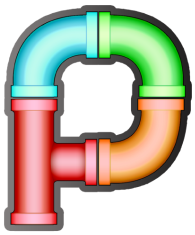
Projet de programmation
RAPPORT FINAL

Mai 2024

Membres du groupe

Khadir Abdessamed
Semani Issac
Tachou Andrei
Zhou Ludovic



Contents

1 Introduction

Ce document a pour objectif d'expliquer étape par étape comment notre groupe ED3 s'est chargé du développement du jeu CrazyPlumber, quelles ont été nos idées et nos solutions algorithmiques aux problèmes que nous avons rencontrés en décortiquant chaque idée importante. Les parties de codes seront accompagnés d'explications détaillées sur leur fonctionnement et leur viabilité par rapport à d'autres méthodes. Ce document ne se substitue pas aux réponses que nous vous fournirons lors de notre présentation mais nous espérons faire de notre mieux pour mettre en lumière notre travail du mieux possible.

2 Structure du code

Le projet est organisé en plusieurs répertoires, chacun contenant des fichiers spécifiques à une fonctionnalité ou un aspect du jeu. Cette organisation nous a permis de maintenir une structure de code claire et modulaire, facilitant ainsi la gestion et la maintenance du projet. Nous avons veillé à inclure des commentaires explicatifs dans le code pour améliorer sa lisibilité et aider les enseignants à comprendre la logique derrière chaque fonction.

2.1 Répertoires principaux

- **model** : Contient les classes principales du jeu, telles que `Map`, `Cell`, et `Play`, qui encapsulent la logique du jeu et les interactions entre les différents éléments.
- **view** : Regroupe les classes responsables de l'affichage graphique du jeu, comme `GamePanel` et les différentes `Overlay` pour gérer les interfaces utilisateur (menu, transition, etc.).
- **control** : Comprend les classes qui gèrent les interactions de l'utilisateur, comme `KeyHandler` et `MouseHandler`, permettant de capturer et de traiter les entrées clavier et souris.
- **exception** : Contient les classes d'exceptions personnalisées, telles que `MapException`, pour gérer les erreurs spécifiques au jeu de manière élégante.
- **res** : Stocke les ressources nécessaires au jeu, comme les fichiers image et son, ainsi que les fichiers de niveaux prédéfinis.

2.2 Diagramme

Le diagramme ci-dessous représente la hiérarchie de répertoires du projet, fournissant une vue d'ensemble de l'organisation du code.

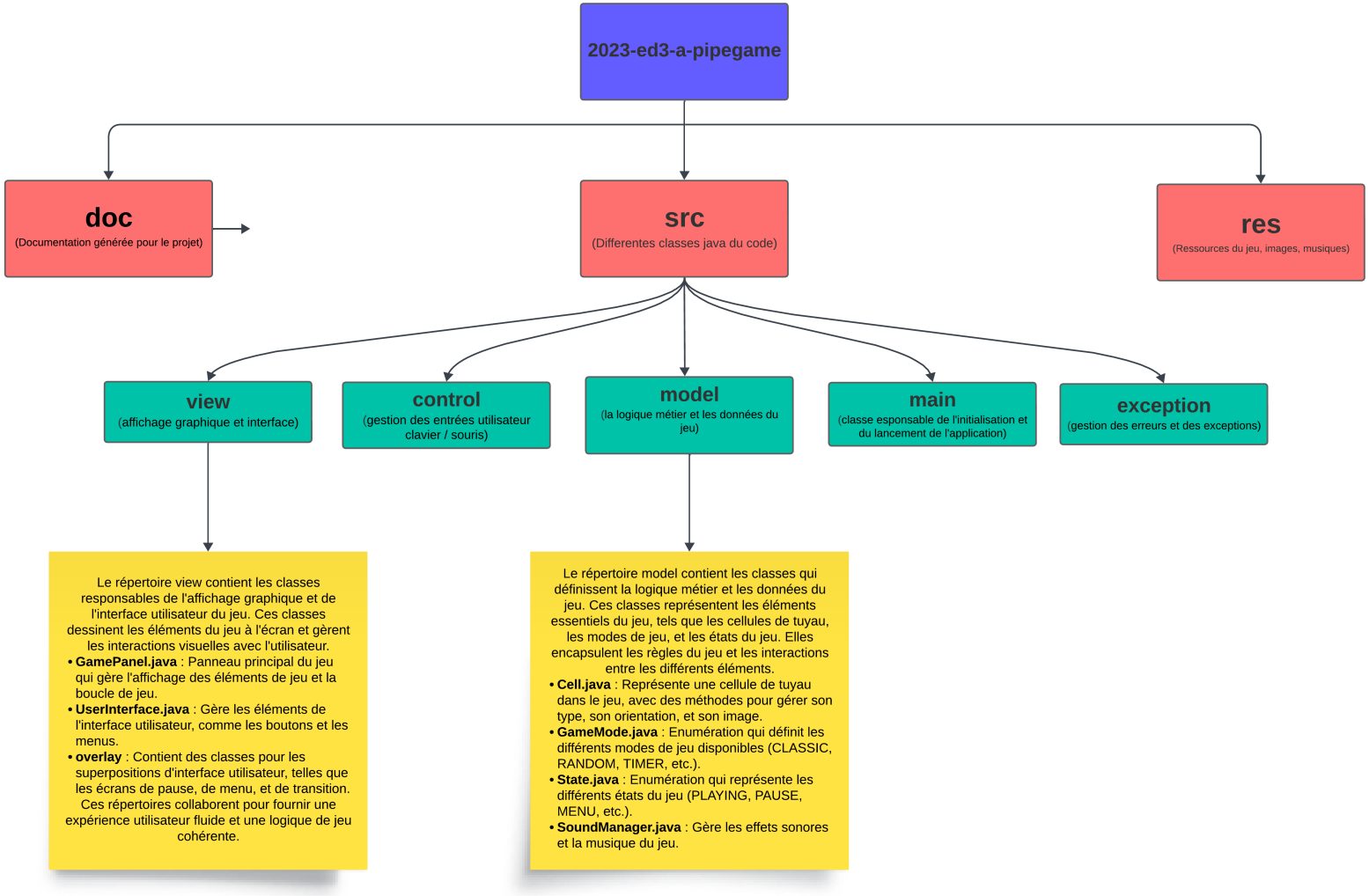


Figure 1: Diagramme de la hiérarchie des répertoires du projet

2.3 Détails des classes principales

- **Map** : Cette classe gère l'état du jeu, y compris le niveau actuel, le nombre minimum de mouvements, les mouvements du joueur, et la logique pour vérifier les connexions entre les tuyaux.
- **Cell** : Représente une cellule du jeu (un tuyau). Elle stocke les informations sur le type de tuyau, son orientation, et ses connexions possibles.
- **Play** : Coordonne le déroulement du jeu en fonction du mode sélectionné (CLASSIC, TIMER, RANDOM, etc.). Elle gère également le passage entre les niveaux et l'état général du jeu.
- **GamePanel** : La classe principale pour le rendu graphique du jeu. Elle contient la boucle de jeu, dessine les éléments à l'écran et gère les changements d'état du jeu.
- **SoundManager** : Gère les effets sonores et la musique de fond, améliorant l'immersion et l'expérience utilisateur.

3 Code et Algorithmes

Lorsqu'on a commencé à travailler sur le projet, la première question qui s'est posé est : comment visualisons nous les "tuyaux" du jeu. Après plusieurs versions différentes nous sommes arrivés à une qui satisfait nos besoin. Cette section décrit la représentation des tuyaux dans le jeu ainsi que l'algorithme utilisé pour vérifier si tous les tuyaux sont correctement connectés.

3.1 Représentation de la Classe des Tuyaux

Dans notre jeu, chaque tuyau est représenté par une instance de la classe `Cell`. Cette classe encapsule les propriétés et les comportements d'un tuyau, y compris son type, son orientation, et les connexions possibles.

Listing 1: Définition de la classe `Cell`

```
public class Cell {
    private int pipeType; // Type du tuyau (1: d part , 2: arriv e ,
       etc.)
    private int orientation; // Orientation du tuyau (0: haut , 1:
       droite , 2: bas , 3: gauche)
    private boolean[] connections; // Connections possibles (haut ,
       droite , bas , gauche)

    public Cell(int pipeType , int orientation , boolean curve) {
        this.pipeType = pipeType;
        this.orientation = orientation;
        this.connections = new boolean[4];
        initializeConnections(curve);
    }

    private void initializeConnections(boolean curve) {
        // Initialiser les connexions bas es sur le type de tuyau et l
           'orientation
    }
```

```
    public boolean[] getCon() {
        return connections;
    }

    public void rotate(SoundManager soundManager) {
        // Faire pivoter le tuyau et mettre    jour les connexions
    }

    public void draw(Graphics2D g2, int x, int y, int tileSize) {
        // Dessiner le tuyau    l' cran
    }
}
```

Chaque instance de `Cell` contient un type de tuyau (`pipeType`), une orientation (`orientation`), et un tableau de connexions (`connections`) indiquant les directions dans lesquelles le tuyau peut se connecter à d'autres tuyaux.

3.2 Algorithme de Vérification des Connexions

L'algorithme de vérification des connexions est utilisé pour déterminer si tous les tuyaux du niveau sont correctement connectés. Cet algorithme est basé sur une recherche en profondeur (DFS) pour explorer toutes les connexions possibles à partir des tuyaux de départ.

Listing 2: Algorithme de vérification des connexions

```
public boolean parcoursProfondeurRec() {
    boolean valid = true;
    for (int[] t : first) {
        valid &= explorer(start[t[0]][t[1]], t[0], t[1]);
    }
    if (!valid) {
        for (Cell[] c_row : start) {
            for (Cell c : c_row) {
                if (c != null && !c.isChecked()) {
                    c.setConnected(false);
                }
            }
        }
    }
    return valid;
}

public boolean explorer(Cell s, int x, int y) {
    boolean[] con = s.getCon();
    boolean b = true;
    s.setChecked();
    if (s.getPipeType() != 1 && !s.isConnected()) {
        s.setConnected(true);
    }
    for (int i = 0; i < 4; i++) {
        if (con[i]) {
            if (i == 0) {
                if (x <= 0 || start[x-1][y] == null || !start[x-1][y].
                    getCon()[2])
                    b = false;
```

```
                else if (!start[x-1][y].isChecked())
                    b &= explorer(start[x-1][y], x-1, y);
            } else if (i == 1) {
                if (y >= start[x].length-1 || start[x][y+1] == null ||
                   !start[x][y+1].getCon()[3])
                    b = false;
                else if (!start[x][y+1].isChecked())
                    b &= explorer(start[x][y+1], x, y+1);
            } else if (i == 2) {
                if (x >= start.length-1 || start[x+1][y] == null || !
                   start[x+1][y].getCon()[0])
                    b = false;
                else if (!start[x+1][y].isChecked())
                    b &= explorer(start[x+1][y], x+1, y);
            } else {
                if (y <= 0 || start[x][y-1] == null || !start[x][y-1].
                   getCon()[1])
                    b = false;
                else if (!start[x][y-1].isChecked())
                    b &= explorer(start[x][y-1], x, y-1);
            }
        }
    }
    return b;
}
```

3.2.1 Description de l'Algorithme

1. **Initialisation** : La méthode `parcoursProfondeurRec` initialise la vérification en parcourant tous les tuyaux de départ (stockés dans `first`).

2. **Exploration** : La méthode `explorer` utilise une recherche en profondeur pour explorer toutes les connexions possibles à partir d'un tuyau donné. Chaque tuyau est marqué comme "vérifié" une fois exploré pour éviter les boucles infinies.

3. **Validation des Connexions** : Pour chaque direction (haut, droite, bas, gauche), l'algorithme vérifie si la connexion est valide. Si toutes les connexions sont valides, le tuyau est marqué comme "connecté".

4. **Résultat Final** : Si toutes les connexions sont valides, la méthode `parcoursProfondeurRec` retourne `true`, indiquant que tous les tuyaux sont correctement connectés. Sinon, elle retourne `false` et marque les tuyaux non connectés.

Cet algorithme permet de vérifier efficacement si tous les tuyaux du niveau sont correctement connectés, en garantissant que chaque tuyau est exploré une seule fois et en validant les connexions directionnelles de manière exhaustive.

3.2.2 Étapes Détailées de l'Algorithme

- **Initialisation** : La méthode `parcoursProfondeurRec` commence par parcourir tous les tuyaux de départ, qui sont stockés dans une liste appelée `first`. Cette liste contient les coordonnées des tuyaux de départ.

- **Exploration** : Pour chaque tuyau de départ, la méthode `explorer` est appelée. Cette méthode utilise une recherche en profondeur (DFS) pour explorer toutes les connexions possibles à partir de ce tuyau. Chaque tuyau exploré est marqué comme "vérifié" pour éviter les boucles infinies.

- **Validation des Connexions** : L'algorithme vérifie les connexions dans les quatre directions cardinales (haut, droite, bas, gauche). Si une connexion est trouvée, l'algorithme continue à explorer à partir de la cellule connectée. Si une connexion n'est pas valide, le tuyau est marqué comme non connecté.

- **Résultat Final** : Après avoir exploré toutes les connexions possibles, la méthode `parcoursProfondeurRec` retourne `true` si tous les tuyaux sont correctement connectés. Sinon, elle retourne `false` et marque les tuyaux non connectés pour indiquer les erreurs de connexion.

Justification de la Complexité

L'algorithme de recherche en profondeur (DFS) est choisi pour sa capacité à explorer exhaustivement toutes les connexions possibles à partir de chaque tuyau de départ. Bien que cet algorithme puisse être coûteux en termes de temps pour des niveaux très complexes, il garantit que toutes les connexions sont vérifiées de manière exhaustive, assurant ainsi la validité des connexions de tuyaux dans le jeu. La complexité de l'algorithme est proportionnelle au nombre de tuyaux et à la complexité des connexions, mais il reste gérable pour les tailles de niveaux typiques dans notre jeu.

3.3 Algorithme de Génération de Niveaux Aléatoires

Dans le mode de jeu "RANDOM", les niveaux sont générés de manière procédurale pour offrir une variété infinie de configurations de tuyaux. Cet algorithme utilise une approche de génération de labyrinthe pour s'assurer que tous les tuyaux sont connectés de manière cohérente.

Listing 3: Algorithme de génération de niveaux aléatoires

```
public void generateRandom() {
    Cell[][] maze = new Cell[6][6];
    for (int i = 0; i < 6; i++) {
        for (int j = 0; j < 6; j++) {
            maze[i][j] = new Cell(0, 0, false);
        }
    }

    fillMaze(maze, 0, 0, 0);

    Random random = new Random();
    for (int i = 0; i < 6; i++) {
        for (int j = 0; j < 6; j++) {
            maze[i][j] = new Cell(maze[i][j].getPipeType(), random.
                nextInt(4), maze[i][j].isCurve());
        }
    }

    this.start = maze;
    this.first.add(new int[]{0, 0});
    resetCells();
```

```
}

private void fillMaze(Cell[][] maze, int i, int j, int d) {
    maze[i][j].setChecked();
    ArrayList<Integer> neighbours = getNeighbors(maze, i, j);

    if (neighbours.isEmpty()) {
        maze[i][j].setPipeType(1);
        first.add(new int[]{i, j});
    }

    for (Integer dir : neighbours) {
        if (dir == 1 && !maze[i-1][j].isChecked()) {
            maze[i][j].incrPipeType();
            maze[i-1][j].incrPipeType();
            if (d == 2 || d == 4) maze[i][j].setCurve();
            fillMaze(maze, i-1, j, 1);
        } else if (dir == 2 && !maze[i][j+1].isChecked()) {
            maze[i][j].incrPipeType();
            maze[i][j+1].incrPipeType();
            if (d == 1 || d == 3) maze[i][j].setCurve();
            fillMaze(maze, i, j+1, 2);
        } else if (dir == 3 && !maze[i+1][j].isChecked()) {
            maze[i][j].incrPipeType();
            maze[i+1][j].incrPipeType();
            if (d == 2 || d == 4) maze[i][j].setCurve();
            fillMaze(maze, i+1, j, 3);
        } else if (dir == 4 && !maze[i][j-1].isChecked()) {
            maze[i][j].incrPipeType();
            maze[i][j-1].incrPipeType();
            if (d == 1 || d == 3) maze[i][j].setCurve();
            fillMaze(maze, i, j-1, 4);
        }
    }
}

public static ArrayList<Integer> getNeighbors(Cell[][] maze, int i, int
    j) {
    ArrayList<Integer> neighbors = new ArrayList<>();

    if (i != 0 && !maze[i-1][j].isChecked()) {
        neighbors.add(1);
    }
    if (j != maze.length - 1 && !maze[i][j+1].isChecked()) {
        neighbors.add(2);
    }
    if (i != maze.length - 1 && !maze[i+1][j].isChecked()) {
        neighbors.add(3);
    }
    if (j != 0 && !maze[i][j-1].isChecked()) {
        neighbors.add(4);
    }

    Collections.shuffle(neighbors);

    return neighbors;
}
```

3.3.1 Description de l'Algorithme

1. **Initialisation** : La méthode `generateRandom` initialise un labyrinthe de cellules avec des tuyaux vides. Chaque cellule est initialisée avec un tuyau de type 0 (vide), orientation 0, et sans courbe.

2. **Exploration et Connexion** : La méthode `fillMaze` utilise une recherche en profondeur (DFS) pour explorer et connecter les cellules adjacentes. Les connexions entre les cellules sont créées en incrémentant le type de tuyau et en ajoutant des courbes lorsque nécessaire. Cette méthode assure que chaque cellule est visitée et connectée de manière cohérente.

3. **Ajout de Courbes** : Des courbes sont ajoutées aux connexions pour créer des chemins variés. Cela se fait en ajustant le type de tuyau et son orientation en fonction de la direction de la connexion et de l'orientation actuelle.

4. **Rotation Aléatoire** : Après avoir connecté toutes les cellules, chaque tuyau est orienté de manière aléatoire à l'aide de la classe `Random`. Cela ajoute un élément de difficulté et de variété, rendant chaque niveau unique.

3.3.2 Étapes Détailées de l'Algorithme

- **Initialisation** :
  - La grille de jeu est représentée par un tableau 2D de cellules (`Cell[][] maze`).
  - Chaque cellule est initialisée avec un tuyau vide (`new Cell(0, 0, false)`).
- **Exploration et Connexion** :
  - La méthode `fillMaze` commence à la cellule (0, 0) et utilise DFS pour explorer les voisins.
  - Pour chaque cellule visitée, la méthode vérifie les voisins non visités (`getNeighbors`) et les connecte en ajustant le type de tuyau.
  - Les connexions sont faites en incrémentant le type de tuyau et en ajoutant des courbes si nécessaire pour maintenir la continuité du labyrinthe.
- **Ajout de Courbes** :
  - Les courbes sont ajoutées pour créer des chemins variés et intéressants.
  - Cela implique de modifier le type de tuyau et son orientation en fonction de la direction de la connexion.
- **Rotation Aléatoire** :
  - Une fois toutes les cellules connectées, chaque tuyau est orienté de manière aléatoire.
  - Cela est accompli en générant un nombre aléatoire pour chaque cellule et en définissant cette valeur comme nouvelle orientation du tuyau.
  - La classe `Random` est utilisée pour introduire cette variabilité.

L'algorithme de génération de niveaux aléatoires est conçu pour créer des niveaux jouables et variés tout en garantissant que tous les tuyaux sont connectés de manière cohérente. Voici quelques points clés de justification :

- **Cohérence** : L'utilisation de DFS pour explorer et connecter les cellules garantit que chaque tuyau est relié de manière cohérente, évitant les chemins morts et les connexions incorrectes.
- **Variété** : L'ajout de courbes et la rotation aléatoire des tuyaux introduisent une grande variété dans les niveaux générés, rendant chaque partie unique.
- **Efficacité** : Bien que DFS puisse être coûteux en termes de temps pour des grilles très grandes, il est suffisamment performant pour les tailles de niveaux typiques de notre jeu (6x6 cellules).
- **Jouabilité** : En ajustant les types de tuyaux et en ajoutant des courbes, l'algorithme crée des niveaux intéressants et stimulants pour les joueurs.

En conclusion, cet algorithme de génération de niveaux aléatoires offre un bon équilibre entre complexité, variété et jouabilité, ce qui est essentiel pour maintenir l'intérêt et l'engagement des joueurs.

4 Problèmes rencontres

Nous avons rencontrés beaucoup d'obstacles à certaines fonctions que nous voulions implémenter, que ce soit à cause des limitations imposés par le sujet, ou de bugs voici les plus grosses difficultés auxquelles nous avons fait face :

4.1 Rotation d'image

La première semaine après le début du projet, nous avions implémenté une structure basique pour la classe des tuyaux et on a donc voulu faire tourner les tuyaux de 90 degré lorsqu'on clique dessus. Monumentale erreur, ça a été 2h continu de bugs graphiques en tout genre, les tuyaux changeaient de couleur, se tordaient, s'effritaient et faisaient tout sauf ce qu'il fallait.

Au final nous avons fait une rotation direct de 90 degré non animé, son équation mathématique fonctionne très bien dans notre code.

Listing 4: Méthode `rotateImage`

```
public void rotateImage() {
    int width = image.getWidth();
    int height = image.getHeight();

    BufferedImage dest = new BufferedImage(width, height, image.getType
        ());

    Graphics2D graphics2D = dest.createGraphics();

    graphics2D.translate((height - width) / 2, (height - width) / 2);
    graphics2D.rotate(Math.PI / 2, width / 2, height / 2);
    graphics2D.drawRenderedImage(image, null);

    this.image = dest;
}
```

4.2 Calcul du Nombre Minimum de Coups

Dans notre projet, nous avons initialement envisagé d'utiliser un algorithme de backtracking pour calculer le nombre minimum de coups nécessaires pour résoudre chaque niveau. Cependant, en raison de la complexité et des limitations de notre structure de données, nous avons dû opter pour une approche alternative.

L'algorithme de backtracking est une technique de recherche exhaustive qui explore toutes les configurations possibles pour trouver une solution optimale. Bien que cet algorithme soit efficace pour certains problèmes, il présente plusieurs défis dans le contexte de notre jeu de tuyaux :

- **Complexité Exponentielle** : Le backtracking implique d'explorer toutes les combinaisons possibles de rotations et de connexions de tuyaux. Étant donné que chaque tuyau peut être orienté dans quatre directions différentes, le nombre de configurations possibles augmente exponentiellement avec le nombre de tuyaux. Cette complexité rend l'algorithme impraticable pour des niveaux plus grands ou plus complexes.

- **Structure des Données** : Notre structure de données actuelle utilise une grille de cellules (`Cell`) pour représenter les tuyaux. Chaque cellule contient des informations sur les connexions et les orientations possibles. Manipuler cette structure de manière efficace pour le backtracking est complexe et sujet à des erreurs, surtout lorsqu'il s'agit de vérifier les connexions et les chemins valides.

- **Performance** : L'implémentation du backtracking nécessite des vérifications fréquentes des connexions et des orientations, ce qui peut être coûteux en termes de temps de calcul. Dans un contexte de jeu où la réactivité est essentielle, un algorithme de backtracking pourrait entraîner des délais inacceptables pour les joueurs.

4.2.1 Approche Alternative

Compte tenu de ces défis, nous avons décidé de baser le nombre de coups minimum sur un multiple du nombre de tuyaux présents dans chaque niveau. Plus précisément, nous avons fixé le nombre de coups minimum pour obtenir trois étoiles à quatre fois le nombre de tuyaux dans le niveau.

$$\text{Nombre de coups minimum} = 4 \times \text{Nombre de tuyaux} \tag{1}$$

Cette approche présente plusieurs avantages :

- **Simplicité** : Cette méthode est simple à implémenter et ne nécessite pas de calculs complexes ou d'algorithmes de recherche exhaustive.
- **Équité** : En fixant le nombre de coups minimum en fonction du nombre de tuyaux, nous garantissons une difficulté cohérente et équilibrée pour chaque niveau.
- **Performance** : Cette approche est beaucoup plus rapide et efficace, ce qui améliore l'expérience utilisateur en réduisant les temps de calcul.

4.2.2 Justification

Nous avons choisi le facteur 4 après avoir testé plusieurs niveaux et observé que ce multiplicateur offrait un bon équilibre entre défi et faisabilité. Ce nombre permet aux joueurs de faire quelques erreurs tout en restant dans une limite raisonnable pour obtenir le score maximal.

En conclusion, bien que le backtracking soit une technique puissante, ses limitations pratiques dans le contexte de notre jeu nous ont conduits à adopter une solution plus simple et plus performante, assurant ainsi une expérience de jeu optimale pour les utilisateurs.

5 Conclusion

Ce projet a été une expérience incroyablement enrichissante et amusante. Nous avons non seulement appris beaucoup sur les algorithmes et la programmation orientée objet, mais nous avons également acquis une solide compréhension de la gestion de projet et de l'importance du travail collaboratif.

Nous sommes particulièrement fiers de notre capacité à utiliser Git de manière efficace pour le contrôle de version et la collaboration. Chaque membre de l'équipe a pu contribuer de manière significative, et nous avons su fragmenter le développement en plusieurs petites étapes, rendant ainsi la gestion du projet plus fluide et organisée. Cette méthodologie nous a permis de surmonter les défis complexes et d'assurer une progression constante vers nos objectifs.

Enfin, nous tenons à remercier notre professeur et nos camarades pour leur soutien et leurs conseils tout au long de ce projet. Leur aide précieuse a été un moteur essentiel de notre réussite.

Sincères salutations,

L'équipe de développement