Parcours dans un labyrinthe

À partir d'une grille rectangulaire $p \times q$ on obtient un labyrinthe en ajoutant un certain nombre de murs pour séparer deux cases voisines. On convient en outre que ce labyrinthe est entouré de murs sur toute sa périphérie, de sorte qu'il n'est pas possible de sortir de la grille. La case située en haut à gauche, de coordonnées $(0, q-1)$, est la case de départ, celle située en bas à droite, de coordonnées $(p-1, 0)$, celle d'arrivée. Nous supposerons que tous ces labyrinthes ont une solution, à savoir un chemin reliant le départ à l'arrivée.

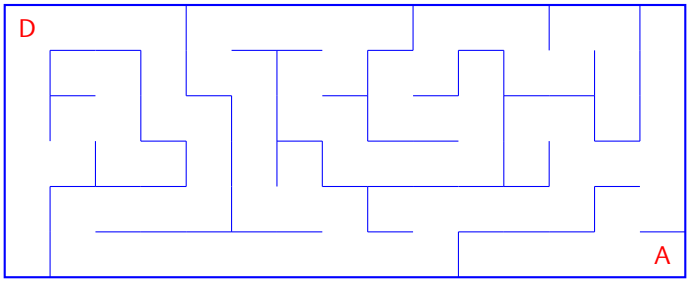

FIGURE 1 – Un exemple de labyrinthe tracé sur une grille 15×6.

Dans un premier temps, nous allons nous intéresser à la façon de trouver une solution ; dans un second temps nous verrons comment générer un certain type de labyrinthe : les labyrinthes *parfaits*, dans lesquels deux cases quelconques peuvent toujours être reliées par un unique chemin.

1. Parcours de labyrinthe

On suppose définie une classe *Case* dont les instances possèdent quatre attributs à valeurs booléennes : `N`, `W`, `S`, `E` qui indiquent dans quelles directions les déplacements sont possibles.
Par exemple, dans le labyrinthe de la figure 1, la case de départ est définie par :

```
N = False,  W = False,  S = True,  E = True
```

et la case d'arrivée par :

```
N = False,  W = True,  S = False,  E = False.
```

Un labyrinthe est alors défini par l'intermédiaire d'une classe *Labyrinthe*. Les instances de cette classe dépendent de deux paramètres : les entiers p et q, et possèdent les attributs suivants :

- `self.p` $= p$;
- `self.q` $= q$;
- `self.tab` est un tableau $p \times q$ dont les éléments sont des instances de *Case*. `tab[`i`][`j`]` désigne la case d'indice (i, j) du labyrinthe.

Par exemple, si `laby` désigne le labyrinthe présenté figure 1, on aura :

```
laby.tab[0, 0].E = True  et  laby.tab[5, 14].N = False.
```

Recherche d'une solution
La démarche que nous allons suivre pour sortir du labyrinthe s'apparente à celle de Thésée dans l'antre du Minotaure : utiliser un fil d'Ariane pour garder trace de notre chemin, et un morceau de craie pour marquer les chemins déjà explorés. Le principe de l'algorithme est le suivant : lorsqu'on arrive sur une case, celle-ci est marquée puis on se rend sur l'une de ses voisines non encore marquée. Lorsqu'il n'en existe pas, on revient sur nos pas en rembobinant le fil d'Ariane jusqu'à la dernière intersection possédant une branche non encore explorée. Cette démarche porte le nom de *parcours en profondeur* car chaque route est explorée dans son entier avant d'en explorer une nouvelle.

Question 1. De manière à pouvoir marquer une case déjà explorée, on ajoute un attribut `etat` à la définition de la classe *Case*. Initialement, toutes les cases du labyrinthe auront cet attribut égal à `True`.
Quelle structure de donnée vous parait-il adaptée pour représenter le fil d'Ariane ?
Rédiger une fonction nommée `explorer` qui prend en paramètre un labyrinthe vierge et retourne la liste des coordonnées des cases à suivre pour relier le départ à l'arrivée.

Question 2. Montrer sur un exemple simple que cette démarche ne garantit pas de donner la solution la plus courte. Proposer une nouvelle démarche garantissant le caractère minimal de la solution retournée (on ne demande pas d'écrire la fonction correspondante).

2. Génération de labyrinthes

Nous allons maintenant nous intéresser à la génération « aléatoire » de labyrinthes. Il n'existe pas de solution canonique à ce problème, aussi allons nous nous restreindre à la création de labyrinthes dits *parfaits*, c'est à dire lorsque deux cases quelconques peuvent toujours être reliées par un *unique* chemin.

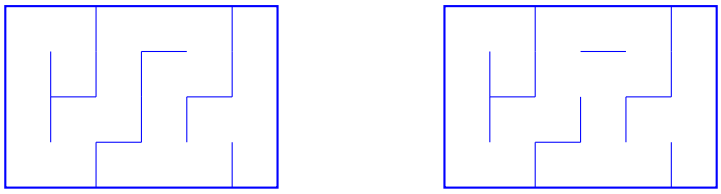

FIGURE 2 – À gauche, un labyrinthe parfait ; à droite un labyrinthe imparfait.

La méthode que nous choisissons d'implémenter consiste à partir d'un labyrinthe donc toutes les cases sont isolées par des murs. Une case est choisie arbitrairement et marquée comme étant « visitée ». On détermine ensuite quelles sont les cases voisines et non visitées, on en choisit une au hasard que l'on marque comme ayant été « visitée », et on ouvre le mur les séparant. On recommence ensuite avec la nouvelle case. Lorsqu'il n'y a plus de case voisine non visitée, on revient à la case précédente.
Lorsqu'on est revenu à la case de départ et qu'il n'y a plus de possibilité, le labyrinthe est terminé.

Question 3. Pour pouvoir aisément distinguer les cases visitées des autres, nous allons de nouveau utiliser l'attribut `etat` de la classe *Case* mais cette fois, la valeur initiale de chacune des cases sera égal à `False` (de sorte qu'une fois le labyrinthe parfait créé toutes les valeurs seront passées à `True`, ce qui le rendra prêt pour son exploration).

1. Définir les classes *Case* et *Labyrinthe* de sorte qu'à la création d'un labyrinthe, toutes les cases soient isolées par des murs.
2. En déduire une fonction `creation` qui prend en arguments deux entiers p et q et qui retourne un labyrinthe parfait de taille $p \times q$.
3. Combien de murs internes y-a-t'il dans un labyrinthe parfait ?

Question 4. Si vous voulez prolonger ce travail, vous pouvez ajouter à la définition de la classe *Labyrinthe* une méthode pour tracer le labyrinthe, ainsi que pour tracer sa solution.

Et si vraiment vous ne trouvez plus à vous occuper . . .