Présentation de Connexe

Projet Java « CYnapse » — 2024-2025

Réalisé par Yani A., Mathis M., Saad T., Laura M., Victor G.

Dépôt GitHub : https://github.com/DynamicField/Connexe

Table des matières

Organisation de l'équipe 3
Problèmes et solutions 5
DFS au temps d'exécution quasi-infini 5
Git et fusion de *pull requests* 5
Labyrinthe imparfait, mais pas toujours 5
Windows, Gradle et accents : un trio embêtant 6
Limites fonctionnelles 7
Taille du labyrinthe 7
Nombre de joueurs 7
Planning de réalisation 8
Diagramme de classes 9
Diagramme de cas d'utilisation 10

Organisation de l'équipe

Avant d'écrire ne serait-ce qu'une ligne de code, nous avons commencé à réfléchir tous ensemble aux bases du projet : les classes de base (`GraphMaze`, `ArrayMaze` et `Point`), la représentation d'un graphe en mémoire, les cas d'utilisation…

Après avoir solidifié nos bases communes, nous avons décidé de travailler chacun de notre côté. Pour ce faire, nous avons séparé le projet en six modules distincts sur lesquels chacun peut travailler sans trop de friction :

Module	**Chargé du module**	**Assistant (revue de code)**
Résolution du labyrinthe	Mathis	Victor
Génération du labyrinthe	Yani	Victor
Modification du labyrinthe	Victor	Laura
Interface JavaFX	Laura	Yani
Interface sur terminal	Saad	Yani
Sauvegarde du labyrinthe	Saad	Yani

Afin d'éviter qu'un morceau de code ne soit compris que par une seule personne, chaque module est accompagné d'un assistant qui s'occupera d'effectuer une revue de code lors d'une *pull request* sur notre dépôt GitHub. Cet assistant devra relire intégralement le code et s'assurer qu'il est conforme à nos trois règles de code, imposées sur le projet entier :

1. Tous les commentaires et symboles du code doivent être en anglais
2. Le code des fonctions doit être suffisamment commenté, environ 1 commentaire par 5 lignes de code
3. Chaque fonction, classe et constructeur public doit comporter un commentaire Javadoc au format Markdown expliquant son fonctionnement (*getters*/*setters* simples exclus)

Ainsi, chaque ligne de code est systématiquement vue et comprise par deux personnes minimum.

Néanmoins, créer des *pull requests* sans crier gare n'est pas idéal. Pour communiquer efficacement sur ce que l'on fait et ce qui doit être fait, nous avons créé un serveur Discord sur lequel nous pouvons échanger sur n'importe quel sujet en rapport avec le projet. Ce serveur nous permet aussi de centraliser les différentes informations du projet : documents, images, ressources, consignes, etc.

Avec cette méthode de communication, il devient beaucoup plus simple de coopérer et d'utiliser chacun de nos modules. Par exemple, Laura était en étroite collaboration avec Mathis pour intégrer ses algorithmes de résolution dans l'interface JavaFX.

Cette organisation modulaire nous a permis d'avancer rapidement en réalisant en parallèle des modules *théoriquement* indépendants… Mais elle n'était pas parfaite pour autant. Certains modules étant intimement liés avec d'autres, il est parfois nécessaire d'attendre que l'un ait terminé sa partie pour pouvoir entamer la sienne.

C'est ce qu'il s'est produit avec le module « Modification du labyrinthe », qui a demandé à Victor d'attendre que Laura finisse de réaliser l'affichage du labyrinthe. Ensuite, il fallait

comprendre l'entièreté du code de Laura *et* le fonctionnement de JavaFX ; ce qui prend un temps non négligeable. La fonctionnalité a finalement dû être intégrée tardivement dans le projet.

Heureusement, les modules de génération, de résolution, de sauvegarde et d'interface ont pu évoluer sans accroc, en intégrant les autres modules aisément grâce à une séparation en branches sur le dépôt Git. Chaque module possède sa propre branche, ce qui permet de facilement récupérer les derniers ajouts d'un module en particulier.

Une fois le cahier des charges minimum rempli (trois algorithmes de résolution, modification du labyrinthe, pas à pas…), nous nous sommes mis à accomplir quelques bonus pour le plaisir de créer :

- Mathis a ajouté des algorithmes de résolution supplémentaires : main gauche et A^*
- Yani a ajouté un mode « arcade » permettant de résoudre le labyrinthe soi-même et en multi-joueur (jusqu'à 8 joueurs), avec trois modes de jeux et prise en charge des manettes

Problèmes et solutions

DFS au temps d'exécution quasi-infini

Lors de la réalisation du projet, nous avons pu nous rendre compte d'un problème avec notre implémentation initiale de la résolution par DFS.

La première implémentation de la fonction de résolution du labyrinthe `solveDFS` faisait (comme son nom l'indique) un parcours en DFS pour chercher le chemin le plus court vers la fin. Cette fonction devait donc parcourir tous les chemins de la case de début à celle de fin. Comme il s'agit d'un DFS, l'algorithme parcourt tous les chemins possibles de A vers n'importe quelle sommet.

Pour un labyrinthe parfait, ce n'est pas un problème : il n'existe qu'un seul chemin de A vers B, et l'algorithme se termine rapidement, sa complexité étant $O(n)$ dans le meilleur cas. Dès qu'il y a des cycles dans le labyrinthe, le nombre de chemins possibles grandit exponentiellement, on atteint $O(n!)$ dans le pire cas[1] ! L'algorithme se met à tourner indéfiniment pour des labyrinthes imparfaits de taille 20x20, et même 15x15.

Pour régler ce problème, nous avons fait en sorte que la fonction ne peut pas repasser par des nœuds déjà visités changeant donc son but de trouver le chemin le plus court à trouver un chemin quelconque vers la fin. On passe alors d'une complexité de $O(n!)$ à $O(n)$.

Git et fusion de *pull requests*

Un problème a eu lieu avec une la première pull request de la modification du labyrinthe qui à cause d'un problème avec le *merge* de la branche main s'est retrouvé avec 94 commits ! Il a alors fallu *rebase* le main et ajouter tous les changements copiés au préalable en un seul commit.

Enfin, à la fin du projet, la réunion de toutes les branches des différentes *Pull Requests* a ajouté de nouveaux bugs et nous a donc demandé de fabriquer quelques « gardes-fous ». Par exemple, on ne peut pas avoir le parcours en pas à pas du labyrinthe immédiatement après l'avoir modifié… Chose que l'on ne pouvait pas prévoir avant de fusionner nos changements.

De plus, l'action de *merge* a souvent demandé de résoudre des conflits, ce qui a subtilement introduit des bugs en remplaçant du code déjà présent avant le *merge*. Quelques correctifs sur le code nous a permis de résoudre ces problèmes apparus soudainement, mais surtout discrètement.

Labyrinthe imparfait, mais pas toujours

Pour générer des labyrinthes imparfaits, la solution la plus simple et de prendre un labyrinthe parfait généré au préalable, et de le rendre imparfait. Pour qu'un labyrinthe soit imparfait, il faut que son graphe des chemins remplisse une de ces conditions :

---

[1] https://www.baeldung.com/cs/simple-paths-between-two-vertices#3-complexity ; n est le nombre de sommets

- il n'est pas connexe (cases ou ilots isolés et inaccessibles)
- il contient un cycle (il existe un chemin pour aller de A vers A)

Pour créer un cycle, c'est facile : il suffit de connecter deux sommets quelconques (on retire un mur). Pour isoler des cases, c'est facile : il suffit de déconnecter deux sommets quelconques (on crée un mur).

Idéalement, nous voulons avoir des labyrinthes imparfaits intéressants, avec des cases inaccessibles *et* des cycles. Alors, naturellement, l'idée est venue de ajouter *et* supprimer des arêtes au hasard du labyrinthe. On a appelé ce procédé « injonction de chaos ».

Sauf que, dans certaines conditions, ajouter puis retirer des arêtes peut produire un labyrinthe parfait ! Si on ajoute une arête pour former un cycle, puis qu'on retire une arête contenue dans ce cycle… Le labyrinthe est à nouveau parfait !

La solution que nous avons choisie est la plus simple : recommencer l'injonction de chaos jusqu'à ce que le labyrinthe soit parfait. Les probabilités pour qu'un labyrinthe demeure parfait après l'injonction sont suffisamment basses pour que l'on ne risque pas une boucle infinie. La vérification de perfection du labyrinthe repose sur deux DFS (une pour la connexité, une autre pour l'absence de cycles), soit une complexité plus que raisonnable.

Windows, Gradle et accents : un trio embêtant

Durant le développement de l'interface terminal sous Windows, nous avons été confrontés à un bug assez pénible : tous les accents (`é`, `à`, etc.) s'affichaient avec des symboles illisibles dans le terminal… Ce qui fait que « Générer un labyrinthe » s'affiche en « G►n►rer un labyrinthe », pas très agréable à voir.

Pour une raison assez étrange, Gradle force la sortie de `System.out` à utiliser un encodage arbitraire, différent de l'UTF-8. La solution était simple mais terriblement difficile à trouver : ajouter les arguments `-Dstdout.encoding=UTF-8 -Dstderr.encoding=UTF-8` lorsqu'on lance le programme.

Malheureusement, ce n'est pas suffisant : Gradle ne configure pas la sortie terminal à utiliser l'UTF-8. Après quelques recherches, on a découvert qu'il fallait activer une option cachée dans Windows : « Utiliser le format Unicode UTF-8 » dans les paramètres. Depuis, plus de souci !

Nous avons aussi modifié le fichier `gradlew.bat` pour qu'il contienne la commande `chcp 65001` qui permet d'utiliser l'encodage UTF-8 sur le terminal.

Limites fonctionnelles

Taille du labyrinthe

Nous avons décidé de limiter la taille maximum d'un labyrinthe à 50x50 (50 lignes, 50 colonnes), soit 2500 cases au total. Un labyrinthe beaucoup trop grand mettraient nos algorithmes à rude épreuve, prenant plusieurs minutes, voire plusieurs heures à s'exécuter ; un temps d'attente intolérable !

Pourtant, ce n'est pas pour cette raison que nous avons limité la taille à 50x50 ; c'est surtout une question d'interface utilisateur. Comment afficher un labyrinthe de 1000 cases sur un écran de 1080 pixels de haut ? Nous aurions pu implémenter un système de zoom et de déplacement du labyrinthe, mais l'application perdrait beaucoup de son intérêt si on ne voit qu'un morceau du labyrinthe sur l'écran… Une taille maximale de 50x50 nous parait suffisante pour être vue en entier sur l'écran, tout en étant difficile à résoudre soi-même, bien sûr !

Nombre de joueurs

Pour le mode Arcade, où plusieurs joueurs peuvent essayer de résoudre le labyrinthe en même temps, nous avons choisi de limiter le nombre de joueurs à 8. Encore une fois, la raison est plus esthétique que technique : l'interface deviendrait surchargée de joueurs, et puis, qui arriverait à brancher six manettes sur un PC ?

Planning de réalisation

Connexe

A. Yani, G. Victor, M. Mathis, M. Laura, T. Saad

Début du projet : mer, 23/04/2025

en cours | en pause

TÂCHE	ATTRIBUÉE À	DÉBUT	FIN	JOURS
Planification & conceptualisation				
Brainstorming et planification des classes / diagrammes UML	Tous	23/04/25	11/05/25	
Génération du labyrinthe				
Stockage et représentation du labyrinthe	Yani	11/05/25	11/05/25	
Génération labyrinthes parfaits Prim & DFS	Yani	12/05/25	15/05/25	
Génération labyrinthes imparfaits	Yani	13/05/25	15/05/25	
Résolution du labyrinthe				
DFS complet et pas à pas	Mathis	11/05/25	21/05/25	
Dijkstra complet et pas à pas	Mathis	13/05/25	16/05/25	
Méthode horaire complet et pas à pas	Mathis	14/05/25	19/05/25	
Méthode de la main gauche complet et pas à pas	Mathis	19/05/25	21/05/25	
Algorithme A* complet et pas à pas	Mathis	22/05/25	23/05/25	
Interface JavaFX & Console				
Menu général et ses options	Laura	11/05/25	15/05/25	
Menu génération d'un labyrinthe + affichage du labyrinthe et animation pas à pas	Laura	12/05/25	17/05/25	
Menu résolution + affichage de la résolution complète + animation pas à pas	Laura	17/05/25	23/05/25	
Mode modification du labyrinthe	Victor	16/05/25	23/05/25	
Sérialisation du labyrinthe (sauvegarde/ouverture)	Saad	11/05/25	11/05/25	
Mode Console	Saad	18/05/25	23/05/25	
Bonus				
Mode arcade + son interface	Yani	20/05/25	23/05/25	

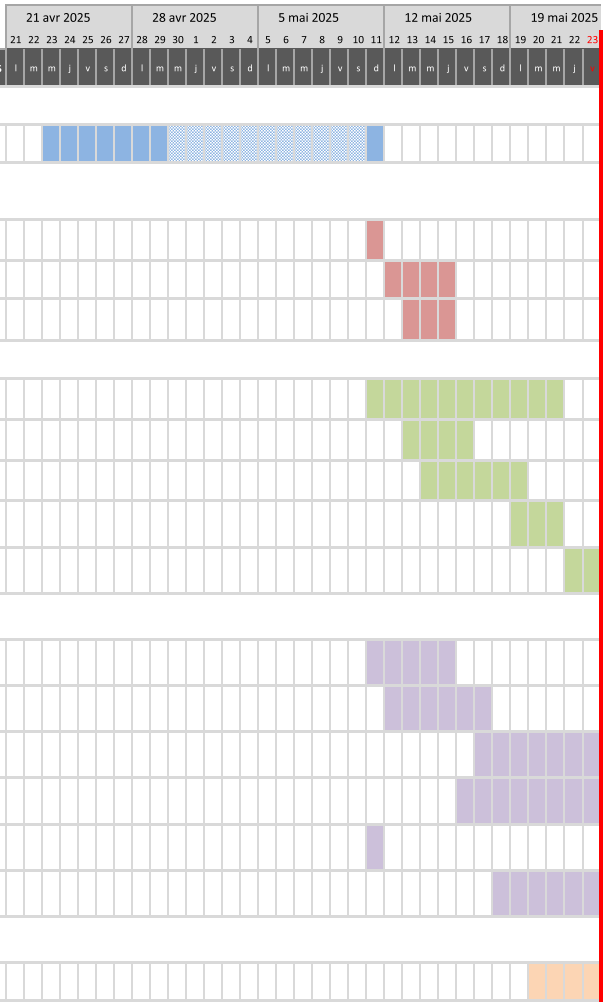

Diagramme de classes

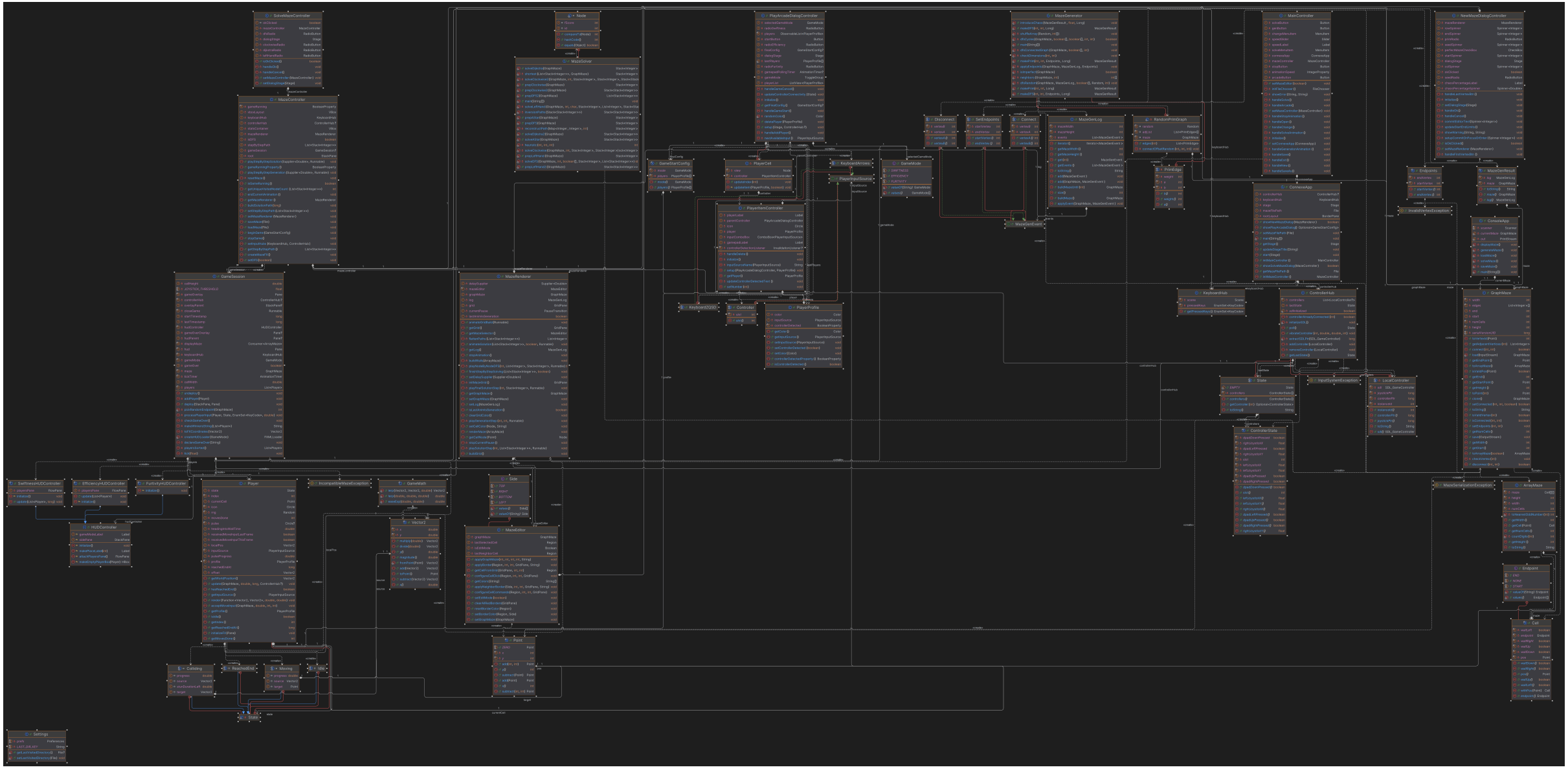
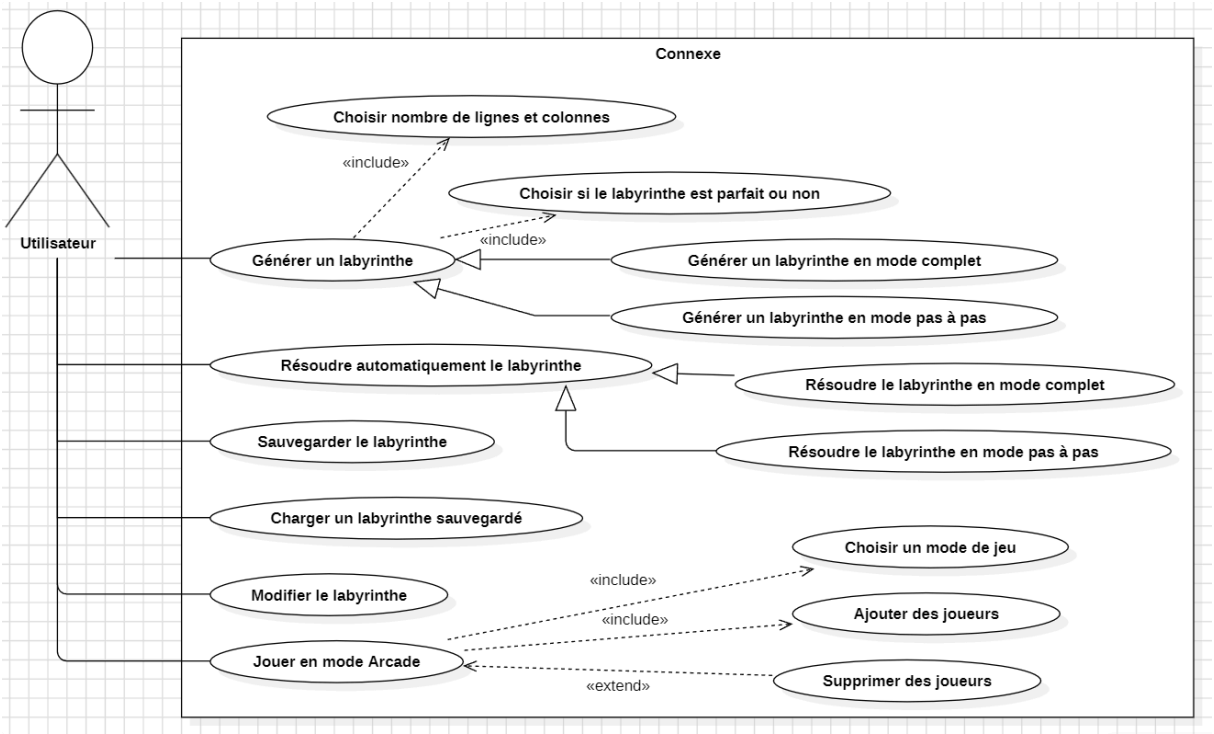
Diagramme de cas d’utilisation