1. **Algorithme de génération de labyrinthe**

Description :

Pour la génération du Labyrinthe nous avons décidé de ne pas s'inspirer ou d'utiliser un algorithme déjà existant, nous avons donc testé beaucoup de choses différentes mais voici ce que nous avons retenue afin de répondre à nos critère qui étaient :

- Génération de labyrinthe imparfait (plusieurs chemin possibles vers la sortie.)
- Gestion du pourcentage de mur dans le labyrinthe selon le choix de l'utilisateur.
- Positionnement du monstre et de la sortie aléatoire pour éviter qu'elles soient prévisibles après plusieurs parties et espacé d'un certain nombre de case au minimum.
- S'étendre un maximum sur la surface totale pour donner un bon labyrinthe visuellement (éviter les regroupement de mur d'un côté et les chemins de l'autre).

Notre algorithme fonctionne en creusant quatres sections, ces sections étant constitués de morceaux de chemin.
Tout d'abord on calcule le nombre de case à creuser (C) à l'aide du pourcentage de mur demandé par l'utilisateur.
On divise ce nombre par quatre puis on stocke le reste (R) que l'on va répartir par la suite.
Avec ces valeurs on détermine la longueur que devront faire chacune de nos quatre sections :

$Soit$:
C : $le\ nombre\ de\ cellule$ à $creuser$
R : $le\ reste$
$Si\ R\ >\ 0$: $C\ /\ 4\ +\ 1$; $R\ -\ 1$
$Sinon$: $C\ /4\ +\ 0$

Ensuite, on se positionne au centre du labyrinthe en prenant comme coordonnées [longueur/2 ; largeur/2] ces valeurs sont arrondies à l'inférieur.
On va creuser nos quatres sections qui partent tous du point de départ et vont chacune vers un point cardinal différent.

Pour chacune des sections notre programme fait l'action suivante :

$Soit$:
L : $Longueur\ ou\ largeur\ du\ Labyrinthe\ (Selon\ la\ direction)$
T : $Taille\ du\ morceau\ choisi\ aléatoirement\ entre\ ses\ bornes$
$Si\ Premier\ morceau$: $L\ /\ 5\ +\ 1\ <=\ T\ <=\ L\ /\ 2$
$Sinon$: $L\ /\ 5\ <=\ T\ <=\ (L\ /\ 5)\ *\ 2$

Une fois T obtenu, l'algorithme va vérifier qu'il peut creuser (Si celle-ci n'est pas déjà creusé et qu'elle se trouve bien dans les limites du Labyrinthe) chacune des cases du morceau dans la direction affectée et soustraire le nombre de case creusé au nombre de case dont à besoin la section.

Pour finir, il choisit une nouvelle direction vers laquelle se diriger (Celle-ci ne doit pas indiquer une bordure et ne peut pas être la direction opposé de celle précédemment sélectionnée)

Cette action est répétée tant que le nombre de cases dont la section à besoin est supérieur à zéro. (L'excédent de case creusé est soustrait au nombre de cases dont à besoin la prochaine section).

Structure de donnée :
Dans la classe MazeGenerator nous utilisons plusieurs structure de donnée tel que :

- Un tableau de booléen pour représenter les murs, dont toute les cases sont initialisé à true au début de la génération
- Une TreeMap dans laquelle sont stockés les coordonnées du Monstre (Sous la forme Tour du jeu : Coordonnée) dans lequel on stockera la première coordonnées du Monstre suite à la génération du Labyrinthe
- Une TreeMap similaire pour le Chasseur qui restera vide
- Une variable pour la sortie qui sera initialisé suite à la génération du Labyrinthe

Efficacité :

Pour parler de l'efficacité de l'algorithme de génération de labyrinthe.
Le temps de génération de labyrinthe est très satisfaisant avec des labyrinthes de 50000 cases fois 50000 généré en 2 secondes à peine.
Un problème existant est que même si le chemin ne revient pas vers son dernier choix de direction, dans l'absolue il est possible que le programme boucle à l'infini en tournant en rond (nord>est>sud>ouest>nord etc… cependant il faudrait aussi que la taille choisie des chemins soient identique donc c'est très peu probable que cela arrive).
Mais cela représente quand même une certaine perte de performance étant donné que le chemin passe par des endroits déjà creusés même si cela peut mener à la création d'un nouveau chemin dans la finalité.

2. Algorithme de jeux

Le package "ai" contient le package "algorithm" ainsi que les classes suivantes :

- **CursiveCoordinate** : Classe permettant à une cellule d'avoir un parent, ce qui est utilisé par la plupart des algorithmes pour trouver un chemin de façon efficace.
- **MazeSolver** : Classe principale qui utilise différents algorithmes pour trouver le chemin le plus court vers la sortie (IA Monstre).
- **MonsterFinder** : Classe représentant l'IA du Chasseur.

Le package "algorithm" contient les classes qui implémentent les algorithmes pour résoudre des labyrinthes. Les trois classes sont **AStarAlgorithm**, **BidirectionalAlgorithm** et **ThetaStarAlgorithm**, qui implémentent toutes l'interface **IMazeSolverAlgorithm**.

- **IMazeSolverAlgorithm** : Interface qui permet à chaque algorithme créé de respecter une architecture qui peut ensuite être facilement utilisée par la classe MazeSolver.

  Structure de donnée pour toutes les classes suivante : Un tableau de booléen a deux dimensions utilisées pour représenter le labyrinthe. Chaque cellule du tableau représente une cellule du labyrinthe, et la valeur true ou false indique si la cellule est un mur ou non.

- **AStarAlgorithm** (Algorithme A*) : C'est un algorithme de recherche de chemin utilisé couramment dans la résolution de problèmes de recherche de chemin dans des graphes. Il est particulièrement efficace pour trouver le chemin le plus court entre deux points dans un graphe pondéré, ce qui en fait un très bon choix pour résoudre notre problème de navigation dans un labyrinthe.
  - Structure de donnée :
    - 2 HashSet openSet et closeSet permettant de stocker les nœuds a visiter et les nœuds déjà visités.
    - 2 HashMap : gScore stocke le coût réel pour atteindre chaque nœud à partir du point de départ, et fScore stocke l'estimation du coût total pour atteindre le point d'arrivée à partir de chaque nœud (le coût réel plus une estimation heuristique du coût restant).
  - Efficacité :
    - Le même labyrinthe pour tous les algorithmes : 5ms
    - Labyrinthe de 50000x50000 : 500 - 2000ms

- **BidirectionalAlgorithm** (Algorithme Bidirectionnel) : Cet algorithme utilise une approche bidirectionnelle pour résoudre le labyrinthe. Il commence à la fois par le début et la fin du labyrinthe et se rencontre au milieu. C'est une approche efficace car elle peut potentiellement réduire de moitié le temps de recherche par rapport à une recherche unidirectionnelle. Cependant, il nécessite de maintenir deux ensembles de nœuds visités et deux files d'attente, ce qui peut augmenter l'utilisation de la mémoire.
  - Structure de donnée :
    - 2 Queue : forwardQueue et backwardQueue, ces deux files d'attente sont utilisées pour stocker les nœuds à visiter lors de la recherche vers l'avant et vers l'arrière respectivement.
    - 2 HashSet : forwardVisited et backwardVisited, ces deux ensembles sont utilisés pour stocker les nœuds déjà visités lors de la recherche vers l'avant et vers l'arrière respectivement.
  - Efficacité :
    - Le même labyrinthe pour tous les algorithmes : 1 - 2ms
    - Labyrinthe de 50000x50000 : Impossible
    - Labyrinthe de 100x100 : 213ms

- **ThetaStarAlgorithm** (Algorithme Theta*) : C'est une variante de l'algorithme A* qui permet des chemins plus courts en autorisant les mouvements en diagonale. Il utilise une file de priorité pour sélectionner le prochain nœud à visiter en fonction d'une fonction d'évaluation (la somme du coût pour atteindre le nœud et une estimation du coût pour atteindre l'objectif). C'est un algorithme efficace pour trouver le chemin le plus court dans un labyrinthe, mais il peut être plus lent que d'autres algorithmes si le labyrinthe est grand et complexe, car il doit évaluer de nombreux chemins.
    - Structure de donnée :
        - PriorityQueue : openset, une file de priorité pour stocker les nœuds à visiter, où les nœuds avec le plus petit fScore (le coût estimé pour atteindre l'objectif) sont visités en premier.
        - HashSet : closeSet permettant de stocker les nœuds a visiter et les nœuds déjà visités.
        - HashMap : gScore stocke le coût réel pour atteindre chaque nœud à partir du point de départ.
    - Efficacité :
        - Le même labyrinthe pour tous les algorithmes : 5ms
        - Labyrinthe de 50000x50000 : 60 - 80ms

Ces trois algorithmes sont bien adaptés pour résoudre des labyrinthes. Le choix de l'algorithme à utiliser dépend des spécificités du labyrinthe et des contraintes de performance. Par exemple, si le labyrinthe est de taille moyenne et que la performance est une préoccupation, **BidirectionalAlgorithm** est sûrement un meilleur choix. Si le labyrinthe permet des mouvements en diagonale et que l'on souhaite un calcul très rapide, **ThetaStarAlgorithm** serait un meilleur choix, dans le cas où le mouvement en diagonale n'est pas autorisé, on peut aussi utiliser **AStarAlgorithm** qui peut aussi être utile dans le cas ou **BidirectionalAlgorithm** n'est pas capable de trouver le chemin du au faite que le calcul est très long.

Pour l'IA du chasseur, je voulais implémenter un algorithme qui, lorsque l'information de la cellule sur laquelle il venait de tirer comportait le point de passage du monstre. Cela créerait un cercle estimant le périmètre dans lequel pourrait se situer le monstre. Ce cercle aurait pour rayon la différence entre le tour actuel et le tour auquel le monstre est passé. Les cellules auraient ensuite été sélectionnées dans le sens horaire en partant de l'Est. Malheureusement, je n'ai pas réussi à l'implémenter à temps pour le rendu.