Rendu Ecrit

Castel Aurélien
Alex Maizeroi

Projet tutoré n°2

Sommaire

I) Introduction

a) Introduction
b) Le transfert de données entre les différents classes (carrearray[])

II) Les menus et choix utilisateurs

a) Frame.java
b) Les différents menus et choix

III) Labyrinthe.java

IV) Grille personnalisée et aléatoire

a) Size.java et Grid.java
b) Carre.java

V) L'algorithme

a) Algorithme.java
b) Algorithme déterministe
c) Algorithme aléatoire

VI) Conclusions personnelles

I) Introduction

Desciption du sujet

L'algorithme d'Ariane est un algorithme de guidage qui a pour but de guider le personnage Thésée depuis une entrée jusqu'à la sortie d'un labyrinthe.
On représentera le labyrinthe sous forme d'une grille carrée. Avec des cases blanches qui représenteront les passages que peut emprunter Thésée et des cases noires qui représenteront les murs du labyrinthe.
Les positions initiales de Thésée (ou entrée) et de la sortie seront placées sur n'importe quelles cases libres (blanches).
À chaque étape d'une simulation, Thésée peut se déplacer soit à la case au nord, soit au sud, soit à l'est, soit l'ouest.
Ce mouvement peut avoir différentes conséquences :

- Si la case est bloquée Thésée ne change pas de case.
- Si est libre alors Thésée peut s'y rendre
- Si la case est la sortie, la simulation est terminée

Fonctionnalités demandées

L'utilisateur peut choisir de charger un grille grâce à un fichier existant (selon un certain format) ou construire sa grille.
S'il construit sa grille il pourra choisir la taille de la grille, la remplir aléatoirement et manuellement. Il peut choisir la position de l'entrée et de la sortie.
L'utilisateur pourra choisir entre deux algorithmes : déterministe ou aléatoire.
Dans l'algorithme déterministe, Thésée devra baser ses décisions selon ses coordonnées actuelles et ses actions précédentes. Il n'aura pas la connaissance de la sortie ou des cases noires.
Dans l'algorithme aléatoire, Thésée aura autant de chance de tirer chacune des directions mais ne pourra pas baser ses mouvements selon ses actions précédentes.
L'utilisateur aura aussi deux modes de visualisation de l'algorithme : manuel ou automatique.
Dans le mode manuel on représentera chaque étape de Thésée. L'utilisateur appuiera sur une touche défiler les étapes.
Dans le mode automatique fera seulement apparaitre ne nombre d'étape nécessaire pour finir la simulation. Pour l'algorithme aléatoire on fera tourner la simulation 100 fois et

afficheront le résultat moyen d'étapes.
Les performances de l'algorithme déterministe devront être meilleures que celles de l'aléatoire en terme de nombre d'étapes.
L'ensemble des interactions avec l'utilisateur ne seront que graphiques.

Explication découpage programme et schémas

Notre projet est constitué de 11 classes soit 10 .java.

Le premier Main.java contient la classe Main et Frame.
La classe Main appelle un objet Frame.
Frame appelle les autres classes extends JPanel pour les ajouter dans la fenêtre. Elle est constituée de bouton, qui ajoutent les JPanel grâce au card cardLayout.
Le choix de l'utilisateur est modifié depuis les données des objets. Ces données permettent notamment d'activer ou désactiver les boutons du cardLayout.
Le projet a été entièrement réalisé à l'intérieur d'une seule fenêtre, pour que le projet puisse être utilisé facilement par l'utilisateur, et plus simple à coder et à mettre à jour. Car tous les composants graphiques une fois finis d'être appelés sont montrées (shows) avec les boutons de Frame, on passe à un autre objet et on grise les anciens boutons que l'utilisateur n'a plus besoins d'utiliser.
La classe Menu est le premier composant graphique qui est montré à l'utilisateur et lui permet de choisir s'il veut créer sa grille ou bien la choisir depuis un fichier choisi.
S'il choisit de créer sa grille alors des objets Size (pour modifier la taille de la Grille) et Grid sont montrés et servent à créer sa grille personnalisée ou bien aléatoire.
La plupart des boutons dans le projet sont sous formes de classes anonymes car ils ne sont pas réutilisables : on les utilise pour modifier des composants graphiques particuliers.
Les classes anonymes utilisées n'ont été utilisées que dans ces cas-là, pour éviter de créer des classes à part faisant très peu de lignes de code.

Les données sortant d'un objet Grid sont la taille de la grille, la sortie et l'entrée et un tableau de Carre.
S'il choisit d'utiliser un fichier l'utilisateur le choisira grâce à un JFileChooser et les données sortant de Menu seront données à un objet Labyrinthe.
Labyrinthe permet de lire depuis un fichier (File) et retourne les mêmes types de données que

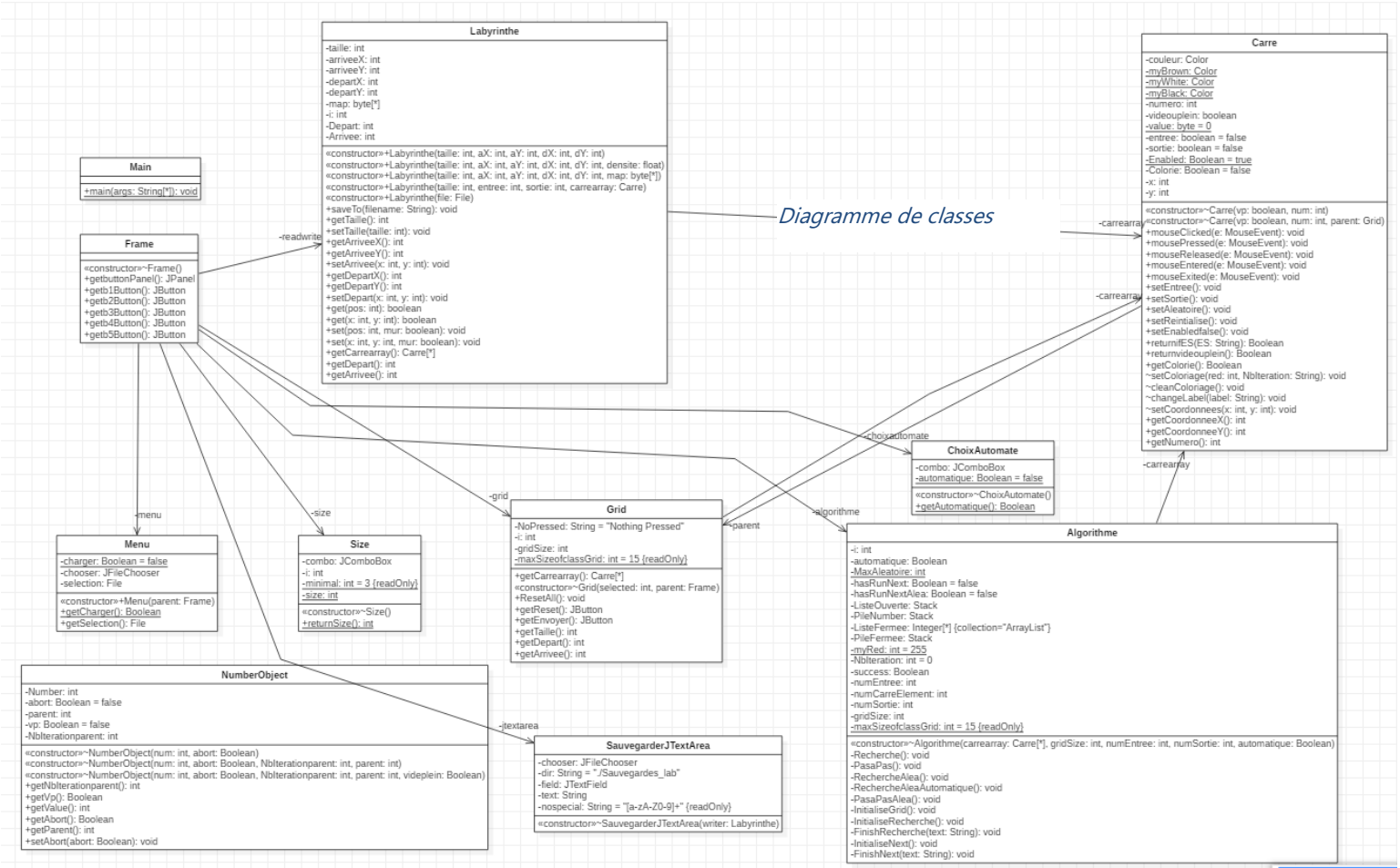
Grid.

Le transfert de données entre les différents classes (carrearray[])

Pour réunir le même type de données depuis Grid ou bien Labyrinthe pour l'Algorithme on utilisera un tableau de Carre, un ensemble d'objet qui contient toutes les données de chaque carrés du tableau. Comme par exemple si un carré est vide ou plein, est une sortie ou une entrée. Ces données qui définissent un Carre peuvent être modifiées seulement dans Grid grâce à un Mouselistener. Ou grâce à des méthodes setters qui sont utilisées pour faire des Carre aléatoirement (dans Grid), ou créer des Carre dans Labyrinthe.

Grâce aux données renvoyées par Grid, l'utilisateur peut sauver sa grille personnalisée. SauverJTextArea permet de choisir l'emplacement et le nom du fichier qu'il souhaite créer (tout en prenant en compte le caractère spéciaux).
Depuis Frame on va créer un objet différent (par rapport à la lecture) de type Labyrinthe depuis les données de Grid. Et Labyrinthe s'occupera de sauver sous forme de fichier la grille suivant un emplacement chemin que SauverJTextArea aura renvoyé.

L'utilisateur peut choisir s'il veut l'Algorithme automatique (sans composant graphique et avec l'aléatoire x 100) ou non grâce à la classe ChoixAutomate.



II) Les menus et choix utilisateurs

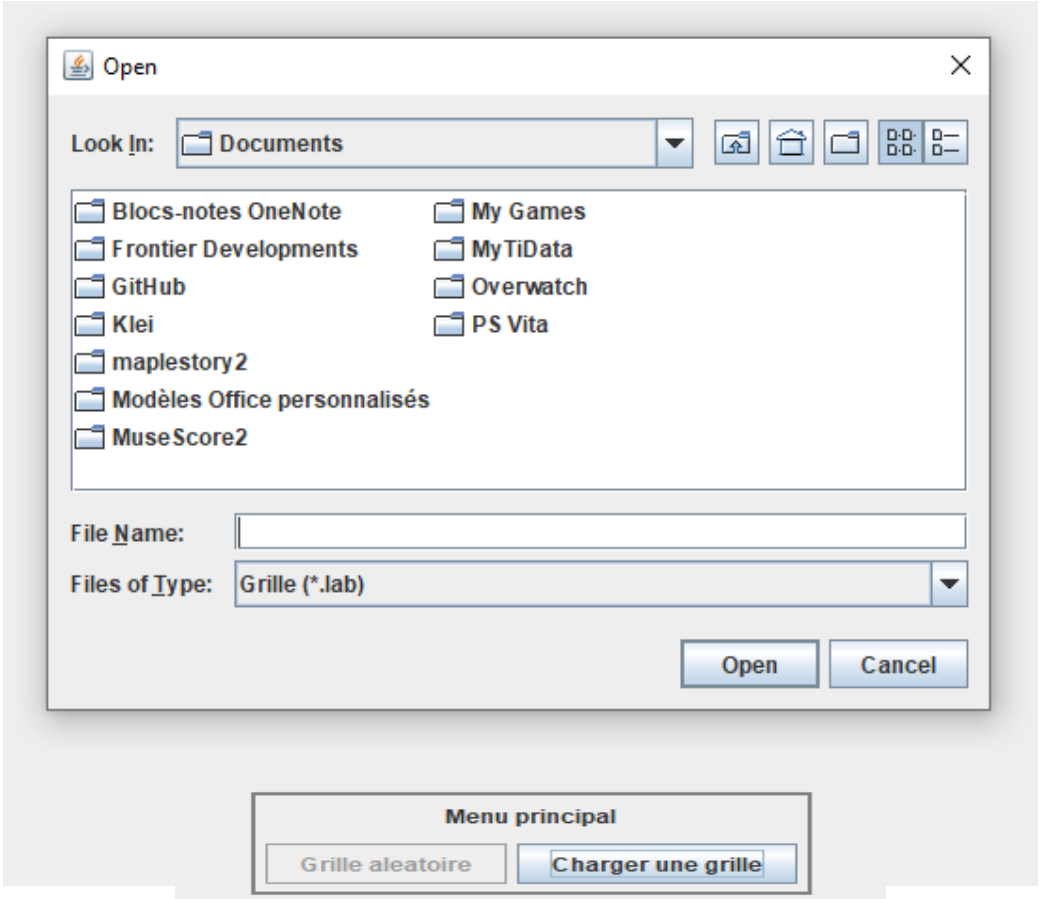
Au commencement Frame initialise des objets des classes qui ont besoins d'être affichées.

Et en premier un objet Menu est montré qui permet à l'utilisateur de choir ce qu'il veut faire.

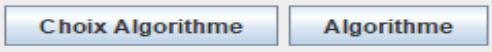
Les données renvoyées sont les attributs charger pour savoir si l'utilisateur veut utiliser un fichier ou non. L'attribut selection qui récupère le fichier sélectionné par chooser, un JFileChooser.

Au début : lorque Frame affiche le Menu.

Ici charger est égal à false et depuis la classe Menu grâce au paramètre parent on peut modifier Frame et afficher les boutons correspondant au choix de l'utilisateur.



Ici charger est égal à true et on demande ce que l'utilisateur veut donner comme fichier.



L'attribut selection que Frame a donné pour Labyrinthe a été accepté et l'utilisateur peut poursuivre le programme. Comme charger est égal à true on n'affiche pas les boutons Taille et Tableau personalisé dans Frame.

III) Labyrinthe.java

Cette classe a été conçue pour répondre à des besoins multiples c'est pourquoi elle gère de nombreux cas d'usage. C'est également pour ça qu'il y a plusieurs constructeurs qui ne sont pas tous utilisée dans ce projet mais elles pourront être utilisée dans de possible ajout de fonctionnalité par exemple. La grande majorité des attributs de cette classe sont des int, puisqu'il n'y a pas eu de besoin spécifique au niveau de cette aspect. Elle gère également l'utilisation par deux types de tableaux au niveau de la map c'est-à-dire par la transmission d'un tableau a une dimension ou a deux selon ce que préfère la personne qui utilise la classe ici nous n'utiliserons que celui a une dimension. Cette classe comprend aussi la lecture ainsi que l'écriture. Pour la lecture c'est un constructeur qui gère tout ça en prenant simplement comme argument un File. Le plus dur aura été d'associer les bonnes infos par rapport à la fin du fichier. Un tableau de boolean qui indique la présence ou non d'un obstacle il a juste été nécessaire de faire des tests notamment des & bit a bit. Pour l'écriture c'était finalement symétrique ce qui fais que cela se ressemble énormément. Il faut noter aussi une méthode pour adapter la donnée au sens de parcours de la grille.

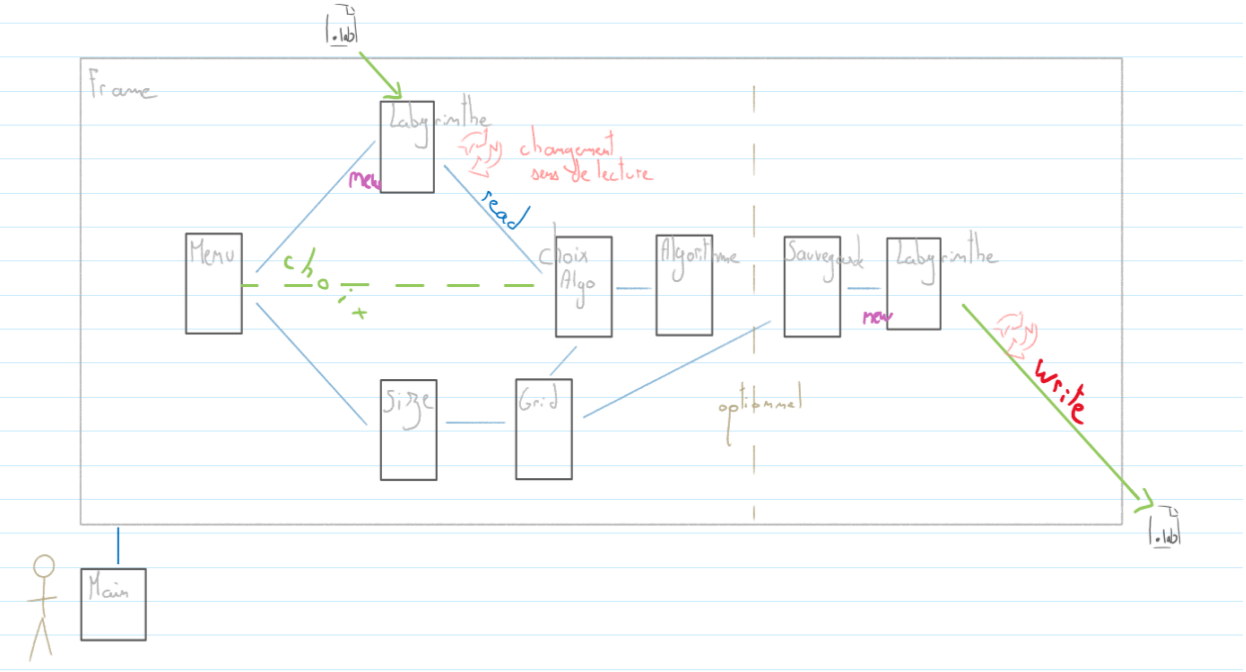
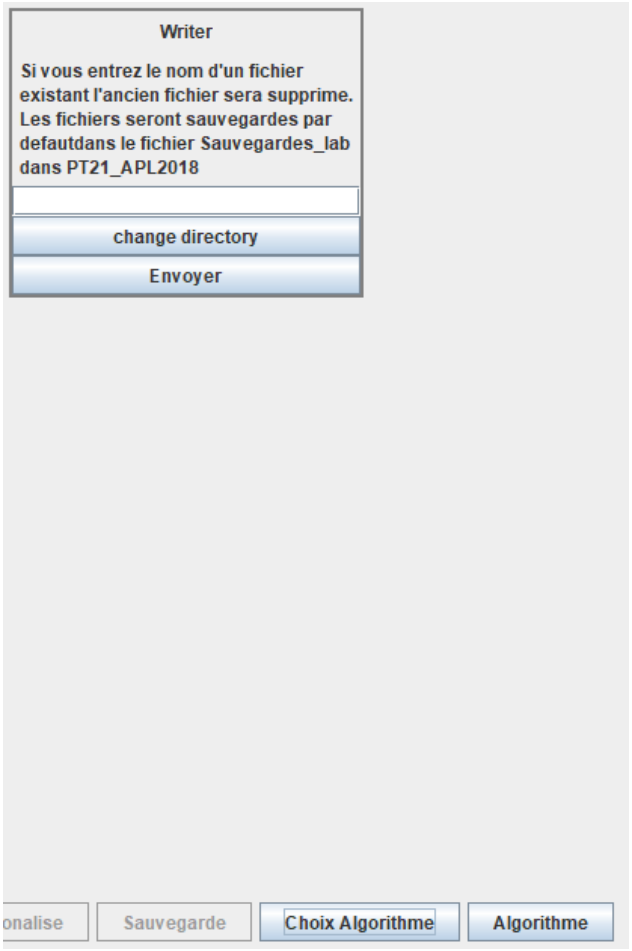


Schéma de classe des fichiers de notre projet et leurs transferts de données. Les flèches rouges montrent où l'on change le sens de lecture pour que la lecture et la sauvegarde fonctionne.

SauvegarderJTextArea



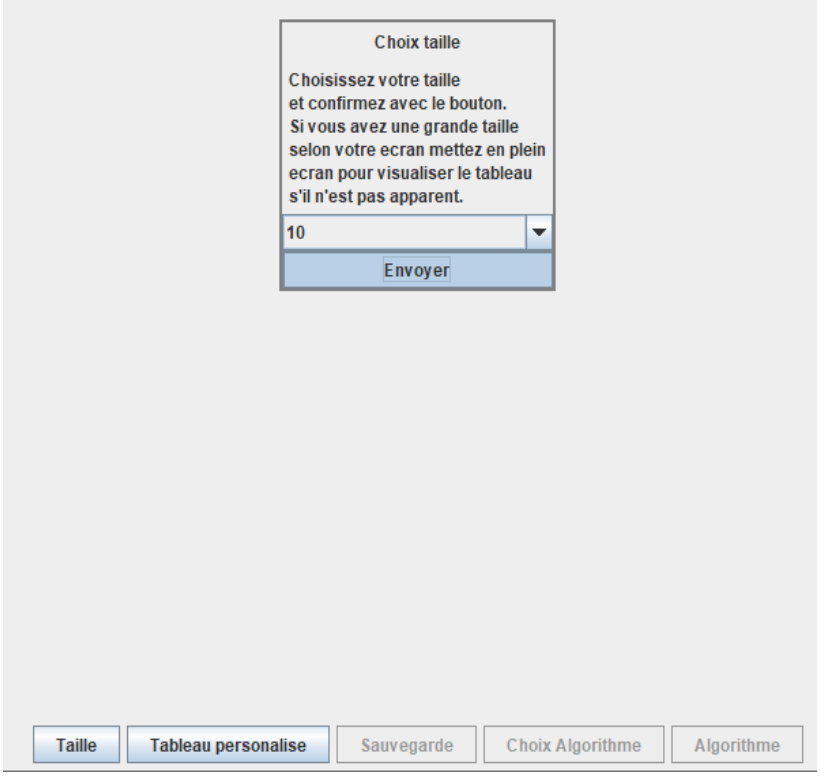
Ce JPanel permet d'afficher un JTextArea pour choisir un nom de fichier et un JFileChooser permet de choisir l'emplacement de son fichier.
Les données de Grid ont permis de créer un objet Labyrinthe pour sauvegarder. Grâce à SauverJTextArea on peut savoir un emplacement pour ce fichier.
Puis on créé le fichier grâce à la méthode saveTo() de Labyrinthe.

IV) Grille personnalisée et aléatoire

Dans le cas où charger est égal à false dans Frame alors elle affichera les boutons Taille et Tableau Personnalisé.

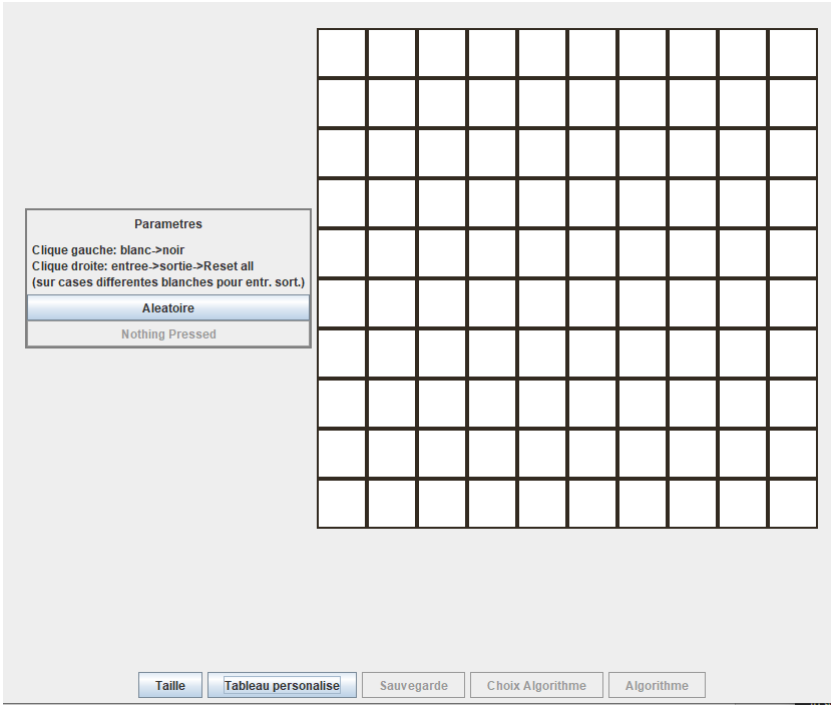
Size.java

Le bouton qui montre la taille affiche un objet Size.



L'utilisateur a appuyé sur Taille et Frame lui a affiché un objet Size. Il a entré une taille de 10 que récupère l'attribut size. Frame créé ensuite un objet Grid qui prend en argument une méthode de Size qui permet de récupérer l'attribut size.

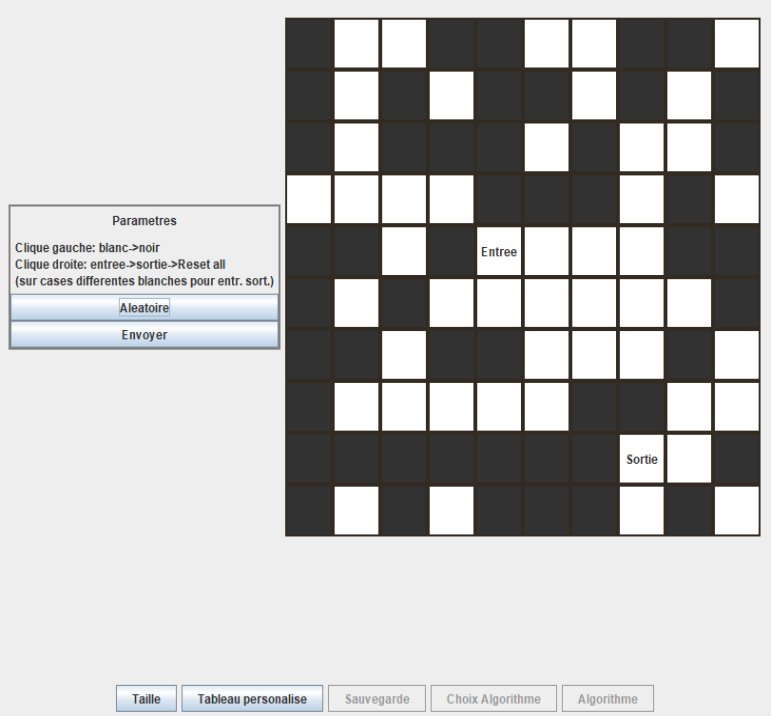
Grid.java



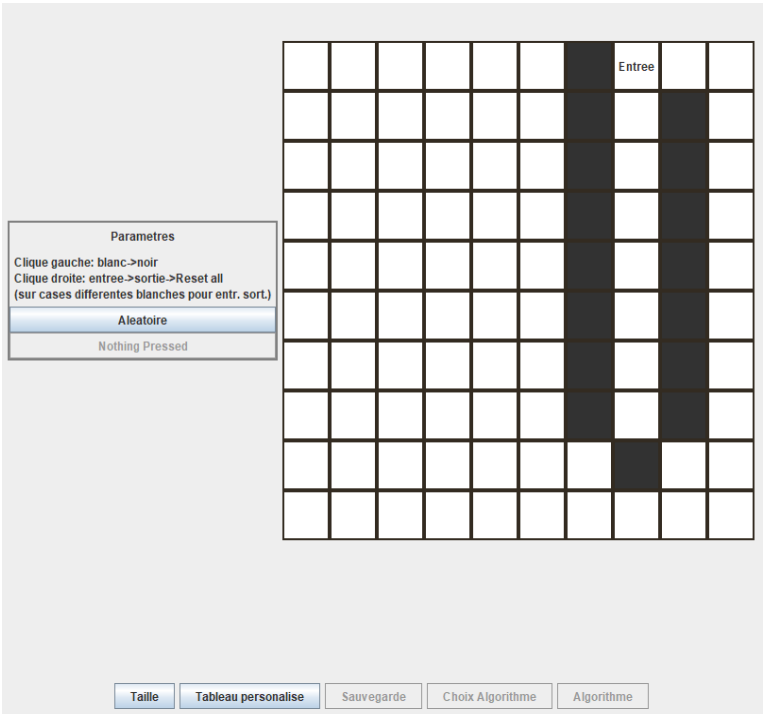
Une fois l'objet Grid initialisé selon la taille de l'utilisateur, celui-ci a plusieurs choix pour construire sa grille.
Chaque objet de type Carre sont affichés à l'intérieur d'un Grid Layout et sauvegardés dans un tableau. Chacun a un mouseListener qui permet de modifier les objets stockés.
Si l'utilisateur clique gauche sur un carré, il deviendra noir et un mur s'il reclique dessus il redeviendra vide et blanc.
Si l'utilisateur clique droit sur un carré vide il deviendra une entrée, s'il clique sur un autre carré blanc celui-ci sera la sortie.
S'il clique une troisième fois avec le clique droit toute la grille sera réinitialisée.

La grille est acceptée seulement si l'utilisateur n'a fait que deux cliques droit acceptés, si il y a une entrée et une sortie grâce à un switch case.

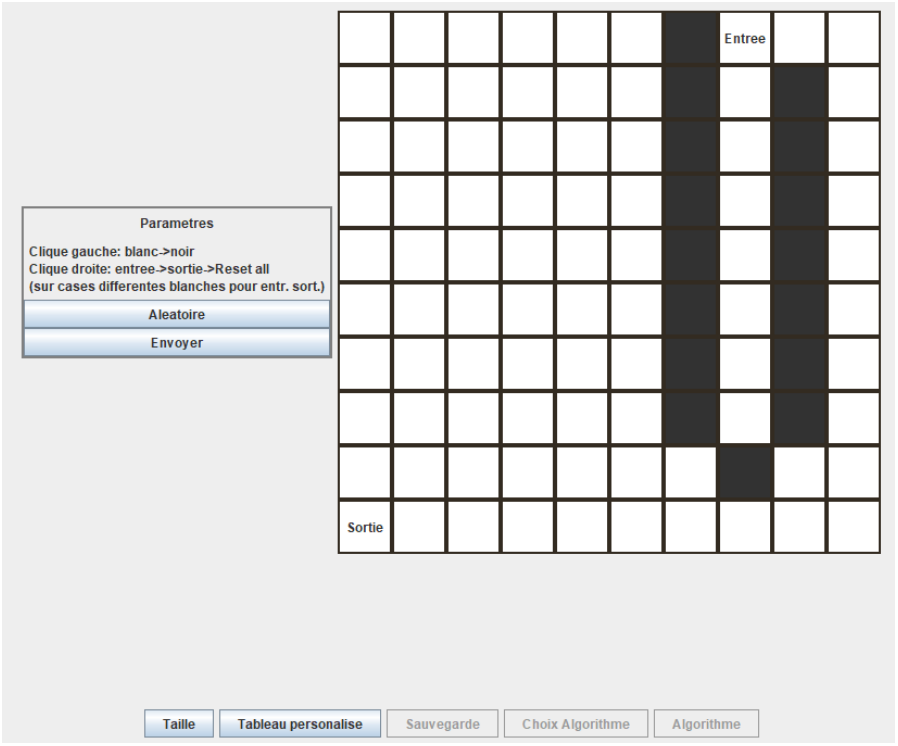
L'aléatoire utilise les méthodes setters de Carre en les appelant aléatoirement. Les entrées et sorties sont d'abord placées aléatoirement puis on appelle ou non les Carre pour qu'il deviennent des murs (si un Carre qui est une entrée ou une sortie est appelé pour devenir un mur, il ne deviendra pas un mur).
De la même façon que si l'utilisateur clique droit deux fois et qu'il a mis des entrées et sorties valables, la valeur qui vérifie si la grille est acceptable est la même que dans l'aléatoire. Donc elle vaut deux et le bouton Envoyer n'est plus grisé (s'il clique droit alors que la grille a été faite aléatoirement, elle sera réinitialisée).



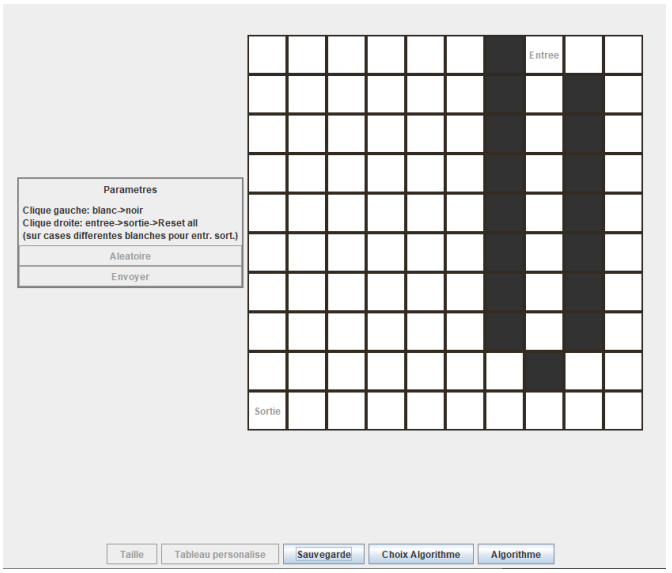
Une grille générée aléatoirement : elle est toujours acceptée



Une grille personnalisée où il manque la sortie : l'utilisateur ne peut pas Envoyer, bouton est grisé



La grille est acceptée il y a une sortie et une entrée

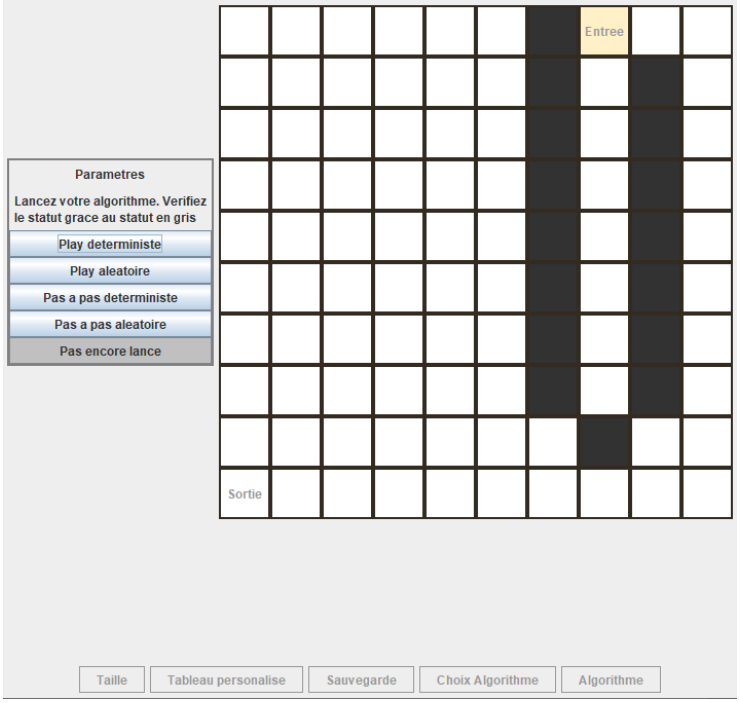


L'utilisateur a envoyé sa grille il peut continuer le programme grâce aux boutons de Frame en bas. Mais les listener ne répondent plus il ne peut plus modifier grâce à l'attribut statique Enabled dans la classe Carre.

V) L'algorithme

Algorithme.java

Selon le choix de l'utilisateur voici comme se déroule Algorithme de manière graphique :



ChoixAutomate retourne la méthode getAutomatique() = false

ChoixAutomate retourne la méthode getAutomatique() = true

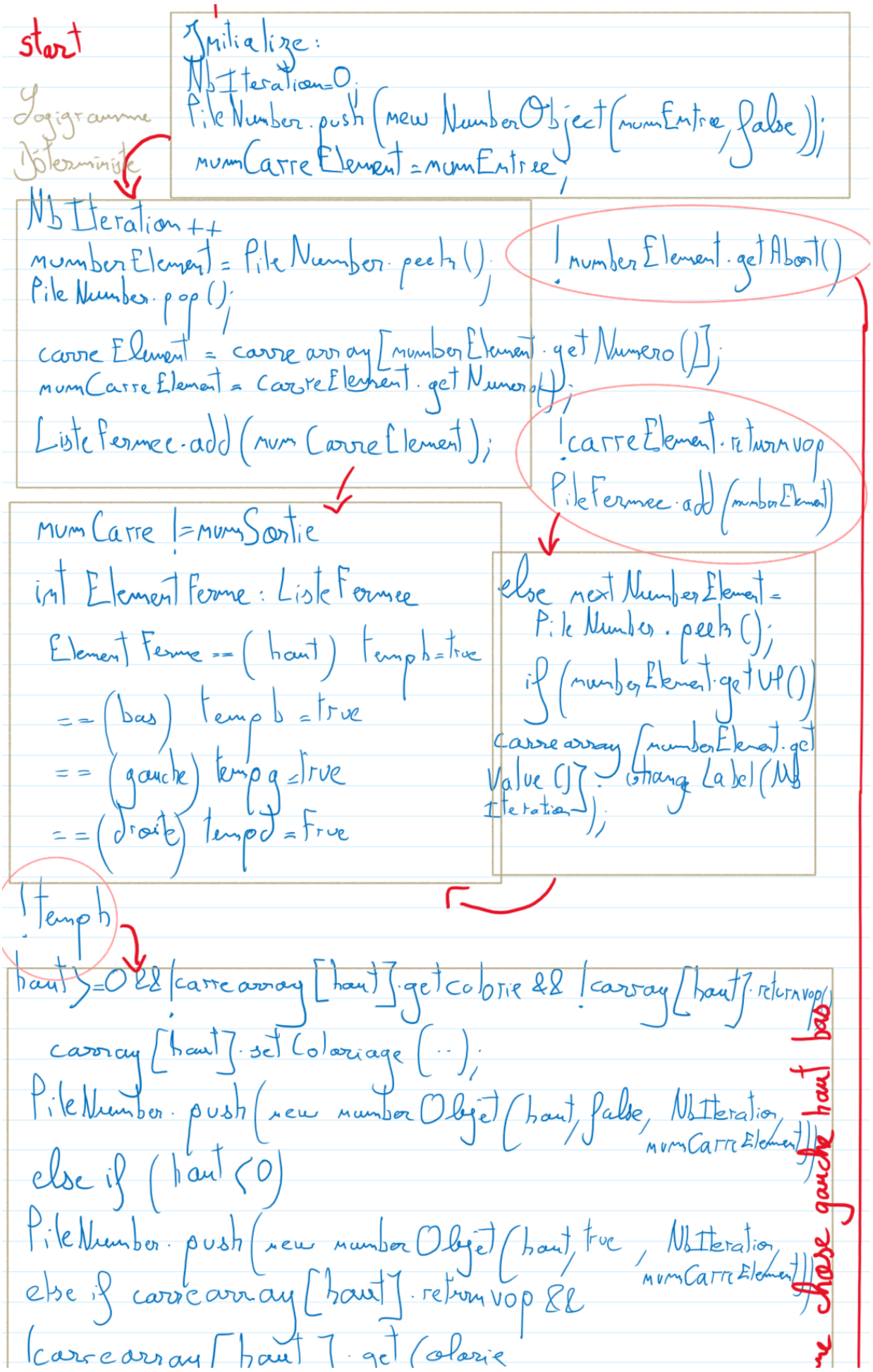
Play déterministe lance l'algorithme déterministe sans interruption.

Play aléatoire (x 100) lance l'algorithme aléatoire sans interruption.

Pas à pas déterministe montre étape par étape le fonctionnement de l'algorithme déterministe tant qu'on appuie dessus la prochaine fois.
Pas à pas aléatoire fonctionne de la même manière que pas à pas déterministe mais montre étape par étape une simulation de l'algorithme aléatoire.

Dés lors que l'on clique sur un bouton différent que celui qu'on a cliqué précédent on retourne à la case d'entrée (utile pour éviter de finir le pas à pas aléatoire).

Algorithme déterministe


start
Logigramme déterministe
Initialize :
NbIteration=0;
PileNumber.push(new NumberObject(numEntree, false));
numCarreElement=numEntree;
NbIteration++
numberElement = PileNumber.peek();
PileNumber.pop();
carreElement = carreArray[numberElement.getNumero()];
numCarreElement = CarreElement.getNumero();
ListeFermee.add(numCarreElement);
!numberElement.getAbort()
!carreElement.returnVop
PileFermee.add(numberElement)
numCarre != numSortie
int ElementFerme : ListeFermee
ElementFerme == (haut) tempb=true
== (bas) tempb=true
== (gauche) tempg=true
== (droite) tempd=true
else nextNumberElement =
PileNumber.peek();
if (numberElement.getVal()
carreArray[numberElement.getValue()].changeLabel(NbIteration);
!tempb
haut>=0 && !carreArray[haut].getColorie && !carray[haut].returnVop()
carray[haut].setColoriage(..);
PileNumber.push(new numberObjet(haut, false, NbIteration, numCarreElement));
else if (haut<0)
PileNumber.push(new numberObjet(haut, true, NbIteration, numCarreElement));
else if carreArray[haut].returnVop &&
!carreArray[haut].getColorie
même chose gauche haut bas

carrearray[haut].setColoriage(..);
PileNumber.push(new numberObjet(haut, false, NbIteration, numCarreElement));
else if (haut<0)
PileNumber.push(new numberObjet(haut, true, NbIteration, numCarreElement));
else if carrearray[haut].returnvop &&
(carrearray[haut].getColorie
PileNumber.push(new numberObjet(haut, true, NbIteration, numCarreElement, true));

même chose gauche haut ba

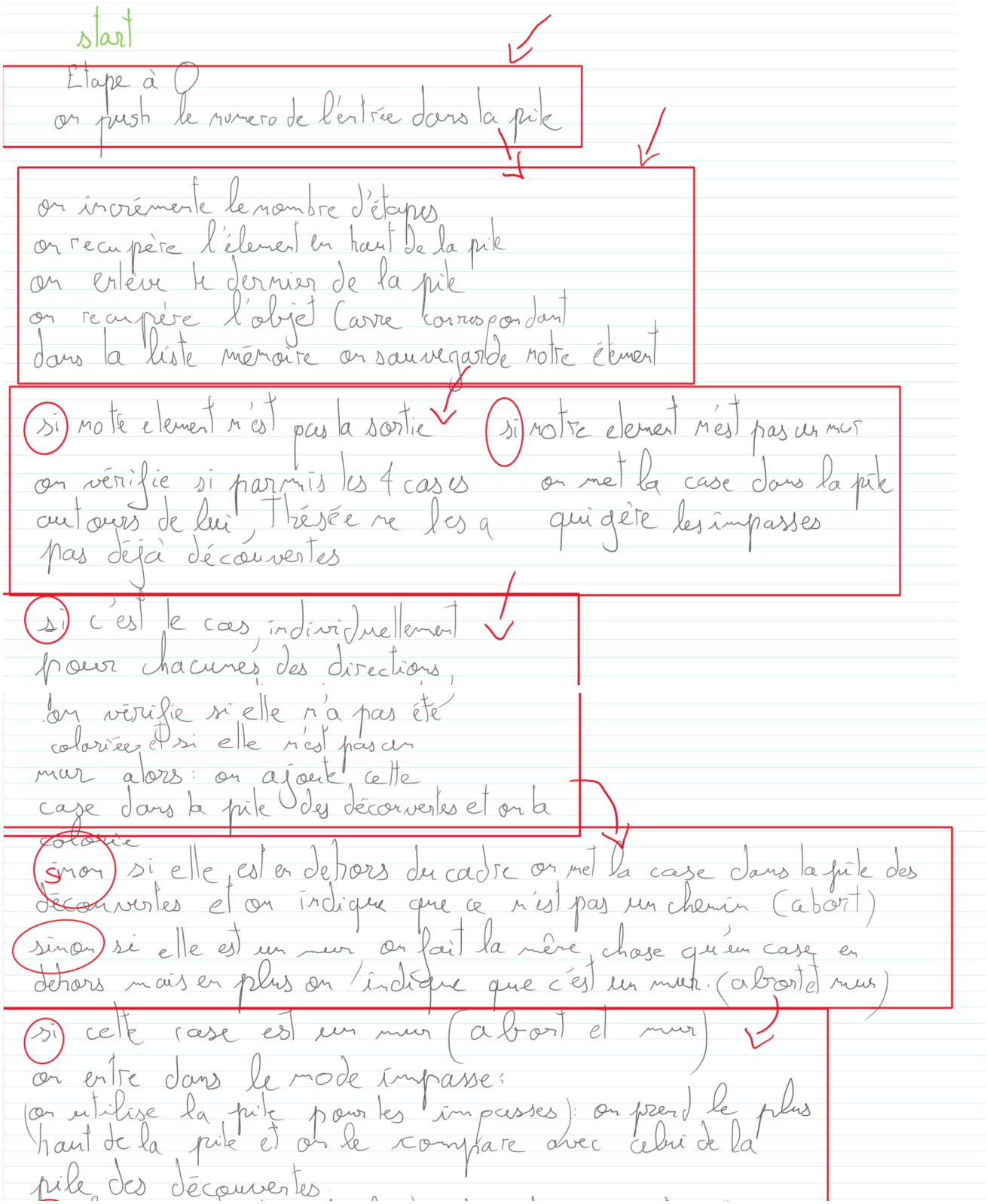
if(numCarreElement != numSortie)
nextNumberElement = PileNumber.peek()
numberElementFermee = PileNumber.peek()
while numberElementFermee.getNbIteration() >= next
numberElement.getNbIterationparent()
NbIteration ++
carrearray[numberElementFermee.getParent()].set
coloriage(... nbIteration);
numberElementFermee = PileFerme.pop();

else if
success == true
Terminé

else if numCarreElement == numSortie
Terminé

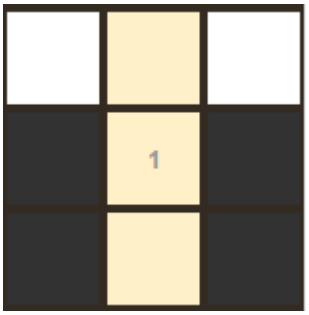
else Impossible

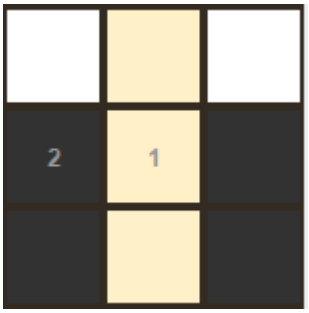
Algorigramme déterministe en français :

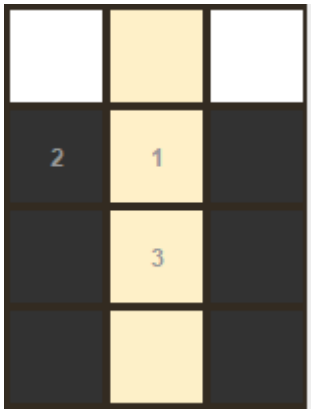


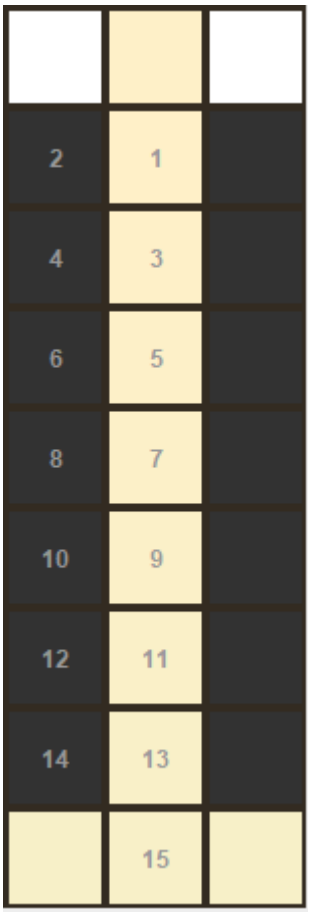
(on utilise la pile pour les "impasses") : on prend le plus haut de la pile et on le compare avec celui de la pile des découvertes.
si le parent de la liste des découverte (donc le prochain qu'on doit découvrir) est inférieur à celui qu'on a récupérer dans la pile pour les impasses on va remonter tant qu'on arrive au prochain qu'on doit découvrir.
A la fin: si la pile des découvertes est vide : impossible
si la case qu'on est entrain de découvrir est la sortie alors

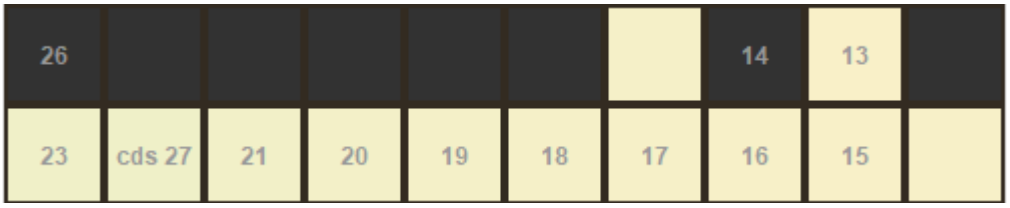
L'algorithme déterministe stocke les prochaines cases à traiter dans une pile. Par exemple ici l'algorithme prend toutes les cases autours de lui, haut, droite, bas, gauche. La prochaine fois lorsqu'on dépilera on se déplacera donc à gauche. Comme il s'agit d'un mur alors on dépile sans avancer et on prend donc maintenant la case du bas par rapport au numéro 1. Thésée peut s'avancer donc il sauvegarde à nouveau les cases autours de lui et le met dans sa pile. Et ainsi de suite. Thésée a donc une préférence à se déplacer vers la gauche donc il essaye toujours les murs à sa gauche.





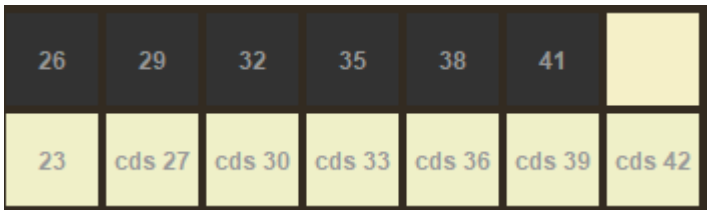




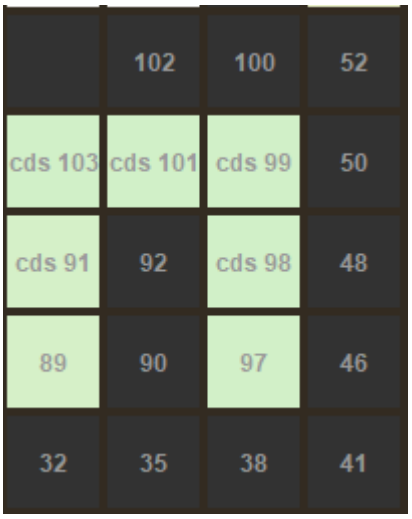


Après avoir continué celui si c'est retrouvé bloqué. A l'étape 23 il a essayé les extrémités (gauche, bas +2) mais n'a pas pu avancer. Il essaye en haut il ne peut pas, il est

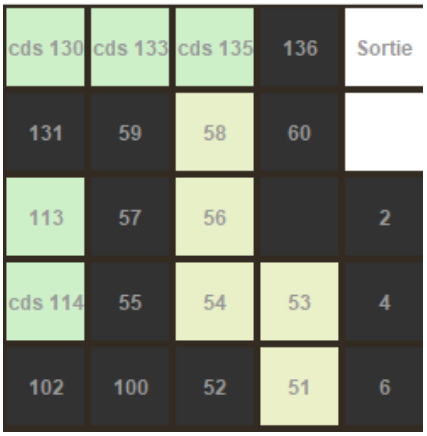
bloqué dans un « cul de sac » (cds).
C'est alors à ce moment qu'on utilise PileFermée qui sauvegarde les anciens déplacements de Thésée (seulement les chemins) et on dépile celle-ci dans le cas où le parent de la prochaine case à traiter (de la pile principale) est inférieur au parent de celui qu'on traite. Et on dépile tant que Thésée n'est pas allé sur la case où doit se rendre pour la prochaine étape.



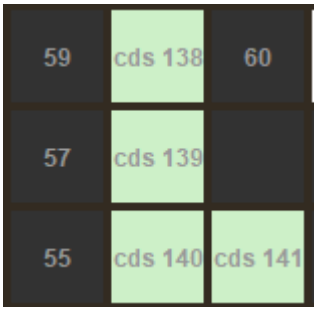
Thésée ici à chaque fois essayé la case en bas et celle du dessus sans succès. Comme il se retrouve bloqué à chaque étape il retourne sur ses pas.

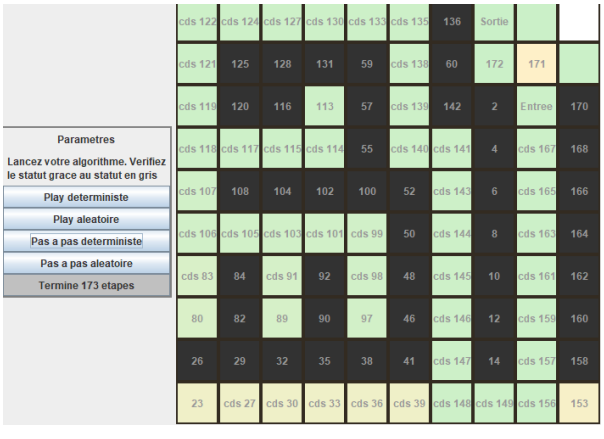


Etape 97 : Thésée grâce à la ListeFermee permet de se souvenir de ne pas retourner sur une case déjà parcourue. Comme il a déjà testé 90,38,46 il est bloqué et remonte jusqu'à 99 car la prochaine case qu'il doit tester est 100.

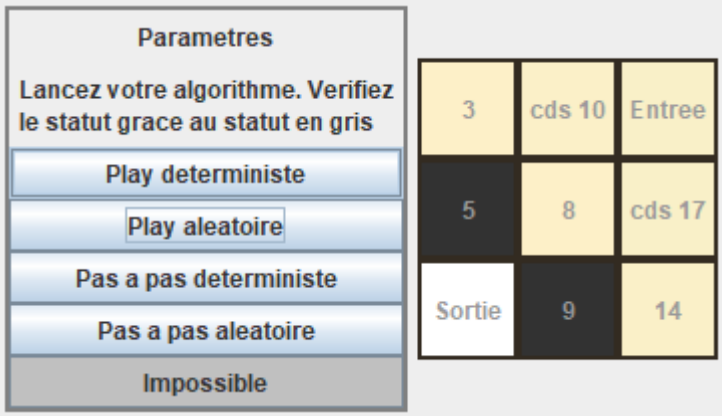


Thésée va tester la case au dessus de lui mais après il sera bloqué.
Il doit aller après sur le mur au dessus de 53. Il va retourner sur ses pas tant qu'il n'a pas atteint la case au dessus de 53.





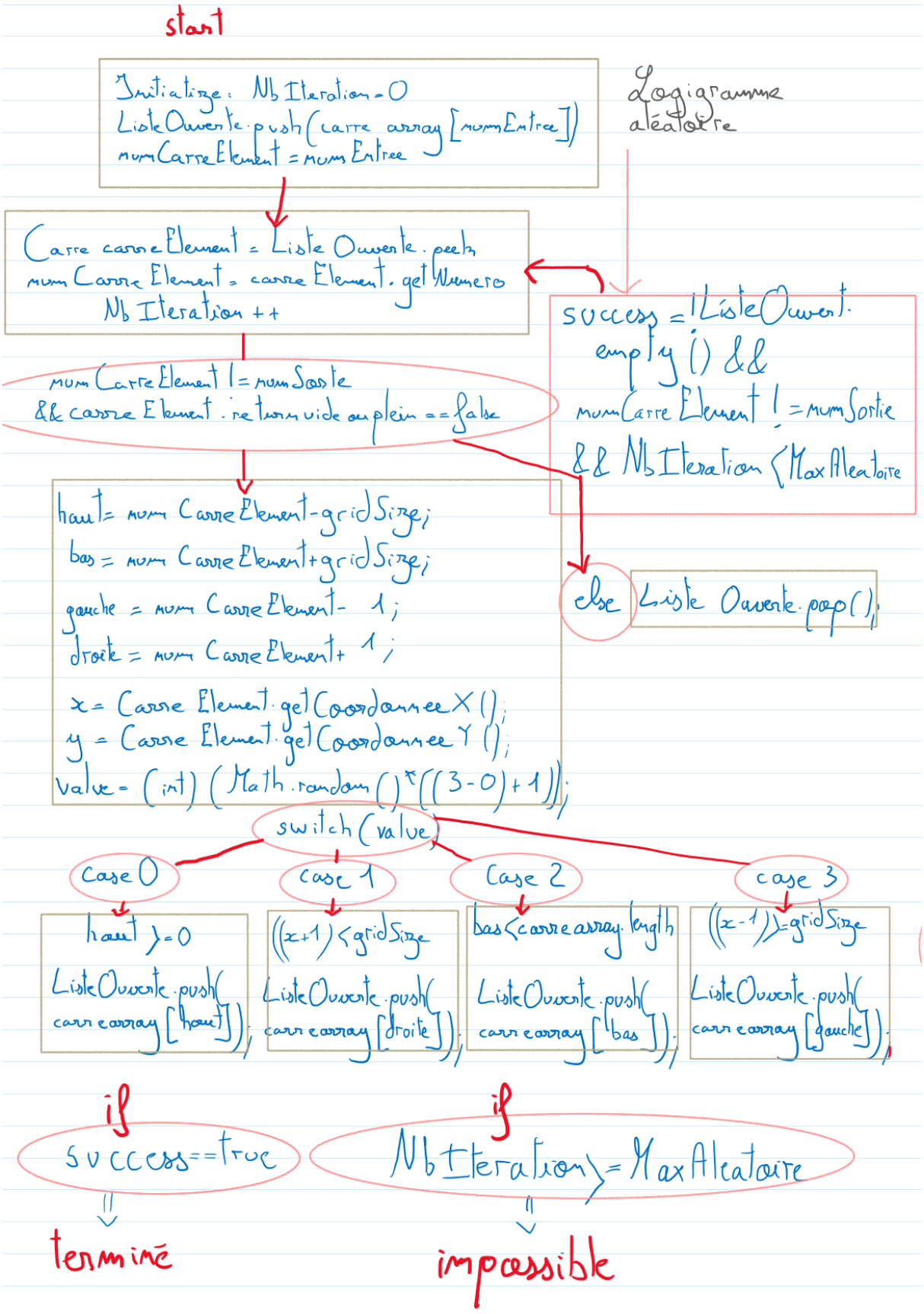
A la fin : étape 172, Thésée a déjà fait la case 60,2,171 donc il ne peut que monter et à l'étape 173 il est sur la sortie.



Si Thésée est dans un labyrinthe bloqué il peut savoir s'il est impossible à réaliser : il testera toutes les cases qu'il peut tant que la pile n'est pas vide. Si elle est vide alors Thésée ne peut pas atteindre la sortie.
Par extension si Thésée atteindra toujours la sortie si elle est atteignable.

Pour récapituler dans l'algorithme aléatoire Thésée utilise une pile principale qui sauve toutes ses découvertes de nouvelles cases.
Cette pile contient des objets qui sauvegardent l'emplacement par rapport à la grille et par rapport au nombre d'étapes.
Thésée a une mémoire : ListeFermée lui permet de ne jamais redécouvrir une case qui a déjà été découverte.
Et PileFermée qui sauvegarde les chemins empruntés pour qu'il puisse se sortir d'une impasse.

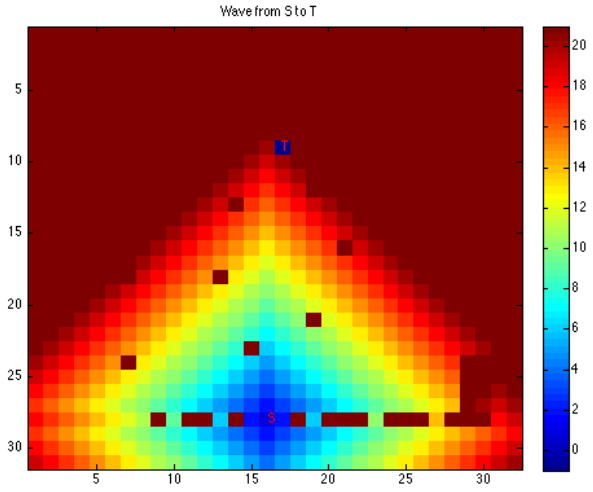
Algorithme aléatoire



L'algorithme fonctionne pratiquement de la même manière que le déterministe mais il n'a pas de mémoire. Il n'utilise donc pas NumberObject, ni de ListeFermée il utilise directement une pile de Carre. À chaque étapes Thésée tire un nombre et selon le nombre il ira dans une direction. Il récupère toujours le dernier carré de sa pile, il ne dépile que lorsque qu'il rencontre un mur. Quand il se retrouve en dehors de la grille, le cas n'est pas compté mais on ajoutera +1 pour l'étape suivante.
L'algorithme aléatoire a une limite pour qu'on puisse voir comment il se comporte. Mais l'algorithme aléatoire x 100 n'en n'a pas pour pouvoir avoir des moyennes justes. On passe d'abord l'algorithme déterministe pour vérifier s'il y a une sortie et s'il n'y en n'a pas on ne fait rien.

VI) Conclusion personnelle Aurélien

Je me suis occupé de la partie graphique (sauf Menu) et de l'algorithme.
Pour l'algorithme je me suis d'abord inspiré de celui de Lee, sous forme de vague avant de comprendre qu'un déplacement était égal à une étape.
J'ai donc changé d'optique en changeant ma file en pile ce qui ressemblait plus aux mouvements de Thésée.
Le projet était cadré pour les déplacements et données connues, j'aurai bien aimé appliquer des algorithmes plus complexes tels que l'A* ou Lee et avoir le plus cours chemin pour m'entrainer davantage sur les algorithmes.



Lee algorithm

Le projet en java a été plutôt difficile à repartir au niveau des tâches car j'avais besoins de composants graphiques propres pour créer mon algorithme. (Carre, NumberObject) Et de fils en aiguilles j'ai pu réutiliser facilement mon code pour créer les autres composants graphiques. Alex a pu m'aider pour que j'insère son code dans le mien, et aussi pour le transfert de données.
J'ai trouvé qu'il était difficile en java de demander de l'aide pour les problèmes spécifiques, tels que l'algorithme. Dès lors qu'on commence à s'aider de plusieurs classes en même temps, il est difficile de contextualiser son souci.
Pour moi le java est un langage que l'on pratique plus facilement seul, car on peut s'aider de classes que l'on créé soit même plutôt que celles de collaborateurs qui peuvent ne pas répondre à nos problèmes.
J'ai même trouvé le langage C plus simple pour créer du travail collaboratif car le langage qui n'est pas objet permet à tout le monde d'utiliser les mêmes types et rapidement contextualiser un problème.
Néanmoins le java en lui-même est plus simple à comprendre car plus récent, et la partie graphique plus à portée de main que celle du projet précédent.
Je pense que pour une prochaine fois il sera plus simple de commencer à créer un diagramme de classe avant de commencer le projet pour bien répartir le travail.

Ce projet m'a permis de mieux comprendre les fonctionnements du transfert de données entre les objets en java.

VII) Conclusion personnelle Alex

Au début du projet contrairement au dernier projet, moi et Aurélien nous sommes mis tout de suite en équipe. On a commencé par réfléchir aux différentes façons de faire ce qui était demander ainsi que les notions que cela allait demander. Bien sûr grand stresser que je suis, en voyant la demande d'un automate de parcours déterministe et un non déterministe je me suis sentie comme les Avengers qui voient débarquer Thanos sur Terre avec le gant de l'infinie. Mais bon je ne suis pas partie non plus trop pessimiste. J'ai fait la classe labyrinthe en la perfectionnant comme je pouvais parce qu'à part ça j'étais complètement bloquer. J'ai aidé Aurélien du mieux que j'ai pu en lui donnant mon avis et en faisant des séances de débogage avec lui. Finalement passer autant de temps sur cette classe m'a permis de bien intégrer les principes de fonctionnement de Java mais aussi de m'essayer aux énumérations. Finalement qu'est-ce que je retiens de tout ça ? Eh bien une grande frustration. J'ai eu de grande difficulté à dormir et mes vacances se sont déroulée dans le stress. Mais j'ai aussi constaté autre chose. Le dernier projet me passionnait car je savais pourquoi je le faisais. Créer un jeu bien que basique étais l'un de mes grands objectifs mais aussi que la finalité était d'apportée du fun aux utilisateurs. Alors que celui-là je travaillais dessus sans vraiment avoir d'autres objectif qu'une note d'autant plus que le projet n'était pas super fun. Bien sûr c'est un avis subjectif qui n'engage que moi. Mais je me rends compte maintenant que la bonne note est une « carotte » qui me motive de moins en moins. Finalement, je me rends compte que cela m'a permis de mieux me connaître. Je pense que je vais reprendre ce projet plus tard dans le but de le modifié et de changer sa finalité pour le rendre plus interactif. En tout cas avec cette expérience de Java qui mon permis de m'entraîner j'ai hâte de me pencher sur le fonctionnement de mode Minecraft.