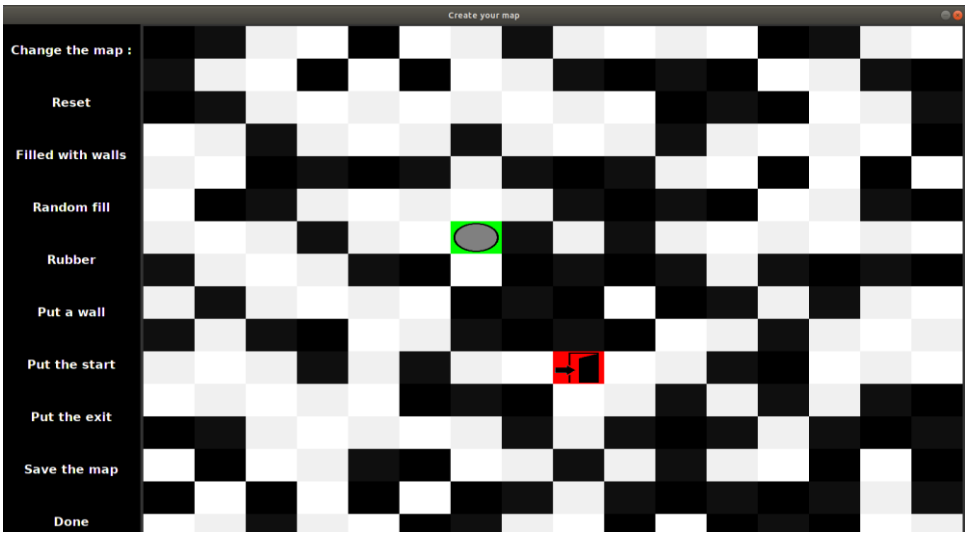
Projet Tutoré Langage Java

L'algorithme d'Arianne

Pierre Ribollet
Louka Doz

Table des matières :

Introduction



Pour le projet d'APL 2.1, l'objectif était de concevoir un algorithme de guidage. Il visait à faire déplacer un objet jusqu'à son but tout en évitant les obstacles.
Plusieurs fonctionnalités étaient demandées :

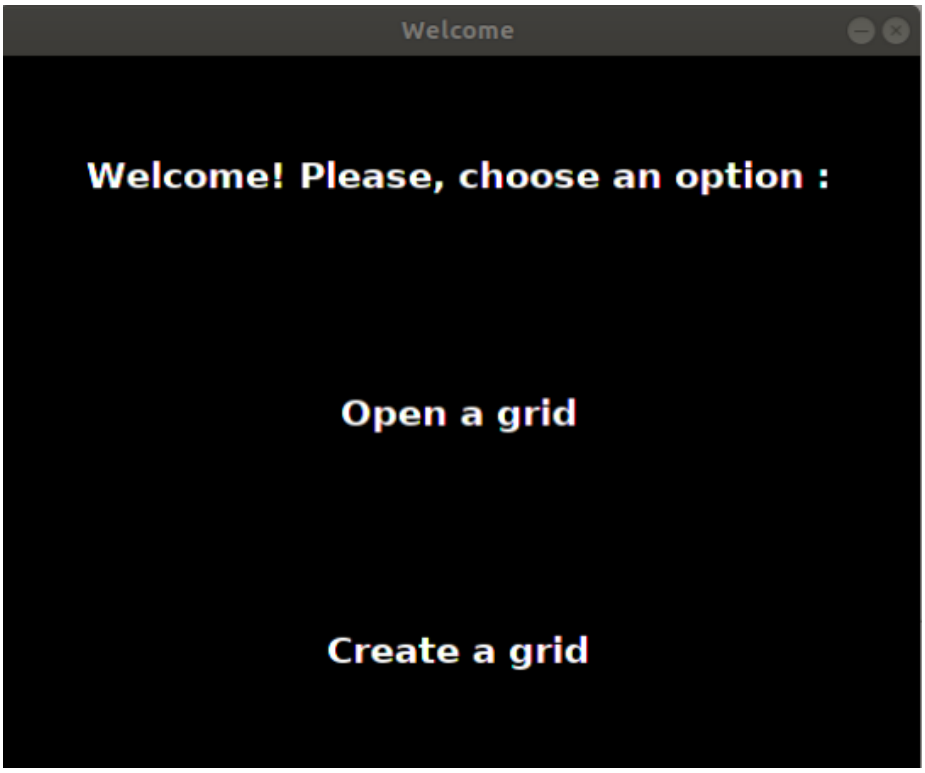
- faire deux algorithmes de « Pathfinding », un qui est totalement aléatoire, et l'autre déterministe : il se base sur les coordonnées actuelles de l'objet ainsi que la mémoire de ses actions précédentes. Il n'a aucune connaissance de la disposition de la carte et doit se repérer par ses propres moyens.
- 2 types de visualisation : manuelle et automatique. En mode automatique, la grille n'est pas affichée et ne demande aucune action de l'utilisateur. Le nombre d'étapes de l'objet sera affiché à la fin. En mode manuelle, la grille est affichée et l'action d'un utilisateur est requis. Celui-ci doit appuyer sur une touche (ici flèche droite) pour faire avancer la simulation. L'utilisateur

aura alors un visuel sur le comportement de notre algorithme.
- L'utilisateur pourra charger une carte déjà faite, ou en faire une lui-même pour la sauvegarder. Pour la création de celle-ci, l'utilisateur dispose d'un éditeur de grille pour pouvoir complètement la personnaliser. Il pourra choisir la taille de la grille, la localisation de l'entrée du labyrinthe et de sa sortie, ainsi que la possibilité de placer des murs et tracer des chemins.

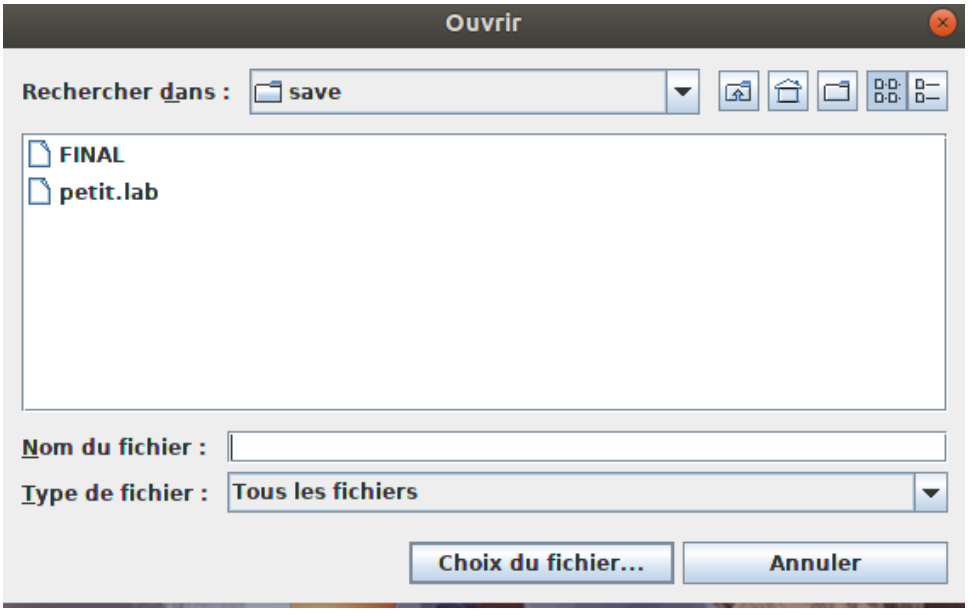
II) Partie gestion des fichiers

A. Ouverture du fichier

Dans le menu du début, l'utilisateur à deux choix :



Il peut charger une carte déjà faite avec l'option : « Open a grid », un gestionnaire s'ouvrira alors et lui permettra de choisir un fichier dans le dossier « save ».

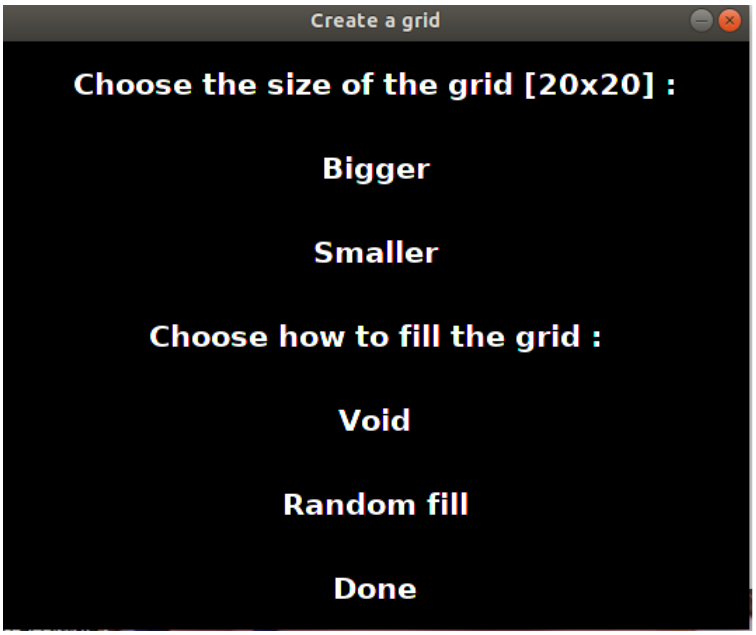


Le fichier se décompose de la forme suivante :

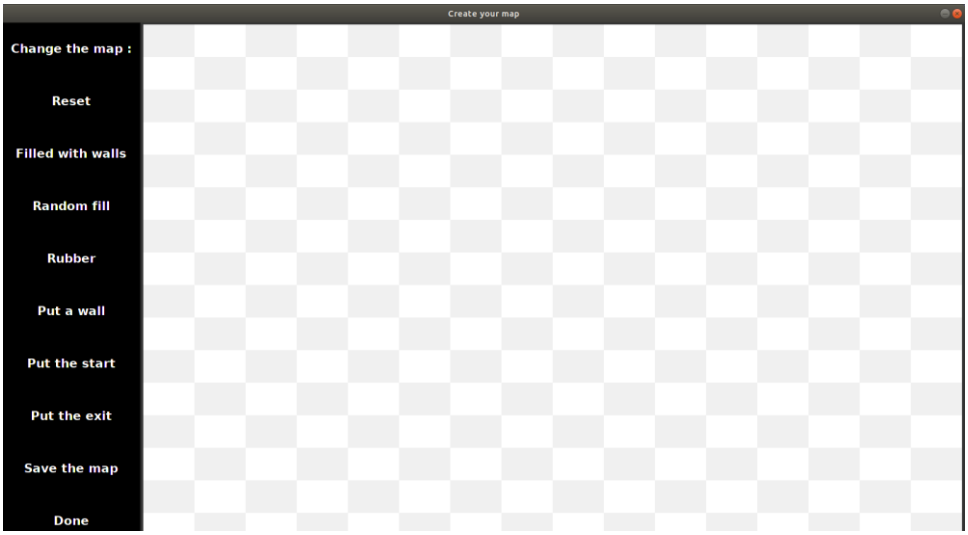
Les caractéristiques de la grille sont stockées sous forme d'octet. La première partie de ce fichier, sert à indiquer la taille de grille, la deuxième et troisième parties correspondent à la ligne et à la colonne où se trouve l'entrée du labyrinthe. La quatrième et cinquième parties correspondent à l'endroit où se situe la sortie de ce labyrinthe. Toute la fin du fichier représente les murs et les chemins de la grille colonne par colonne. Un 0 représente le chemin, alors que le 1 représente 1 mur. Une méthode nommée : « loadmap() » permet de décomposer ce fichier pour récupérer les valeurs citées plus haut. Il les stockera alors dans les attributs de sa classe et pourra être utilisé pour faire l'affichage d'une grille.

B. Sauvegarde des données

Avec l'option : « Create a grid », l'utilisateur sera dirigé vers l'éditeur de grille. Il peut ainsi, choisir la taille et le mode de remplissage de sa grille, totalement vide ou aléatoire.



A ce stade-là, l'utilisateur aura tous les outils à sa disposition pour créer sa grille personnalisée.

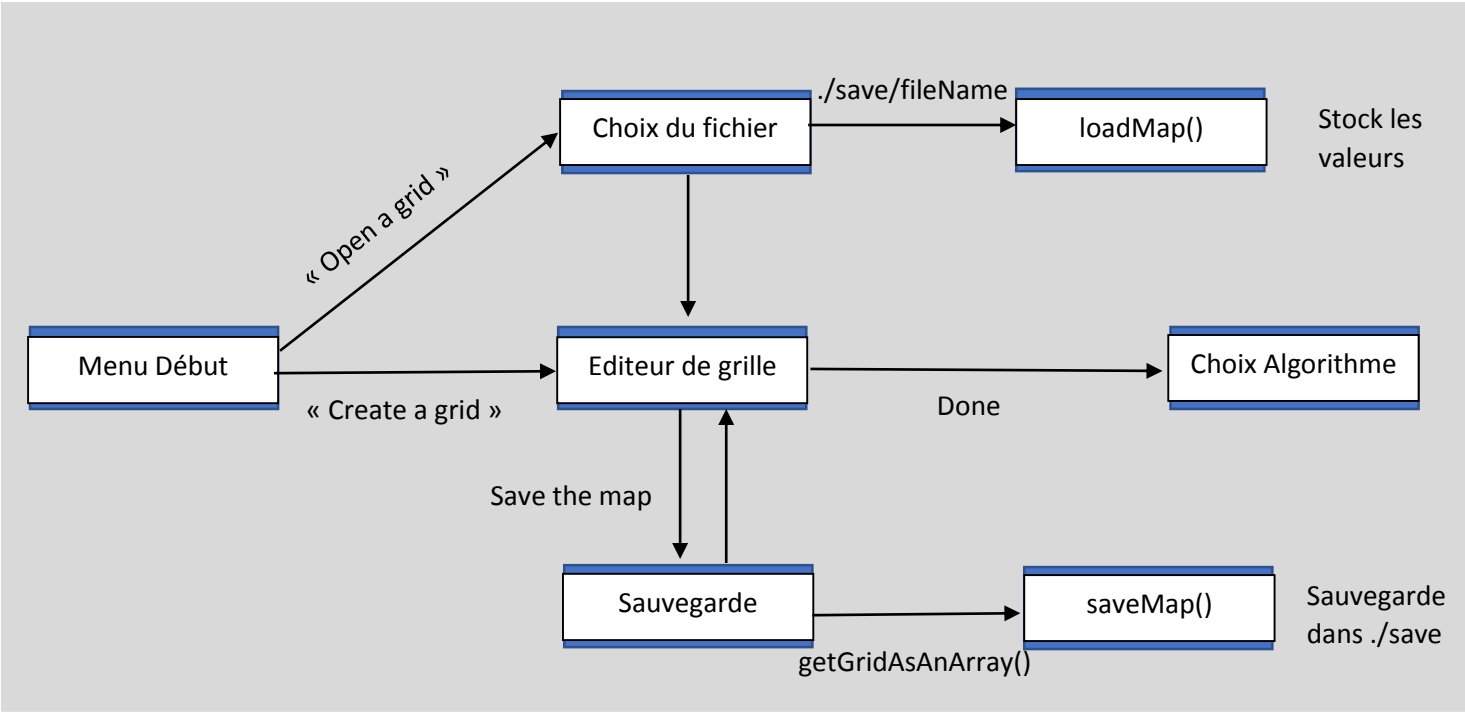


Diverses options s'offrent à lui :

- Reset : Permet de vider toute la grille
- Filled with walls : Rempli la grille de mur
- Random fill : Crée une grille aléatoire
- Rubber : Placer un chemin
- Put a wall : Placer un mur
- Put the start : Placer la sortie
- Put the exit : Placer la sortie
- Save the map : Sauvegarde la grille dans le fichier /save
- Done : Permet de finir la création de grille et accéder aux algorithmes

Une méthode nommée : « getGridAsAnArray() » permet de renvoyer la valeur de toutes les cases de la grille, 0 si c'est un chemin, 1 si c'est un mur, 2 si c'est l'entrée et enfin 3 si c'est la sortie. Cette méthode est invoquée lorsque l'utilisateur clique sur « Save the map », permettant ainsi de récupérer la grille sous forme d'un tableau d'entier, qui sera ensuite renvoyé vers la méthode « saveMap() » qui écrira toutes les caractéristiques de la grille sous forme de fichier vu précédemment.

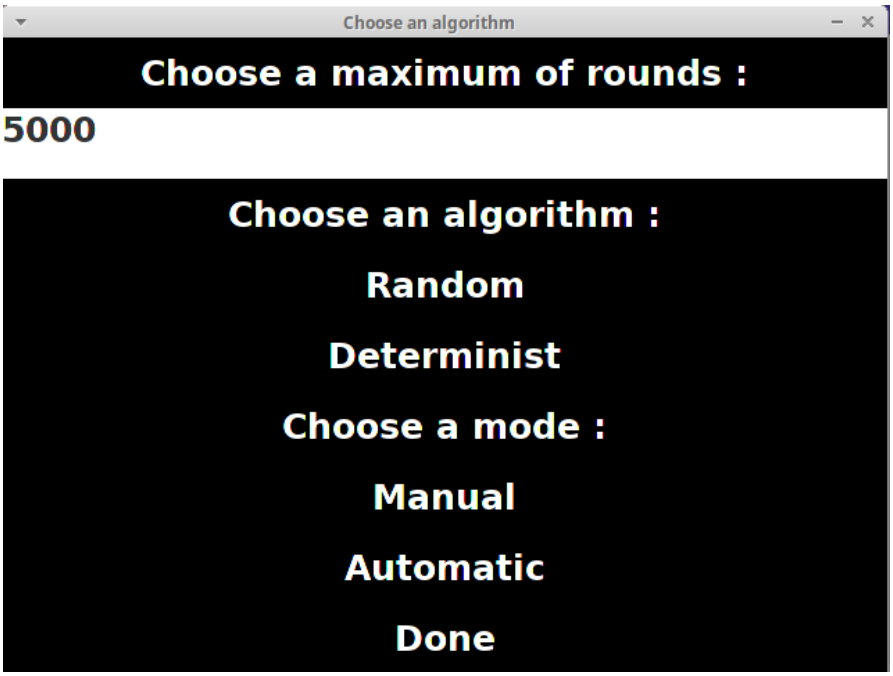
En résumé :


./save/fileName
Choix du fichier
loadMap()
Stock les valeurs
« Open a grid »
Menu Début
Editeur de grille
Choix Algorithme
« Create a grid »
Done
Save the map
Sauvegarde
saveMap()
Sauvegarde dans ./save
getGridAsAnArray()

III) Partie Algorithme

A. Déroulement de la simulation

Une fois le labyrinthe créé, il reste une dernière étape avant le lancement de la simulation :

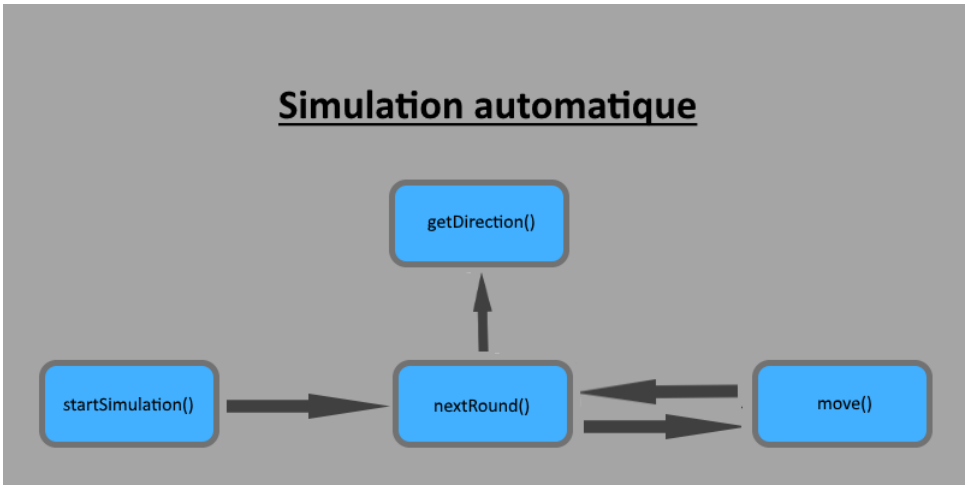


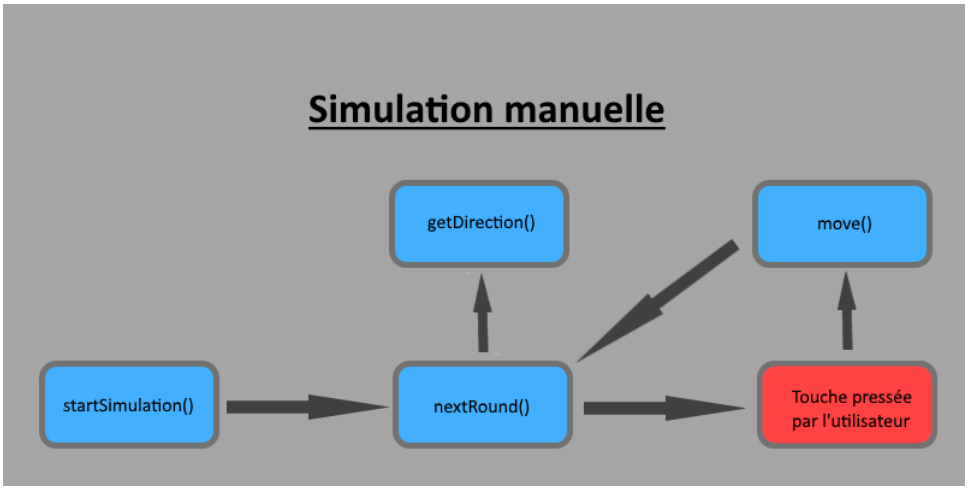
L'utilisateur doit choisir :

- Un nombre maximum de tours compris entre 2 et 10000 après lequel la simulation

- Entre un algorithme aléatoire, dont les déplacements seront aléatoires, et un algorithme déterministe, qui se basera sur ces coordonnées actuelles ainsi que sa mémoire pour trouver le plus rapidement et efficacement la sortie

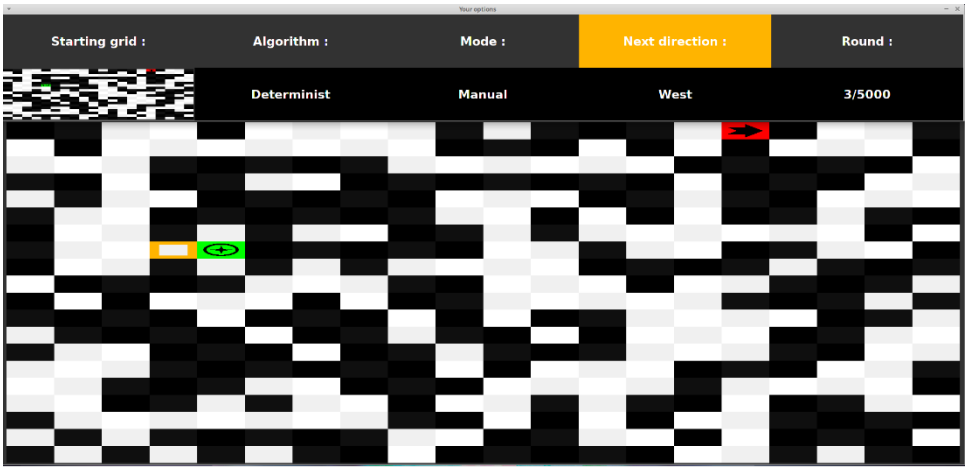
- Entre une simulation manuelle, durant laquelle l'utilisateur peut voir l'avancement de celle-ci et doit appuyer sur une touche pour faire avancer la simulation tour par tour, et une simulation automatique, durant laquelle l'utilisateur ne voit pas la simulation et ne connait que les résultats à la fin de la simulation (Sortie trouvée ou non, nombre de tours, moyenne des tours nécessaires pour trouver la sortie dans le cas d'un algorithme aléatoire)



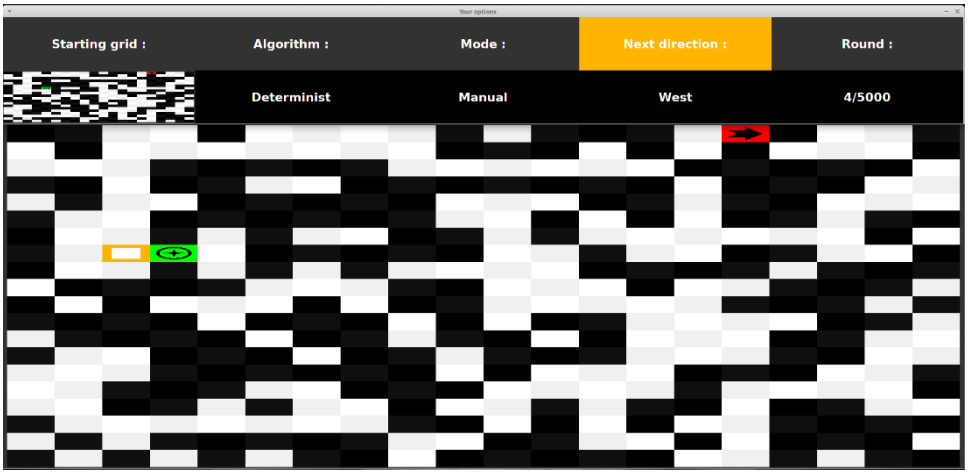


La simulation va tenter de trouver la sortie avant d'atteindre le nombre maximal de tours. Pour cela, à chaque tour, réalisé par la méthode nextRound(), elle détermine une nouvelle direction, avec getDirection(), vers laquelle aller. Une fois la direction obtenue, la méthode move() est appelée et elle à 3 actions à sa disposition :

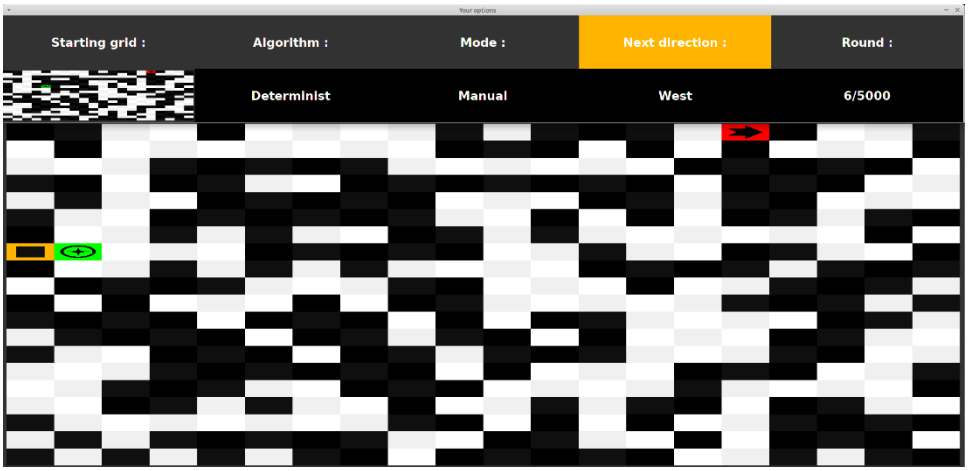
1/ Dans le cas où la prochaine case est vide...



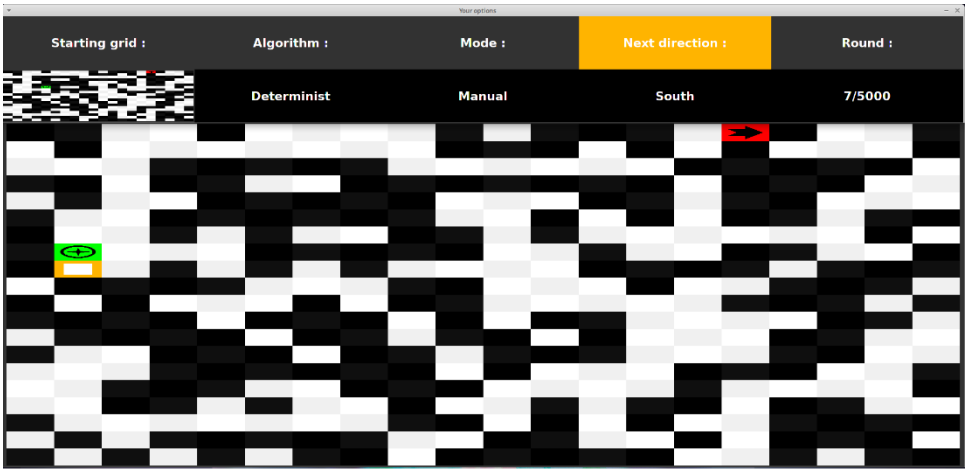
...elle s'y déplace.



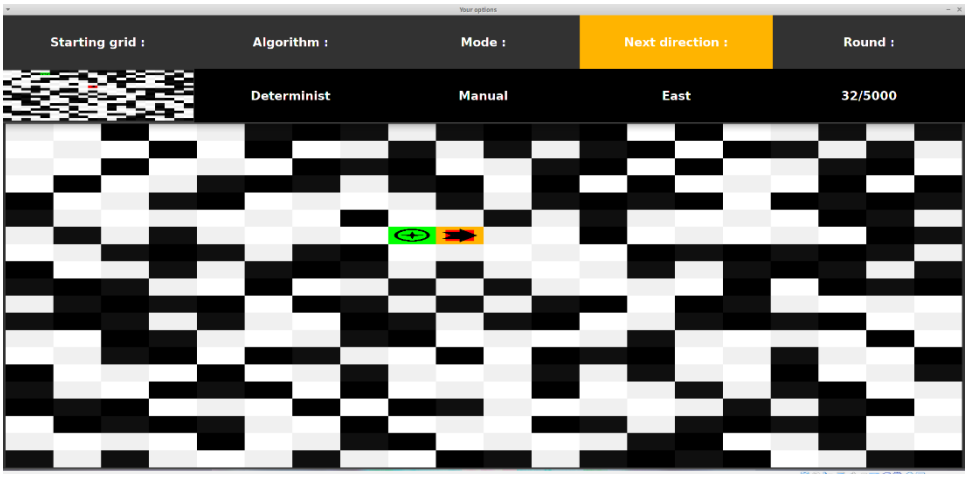
2/ Dans le cas où la prochaine case est un mur...



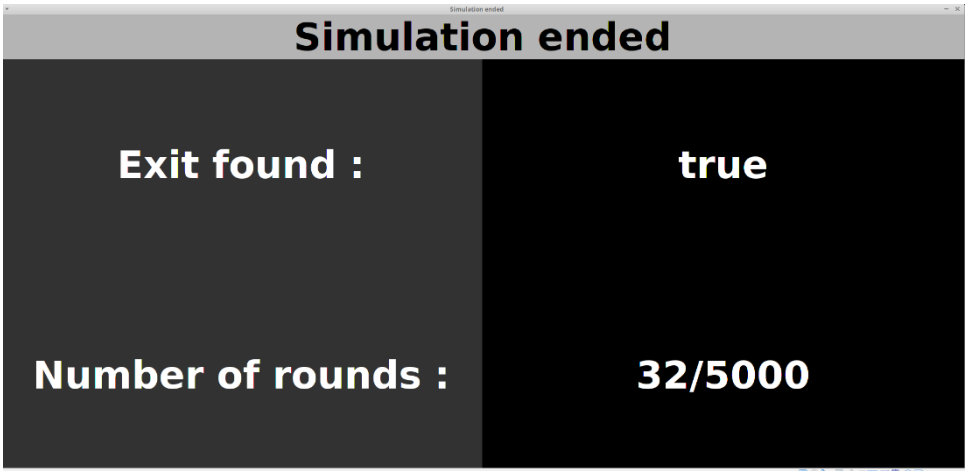
...elle ne se déplace pas mais l'action est tout de même comptabilisée.



3/ Enfin, dans le cas où la prochaine case est la sortie...



...elle s'y déplace et la simulation est terminée.



B. Algorithme aléatoire

L'algorithme aléatoire est assez simple. Lorsque la méthode getDirection() est appelée, l'algorithme aléatoire est demandé par la méthode getRandomDirection() qui va tirer aléatoirement un entier entre 0 et 3. 0 est le Nord, 1 est l'Est, 2 est le Sud et 3 est l'Ouest.
L'algorithme vérifie quand même que la prochaine direction ne dépasse pas les limites de la fenêtre. Si c'est le cas, il refait un tirage.

C. Algorithme déterministe

Le cas de l'algorithme déterministe est plus compliqué. Il est appelé par la méthode getDeterministDirection(). Il utilise deux tableaux :

- unknownPathsID : qui contient les ID, autrement dit, les numéros des cases que l'algorithme découvre avec la méthode lookAround(), expliquée plus tard

- knownPaths : qui contient "false" si la case n'a jamais été empruntée ou sinon "true" ("true" peut aussi signifier que c'est un mur)

L'algorithme déterministe fonctionne selon le principe d'une pile :

Lorsque la simulation se trouve sur une case qu'elle n'a jamais emprunté auparavant, elle appelle la méthode lookAround() qui à 2 fonctions:

- Tout d'abord, elle vérifie si la position actuelle de la simulation est juste à côté de la sortie. Si oui, getDeterministDirection() renvoie directement la direction vers laquelle se trouve la sortie.

- Si non, elle regarde pour chaque direction (Nord,Est,Sud,Ouest) : si la simulation peut s'y rendre sans sortir de la limite de la fenêtre, si la case dans cette direction n'est pas notée "true", c'est-à-dire qu'elle n'a pas été empruntée ou que ce n'est pas un mur. Si toutes ces conditions sont réunies, alors, avec la métode newPath(), elle ajoute à la fin de la pile : l'ID de la position actuelle suivie de l'ID de la case concerné par la direction

- Une fois ce travail terminé, elle finit en indiquant que la position actuelle est maintenant un chemin emprunté. La case de la position actuelle est notée "true". L'algorithme passe alors à l'étape suivante.

Cependant, si la simulation se trouve sur une case qu'elle a déjà emprunté, alors, il n'y pas nécessité à utiliser lookAround() et l'algorithme passe immédiatement à l'étape suivante.

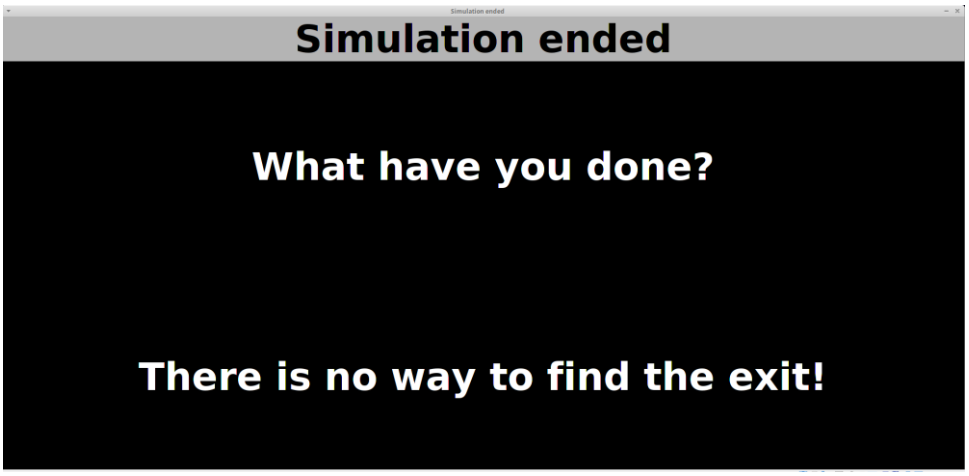
L'étape suivante consiste à utiliser les valeurs de la pile pour trouver une nouvelle direction non empruntée. La méthode getLastIDNoted() va renvoyer le numéro de case le plus récent de la pile, c'est-à-dire celui rentré en dernier, puis elle le supprime de la pile.
En clair, si la méthode lookAround() a été appelée et qu'elle a trouvé de nouvelles directions vers laquelle la simulation pourrait aller, alors, getLastIDNoted() va renvoyer une de ces directions.
Mais si lookAround() n'a pas trouvé de nouveau chemin ou que la méthode n'a juste pas été appelée, alors il y a 2 possibilités :

- La direction précédente était un mur, la simulation ne s'est donc pas déplacée, donc getLastIDNoted() va renvoyer le même numéro de case que celui de la position actuelle. Or, il n'est pas possible de se déplacer vers la même position, donc getLastIDNoted() est appelée une deuxième fois.

- Toutes les directions autour de la position actuelle sont déjà empruntée ou sont des murs, getLastIDNoted() va alors renvoyer le numéro d'une case par laquelle la simulation est déjà passée jusqu'à tomber sur une case non explorée. La simulation revient sur ces pas jusqu'à tomber sur un nouveau chemin.

C'est ainsi que l'algorithme déterministe fini toujours par trouver la sortie. Toujours ? Non ! Une seule condition peut empêcher l'algorithme déterministe de trouver la sortie : si l'utilisateur a créé un labyrinthe dans lequel aucun chemin ne relie l'entrée du labyrinthe à la sortie.
Mais dans ce cas précis, l'algorithme à une solution ! Il est capable de comprendre que si la pile est vide, c'est qu'il a testé toutes les cases possibles auxquelles il pouvait accéder sans que l'une d'entre-elles soit la

sortie. Il précise donc à l'utilisateur que la sortie ne pourra jamais être trouvée.



III) Conclusion

Pierre RIBOLLET :

Ce projet m'a permis de comprendre certaines notions qui étaient encore floues pour moi, notamment sur la manipulation des fichiers et des octets. Je suis satisfait de notre travail dans l'ensemble. Celui-ci m'a donné envie de me lancer dans d'autres projets pour consolider mes acquis ainsi qu'étendre mes connaissances.

Louka DOZ :

La plupart des choses que j'ai réalisées dans ce projet m'ont permis de consolider mes bases en java mais j'ai tout de même appris des choses. C'est un langage que j'apprécie et j'ai donc trouvé ce projet plus plaisant à réaliser que le premier.
Je n'ai pas eu de grosses difficultés dans ce programme si ce n'est de sans cesse tenter d'optimiser le code. Une des difficultés que je peux noter, est le fait de relier un tableau à une dimension contenant les types en colonnes par colonnes (enregistré ainsi dans les fichiers) et le fait que les Panels de la grille soient rangés en ligne par ligne. Je pense que trouver la solution mathématique à ce genre de problème me sera utile pour le futur.
Ce que j'ai réellement apprécié, c'est clairement résoudre l'algorithme déterministe. C'est très satisfaisant de trouver la solution. Je pourrai regarder pendant des heures l'algorithme déterministe résoudre n'importe quel labyrinthe. C'est impressionnant de voir à quel point le monde de l'informatique est illimité.
Je suis certain que ce genre de projet apporte beaucoup d'expérience et c'est pour cela que je vais continuer mon projet en java en cours. Plus je réalise ces projets, plus je me conforte dans l'idée que je souhaite programmer des logiciels, jeux, intelligences artificielles dans ma vie professionnelle.
J'attends avec impatience de voir ce qu'un projet avec un grand groupe peut donner.