TP C#10 : Amazing Maze

1 Consignes de rendu

À la fin de ce TP, vous devrez rendre une archive respectant l'architecture suivante :

```
– tpcs10-login_x.zip
    |– rendu-tpcs10-login_x/
        |– AUTHORS
        |– README
        |– Maze/
            |– Maze.sln
            |– Maze/
                |– Tout sauf bin/ et obj/
```

Bien entendu, vous devez remplacer *login_x* par votre propre login. N'oubliez pas de vérifier les points suivants avant de rendre :

- Le fichier AUTHORS doit être au format habituel : ** login_x$* où le caractère '$' représente un retour à la ligne.
- Le fichier README doit contenir les difficultés que vous avez rencontrées sur ce TP, et indiquera les bonus que vous avez réalisés.
- Pas de dossiers bin ou obj dans le projet.
- **Le code doit compiler !**

2 Introduction

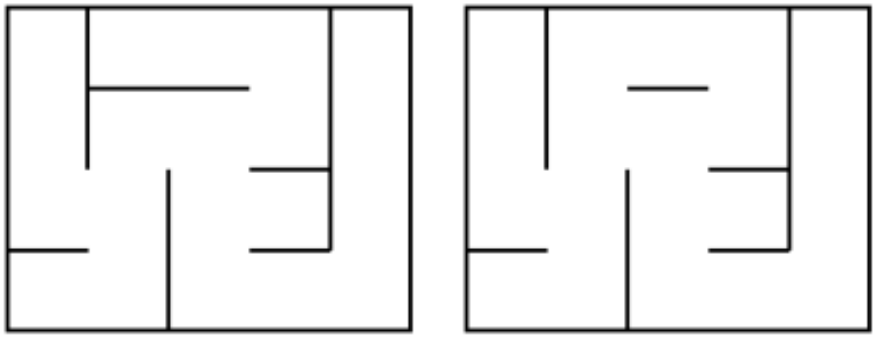
L'objectif de ce TP est de réaliser un générateur de labyrinthe basé sur un générateur de nombre aléatoire (afin d'avoir un labyrinthe de forme aléatoire), ainsi qu'une résolution automatique de labyrinthe. Pour cela, nous allons principalement utiliser les classes et le pouvoir de la récursivité.

3 Cours

3.1 Les labyrinthes

> Le mot désigne dans la mythologie grecque une série complexe de galeries construites par Dédale pour enfermer le Minotaure. En latin, "labyrinthus" signifie "enclos de bâtiments dont il est difficile de trouver l'issue".
>
> Wikipédia

Un labyrinthe est une zone encadrée de mur, contenant au moins une entrée et une sortie, et un chemin reliant les deux. On peut diviser en général les labyrinthes en deux catégories : les labyrinthes parfaits (tous les murs sont reliés entre eux) et les labyrinthes imparfaits (il existe un ou plusieurs îlots de murs isolés et ils sont plus compliqués à résoudre).



Un labyrinthe parfait et un labyrinthe imparfait

Dans notre cas, nous allons nous intéresser aux labyrinthes parfaits, avec une seule entrée et une seule sortie.

3.2 Les énumérations

Une énumération permet de déclarer rapidement un type de variable simple, pouvant prendre N valeurs différentes, connues dès le début. L'utilisation des énumérations permet de clarifier votre code, en utilisant des mots comme `"UP"` ou `"DOWN"` pour indiquer une direction plutôt que d'utiliser un entier par exemple, où seul le développeur qui l'aura codé saura à quelle direction correspond le chiffre 1, par exemple.

On utilise le mot-clé `enum` pour déclarer une énumération. Par convention, on met généralement une majuscule au nom de l'énumération, et on écrit en majuscules les valeurs possibles que peuvent prendre les variables du type de l'énumération.

```
1  enum Direction /* On déclare l'énumération */
2  {
3      NORTH,
4      SOUTH,
5      WEST,
6      EAST
7  };
8
9  static void main()
10 {
11     /* Déclarations de variable de type Direction */
12     Direction compassDirection = Direction.NORTH;
13     Direction goodDirection = Direction.WEST;
14
15     if (compassDirection == goodDirection)
16         Console.WriteLine("You are going in the good direction!");
17     else
18         Console.WriteLine("Something is wrong ...");
19 }
```

3.3 Les listes et les n-uplets

Nous avons déjà vu les listes et leur fonctionnement, assez similaire à ce que vous avez pu utiliser en Caml dans un TP précédent. Les n-uplet, eux, permettent de rassembler en une seule variable plusieurs champs différents. Comme leur nom l'indique, vous n'êtes pas limité dans le nombre de champs que contiendra votre n-uplet. Attention, une fois déclarée, la valeur de ses champs ne peut plus être modifiée. Les n-uplet permettent de retourner plusieurs valeurs à partir d'une seule fonction. De manière similaire aux listes, les types des variables stockées dans le n-uplet sont déclarés à la création du n-uplet, entre les chevrons. On parle de template du n-uplet. Ces types peuvent être des types simples (`int`, `char` ...), des classes que vous avez déclarées ou bien encore d'autres n-uplet/listes templatés !

```
1  static void main()
2  {
3      /* Exemples valides */
4      Tuple<int, char> a = new Tuple<int, char>(2018, 's');
5      Tuple<string, Tuple<int, char>> b =
6          new Tuple<string, Tuple<int, char>>("ACDC", a);
7      int p = a.Item1; // p = 2018
8      List<Tuple<int, char>> l = new List<Tuple<int, char>>();
9  }
```

3.4 Les valeurs par défaut des arguments d'une fonction

Le C# nous permet de déclarer des valeurs par défaut aux arguments d'une fonction. Par exemple :

```
1  static int amazingFunction(int a, int b = 0)
2  {
3       return a + b;
4  }
5
6  static void Main(string[] args)
7  {
8      int x = amazingFunction(1, 1);    // x = 2
9      int y = amazingFunction(1, 0);    // y = 1
10     int z = amazingFunction(1);       // z = 1
11 }
```

Dans l'exemple, la ligne 8 et la ligne 9 correspondent à un appel de fonction "classique", comme tous ceux que vous avez pu faire jusqu'à présent. Les valeurs de `a` et `b` dans la fonction sont remplacées par celles passées en paramètre. Mais à la ligne 10, la valeur de `b` n'est pas précisé lors de l'appel. La variable `b` prend donc la valeur par défaut (0 ici) de l'argument.

Cela permet de gagner du temps lorsqu'une fonction s'appelle dans la très grande majorité des cas avec les mêmes arguments, mais que, dans quelques cas, ces arguments

changent. Vous pouvez mettre autant d'arguments avec des valeurs par défaut que vous souhaitez pour une fonction, à condition que ceux-ci soient à la fin des arguments de la fonction. Attention à ne pas en abuser, vous risqueriez de vous perdre dans votre propre code : un grand pouvoir implique de grandes responsabilités !

```
1 static int amazingFunction1(int a, int b = 5, int c = 1); //Works
2 static int amazingFunction2(int a = 2, int b = 75); //Works too
3 static int amazingFunction3(int a = 4, int b); //Does not work!
```

4 Exercices

Avant de commencer, vous devez aller télécharger le fichier au format zip sur l'intranet. Il contient le squelette des fonctions à écrire. Attention, comme d'habitude, vous ne devez pas modifier le prototype des fonctions fournies, mais rien ne vous empêche d'ajouter les vôtres en plus. Cliquez sur les "+" à coté des régions pour les ouvrir et voir leur contenu.

La solution est divisée en trois fichiers :

- `Program.cs`, qui contient le `main`
- `Maze.cs`, qui contient la classe `Maze`. Elle représente le labyrinthe en lui même sous forme de tableau de cases
- `Cell.cs`, qui contient la classe `Cell`. La classe `Cell` représente une case du tableau du labyrinthe.

4.1 Initialisation

Avant de vouloir générer ou résoudre notre premier labyrinthe, il faut dans un premier temps créer les classes décrites précédemment. Votre première tâche va être de compléter les constructeurs des deux classes.

Maze

Le constructeur de `Maze` prend en paramètre la taille du labyrinthe, et les positions `(x, y)` de départ et d'arrivée. La classe `Maze` contient le tableau de `Cell` représentant le labyrinthe, la taille de ce tableau, ainsi qu'un `Tuple` représentant les positions de la case de départ. Un générateur de nombres pseudo-aléatoires est déjà déclaré, on s'en servira plus tard afin de générer le labyrinthe.

Cell

Le constructeur de `Cell` prend en paramètre le type de la case. Le type de la case est représenté par une énumération. Pour l'instant, les trois types qui nous intéressent sont `START` pour la case de départ, `END` pour la case de fin et `NONE` pour toutes les autres cases. La classe `Cell` contient `linked`, la liste des positions des cases avec lesquelles elle est liée (vide par défaut), son type ainsi que qu'un booléen nommé `visited` (faux par défaut, nous verrons plus tard son utilité).

En plus du constructeur, vous devez également ajouter des getteurs pour `linked`, `visited` et `type`, et un setteur pour `visited`. Toutes les fonctions sont déjà déclarées dans la région `Initialization`.

4.2 Affichage

Maintenant que nous avons généré un labyrinthe, même avec uniquement des murs fermés, il va falloir l'afficher.

```
Maze m = new Maze(3, 2, 0, 0, 2, 1);
m.display();

/* Result - Start (0,0) & End (2,1) */
+--+--+--+
|SS|  |  |
+--+--+--+
|  |  |EE|
+--+--+--+
```

Cell

Dans `Cell`, complétez la fonction `display`. Elle doit retourner `"SS"` si la case est celle du début, `"EE"` si c'est celle de la fin, ou deux espaces sinon.

Maze

La fonction `verticalWall` retourne `" "` si les coordonnées `(x1, y)` et `(x2, y)` sont valides et que les deux cellules sont liées. Elle retourne `"|"` sinon. De même,

`horizontalWall` retourne deux espaces si les coordonnées `(x, y1)` et `(x, y2)` sont valides et que les deux cellules sont liées. Elle retourne `"--"` sinon.

Implémentez ensuite `display`, qui grâce à la fonction `display` de `Cell`, et les fonctions `horizontalWall` et `verticalWall` génère l'affichage final du labyrinthe dans la console.

4.3 Génération du labyrinthe

Bien qu'il existe d'autres méthodes, nous allons utiliser une version de l'algorithme d'exploration exhaustive afin de générer notre labyrinthe.

Cell

On va d'abord compléter la fonction `addLink` (qui ajoute la case aux coordonnées indiquées à la liste des cases avec la laquelle la case actuelle est liée), ainsi que `isLinked` (qui permet de savoir si la case actuelle est liée à la case aux coordonnées indiquée).

Maze

Avant de continuer, implémentez d'abord la fonction `getNotVisitedNeighbor`, qui retourne la liste des coordonnées des cases voisines de celle passée en paramètre et qui n'ont pas été visitées.

Voici l'algorithme que nous allons suivre. Grâce à lui, vous pouvez compléter les fonction `generate` et `generateRec`. Pour chaque case, il va sélectionner une nouvelle case aléatoirement et casser le mur entre les deux, et cela, récursivement jusqu'à ce que le labyrinthe soit entièrement rempli. Pour être sûr qu'on ne va sur chaque case qu'une seule fois, on utilise le booléen `visited` de chaque `Cell`.

```
Mettre des murs entre toutes les cases
Marquer toutes les cases comme non-visitées
Choisir une case C1 aux coordonnées aléatoires

Appliquer l'algorithme suivant à partir de C1 :
Marquer C1 comme visitée
Tant que C1 a au moins une voisine qui n'a pas été visitée
    Choisir C2, une des voisines de C1 qui n'a pas été visitée, de manière aléatoire
```

```
    Supprimer le mur entre C1 et C2
    Recommencer récursivement sur C2
Fin Tant que
```

4.4 Résolution du labyrinthe

Cell

Ajoutez dans `Cell` la fonction `isPath`, qui rajoute la case actuelle dans le chemin final. Pour cela, on va changer le type de la case (uniquement s'il est à `NONE`) en `PATH`.

Maze

Implémentez les fonctions `cleanVisited`, `solve` et `solveRec` à partir de l'algorithme suivant. Le labyrinthe étant parfait, on se contente d'effectuer ici un parcours profondeur. A partir du départ, l'algorithme va essayer tous les chemins possibles jusqu'à en trouver un qui va jusqu'à l'arrivée, puis il va reconstruire le chemin (de la fin vers le début) en remontant les appels récursifs. Quand la fonction récursive retourne `Vrai` c'est que la case actuelle est sur le bon chemin, si elle retourne `Faux`, c'est que le chemin emprunté a mené à une impasse.

```
Marquer toutes les cases comme non-visitées
Se placer sur l'entrée

Tant que toutes les cases liées à la case actuelle (C1) n'ont pas été visitées
    Choisir C2, une des cellules non-visitées liées à C1, de manière aléatoire.
    Marquer C2 comme visitée
    Si C2 est l'arrivée
        Retourner Vrai
    Sinon
        Si C2 n'a aucune voisine non-visitée
            Retourner Faux
        Sinon
            Faire un appel récursif sur C2
            Si l'appel retourne Vrai
                Ajouter C1 au chemin
```

```
                Retourner Vrai
            Si l'appel retourne Faux et que C1 n'a plus de voisines non-visitées
                Retourner Faux
            Fin Si
        Fin Si
    Fin Si
Fin Tant que
```

Affichage

Mettez à jour toutes les fonctions d'affichage afin d'afficher le chemin sous cette forme :

```
Maze m = new Maze(3, 2, 0, 0, 2, 1);
m.generate(); m.display();
m.solve(); m.display();

/* Result of the two calls of the display function */
+--+--+--+               +--+--+--+
|SS      |               |SSXXXXXX|
+--+  +  +               +--+  +XX+
|     |EE|               |     |EE|
+--+--+--+               +--+--+--+
```

4.5 Bonus

Vous pouvez implémenter les bonus suivants une fois que la partie obligatoire marche à 100% :

- `Better Display` : Améliorer l'affichage du labyrinthe, ajouter de la couleur dans la console pour le chemin et les murs ...
- `Better Interface` : Ajouter un menu pour paramétrer le labyrinthe ...
- `Better Path` : Supprimer quelques murs aléatoirement après la génération, puis ensuite le résoudre non pas en trouvant la première solution possible mais la meilleure (vous pouvez chercher du coté de l'algorithme de Dijkstra par exemple)
- `Better Idea` : ce que vous voulez, mais n'oubliez pas de le préciser dans le README!

Brace yourselves, deadline is coming.