Groupe H3

S3.02 – Rendu de développement efficace

Ce document est un rapport de la partie algorithmique de notre projet. Ce dernier va décrire chacun des algorithmes ayant été implémentés à la fois concernant la génération du labyrinthe mais aussi concernant les IA utilisées pour le rôle du monstre et celui du chasseur.

Les IA du monstre:

Le Mode Facile:

Description:

Pour ce que l'on a qualifié de mode facile, nous avons opté pour un algorithme de recherche en profondeur (Depth-First Search - DFS) qui parcourt le labyrinthe en suivant systématiquement une séquence de mouvements prédéfinie (nord, ouest, est, sud). Cela rend son parcours généralement assez long puisqu'il ne garantit pas nécessairement de trouver le chemin le plus court vers la sortie. Ces faiblesses nous sont apparus comme un élément d'un bon algorithme pour un mode facile.

Cette algo à été implémenter dans les classes Monster et MonsterView.

Structures de données :
- Liste ("marquer") : Utilisée pour suivre les cases déjà explorées.
- Pile('"stack"):
    - Utilisée pour stocker les coordonnées des cases à explorer.
    - Les coordonnées sont empilées et défilées pendant la recherche en profondeur.

Ces structures sont utilisées dans la classe Monster.

Bien que la file aurait pu être une alternative viable, l'utilisation d'une pile s'est avérée être un choix logique et efficace pour le modèle d'exploration en profondeur du labyrinthe puisque la nature Last In, First Out de la pile s'aligne naturellement avec le concept d'exploration en profondeur.

Pseudo-code:

```
Fonction iaMove():
    Si la pile (stack) est vide:
        Initialiser la pile (initializeStack())

    Tant que la pile n'est pas vide:
        Obtenir la coordonnée actuelle depuis le sommet de la pile
        Si la coordonnée actuelle est la sortie:
            Retourner une nouvelle coordonnée (0, 0)  // Sortie
            atteinte
        Sinon:
            Pour chaque direction (voisin) valide:
                Si le voisin est accessible:
```

```
                    Ajouter la coordonnée du voisin à la liste des
                    cases explorées (marquer)
                    Ajouter la coordonnée du voisin à la pile
                    Retourner une nouvelle coordonnée représentant le
                    mouvement vers le voisin (la coordonnée retournée
                    sera donc (1, 0) ou(0, 1) ou (1, 1) ou (-1, 0)...

            Si la pile n'est pas vide:
                Retourner en arrière en déstackant la coordonnée
                actuelle
                Mettre à jour la coordonnée actuelle
Fin de la fonction
```

Le Mode Difficile:

Description:

L'algorithme que nous avons implémenté pour le mode difficile est une version améliorée de l'algorithme précédent. En effet, il correspond à un algorithme de parcours en profondeur amélioré par une optimisation grâce à l'utilisation de la distance de Manhattan dans un labyrinthe. Cet algorithme prend donc en compte une heuristique qui ici est la distance directe entre le monstre et la sortie). Cela lui permet de trouver la sortie bien plus rapidement qu'un simple algorithme de parcours en largeur.

Structures de données :

- Liste ("marquer") : Utilisée pour suivre les cases déjà explorées.
- Pile('"stack"):
    - Utilisée pour stocker les coordonnées des cases à explorer.
    - Les coordonnées sont empilées et défilées pendant la recherche en profondeur.

Ces structures de données sont utilisées dans la classe Monster.

De même que pour l'algorithme précédent : la pile (Last In, First Out) est adaptée à la gestion d'une exploration en profondeur puisqu'elle permet de traiter en priorité les derniers éléments ajoutés (i.e. la dernière cellule explorée) avant de revenir en arrière. Cela correspond au comportement souhaité pour un parcours en profondeur.

Pseudo code :

```
Fonction iaMoveImproved():
    Si la pile (stack) est vide:
        Initialiser la pile (initializeStack())

    Tant que la pile n'est pas vide:
```

```
        Obtenir la coordonnée actuelle depuis le sommet de la pile
        Si la coordonnée actuelle est la sortie:
            Retourner une nouvelle coordonnée (0, 0)  // Sortie
            atteinte
        Sinon:
            Initialiser une distance minimale à une valeur maximale
            Initialiser nextMove à null

            Pour chaque voisin valide:
                Calculer la distance de Manhattan entre le voisin et la
                sortie

                Si la distance est inférieure à la distance minimale:
                    Mettre à jour la distance minimale et nextMove avec
                    les informations du voisin

            Si nextMove n'est pas null:
                Ajouter la coordonnée du nextMove à la liste des cases
                explorées (marquer)
                Ajouter la coordonnée du nextMove à la pile
                Retourner une nouvelle coordonnée représentant le
                mouvement entre la coordonnée actuelle et
                nextMove // la coordonnée retournée sera donc
                (1, 0) ou (0, 1) ou (1, 1) ou (-1, 0)...
            Sinon:
                Retourner en arrière en dépilant la coordonnée
                actuelle

    Retourner null  // Aucun mouvement possible
Fin de la fonction
```

Les IA du chasseur:

Le Mode Facile:

Description:

Pour la version simple du chasseur, nous avons opté pour une approche totalement aléatoire : le chasseur tire sur une case sans tenir compte des informations qu'il a pu récolter.

Cet algorithme a été implémenté dans la classe HunterView.

Objet Random :

- Description : L'objet Random est une classe du langage de programmation qui génère des nombres aléatoires.
- Utilisation : Il est utilisé pour générer des indices de colonne et de ligne aléatoires dans la grille du labyrinthe.

Pseudo-code:

```
Fonction playAISimple():
    Créer un nouvel objet Random (random)
    Obtenir un nombre aléatoire pour la colonne (randomColumn) compris
    entre 0 et le nombre de colonnes du labyrinthe - 1
    Obtenir un nombre aléatoire pour la ligne (randomRow) compris entre
    0 et le nombre de lignes du labyrinthe - 1

    Obtenir l'objet représentant le chasseur depuis le labyrinthe
    (maze.getHunter())
    Appeler la méthode hit() du chasseur avec les coordonnées
    aléatoires (randomColumn, randomRow)

    Retourner un nouvel objet Coordinate avec les coordonnées
    aléatoires (randomColumn, randomRow)
Fin de la fonction
```

Le Mode Difficile:

Description:

Pour la version complexe du chasseur, il a été choisi un système également basé sur l'aléatoire mais prenant cette fois-ci en compte les informations récoltées lors des précedents tours. Par exemple, si le chasseur a découvert une case sur laquelle le monstre est passé il y a 3 tours, le prochain tir du chasseur se fera sur les 3 cases entourant celle découverte par le chasseur. Bien sûr, si le chasseur a découvert plusieurs cases sur lesquelles le monstre est passé, alors la case choisie comme point de repère sera celle sur laquelle le monstre est passé le plus récemment.

Cette algorithme a été implémenté dans la classe HunterView.

Objet Random :

- Description : L'objet Random est une classe du langage de programmation qui génère des nombres aléatoires.
- Utilisation : Il est utilisé pour générer des indices de colonne et de ligne aléatoires dans la grille du labyrinthe.

Pseudo-code:

```
Fonction playAIHard():
    Créer un nouvel objet Random (random)
    Obtenir les coordonnées de la cellule où le monstre a été vu pour
    la dernière fois (myCell) en appelant
    maze.findShortedLastAppearance()
    Initialiser des variables pour le déplacement aléatoire :
        - int moreOrLess
        - int randomColumn
        - int randomRow
        - boolean end = false

    // Déplacement horizontal
    Tant que end est faux:
        moreOrLess = random.nextInt(2)
```

```
        Si moreOrLess est 0:
            randomColumn = myCell.getColumn() +
            random.nextInt(maze.getMaze()[myCell.getColumn()][myCell.ge
            tRow()].getLastMonsterAppearanceReverse(maze.getCompteur())
            + 1)
        Sinon:
            randomColumn = myCell.getColumn() -
            random.nextInt(maze.getMaze()[myCell.getColumn()][myCell.ge
            tRow()].getLastMonsterAppearanceReverse(maze.getCompteur())
            + 1)

        Si randomColumn est dans les limites du labyrinthe:
            end = vrai

    end = faux

    // Déplacement vertical
    Tant que end est faux:
        moreOrLess = random.nextInt(2)
        Si moreOrLess est égal à 0:
            randomRow = myCell.getRow() +
            random.nextInt(maze.getMaze()[myCell.getColumn()][myCell.ge
            tRow()].getLastMonsterAppearanceReverse(maze.getCompteur())
            + 1)
        Sinon:
            randomRow = myCell.getRow() -
            random.nextInt(maze.getMaze()[myCell.getColumn()][myCell.ge
            tRow()].getLastMonsterAppearanceReverse(maze.getCompteur())
            + 1)

        Si randomRow est dans les limites du labyrinthe:
            end = vrai

    // Appliquer le déplacement
    Obtenir l'objet représentant le chasseur depuis le labyrinthe
    (maze.getHunter())
    Appeler la méthode hit() du chasseur avec les nouvelles coordonnées
    (randomColumn, randomRow)

    Retourner un nouvel objet Coordinate avec les nouvelles coordonnées
    (randomColumn, randomRow)
Fin de la fonction
```

La génération du labyrinthe :

Tout d'abord il faut savoir que la génération du labyrinthe s'effectue en plusieurs parties:

1. la génération des cellules,
2. Le placement de l'entrée et de la sortie
3. La génération des obstacle
4. Vérification qu'un chemin existe entre l'entrée et la sortie
5. L'affichage
6. Le processus

Toutes les méthodes permettant la génération du labyrinthe sont implémentées dans la classe **Maze** et utilisées dans la classe **IHM.**

Génération des cellule du labyrinthe :

Premièrement, la classe **Maze** qui correspond au labyrinthe contient un constructeur qui prend en paramètre le nombre de colonnes et le nombre de lignes de celui-ci.

Notre labyrinthe sera un tableau à deux dimensions de Cellules (**Cell**) qui sera instancié dans le constructeur de notre classe **Maze.**

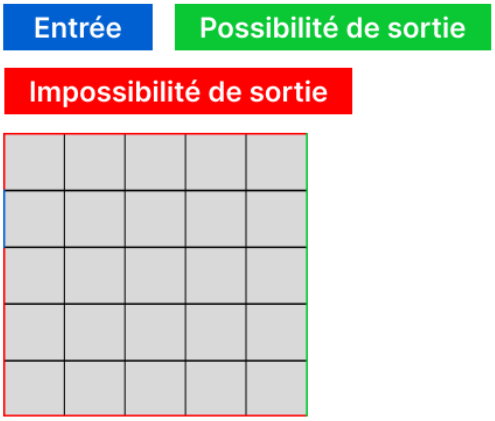
L'objet **Cell** prend en paramètre de son constructeur des coordonnées (**Coordinate(colonne, ligne)**) et une **CellInfo** qui correspond à une énumération dénombrant les éléments à mettre dans la cellule: Vide, Monstre, Entrée…

La fonction permettant la génération du labyrinthe fonctionne de cette manière:

```
Fonction resetMaze():
    On parcourt le tableau de cellules instancié dans le constructe
    et on met toutes les cellules de notre labyrinthe à vide
(CellInfo.EMPTY)
Fin de la fonction
```

Le placement de l'entrée et de la sortie:

Pour permettre la génération d'une entrée et d'une sortie ayant une distance assez longue pour permettre un peu de difficulté côté monstre, nous avons choisi de générer l'entrée et la sortie face à face mais toujours de manière aléatoire.



Dans le cas ci-dessus, la sortie sera générée aléatoirement dans une des cellules longeant la ligne verte (l'entrée peut apparaître en bas en haut à gauche ou à droite).

La fonction permettant le placement de l'entrée et de la sortie fonctionne de cette manière:

```
Fonction generateEnterExit():
      Initialiser un objet Random pour générer un nombre aléatoire
      Générer un nombre aléatoire entre 0 et 3 inclus
      Initialiser les coordonnées de l'entrée et de la sortie
      Sélectionner la position de l'entrée et de la sortie en fonction de
      la valeur aléatoire
      Si rand est égal à 0:
```

```
placer l'entrée aléatoirement dans la ligne du haut et la sortie
aléatoirement dans la ligne du bas
placer l'entrée aléatoirement dans la colonne de droite et la
sortie aléatoirement dans la colonne de gauche
placer l'entrée aléatoirement dans la ligne du bas et la sortie
aléatoirement dans la ligne du haut
placer l'entrée aléatoirement dans la colonne de gauche et la
sortie aléatoirement dans la colonne de droite
pour finir placer le monstre aux mêmes coordonnées que la sortie
Fin du traitement.
```

Cette approche nous permet de maintenir un niveau de difficulté adéquat même si le joueur (monstre) a une idée approximative de l'emplacement de la sortie. Les obstacles contribuent à maintenir une complexité malgré cette connaissance du joueur.

La génération des obstacles :

La complexité de cette implémentation réside principalement dans la contrainte qui stipule qu'elle doit être effectuée à partir d'un pourcentage. Voici donc comment nous avons implémenté cela.

Tout d'abord, pour mieux comprendre l'implémentation, il faut savoir que 100% correspond au nombre de cellules du labyrinthe auquel on soustrait la distance entre la case (0, 0) et la dernière case du labyrinthe(tout en bas à droite). Illustré ci-dessous:

Le chemin en rouge représente le plus court chemin du plus long chemin possible (c'est-à-dire si on a une entrée qui se place en 0, 0 et une sortie en bas à droite ou inversement).
Les case bleues représentent alors le nombre maximum d'obstacles que l'on pourrait générer pour être sur qu'il y ait toujours au moins un chemin possible dans notre labyrinthe entre l'entrée et la sortie.

Le calcul pour obtenir le nombre d'obstacle maximum est le suivant :

(nbColonne x nbLigne - 2) - ((nbColonne - 1) + (nbLigne - 1) - 1)

Donc dans l'exemple au dessus cela nous donne :

(5 x 5 - 2) - ((5 - 1) + (5 - 1) - 1) = 16

Ici, le 100% se traduit donc par 16.

La fonction permettant la génération des obstacle fonctionne de cette manière:

```
fonction genererObstacles(pourcentageObstacles):
      si pourcentageObstacles < 0 ou pourcentageObstacles > 100 alors
      lancer une exception avec le message "Le pourcentage d'obstacles
      doit être compris entre 0 et 100."
      Convertir le paramètre en nombre d'obstacles à partir du maximum
      d'obstacles possible comme vu plus haut(parametre * maxObstacle /
      100)
      tant que i < nombre d'obstacles
      générer des x et Y aléatoires
      vérifier que les coordonnées(x, y) sont vide, si oui:
      placer l'obstacle et incrémenter i
      sinon recommencer
Fin du traitement.
```

Un chemin existe-il entre l'entrée et la sortie :

Enfin, avant d'aller plus loin, il faut vérifier qu'un chemin existe bel est bien entre l'entrée et la sortie sinon le monstre ne pourra jamais atteindre cette dernière:

Pour ce faire nous avons opté pour l'algorithme BFS qui va rechercher un chemin possible entre l'entrée et la sortie.

La fonction permettant cela est une fonction de type boolean qui retournera true si un chemin existe et false sinon.

La fonction permettant la génération des obstacle fonctionne de cette manière:

```
fonction existeChemin():
    Récupérer les coordonnées de l'entrée et de la sortie
    Vérifier si les coordonnées d'entrée ou de sortie sont manquantes
    si entrée est null ou sortie est null alors
        retourner faux
    fin du traitement

    Initialiser la matrice des distances
    Initialiser la file de priorité avec la comparaison basée sur les
distances


    Initialiser les déplacements possibles

    Initialiser la distance à l'entrée comme 0 et l'ajouter à la file de
priorité

    Boucle principale de l'algorithme BFS
    tant que la filePriorité n'est pas vide faire
        courant = filePriorité.retirer()
        Vérifier si le nœud courant est la sortie
        si l'état de la cellule à la position courante est égal à l'état
de la sortie alors
            retourner vrai // Chemin trouvé
        fin du traitement
```

```
        Parcourir les voisins du nœud courant
        pour chaque direction de 0 à 3 faire
            prochainX = courant.colonne + dx[direction]
            prochainY = courant.rangée + dy[direction]

            Vérifier les limites du labyrinthe
            si prochainX est dans [0, colonnes-1] et prochainY est dans
[0, rangées-1] alors
                Vérifier si la cellule est un mur et si elle n'a pas
encore été visitée
                si l'état de la cellule à la position prochainX,
prochainY n'est pas un mur
                        et distances[prochainX][prochainY] est égal à la
valeur maximale d'entier alors
                    Mettre à jour la distance et ajouter le voisin à la
file de priorité
                    distances[prochainX][prochainY] =
distances[courant.colonne][courant.rangée] + 1
                    filePriorité.ajouter(nouvelle Coordinate(prochainX,
prochainY))
                fin si
            fin si
        fin pour
    fin tant que
    // Aucun chemin trouvé
    retourner faux
fin du traitement
```

Nous avons choisi d'utiliser une file de priorité (priority queue). Effectivement, dans le contexte de la recherche de chemin le plus court, la file de priorité permet d'explorer en priorité les nœuds qui sont plus proches du point de départ.

L'affichage du tout :

En ce qui concerne l'affichage, nous avos deux fonctions d'affichage : une dans HunterView (elle correspond à l'affichage du labyrinthe pour le chasseur) et une dans MonsterView (elle correspond à l'affichage du labyrinthe pour le monstre).

L'affichage côté chasseur : le chasseur ne voit pas à quoi correspondent les cases du labyrinthe, mis à part après avoir tiré dessus.

```
fonction affichage():
    pour chaque colonne de 0 à nombreDeColonnesDuLabyrinthe - 1 faire
        pour chaque rangée de 0 à nombreDeRangéesDuLabyrinthe - 1 faire
            obtenir le case correspondant à la position (colonne, rangée)
dans la grille
            obtenir la cellule correspondante dans le labyrinthe

            définir l'état de la cellule
            définir la case comme celui d'une cellule cachée

            si la cellule est découverte alors
                afficher le contenu de la cellule
fin fonction
```

Processus :

Dans la classe IHM, le labyrinthe et généré comme suit:

tant qu’un chemin n’existe pas on :

-génère le labyrinthe

-place l’entrée et la sortie

-génère les obstacle

-et on vérifie qu’un chemin existe: si oui la boucle se termine et le jeu se lance sinon la boucle continue

Ce système fonctionne parfaitement pour nos attentes. Le seul inconvénient est que parfois la boucle peut s'effectuer de nombreuses fois, car un chemin n'est pas immédiatement trouvé. Cela ralentit le lancement du jeu de quelques secondes, particulièrement lorsque le pourcentage d'obstacles entré par l'utilisateur approche de 100%.