Rapport
Mini-Projet Labyrinthe

Le 15 Novembre 2025
version 1.0

Boubacar Sadio DIALLO

Fonction : Apprenti ingénieur en Informatique
boubacar-sadio.diallo@ensicaen.fr

Professeur chargé de cours :

Julien RABIN

Table des matières

Table des figures

Liste des tableaux

1 Introduction

Ce document présente le rapport final du mini-projet de développement d'un jeu de labyrinthe en console.

L'objectif principal de ce projet est de réaliser une application complète en langage C, capable de **générer**, **sauvegarder**, **charger** et permettre de **jouer** à des labyrinthes en deux dimensions.

Conformément au cahier des charges, le jeu implémente plusieurs fonctionnalités clés :

— **La génération de labyrinthes parfaits** (connectés et sans boucle) de taille dynamique, basée sur l'algorithme de Kruskal.

— Un **système d'objets** incluant une clé (`k`) obligatoire pour déverrouiller la sortie (`_`), des trésors (`b`) et des pièges (`m`).

— Une **logique de jeu** où le joueur (`P`) doit trouver la clé puis la sortie, tout en gérant un score.

— La **persistance des données**, incluant la sauvegarde des labyrinthes (.cfg) et la gestion d'un tableau des 10 meilleurs scores pour chaque labyrinthe (.score).

Pour répondre à ces exigences, le projet a été développé en **Langage C** (norme C99) et s'appuie sur plusieurs outils standards pour garantir sa robustesse et sa maintenabilité :

— Un **Makefile** pour automatiser la compilation du jeu et des tests.

— **Doxygen** pour la génération de la documentation technique du code.

— **MinUnit** pour la mise en place de tests unitaires validant la logique de génération.

Ce rapport détaillera les choix d'architecture, l'implémentation des algorithmes clés, le manuel utilisateur, et les résultats des tests effectués.

1.1 Structure du Projet

Pour garantir la modularité et la maintenabilité du code, le projet a été organisé en suivant une arborescence claire. Le principe de base est la séparation entre les déclarations (`.h`) et les définitions (`.c`), répartis dans les dossiers `include` et `src`.

— `include/` : Ce répertoire contient tous les fichiers d'en-tête (`.h`). Un fichier d'en-tête agit comme le "mode d'emploi" d'un module : il présente publiquement les **prototypes des fonctions** (ce qu'elles font) et les **définitions des structures** (`Labyrinthe`, `Player`) que les autres parties du programme ont le droit d'utiliser.

— `src/` : Contient tous les fichiers source (`.c`). C'est ici que se trouve le **code réel** (le "corps") des fonctions. Chaque fichier `.c` (comme `jeu.c`) contient la logique et le fonctionnement interne des fonctions qui ont été déclarées dans son `.h` correspondant (comme `jeu.h`).

— `tests/` : Ce dossier regroupe les fichiers de tests unitaires (utilisant MinUnit). Ces tests sont compilés en un exécutable séparé pour valider la logique du cœur du programme (génération) sans impacter l'application principale.

— `data/` : Répertoire utilisé pour la persistance des données. Il contient les labyrinthes sauvegardés (fichiers `.cfg`) et les tables des meilleurs scores (fichiers `.score`).

— `lib/` et `bin/` : Dossiers (créés par le Makefile) contenant respectivement les fichiers objets (`.o`) compilés et les exécutables finaux (`exec` et `execTest`).

— `Makefile` : Fichier central pour l'automatisation de la compilation. Il gère la création du jeu principal, la compilation et l'exécution des tests, ainsi que le nettoyage du projet et la generation de la documentation.

2 Architecture et Conception par Étapes

L'architecture du projet a été construite de manière incrémentale, en suivant les étapes définies par le cahier des charges. Chaque étape a introduit de nouveaux modules et de nouvelles structures de données.

2.1 Étape 1 : Génération d'un Labyrinthe Parfait

Le premier objectif était de générer et d'afficher un labyrinthe parfait de taille fixe 11*25.

2.1.1 Structures de Données

Le choix initial s'est porté sur une structure `Labyrinthe` contenant une grille `int ** array`. L'utilisation d'entiers (plutôt que de caractères) a permis de stocker des états multiples :

— `-1` pour un mur (`#`).
— `-2` pour l'entrée (`o`).
— `-3` pour la sortie (`_`).
— `0` pour un chemin vide.
— `>0` pour les identifiants d'ensemble durant la génération.

Une structure `Cellule` a aussi été créée pour représenter les murs potentiels lors de la génération (coordonnées du mur et des deux cases qu'il sépare).

2.2 Algorithme de Génération

L'algorithme implémenté (dans `generation.c`) utilise la propriété des labyrinthes parfaits telle que 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, suivant une approche ascendante (bottom-up).

Le processus suit les étapes décrites dans le cahier des charges :

1. **Initialisation :** L'algorithme associe une valeur unique à chaque cellule "chemin" (aux indices impairs) via la fonction `associe_valeur_cellule`. Il part d'un labyrinthe où tous les murs sont fermés (tel que créé par `creer_labyrinthe`). Une structure de données Union-Find (`int *sets`) est initialisée où chaque cellule est son propre ensemble.
2. **Aléatoire :** Une liste de tous les murs potentiels est créée (allouée dynamiquement). À chaque itération, on choisit un mur à ouvrir de manière aléatoire, ce qui est implémenté en mélangeant la liste complète au début avec `melanger_murs`.
3. **Vérification et Fusion :** L'algorithme parcourt la liste des murs mélangés.
   — À chaque fois que l'on tente d'ouvrir un mur, on vérifie que les deux cellules adjacentes ont des identifiants d'ensemble différents. C'est le rôle de l'appel : `find(sets, id1) != find(sets, id2)`.
      — Si les identifiants sont identiques, c'est que les deux cellules sont déjà reliées (elles appartiennent au même chemin). On ne peut donc pas ouvrir le mur, ce qui garantit l'absence de boucle.
      — Si les identifiants sont différents, le mur est "abattu" (`lab->array[m.x][m.y] = 0`). Les deux ensembles sont fusionnés (via `unite(sets, id1, id2)`) et l'identifiant de la première cellule est affecté à toutes les cellules du second chemin dans la grille (via `fusionner_identifiants_labyrinthe(lab, id2, id1)`).

2.2.1 Affichage

Une simple fonction `afficher_labyrinthe` (dans `affichage.c`) parcourt la grille `int **array` et affiche le caractère correspondant à chaque code entier.

2.3 Étape 2 : Menu, Sauvegarde et Jeu

Cette étape a introduit la persistance des données et l'interactivité.

2.3.1 Gestion des Menus

La fonction `menu()` (dans `menu.c`) gère la boucle principale du programme, permettant à l'utilisateur de choisir une action via une structure `switch`.

2.3.2 Sauvegarde et Chargement (Persistance)

Le module `sauvegarde.c` a été créé.
- `sauvegarder_labyrinthe()` écrit la grille (en convertissant les codes entiers en caractères) dans un fichier `./data/nom.cfg`.
- `charger_labyrinthe()` lit un `.cfg` et ré-alloue la structure `Labyrinthe` en mémoire.
- Un fichier `data/index.txt` est maintenu pour lister les noms des labyrinthes disponibles.

2.3.3 Logique de Jeu Initiale

Le module `jeu.c` a été introduit.
- La structure `Player` (ou `Joueur` pour le franglish :)) a été créée pour encapsuler l'état de la partie. Initialement (à l'Étape 2), elle ne stocke que la position du joueur (`pos`). Cette structure sera enrichie à l'étape 3 pour inclure le score et l'inventaire.
- La fonction `jouer()` implémente la boucle de jeu principale. Conformément au cahier des charges, elle gère les déplacements via les touches `z` (haut), `q` (gauche), `s` (bas) et `d` (droite).
- La lecture de la touche (suivie de "Entrée") est gérée par `scanf()`, ce qui bloque l'exécution en attendant la saisie de l'utilisateur.
- À chaque tour de boucle, la fonction `jouer()` exécute la séquence suivante :
  1. Efface l'écran de la console (via `system("clear")`) pour simuler une animation et un affichage "en place".
  2. Affiche l'état actuel du labyrinthe ainsi que la position du joueur (via `afficher_labyrinthe()`).
  3. Attend la saisie d'une touche de déplacement.
  4. Calcule la nouvelle position potentielle et vérifie sa validité (pas un mur) via `tenter_deplacement()`.
  5. Vérifie si le joueur a atteint la case de sortie (via `verifier_sortie()`).
- Lorsque le joueur arrive à la fin du labyrinthe (la condition de victoire est remplie), la boucle s'arrête, la fonction `jouer()` se termine, et le contrôle est rendu au `menu()` principal, comme demandé par le cahier des charges.

2.4 Étape 3 : Objets, Score et Classement

Cette étape a finalisé la logique de jeu en ajoutant un système de score.

2.4.1 Extension des Structures

- **Labyrinthe :** La grille `int **array` gère désormais de nouveaux codes pour les objets : `CLE (-4)`, `BONUS (-5)`, `MALUS (-6)`.
- **Player :** La structure a été étendue pour inclure `int score`, `int moves`, et `int has_key`.
- **ScoreEntry (Nouvelle) :** Une nouvelle structure a été créée pour la persistance des scores (`nom[50]`, `score_final`, `moves`).

2.4.2 Mise à jour de la Génération

La fonction `placer_objets()` a été ajoutée. Elle est appelée à la fin de `generer_labyrinthe` pour placer aléatoirement la clé et les objets sur des cases "chemin" (`0`). Le nombre d'objets est calculé dynamiquement en fonction de la taille du labyrinthe.

2.4.3 Système de Score (score.c)

Un nouveau module, `score.c`, gère le classement.
— `gerer_highscore()` est appelée à la fin d'une partie.
— Elle lit le fichier binaire `./data/nom.score` (avec `fread`) dans un tableau de `ScoreEntry`.
— Elle compare le nouveau score à ceux existants.
— Si c'est un record, elle demande le nom du joueur, ajoute le nouveau score, trie le tableau (avec `qsort`), et réécrit les 10 meilleurs scores (avec `fwrite`).
— Une nouvelle option de menu appelle `afficher_scores()`.

2.5 Étape 4 : Niveaux de Difficulté (Non réalisée)

(Cahier des charges : 10 points)
Cette étape, qui prévoyait l'ajout de labyrinthes avec plusieurs chemins possibles et de monstres gérés par pointeurs de fonction, n'a pas été implémentée dans le cadre de ce projet. Elle est discutée dans la section `Conclusion` en tant qu'amélioration future possible.

3 Manuel Utilisateur

3.1 Introduction

Ce document explique comment compiler, lancer et jouer au jeu du Labyrinthe.
Ce programme est un jeu en console qui génère des labyrinthes parfaits. Le joueur doit naviguer dans le labyrinthe, trouver la **clé** (`k`) pour déverrouiller la **sortie** (`_`), tout en ramassant des **trésors** (`b`) et en évitant des **pièges** (`m`). Le jeu sauvegarde les 10 meilleurs scores pour chaque labyrinthe comme vu ci-dessus.

3.2 Compilation du Projet

Ce projet utilise `make` pour gérer la compilation, la generation et l'exécution.

3.2.1 Prérequis

Pour compiler et exécuter ce projet, vous aurez besoin de :
— Un système d'exploitation de type UNIX (Linux ou macOS).
— Le compilateur `gcc`.
— L'utilitaire `make`.

3.2.2 Commandes (via le Script d'Automatisation)

Pour simplifier la compilation, l'exécution et les tâches de maintenance, un script shell `script.sh` est fourni.

1. Placez-vous à la racine du projet (là où se trouve le `Makefile`) et le script.sh.
2. **Rendre le script exécutable** (cette commande n'est à exécuter qu'une seule fois) :

```
chmod +x script.sh
```

3. **Utiliser le script** pour toutes les actions principales. Le script appelle le `Makefile` pour vous :
   — Pour **compiler** le jeu principal et les tests :

```
./script.sh all
```

— Pour **lancer le jeu** (compile si nécessaire) :
`./script.sh run`

— Pour **lancer les tests unitaires** :
`./script.sh test`

— Pour **générer la documentation** (Doxygen doit être installé) :
`./script.sh doc`

— Pour **nettoyer** tous les fichiers compilés :
`./script.sh clean`

— Pour **archiver le projet** :
`./script.sh zip`

4. Les commandes de compilation créent deux exécutables dans le dossier `./bin/` :
   — `./bin/exec` (Le jeu principal, lancé par `./script.sh run`)
   — `./bin/execTest` (Les tests, lancés par `./script.sh test`)

3.3 Lancer le Jeu

Pour démarrer le jeu après la compilation, utilisez la commande suivante :

```
make run
```

Vous pouvez également lancer l'exécutable manuellement :

```
./bin/exec
```

3.4 Le Menu Principal

Au lancement, vous verrez le menu principal :

```
=== MENU PRINCIPAL ===
1. Créer un labyrinthe
2. Charger un labyrinthe
3. Jouer
4. Voir les meilleurs scores
5. Quitter
```

— **1. Créer :** Vous demande une hauteur, une largeur (impaires) et un nom pour générer un nouveau labyrinthe.
— **2. Charger :** Affiche un aperçu de tous les labyrinthes sauvegardés et vous demande d'en choisir un par son nom.
— **3. Jouer :** Lance la partie sur le labyrinthe actuellement chargé.
— **4. Voir les meilleurs scores :** Affiche le classement des 10 meilleurs scores pour le labyrinthe chargé.
— **5. Quitter :** Quitte le programme.

3.5 Comment Jouer ?

3.5.1 Symboles

— `P` : Votre joueur
— `#` : Un mur (infranchissable)
— `o` : L'entrée
— `_` : La sortie (verrouillée au début)
— `k` : La clé (nécessaire pour ouvrir la sortie)
— `b` : Un trésor (Bonus de +50 points)
— `m` : Un piège (Malus de -25 points)

3.5.2 Commandes

Utilisez les touches suivantes (suivies de "Entrée") pour vous déplacer :
— `z` : Se déplacer vers le **Haut**
— `s` : Se déplacer vers le **Bas**
— `q` : Se déplacer vers la **Gauche**
— `d` : Se déplacer vers la **Droite**
— `x` : Quitter la partie en cours et retourner au menu.

3.5.3 Objectif

1. Parcourez le labyrinthe pour trouver la clé (`k`).
2. Une fois la clé ramassée, dirigez-vous vers la sortie (`_`).
3. Tentez d'obtenir le meilleur score en ramassant des bonus (`b`) et en évitant les pièges (`m`). Votre score final est pénalisé par le nombre de déplacements.
4. Si votre score est dans le top 10, vous pourrez enregistrer votre nom.

4 Tests Unitaires

Pour garantir la robustesse et la fiabilité du code, une approche de tests unitaires a été mise en œuvre. Le framework minimaliste **MinUnit** a été choisi pour sa simplicité d'intégration dans un projet C.

La stratégie de test a consisté à créer des exécutables de test distincts du programme principal, compilés et exécutés via des cibles dédiées dans le `Makefile` (`make test`).

Cette approche nous a permis de valider des composants logiques critiques de manière isolée, en se concentrant sur un domaine principal : la génération du labyrinthe.

4.1 Tests de Génération

Pour valider la robustesse de l'algorithme de génération, trois tests unitaires critiques ont été mis en place :

4.1.1 Test de Parité des Dimensions

L'algorithme de génération repose sur des indices impairs pour définir les cellules "chemin". Ce test de robustesse vérifie que la fonction `creer_labyrinthe` refuse de créer un labyrinthe si la hauteur ou la largeur fournie est un nombre pair. Il s'assure qu'elle retourne bien `NULL` dans ce cas, prévenant ainsi une erreur d'exécution.

4.1.2 Test du Nombre de Murs

Ce test valide la logique de création de la liste des murs. Il calcule d'abord le nombre théorique de murs internes (horizontaux et verticaux) pour un labyrinthe de taille donnée. Ensuite, il simule la boucle de la fonction `generer_labyrinthe` pour compter le nombre de murs qu'elle génère. Le test réussit si le nombre calculé par la boucle correspond exactement au nombre théorique attendu.

5 Conclusion

Cette section fait le bilan du projet en répondant aux exigences du manuel développeur, en présentant ce qui a été réalisé, les limitations connues et les pistes d'améliorations futures.

5.1 Bilan (Ce qui a été réalisé)

L'ensemble des objectifs fixés par le cahier des charges a été atteint. Le programme est fonctionnel et stable. Les fonctionnalités suivantes sont implémentées et validées :

— Génération de labyrinthe parfait dynamique.
— Sauvegarde et chargement des labyrinthes (.cfg).
— Placement dynamique des objets (clé, bonus, malus).
— Logique de jeu complète (déplacement, ramassage).
— Système de 10 meilleurs scores par labyrinthe (.score).
— Documentation Doxygen complète.
— Tests unitaires (MinUnit) validant la génération.

5.2 Limitations et Bugs Connus

Malgré sa fonctionnalité, le projet actuel présente certaines limitations inhérentes à une application console :

— Le jeu fonctionne en console uniquement, avec une boucle de jeu bloquée par la fonction `scanf`.
— Le rafraîchissement de l'écran (via `system("clear")`) peut provoquer un scintillement sur certains terminaux.

5.3 Améliorations (Ce qui reste à faire)

Le projet dispose d'une base solide en C, permettant d'envisager plusieurs améliorations futures :

— L'amélioration principale serait le portage vers une **interface graphique 2D** (en utilisant une bibliothèque comme **SDL2** ou **Raylib**) pour une expérience de jeu en temps réel.
— Ajouter différents niveaux de difficulté, qui influenceraient la taille du labyrinthe et la densité des pièges.
— Permettre le déplacement à l'aide des touches fléchées (nécessiterait une bibliothèque comme 'ncurses' ou une GUI).
— L'implémentation d'algorithmes non triviaux, en particulier le parcours **DFS** (parcours en profondeur) nécessaire pour valider la connexité du labyrinthe dans les tests unitaires.

5.4 Bilan Personnel et Difficultés Rencontrées

Ce projet a été une excellente opportunité de mettre en pratique les aspects fondamentaux de la programmation en C. Les principales difficultés rencontrées ont été :

— La **gestion rigoureuse de la mémoire** : Assurer l'absence de fuites mémoire (‘malloc‘/‘free‘), notamment lors de la génération dynamique et de la libération des structures (labyrinthe, grille 2D, listes de murs).
— La configuration d'un **Makefile** robuste, capable de gérer des exécutables multiples (jeu principal et tests) tout en gérant correctement les dépendances.

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE