Rapport développement efficace

Baptiste Bertout – author · Pierre Planchon – author · Arthur Keller – author · Gaspard Souliez – author · Mathis Decoster – author

Table des matières

1. Description

1.1 Génération de labyrinthe

1.1.1 Algorithme utilisé

Le labyrinthe est généré à l'aide de la classe `GenerateBooleanMaze`, qui utilise une approche itérative pour créer un labyrinthe aléatoire.

La méthode `generateBooleanMazeMonster(int height, int width)` de cette classe initialise un tableau de booléens représentant le labyrinthe du monstre.

L'algorithme utilise une approche aléatoire pour définir les emplacements des mûrs en fonction du pourcentage spécifié dans les paramètres.

La méthode `isMazeSolvable()` de la classe `Board` utilise une approche de recherche en profondeur (*DFS*) pour déterminer si le labyrinthe est jouable.

Elle explore itérativement les possibilités de déplacement du monstre depuis sa position initiale jusqu'à la sortie, en utilisant une pile pour suivre le chemin exploré.

Si le monstre peut atteindre la sortie, la méthode retourne true, sinon elle retourne false.

Cette vérification est cruciale pour garantir qu'un labyrinthe jouable est généré lors dès l'initialisation.

1.2 IA

1.2.1 Monster

L'intelligence artificielle (*IA*) du monstre est fondée sur l'application de l'algorithme de Dijkstra, afin d'identifier le chemin optimal vers la sortie du labyrinthe. L'implémentation de cet algorithme est encapsulée au sein de la classe `IAMonstre`.

1.2.1.1 Algorithme de Dijkstra

L'algorithme de parcours en largeur est exploité pour déterminer le plus court chemin entre la position actuelle du monstre et la sortie du labyrinthe. La méthode `findShortestPath()` coordonne cet algorithme, intégrant une structure de données de file (*Queue*) pour effectuer une exploration des différents déplacements possibles.

1.2.1.1 Choix du prochain mouvement

La méthode `chooseNextMove()` joue un rôle crucial dans la décision du prochain mouvement du monstre. En cas de découverte d'un chemin par l'algorithme de Dijkstra, le monstre opte pour une stratégie intelligente en se déplaçant vers la deuxième coordonnée du chemin, garantissant ainsi un cheminement optimisé.

1.2.2 Hunter

L'intelligence artificielle (*IA*) du Hunter repose sur un système de portée. Dans un premier temps, il cherche une piste en tirant aléatoirement sur la labyrinthe, puis s'il trouve une trace du monstre, il estime une zone dans laquelle il va dcevoir tirer aléatoirement au prochain tour, pour avoir plus de probabilités de trouver le monstre. Ce système est implémenté dans la classe `IAHunter` .

1.2.2.1 Pseudo code

PSEUDOCODE

```
méthode play() :
Coordonnée c

si (aucun chemin trouvé) :
    c -> coordonnée d'une des cellules aléatoire du tableau.

    si (on ne peut pas ajouter la cellule à la liste)
        on redémarre la méthode.

sinon :
    portée -> tour actuel - tour de la dernière case découverte.

    si (la portée est supérieur à 1) :
        portée -> portée / 2

    on créer une liste "L" de cases, dans un rayon égal à la portée autour de la
dernière case découverte.

    c -> une valeur aléatoire de la liste "L".

retourne : c
```

2. Structures de données

2.1 Génération de labyrinthe

Les principales structures de données utilisées sont des tableaux à double dimension (boolean[][] et int[][]) pour représenter le labyrinthe, ainsi que des variables entières pour stocker les positions du monstre et de la sortie. Examinons de manière plus détaillée comment ces structures de données sont utilisées

2.1.1 Variable 'tab'

Description

Représente le labyrinthe sous forme d'un tableau 2D de booléens (true pour une case vide, false pour un mûr).

Utilisation

Initialisée et manipulée dans les méthodes de la classe Board et de la classe GenerateBooleanMaze.

La méthode `generateBooleanMazeMonster()` remplit ce tableau en plaçant aléatoirement des mûrs.

Les méthodes` randomMonsterPosition()` et `randomExitPosition()` utilisent ce tableau pour éviter de placer le monstre et la sortie sur des mûrs.

La méthode `isMazeSolvable()` utilise ce tableau pour déterminer la solvabilité du labyrinthe.

2.1.2 Variable 'positions'

Description

Représente les coordonnées réservées dans le labyrinthe pour la sortie (les coins du labyrinthe).

Utilisation

Initialisée dans la classe GenerateBooleanMaze et utilisée pour éviter de placer des mûrs sur ces positions.

Utilisée dans les méthodes `randomMonsterPosition()` et `randomExitPosition()` pour s'assurer que le monstre et la sortie ne sont pas placés sur des positions réservées.

2.1.3 Variable statiques (static int)

Description

Stockent les coordonnées du monstre et de la sortie.

Utilisation

Les variables xMonstre, yMonstre, xExit, et yExit sont utilisées pour représenter les positions du monstre et de la sortie dans le labyrinthe.

Ces variables sont statiques, ce qui signifie qu'elles appartiennent à la classe plutôt qu'à une instance spécifique.

2.1.4 Algorinthme de parcours en profondeur

Description

Utilisé dans la méthode `isMazeSolvable()` pour déterminer si le labyrinthe est resolvable.

Utilisation

Une pile (Deque<int[]> stack) est utilisée pour effectuer le parcours en profondeur.

Les coordonnées du monstre sont ajoutées à la pile, et l'algorithme explore les voisins pour trouver un chemin jusqu'à la sortie.

2.2 IA du Monstre

2.2.1 File (Queue)

La file est utilisée dans l'algorithme de parcours en largeur (Dijkstra) pour explorer systématiquement les différentes possibilités de déplacement à partir de la position actuelle du monstre. Cette structure de données suit le principe "premier entré, premier sorti" et garantit que les coordonnées sont explorées dans un ordre stratégique.

Utilisation

La file est employée dans la méthode `findShortestPath()` de la classe `IAMonstre`.

2.2.2 Ensemble (Set)

Un ensemble est utilisé pour suivre les coordonnées déjà visitées afin d'éviter de revisiter les mêmes emplacements. Cela optimise l'algorithme en éliminant les redondances lors de l'exploration.

Utilisation

L'ensemble est utilisé dans l'algorithme de parcours en largeur (Dijkstra) pour vérifier si une coordonnée a déjà été visitée, dans la méthode `findShortestPath()` de la classe `IAMonstre`.

2.2.3 Liste (List)

Une liste est utilisée pour stocker le chemin optimal trouvé par l'algorithme de Dijkstra. Cela permet un accès facile aux coordonnées successives du chemin.

Utilisation

La liste est employée dans la méthode `chooseNextMove()` pour déterminer le prochain mouvement du monstre.

2.3 IA du Chasseur

2.3.1 Ensemble (Set)

Un ensemble est utilisé pour suivre les cases acuellement découverte afin d'éviter au chasseur de retirer sur ces mêmes cases, ce qui permet de gagner en efficacité.

Utilisation

Cet ensemble est utilisé dans la méthode `Play()` mais n'est utilisé seulement tant que le chasseur n'a pas découvert de case où le monstre est passé.

En effet dès lors que le chasseur découvre un point de passage, l'ensemble des cases découvertes redeviennent inconnues pour l'IA.

2.3.2 Liste (List)

Une liste est utilisée pour stockées les cases adjacentes de la dernière casse représentant un point de passage du monstre.

Utilisation

Cette liste est utilisé seulement dans la méthode `Play()` pour pouvoir fournir un tire aléatoire parmi l'ensemble des cases adjacentes stockées.

2.3.3 Variable 'tabHunter'

Représente un tableau à double dimensions de ICellEvent permettant de stocker les cases découvertes par le chasseur ainsi que leur informations associées.

Utilisation

Ce tableau est utilisé dans différentes méthodes de la classe `IAHunter` mais l'est principalement dans la méthode `aroudCell()`, ce qui permet d'avoir les cases adjacentes comme l'explique la section `2.3.2`.
Ce tableau permet de selectionner les cases adjacentes et de les garder si elles ne sont pas des mûrs, ce qui permet de n'avoir que des bonnes possiilitées pour le prochain tir du chasseur.

3. Efficacité

3.1 Génération du labyrinthe

L'efficacité de la génération du labyrinthe repose sur l'utilisation d'une approche aléatoire pour définir les emplacements des mûrs.

Cela permet de créer des labyrinthes variés, tout en garantissant qu'ils soient jouables.

La méthode `isMazeSolvable()` utilise une recherche en profondeur (*DFS*) pour vérifier que le monstre peut atteindre la sortie, assurant ainsi que le labyrinthe soit jouable.

3.2 IA du monstre

3.2.1 Algorithme de Dijkstra

Efficacité

L'utilisation de l'algorithme de parcours en largeur (Dijkstra) est pertinente dans le contexte de la recherche du chemin le plus court vers la sortie du labyrinthe. Cet algorithme garantit que le monstre explore d'abord les chemins les plus proches, conduisant ainsi à une prise de décision basée sur des informations actuelles et optimales.

Nombre d'opérations

Le nombre d'opérations est proportionnel au nombre de nœuds (cellules) explorés. Dans le scénario le plus défavorable, où le monstre explore la totalité du labyrinthe, la complexité est relativement élevée, cependant, elle est souvent optimisée par les caractéristiques spécifiques du labyrinthe.

3.2.2 Utilisation de File (*Queue*) et Ensemble (*Set*)

Efficacité

L'utilisation de la file pour l'exploration et de l'ensemble pour suivre les coordonnées déjà visitées contribue à l'efficacité de l'algorithme. La file permet une exploration systématique, et l'ensemble évite la réexploration inutile d'emplacements déjà visités.

Nombre d'opérations

L'utilisation de la file et de l'ensemble contribue à la gestion efficace des coordonnées et minimise les redondances.

3.2.3 Stratégie de choix du prochain mouvement

Efficacité

La stratégie de déplacement, basée sur le chemin optimal identifié par l'algorithme de Dijkstra, démontre son efficacité dès lors que des itinéraires viables sont disponibles. Elle garantit que le monstre se déplace de manière judicieuse et optimale.

Aspect Décevant/Intéressant

Toutefois, en cas d'absence de chemin optimal, l'adoption d'un mouvement aléatoire peut être perçue comme décevante.

3.3 IA du Chasseur

L'IA du chasseur n'est pas vraiment intelligente mais elle l'est un minimum pour que les chance de trouver le monstre soit accrue par rapport à celle que l'on aurait eu si nous avions fait une IA aléatoire.

Nous avons essayer au maximum de réduire l'inégalité entre le Monstre et le chasseur et faisant en sorte qu'il tire dans une zone diminué à partir du moment où il a trouvé une portion de chemin emprunté par le Monstre.

Plus la différence entre le tour actuel et le tour de la case découverte, plus la zone à découvrir est petite.

Malheureusement, les tirs aléatoire ne rivalise pas avec la découverte de chemin de l'IA du Monstre ou la vue d'un joueur humain. Le chasseur ne trouve que très rarement le monstre.

Ce qui aurait pu être sujet d'amélioration, c'est le début. Lors des premières recherches du monstre et son chemin il aurait peut être judicieux de diviser les recherches successivement en 4 quarts de terrains.

Last updated 2024-01-15 23:53:40 +0100