BARSONI Dorian

Projet Info4A : Partie 4B

Dans la partie 4B du projet il était demandé de réaliser l'implémentation des différentes fonctions de la classe Labyrinthe ainsi que leurs tests, puis la réécriture de distMin(). Il était également demandé d'expliquer la contribution personnelle réalisée. Nous commencerons par voir la solution proposée pour la méthode distMin(), puis nous verrons la contribution personnelle que j'ai apporté au projet.

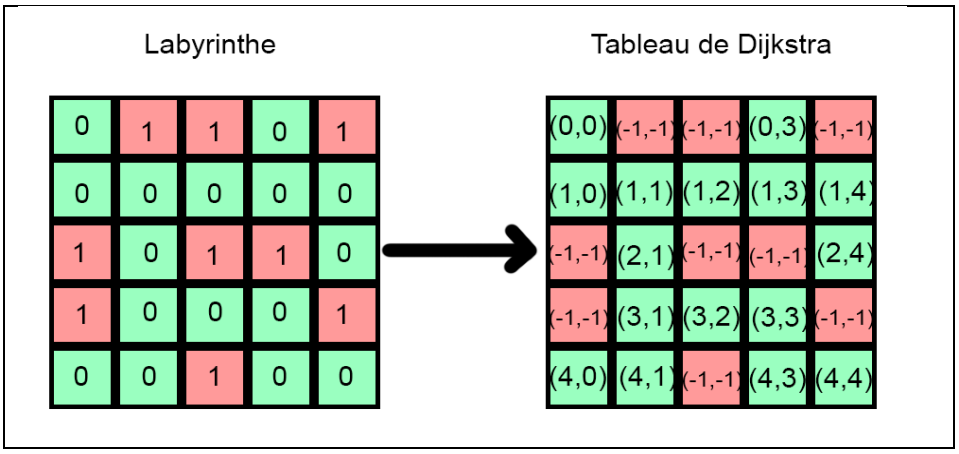
1) La méthode distMin()

Pour rappel la méthode distMin() a pour but de renvoyer la distance minimale qui sépare un case de la grille id1 d'une autre id2. La solution qui apparait logique est donc d'utiliser un algorithme de Dijkstra adapté à notre problème.

Pour ce faire j'ai donc créé 2 nouvelles classes :

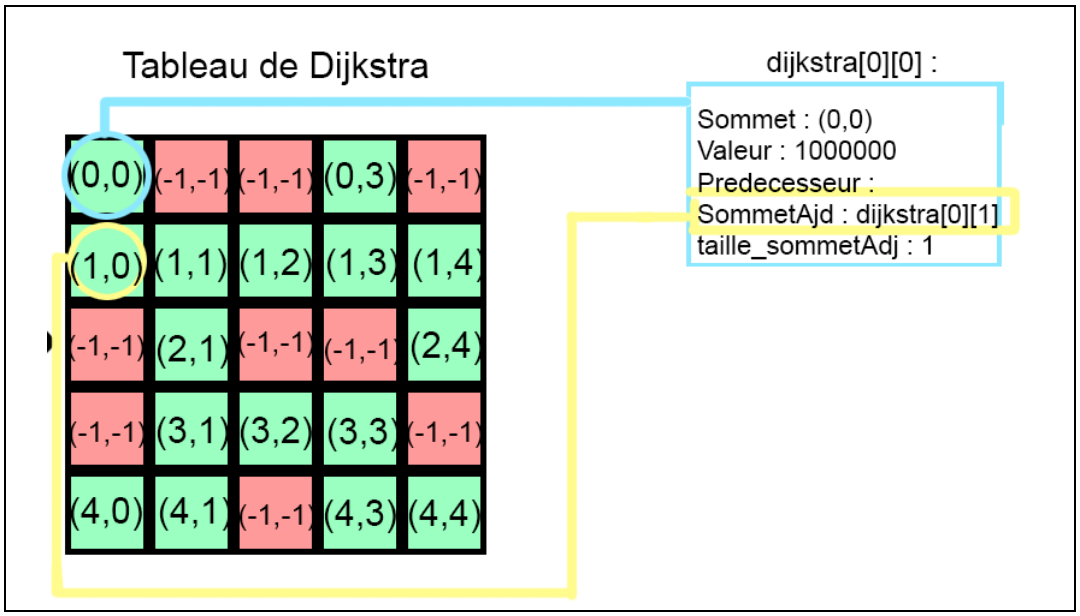
- La classe Point, représentant un point. Possédant un entier x et un entier y, ainsi que les méthodes usuelles getX(), getY(), setX(int x), setY(int y), une méthode aff() pour afficher le point, un constructeur par défaut créant un point (0,0), un constructeur créant un point (x,y) et une méthode booléenne sontEgaux(Point p) qui retourne true si 2 points sont égaux et false s'ils ne le sont pas.
- La classe Dijkstra qui possède un Point nommé sommet, un entier représentant sa valeur, deux pointeurs sur des instances de Dijkstra nommées predecesseur et sommetAdj (pour adjacent), ainsi qu'un entier taille_sommetAdj. La classe possède un constructeur acceptant un entier pour fixer la taille de sommetAdj, un destructeur pour détruire sommetAdj et predecesseur, une méthode setPredecesseur(Dijkstra* predecesseur) pour changer le predecesseur, et une méthode addSommetAdj(Dijkstra d) pour ajouter une instance de Dijkstra dans sommetAdj.
- La classe PileDijkstra qui est la même classe que Pile mais pour stocker des éléments Dijkstra (push(), pop()…)

L'idée afin d'obtenir la distance minimale entre deux points est la suivante : On crée un tableau de Dijkstra de même taille que le labyrinthe et on le parcours à l'aide de 2 boucles for sur i et j. Si la valeur (i,j) du labyrinthe vaut 0 alors dans le tableau de Dijkstra on crée une nouvelle instance de Dijkstra en lui passant comme paramètre le point de coordonnée (i, j). Si la case (i,j) on fait pareil sauf qu'on passe en paramètre le pont (-1,-1) .



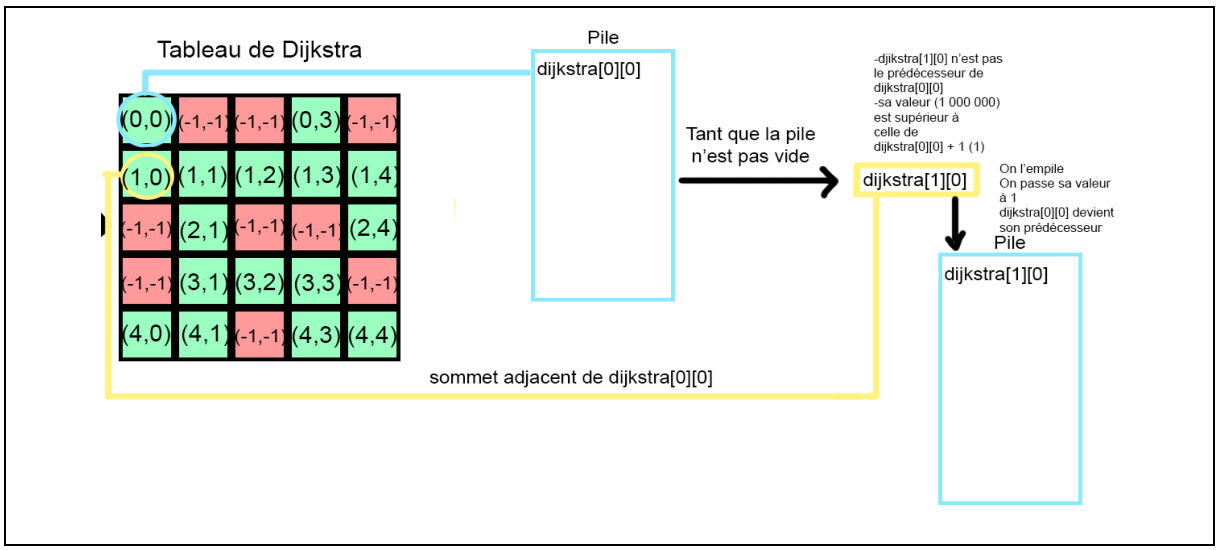
Projet Info4A : Partie 4B

Ensuite on parcourt le labyrinthe. Pour chaque case ayant la valeur 0, on regarde les 4 cases autour d'elles. Si certaines de ces cases valent 0 on les ajoute dans le tableau sommetAdj de la cellule (i,j) du tableau de Dijkstra.



On crée une instance de Dijkstra début et fin. Début prend la valeur du tableau de Dijkstra correspondant à l'id1 et fin celle à l'id2. On passe la valeur de de début à 0, on fixe son prédécesseur à lui-même et on le et dans la pile. Tant que la pile n'est pas vide :

- On place l'instance de Dijkstra dépilé dans sommetCourant
- On parcourt tous les sommets adjacents de sommetCourant
- Si le sommet adjacent n'est ni le prédécesseur de sommet courant ni le sommet fin :
- Si la valeur du sommet adjacent est supérieure à la valeur du sommet courant + 1, on empile le sommet adjacent, on définit que sommetCourant est son prédécesseur et on définit sa valeur à celle de sommetCourant + 1



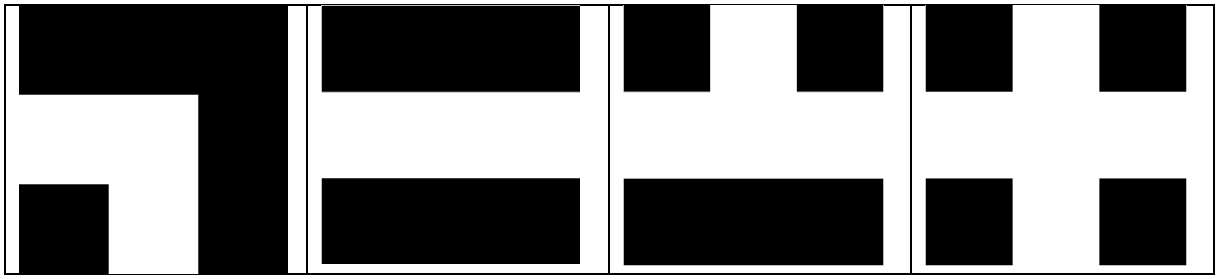
Un fois cela fait on peut remonter le chemin. On prend le sommet fin, et on remonte de prédécesseur en prédécesseur jusqu'à arriver au sommet début. On décompte le nombre d'itérations qu'on retourne.

Ceci était la façon dont j'imaginais le fonctionnement de la méthode distMin(). Cependant je me suis heurté à un problème de fragmentation que je crois avoir localisé au niveau du destructeur. Je n'ai malheureusement pas réussi à corriger l'anomalie, de ce fait ma méthode distMin() est inutilisable.

2) Contribution personnelle

Ma contribution personnelle apportée au projet est celle qui a été présenté lorsque j'ai rendu la partie 2. Il s'agit d'une nouvelle production des labyrinthes.

L'idée qui m'est venue était de créer plusieurs paternes récurrent que l'on peut retrouver dans un labyrinthe dit « intéressant ». Les paternes utilisés sont les suivants (les carrés noirs sont des murs, les blancs sont du vide) :



Une fois ces paternes créés sous forme de tableaux de taille 9 je les ai tous regroupés dans un tableau de pointeur. J'ai également créé une fonction rotate() permettant d'effectuer une rotation de 90° dans le sens horaire des paternes. La fonction selec_pattern() permet, lorsqu'on lui donne un tableau de taille 9, de sélectionner un de ses 4 paternes de manière aléatoire, puis de lui effectuer un nombre aléatoire de rotation (compris entre 0 et 3).

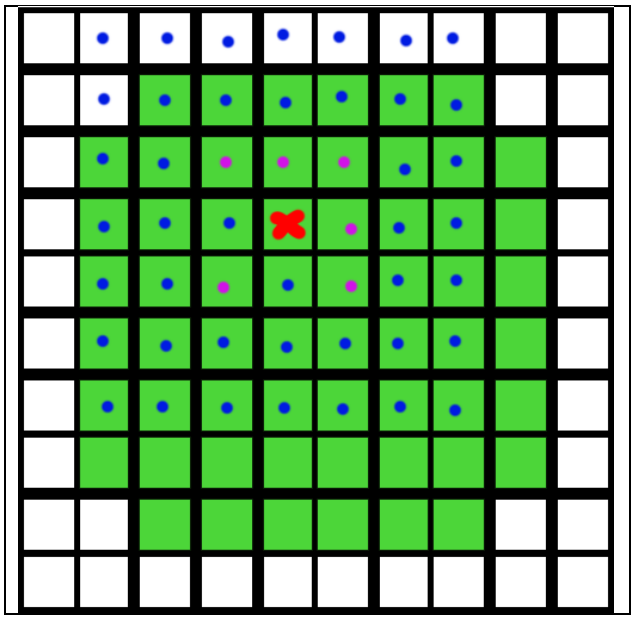
Quand bien même la rotation n'a pas d'effet sur le paterne 4, et 2 rotations correspondent à aucune rotation pour le paterne 2, je n'ai pas fait de cas particulier pour simplifier le code.

Maintenant qu'on a ces paternes on souhaite les appliquer sur la grille mais pas n'importe comment. Il ne faut pas que les paternes empiètent les uns sur les autres (c'est-à-dire qu'en positionner un en affecte un autre, ou encore que cela rende le labyrinthe non connexe). Je prends l'exemple d'un labyrinthe 10x10 pour expliquer le raisonnement :

- En vert nous avons les cases du labyrinthe ou, initialement, les paternes peuvent êtres positionnés
- La croix rouge représente le point d'application du paterne
- Les points magentas représentent les murs

Projet Info4A : Partie 4B

- Les points bleus représentent les cases interdites pour que 2 paternes n'empiètent pas l'un sur l'autre



Les cases vertes sont ainsi sélectionnées pour éviter qu'un paterne soit coupé et ne sorte du tableau. De plus les coins du carré vert sont exclus car dans certain cas la fonction selec_patern() peut retourner le paterne 1 avec une rotation qui rendrait le labyrinthe non connexe.

On parcourt donc les cases vertes une à une en ayant à chaque fois une chance sur 2 d'y placer un paterne. Une fois qu'un paterne doit être placé on le copie sur la grille. Le point sélectionné représente le centre du paterne. Puis on réalise un carré 7x7 autour du paterne dans lequel on remplace tous les 0 par des 3 (également sur la croix rouge). Cela permet d'espacer les paternes de sorte qu'il y ait toujours une ligne de vide entre eux.

À chaque fois qu'on place un paterne on compte le nombre de mûrs présents sur la Grille. Si jamais le nombre de cases vides est inférieure ou égale à k (nombre de cases vides souhaité dans le labyrinthe) on arrête la construction du labyrinthe.

Enfin, une fois que les paternes sont placés, pour terminer la construction du labyrinthe on utilise une boucle while. Tant que le nombre de cases vides est différents de k et tant que le critère de sécurité est différent de 5 (choisit arbitrairement), on parcourt la Grille de la seconde case à l'avant dernière. Si la valeur de la case vaut 0 on y place un mûr et on vérifie si le labyrinthe est connexe. S'il ne l'est pas on annule la modification et on passe à la case suivante. S'il l'est alors on a une chance sur 2 d'effectivement remettre la case à 0. Ensuite on décompte le nombre de mûrs pour en déduire le nombre de cases vides. S'il est inférieur ou égale à k alors on sort de la boucle. On termine le while en incrémentant le critère de sécurité de 1.

(Je tiens à préciser ici que dans la zones des tests automatiques on peut lire « | !| Le labyrinthe ne contient aucune case blanche | !| ». Ceci est dû au fait que lorsque l'on test la fonction connexe on vérifie également le cas où le labyrinthe est uniquement composé de mûrs. Cela renvoie, comme demandé, un message d'erreur)