Rapport sur la partie algorithmique du projet 'Chasse Au Monstre'

Anas Ouhdda, Karim Aoulad-Tayab, Yliess Elatifi, Atilla Tas, Selim Hamza

Contents

Construction du Labyrinthe **2**

La stratégie IA du Monstre **3**

La stratégie IA du Chasseur **3**

Recul sur les algorithmes utilisés **4**

Ce rapport est effectué dans le cadre de la ressource 'Dev efficace' de S3, ce rapport présente les élements algorithmiques utilisés pour notre projet 'Chasse au Monstre'.

Construction du Labyrinthe

La labyrinthe est généré de manière aléatoire à l'aide de cette la classe MazeGenerator, ce labyrinthe doit être validé avec la classe MazeValidator (en on reparlera plus tard dans le dernier chapitre...).

Au sein de la classe MazeGenerator, nous avons implementé l'algorithme de parcours en profondeur (dfs) pour générer le labyrinthe, ce qui est important à noter pour cet algorithme c'est tout d'abord le fait de générer l'entrée et la sortie, car ils seront toujours à des emplacements aléatoires sur chaque partie et c'est un préliminaire avant de démarrer l'algo.

Premièrement nous appellons `this.generateEntanceAndExit()`, ici nous définissons aléatoirement si l'entrée/sortie se trouve dans le premier quart du labyrinthe ou s'il se trouve en partant du 1er quart jusqu'au dernier, cela permet de placer l'entrée et la sortie n'importe où comme demandé dans le sujet.

```
# ...
if (random.nextInt(2) == 1) {
    exitx = random.nextInt((height - (height / 4) - 1), height - 1);
    entrancex = random.nextInt(0, height / 4);
} else {
    entrancex = random.nextInt(height - (height / 4) - 1, height - 1);
    exitx = random.nextInt(0, height / 4);
}
# ...
```

Puis ensuite vient `this.dfs()`, c'est cette méthode qui éxecutera l'algorithme dfs (depth-first search ou 'parcours en profondeur').

> Vous pouvez consulter le code de l'algo sur ce lien.

Ici dfs va dépiler l'entrée pour débuter son parcours puis empiler chaque voisin qu'il trouvera pour faire son exploration, s'il ne trouve rien il revient en arrière grâce aux prédécesseurs stockés dans la **pile**.

Choix de la structure de donnée ?

Ici, nous avons opté pour la pile (stack), c'est l'implémentation naturelle pour un parcours en profondeur, cependant on aurait pu l'implémenter par récursion, cela reprend le même principe que la pile sauf qu'ici la pile sera le callstack de la JVM. Malheuresement c'est plus coûteux en mémoire de manière générale (pas tout le temps) et une implémentation itérative sera toujours plus performante (surtout pour les langages impératifs/orienté-objet).

Donc si on n'a pas atteint la sortie (générée précédemment), on continue la boucle jusqu'à ce qu'on trouve la sortie ou que la stack soit vide (càd qu'on a exploré toutes les cases du labyrinthe).

Efficacité de l'algorithme ?

Avant de choisir dfs, nous avions une variante très lente (en prenant en compte tout l'over-head de JavaFX) que nous avons dû remplacer, pour cela nous avons testé ce site fait par un ancien de l'IUT qui simule quelques algorithmes de génération et de résolution de chemin en Javascript, et nous avons remarqué que dfs était plutôt performant pour la génération après plusieurs tests, nous avons donc utilisé cet algorithme et l'éxecution dans notre jeu était bel et bien plus rapide par rapport à avant.

La stratégie IA du Monstre

La stratégie IA du Monstre fait parti du modèle du jeu, et utilise l'algorithme A*, son implémentation se trouve dans la classe AStar dans la méthode `execute()`.

A* introduit 2 nouveautés par rapport à Dijkstra, car en effet cet algorithme est une extension de ce fameux algorithme de recherche de chemin: - **L'introduction d'une heuristique**

> Cet algorithme introduit une estimation du point actuel vers la sortie, il existe plusieurs heuristiques qui sont plus ou moins efficaces, selon la contexte certains seront plus efficaces, par exemple: la **distance de Manhattan** est utilisé pour les déplacements horizontaux et verticaux principalement donc idéal pour un labyrinthe (donc idéal pour notre cas). Cette estimation est très utile car elle permet de mieux orienter son choix de chemin même sur un point assez eloigné de la sortie, le but étant de ne pas sur-estimer, ni sous-estimer la valeur de cette heuristique pour avoir un résultat optimal.

- **Choix de la structure de données ?**

> Contrairement à Dijktra, ici on utilise une file qui se trie suivant un critère particulier, celui de l'heuristique la plus faible. La priorité sera toujours à la cellule ayant la plus faible heuristique car cela veut dire qu'elle est fondamentalement proche de la sortie. On implémente également une Map pour stocker la distance "réelle" de chaque noeud (càd la distance d'une cellule par rapport à l'entrée).

Mais comment fonctionne cette le parcours ? A* commence par un point d'entrée (l'entrée du labyrinthe qui a été générée précédemment) et on assigne la distance réelle pour chaque cellule en lui ajoutant l'heuristique pour une meilleur estimation. Et l'exploration sera donc pilotée par la file de priorité qui choisera toujours les meilleurs cellules pour obtenir le chemin le plus court.

Efficacité de l'algorithme ?

Parmi tous les algorithmes qui existent, c'est incontestablement le plus efficace que ce soit en compléxité temporelle qui est O(|E|), cela correspond au nombre de fois (un nombre constant) qu'on parcourera les arêtes. Mais c'est également celui qui trouve rapidement le plus court chemin.

La stratégie IA du Chasseur

La stratégie IA du Chasseur fait également parti du modèle du jeu implémentée dans la classe Random-Controlled. Le chasseur adopte une stratégie pour tirer sur les cases du labyrinthe. La stratégie actuelle

consiste à explorer les cases voisines d'une case déjà visitée. Si le monstre a traversé l'une de ces cases voisines, le chasseur tire sur cette case. Sinon, le chasseur choisit une case aléatoire non encore explorée.

Choix de la structure de données ?

Sans surprise, la structure de données est une stack car l'algorithme utilisé s'apparente beaucoup à un dfs classique à la seule différence qu'ici on modélise l'impasse par une case déjà visitée ou une case auquel le chasseur à déjà tirée. L'exploration se fait plus ou moins comme une exploration classique de chemin sauf qu'ici on cherche pas la sortie mais le monstre en suivant les indications des cases que le monstre a visité (ce seront ces cellules qu'on ajoutera dans la stack sinon ça sera évidemment des cases aléatoires) pour que l'IA étoffe son choix dans la partie.

Efficacité de l'algorithme ?

Cela dépend de la taille du labyrinthe, cependant il aura la même complexité qu'un dfs classique voir même pire car cela dépend un peu de la "chance" du chasseur IA qui pourra tomber sur une case déjà visitée du monstre après avoir tiré aléatoirement, donc cela reste tout de même de l'aléatoire mais contrôlé un minimum de sorte à ce qu'il y ait un minimum de compétitivité dans les parties.

Recul sur les algorithmes utilisés

Un MazeValidator de trop ?

Concernant MazeValidator, cela pose un problème d'optimisation, en effet lorsque notre labyrinthe est générée nous avons aucune certitude que la génération ait générée un chemin unique, nous devons passer ce labyrinthe dans la classe `MazeValidator` pour vérifier que le labyrinthe possède un chemin. Or, procéder de cette manière ralentit le processus car nous avons 2 passes: - 1ère passe : Générer le labyrinthe - 2ème passe : Valider le labyrinthe

Si la validation est incorrecte, alors nous devons revenir à la première passe donc nous faisons 2 fois le même parcours ce qui est côuteux en terme de temps d'éxecution.

> À noter que l'algorithme utilisé pour MazeValidator est A*, avant cela nous utilisions dfs mais cela nous a semblé beaucoup plus judicieux de passer par A* car cela permettait de gagner en performances dans les parties.

Structure de données inutile pour A*

Nous avons vu qu'on a utilisé une Map pour stocker la distance réelle de chaque cellule, cependant stocker cela dans un attribut d'un objet Cellule serait préférable car nous utiliserions uniquement une seule structure de données (celle de la file de priorité), cela serait donc moins coûteux en mémoire. JavaFX prend déjà assez de mémoire RAM à lui tout seul car il embarque tout le runtime Java pour s'éxecuter correctement.