Rapport labyrinthe

Explication des structures présentes :

Cell : Elle représente une case du labyrinthe. Elle contient un tableau des 4 directions possibles (adjacent_cell[4]). Un booléen is_visited qui permet de connaitre les cases sur lesquelles ont est passé lors de la génération et de la résolution, from un pointeur qui permet d'indiquer de quelle case on vient lors de la résolution et un booléen is_in_path qui indique si la case fait partie du chemin pendant la résolution.

Grid : Elle représente le labyrinthe. Elle contient un tableau à 2 dimensions « cells » qui contient des éléments de struct cell , et les dimensions du labyrinthe (ligne et colonne).

Position : Elle représente une case du labyrinthe. Elle stock les coordonnées x et y.

Tab : C'est un tableau dynamique que l'on peu considérer comme une pile, il est utile pour la génération du labyrinthe. Il stock les structures positions « position » .

Queue : Elle est utilisée pour la résolution du labyrinthe, c'est là où « attendent » les cases d'être visitées.

enum Direction : défini les 4 directions possibles.

Génération :

On initialise une pile Tab pour stocker les positions, puis on sélectionne une case aléatoire pour démarrer la génération du labyrinthe, que l'on met dans la structure Tab et marquée comme visitée. Tant que la structure n'est pas vide, on regarde si la case en haut de la pile a des voisins sur lesquels on n'est pas passé. S'il n'y a pas de case disponible, on dépile jusqu'à ce que la case en haut de la pile ait un voisin disponible. Si la case a un voisin disponible, on choisit une direction aléatoire, si celle-ci est hors des limites ou déjà visitée, on relance jusqu'à avoir une case non visitée. Puis on l'empile et on la marque comme visitée et dit qu'elles sont adjacentes avec « adjacent_cells ». Quand toutes les cases sont visitées, la pile se vide entièrement car elle cherche une case sur laquelle on n'est pas passé. On réinitialise is_visited à False pour la résolution.

Résolution :

Pour la résolution, on utilise une queue que l'on initialise et où on y met la position de départ que l'on marque comme visitée. On part de la case en bas à gauche du labyrinthe pour arriver à celle en haut à droite. Tant que la queue n'est pas vide, on analyse chaque case, on prend la première case de la queue, on la marque comme visitée et on vérifie les cases adjacentes, chaque case adjacente disponible et non visitée est mise dans la queue et marquée comme visitée.

Sur chaque case rentrée dans la queue, on dit de quelle case elle vient grâce à « from ». Une fois terminé, on met la case d'arrivée dans le chemin (is_inpath = True). Pour trouver le chemin, on part de la case d'arrivée et tant que l'on n'est pas arrivé à la case de départ, on recule d'une case et on la met dans le chemin (is_in_path=True).