



Rapport de projet Cynapse

Filière: Première année d'ingénieur en informatique
ING1 GI groupe 7

Présenté par :

Bari-Joris BOICOS - Lorelle WENG

Florianne PAN - Jonathan NGO

Junjie SHAO

Année: 2024-2025

Sommaire

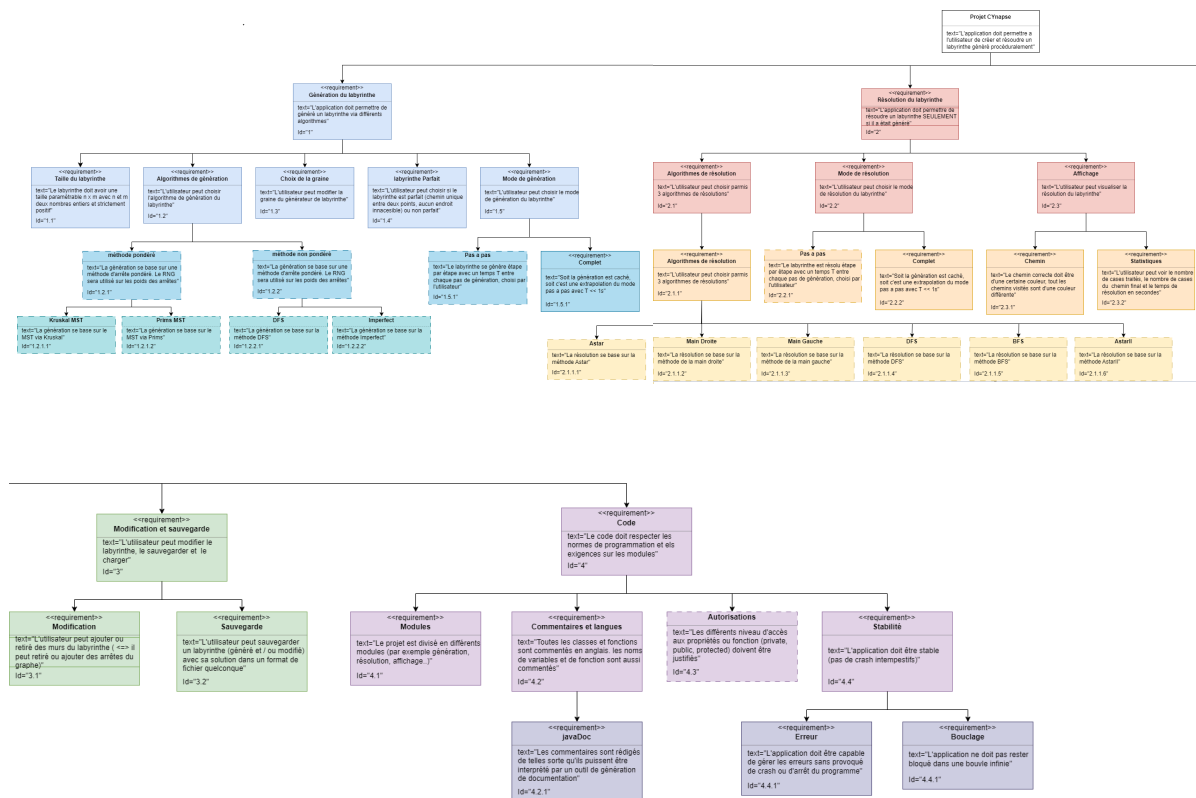
1. Présentation du projet.....	3
2. Cahier des charges.....	4
3. Diagrammes UML du projet.....	6
4. Répartition des tâches.....	8
5. Développement du projet.....	9
5.1. Architecture Globale.....	10
5.2. Le générateur.....	12
5.3. Le solveur.....	19
5.4. L'application en ligne de commande (CLI).....	23
5.5. JavaFX.....	26
6. Conclusion.....	29

1. Présentation du projet

Le projet CYNapse vise à développer une application permettant de générer et de résoudre des labyrinthes à l'aide d'une interface graphique intuitive. Cette application offre à l'utilisateur la possibilité de créer des labyrinthes grâce à une génération procédurale basée sur une graine définie, et sur l'algorithme de génération de son choix. L'utilisateur a également la possibilité de choisir sa méthode de résolution de labyrinthe, et par la suite, de générer la solution sur un labyrinthe. Il est aussi permis de modifier localement la topologie du labyrinthe en sélectionnant graphiquement des positions pour ajouter ou supprimer des murs.

Afin de mener à bien ce projet, nous avons utilisé plusieurs outils: le langage de programmation Java a été choisi, car il était requis dans le cadre du projet, le framework javaFX a été utilisé afin de concevoir une interface graphique fluide, ergonomique et facile à manipuler, le logiciel de dessin graphique draw.io afin de dessiner des diagrammes de classe, de séquence, de cas d'utilisation, ainsi que le cahier des charges et GitHub comme système de gestion de versions, afin de faciliter la collaboration et le suivi des modifications du code.

2. Cahier des charges



lien vers le cahier des charges détaillé en plus visible : <https://drive.google.com/file/d/1-z3FAPb6yfkFmB98rGCCtHmw7vLykrzT/view?usp=sharing>

L'application Cynapse a été conçue dans le cadre de ce projet pour offrir une plateforme interactive de génération et de résolution de labyrinthes. Le cahier des charges impose un ensemble d'exigences fonctionnelles et techniques visant à garantir à la fois la richesse fonctionnelle de l'outil et sa robustesse technique.

L'utilisateur peut générer un labyrinthe en spécifiant une graine ainsi que les dimensions souhaitées. Deux modes d'affichages sont proposés: dans le mode complet, la génération s'effectue en tâche de fond, suivie de l'affichage du labyrinthe final, dans le mode pas-à-pas, l'utilisateur assiste à chaque étape de génération, avec un contrôle sur la vitesse d'animation.

Les labyrinthes générés peuvent être parfaits, c'est-à-dire sans cycles et avec une unique solution entre deux points donnés, ou imparfaits, présentant des cycles ou même des impasses, donc potentiellement sans solution.

Nous avons implémenté divers algorithmes de génération qui sont définis dans l'énumération GenMethodName: l'algorithme de Kruskal, qui utilise des arêtes pondérées pour construire un arbre couvrant minimal; l'algorithme de Prim, qui présente une logique de croissance progressive similaire

à Kruskal; l'algorithme DFS (Depth First Search), basé sur une exploration récursive ou avec une pile; ainsi qu'un mode "imperfect" qui génère un labyrinthe avec un nombre d'arête aléatoire entre $\frac{3}{8}$ et $\frac{6}{8}$ du nombre total d'arêtes possible.

La résolution d'un labyrinthe ne peut être initiée qu'après sa génération. Nous avons implémenté six algorithmes de résolution qui sont regroupés dans l'énumération SolveMethodName. Parmi ceux-ci figurent l'algorithme Astar, une méthode heuristique optimale, les méthodes du mur droit et du mur gauche, qui sont des approches déterministes basées sur le suivi d'un mur, ainsi que les algorithmes classiques DFS et BFS, respectivement d'exploration en profondeur et en largeur. Une variante avancée appelée AstarII est également disponible à titre comparatif.

Deux modes de résolution sont proposés : le mode complet qui effectue l'exécution intégrale avec visualisation finale, et le mode pas-à-pas qui permet une animation progressive avec un affichage clair des cases traitées et du chemin final.

À l'issue de la résolution, plusieurs métriques de performance sont affichées, notamment le nombre de cases visitées, la longueur du chemin final, ainsi que le temps de traitement.

L'utilisateur a la possibilité d'ajouter ou de supprimer localement des murs dans le labyrinthe. La résolution s'adapte dynamiquement à ces changements sans nécessiter un recalcul complet, sauf dans le cas de modifications majeures.

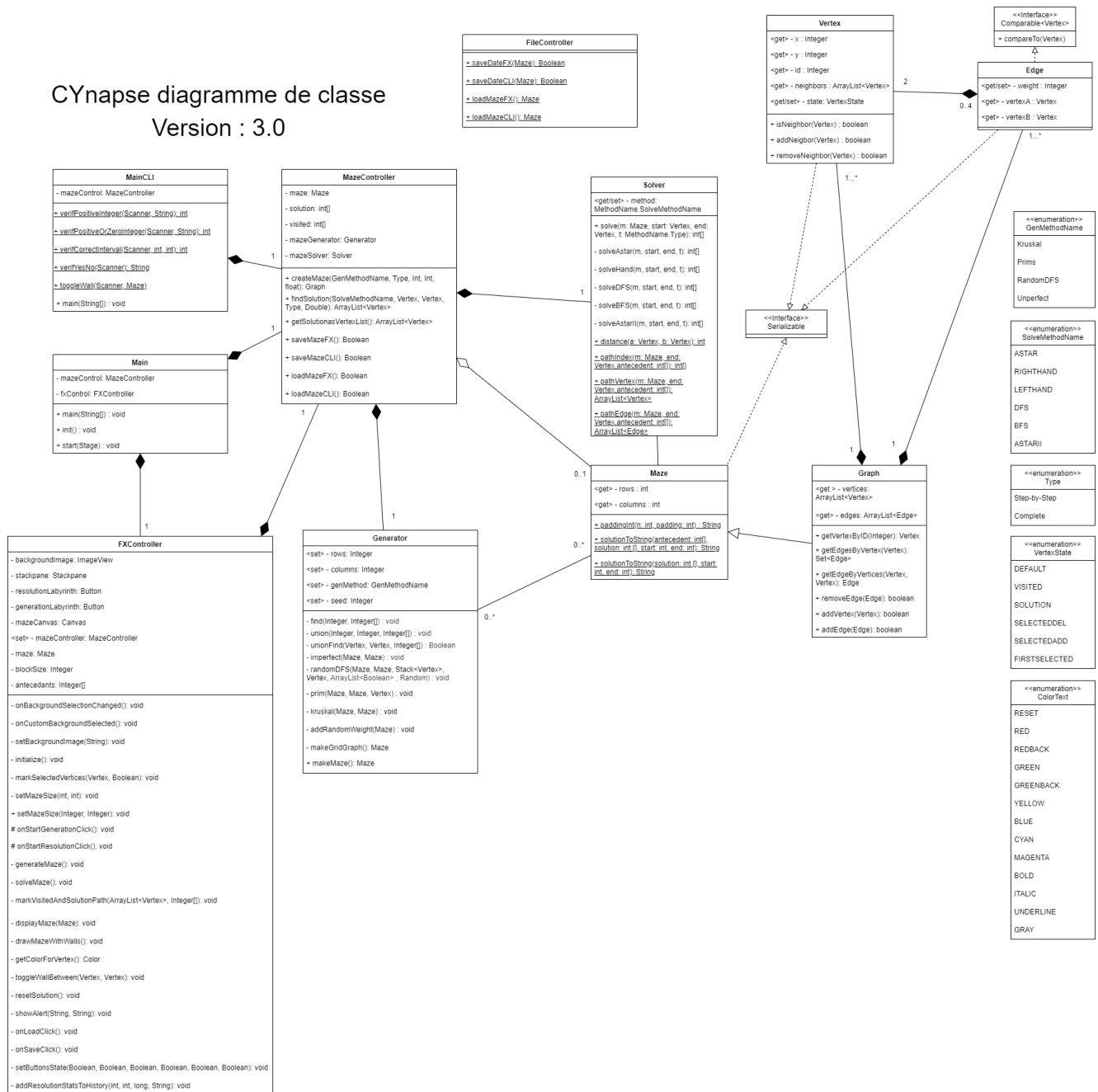
L'application permet de sauvegarder un labyrinthe, qu'il soit généré ou modifié, avec l'extension ".ser". L'utilisateur peut ensuite charger ce fichier et continuer à l'exploiter comme s'il venait d'être généré.

Le développement de l'interface a été réalisé en JavaFX afin de garantir une interface graphique intuitive et ergonomique. L'application offre ainsi une expérience utilisateur fluide et stable, sans crash ni blocage intempestif.

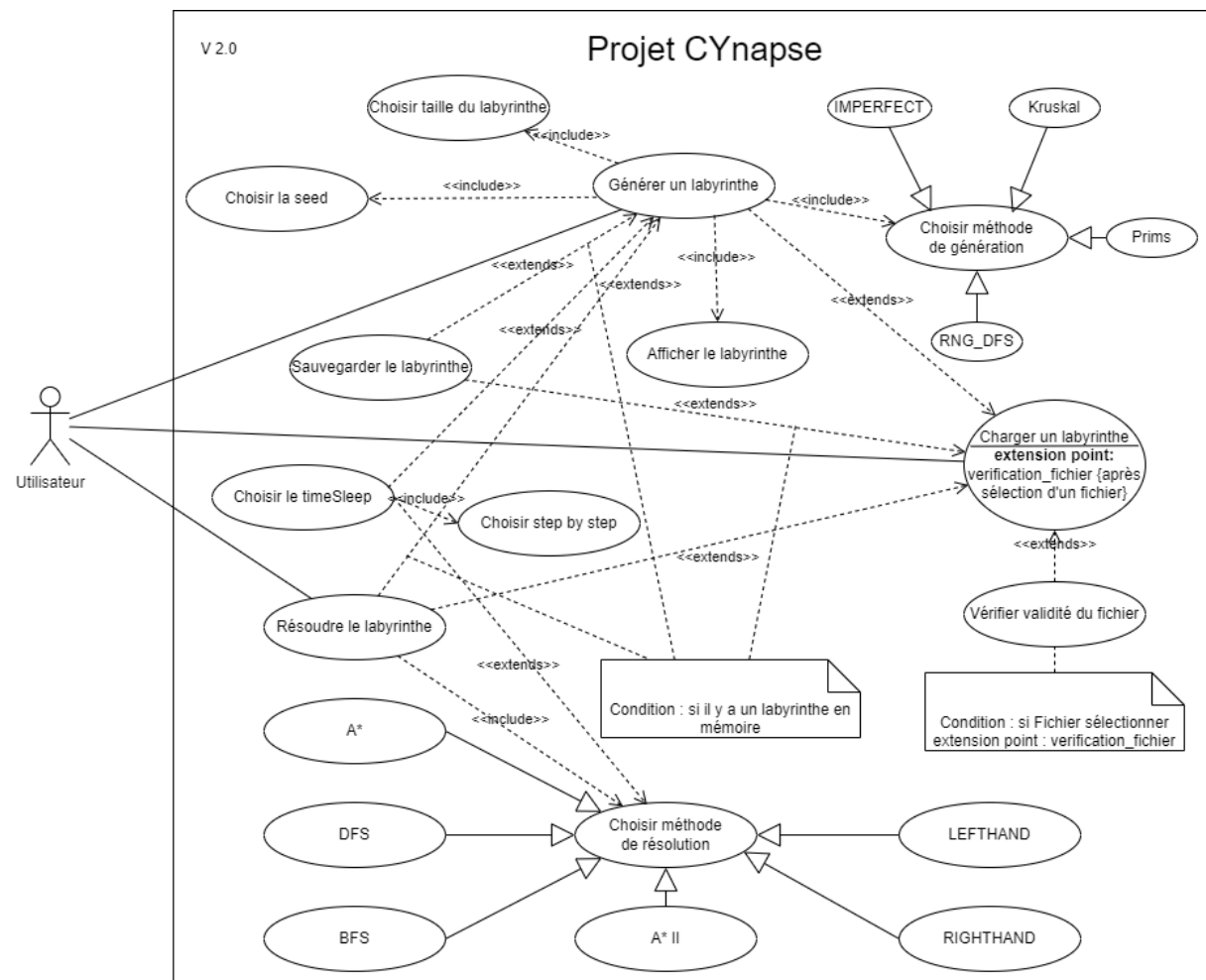
Le code source est documenté en anglais, avec des commentaires compatibles JavaDoc. Il est organisé en modules distincts, ce qui facilite la maintenabilité et les évolutions futures du projet.

3. Diagrammes UML du projet

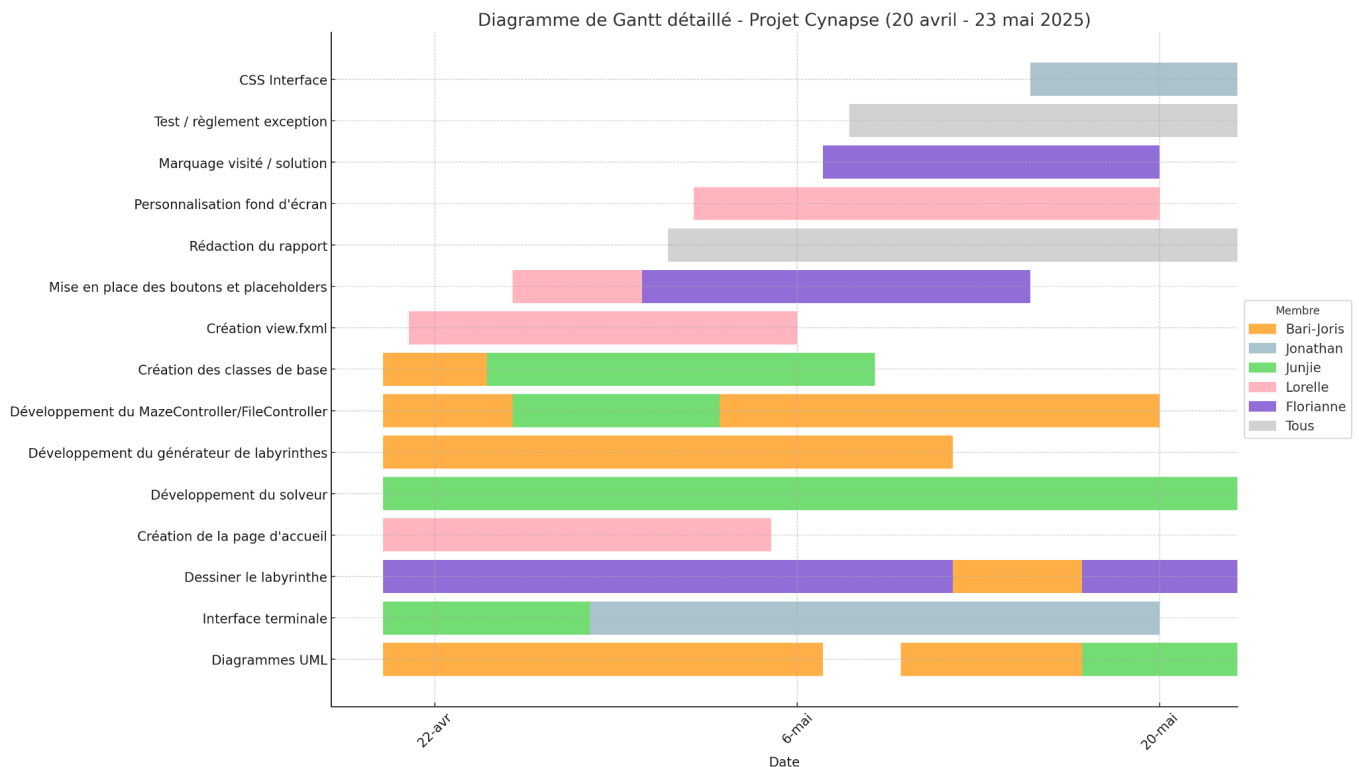
CYnapse diagramme de classe Version : 3.0



Un diagramme de cas d'utilisation a également été ajouté:



4. Répartition des tâches



Bari-Joris s'est tout d'abord occupé de la création du diagramme UML, puis a implémenté les algorithmes de génération de labyrinthe qui sont dans `Generator.java`. Il a par la suite géré `MazeController.java` et `FileController.java` qui sont respectivement là où les labyrinthes sont générés, résolus ou stockés et là où les labyrinthes peuvent être sauvegardés ou chargés.

Junjie quant à lui, s'est occupé des algorithmes de résolution de labyrinthe qui sont dans `Solver.java`. Il a de plus contribué au développement de `Maze.java`.

Jonathan a travaillé sur la mise en place de l'interface terminale (CLI) puis s'est chargé du style de l'interface graphique, que l'on peut retrouver dans le fichier `style.css`.

Lorelle et Florianne étaient chargées de l'interface graphique JavaFX: Lorelle s'est occupée de la création de la page d'accueil, de la personnalisation du fond d'écran et de la mise en place des boutons. Florianne quant à elle, s'est occupée du dessin dynamique du labyrinthe, du marquage des cases visités et la solution et plusieurs autres éléments graphiques. De plus, Bari-Joris a contribué à l'amélioration de la performance de l'affichage graphique du labyrinthe.

Enfin, les phases transversales comme la rédaction du rapport et les tests ont mobilisé tous les membres de l'équipe, afin de s'assurer de la qualité et du bon déroulement du projet.

5. Développement du projet

Nous verrons par la suite, le détail de notre projet. Cela comprend l'architecture globale du projet, donc la disposition des classes, ressources etc, puis les algorithmes de génération de labyrinthe suivi des algorithmes de résolutions de labyrinthe. Nous verrons ensuite, l'application en ligne de commande et ses fonctionnalités, et enfin, l'interface graphique JavaFX.

5.1. Architecture Globale

L'architecture globale de notre projet suit une structure classique de projet Java basé sur Maven. A la racine, on retrouve les fichiers de configuration tels que pom.xml (fichier de configuration Maven), README.md, et les scripts mvnw pour exécuter Maven. Les diagrammes de classe/cas d'utilisation et le cahier des charges réalisés à l'aide de draw.io sont regroupés dans le dossier drawIoDiagram.


📁 .idea	test merge	2 weeks ago
📁 .mvn/wrapper	Initialisation	last month
📁 drawIoDiagram	color changes	3 hours ago
📁 src/main	enlever alerte	2 hours ago
📄 .gitignore	update: ignorer .ser	last week
📄 Projet_CYnapse_ING1_2024_2025.pdf	Debut du CLI	2 weeks ago
📄 README.md	Update README.md	4 hours ago
📄 mvnw	Initialisation	last month
📄 mvnw.cmd	Initialisation	last month
📄 pom.xml	javadoc génère pour les private	6 hours ago
📄 test.ser	update: changement du labyrinthe exemple	yesterday


Le cœur de l'application se trouve dans src/main/java/com/example/projetcynapseing1/, où l'on trouve les différentes classes du projet. Parmi elles, Main.java constitue le point d'entrée principal de l'application graphique, tandis que MainCLI.java permet une utilisation en ligne de commande. Les classes telles que Graphe.java, Edge.java et Vertex.java modélisent la structure du graphe utilisé dans le projet. Maze.java et MazeController.java sont liées à la génération et à la gestion des labyrinthes. Solver.java implémente les algorithmes de résolution, et Generator.java s'occupe de la génération de graphe. MethodName.java contient les noms des différents algorithmes de génération et de résolution.





ProjetCynapse-ING1 / src / main / java / com / example / projetcynapseing1 /			Add file	...
Floriannee enlever alerte			904638c · 2 hours ago	History
Name	Last commit message	Last commit date		
..				
Edge.java	Merge remote-tracking branch 'origin/temp' into Floriannee	4 days ago		
FXController.java	enlever alerte	2 hours ago		
FileController.java	fix: la sauvegarde CLI ne marchait pas	3 hours ago		
Generator.java	scroll bar color when hover	yesterday		
Graph.java	changement pour la javadoc	6 hours ago		
Main.java	affichage plus grand	yesterday		
MainCLI.java	Merge branch 'temp' of https://github.com/Minetriforce/ProjetCynapse-...	2 hours ago		
Maze.java	changement pour la javadoc	6 hours ago		
MazeController.java	fix: la sauvegarde CLI ne marchait pas	3 hours ago		
MethodName.java	update: changement du labyrinthe exemple	yesterday		
Solver.java	update: message d'erreur quand pas de méthode de résolution choisie	yesterday		
Vertex.java	fixcreunion	4 days ago		

La classe FXController.java gère l'interaction entre l'interface graphique et la logique métier, tandis que FileController.java permet de charger et de sauvegarder des graphes pour l'interface graphique tout autant que sur les lignes de commande.

Les ressources nécessaires à l'interface graphique (le fichier view.fxml, le fichier style.css et le dossier "images") sont placés dans src/main/resources. L'architecture respecte donc une séparation claire entre la logique applicative, les contrôleurs, les ressources, et les éléments de configuration, facilitant la maintenance et l'évolution du projet.

ProjetCynapse-ING1 / src / main / resources /  Add file ▾ ⋮

 LorelleW Merge remote-tracking branch 'origin/temp' into Lorelle f0fec4e · 5 hours ago [History](#)

Name	Last commit message	Last commit date
 ..		
 images	Merge branch 'Lorelle' into temp	4 days ago
 style.css	changement commentaire	6 hours ago
 view.fxml	changement affichage	7 hours ago

5.2. Le générateur

Le générateur, au sens de génération de labyrinthe, est une classe réunissant plusieurs algorithmes pour générer des labyrinthes parfaits ou imparfaits. Le but du générateur est donc de prendre en arguments dans son constructeur et de renvoyer un objet de type Maze (cf diagramme de classe).

Préambule : tous les algorithmes de génération fonctionnent à partir de deux graphes distincts : un graphe en grille et un graphe résultat.

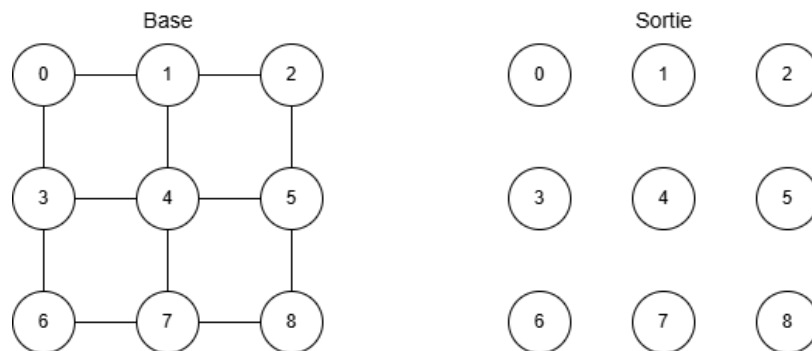


fig 5.1.1 : composants des algorithmes

L'objectif est d'utiliser le graphe de base comme un plan pour le graphe de sortie. Celui-ci nous permet aussi de savoir quelles sont les arêtes autorisées ou non.

Sur l'exemple précédent : l'arête du sommet 0 à 1 est autorisée, tandis que celle de 0 à 2, qui n'est pas sur le graphe de base, n'est donc pas autorisée.

Méthode de Kruskal (Parfait)

Par définition : un labyrinthe parfait est un labyrinthe dans lequel tout point est accessible et qu'il ne contient pas de cycles. Cette définition peut être assimilée à un arbre couvrant minimal (Minimum Spanning Tree ou MST) dans un graphe.

Cependant, un arbre couvrant minimal n'est valable que dans un graphe avec des arêtes pondérées, ainsi pour cette première méthode, le graphe passe par une méthode d'initialisation qui ajoute un poids aléatoire entier entre zéro et cent. (c.f. algorithmes outils)

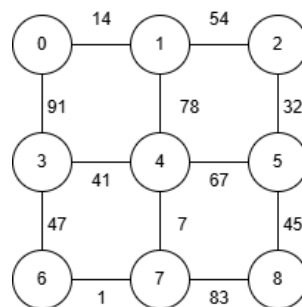


fig 5.2 : attribution de poids aléatoires sur les arêtes

De ce graphe, l'algorithme de Kruskal permet d'en tirer un arbre couvrant minimum : le but est de trier toutes les arêtes du graphe par poids croissant puis de parcourir cette liste en ajoutant des arêtes si elles ne créent pas de cycles dans le graphe (c.f. algorithme union-find dans les algorithmes outils)

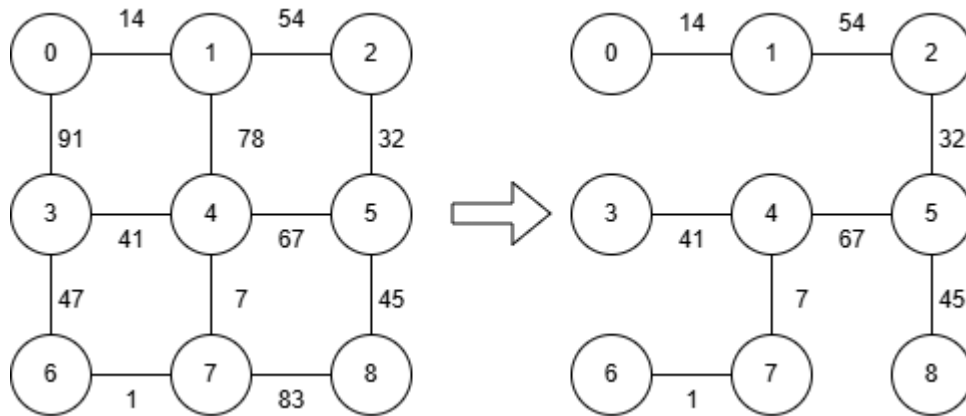


fig 5.3 : application de l'algorithme de Kruskal sur un graphe pondérée

La génération est finie, le graphe de sortie est alors un labyrinthe parfait.

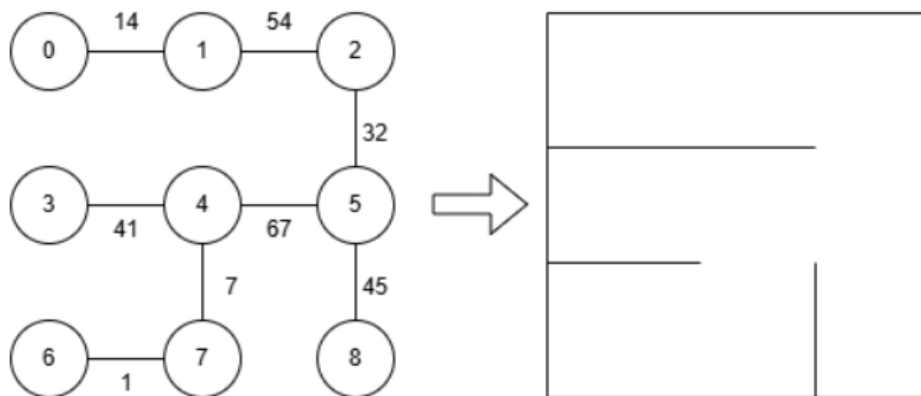


fig 5.4 : mise en forme du graphe

Complexité temporelle : $O(E + V)$

Soit un labyrinthe de taille $n = l * L$ avec n le nombre de sommets du graphe.

L'algorithme de détection de cycle n'est pas pris en compte. (complexité constante, voir plus loin algorithme outils). L'algorithme de étant celui implémenté de base dans la classe Collections de Java, celui-ci est le trie fusion avec une complexité de $O(E * \text{Log}(E))$.

Méthode de Prim (parfait)

Le principe de cet algorithme est le même que Kruskal à la différence qu'il transforme le problème global en une succession de problèmes locaux. On cherche encore une fois à trouver l'arbre couvrant minimum dans un graphe en grille dont les poids ont été choisis aléatoirement. (cf Fig 5.3).

Le but de cet algorithme est que pour chaque itération, la liste des arêtes liées à l'arbre actuel est triée par ordre croissant et la première d'entre elles est la plus grande.

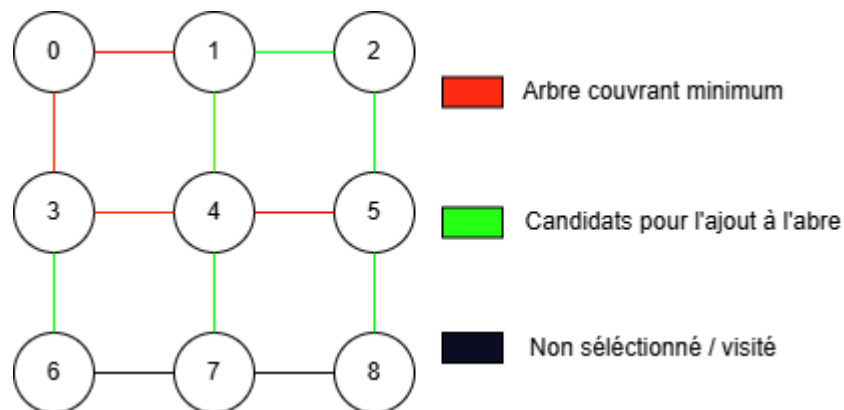


fig 5.5 : étape n de la méthode de Prim

Complexité temporelle : $O((E + V) * E * \log(E))$

L'algorithme de détection de cycle est en temps constant $O(1)$, le but étant d'avoir un arbre couvrant minimum, la boucle se fera sur le nombre d'arêtes E et le nombre de sommets V . De plus, le tri fusion a une complexité de $O(E * \log(E))$, répété à chaque itération.

Méthode DFS aléatoire

Cette méthode de génération se base sur l'algorithme de *Depth First Search* modifié afin de "choisir aléatoirement son prochain voisin".

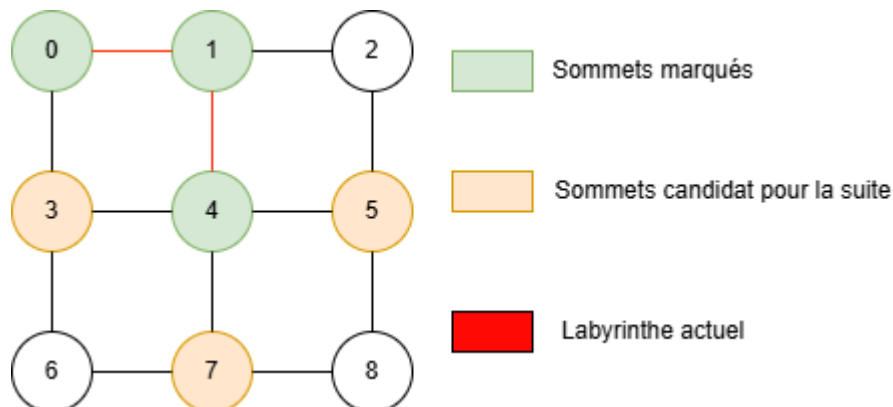


fig 5.6 : étape n du DFS aléatoire : liste des sommets candidats et choix aléatoire

Dans la figure 5.6, l'algorithme est actuellement sur le sommet 4. Ainsi, la liste des voisins possibles sera faite -seulement des voisins non marqués- et l'algorithme va choisir aléatoirement un sommet adjacent tout en ajoutant le sommet 4 à la pile des sommets visités.

Dans le cas où il n'y a que des sommets marqués comme voisins du sommet actuel : l'algorithme va dépiler le sommet précédent de la pile des sommets visités, et ce jusqu'à avoir un sommet dont l'un des voisins est non marqué.

Complexité temporelle :

Soit un labyrinthe de taille $n = l * L$ avec n le nombre de sommets du graphe.

L'algorithme de détection de cycle n'est pas pris en compte.

Détecteur de cycle

Dans tous les algorithmes de génération, il faut à chaque étape vérifier si l'ajout d'une arête au graphe finale (le labyrinthe) crée un cycle ou non, le but étant d'avoir toujours des labyrinthes parfaits, c'est-à-dire des graphes connexes sans cycles.

Comme cet algorithme de détection de cycle est appelé à de très nombreuses reprises, il faut donc qu'il soit efficace. La première idée était un simple DFS -*depth first search*- pour savoir si deux sommets sont déjà connectés par un autre chemin.

```
* This function is used to know if a path exists between s1 and s2 in graph,
* using Depth-first search (DFS)
*
* @param graph
* @param s1 (Vertex)
* @param s2
* @return Boolean (true: a path exists, false: there's no path)
* @see Vertex
* @see Maze
*/
private Boolean DFScheck(Maze graph, Vertex s1, Vertex s2) {
    ArrayList<Boolean> mark = new ArrayList<Boolean>();
    for (int i = 0; i < graph.getVertices().size(); i++) {
        mark.add(e:false);
    }
    DFSRec(graph, s1, mark);
    return (mark.get(s2.getID()));
}

/**
 * Recursive Depth-first Search.
 *
 * @param graph
 * @param s
 * @param mark
 * @return mark
 */
private ArrayList<Boolean> DFSRec(Maze graph, Vertex s, ArrayList<Boolean> mark) {
    mark.set(s.getID(), element:true);
    for (Vertex neighbor : s.getNeighbors()) {
        if (mark.get(neighbor.getID()) == false) {
            DFSRec(graph, neighbor, mark);
        }
    }
    return mark;
}
```

fig 5.7 : code du détecteur anti cycle version DFS

Complexité temporelle : $O(E + V)$

- E : nombre d'arêtes du graphe
- V : nombre de sommets

Étant donné que l'arbre généré du labyrinthe augmente de taille à chaque itération, cette fonction à alors une **croissance Exponentielle**. Cette première méthode est connue et donc facile à implémenter mais devient limitante pour de grands labyrinthes.

Un autre algorithme permettant de détecter plus efficacement les cycles est l'Union-find.

```

/**
 * get the representative Vertex ID of the group of vertex i
 *
 * @param i      ID of the Vertex
 * @param parent list of groups (trees)
 * @return representative ID of the tree OR recursive call
 */
private int find(Integer i, int[] parent) {
    if (parent[i] != i) {
        parent[i] = find(parent[i], parent); // path compression to reduce access time
    }
    return parent[i];
}

/**
 * Create a union between two tree (groups)
 *
 * @param i      Representative ID of the first tree
 * @param j      Representative ID of the second tree
 * @param parent list of groups (trees)
 */
private void union(Integer i, Integer j, int[] parent) {
    Integer irep = find(i, parent);
    Integer jrep = find(j, parent);

    parent[jrep] = irep;
}

/**
 * Complete union find method.
 * It is used to detect cycle in a graph : If two vertices are in the same tree
 * (i.e. have the same representative vertex ID), then return true
 *
 * @param a      First Vertex to check
 * @param b      Second Vertex to check
 * @param parent List of groups (trees)
 * @return True: these two Vertices are in the same tree; False: these two
 *         Vertices are NOT in the same tree
 */
private Boolean unionFind(Vertex a, Vertex b, int[] parent) {
    int arep = find(a.getID(), parent);
    int brep = find(b.getID(), parent);

    return arep == brep;
}

```

fig 5.8 : code du détecteur anti cycle version Union-Find

Le principe : chaque sommet à un représentant appelé “représentant de l'arbre”.

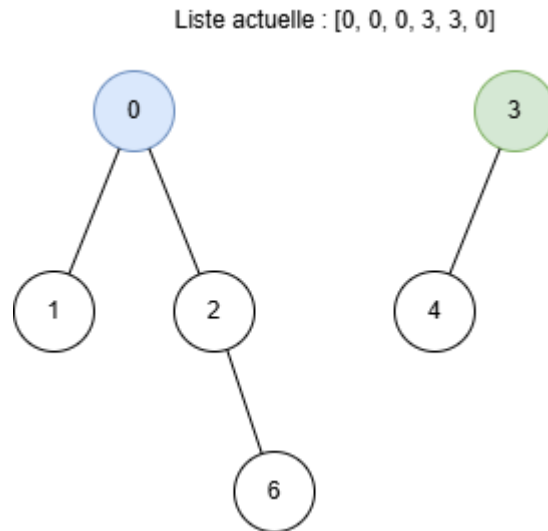


fig 5.9 : exemple d'arbres

Ainsi, le temps de détection d'un cycle entre deux sommets est donc réduit à une vérification.

Initialisation de l'algorithme Union-Find:

Une liste de n éléments est créée, avec n le nombre de sommets du graphe. Chaque index de cette liste représente le numéro du sommet et l'élément stocké à cet index est le numéro du sommet représentant de l'arbre.

Exemple d'initialisation avec 5 éléments : $\text{parents} = [0, 1, 2, 3, 4]$

Union : lorsqu'un sommet est ajouté à un arbre -par exemple le sommet 1 est ajouté à l'arbre 0- alors l'index de 1 dans la liste va contenir le représentant de l'arbre dans lequel il a été ajouté.

Exemple avec le sommet 1 ajouté à l'arbre 0 : $\text{parents} = [0, 0, 2, 3, 4]$

Find : la fonction find est utilisée pour avoir l'indice du représentant de l'arbre auquel appartient un sommet v . Dans l'exemple ci-dessus : $\text{find}(1) = 0$

Dans le cas où $\text{find}(0) \neq 0$, après avoir exécuté l'instruction $\text{find}(1) = 0$, alors la fonction find va exécuter un principe de réduction de chemin -path compression- pour réduire le nombre d'itération qu'il faut pour atteindre le vrai représentant de l'arbre du sommet v .

La fusion de ces deux principes donne ainsi la fonction Union-Find qui, pour deux sommets, renvoie un booléen indiquant si ceux-ci sont reliés par un chemin (non calculé).

Complexité temporelle : $O(1)$

Cette algorithm se base sur l'accès à un élément à l'index i d'une liste, la complexité est donc constante.

Méthode "imperfect" (imparfaite)

Comme son nom l'indique, le but de cet algorithme est de générer des labyrinthes imparfaits : ils peuvent contenir des cycles ou ne pas être connexes (endroits inaccessibles).

Déroulé de l'algorithme:

Initialisation : taille de labyrinthe $I * L$ avec n sommet $-I*L=n-$. On note E le nombre d'arêtes d'un graphe en grille de taille $I*L$.

Étape 1 : L'algorithme tire un nombre aléatoire entre $\frac{3}{8} E$ et $\frac{6}{8} E$ appelé E_s .

Étape 2 : L'algorithme effectue une boucle sur E_s éléments en **choisissant aléatoirement une arête dans le graphe en grille** pour l'ajouter au labyrinthe final.

Complexité temporelle: $O(n)$

La complexité est variable entre $\frac{3}{8} E$ et $\frac{6}{8} E$. La complexité est alors de $O(n)$.

5.3. Le solveur

Les solveurs de labyrinthe ont pour but de trouver un chemin d'un point de départ vers un point d'arrivée dans un labyrinthe.

Les solveurs prennent donc en entrée un labyrinthe (*Maze m*), un sommet de départ (*Vertex start*), un sommet d'arrivée (*Vertex end*), ainsi qu'une méthode d'affichage (*MethodName.Type t*).

Le moyen le plus courant de stocker la solution est d'utiliser un tableau d'entier *antecedents*, dans lequel on indique pour chaque sommet *i*, le sommet *antecedents[i]* qui le précède dans la recherche du chemin.

Le chemin:

start -> ... -> antécédent de l'antécédent de *end* -> antécédent de *end* -> *end*

Afin d'illustrer des sommets visités dans la recherche du chemin, nous avons utilisé un tableau d'entier *orders*. Le sommet *orders[0]* est *start*, le sommet *orders[1]* est le premier sommet visité, etc, *end* est le dernier sommet visité (s'il est accessible depuis *start*). Si *orders[i] = -1*, il n'y a plus de sommet visité par la suite.

Les solveurs renvoient *antecedents* ou *orders* en fonction de *t* (*COMPLETE* ou *STEPPER*).

Nous allons tester chacun des algorithmes suivant à l'aide d'un labyrinthe de taille 3x3, d'une seed 1, dont le point de départ est à la case 0 et le point d'arrivée à la case 8.

On va noter par la suite: *E* le nombre de sommets et *V* le nombre d'arêtes

Maze after editing walls:

0	1	2
3	4	5
6	7	8

DFS:

DFS font le parcours DFS (parcours en profondeur) depuis le sommet de *start*, puis renvoient le résultat dès que *end* a été visité.

Solution found:

0	1	2
3	4	5
6	7	8

Complexité temporelle: $O(E + V)$

antecedents = [0, 1, 2, 0, 3, 5, 6, 4, 7]

orders = [0, 3, 4, 7, 8, -1, -1, -1, -1]

path: 0 -> 3 -> 4 -> 7 -> 8

BFS:

BFS font le parcours BFS (parcours en largeur) depuis le sommet de *start*, puis renvoient le résultat dès que *end* a été visité.

(Dijkstra):

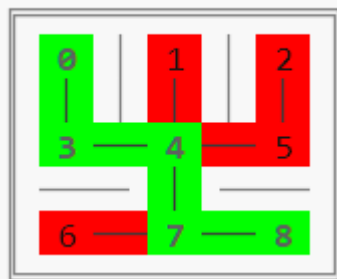
L'algorithme de Dijkstra prend en entrée un graphe pondéré, et un sommet de départ. Puis elle indique en sortie le chemin le plus court vers chaque sommet depuis le sommet de départ.

Elle utilise notamment une file de priorité comme stocker les prochains sommets à visiter, en fonction du poids du chemin depuis le sommet de départ.

Or dans notre labyrinthe, toutes les arêtes ont la même pondération, ie. poids d'un chemin = nombre d'arêtes dans le chemin.

Ainsi, appliquer l'algorithme de Dijkstra revient à appliquer un BFS pour nos labyrinthes. (Donc nous ne l'avons pas implémenté).

Solution found:



Complexité temporelle: $O(E + V)$

antecedents = [0, 4, 5, 0, 3, 4, 7, 4, 7]

orders = [0, 3, 4, 1, 5, 7, 2, 6, 8]

path: 0 -> 3 -> 4 -> 7 -> 8

A*:

L'algorithme A* prend en entrée un graphe pondéré, un sommet de départ, un sommet d'arrivée, ainsi que le choix d'une heuristique.

Cet algorithme ressemble à L'algorithme de Dijkstra, mais elles diffèrent notamment pour le calcul de pondération pour la file de priorité. Elle compare non seulement la longueur du chemin déjà parcouru pour un sommet, mais aussi une distance heuristique par rapport à *end*.

Le plus courant est d'utiliser comme distance heuristique:

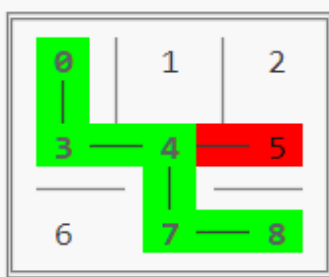
- distance vol d'oiseau (euclidienne): $\sqrt{(a.x - b.x)^2 + (a.y - b.y)^2}$
- distance de Manhattan: $|a.x - b.x| + |a.y - b.y|$
- distance de Manhattan multiplié par une constante: $k * (|a.x - b.x| + |a.y - b.y|)$

La distance vol d'oiseau correspond à la distance la plus naturelle.

La distance de Manhattan modélise parfaitement les labyrinthes, où l'on ne peut pas faire de déplacement diagonal.

Nous avons choisi la distance de Manhattan * 2 pour mettre plus d'importance à la distance heuristique qu'au poids du chemin déjà parcouru.

Solution found:



Complexité temporelle: $O((E + V) * \log(V))$

antecedents = [0, 1, 2, 0, 3, 4, 6, 4, 7]

orders = [0, 3, 4, 5, 7, 8, -1, -1, -1]

LeftHand / RightHand:

Cet algorithme résout le labyrinthe en longeant tout au long du mur de droite. (C'est un DFS ayant un ordre particulier sur l'ordre de visite des successeurs.)

Pour établir un sens d'orientation sur les sommets d'un labyrinthe, on utilise la différence d'id de 2 sommets u et v , notons ui l'index du sommet u , vi l'index du sommet v , et c le nombre de colonnes.

Si:

- $ui + 1 = vi$: v est à droite de u
- $ui - 1 = vi$: v est à gauche de u
- $ui + c = vi$: v est en bas de u
- $ui - c = vi$: v est en haut de u

On stocke ces valeurs indiquant les directions dans un tableau $\text{int}[4]$ $\text{directions} = \{1; c; -1; -c\}$.

Et notons di l'index de direction, $\text{directions}[di]$ indique la valeur correspondant la direction di :

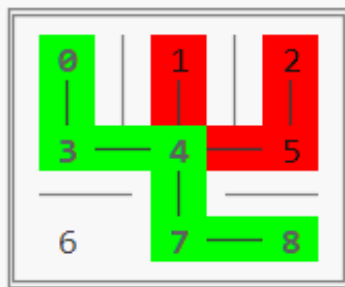
- direction droite: $di = 0$; $\text{directions}[di] = 1$
- direction droite: $di = 1$; $\text{directions}[di] = c$
- direction droite: $di = 2$; $\text{directions}[di] = -1$
- direction droite: $di = 3$; $\text{directions}[di] = -c$

À chaque visite de sommet, la direction lors du début de la visite de ce sommet correspond à la direction opposée à la direction d'entrée vers ce sommet.

Une fois le direction d'entrée reconnue, on ajoute successivement les sommets correspondants aux 3 autres directions, dans l'ordre inverse dans laquelle nous allons les visiter (on utilise une structure de pile). Pour passer à la prochaine direction à droite, il suffit d'incrémenter di de 1.

Nous utilisons aussi un tableau d'entier indiquant si un sommet a déjà été visité, afin d'éviter de revisiter en boucle.

Solution found:



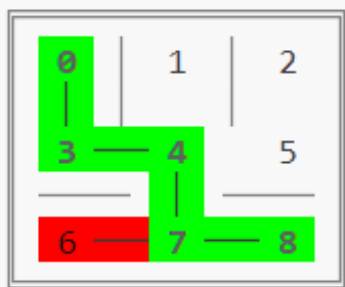
Left hand

Complexité temporelle: $O(E + V)$

$\text{antecedents} = [0, 4, 5, 0, 3, 4, 6, 4, 7]$

$\text{orders} = [0, 3, 4, 1, 5, 2, 7, 8, -1]$

Solution found:



Right hand

Complexité temporelle: $O(E + V)$

$\text{antecedents} = [0, 1, 2, 0, 3, 5, 7, 4, 7]$

$\text{orders} = [0, 3, 4, 7, 6, 8, -1, -1, -1]$

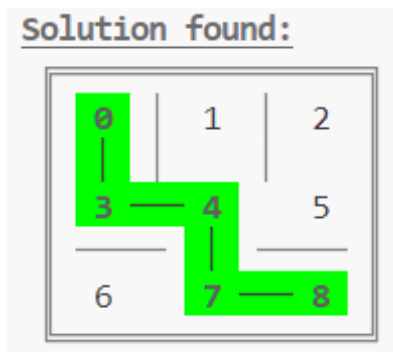
A*2:

A* est efficace, car il prend en compte la distance entre le sommet de départ et le sommet d'arrivée. Toutefois, pour certains cas particuliers de labyrinthe, cette distance est trompeuse.

C'est la raison pour laquelle nous avons implémenté A*2, pour résoudre ce genre de cas particulier.

L'idée est de redéfinir le point d'arrivée de A* dans les cas particuliers pour avoir une distance heuristique plus pertinente.

L'algorithme A*2 part tout d'abord depuis *end*, puis visite successivement les sommets accessibles, et ce, tant qu'il n'y a pas de d'intersection. Lorsque l'on croise une intersection, on lance A* de *start* vers le premier point d'intersection rencontré. Enfin il suffit de combiner les 2 parcours depuis *end* et depuis *start* afin d'obtenir le résultat final.



Complexité temporelle: $O((E + V) * \log(V))$

antecedents = [0, 1, 2, 0, 3, 5, 6, 4, 7]

orders = [0, 3, 4, 7, 8, -1, -1, -1, -1]

5.4. L'application en ligne de commande (CLI)

Lorsqu'on lance le MainCLI, nous avons plusieurs options: celui de charger un labyrinthe pré-téléchargé ou bien de générer un nouveau labyrinthe.

```
PROJET CYNAPSE

MENU
1 - Generate a maze
2 - Load a maze from a file
Choose an option: 
```

Si on choisit de générer un labyrinthe, le terminal nous demande de choisir parmi quatre algorithmes de génération. Après avoir choisi l'algorithme de génération souhaité, le labyrinthe est généré avec des informations supplémentaires telles que le nom de l'algorithme choisi et le temps nécessaire à la génération.

```
Choose an option: 1
Enter the number of rows:
10
Enter the number of columns:
10
Enter a seed (or 0 for random):
10
How would you like to generate it?
1 - Prim
2 - Kruskal
3 - RNG_DFS
4 - IMPERFECT
```

```
End of PRIM generation
Timestamp : 33ms

Generated Maze:


|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  |
| 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 |
| 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 |
| 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 |
| 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 |
| 60 | 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 |
| 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 |
| 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 |
| 90 | 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99 |


```

Après avoir généré ce labyrinthe, nous pouvons choisir d'éditer les murs du labyrinthe. L'exemple suivant concerne l'addition du mur entre les cases 66-76 puis la suppression du mur entre les cases 66-76.

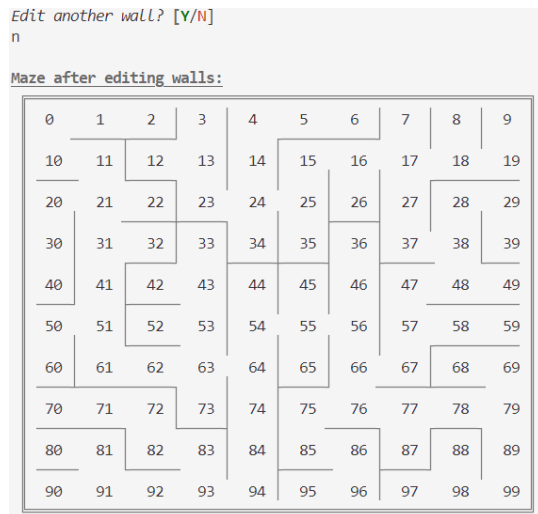
```
Edit another wall? [Y/N]
y
Enter two adjacent cell IDs to add/remove a wall between them.
First cell ID: 66
Second cell ID: 76
Wall added between 66 and 76
```

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

```
Edit another wall? [Y/N]
y
Enter two adjacent cell IDs to add/remove a wall between them.
First cell ID: 66
Second cell ID: 76
Wall removed between 66 and 76
```

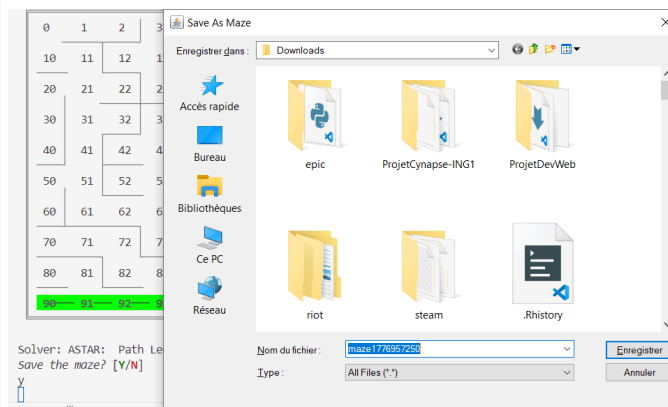
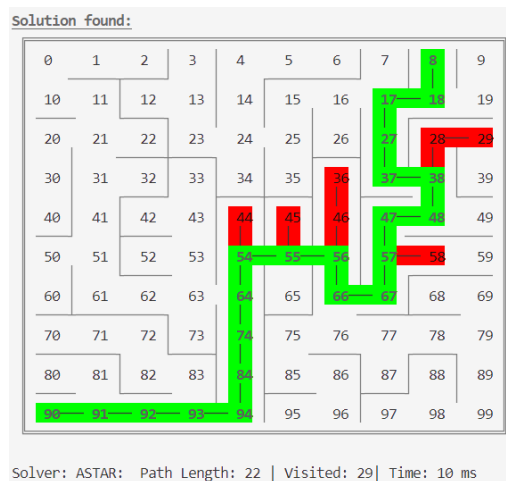
0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

Une fois toutes les modifications faites, nous pouvons interagir avec “N”, ce qui nous affiche la dernière version du labyrinthe. Nous pouvons personnaliser les points de départ et d’arrivée.

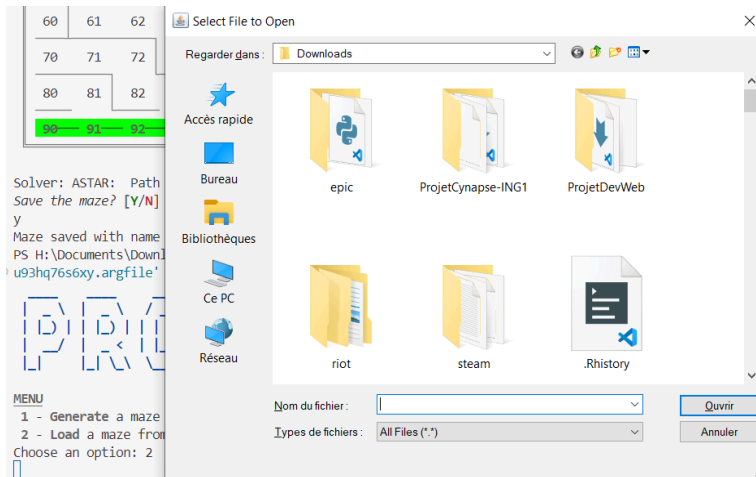


```
Starting point :
90
Ending point :
8
Solve the maze? [Y/N]
y
How would you like to solve it?
1 - ASTAR
2 - BFS
3 - DFS
4 - RIGHTHAND
5 - LEFTHAND
6 - ASTARII
1
Solver: ASTAR
```

Une fois la méthode de solution sélectionnée, le labyrinthe se résout et renvoie les solutions à l'utilisateur. Il affiche de plus le nom de la méthode de solution choisie, la longueur du chemin de la solution, le nombre de cases visitées, ainsi que le temps passé à résoudre. Par la suite, on nous propose de sauvegarder le labyrinthe généré: une nouvelle fenêtre s'affiche, et nous permet d'enregistrer le labyrinthe sous format .ser.



Si nous lançons le programme de nouveau et décidons de charger un labyrinthe, une nouvelle fenêtre apparaît ce qui nous permet de charger un labyrinthe depuis nos dossiers.



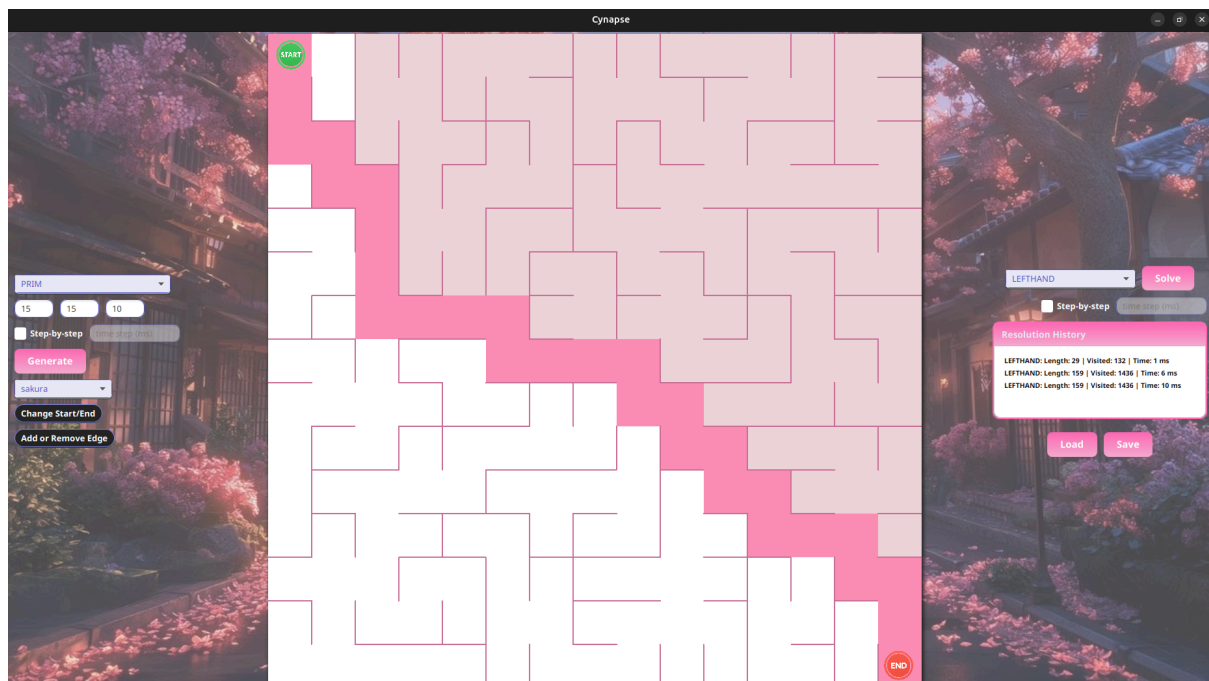
MENU
1 - Generate a maze
2 - Load a maze from a file
Choose an option: 2
Selected file : H:\Documents\Downloads\maze1776957250.ser
Maze Loaded

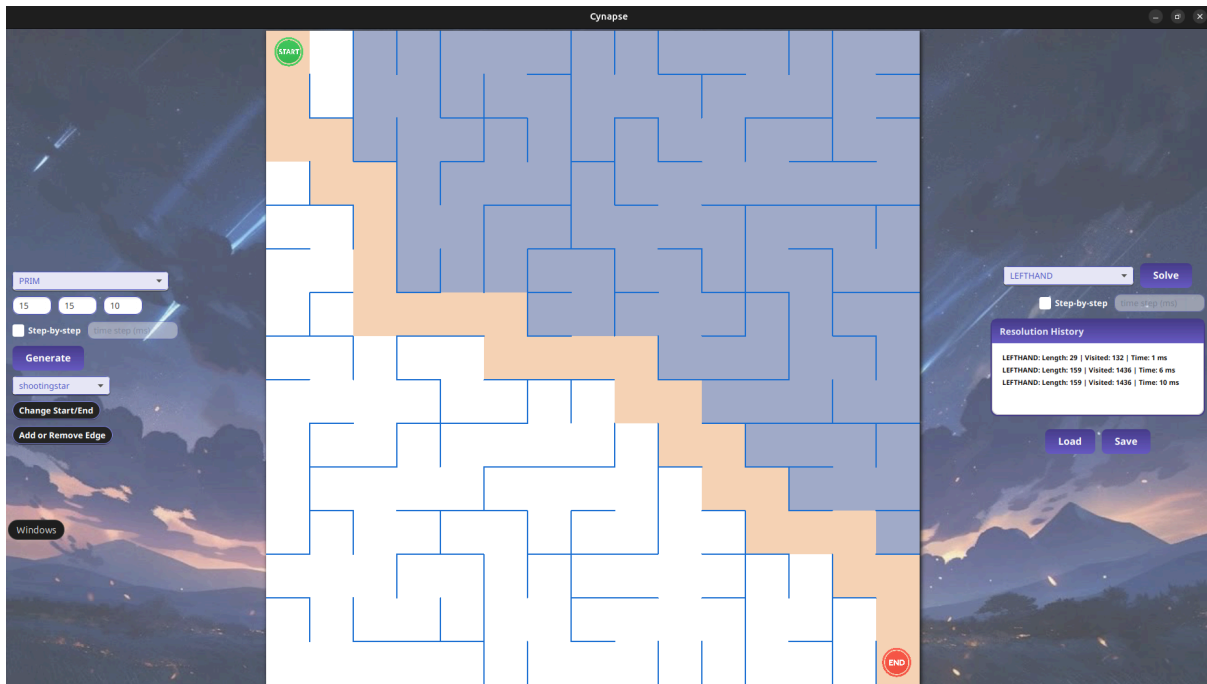
0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

5.5. JavaFX

JavaFX est un framework qui permet de créer l'interface graphique de notre application. L'utilisateur peut interagir avec cette interface, afin de jouer entre les différents algorithmes de génération et de résolution de labyrinthe et de créer un labyrinthe avec des dimensions et méthodes spécifiques. L'utilisateur peut modifier les labyrinthes qu'il crée, mais aussi de personnaliser l'affichage de l'interface.

L'interface repose sur un conteneur principal de type StackPane qui superpose plusieurs éléments graphiques, notamment une image de fond personnalisable et d'un canvas, sur lequel le labyrinthe est dessiné de manière dynamique. Ce choix technique permet de combiner un affichage esthétique avec une zone de dessin claire et réactive. L'utilisateur peut sélectionner les méthodes de génération et de résolution via des menus déroulants, paramétrer les dimensions du labyrinthe (nombre de lignes et colonnes), définir la graine aléatoire, et, s'il le souhaite, régler le délai d'animation, avant de lancer les opérations via des boutons dédiés. Plusieurs fonctionnalités avancées complètent l'interface, comme la sélection et la modification des points de départ et d'arrivée, l'édition manuelle des murs du labyrinthe, la visualisation pas à pas des phases de génération ou de résolution avec un contrôle du temps d'animation, la possibilité de charger et sauvegarder des labyrinthes, ainsi que la personnalisation de l'image en arrière-plan parmi plusieurs thèmes prédéfinis ou bien une image chargée depuis ses fichiers locaux, ce qui améliore l'esthétique de cette affichage, ainsi que le confort de l'utilisateur.





L'architecture de cette partie repose sur le contrôleur FXController qui gère l'ensemble des interactions entre la vue définie dans le fichier FXML et la logique métier contenue dans MazeController, responsable de la création et de la résolution des labyrinthes. Dès l'initialisation, tous les éléments de l'interface sont liés aux données et aux actions via le fichier FXML et son contrôleur. Les menus déroulants sont automatiquement remplis à partir des énumérations définies dans la logique métier, garantissant la cohérence entre l'interface et les fonctionnalités disponibles. Chaque interaction de l'utilisateur, telle qu'un clic de bouton, d'une sélection dans un menu ou d'une action sur le canvas, est capturée par le contrôleur, qui appelle alors les méthodes adéquates de MazeController pour effectuer les opérations nécessaires. L'affichage est ensuite mise à jour pour refléter le nouvel état du labyrinthe.

Le rendu dynamique s'effectue sur le canvas, qui affiche les murs, les chemins, ainsi que les points de départ et d'arrivée, représentés graphiquement avec des couleurs et icônes spécifiques afin de faciliter la compréhension visuelle et d'offrir un retour clair à l'utilisateur sur l'état du labyrinthe à chaque étape.

D'un point de vue technique, l'application utilise des threads distincts pour gérer la génération et la résolution du labyrinthe, assurant une exécution animée sans blocage de l'interface utilisateur. Cette gestion multi-threadée garantit une visualisation fluide et progressive des algorithmes, offrant ainsi une expérience agréable et réactive pour l'utilisateur. La modification du labyrinthe est également interactive : l'utilisateur peut sélectionner deux cellules adjacentes pour ajouter ou supprimer un mur, modifiant ainsi immédiatement la structure en mémoire et entraînant une actualisation instantanée de l'affichage pour une rétroaction visuelle immédiate. Chaque cellule possède un état qui détermine sa couleur d'affichage: visité, faisant partie de la solution ou en cours d'ajout/suppression de mur, ce qui facilite la lecture et le suivi des algorithmes. Ces couleurs s'adaptent en fonction du fond d'écran également si c'est un des fonds d'écrans proposés par défaut.

Par ailleurs, il y a un historique qui affiche les informations clés lors de la résolution d'un labyrinthe, comme le nombre de cases du chemin final, le nombre de cases explorées et le temps nécessaire, permettant à l'utilisateur d'évaluer la performance des algorithmes.

6. Conclusion

Le projet CYNapse nous a permis de concevoir et de développer une application interactive et modulaire dédiée à la génération et à la résolution de labyrinthes, en s'appuyant sur une interface graphique intuitive réalisée avec JavaFX. Tout au long de ce projet, nous avons mobilisé des compétences variées allant de la modélisation algorithmique à l'architecture logicielle, en passant par l'intégration d'interfaces utilisateur et la gestion collaborative via GitHub.

La richesse fonctionnelle de l'application, notamment la diversité des algorithmes de génération (Kruskal, Prim, DFS, imperfect) et de résolution (BFS, DFS, A*, A*2, mur gauche/droite), ainsi que la possibilité d'interactions dynamiques avec les labyrinthes, témoigne de notre investissement collectif et de notre volonté d'aboutir à un produit complet et ergonomique.

Ce projet a été une véritable opportunité d'apprentissage technique et méthodologique, tant au niveau individuel que collectif. Il nous a permis de mettre en pratique les principes de programmation orientée objet, de gestion de projet, et de conception d'interfaces utilisateurs modernes. Il a également renforcé notre capacité à travailler en équipe, à documenter efficacement notre code, et à structurer nos développements de manière évolutive.

Enfin, CYNapse constitue une base solide pour de futures améliorations possibles : ajout de nouveaux algorithmes, mode multijoueur, export d'animations, ou encore déploiement web. Les perspectives d'évolution sont nombreuses, et nous sommes fiers de ce premier jalon technique accompli dans notre parcours d'ingénieur.