Université de Lille

Département informatique

SAÉ 3.02

RAPPORT UE 2

Membre de l'équipe

Samy Van Calster
Lisa Haye
Valentin Hebert
Abdallah Toumji

11 janvier **2023 2024**

Sommaire

Génération de labyrinthe

Génération Aléatoire

Pour la création du plateau, nous avons opté pour la génération d'un terrain entièrement aléatoire pour chaque case. L'attribut "wallProbability" de la classe plateau est utilisé pour définir la probabilité qu'une case soit un mur ou une case vide.

```
public class Plateau {

   protected static double wallPropabilitie = 0.2;
   private static final Random rand = new Random();
   protected ICellEvent[][] cells;
   protected static final int DEFAULT_ROW = 20;
   protected static final int DEFAULT_COL = 20;
```

En utilisant les constructeurs, nous spécifions les dimensions du plateau.

```
public Plateau(int nbRow, int nbCol) {
   this.cells = new ICellEvent[nbRow][nbCol];
}

public Plateau() {
   this(DEFAULT_ROW, DEFAULT_COL);
}
```

Ensuite, nous avons développé la méthode createArea, qui permet de générer les cases du tableau comme précédemment expliqué.

```
public ICellEvent[][] createArea(double probability, int height, int weight) {
   double test;
   ICellEvent[][] area = new ICellEvent[height][weight];
   for (int row = 0; row < height; row++) {
       for (int col = 0; col < weight; col++) {
           test = rand.nextDouble();
           area[row][col] = (test <= probability)? new Cell(CellInfo.WALL, new
Coordinate(row, col)) : new Cell(CellInfo.EMPTY, new Coordinate(row, col));
       }
   }
   return area;
}
```

Algo avancé pour la génération

Ensuite, dans une superclasse Maze étendant Plateau, nous avons ajouté plusieurs coordonnées en tant qu'attributs, comme représenté ci-dessous :

- **hunterCo** : les dernières coordonnées du chasseur (par défaut, à null)
- **monsterCo** : l'emplacement du monstre (par défaut, l'entrée du labyrinthe)
- **exitCo** : l'emplacement de la sortie pour la victoire du monstre

```
public class Maze extends Plateau {

   // Coordinate actuel du chasseur
   private ICoordinate hunterCo;

   // Coordinate actuel du monster
   private ICoordinate monsterCo;

   // Coordinate actuel de la sortie
   private ICoordinate exitCo;
```

Cela nous a permis de mettre en place une méthode pour vérifier la validité d'un terrain de jeu, en fonction de l'existence d'un chemin possible entre deux cellules. La méthode **isPathPossibleDFS** utilise une **recherche en profondeur (DFS)** pour déterminer la faisabilité d'un chemin entre deux cellules voisines du labyrinthe.

```
private boolean isPathPossibleDFS(int currentLine, int currentColumn, int endLine, int
endColumn, boolean[][] visited) {

   if (currentLine == endLine && currentColumn == endColumn) {
       return true;
   }

   visited[currentLine][currentColumn] = true;

   int[] rowOffsets = { -1, 1, 0, 0 };
   int[] colOffsets = { 0, 0, -1, 1 };

   for (int i = 0; i < 4; i++) {
       int newRow = currentLine + rowOffsets[i];
       int newColumn = currentColumn + colOffsets[i];

       if (isInMaze(newRow, newColumn) && !visited[newRow][newColumn] &&
cells[newRow][newColumn].getState() != CellInfo.WALL) {
           return isPathPossibleDFS(newRow, newColumn, endLine, endColumn, visited);
       }
   }
   return false;
}
```

Ensuite, nous avons mis en place une méthode qui utilise notre **isPathPossibleDFS**, mais avec l'échelle de deux coordonnées.

```
public boolean isPathPossible(int startLine, int startColumn, int endLine, int
endColumn) {
   if (!isInMaze(startLine, startColumn) || !isInMaze(endLine, endColumn)) {
       return false;
   }
   boolean[][] visited = new boolean[cells.length][cells[0].length];

   return isPathPossibleDFS(startLine, startColumn, endLine, endColumn, visited);
}
```

Grâce à cette méthode, nous sommes en mesure de tester l'existence d'un chemin entre l'entrée (la première position du monstre) et la sortie. Si aucun chemin n'est trouvé, nous utilisons une autre méthode pour déterminer le chemin le plus court.

```
private ICoordinate shortPathCellFor(ICoordinate start, ICoordinate end) {
   int dx = end.getCol() - start.getCol();
   int dy = end.getRow() - start.getRow();

   if (Math.abs(dx) > Math.abs(dy)) {
       if (dx > 0) {
           int newRow = start.getRow();
           int newCol = start.getCol() + 1;
           return new Coordinate(newRow, newCol);
       } else if (dx < 0) {
           int newRow = start.getRow();
           int newCol = start.getCol() - 1;
           return new Coordinate(newRow, newCol);
       }
   } else {
       if (dy > 0) {
           int newRow = start.getRow() + 1;
           int newCol = start.getCol();
           return new Coordinate(newRow, newCol);
       } else if (dy < 0) {
           int newRow = start.getRow() - 1;
           int newCol = start.getCol();
           return new Coordinate(newRow, newCol);
       }
   }
   return start;
}
```

Ensuite, avec une autre méthode, nous construisons le chemin le plus court.

```
private List<ICoordinate> createPath(ICoordinate monsterCell, ICoordinate exitCell) {

   List<ICoordinate> res = new ArrayList<>();
   ICoordinate currentCoordinate = monsterCell;

   if (isInMaze(monsterCell) && isInMaze(exitCell)) {
       while (!currentCoordinate.equals(exitCell)) {
           res.add(currentCoordinate);
           currentCoordinate = shortPathCellFor(currentCoordinate, exitCell);
       }
       res.add(currentCoordinate);
   }
   return res;
}
```

Algorithme pour les IA

L'algorithme utilisé pour les intelligences artificielles est l'algorithme A*, implémenté dans la classe AstarNode.java. Cet algorithme est chargé de plusieurs aspects cruciaux, tels que le calcul des heuristiques, la construction du chemin, et la validation des mouvements.

```
public static List<ICoordinate> aStarPathfinding(Maze board) {
        ICoordinate start = board.getMonsterCo();
        ICoordinate goal = board.getExitCo();

        List<ICoordinate> openSet = new ArrayList<>();
        List<ICoordinate> closedSet = new ArrayList<>();
        openSet.add(start);

        Map<ICoordinate, ICoordinate> cameFrom = new HashMap<>();

        Map<ICoordinate, Integer> gScore = new HashMap<>();
        gScore.put(start, 0);
        Map<ICoordinate, Integer> fScore = new HashMap<>();
        fScore.put(start, heuristic(start, goal));

        while (!openSet.isEmpty()) {
            ICoordinate current = getLowestFScore(openSet, fScore);
            if (current.equals(goal)) {
                return reconstructPath(cameFrom, goal);
            }

            openSet.remove(current);
            closedSet.add(current);

            for (int[] direction : DIRECTIONS) {
                int x = current.getCol() + direction[0];
                int y = current.getRow() + direction[1];
                Coordinate neighbor = new Coordinate(x, y);

                if (!isValidMove(board, neighbor) || closedSet.contains(neighbor)) {
                    continue;
                }

                int tentativeGScore = gScore.getOrDefault(current, Integer.MAX_VALUE) +
1;

                if (!openSet.contains(neighbor)) {
                    openSet.add(neighbor);
                } else if (tentativeGScore >= gScore.getOrDefault(neighbor,
Integer.MAX_VALUE)) {
                    continue;
                   }
                cameFrom.put(neighbor, current);
                gScore.put(neighbor, tentativeGScore);
                fScore.put(neighbor, tentativeGScore + heuristic(neighbor, goal));
            }
        }
   return Collections.emptyList();
}
```

Dans le contexte de la classe AstarNode.java, l'algorithme A* est mis en œuvre pour effectuer le calcul des heuristiques. Ces heuristiques sont des estimations de la distance entre deux points dans un espace donné. Elles jouent un rôle crucial dans le processus de prise de décision de l'IA, permettant ainsi de déterminer la meilleure voie à suivre.

```
private static int heuristic(ICoordinate a, ICoordinate b) {
    return Math.abs(a.getCol() - b.getCol()) + Math.abs(a.getRow() - b.getRow());
}
```

De plus, la classe AstarNode.java est également chargée de construire le chemin optimal pour l'IA. Cela signifie qu'elle évalue les différentes options disponibles et choisit la séquence de mouvements qui mène à la destination tout en minimisant le coût total, en tenant compte des heuristiques calculées précédemment .

```
private static List<ICoordinate> reconstructPath(Map<ICoordinate, ICoordinate> cameFrom,
ICoordinate target) {
        List<ICoordinate> path = new ArrayList<>();
        ICoordinate current = target;
        while (current != null) {
            path.add(current);            current = cameFrom.get(current);
        }
        Collections.reverse(path);
        return path;
}
```

De plus, la classe AstarNode.java est également responsable de la construction du chemin optimal pour l'IA. Cela implique l'évaluation des différentes options disponibles, le choix de la séquence de mouvements menant à la destination tout en minimisant le coût total, en prenant en compte les heuristiques calculées précédemment.

```
private static boolean isValidMove(Maze board, Coordinate coord) {
    int col = coord.getCol();
    int row = coord.getRow();
    int sizeCol = board.getSizeCol();
    int sizeRow = board.getSizeRow();

    boolean withinBounds = col >= 0 && col < sizeCol && row >= 0 && row < sizeRow;
    return withinBounds && board.get(row,
col).getState().equals(ICellEvent.CellInfo.EMPTY);
}
```

Cet algorithme est utilisé à la fois pour le monstre et le chasseur. Leur implémentation diffère dans leur manière d'agir. Par exemple, le chasseur va rechercher une piste du monstre puis lancer l'algorithme pour anticiper les actions du monstre.

Malheureusement, en raison d'une contrainte de temps, nous n'avons pas pu l'implémenter dans les classes IAMonsterStrategy.java et IAHunterStrategy.java.

Structures de données

Classe Game

ArrayList pour stocker les vues

- ➔ L'emploi d'ArrayList dans la classe Game pour stocker les vues comporte divers avantages. Une ArrayList est une structure de données dynamique en Java qui peut accueillir un nombre variable d'éléments. Dans le contexte de notre jeu, cette souplesse est bénéfique, car le nombre de vues peut varier tout au long de la partie.
- ➔ L'utilisation d'ArrayList facilite la gestion flexible des vues grâce à l'ajout et à la suppression dynamiques d'éléments. À la différence d'un tableau de taille fixe, il n'est pas nécessaire de spécifier une taille initiale, simplifiant ainsi la gestion des vues qui peuvent être ajoutées ou retirées au fil de l'évolution du jeu.

Intégration des interfaces IMonsterStrategy et IHunterStrategy

- ➔ L'intégration de ces interfaces (IMonsterStrategy et IHunterStrategy) montre que nous avons adopté une approche fondée sur la programmation par interface. Cette approche offre une flexibilité considérable, car elle permet de changer les stratégies du monstre et du chasseur sans nécessiter de modifications directes dans le code source de notre jeu.
- ➔ Les interfaces fournissent une abstraction en distinguant la définition du comportement des objets de leur implémentation concrète. Cette approche encourage la modularité du code et simplifie la tâche de maintenance.

Classe MazeCellPane

Association type de cellule - Image

- ➔ Une **HashMap** est utilisée pour établir une correspondance entre le type de cellule (clé) et son image respective (valeur). Cette méthode simplifie la gestion des images associées à chaque type de cellule dans notre jeu.

Recherche rapide à partir du type de cellule

- ➔ La structure de données HashMap permet une recherche rapide de l'image correspondant à un type de cellule spécifique. En exploitant la clé (représentant le type de cellule), l'accès direct à la valeur (l'image) associée est facilité, améliorant ainsi l'efficacité, notamment lors du rendu du labyrinthe.

Facilité de gestion dynamique

- ➔ La HashMap offre une flexibilité permettant d'ajouter, de supprimer ou de modifier les associations entre les types de cellules et les images. Cette caractéristique peut se révéler particulièrement utile lorsque les types de cellules évoluent au cours de l'exécution du programme, comme l'ajout de nouveaux éléments au fur et à mesure de la progression du jeu.

Classe Maze

Utilisation de tableaux bidimensionnels pour le labyrinthe et les vues

- ➔ L'utilisation fréquente de tableaux à deux dimensions pour représenter des structures de grille, telles qu'un labyrinthe, est pertinente. Le recours à cette structure pour stocker le labyrinthe et les différentes vues est avisé lorsque la taille du labyrinthe est connue à l'avance.
- ➔ L'accès direct aux cases du tableau simplifie la manipulation du labyrinthe. Cette caractéristique revêt une importance particulière dans un contexte de jeu, où il est fréquent de devoir vérifier ou modifier l'état d'une cellule spécifique.
- ➔ Il est également cohérent d'utiliser un tableau à deux dimensions, car le labyrinthe est déjà modélisé sous cette forme.

Facilité d'instanciation et de mise en œuvre.

- ➔ En connaissant préalablement la taille des tableaux, leur instanciation devient simple, ce qui peut améliorer la lisibilité du code. Cela élimine également le besoin de redimensionner dynamiquement les tableaux.
- ➔ L'accès aux éléments du tableau est intuitif, ce qui facilite la manipulation du labyrinthe et des différentes vues, rendant ainsi les opérations plus simples et plus efficaces.

Classe AStarNode et IAMonsterStrategy

Efficacité en ressources :

➔ Les tableaux à deux dimensions sont généralement plus efficaces en termes de consommation de ressources par rapport à certaines structures de données plus complexes. Ils offrent un accès direct aux éléments, ce qui peut réduire la surcharge liée à la recherche et à la manipulation des données.

Simplicité d'accès :

➔ Les tableaux à deux dimensions offrent une simplicité d'accès, ce qui est crucial pour stocker des informations sur les directions possibles. L'accès direct à une cellule spécifique du tableau facilite la récupération des données sans nécessiter des opérations complexes.

Amélioration de la lisibilité du code :

➔ L'utilisation de tableaux à deux dimensions peut améliorer la lisibilité du code, car elle reflète souvent de manière intuitive la structure des données qu'elle représente. Cela rend le code plus compréhensible pour les développeurs qui examinent le code ultérieurement.

Potentiel pour des améliorations futures :

➔ Si les directions possibles peuvent évoluer ou être étendues à l'avenir, l'utilisation d'un tableau à deux dimensions offre la flexibilité nécessaire pour ajouter de nouvelles directions sans avoir à réorganiser entièrement la structure de données.

Efficacité

Les choix d'algorithme et de structure de données effectués dans le cadre du développement du jeu démontrent une approche réfléchie visant à optimiser les performances et la flexibilité du code. Voici une synthèse des raisons pour lesquelles ces choix sont efficaces :

Utilisation d'ArrayList dans la classe Game

Une ArrayList est rapide à parcourir et elle ne prend pas plus d'espace mémoire que nécessaire étant donné que son attribution est dynamique c'est pourquoi nous l'avons utilisé pour gérer les petites listes des observateurs.

Utilisation de HashMap dans la classe MazeCellPane

La structure de données HashMap facilite la gestion des images associées à chaque type de cellule. La recherche rapide basée sur le type de cellule permet un rendu efficace du labyrinthe.

Utilisation de tableaux bidimensionnels dans la classe Maze, AStarNode et IAMonsterStrategy

Les tableaux à deux dimensions sont appropriés pour modéliser un labyrinthe, offrant un accès direct aux cases et étant donc très rapide et efficace ce qui permet de ne pas ralentir le jeu malgré de nombreux accès à ces tableaux.

Génération Aléatoire :

Avantages
➔ **Efficacité avec la probabilité :** En utilisant une probabilité pour déterminer si une case est un mur ou vide, il est seulement nécessaire de faire une opération par case qui est une inéquation. Cette méthode permet d'ajuster la difficulté du jeu en modifiant simplement cette probabilité, tout en maintenant une vitesse d'exécution optimale.
➔ **Efficacité d'ajustement des dimensions :** En spécifiant les dimensions du plateau dans les constructeurs, l'instanciation du plateau est efficace peu importe les dimensions. En effet la rapidité d'exécution reste constante, indépendamment de la taille du plateau car il n'est pas nécessaire de créer un plateau d'une taille non désirée seulement pour le modifier plus tard.
Limitations / Axes d'amélioration
➔ **Manque de connectivité garantie :** L'approche de génération purement aléatoire peut conduire à des labyrinthes non connectés, rendant le jeu impossible ou trop facile. Une amélioration pourrait consister à introduire une étape de validation pour garantir que le labyrinthe est utilisable du labyrinthe, permettant ainsi d'avoir un labyrinthe de qualité à chaque fois.

Algorithme Avancé pour la Génération

Avantages
➔ **Validation de la validité du labyrinthe :** L'utilisation de la méthode isPathPossibleDFS, qui incorpore un parcours en profondeur, pour valider la possibilité d'un chemin entre deux cellules est la méthode la plus efficace pour garantir un labyrinthe solvable. ➔ **Calcul du chemin le plus court :** La méthode shortPathCellFor pour la construction d'un chemin optimal est une fonctionnalité efficace car elle ne nécessite pas beaucoup d'itération afin de trouver ledit chemin améliorant l'efficacité de création du labyrinthe.
Limitations / Axes d'amélioration
➔ **Complexité du code :** En cas de labyrinthe insolvable un chemin en ligne droite vers la sortie sera "forcé" rendant le jeu prévisible dans cette situation. Un axe d'amélioration consisterait à supprimer des murs au hasard jusqu'à ce qu'un chemin soit possible mais cela pourrait demander beaucoup de ressources lors de la création d'un labyrinthe

Algorithmes pour les Intelligences Artificielles (IA) :

Avantages
➔ **Utilisation de l'algorithme A*** : L'algorithme A* est un choix solide pour la résolution de chemins, offrant une recherche efficace et la possibilité de trouver le chemin le plus court. ➔ **Sauvegarde :** En sauvegardant le résultat de l'algorithme A* Il n'est pas nécessaire de le répéter plusieurs fois, préservant ainsi un temps exponentiel à la longueur du chemin.
Limitations / Axes d'amélioration
➔ **Contrainte de temps** : La non-implémentation dans les classes IAMonsterStrategy.java et IAHunterStrategy.java en raison d'une contrainte de temps est compréhensible, mais cela laisse une opportunité d'amélioration pour une version future.

Conclusion :

En général, nos choix d'algorithme et de structure de données montrent une compréhension approfondie des exigences du jeu, avec des solutions pour la validation des chemins, et les comportements des intelligences artificielles. Les axes d'amélioration potentiels sont liés à la création de méthodes de vérification et à l'implémentation des parties non réalisées en raison de contraintes de temps. Pour aller plus loin, des ajustements pour garantir la variabilité des labyrinthes générés pourraient également être envisagés.

En résumé, ces choix se sont révélés efficaces en termes de rapidité d'exécution, d'utilisation des ressources et de nombres d'utilisations, envisager des évolutions futures du jeu qui pourraient nécessiter des ajustements ou des extensions dans le code actuel.