Contents

1 Développement efficace SAE3.02 G1

Pour réaliser ces algorithmes, nous avons utlisé un tableau à deux dimensions de **Cell** représentant le maze qui, lors de son initialisation, est un tableau de cellule vide rempli au fur et à mesure de la partie pour le hunter ou un tableau rempli avec les bonnes cellules pour le monster.

L'usage d'un tableau à deux dimensions fut un choix logique, car il permet un accès direct à la case correspondante à la coordonnée voulue, ce qui en améliore considérablement la rapidité d'exécution du programme. D'autre part, nous aurions pu utiliser une liste de **Cell** mais cela aurait considérablement réduit l'efficacité programme car il aurait fallu parcourir toute la liste jusqu'à atteindre la cellule voulue.

Nous avons utilisé cette structure de données pour les classes **IAHunterRandom.java**, **IAHunter.java**, **IAMonster.java** et **DFSMonster.java**.

1.1 IA Hunter

Les IA du Hunter sont des IA simples qui se basent sur le peu d'informations auxquelles peut accéder le Hunter ce qui rend le jeu difficile pour elles.
Le développement des IA Hunter est assez limité et le nombre d'informations utilisables est faible et l'accès à ces informations repose sur la chance au début.

1.1.1 Aléatoire

L'algorithme utilisé est personnel. Il se trouve dans la classe **IAHunterRandom.java** et voici le pseudo-code associé :

```
Initialisation de la variable coord (ICoordinate) qui sera retournée à la fin
Initialisation des variables row et col (int)
faire
    row = valeur aléatoire compris 0 et la taille maximale d'une ligne du tableau
    col = valeur aléatoire compris 0 et la taille maximale d'une colonne du tableau
    coord = instanciation d'une nouvelle Coordinate de paramètre row et col
```

```
tant que coord est dans le tableau et que la cellule à la coordonnée de coord est un mur
retourne coord
```

Cet algorithme est très simple et tire aléatoirement sur une case, il permet de jouer contre une IA facile à battre bien que cela rend la tâche facile.
Les choix algorithmiques sont simples, car ils prennent en compte quasiment aucune information et tire au hasard.
La seule information utilisée ets que l'algorithme ne tire pas deux fois sur une même mur.
L'éxecution de cet algorithme est donc très rapide et ne demande pas beaucoup de calculs.

1.1.2 Aléatoire amélioré

L'algorithme utilisé est personnel. Il se trouve dans la classe **IAHunter.java** et voici le pseudo-code associé :

Algorithme principal :

```
Initialisation de la variable coord (ICoordinate) qui sera retournée à la fin
si lastPositionMonster n'est pas null
    around (liste de cellule) = around(lastPositionMonster.getCoord())
    faire
        coord = un élément aléatoire retiré de around
    tant que la cellule à la coordonnée de coord est un mur, donc déjà découverte
sinon
    faire
        coord = nouvelle instance de Coordinate de paramètre,
                entier aléatoire compris entre 0 et taille maximale d'une ligne,
                et entier aléatoire compris entre 0 et taille maximale d'une colonne
    tant que la cellule à la coordonnée de coord est un mur, donc déjà découverte
retourne coord
```

La méthode around prend en paramètre une coordonnée et, en fonction de la portée qui est égale au tour actuel - le dernier tour où le monstre a été trouvé (= toutes les cellules où le monstre pourrait se trouver), retourne une liste de **Cell** autour de la coordonnée d'une portée définie précédemment.

Cet algorithme est plus complexe que le précédent, car il prend en compte les informations des cases où le montre est passé.
Il permet une difficulté un peu plus élevée, mais il est très semblable à de l'aléatoire si le hunter ne tire jamais sur une case où est passé le monstre.
L'éxecution de cet algorithme est donc assez rapide et demande un nombre de calculs assez faible.

1.2 IA Monster

1.2.1 Algorithme A*

L'algorithme utilisé est l'algorithme A. *Il se trouve dans la classe **IAMonster.java** et a été implémenté uniquement pour le Monstre.*
*Pour son implémentation, nous avons utilisé deux ensembles de **IAMonster.Cellule**.*
*Ces **cellules** sont différentes de celles utilisées pour le labyrinthe, elles sont spécifiques à l'implémentation de l'algorithme A.* Elles stockent en plus la distance entre la cellule et la cellule de départ, l'heuristique, et la cellule parente.
Nous avons également utilisé une liste de **ICoordinate** pour stocker le chemin à suivre.
Nous avons fait le choix d'utiliser ces types de Cellule pour la raison que nous avons besoin de stocker des informations supplémentaires sur les cellules, et que l'utilisation d'une classe interne nous paraissait plus logique que l'utilisation d'une classe externe.
De plus, notre classe a besoin d'avoir un accès au labyrinthe du monstre, donc à l'attribut **maze** de la classe **IAMonster**.
Nous avons aussi utilisé un ensemble de cellules déjà visitées, cela nous permet d'être sûr que nous ne visitons pas deux fois la même cellule, et donc de ne pas avoir de boucle infinie.
La méthode peut provoquer des **RuntimeExceptions**, si aucun chemin n'est trouvé dans le labyrinthe ou si le monstre n'est pas correctement initialisé dans le labyrinthe.
L'utilisation d'une liste de **ICoordinate** pour stocker le chemin à suivre nous a paru plus logique que l'utilisation d'une liste de **Cell**, car nous n'avons besoin que des coordonnées.

Cet algorithm est assez complexe et demande un certain nombre de calculs, mais il permet de trouver un chemin optimal rapidement pour jouer avec une difficulté maximale.
C'est l'algorithme idéal, car il trouve le meilleur chemin vers la sortie plus rapidement que l'algorithme de Dijkstra ou aussi vite dans les pires cas.

1.2.2 Algorithme DFS

L'algorithme utilisé est l'algorithme DFS. Il se trouve dans la classe **DFSMonster.java** et a été implémenté uniquement pour le Monstre.
Pour l'implémentation, nous avons utilisé une pile de **ICoordinate** pour stocker le chemin à suivre.
Nous avons également utilisé une liste de **ICoordinate** pour stocker la liste des cellules jusqu'à la cellule de sortie.
Enfin, un ensemble de **DFSMonster.PathCoordinate** a été utilisé pour stocker les cellules déjà visitées.
DFSMonster.PathCoordinate est une classe interne à **DFSMonster** qui stocke une **ICoordinate** et son parent. Cela nous aide à retrouver le chemin à

suivre.
Plutôt que d'utiliser une Map pour retrouver le parent, nous avons préféré utiliser une classe interne afin que le code soit moins complexe à comprendre.

Cet algorithme est plus simple et moins efficace que A*, mais demande beaucoup de calculs. Il permet de trouver un chemin pas forcément optimal ce qui permet de laisser plus de chance à l'adversaire.
Cependant, les chemins parfois complexes peuvent donner du fil à retordre au Hunter.

1.3 Génération du labyrinthe

1.3.1 Algorithme de Prim

Pour la génération du labyrinthe, nous avons utilisé l'algorithme de Prim.
Il réalise un labyrinthe parfait, c'est-à-dire qu'il n'y a qu'un seul chemin entre deux points du labyrinthe. Pour cela, il prend un labyrinthe rempli de murs et créé un tableau de boolean de même taille afin de savoir quelles cellules sont déjà explorées, et prend comme cellule initiale une cellule aléatoire.
Pour créer le labyrinthe, on calcule la "frontière" de la cellule actuelle, autrement dit, les cellules adjacentes à deux de distances qui ne sont pas encore explorées. On choisit ensuite une cellule de la "frontière" aléatoirement, on la marque comme explorée et on la relie à la cellule actuelle. En supprimant les murs entre les deux cellules (et sur les cellules). On répète ensuite l'opération avec une nouvelle cellule de la frontière jusqu'à ce que la frontière soit vide, et donc qu'on ait exploré toutes les cellules du labyrinthe.
Pour rendre le labyrinthe plus jouable, après la génération d'un labyrinthe aléatoire, on supprime des murs aléatoires en vérifiant qu'ils ne génèrent pas de chemin isolé.
De cette manière, les labyrinthes générés sont toujours jouables, et les algorithmes de recherche de chemin marchent dans tous les cas.

Cet algorithme est implémenté dans la classe **MazeFactory.java**, avec cette manière de faire, nous pouvons créer un labyrinthe de n'importe quel taille, qu'il soit carré ou rectangulaire, même si nous avons limité la génération à un labyrinthe de taille 16*16 car au-delà, le labyrinthe est trop grand pour être affiché sur l'écran.