

Axel Saint-Maxin  
Noa Moreau  
Jules Delespierre

## Présentation des algorithmes utilisés dans la chasse aux monstres

Algorithme Aléatoire (MonsterStrategyRandom et HunterStrategyRandom) :

Cet algorithme est l'un des plus simples possibles. En effet pour les étapes pour le monster sont :

- 1) Récupérer la position du monstre
- 2) Décider via l'aléatoire si on va à droite, à gauche ou on ne bouge pas sur cet axe.
- 3) Décider via l'aléatoire si on va en haut, en bas ou on ne bouge pas sur cet axe.
- 4) Vérifier que cette position obtenue est bien dans le labyrinthe, que ce n'est pas un mur et que ce n'est pas la position actuelle
- 5) Mettre la nouvelle coordonnée en tant qu'actuelle
- 6) Envoyer la coordonner

IA est très rapide, en effet, à part de rare cas comme un cul-de-sac.



En effet dans ce cas, l'IA n'a qu'une chance sur 8 de trouver un coup jouable. Mais même dans ce cas extrême l'IA est très rapide, car n'effectue que de simples calculs

Et en stockage, il est très bon en effet, nous utilisons une variable pour stocker sa position actuelle et une matrice pour les murs du labyrinthe. Pour améliorer, nous pouvons utiliser une liste au lieu de la matrice pour passer d'une complexité de  $O(2)$  à  $O(1)$

Les étapes pour le hunter sont :

- 1) Récupérer la taille du labyrinthe
- 2) Prendre une coordonnée aléatoire dans les bornes de la taille du labyrinthe
- 3) Envoyer la coordonner
- 4)

Cette IA est très rapide (2), on peut imaginer des IA plus rapides comme par exemple avec des constantes, mais elles seront moins bonnes. Et en plus de cela, l'IA ne stocke que des données essentielles. Elle est aussi

très forte, en effet, à chaque coup, elle a  $1/(\text{longueur} * \text{largeur})$  de gagner.

Mais on peut l'améliorer sur 2 points.

- 1) Elle pourrait stocker les murs pour éviter de tirer plusieurs fois sur un mur. Grâce à cette modification, on pourrait augmenter les probabilités de gagner à chaque mur trouvé.
- 2) Garder la position du monstre la plus récente. En effet, si nous sommes au tour 5 sur un labyrinthe de  $10*10$ . Et que la case tirer nous indique que le monstre est passé sur cette case au tour 2 cela veut dire que le monstre est dans un rayon de  $(5 - 2)$  tour autour de la case. On passe donc d'une probabilité de gagner de  $1/100$  à  $1/49$  on a donc 2 fois plus de chance de gagner

Avec ces 2 modifications, l'IA est encore plus efficace.

L'algorithme de Pledge (MonsterStragyMainDroite):

C'est un algorithme assez naturel en effet, c'est celui qu'applique énormément de personne pour sortir d'un labyrinthe.

Il stocke les murs dans une matrice et il stocke la position actuelle du monstre. Il ne stocke pas la sortie, car il n'a pas besoin.

On pourrait l'améliorer en utilisant une liste au lieu d'une matrice.

Cet algorithme n'est pas le meilleur si on connaît la sortie en effet il existe le parcours en largeur, A\* ou autre qui sont beaucoup plus rapide, mais en cas de brouillard activé cet algorithme fonctionne quand même alors que les autres deviennent inutiles.

## Algorithme du labyrinthe

### Description :

Algorithme fait main "Par taux de chemin voisin".

Nécessite les classes Maze.java et Direction.java

Variable en paramètres :

int x : taille du labyrinthe en largeur

int y : taille du labyrinthe en longueur

double taux : taux de mur du labyrinthe

Les étapes :

- 1) Initialise le labyrinthe plein de murs sauf le centre qui est un chemin.
- 2) Calcule le nombre de VoisinChemin (les 4 cases adjacentes aux centres finissent à 1 et les autres à 0 pour cette première itération).
- 3) Calcule le nombre de mur nécessaire et le nombre de mur initial, initialise les variables nécessaires dans la future boucle de génération (cpt (max iteration), coord (pour empêcher de recalculer les coordonnées déjà calculer) , listPossible (direction possible)
- 4) Ouverture de la grande boucle de génération, si le MAX\_ITERATION (mis à 100) la boucle prend fin, de même si le intiWall atteint le nombre de mur demandé par le taux.
  - On calcule nbVoisinChemin
  - On prend une case qui ne fait pas partie de la liste coord
  - On l'ajoute à la liste coord
  - On ajoute à la liste des directions possibles toutes les directions possibles
  - On prend une direction random parmi celle possible
  - On tente en fonction de la direction obtenu et du nombre de voisinChemin qu'elle implique différentes probabilité (0% -> 3 et 4 murs ,10% -> 2,90% -> 1, 0 chemin impossible car point d'origine au minimum)
  - Si la probabilité réussit, on crée le chemin et on ajuste la valeur de InitWall. Sinon on retente le choix de la direction.
  - On vide la liste des directions possibles
- 5) On écrit dans un csv le labyrinthe avant génération de la position de la sortie et du monstre pour faire de l'exportation au format :  
1,1,0,0,0  
0,1,0,1,1

0,0,0,1,1

### Structures de données :

- `int[][] labyrinthe` : Cette structure de données contient le labyrinthe (chemin et mur) sous forme de 0 (chemin) et de 1 (mur). Forme la plus claire pour la compréhension une grille en 2 dimensions avec un accès de chaque données grâce à 2 boucles faciles.
- `int[][] nbVoisinChemin` : Celle-ci est une sorte de clone du labyrinthe, elle contient pour chaque case, en fonction de ses 4 voisins le nombre de chemin à côté de lui. Utile pour les calculs de probabilité de génération de labyrinthe. Même raison que pour le labyrinthe.
- `list listPossible` : Pour toute les Directions possibles par une case en fonction de si ces voisins sont des murs ou des chemins. Facile d'accès par index, calcul de taille facile et peu de données (maximum 3 énumération) donc optimisation de calcul non nécessaire.

### Efficacité :

- Ma condition de boucle "`initNbWall > nbWall`" permet de respecter au nombre près le nombre de chemin voulu ce qui réduit le nombre de calcul par rapport à ceux qui feront une boucle entière du labyrinthe au risque de trop généré de chemin
- La liste de coordonnées permet d'empêcher tout surplus de calcul si la coordonnée à déjà était testé
- La liste permet de facilement retrouver une coordonnées en usant d'une autre, une Map n'aurait pas été adaptée du fait qu'une coordonnées c'est 2 valeurs et que la clé n'aurait peut être que l'ensemble x et y ou un nouvelle identifiant créé.
- Pour la vitesse d'exécution le labyrinthe peut générer du 30\*30 quasiment instantanément au delà le temps augmente exponentiellement dû au fait de vérifier tous le labyrinthe à chaque fois mais c'était nécessaire pour avoir une précision optimal
- En point fort, cet algorithme est fait main, les formes de labyrinthe sont très bonnes, seul le hasard décide du nombre de chemin, il n'y a aucune déformation qu'importe la taille et l'exportation en csv ne pose aucun problème.
- En point faible, l'algorithme est lent à grande échelle, l'échelle nécessaire pour voir ces ralentissement dépassant 30\*30 et bien supérieur à ce qui est nécessaire pour jouer une partie parfaite, au-delà le jeu serait moins intéressant, donc ce défaut est

minimisé par l'utilisation faite de l'algorithme. Un second point faible, l'algorithme manque un peu de modularité, il était prévue de permettre de changer les constantes ONE\_WAY\_CHANCE, TWO\_WAY\_CHANCE et THREE\_WAY\_CHANCE mais la forme du labyrinthe était moins intéressante que les paramètres actuels qui sont des paramètres optimaux.