DORNY Nathan
BERRAKANE Adham
CATTANEO Théo
BARBEAU Simon
Groupe : G3

Kryyptonn | Limule45 | Adham59 | Simon

S3.02 : Chasse aux Monstres

-

Développement d'Application (Partie Dev Efficace)

Rappel de rendus :

- ☑ ~~- Description Génération de labyrinthe~~
- ☑ ~~- Description IA~~
- ☑ ~~- Structures de données~~
- ☑ ~~- Efficacité~~

SOMMAIRE

Cliquable

I] Description de la génération du labyrinthe :

Méthode de Recherche de Chemin avec BFS dans un Labyrinthe :

La méthode asValidPathToExitWithBFSAlgo implémente un algorithme de recherche de chemin utilisant la stratégie BFS (Breadth-First Search) pour déterminer s'il existe un chemin valide du monstre à la sortie dans un labyrinthe.

Fonctionnalités clés :

- **Recherche en Largeur (BFS)** : L'algorithme parcourt le labyrinthe en explorant les voisins du monstre en largeur, en utilisant une file d'attente pour organiser les coordonnées à explorer.

- **Poids pour prioriser** : La méthode utilise des poids associés à chaque coordonnée pour prioriser la recherche. Les voisins avec un poids plus petit, calculé en fonction de la distance à la sortie, sont explorés en priorité.

- **Gestion des murs** : Même si un voisin a un poids égal à la coordonnée actuelle, il est ajouté à la file d'attente si la cellule associée n'est pas un mur. Cela permet de contourner les obstacles et d'explorer d'autres options.

- **Condition de sortie** : La méthode s'arrête dès qu'elle atteint la sortie du labyrinthe. L'algorithme retourne true si un chemin valide est trouvé, sinon false.

Cette méthode est implémentée dans la classe Game.java sous le nom de asValidPathToExitWithBFSAlgo(). Cet algorithme va permettre la vérification de au moins un chemin possible lors de l'initialisation de la map. Si l'algorithme n'en trouve pas alors une nouvelle initialisation est faite jusqu'à ce qu' un chemin soit possible dans la map générée.

L'initialisation avant utilisation de l'algorithme est une initialisation qui est random pour les murs, random pour le spawn du monstre mais que sur la première colonne et random pour la sortie sur les 3 dernières colonnes du labyrinth.
Le taux de spawn des murs est ajustable en pourcentage depuis le menu des paramètres.

Donc ici l'utilisation de cet algorithme nous permet ici d'avoir une infinité de maps jouables.

D'un points de vue modulaire :

- La méthode encapsule la logique de recherche de chemin, ce qui permet une réutilisation facile du code. Il nous est possible d'appeler avec différents labyrinthes sans avoir à réécrire le code d'exploration à chaque fois.

D'un point de vue qualité du labyrinthe :

- L'utilisation des poids pour prioriser les voisins permet une recherche plus intelligente et efficace. Les poids sont basés sur la distance à la sortie, ce qui améliore la qualité de la recherche en favorisant les chemins plus courts.

- La gestion intelligente des murs permet au monstre de contourner les obstacles. Même si un voisin a un poids égal, il est exploré s'il n'est pas un mur, ce qui contribue à une meilleure qualité de la recherche.

Voici la méthode asValidPathToExitWithBFSAlgo() implémentée ci dessous :

```java
public boolean asValidPathToExitWithBFSAlgo(Maze maze) {

    //Partie bonne pour le moment
    ICoordinate exit = maze.getExit();
    ICoordinate monsterPosition = maze.getMonsterPosition();

    if (exit == null || monsterPosition == null) {
        return false;
    }

    Queue<ICoordinate> queue = new LinkedList<>();
    Set<ICoordinate> visited = new HashSet<>();
    queue.add(monsterPosition);
    visited.add(monsterPosition);

    while (!queue.isEmpty()) {
        ICoordinate current = queue.poll(); // On prend le premier element de la queue

        if (current.getCol() == exit.getCol() && current.getRow() == exit.getRow()) {
            return true;
        }

        for (ICoordinate neighbor : maze.getNeighbourgs(current)) {
            ICoordinate neighborWeightLess = maze.getCellWeigthToExit(neighbor) <= maze.getCellWeigthToExit(current) ? neighbor : current;
            if (!visited.contains(neighborWeightLess) && this.maze.getCellGroundInfo(neighborWeightLess) != CellInfo.WALL) {
                queue.add(neighborWeightLess);
                visited.add(neighborWeightLess);
            }
            if(neighborWeightLess == exit) {
                return true;
            }
        }
    }
    return false;
}
```

II] IA utilisées

A] Monstre Random

Cet algorithme se base sur une génération de déplacements aléatoires dans le labyrinthe. On pourra dire que le monstre peut sortir du labyrinthe à un temps plus ou moins long en fonction de la chance qu'il aura. Il sera donc peu facile pour le Chasseur s' il s'agit d'un humain de déterminer ou tirer pour trouver sa position.

Voici le pseudo-code de l'algorithme Monstre random "MonsterBotRandom.java" contenu dans le paquetage "fr.univlille.info.G3.model.player.monster".

```
Méthode jouer() : Coordonnée
   ListeDeDeplacementsValides ← vide
   DéplacementBas ← (0, +1)
   DéplacementHaut ← (0, -1)
   DéplacementDroit ← (+1, 0)
   DéplacementGauche ← (-1, 0)
   Si deplacementEstValide(DéplacementBas) :
      ListeDeDeplacementsValides.ajouter(DéplacementBas)
   Fin Si
   Si deplacementEstValide(DéplacementHaut) :
      ListeDeDeplacementsValides.ajouter(DéplacementHaut)
   Fin Si
   Si deplacementEstValide(DéplacementDroit) :
      ListeDeDeplacementsValides.ajouter(DéplacementDroit)
   Fin Si
   Si deplacementEstValide(DéplacementGauche) :
      ListeDeDeplacementsValides.ajouter(DéplacementGauche)
   Fin Si
   Si ListeDeDeplacementsValides est vide :
      Lancer une exception "Le monstre ne peut pas bouger"
   Fin Si
   Choisir un déplacement aléatoire parmi ListeDeDeplacementsValides
   Retourner le déplacement choisi
Fin Méthode
```

Explications Détaillées :

La méthode play va créer une liste vide pour stocker les déplacements possibles. Les déplacements possibles seront ensuite calculés par une méthode dont on entrera en paramètre les coordonnées du nouveau déplacement et ensuite ajouté dans une liste. Les déplacements possibles sont ceux qui ne dépassent pas le terrain ainsi qui ne franchissent pas les murs. Enfin, nous retournons un coordonnée aléatoirement choisi dans la liste de déplacements possibles.

Avantages et Inconvénients :

Il est imprévisible car le monstre se déplace sans forcément se diriger vers la sortie. Il y a aussi peu de stratégie envisageable côté chasseur si c'est un humain puisqu'il faudrait trouver les traces de pas du chasseur et tirer au hasard sur les cases possibles. Les durée de parties peuvent être très longues.

B] Monstre A* :

Pour notre monstre intelligent nous avons décidés d'utiliser l'algorithme A* car il permet de trouver le plus court chemin de manière efficace.

Pour cela, nous avons utilisés plusieurs structures de données :

- un tableau à double entrée d'entier représentant le tour auquel nous sommes passés sur cette case, -1 par défaut.
- un tableau à double entrée d'entier représentant le nombre de pas jusqu'au point de départ, -1 par défaut.
- Les coordonnés du point de départ du monstre
- Les coordonnées de la sortie
-

<u>L'algorithme de comptage des pas marche de la façon suivante :</u>

current <- coordonnées départ monstre
tour de current <- 0
pas de current <- 0
etape = 1
Tant que current différent de coordonnées départ monstre
// Calcul pas des voisins
Coordonnées voisins <- getNeighbourgs(current)
Pour chaque élément de voisins
si pas de voisin = -1
pas de voisin = pas de current + 1
etape current <- étape
étape;
// Choisis prochain current
current = null
Pour chaque ligne
Pour chaque colonne
si étape de la case à la ligne et colonne = -1 et n'est pas un mur
si pas de la case + manhattan de la case jusqu'à coordonnées départ monstre < pas de current + manhattan de current jusqu'à coordonnées départ monstre
current = case
renvoie tableau à double entrée des pas

L'algorithme de comptage des pas marche de la façon suivante :

initialize()

tableau à double entrée des pas = algo comptage des pas

play()

Coordonnées voisins <- getNeighbourgs(case actuel monstre)

Pour chaque voisins

Si pas de voisin < pas de *coordonnées monstre*

renvoie voisin

renvoie null

Nous avons décidé d'utiliser cet algorithme plutôt que celui de Dijkstra car il ne passe pas sur toutes les cases et fait donc moins de calculs.

L'avantage de notre algorithme, c'est qu'il fait tous les calculs lors du premier tour et qu'il est donc extrêmement rapide après ça.

C] Chasseur Random :

Cet algorithme est implémenté dans la classe HunterBotRandom, où un chasseur est créé pour prendre des décisions de manière aléatoire dans un labyrinthe représenté par la classe MazeHunter. La méthode play est chargée de produire des coordonnées aléatoires à chaque étape du jeu. Cette approche confère au chasseur un comportement imprévisible, car il opte de manière aléatoire pour ses actions, ajoutant ainsi une dimension d'incertitude à ses choix au cours du jeu.

Pseudo-code de l'algorithme :

La class HunterBotRandom est Hunter

```
    method initialize(nbRows, nbCols):
    maze = new MazeHunter(nbRows, nbCols)

    method play():
        randomRow = random() * maze.getRowNb()
        randomCol = random() * maze.getColNb()
        return new Coordinate(randomRow, randomCol)
```

Explications détaillées :

La méthode initialize(nbRows, nbCols):

Définit une méthode appelée initialize prenant deux paramètres : nbRows (nombre de lignes) et nbCols (nombre de colonnes).
À l'intérieur de la méthode : on crée une nouvelle instance de la classe MazeHunter avec les dimensions spécifiées par nbRows et nbCols. Puis on stocke cette instance dans la variable maze.

La méthode play():

Génère un nombre aléatoire entre 0 et 1 (non inclus) et le multiplie par le nombre de lignes du labyrinthe (maze.getRowNb()). Cela donne une position de ligne aléatoire dans le labyrinthe. Stocke cette position de ligne aléatoire dans la variable randomRow. Répète le processus pour les colonnes en générant un nombre aléatoire et le multipliant par le nombre de colonnes du labyrinthe (maze.getColNb()), stockant le résultat dans randomCol. Créer un nouvel objet Coordinate avec les valeurs de randomRow et randomCol. Renvoie cet objet Coordinate.

D] Chasseur Clever :

Pour le fonctionnement de cette algorithme nous avons besoin de nous y prendre étape par étape :

1) Initialisation de la coordonnée de shoot du hunter a une case random :

2) Récupération de toutes les cases qui ont été touchées par le chasseur grâce à une méthode qui va vérifier toutes les cases de la map et cela à tous les tours. Toutes les cases touchées vont être stockées dans une liste.

   Si aucune des cases n'a été touché donc dans le cas du premier du tour on retourne donc la coordonnée d'un cellule aléatoire initialisée plus haut.

3) Ensuite dans toutes les cases qui ont été touchées par le chasseur on va chercher la case qui a été touchée le plus récemment par le monstre. Les coordonnées de cette case vont donc devenir les coordonnées principales a renvoyées.

4) Ensuite nous allons grâce à une méthode de maze enregistrer tous les voisins grâce à la nouvelle coordonnée enregistrer ci dessus.
   Avec cette liste de voisins nous allons récupérer la cellule parmi tous les voisins dont la présence du monstre a été la plus récente.

5) Finalement nous allons enregistrer la coordonnée du prochain coup qui sera donc un la coordonnée d'un des voisins que nous avons pu récupérer à l'étape 4. Et nous allons retourner les coordonnées de la cellule qui sera donc le prochain coup du chasseur.

Voici le pseudo-code de la classe Clever :

Récupérer les cellules touchées par le chasseur :

```
Liste<ICoordinate> cellulesTouchees = nouvelle Liste<ICoordinate>();
pour chaque ligne i de 0 à maze.nbDeLignes() {
   pour chaque colonne j de 0 à maze.nbDeColonnes() {
      si (maze.dernierTirChasseur(new Coordinate(i, j)) != -1) {
         cellulesTouchees.ajouter(nouveau Coordinate(i, j));
      }
   }
}
```

Choisir une position aléatoire dans le labyrinthe :

```
ICoordinate position = nouveau
Coordinate(générerNombreAléatoire(maze.nbDeLignes()),générerNombreAléatoire(maze.
nbDeColonnes()));
```

Vérifier si la position choisie est un mur :

```
si (maze.informationSol(position) == CellInfo.MUR) {
        retourner position;
 }
```

Récupérer les cellules touchées par le chasseur :

```
Liste<ICoordinate> cellulesTouchees = nouvelle Liste<ICoordinate>();

pour chaque ligne i de 0 à maze.nbDeLignes() {
    pour chaque colonne j de 0 à maze.nbDeColonnes() {
        si (maze.dernierTirChasseur(new Coordinate(i, j)) != -1) {
            cellulesTouchees.ajouter(nouveau Coordinate(i, j));
        }
    }
}
```

Si aucune cellule n'a été touchée, retourner la position choisie :

```
SI (cellulesTouchees.taille() == 0) {
  retourner position;
 }
```

Trouver la cellule touchée le plus tard parmi celles touchées :

```
TourMonstreMax ← 0
meilleurePosition ← position

Pour chaque cellule c dans cellulesTouchees {
    Si maze.dernierePrésenceMonstre(c) > tourMonstreMax
      tourMonstreMax = maze.dernierePrésenceMonstre(c)
      meilleurePosition = nouveau Coordinate(c.ligne(), c.colonne())
    Fin Si
Fin Pour
```

Trouver la cellule touchée le plus tard par le monstre parmi les voisins de la meilleure position :
List<ICoordinate> voisins = maze.voisins(meilleurePosition);

pour chaque voisins dans la liste de voisins {
    si (maze.dernierePrésenceMonstre(voisin) > tourMonstreMax) {
        tourMonstreMax = maze.dernierePrésenceMonstre(voisin);
        meilleurePosition = nouveau Coordinate(voisin.ligne(), voisin.colonne());
    }
}

A la toute fin :
retourner le futur coup du chasseur;

III] Structures de données

Au cours de notre projet, nous avons utilisé différentes structures de données :

- Les Listes - ArrayList (Liste extensible) / LinkedList (Liste chaînée)

Dans plusieurs parties du projet, nous avons régulièrement eu recours à l'utilisation de listes, telles que les listes extensibles (ArrayList) et les listes chaînées (LinkedList). Lors de la récupération des voisins d'une cellule, il s'agit d'une liste de ICoordinate qui sera retournée. Pour la créer, nous avons utilisé des ArrayList pour ajouter des éléments dans la liste.

- Les Tableaux et Doubles Tableaux / Matrices

Pour représenter le labyrinthe, nous avons utilisé un tableau d'ICell. Pour la génération des murs de la carte, l'utilisation d'un double tableau de booléens générés aléatoirement s'est avérée plus pratique.

- Les HashSet & Les Queues (Files)

Lors de la génération de la carte, nous avons utilisé un algorithme BST. Pour le fonctionnement de cet algorithme, nous avons préféré utiliser les Queues plutôt que la classe Stack, bien que la réalisation avec des Piles (Stack) aurait également été possible.
Ensuite, pour stocker les chemins possibles dans une liste qui ne conserve pas les éléments en double, les sets sont très utiles. Les set sont des collections qui utilisent une table de hachage pour le stockage s'est révélé très efficace.

IV] Efficacité

⬡ Pour la génération de la Map :

Choix Algorithmiques et Structures de Données pour la Recherche de Chemin dans un Labyrinthe avec BFS :

Efficacité Algorithmique :

- Breadth-First Search (BFS) : Choisi pour sa simplicité et son efficacité dans la recherche du chemin le plus court.

- Poids pour Prioriser : Utilisation de poids basés sur la distance à la sortie pour accélérer la recherche en explorant d'abord les voisins les plus prometteurs.

- Gestion Intelligente des Murs : Évitement des blocages en explorant même les voisins avec un poids égal, tant qu'ils ne sont pas des murs.

Modularité et Maintenance :

- Encapsulation Modulaire : Logique encapsulée dans une méthode modulaire pour faciliter la maintenance et l'évolution du code.

- Adaptabilité à des Labyrinthes Complexes : Possibilité d'optimiser pour des labyrinthes avec des caractéristiques plus complexes.

Axes d'Amélioration :

- Optimisation des Performances : Analyse approfondie des performances et ajustements potentiels pour une efficacité maximale.

- Adaptabilité à des Labyrinthes Variés : Test sur une variété de labyrinthes pour une adaptation optimale.

- Gestion de Chemins Multiples : Extension pour gérer et fournir plusieurs chemins optimaux ou alternatives.

⬡ Le Hunter Bot Clever :

Efficacité Algorithmique :

- Aspect Aléatoire Initial : L'introduction d'une position initiale aléatoire pour le monstre ajoute un degré d'imprévisibilité et de diversité dans le comportement du monstre, ce qui peut être bénéfique pour la variété du jeu.

- Réaction aux coups du chasseur : La logique de réaction aux coups du chasseur en choisissant des positions moins touchées est une stratégie défensive qui peut potentiellement augmenter la survie du monstre.

- Priorisation des Derniers Tours : La priorisation des positions touchées le plus tard par le chasseur et le monstre renforce la probabilité de choisir des positions stratégiques dans le labyrinthe.

Modularité et Maintenance :

- Encapsulation des Logiques : La logique est encapsulée dans une méthode unique, facilitant la maintenance et l'évolution du code. Les modifications futures peuvent être apportées à cet endroit sans perturber d'autres parties du code.

- Utilisation de Structures de Données : L'utilisation de listes (ArrayList) pour stocker les cellules touchées par le chasseur et les voisins permet une manipulation facile des données.

Axes d'Amélioration Possibles :

- Gestion des Séquences Prévisibles : L'algorithme peut être amélioré en tenant compte de schémas prévisibles où le chasseur et le monstre suivent des séquences répétitives. Cela peut rendre le comportement du monstre plus dynamique.

- Adaptabilité Contextuelle : L'introduction de mécanismes d'adaptabilité en fonction du contexte du labyrinthe (ex : présence d'obstacles, configuration spécifique) pourrait rendre le monstre plus réactif aux conditions changeantes.

⬡ Le Monster Bot A Star :

Efficacité Algorithmique :

- Pathfinding Optimisé : Utilisation de l'algorithme A* pour déterminer le chemin le plus court vers la sortie. L'algorithme calcule le chemin le plus efficace en tenant compte des obstacles (murs) et des distances.
- Calcul de Chemin à la Demande : Le chemin est re calculé seulement si nécessaire (quand steps est null). Cela évite les re calculs inutiles à chaque tour et économise des ressources.

Modularité et Maintenance :

- Utilisation de Structures de Données : Utilisation d'un double tableau (int[][] steps) pour représenter le labyrinthe et les étapes du chemin. Cette structure est simple et efficace pour stocker et accéder aux informations de distance pour chaque cellule.

Axes d'Amélioration :

- Adaptation Dynamique de l'Heuristique : Améliorer l'algorithme pour qu'il ajuste dynamiquement l'heuristique en fonction de la configuration actuelle du labyrinthe et des actions du chasseur.

- Gestion des Situations d'Impasse : Introduire des mécanismes pour mieux gérer les situations où le monstre se retrouve dans une impasse, en cherchant des itinéraires alternatifs ou en modifiant temporairement sa stratégie.

- Réduction des coûts de traitements : La mise en cache de certaines parties déjà calculée aurait pu permettre à l'algorithme d'être encore plus rapide.