

# Cahier des charges



Génération et résolution de labyrinthes procéduraux

Projet développé en langage Rust

**Membres de l'équipe :**

Mehdi Turki, Ulysse Toussaint, Jimmy Ngome Ongono, Wali Brochard

February 24, 2025

# Contents

<b>1</b>	<b>Le projet</b>	<b>2</b>
1.1	Qui sommes-nous et quel est le projet ? . . . . .	2
<b>2</b>	<b>Gestion du projet</b>	<b>2</b>
2.1	Présentation des membres . . . . .	2
2.1.1	Mehdi Turki . . . . .	2
2.1.2	Ulysse Toussaint . . . . .	2
2.1.3	Jimmy Ngome Ongono . . . . .	2
2.1.4	Wali Brochard . . . . .	2
2.2	Tableau d'avancement . . . . .	2
2.2.1	répartition des tâches . . . . .	2
<b>3</b>	<b>Objet de l'étude</b>	<b>5</b>
3.1	Qu'est-ce que le projet peut nous apporter ? . . . . .	5
<b>4</b>	<b>Détails du projet</b>	<b>5</b>
4.1	Librairie dédiée . . . . .	5
4.1.1	Importance d'une librairie . . . . .	5
4.1.2	Structure de la librairie . . . . .	5
4.1.3	Implémentation Pratique . . . . .	5
4.2	Interface utilisateur et Site Web . . . . .	6
4.3	Génération du niveau de manière procédurale . . . . .	6
4.3.1	Principe de la génération procédurale . . . . .	6
4.3.2	Structure de données pour le labyrinthe . . . . .	7
4.3.3	Algorithmes de génération . . . . .	7
4.3.4	Personnalisation des labyrinthes . . . . .	7
4.3.5	Validation du labyrinthe . . . . .	8
4.3.6	Optimisation et performance . . . . .	8
4.4	Résolution du niveau . . . . .	8
<b>5</b>	<b>Les limites techniques du projet</b>	<b>8</b>
5.1	Complexité des algorithmes . . . . .	8
5.1.1	Génération de labyrinthes . . . . .	8
5.1.2	Résolution de labyrinthes . . . . .	9
5.1.3	Limitations supplémentaires . . . . .	9
5.2	Performance et gestion de la mémoire . . . . .	9
5.3	Modularité et intégration de la librairie . . . . .	9
5.3.1	Modularité des composants . . . . .	9
5.3.2	Intégration dans le projet global . . . . .	10
5.3.3	Gestion des dépendances externes . . . . .	10
5.3.4	Avantages et limitations de la modularité . . . . .	10
5.4	Interface utilisateur (UI) . . . . .	10
5.4.1	Intégration de la logique métier dans l'interface . . . . .	11
5.4.2	Limitations des bibliothèques disponibles en Rust pour les interfaces graphiques . . . . .	11
5.4.3	Représentation et interactivité de l'interface . . . . .	11
5.4.4	Solutions potentielles aux limitations . . . . .	11
5.5	Tests et validation . . . . .	12
5.6	Visualisation des résultats . . . . .	12
<b>6</b>	<b>Conclusion</b>	<b>13</b>

# 1 Le projet

## 1.1 Qui sommes-nous et quel est le projet ?

Dans le cadre de notre S4, nous réalisons un projet de développement d'application. Pour des raisons pratiques, les élèves d'EPITA Strasbourg qui restaient en France ont été amenés à poursuivre leurs études à Lyon. C'est ainsi que Jimmy et Wali se sont joints à Mehdi et Ulysse pour former le groupe Mazemaster.

L'objectif de ce nouveau groupe est très simple : développer une application capable de générer de manière procédurale un labyrinthe pour ensuite le résoudre sous les yeux de nos chers utilisateurs. Le but est simple : attirer l'œil sur des mécaniques bien trop méconnues du grand public (ici principalement la génération de mondes procéduraux comme dans Minecraft).

## 2 Gestion du projet

Ulysse fait la gestion de projet.

### 2.1 Présentation des membres

#### 2.1.1 Mehdi Turki

Depuis tout petit, j'ai toujours été curieux du monde qui m'entourait. En grandissant, je me suis rendu compte que l'informatique prenait beaucoup de place dans ce dernier. Je m'y suis donc intéressé et je me suis réellement lancé dans la programmation avec EPITA. Ce projet me permet de découvrir l'importance d'un réseau, une facette de l'informatique que j'appréhendais beaucoup. Ce projet va me permettre de perfectionner mes compétences en Rust, plus précisément dans l'élaboration et la résolution d'algorithmes.

#### 2.1.2 Ulysse Toussaint

J'ai choisi de me former au développement informatique car je souhaitais repousser les limitations rencontrées lors de l'utilisation d'applications. Initialement autodidacte, j'ai participé à divers projets pour renforcer mes compétences. Par la suite, j'ai consacré mon temps libre au télétravail avec plusieurs entreprises avant de rejoindre EPITA pour affiner mes connaissances.

#### 2.1.3 Jimmy Ngome Ongono

J'ai grandi avec des films de sciences fictions et comics de super-héros qui ont emplis mon esprit de rêves. Iron Man, Pacific Rim et d'autres encore ont forgé mon attrait pour les sciences et tout particulièrement pour l'informatique. Aujourd'hui, j'ai la chance de pouvoir travailler avec des personnes qui possèdent les mêmes intérêts que moi. Après délibération nous avons décidé que je me chargerai de l'interface utilisateur, ayant l'année dernière déjà, travaillé sur une tâche similaire pour le projet de jeux vidéo.

#### 2.1.4 Wali Brochard

J'ai toujours suscité un certain intérêt pour la résolution de problèmes et la logique. Cet intérêt m'a naturellement conduit à m'intéresser à l'informatique et à la programmation. Dans le cadre de ce projet, je suis en charge de la génération des labyrinthes. Cela va me permettre d'approfondir mes compétences en algorithmes et en programmation procédurale, tout en découvrant de nouvelles facettes du langage Rust.

### 2.2 Tableau d'avancement

#### 2.2.1 répartition des tâches

- **Ngome Ongono Jimmy** : Développement de l'interface et du site web
- **Brochard Wali** : Génération du niveau
- **Toussaint Ulysse** : Génération du niveau
- **Turki Mehdi** : Résolution du niveau

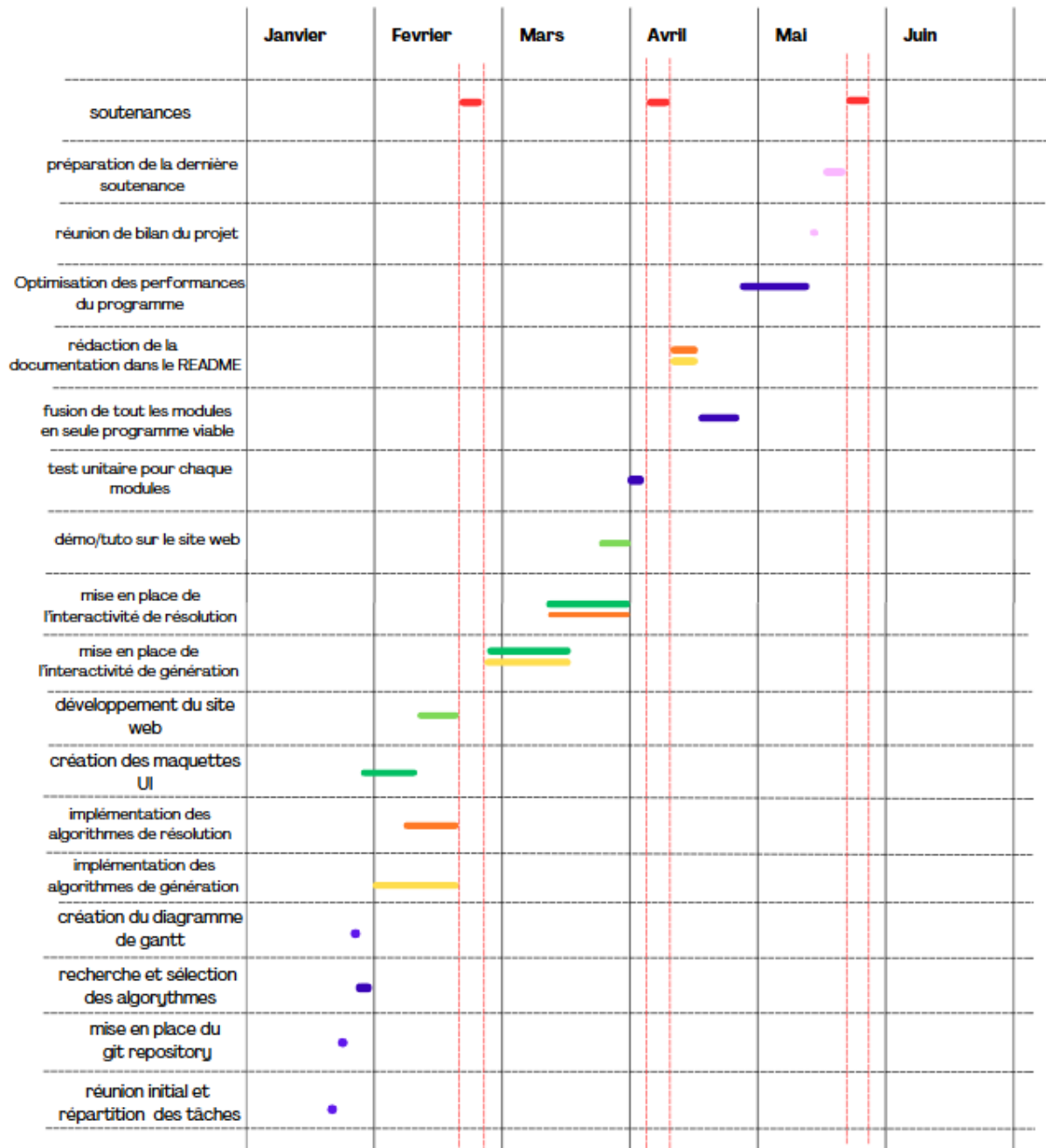


Figure 1: diagramme de Gantt

---

Légende:








	travaux nécessitant toute l'équipe
	réunion
	algorithme de résolution
	algorithme de génération
	soutenance
	Interface utilisateur de l'application
	développement du site web

Figure 2: légende

## 3 Objet de l'étude

### 3.1 Qu'est-ce que le projet peut nous apporter ?

Au-delà de véritables compétences en Rust, qui nous seront extrêmement utiles à l'avenir, que ce soit dans le cadre de nos études ou dans notre future vie professionnelle, ce projet nous offre l'opportunité de travailler en équipe. Cette collaboration nous permettra de développer un sens plus affûté de l'esprit d'équipe, tout en améliorant nos capacités de communication et notre organisation collective.

De plus, le fait de concevoir une application complète, depuis la phase de réflexion jusqu'à la mise en œuvre, nous donnera une expérience concrète en gestion de projet. Nous aurons l'occasion d'apprendre à surmonter les défis techniques, à partager nos connaissances et à prendre des décisions collectives pour atteindre un objectif commun. Ces compétences, à la fois techniques et humaines, seront précieuses pour nos futurs projets académiques ou professionnels.

## 4 Détails du projet

### 4.1 Librairie dédiée

#### 4.1.1 Importance d'une librairie

La création d'une librairie dédiée dans le projet de labyrinthe est cruciale pour centraliser les fonctionnalités essentielles, comme la génération et la résolution de labyrinthes, tout en assurant une architecture modulaire et facile à maintenir. En structurant chaque fonctionnalité dans des modules spécifiques, il devient possible de modifier ou d'étendre les composants sans impacter l'ensemble de l'application. De plus, le langage Rust, avec sa gestion fine de la mémoire et ses garanties de sécurité, assure des performances élevées et une robustesse accrue, ce qui est particulièrement pertinent pour des applications complexes. Enfin, une organisation claire et modulaire simplifie la collaboration, permettant à plusieurs développeurs de travailler efficacement sur différentes parties du projet.

#### 4.1.2 Structure de la librairie

La librairie est organisée en plusieurs modules distincts, chacun étant responsable d'un aspect spécifique du projet. Cette séparation permet une gestion claire et un développement extensible. Voici la structure du projet :

**Organisation des dossiers :**

```
src/  
lib.rs          # Point d'entrée de la librairie  
generator/  
  mod.rs        # Définitions principales  
  algorithms/   # Algorithmes (DFS, Prim, etc.)  
solver/  
  mod.rs        # Résolution des labyrinthes  
  strategies/   # Algorithmes de résolution (A*, BFS, etc.)  
utils/  
  mod.rs        # Fonctions utilitaires
```

Chaque module joue un rôle précis : le module *generator* est dédié à la création de labyrinthes en utilisant différents algorithmes comme DFS ou Prim, tandis que *solver* implémente des stratégies de résolution comme A\* ou BFS et *utils* regroupe des fonctions utilitaires pour des tâches transversales.

#### 4.1.3 Implémentation Pratique

Pour garantir la fiabilité et la maintenabilité de la librairie, plusieurs étapes sont suivies durant son implémentation. La première consiste à définir la structure de base du projet en utilisant des outils comme Cargo, ce qui permet d'organiser le code dès le départ. Une fois cette structure mise en place, les dépendances nécessaires, telles que des bibliothèques pour la génération aléatoire, sont ajoutées. Ensuite, les fonctionnalités principales sont développées au sein des modules dédiés, avec une attention particulière à la clarté du code et à sa modularité.

L'ajout de tests unitaires est une étape clé pour valider chaque module et garantir le bon fonctionnement des algorithmes. Cela inclut des tests pour vérifier la génération correcte des labyrinthes, la précision des algorithmes de résolution, et la stabilité de l'ensemble du projet. Enfin, une documentation détaillée est générée pour faciliter l'utilisation de la librairie par d'autres développeurs et garantir une compréhension claire de son fonctionnement. Cette documentation est essentielle pour assurer la pérennité du projet et encourager son adoption par une communauté plus large.

Avec cette approche, la librairie constitue une base robuste pour l'application, tout en offrant la flexibilité nécessaire pour évoluer et s'adapter à de nouveaux besoins.

La librairie sera ensuite utilisée pour être implémenté dans le projet relatif à l'exécutable binaire facilitant ainsi la structure du projet.

## 4.2 Interface utilisateur et Site Web

Un bon produit n'est rien sans un bon packaging ! Comme mentionné dans le cahier des charges, l'objectif principal est d'attirer l'attention de l'utilisateur sur des fonctionnalités souvent perçues comme ennuyeuses ou peu attrayantes. Une grande partie de notre travail consistera donc à réussir à capter et maintenir l'intérêt de l'utilisateur tout au long de son expérience avec l'application.

Interface utilisateur globale L'interface utilisateur sera intuitive et bien structurée. Un menu principal proposera deux options principales : - **Générer un labyrinthe** : Permet de configurer et visualiser la génération procédurale d'un labyrinthe. - **Résoudre un labyrinthe** : Montre en temps réel la résolution d'un labyrinthe déjà généré ou importé.

Une troisième option, plus discrète, présentera une courte description de l'équipe dans une section "Qui sommes-nous ?" ou "Crédits". Cette section mettra en avant les membres du projet, leurs rôles et leur contribution, tout en créant un lien humain avec l'utilisateur.

Pour renforcer l'aspect visuel et immersif, chaque bouton du menu principal sera animé. Par exemple, en survolant un bouton, une petite animation dynamique pourrait se déclencher pour attirer l'attention de l'utilisateur. Lorsque le labyrinthe est généré, il se dessinera progressivement sous les yeux de l'utilisateur, avec une animation fluide et captivante. De la même manière, la résolution sera visualisée étape par étape, permettant à l'utilisateur de suivre en direct les mécanismes des algorithmes employés.

Site web accompagnateur En complément de l'application, un site web accompagnera le projet. Ce site servira de vitrine et de support pour promouvoir et documenter notre travail.

Nous avons prévu de nous concentrer sur le **front-end**, qui constituera un **Minimum Viable Product (MVP)**. Le site sera développé en utilisant des technologies modernes et accessibles telles que : - **HTML** pour la structure du contenu, - **CSS** pour le style et les animations, - **JavaScript** pour les interactions dynamiques, comme des démonstrations visuelles de la génération et de la résolution de labyrinthes.

Objectif global L'ensemble de ces efforts vise à rendre le projet aussi attrayant que fonctionnel. En combinant un design soigné, des animations engageantes et une présentation claire, nous voulons offrir une expérience utilisateur mémorable, même pour un sujet souvent considéré comme technique ou abstrait. L'idée est non seulement d'expliquer les concepts derrière la génération procédurale, mais aussi de les rendre visuellement et émotionnellement captivants.

## 4.3 Génération du niveau de manière procédurale

La génération de labyrinthes de manière procédurale repose sur des algorithmes capables de créer des structures uniques et jouables, garantissant que chaque labyrinthe est navigable et offre un défi adapté. Plusieurs étapes sont nécessaires pour concevoir un générateur efficace.

### 4.3.1 Principe de la génération procédurale

La génération procédurale consiste à produire un labyrinthe en suivant un ensemble de règles et d'algorithmes déterministes ou pseudo-aléatoires. L'objectif est de garantir :

- **Unicité** : Chaque labyrinthe doit être unique à chaque exécution.
- **Navigabilité** : Le labyrinthe doit être solvable avec un chemin unique ou plusieurs solutions selon les besoins.
- **Personnalisation** : Les utilisateurs doivent pouvoir définir certains paramètres, comme la taille ou le type de labyrinthe.

### 4.3.2 Structure de données pour le labyrinthe

La représentation interne du labyrinthe est essentielle pour sa génération. Plusieurs choix de structures sont possibles :

- **Grille 2D** : Une matrice où chaque cellule représente une unité du labyrinthe (mur, chemin, départ, arrivée). Exemple en Rust :

```
let mut labyrinth: Vec<Vec<char>> = vec![vec!['#'; width]; height];
```

- **Graphe** : Chaque cellule est un nœud, et les connexions entre cellules sont des arêtes. Cela permet une plus grande flexibilité pour la génération de labyrinthes non réguliers. Une bibliothèque comme `petgraph` peut être utilisée pour manipuler ces graphes.

### 4.3.3 Algorithmes de génération

**1. Depth-First Search (DFS)** Le DFS est l'un des algorithmes les plus utilisés pour la génération de labyrinthes. Il suit un chemin en profondeur, puis revient en arrière pour explorer les alternatives.

**Étapes :**

1. Partir d'une cellule initiale.
2. Marquer la cellule comme visitée et ajouter ses voisins non visités à une pile.
3. Sélectionner aléatoirement un voisin non visité, briser le mur entre les deux, et répéter le processus.
4. Revenir en arrière lorsque tous les voisins sont visités.

**Avantages :** Produit des labyrinthes avec un chemin unique entre deux points.

**2. Algorithme de Prim** L'algorithme de Prim est basé sur l'ajout progressif de murs au labyrinthe en maintenant une frontière active.

**Étapes :**

1. Choisir une cellule initiale et l'ajouter au labyrinthe.
2. Ajouter ses voisins à une liste de murs actifs.
3. Choisir un mur aléatoire dans cette liste, le retirer, et l'utiliser pour connecter une cellule non visitée au labyrinthe.
4. Répéter jusqu'à ce que toutes les cellules soient connectées.

**Avantages :** Produit des labyrinthes plus complexes avec des embranchements multiples. **Inconvénients :** Plus gourmand en mémoire à cause de la gestion des murs actifs.

**3. Binary Tree Maze** Cette méthode génère des labyrinthes en parcourant une grille et en brisant les murs selon une règle spécifique (par exemple, choisir aléatoirement entre le mur supérieur ou le mur gauche).

**Étapes :**

1. Parcourir chaque cellule de la grille.
2. Briser un mur selon une règle définie (par exemple, toujours vers le haut ou vers la gauche).

**Avantages :** Simple à implémenter et rapide. **Inconvénients :** Produit des labyrinthes peu variés.

### 4.3.4 Personnalisation des labyrinthes

Le générateur doit permettre la personnalisation pour s'adapter aux besoins de l'utilisateur :

- **Taille du labyrinthe** : Largeur et hauteur définies par l'utilisateur.
- **Type d'algorithme** : Choisir entre DFS, Prim ou un autre algorithme.
- **Complexité du labyrinthe** : Ajuster la densité des murs ou le nombre de chemins alternatifs.



#### 4.3.5 Validation du labyrinthe

Pour garantir la navigabilité du labyrinthe :

- Vérifier qu'il existe un chemin entre le point de départ et d'arrivée.
- Tester le labyrinthe avec des algorithmes de résolution pour détecter les erreurs potentielles.
- Implémenter des tests unitaires pour valider les résultats du générateur.

#### 4.3.6 Optimisation et performance

- **Gestion efficace de la mémoire** : Utiliser des structures adaptées (comme des matrices ou des slices) pour éviter une consommation excessive de mémoire.
- **Parallélisme** : Exploiter les capacités de Rust pour paralléliser certaines parties du générateur, comme le traitement des cellules dans les labyrinthes très grands.

### 4.4 Résolution du niveau

Le domaine de l'informatique est vaste et en constante évolution, englobant une multitude de sous-disciplines et d'applications. Plusieurs notions principales contribuent au développement de ce domaine. Cela inclut la conception, le développement, le test et la maintenance de logiciels. Les développeurs créent des applications pour ordinateurs, smartphones et autres dispositifs. L'IA se concentre sur la création de systèmes capables d'effectuer des tâches nécessitant normalement l'intelligence humaine, comme la reconnaissance vocale, la prise de décision et la traduction.

Dans le cadre de ce projet, mon rôle sera d'analyser et de réaliser les algorithmes nécessaires afin de permettre la résolution des labyrinthes dans n'importe quelle circonstance. Plusieurs options de résolution possibles peuvent être utilisées dans le cas de figure présent, comme par exemple :

- **L'algorithme DFS (Depth-First Search)** : il s'agit ici d'un classique pour résoudre les labyrinthes. On démarre à une position initiale et on explore autant que possible en profondeur avant de revenir en arrière lorsqu'on rencontre un mur.
- **L'algorithme BFS (Breadth-First Search)** : il est également envisageable, et plus efficace si l'on cherche le plus court chemin, car BFS explore niveau par niveau. Cela peut être utile si l'on souhaite également trouver un chemin optimal.

## 5 Les limites techniques du projet

### 5.1 Complexité des algorithmes

Un des défis majeurs du projet réside dans la complexité des algorithmes utilisés, aussi bien pour la génération que pour la résolution des labyrinthes. Ces deux aspects impliquent des choix techniques cruciaux, car ils influencent directement les performances, l'efficacité et l'optimisation des ressources.

#### 5.1.1 Génération de labyrinthes

Les algorithmes de génération, tels que DFS (Depth-First Search) et Prim, présentent des défis spécifiques :

**DFS (Depth-First Search)** Cet algorithme génère un labyrinthe en explorant un chemin en profondeur jusqu'à ce qu'il ne puisse plus avancer, puis revient sur ses pas pour explorer de nouvelles branches. Complexité temporelle :  $O(V + E)$ , où  $V$  est le nombre de cellules et  $E$  les connexions possibles. Complexité spatiale : L'utilisation d'une pile pour stocker les chemins explorés peut poser des problèmes pour de très grands labyrinthes, notamment en raison de la limite de la pile de la mémoire. Optimisation nécessaire : Implémenter une pile optimisée ou utiliser des techniques comme le *tail-recursion* si applicable en Rust.

**Algorithme de Prim** Cet algorithme est basé sur la sélection aléatoire des murs adjacents d'une cellule visitée et leur ouverture pour créer un chemin. Complexité temporelle :  $O(E \cdot \log(V))$  en moyenne, en utilisant une file de priorité pour sélectionner les murs. Complexité spatiale : La gestion de la file de priorité et des structures de données associées peut nécessiter une allocation dynamique importante en

mémoire. Défis spécifiques : L'implémentation de structures comme une heap en Rust, qui demande une gestion stricte de la mémoire, peut s'avérer complexe.

### 5.1.2 Résolution de labyrinthes

La résolution d'un labyrinthe implique l'utilisation d'algorithmes comme A\* ou BFS, chacun ayant ses avantages et ses inconvénients selon les cas d'utilisation.

BFS (Breadth-First Search) BFS explore les cellules du labyrinthe niveau par niveau, garantissant de trouver le chemin le plus court. Complexité temporelle :  $O(V + E)$ , ce qui est efficace pour les labyrinthes de taille modérée. Complexité spatiale : La file utilisée pour stocker les cellules à visiter peut devenir très grande, surtout pour des labyrinthes denses. Défis spécifiques : En Rust, la gestion dynamique de cette file nécessite une compréhension approfondie des types comme VecDeque pour éviter des erreurs de gestion de mémoire.

A (A-Star)\* Cet algorithme améliore BFS en utilisant une heuristique pour guider l'exploration des cellules, ce qui peut réduire le temps de calcul. Complexité temporelle :  $O(E)$  dans le meilleur cas avec une bonne heuristique, mais peut être aussi coûteux qu'un DFS si l'heuristique est mal choisie. Complexité spatiale : L'utilisation d'une file de priorité (heap) pour maintenir les cellules à explorer peut demander une gestion rigoureuse des structures de données. Défis spécifiques : Implémenter une heuristique efficace (par exemple, la distance de Manhattan) et gérer correctement les structures complexes en Rust.

### 5.1.3 Limitations supplémentaires

Problèmes de scalabilité Les labyrinthes de très grande taille peuvent entraîner une explosion de la complexité temporelle et spatiale. Par exemple, un labyrinthe de  $1000 \times 1000$  cellules implique la gestion d'un million de cellules, ce qui peut saturer la mémoire pour certains algorithmes.

Gestion des contraintes de Rust La stricte gestion des emprunts et de la propriété des données en Rust (via le système de Borrow Checker) peut compliquer l'implémentation d'algorithmes manipulant des structures dynamiques, comme les graphes ou les piles.

Adaptation aux besoins du projet Il peut être nécessaire d'adapter ou de simplifier certains algorithmes pour garantir une performance acceptable tout en respectant les contraintes de temps de développement.

## 5.2 Performance et gestion de la mémoire

Rust, bien qu'efficace en termes de gestion de la mémoire, impose des contraintes strictes sur la manière dont les ressources sont allouées et libérées. Ces contraintes peuvent poser problème dans :

La gestion des labyrinthes de grande taille, où les allocations dynamiques peuvent être coûteuses. La manipulation de structures complexes (par exemple, des graphes ou des matrices) dans un contexte de calcul intensif.

## 5.3 Modularité et intégration de la librairie

L'utilisation d'une librairie dédiée constitue un choix stratégique dans le projet pour structurer et centraliser les fonctionnalités essentielles, telles que la génération et la résolution des labyrinthes. Cependant, ce choix s'accompagne de plusieurs défis techniques et organisationnels qu'il est important de considérer.

### 5.3.1 Modularité des composants

Pour garantir une architecture claire et maintenable, chaque aspect du projet est divisé en modules spécifiques :

Génération des labyrinthes : Ce module se concentre sur la création des labyrinthes à l'aide d'algorithmes comme DFS ou Prim. Résolution des labyrinthes : Ce module implémente les algorithmes de résolution (A\*, BFS, etc.), tout en permettant une personnalisation pour différents scénarios. Utils : Ce module contient des fonctions auxiliaires (comme la gestion des fichiers ou la génération aléatoire), nécessaires au bon fonctionnement des autres parties.

Défis liés à la modularité :

Indépendance fonctionnelle Chaque module doit être capable de fonctionner indépendamment pour assurer une séparation claire des responsabilités. Par exemple, le module de génération ne doit pas

dépendre directement du module de résolution. Cela nécessite une interface bien définie entre les modules, avec des fonctions exposées clairement (via `pub fn` en Rust), permettant une intégration facile sans créer de dépendances circulaires.

**Tests unitaires modulaires** Chaque module doit être accompagné de tests unitaires spécifiques pour vérifier son bon fonctionnement de manière isolée. Par exemple, les tests pour la génération doivent valider que les labyrinthes générés respectent les contraintes spécifiées (connectivité, unicité du chemin, etc.). L'ajout de tests intégrés entre les modules est également crucial pour garantir leur interaction harmonieuse.

### 5.3.2 Intégration dans le projet global

Une fois les modules développés, leur intégration dans une librairie commune doit être réalisée avec soin. Cela implique plusieurs étapes :

**Définir un point d'entrée unique** Un fichier `lib.rs` centralise l'accès aux différentes fonctionnalités des modules. Par exemple :

```
pub mod generator; pub mod solver; pub mod utils;
```

Cela garantit que les utilisateurs de la librairie n'ont pas à interagir directement avec les implémentations internes, mais seulement avec les fonctions exposées.

**Interaction entre modules** Les modules doivent interagir uniquement via des API bien définies. Par exemple, le module de résolution ne devrait pas manipuler directement les structures internes du module de génération, mais plutôt recevoir des données formatées via une interface commune.

**Gestion des dépendances internes** Les dépendances entre modules doivent être réduites au minimum pour éviter les conflits ou les bugs. Cela implique une structuration claire des données partagées, comme une représentation commune du labyrinthe (par exemple, une matrice ou un graphe).

### 5.3.3 Gestion des dépendances externes

Certains aspects du projet, comme la génération aléatoire ou l'optimisation des performances, nécessitent l'utilisation de bibliothèques externes.

**Défis associés :**

**Compatibilité avec le projet** Toutes les dépendances doivent être compatibles avec la version de Rust utilisée. Par exemple, des bibliothèques comme `rand` pour la génération aléatoire ou `petgraph` pour la manipulation de graphes doivent être bien configurées dans le fichier `Cargo.toml`. Il est important de limiter les dépendances inutiles pour réduire la taille de la librairie et éviter les conflits potentiels.

**Mises à jour et maintenance** Les dépendances externes doivent être surveillées pour garantir leur compatibilité avec les futures versions de Rust. Cela peut nécessiter des mises à jour régulières du projet pour éviter les obsolescences.

**Documentation et apprentissage** L'intégration de certaines dépendances peut nécessiter un temps d'apprentissage pour comprendre leur fonctionnement, notamment si elles utilisent des paradigmes avancés de Rust, comme les traits ou les lifetimes complexes.

### 5.3.4 Avantages et limitations de la modularité

Bien que la modularité offre des avantages significatifs, comme une meilleure organisation et une facilité de maintenance, elle s'accompagne de certaines limitations :

**Surcharge initiale :** La création de plusieurs modules et leur intégration dans une librairie peut prendre plus de temps que de coder tout dans un seul fichier. **Gestion de la complexité :** Le besoin de définir des interfaces claires et de maintenir la cohérence entre les modules peut augmenter la complexité globale du projet.

## 5.4 Interface utilisateur (UI)

La création d'une interface utilisateur (UI) fonctionnelle et intuitive est un aspect essentiel du projet, permettant aux utilisateurs d'interagir facilement avec les fonctionnalités de génération et de résolution de labyrinthes. Cependant, elle comporte plusieurs défis techniques et conceptuels, que ce soit en termes d'intégration avec la logique métier ou de choix des outils et bibliothèques disponibles en Rust.

### 5.4.1 Intégration de la logique métier dans l'interface

Un des principaux défis consiste à intégrer de manière fluide la logique métier (génération et résolution) dans l'interface utilisateur.

**Couplage entre UI et backend** L'interface doit pouvoir communiquer efficacement avec les modules de génération et de résolution pour récupérer les données nécessaires, comme le labyrinthe généré ou le chemin résolu. Cela nécessite la mise en place d'un système de gestion d'événements ou d'appels de fonctions asynchrones pour coordonner les interactions entre l'UI et la logique métier.

**Représentation visuelle des labyrinthes** Les labyrinthes doivent être affichés sous une forme compréhensible, que ce soit sous forme de grille (2D) ou d'un graphe visuel. Cela demande une conversion des structures internes (comme des matrices ou des graphes) en représentations graphiques adaptées. Il faut également gérer les mises à jour dynamiques, par exemple lorsqu'un utilisateur génère un nouveau labyrinthe ou visualise sa résolution.

**Gestion des entrées utilisateur** L'interface doit permettre aux utilisateurs de configurer certains paramètres (taille du labyrinthe, algorithme de résolution, etc.) de manière intuitive. Cela peut impliquer des contrôles interactifs, comme des curseurs, des menus déroulants ou des boutons, qui nécessitent une gestion précise des événements dans le code.

### 5.4.2 Limitations des bibliothèques disponibles en Rust pour les interfaces graphiques

Bien que Rust soit performant pour le développement d'applications systèmes et backend, il est encore jeune dans le domaine des interfaces graphiques, ce qui limite les options disponibles.

**Bibliothèques graphiques disponibles** **egui** : Une bibliothèque moderne et légère pour créer des interfaces graphiques en Rust. Elle est facile à utiliser pour des interfaces simples mais peut être limitée pour des visuels complexes comme l'affichage d'un labyrinthe interactif. **GTK-rs** : Une liaison Rust pour GTK, une bibliothèque mature et puissante utilisée dans de nombreuses applications. Bien qu'elle offre des fonctionnalités avancées, son apprentissage et son intégration peuvent être complexes, en particulier pour les débutants en Rust. **Druid** : Une autre option intéressante pour le développement d'UI en Rust, qui met l'accent sur la simplicité et la réactivité. Cependant, elle est encore en cours de développement actif et peut manquer de certaines fonctionnalités nécessaires.

**Défis liés aux bibliothèques** **Manque de maturité** : Comparé à des frameworks UI bien établis dans d'autres langages (comme Qt ou Tkinter), les bibliothèques Rust manquent parfois de documentation complète ou de fonctionnalités avancées. **Performance** : Certaines bibliothèques, bien qu'efficaces pour des tâches simples, peuvent présenter des problèmes de performance pour des interfaces complexes ou interactives. **Courbe d'apprentissage** : L'utilisation de ces bibliothèques nécessite souvent une compréhension approfondie des concepts spécifiques à Rust, tels que les lifetimes, la gestion de la mémoire, ou les systèmes de threads.

### 5.4.3 Représentation et interactivité de l'interface

La représentation visuelle des labyrinthes et leur interactivité ajoutent des défis supplémentaires :

**Affichage dynamique des labyrinthes** Le labyrinthe doit pouvoir être affiché sous forme de grille ou de graphe, avec des éléments visuels distincts pour les murs, les chemins et les points de départ et d'arrivée. Une mise à jour dynamique doit être possible pour permettre à l'utilisateur de visualiser en temps réel la progression de l'algorithme de résolution.

**Interactivité** L'utilisateur doit pouvoir interagir avec le labyrinthe, par exemple en choisissant une cellule de départ ou d'arrivée, ou en modifiant manuellement certaines parties du labyrinthe. Cela nécessite la gestion des clics de souris, des mouvements ou d'autres événements utilisateur, qui doivent être traduits en actions dans la logique métier.

### 5.4.4 Solutions potentielles aux limitations

Pour surmonter ces défis, plusieurs approches peuvent être envisagées :

**Choix d'une bibliothèque adaptée** Pour une UI simple et rapide à développer, **egui** peut être un bon choix grâce à sa simplicité. Pour une interface plus sophistiquée, **GTK-rs** ou **Druid** offrent des options plus robustes mais nécessitent plus d'efforts en termes d'apprentissage.

**Découplage UI/Backend** Utiliser une architecture découplée, où la logique métier est totalement séparée de l'interface utilisateur. Cela permet de développer et de tester les deux parties indépendamment. Un système d'événements ou de messages (comme le modèle "Observer" ou un bus d'événements) peut faciliter cette communication.

Optimisation des performances Réduire les calculs graphiques inutiles en mettant à jour uniquement les parties de l'interface qui changent (par exemple, lorsque l'algorithme résout une cellule). Exploiter les capacités de parallélisme de Rust pour effectuer des tâches complexes, comme le rendu des labyrinthes, en parallèle avec la logique métier.

## 5.5 Tests et validation

Le projet implique des algorithmes non triviaux qui nécessitent une validation rigoureuse :

Tester de manière exhaustive tous les cas possibles pour garantir la génération correcte des labyrinthes et la précision des algorithmes de résolution. Mettre en place des tests unitaires en Rust, qui peuvent être fastidieux, surtout pour les scénarios impliquant des entrées et des sorties complexes.

## 5.6 Visualisation des résultats

Représenter visuellement le labyrinthe et ses solutions peut poser plusieurs problèmes techniques :

La création d'un affichage graphique ou textuel clair et adapté, tout en évitant des performances médiocres pour des labyrinthes de grande taille. La gestion de l'interaction utilisateur dans l'interface, par exemple, pour choisir la taille du labyrinthe ou le type d'algorithme à utiliser.

## 6 Conclusion

Ce projet représente une opportunité unique de combiner des compétences en programmation, en gestion de projet et en travail d'équipe pour réaliser une application fonctionnelle et innovante. En créant une interface capable de générer des labyrinthes de manière procédurale et de les résoudre grâce à des algorithmes avancés, nous avons pour objectif de repousser nos limites techniques tout en mettant en pratique les concepts étudiés au cours de notre formation.

Les défis identifiés, tels que la complexité des algorithmes, l'intégration modulaire, ou encore les contraintes liées à la création d'une interface utilisateur intuitive, offrent autant de possibilités d'apprentissage et d'amélioration. La modularité de la librairie, la gestion efficace des ressources et l'attention portée à la validation et aux tests garantissent que le produit final sera robuste, évolutif et performant.

Au-delà de l'aspect technique, ce projet favorise le développement de compétences humaines essentielles comme l'esprit d'équipe, la communication et la gestion des responsabilités. Ces acquis seront précieux pour nos futures carrières, que ce soit dans le domaine de la programmation ou dans d'autres secteurs connexes.

En conclusion, ce cahier des charges établit une vision claire des objectifs du projet, des étapes nécessaires à sa réalisation et des défis à surmonter. En suivant ce cadre, nous sommes confiants de pouvoir livrer une application de qualité répondant aux exigences fixées, tout en acquérant des compétences précieuses pour notre développement académique et professionnel.