettre à jour de n'importe quelle partie du programme et tout au long de la période de vie du labyrinthe : de la création d'une grille vierge à la publication des résultats en passant par la génération et la résolution. Les données utiles à la création du labyrinthe sont sa taille, son temps d'animation, sa méthode de génération et sa méthode de résolution. Elles sont sélectionnées sur la page d'accueil puis stockées lors du passage à la page de génération. Ensuite, les étapes de génération utiles à l'animation sont elles aussi stockées. Il en est de même pour

les étapes de résolution et les résultats de résolution lorsque l'on arrive sur la page de résolution. Ces données sont ensuite effacées lorsque l'on revient à la page d'accueil.

On a choisi, pour simplifier l'affichage, de considérer l'élément de base du labyrinthe comme une pièce comportant quatre portes menant chacune vers un des points cardinaux. Chacune de ces portes mènent directement vers une pièce voisine. Voici sa représentation graphique :



Figure 1

On se servira de la classe `Case` pour afficher ces murs, portes et pièces. Une `Case` est défini par ses coordonnées dans la grille de `Case` (de dimension  $(2*\text{nombrePieceHauteur}+1)*(2*\text{nombrePieceLongueur}+1)$ ). Ces index permettent de placer un `Rectangle` (`javafx.scene.shape.Rectangle`) d'une couleur défini à l'écran.

Les pièces sont obtenues grâce à la classe `Piece` qui hérite de `Case`. En plus des caractéristiques de `Case`, une pièce a des voisins et des booléens pour savoir si les portes sont ouvertes pour chaque point cardinal. Si il n'y a pas de voisin, sur les bords par exemple, le voisin est égale à `null`.

*Remarque* : les attributs `visite` et `kruskalIndex` aurait dû se trouver dans un classe fille de `Piece` nommée `PieceKruskal`.

Les labyrinthes héritent tous de `arbreLabyrinthe`. Cette classe abstraite contient la fonction abstraite `generer` qui permettra de lancer la partie algorithmique de chaque méthode. `arbreLabyrinthe` construit une grille de `Piece` (de dimension `nombrePieceHauteur*nombrePieceLongueur`) en reliant chacun avec ses voisins. Elle contient aussi des fonctions pour calculer le taux de visite, pour choisir le départ et l'arrivée au hasard et pour afficher le labyrinthe dans la console. La fonction `generer` prend en argument `Case[][]` car en plus de

lancer l'algorithme, elle remplit une `LinkedList<Case[]>` et un tableau de couleur pour sauvegarder les étapes importante de l'algorithme à afficher sur la fenêtre.

Pour afficher ce labyrinthe on a choisi de représenter les pièces, les murs et les portes par des Case de même dimension. On affiche donc à l'écran une grille de Case de dimension  $(2*\text{nombrePieceHauteur}+1)*(2*\text{nombrePieceLongueur}+1)$  avec des couleurs différentes suivant les informations que l'on veut faire passer.

On a pris pour convention que pour les case de couleur :

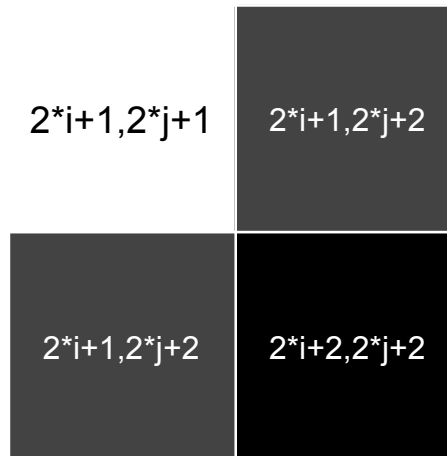
- BLANC : Case faisant partie du labyrinthe
- NOIR : Case non accessible
- JAUNE : Case faisant partie du chemin de résolution

Voici en dessous une figure schématique de la représentation d'un labyrinthe "vide" de taille 3x4 (c'est à dire que toutes les portes sont fermés) :

0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8
1,0	0,0	1,2	0,1	1,4	0,2	1,6	0,3	
2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7	2,8
3,0	1,0	3,2	1,1	3,4	1,2	3,6	1,3	3,8
4,0	4,1	4,2	4,3	4,4	4,5	4,6	4,7	4,8
5,0	2,0	5,2	2,1	5,4	2,2	5,6	2,3	5,8
6,0	6,1	6,2	6,3	6,4	6,5	6,6	6,7	6,8

Figure 2 : Représentation graphique du labyrinthe

On rappelle que la pièce  $i,j$  du labyrinthe est représentée par 9 Cases (ref. figure 1). Cependant, la première ligne et la première colonne sont toujours en noirs. Et, comme chaque une porte Nord (resp. Ouest) est aussi une porte Sud (resp. Est) d'une autre pièce, on choisit d'afficher une pièce à l'aide de 4 Cases : la pièce, sa porte Sud, sa porte Est et le mur du coin en bas à droite.



C'est suivant cette logique qu'est construite la fonction `dessinerLaby` dans `LabyrintheAffichage`. Cette fonction sert à dessiner le labyrinthe avec le départ en vert et l'arrivée en rouge en arrivant sur la page de résolution.

Une fois que l'algorithme est effectué et que la `LinkedList<Case[]>` et le tableau de couleur sont stockés, on peut procéder à l'étape d'animation. Pour cela on utilise la bibliothèque `javafx.animation`. L'animation élémentaire est de la classe `FillTransition`, elle permet de changer la couleur d'un rectangle sur une durée définie. Ainsi, en appliquant en parallèle l'animation élémentaire à chaque `Case` d'une `Case[]` de la `ListLinked` qui stocke les étapes logiques de fonctionnement de l'algorithme, chacune de ces `Case` va changer de couleur en même temps. En mettant en séquentiel ces animations parallèles suivant l'ordre de la `LinkedList`, on peut suivre l'évolution de l'algorithme qui a été à l'œuvre. Le tableau de couleur permet de choisir la couleur des animations en parallèle.



### 3.3. Conception des algorithmes

On s'intéressera ici à la conception des différents algorithmes, pour revoir la logique algorithmique veuillez regarder dans le livrable 1. Tous ces fichiers sont dans leurs packages respectifs dans la partie model. Chaque algorithme est composé d'au moins deux classes : `PieceAlgorithme` qui hérite de `Piece`, et `AlgorithmeResolve` ou `AlgorithmeResolve` qui hérite de `arbreLabyrinthe`. Les différentes explications parleront de l'implémentation de l'algorithme en Java et des choix d'animations.

#### 3.3.1. Algorithme de Kruskal

C'est le premier algorithme produit, il comporte donc le défaut de ne pas avoir son package et de ne pas avoir de classe `KruskalPiece`. Ici, `kruskalIndex`, un entier qui identifie l'appartenance à un groupe de pièces communicantes, se trouve dans `Piece`. Initialement, chaque case à un index différent que l'on range dans `sameKruskalIndex`. `sameKruskalIndex` est de classe `ArrayList<LinkedList<Case>>`, il permet de ranger chaque case dans un groupe de cases ayant le même `kruskalIndex`. Ensuite, on itère sur n-1 portes fermées. Pour cela, on choisit les coordonnées d'une pièce au hasard ainsi qu'une de ses 4 portes. Cela est fait grâce au tableau d'entiers `randomMur` qui prend 3 valeurs aléatoires à chaque itération. Après avoir vérifié qu'elle soit fermée ou qu'elle mène bien vers une autre pièce (`conditionRaccordementKruskal`), on ouvre la porte et met à jour les `kruskalIndex` ainsi que leur stockage (`ouvrirPorte`). On répète l'opération n-1 fois et on obtient le labyrinthe. Pour l'animation, à chaque itérations on montre en jaune la porte fermée sélectionnée qui s'ouvre, puis on anime en parallèle toutes les cases ayant le même `kruskalIndex` que la pièce prise au hasard lors de l'itération. La couleur de l'animation est choisie aléatoirement grâce au code couleur `rgb`.

#### 3.3.2. Algorithme de Trémaux

La classe `pieceTremaux`, qui hérite de `Piece`, permet d'ajouter des marqueurs sur les portes des jonctions. Lorsque l'on intègre ces nouvelles `pieceTremaux` pour remplacer les anciennes `Piece`, il faut mettre à jour les voisinages dans les deux sens (de la nouvelle pièce vers les voisins et des voisins vers la nouvelle pièce). Les jonctions sont les pièces cul de sac ou qui ont au moins deux portes voisines ouvertes. Les marqueurs sont implémentés par les entiers `compteurDirection` initialisés à zéro si la porte est ouverte à 2 sinon. Au début de la résolution, on choisit une direction possible aléatoirement à partir du départ (`premiereDirection`) puis on marque la porte utilisée, on incrémente

compteurDirection. Ensuite, on itère jusqu'à temps que l'actualPiece soit l'arrivée en procédant comme il va être expliqué. Si la pièce est une jonction, on la transforme en pieceTremaux si ce n'est pas déjà le cas, on change de direction suivant les priorités suivantes. On fait le choix parmi les directions possibles, c'est-à-dire les direction dont les compteurDirection $\leq$ 2. Premièrement, on priorise les directions dont compteurDirection=0, pour connaître ces directions on regarde si un nouveau tableau directionsVierge n'est pas vide. Ensuite, si l'entrée dont on vient a un compteur à 1 on fait demi-tour. Et sinon, on choisit aléatoirement parmi directionsPossible. Bien sûr, à chaque fois on n'oublie pas de marquer le passage en arrivant et en sortant de la pièce. On remarque qu'il y a une securite pour éviter que la boucle while tourne à l'infini. Cette entier permet aussi de compter le temps d'exécution du programme.

Pour l'animation, on ajoute actualPiece à chaque itération. Pour la couleur de l'animation, on choisit le orange si c'est la première fois que l'on passe ( compteur à 1 ), sinon on met une couleur plus foncée. Il faut aussi faire attention en cas de repassage sur la case de départ dans lequel cas il faut aussi continuer en orange. A la fin, le tracé en jaune montre le chemin.

### 3.3.3. Algorithme de Dijkstra

La classe PieceDijkstra permet d'attribuer des coûts aux portes et aux pièces afin d'établir un chemin avec un coût minimal entre le départ et l'arrivée. Pour la résolution, la classe DijkstraResolve hérite de arbreLabyrinthe et mets en oeuvre la méthode générer de la façon suivante : une phase d'initialisation et une boucle principale composée de grandes méthodes. On part avec un ensemble S composé de la pièce départ que l'on prendra comme pivot. Tant que S ne contient pas la pièce arrivee, on itère sur le pivot. On étudie les pièces voisines de la pièce pivot à l'aide de la méthode etudeVoisins, on vérifie qu'elles sont accessibles via des portes ouvertes (porteOpen) et qu'elles ne font pas déjà partie de S (contenuDans), afin de ne pas revenir sur nos pas. On ajoute alors les pièces sélectionnées à l'ensemble X des pièces atteintes, et on regarde si le coût du chemin pour les atteindre est supérieur à celui que l'on suit actuellement et dont pivot possède le coût. Si c'est le cas, on remplace le coût pour atteindre ces pièces voisines en sommant le coût du chemin jusqu'au pivot et le coût du passage de la porte. On a alors des chemins de coût minimal à chaque itération. Ensuite, grâce à la fonction newPivot, on choisit parmi les pièces de X/S (les plus "lointaines" de depart et donc plus proche de arrivee) celle possédant le coût de chemin minimal pour en faire notre nouveau pivot. Si plusieurs pièces possèdent une même valeur, on choisit arbitrairement nord, est, sud puis ouest. On itère alors à nouveau sur ce pivot que l'on a ajouté à S au préalable. Ainsi toutes les pièces atteignables en un coût inférieur au pivot sont dans S et les autres sont soit dans X soit pas encore atteintes. Cela nous assure d'atteindre arrivee avec un coût minimal car comme la

génération de kruskal ne permet qu'un seul chemin, il n'est pas nécessaire d'aller jusqu'à l'itération où départ devient pivot (synonyme qu'il ne peut être atteint plus vite). Tout au long de cette boucle, on met à jour l'affichage grâce aux différentes fonctions `coloriage`.

A la fin de la boucle on a atteint départ avec un coût minimal, On remplit alors le tableau des résultats.

### 3.3.4. Algorithme du Mur droit

La classe `Aventurier` permet de simuler une entité se déplaçant au sein du labyrinthe, de définir son repère propre et d'interagir avec. En effet, cet algorithme repose sur un principe simple: suivre en permanence le mur situé sur sa droite. Il est donc nécessaire de modéliser une entité capable de distinguer son avant et son arrière, sa gauche et sa droite (ici personnifiée par un aventurier pour une visualisation plus simple). On désignera par la suite direction comme étant les points cardinaux (repère fixe du labyrinthe) et orientation le repère mouvant de l'aventurier. La méthode `rechercheDirection` permet à tout moment de faire corrélérer les deux repères en question pour savoir à une position quelconque quelle orientation correspond à quelle direction.

Points cardinaux	Nord	Est	Sud	Ouest
Valeurs points cardinaux	0	1	2	3
Valeurs Orientation	0 Face	1 Droite	2 Arrière	3 Gauche



Rotation de 180° (2 crans)

Points cardinaux	Nord	Est	Sud	Ouest
Valeurs points cardinaux	0	1	2	3
Valeurs Orientation	2 Arrière	3 Gauche	0 Face	1 Droite

Ensuite la méthode `Decision` permettra de définir la porte à prendre suivant la pièce dans laquelle l'aventurier se trouve. Il n'y a que 5 situations possibles :

- 1) Il y a un passage à droite (tourne à droite)
- 2) Il y a un passage à gauche et tout droit (continue tout droit)
- 3) Il y un passage seulement à gauche (tourne à gauche)
- 4) Il y a un passage seulement tout droit (continue tout droit)
- 5) C'est un cul-de-sac (fait demi-tour)

La méthode `rotation` permettra ensuite de tourner l'aventurier de la bonne manière si on rentre en donnée le résultat de la méthode précédente.

La classe `Murdroit_Resolve` va surtout servir à agencer les méthodes précédentes et à les mettre en lien avec le labyrinthe généré aléatoirement. Après l'initialisation réalisée et l'aventurier orienté par défaut au Nord pour commencer, l'algorithme entre dans une boucle dont il ne peut partir que lorsque la sortie a été trouvée (`Detectionfin(position) == true`). A chaque itération, la méthode d'Aventurier `Decision` va être appelée sur la pièce actuelle. La pièce suivante va être choisie et l'aventurier sera tourné pour toujours avancer de face. L'algorithme se déplacera dans la matrice de cases suivant les points cardinaux (ex: Nord = ligne du dessus). En simultanée, la variable `parallelChange` parcourra les mêmes pièces en les coloriant en orange sur l'interface graphique.

A noter que cet algorithme peut ne jamais trouver la sortie si un chemin du labyrinthe est une boucle.

### 3.3.5. Algorithme A\*

La classe `pieceAstar` héritant de `piece` permet d'ajouter à la classe `piece` des informations qui permette l'utilisation de cet algorithme, ces variables propres à chaque cases sont `g` le coût de déplacement minimum pour arriver jusqu'à cette case en partant du départ, `h` qui correspond à la distance heuristique de la case envisagée jusqu'à l'arrivée et enfin la variable `f` qui est la somme des deux dernières variables. Il y a aussi `parent` qui indique la pièce parcourue juste avant pour arriver jusqu'à cette pièce en empruntant le chemin le plus court.

La classe `astarResolve` hérite de `arbreLabyrinthe` qui contient l'algorithme à proprement parler. Tout d'abord on commence par repérer le départ et l'arrivée qui sont nécessaire pour le calcul des coûts de déplacement on crée deux `ArrayList`: `openList` et `closeList`.

`closeList` représente les cases qui ont déjà été visitées et dont seul le coût `g` et la case `parent` seront potentiellement recalculés si on trouve un chemin plus court pour cette case.

`openList` représente toutes les cases en contact direct avec des cases présentes dans `closeList` il s'agit des prochaines case envisagées pour être visités.

La boucle commence après avoir définie la variable `currently` qui correspond à la case sur laquelle on se trouve actuellement.

On attribue donc la case de départ à `currently` et ensuite la boucle commence: -on ajoute toutes les case mitoyennes à l'`openList`

- on choisit la case de l'openList avec le f le plus petit
- cette case a alors la case précédente comme parent si elle est mitoyenne à cette case et si le parent actuel représente un coût plus élevé
- cette case passe dans la closeList
- on ajoute toutes les case mitoyennes à l'openList
- ...

La boucle prend fin lorsque l'on arrive aux coordonnées de l'arrivée. Pour représenter le chemin on remonte alors les parents des cases parcourues.

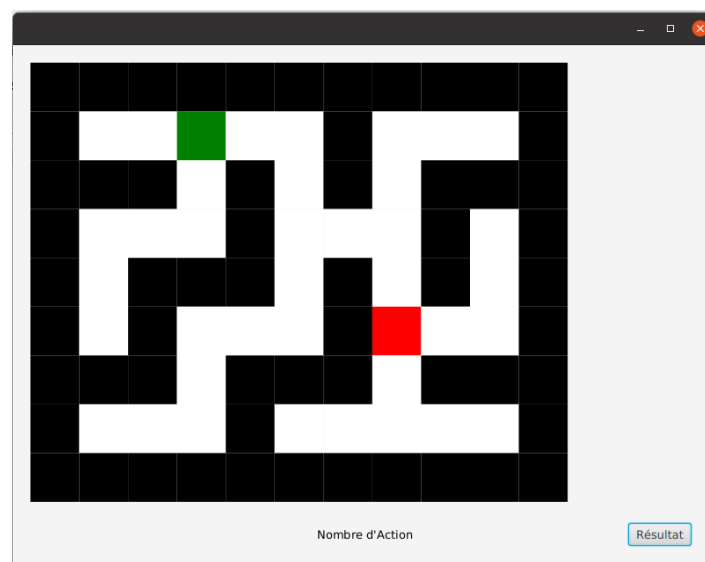
## 3.4. Tests

### 3.4.1. Test sur la console

Pour tester nos algorithmes et comprendre les erreurs, nous travaillons avec la console. Nous utilisons `system.out.println` pour comprendre comment le programme manipule les données. Pour se repérer, on inscrit le fichier et le numéro de ligne dans la sortie console. De plus, dans la sortie `default` de chaque `switch`, on inscrit un message d'erreur descriptif avec le nom du document.

### 3.4.2. Labyrinthe de test

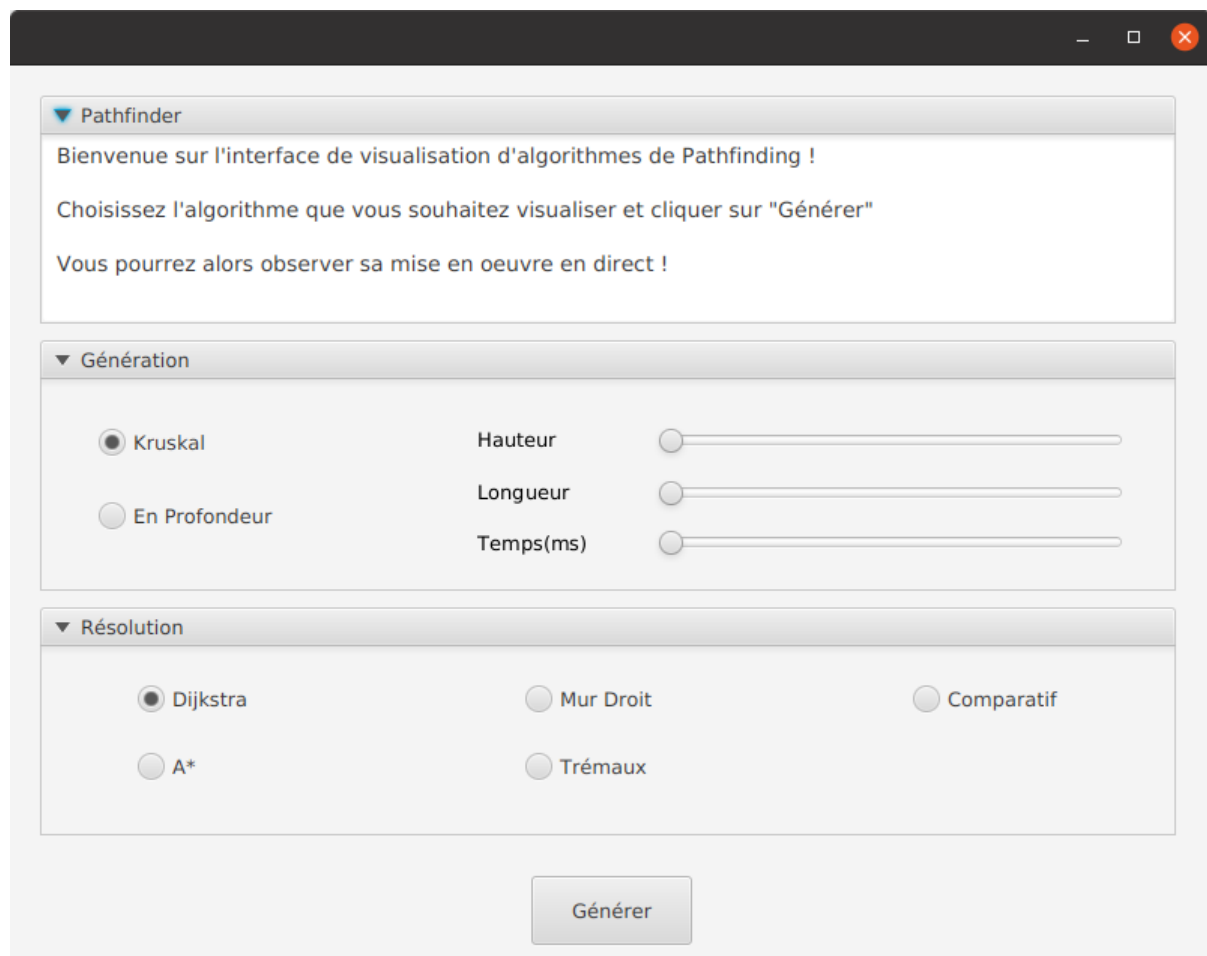
On a créé une classe `LabyTest` qui permet de faire marcher son algorithme de résolution sur un même labyrinthe immuable et connu. On peut ainsi choisir l'arrivée et la sortie, et mieux comprendre les problèmes car ils sont identiques d'une manipulation à l'autre, au hasard près. Voici une représentation de ce labyrinthe :



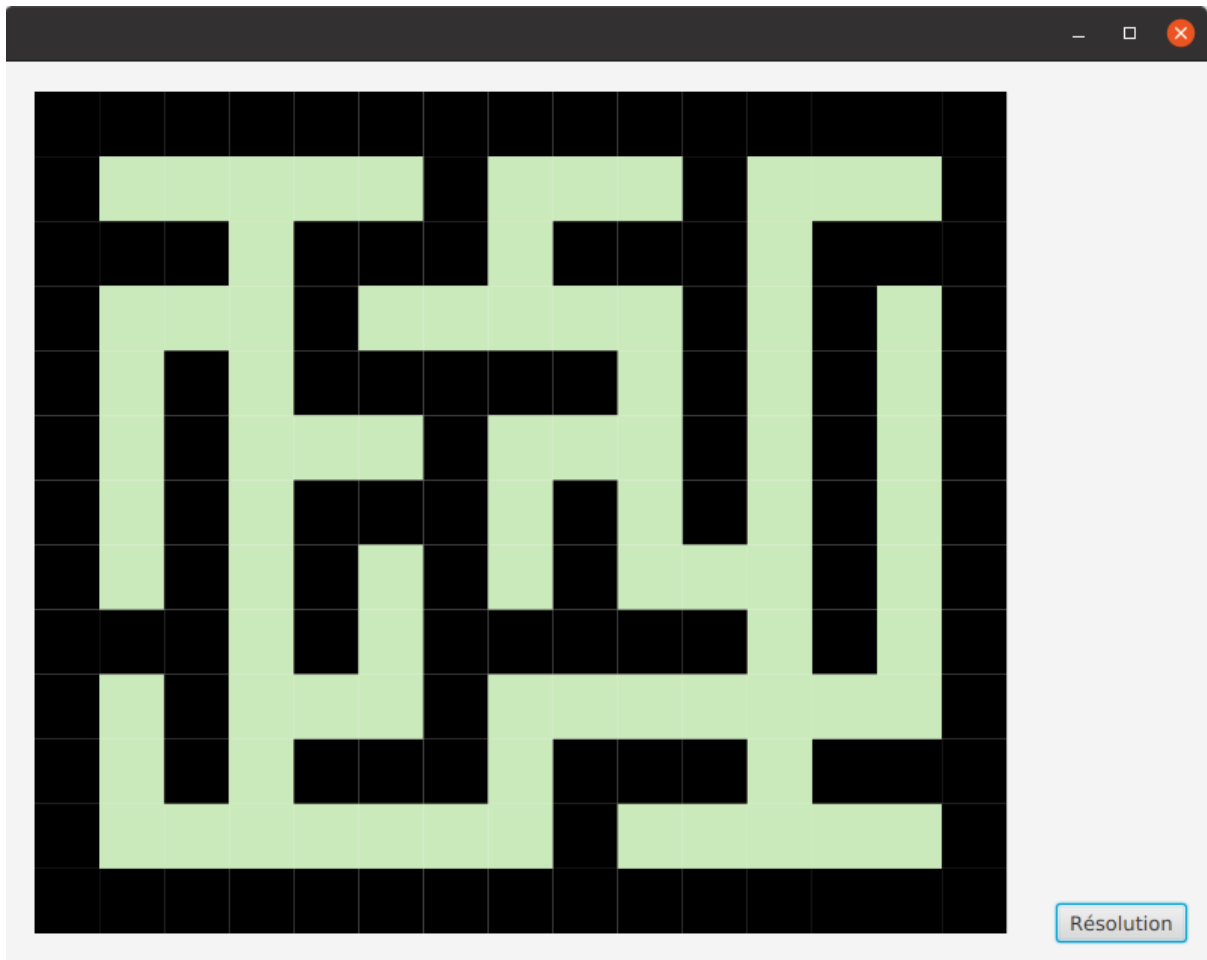
## 4. Manuel utilisateur

Une fois le programme lancé, on arrive sur la page d'accueil. Ici, vous sélectionnez les différents paramètres utiles à la génération et à la résolution. Ensuite cliquer sur Générer.

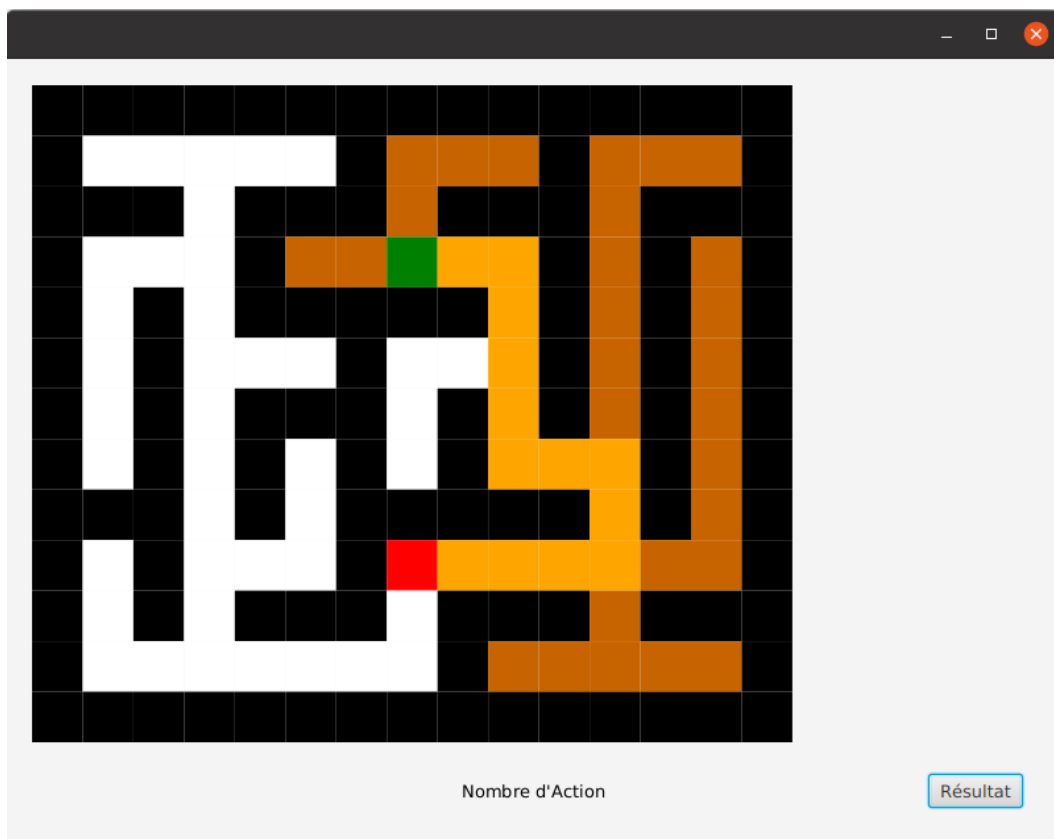
*Remarque* : Evitez de mettre des couples Hauteur Longueur tel que  $\text{Hauteur} \times \text{Longueur} > 900$ , sinon l'ordinateur est trop lent.

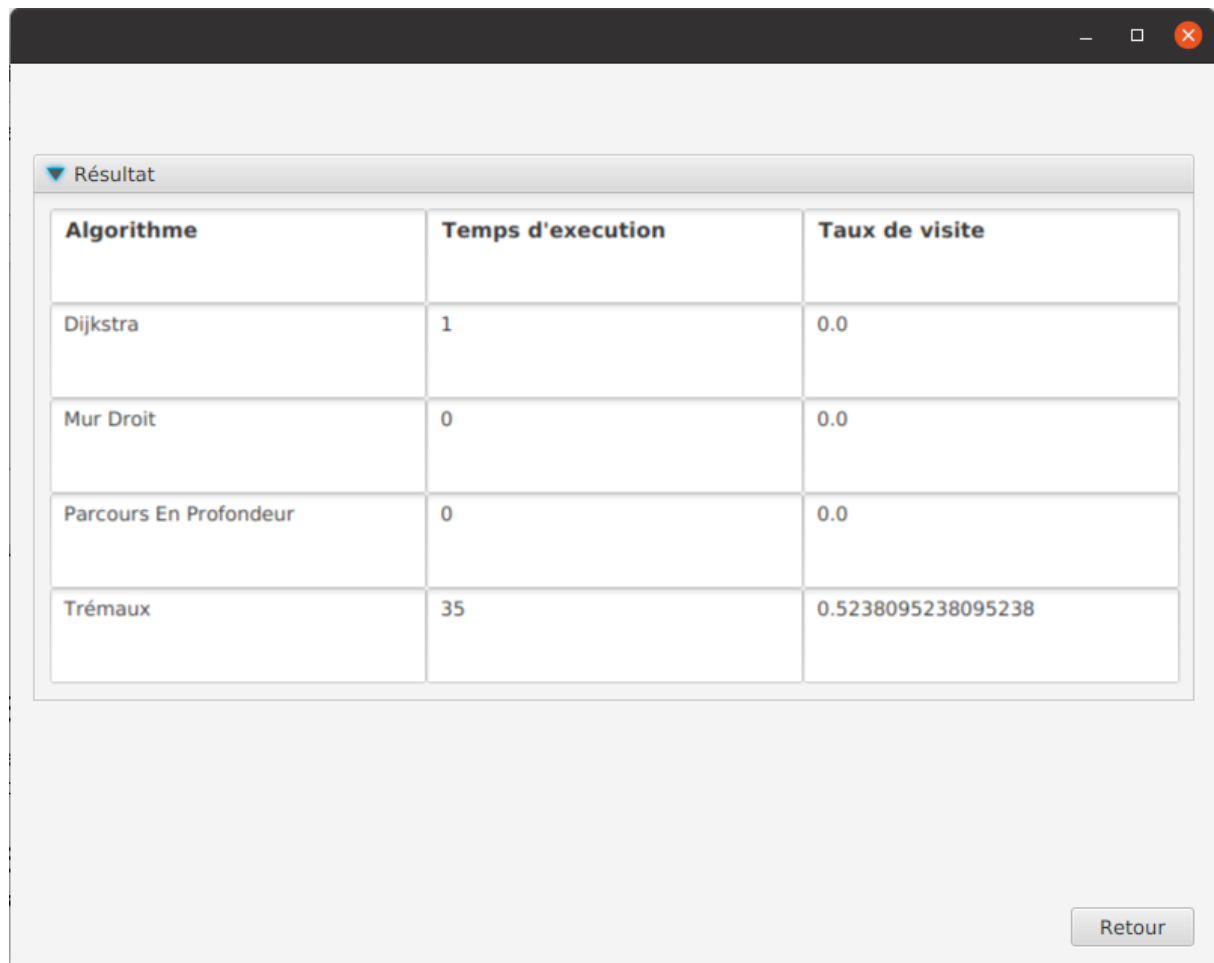


Vous arrivez sur la page de génération où l'animation commence immédiatement. Vous pouvez passer l'animation en cliquant directement sur résolution.



Vous arrivez sur la page de résolution (ici par Trémaux). L'animation se lance aussi directement et vous pouvez passer à la suite en cliquant sur "Résultat".





▼ Résultat		
Algorithme	Temps d'execution	Taux de visite
Dijkstra	1	0.0
Mur Droit	0	0.0
Parcours En Profondeur	0	0.0
Trémaux	35	0.5238095238095238

Retour

Une fois avoir comparé les résultats, vous pouvez revenir à la page d'accueil.



## 5. Travail de groupe

### 5.1. Projet Informatique

C'était pour nous tous le plus gros projet informatique auquel nous avons participé. De part notre expérience, nous n'avions pas une organisation bien définie de comment procéder pour mener à bien le projet. De plus, nous étions ambitieux et non conscients de la tâche que l'on devait fournir pour terminer l'application. Nous pensions un projet informatique comme en grande partie du codage pur alors que cette tâche ne représente que la partie finale du projet. En effet, nous avons vu l'importance de bien définir en amont les structures de données, les interactions entre classe, les problématiques abordées, le découpage en petites tâches ... Et ce d'autant plus en groupe car il est important que tout le monde commence à coder avec les mêmes normes pour l'intégrité et le bon fonctionnement du programme. Une autre partie des problèmes en projet d'informatique est le partage du code. Initié à Git cette année, il y a eu parfois des problèmes qui nous empêchaient de partager aux autres ce que l'on avait fait. La partie commentaire est elle aussi un point important qu'il ne faut pas négliger. On voit aussi qu'une partie optimisation en temps et en données aurait été utile. En effet, nos ordinateurs commencent à bloquer quand les grilles deviennent trop grandes.

### 5.2. Travailler en groupe

Ce projet nous a permis de travailler en groupe. Étant un projet en parallèle des autres cours sur une longue durée, il a fallu rester motivé et savoir respecter les différentes dates clés (Livrable 1, Livrable 2, Livrable 3). Nous avons remarqué que nous avons travaillé non pas de manière continue mais par des petites fulgurances. Pour la communication dans le groupe, nous utilisons Git, un groupe sur une messagerie instantanée et Discord ou BBB pour des réunions.

## 6. Conclusion

Pour conclure, nous avons pu, à travers ce projet, approfondir nos compétences techniques avec Java, Scenebuilder et eclipse en découvrant et appliquant des algorithmes classiques de pathfinding. La visualisation a permis de donner une autre dimension à notre code car nous avons dû réfléchir à une structure (`model.Case`, `model.Piece`, `model.arbreLabyrinthe`) permettant de représenter graphiquement nos variables. De plus, le travail en groupe nous a formé à l'emploi de git, nous avons pu améliorer nos relations humaines au travers de réunions régulières sur l'avancée du projet, que ce soit avec et sans le tuteur. Ce fut un challenge intéressant, de mener un projet de groupe sur une si longue durée avec un fort caractère technique et nous sommes satisfaits et fiers de ce que nous avons pu produire et surtout apprendre à travers celui-ci.

## 7. Annexes

[https://www.lama.univ-savoie.fr/mediawiki/index.php/G%C3%A9n%C3%A9ration\\_et\\_r%C3%A9solution\\_de\\_labyrinthes\\_II](https://www.lama.univ-savoie.fr/mediawiki/index.php/G%C3%A9n%C3%A9ration_et_r%C3%A9solution_de_labyrinthes_II)