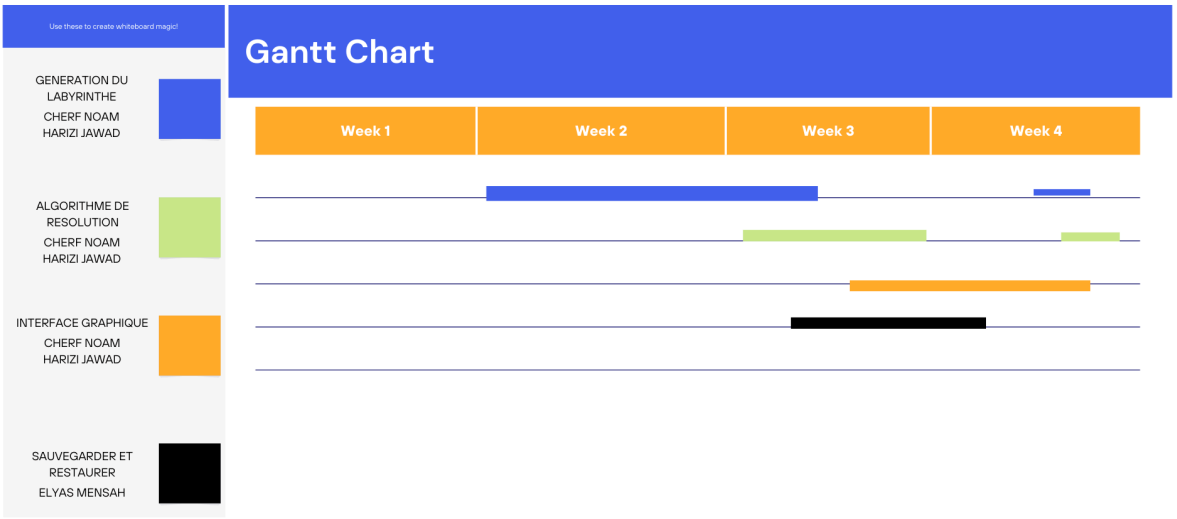
Année : 2024-2025
ING GI groupe 33

Rapport projet java Cynapse :

Diagramme de Gantt :



1. Présentation de l'équipe et du projet

Le projet Cynapse a pour objectif de concevoir une application capable de générer et de résoudre automatiquement des labyrinthes à l'aide de différents algorithmes, le tout en Java avec une interface graphique JavaFX. L'équipe est composée de :

- **Harizi Jawad** et **Cherf Noam** : responsables de la génération du labyrinthe, des algorithmes de résolution (Left, Right, DFS, BFS) et de l'interface graphique.

- **Elyas Mensah** : responsable de la sauvegarde et de la restauration du labyrinthe et création du Use Case.

Notre travail a été organisé autour d'un dépôt GitHub avec des commits réguliers. Des réunions de coordination hebdomadaires ont permis de répartir les tâches et de faire le point sur les difficultés rencontrées notamment grâce à Discord qui nous a permis de mieux organiser notre travail.

Nous avons aussi eu des réunions régulières avec notre tuteur qui nous a donné plusieurs idées qu'on a pu ajouter à notre code.

Déroulement

Nous avons rapidement entamé le projet avant les partiels, en développant un premier algorithme qui générait uniquement les cases et les murs sans créer un véritable labyrinthe.

Dans un premier temps, nous nous sommes concentrés sur la génération d'un labyrinthe parfait puis imparfait (l'algorithme de génération est détaillé plus bas). Concernant la génération pas à pas, nous avons décidé de la reporter à plus tard car à ce moment-là nous ne maîtrisions pas bien l'outil Timeline qui permet de "retarder" le code. Nous avons ainsi implémenté les algorithmes de résolution sans mode pas à pas pour les mêmes raisons. Au départ, ces codes faisait tous parti d'une seule et même classe Resolution.java mais après un rdv avec notre tuteur nous avons convenu de séparer chaque algorithme avec sa propre classe pour des soucis d'organisation et de visibilité et donc nous possédons une classe BFS.java, DFS.java, Right.java, Left.java qui héritent d'une classe Resolution.java .

Finalement, durant le temps restant, nous avons ajouté minutieusement ce qu'il manquait afin de compléter tout ce qui était demandé par le cahier des charges (nombre de cases traitées, nombre de cases dans le chemin représentant la solution, temps d'exécution, ajout d'un bouton reset, etc) et aussi une modification de l'interface pour la rendre plus responsive et plus belle, à l'aide d'un fichier CSS.

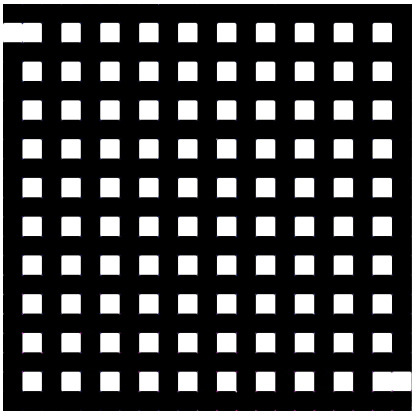
Algorithme de génération du labyrinthe

il faut d'abord différencier labyrinthes dits « parfaits » où chaque cellule est reliée à toutes les autres et, ce, de manière unique et les labyrinthes dits « imparfaits » qui sont tous les labyrinthes qui ne sont pas parfaits (ils peuvent donc contenir des boucles, des îlots ou des cellules inaccessibles).

L'algorithme utilise une propriété des labyrinthes parfaits précédemment énoncée telle quelle : *Chaque cellule est reliée à toutes les autres, et ce, de manière unique*. Il fonctionne en fusionnant progressivement des chemins depuis la simple cellule jusqu'à l'obtention d'un chemin unique, il suit donc une approche ascendante.

On représente notre labyrinthe par une grille[height][width].

L'algorithme associe une valeur unique à chaque cellule (leur numéro de séquence, par exemple) et part d'un labyrinthe où tous les murs sont fermés .



Ici chaque mur vaudra dans la grille -1 et on affecte une valeur dans l'ordre 1,2,3.. à chaque case blanche.

À chaque itération, on choisit un mur à ouvrir de manière aléatoire ie que l'on va changer la valeur du mur par la valeur de la case à gauche par exemple et on changera la valeur de la case de droite par cette même case de gauche de telle sorte que la valeur du mur cassé et des cases à gauche et à droite de ce mur soit les meme.

Lorsqu'un mur est alors ouvert entre deux cellules adjacentes, les deux cellules sont liées entre elles et forment un chemin.

Ensuite on continue sur l'ensemble des murs qui sont entre deux cases (donc pas ceux à des positions [paire][paire] comme on le voit dans la grille) chaque fois que l'on tente d'ouvrir un mur entre deux cellules, on vérifie que ces deux cellules ont des identifiants différents.

- Si les identifiants sont identiques, c'est que les deux cellules sont déjà reliées et appartiennent donc au même chemin. On ne peut donc pas ouvrir le mur.
- Si les identifiants sont différents, le mur est ouvert, et l'identifiant de la première cellule est affecté à toutes les cellules du second chemin.

Finalement, on obtient un chemin unique lorsque les valeurs de toutes les cases sont les mêmes.

Pour l'affectation des identifiants aux cellules, une variante plus efficace serait d'employer des structures d'Union-Find, qui permettent de diminuer la complexité algorithmique de cette opération mais étant donné que l'on ne maîtrise pas assez cela on a codé une fonction qui le fait mais avec plus d'itération :replaceCellValue(int oldId, int newId).

2. Problèmes rencontrés et solutions apportées

Génération du labyrinthe

Nous avons tout d'abord commencé par essayé nous même de représenter des labyrinthes à l'aide de matrice de boolean (True si il y a un mur et False sinon) mais cela nous a vite posé problème on a donc décidé d'utiliser finalement l'algorithme expliqué précédemment. De même que ce nouvel algorithme nous permet d'avoir une mazeGrid qui nous sera utile plus tard pour calculer les plus courts chemins à l'aide de distance associées aux cases.

Cependant nous avions eu au départ un problème avec ce nouvel algorithme car quand on remplace les valeurs de chaque côté du mur, nous ne remplacions que la case à gauche et à droite du mur sans celle a coté si il y'en a, ce qui ne permet pas à l'algorithme de finir correctement, c'est pourquoi nous avons mis en place replaceCellValue(int oldId, int newId).

Résolution du labyrinthe

Plusieurs algorithmes ont été implémentés :

BFS (Breadth-First Search) utilise une **file** pour explorer le labyrinthe en largeur. On part du point de départ et on visite les cases accessibles une à une, en priorité celles les plus proches. Cela garantit de trouver le **plus court chemin** jusqu'à l'arrivée. Chaque case est marquée avec sa distance depuis le départ.

DFS (Depth-First Search) utilise une **pile** et explore en profondeur. Il avance autant que possible dans une direction, puis revient en arrière en cas d'impasse. C'est rapide pour **explorer ou générer un labyrinthe**, mais le chemin trouvé n'est pas forcément le plus court.

Main gauche (Left) et **main droite (Right)** : Suivant le côté choisi, on colle notre main droite ou gauche au mur et on avance ainsi jusqu'à arriver au bout du labyrinthe.

Difficultés

Pour le DFS et le BFS nous n'avons pas rencontré de problème particulier. Mais nous avons eu du mal à avoir l'idée du dernier algorithme finalement on a décidé de reprendre l'algorithme le plus connu pour sortir d'un labyrinthe c'est à dire tout le temps coller le mur de droite jusqu'à arriver à la sortie l'algorithme de la main droite qu'on a su adapter à java. Nous avons aussi dû adapter le code puisque qu'au début on n'avait pas de message afin d'afficher que le labyrinthe n'avait pas de solution on a hésité à utiliser un message simple mais nous avons finalement choisi une alerte afin que le message soit plus visible.

Nous avons eu de même un problème pour la résolution par la droite car il tournait en boucle quand il y avait un îlot(un cycle) par exemple car il tournait en boucle autour de cet îlot. Pour régler ce problème, nous avons modifié le code de tel sorte qu'il ne passe pas deux fois par la même case .D'ailleurs pour cet algorithme il fallait changer notre fonction searchNeighbor et rajouter un paramètre prenant en compte la direction (nord sud est ouest).

Pour le temps d'exécution et l'animation pour le pas à pas. Nous avons pris du temps pour trouver la bonne méthode puis en cherchant nous avons pu adapter cette méthode.

On a aussi dû réduire la taille des boutons proportionnellement à la taille du labyrinthe, afin que pour des tailles trop grandes le labyrinthe ne dépasse pas sur les différents côtés.

Interface utilisateur

La synchronisation de l'interface graphique avec les animations des algorithmes de résolution a nécessité l'usage de Timeline de JavaFX. Il a fallu ajuster les délais et gérer les différents états du labyrinthe afin d'obtenir le mode pas à pas. De même qu'il a fallu trouver un rapport de proportionnalité entre le temps de génération et la taille du labyrinthe pour éviter qu'il soit trop lent ou trop rapide suivant sa hauteur et sa largeur.

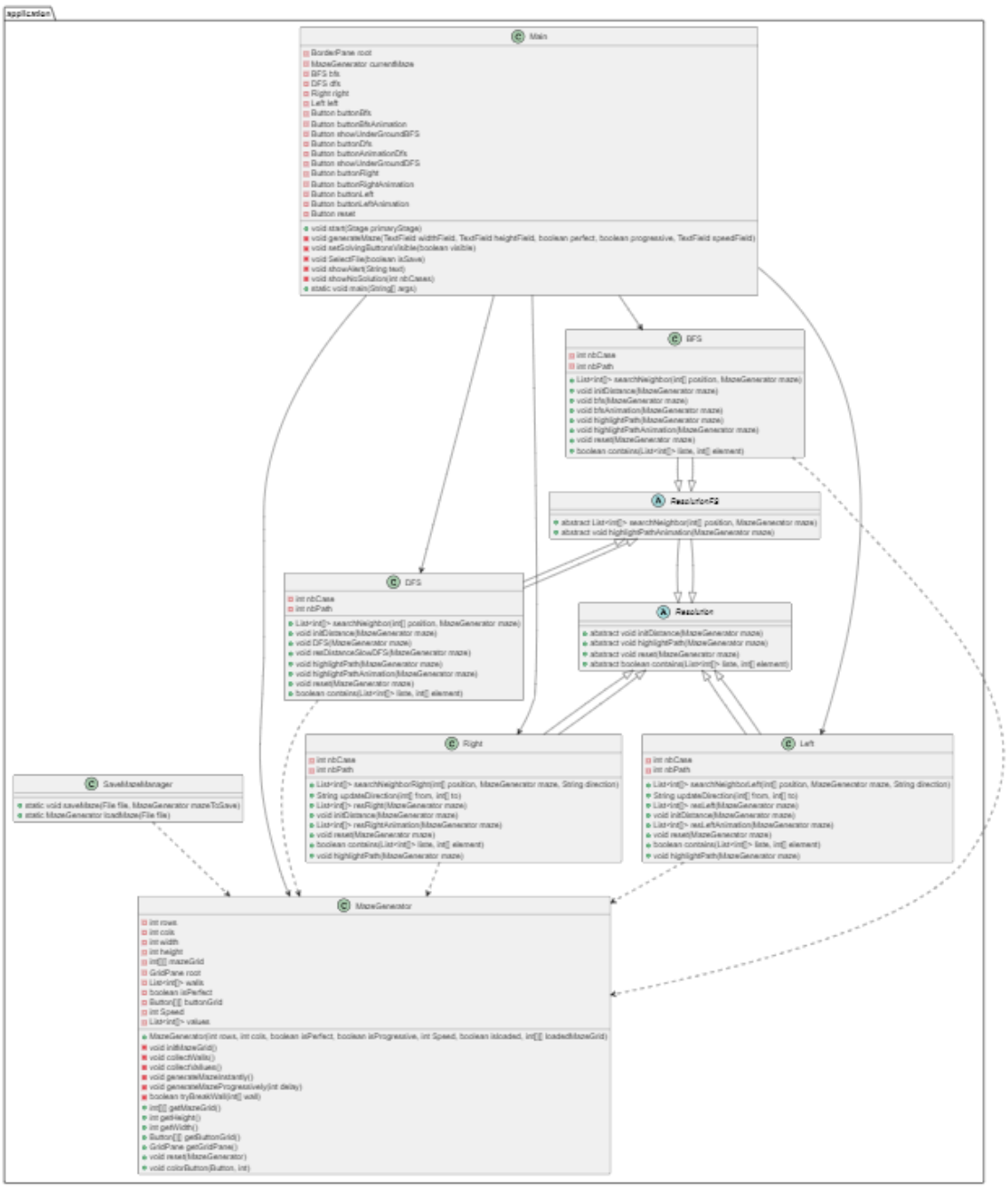
De plus on a changé notre façon de casser ou rajouter un mur car au début l'utilisateur devait rentrer les coordonnées de ce mur mais ce n'était pas optimal car il ne pouvait pas vraiment identifier le mur quand le labyrinthe était grand, ainsi on a changé les murs en boutons dans l'interface pour permettre à l'utilisateur de cliquer sur le mur pour le supprimer ou sur une case pour ajouter un mur.

Sauvegarde / restauration

Nous avons également intégré une fonctionnalité permettant de sauvegarder l'état actuel du labyrinthe et de le recharger et ce sur le disque dur de l'ordinateur (au début on pensait que c'était à travers l'application). Cela permet à l'utilisateur de reprendre une session précédente sans avoir à tout recommencer.

3. Diagrammes

Diagramme de classe réalisé par **Harizi Jawad** et **Cherf Noam :**



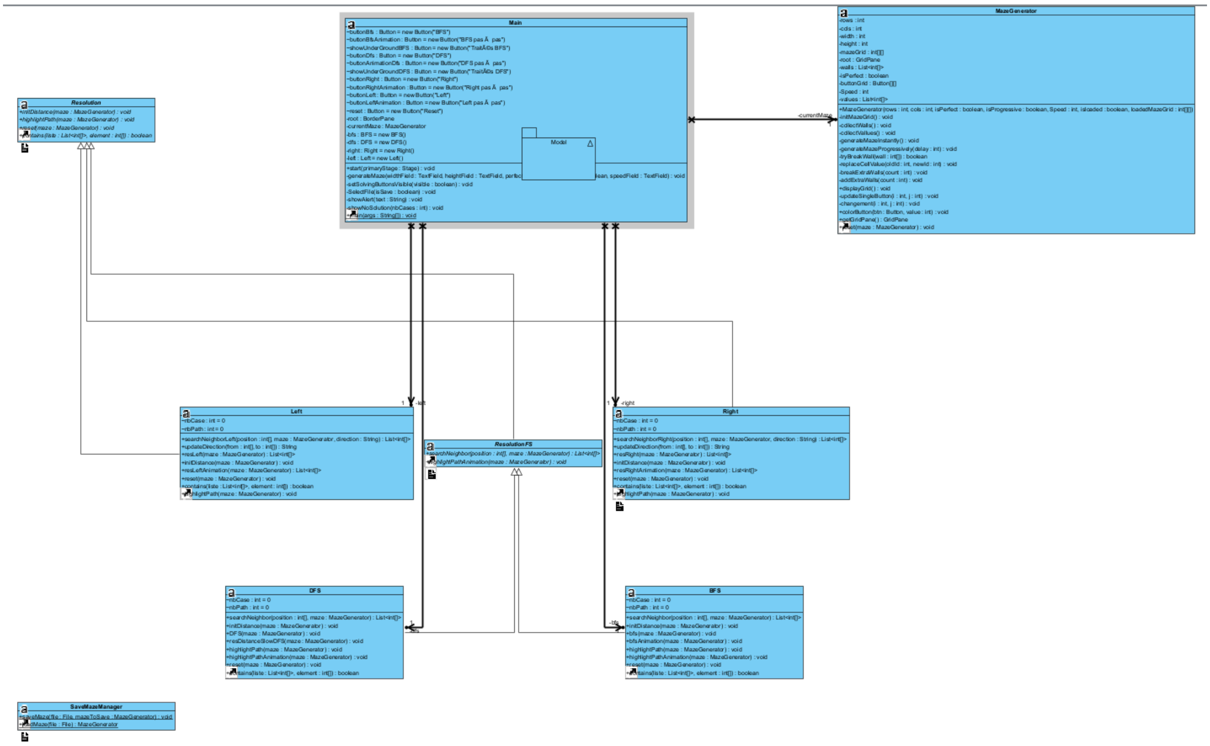

Main
~buttonBfs : Button = new Button("BFS")
~buttonBfsAnimation : Button = new Button("BFS pas Ã pas")
~showUnderGroundBFS : Button = new Button(" TraitÃ©s BFS")
~buttonDfs : Button = new Button("DFS")
~buttonAnimationDfs : Button = new Button("DFS pas Ã pas")
~showUnderGroundDFS : Button = new Button(" TraitÃ©s DFS")
~buttonRight : Button = new Button("Right")
~buttonRightAnimation : Button = new Button("Right pas Ã pas")
~buttonLeft : Button = new Button("Left")
~buttonLeftAnimation : Button = new Button("Left pas Ã pas")
~reset : Button = new Button("Reset")
-root : BorderPane
-currentMaze : MazeGenerator
-bfs : BFS = new BFS()
-dfs : DFS = new DFS()
-right : Right = new Right()
-left : Left = new Left()
+start(primaryStage : Stage) : void
-generateMaze(widthField : TextField, heightField : TextField, perfec
lean, speedField : TextField) : void
-setSolvingButtonsVisible(visible : boolean) : void
-SelectFile(isSave : boolean) : void
-showAlert(text : String) : void
-showNoSolution(nbCases : int) : void
ain(args : String[]) : void
Model
-currentMaze
1
MazeGenerator
-rows : int
-cols : int
-width : int
-height : int
-mazeGrid : int[][]
-root : GridPane
-walls : List<int[]>
-isPerfect : boolean
-buttonGrid : Button[][]
-Speed : int
-values : List<int[]>
+MazeGenerator(rows : int, cols : int, isPerfect : boolean, isProgressive : boolean, Speed : int, isloaded : boolean, loadedMazeGrid : int[][])
-initMazeGrid() : void
-collectWalls() : void
-collectValues() : void
-generateMazeInstantly() : void
-generateMazeProgressively(delay : int) : void
-tryBreakWall(wall : int[]) : boolean
-replaceCellValue(oldId : int, newId : int) : void
-breakExtraWalls(count : int) : void
-addExtraWalls(count : int) : void
+displayGrid() : void
-updateSingleButton(i : int, j : int) : void
-changement(i : int, j : int) : void
+colorButton(btn : Button, value : int) : void
+getGridPane() : GridPane
et(maze : MazeGenerator) : void
Resolution
+initDistance(maze : MazeGenerator) : void
+highlightPath(maze : MazeGenerator) : void
+reset(maze : MazeGenerator) : void
ntains(liste : List<int[]>, element : int[]) : boolean
1
-left
Left
~nbCase : int = 0
~nbPath : int = 0
+searchNeighborLeft(position : int[], maze : MazeGenerator, direction : String) : List<int[]>
+updateDirection(from : int[], to : int[]) : String
+resLeft(maze : MazeGenerator) : List<int[]>
+initDistance(maze : MazeGenerator) : void
+resLeftAnimation(maze : MazeGenerator) : List<int[]>
+reset(maze : MazeGenerator) : void
+contains(liste : List<int[]>, element : int[]) : boolean
hlightPath(maze : MazeGenerator) : void
ResolutionFS
+searchNeighbor(position : int[], maze : MazeGenerator) : List<int[]>
ghlightPathAnimation(maze : MazeGenerator) : void
1
-right
Right
~nbCase : int = 0
~nbPath : int = 0
+searchNeighborRight(position : int[], maze : MazeGenerator, direction : String) : List<int[]>
+updateDirection(from : int[], to : int[]) : String
+resRight(maze : MazeGenerator) : List<int[]>
+initDistance(maze : MazeGenerator) : void
+resRightAnimation(maze : MazeGenerator) : List<int[]>
+reset(maze : MazeGenerator) : void
+contains(liste : List<int[]>, element : int[]) : boolean
hlightPath(maze : MazeGenerator) : void
DFS
~nbCase : int = 0
~nbPath : int = 0
+searchNeighbor(position : int[], maze : MazeGenerator) : List<int[]>
+initDistance(maze : MazeGenerator) : void
+DFS(maze : MazeGenerator) : void
+resDistanceSlowDFS(maze : MazeGenerator) : void
+highlightPath(maze : MazeGenerator) : void
+highlightPathAnimation(maze : MazeGenerator) : void
+reset(maze : MazeGenerator) : void
tains(liste : List<int[]>, element : int[]) : boolean
1
-bfs
BFS
~nbCase : int = 0
~nbPath : int = 0
+searchNeighbor(position : int[], maze : MazeGenerator) : List<int[]>
+initDistance(maze : MazeGenerator) : void
+bfs(maze : MazeGenerator) : void
+bfsAnimation(maze : MazeGenerator) : void
+highlightPath(maze : MazeGenerator) : void
+highlightPathAnimation(maze : MazeGenerator) : void
+reset(maze : MazeGenerator) : void
tains(liste : List<int[]>, element : int[]) : boolean
SaveMazeManager
aveMaze(file : File, mazeToSave : MazeGenerator) : void
dMaze(file : File) : MazeGenerator

Diagramme UseCase réalisé par **Elyas MENSAH** et **Riyad Zalegh** :