TP 10 : Puzzle Solver

1 Consignes de rendu

À la fin de ce TP, vous devrez rendre un dépôt Git respectant l'architecture suivante :

```
csharp-tp10-sherlock.holmes/
|-- README
|-- .gitignore
|-- Puzzle/
    |--Puzzle.sln
    |--puzzle_game
        |--Board
            |--Tiles
                |--Tile.cs
            |--Board.cs
            |--BoardCheck.cs
            |--BoardMove.cs
            |--BoardPrint.cs
            |--BoardSolver.cs
            |--Direction.cs
        |--GenericTree
            |--GenericTree.cs
            |--MinHeap.cs
        |-- Viewer
            |-- tout
```

N'oubliez pas de vérifier les points suivants avant de rendre :

— Remplacez `prenom.nom` par votre propre login.
— Le fichier `README` est obligatoire.
— Pas de dossiers `bin` ou `obj` dans le projet.
— Respectez scrupuleusement les prototypes demandés.
— Retirez tous les tests de votre code.
— **Le code doit compiler !**

README

Vous devez écrire dans ce fichier tout commentaire sur le TP, votre travail, ou plus généralement vos forces / faiblesses, vous devez lister et expliquer tous les boni que vous aurez implémentés. Un `README` vide sera considéré comme une archive invalide (malus).

2 Introduction

2.1 Objectifs

L'objectif de ce TP est de vous introduire à l'utilisation de nouvelles structures de données permettant une représentation plus adaptée des données tout en abordant brièvement la notion de complexité algorithmique mais aussi la praticité des Modèles mathématiques dans la réalisation et l'optimisation d'algorithmes. Cela se fera à travers un exercice concret, la résolution d'un puzzle : le taquin (15 - puzzle).

Attention TP long !

Le TP est assez long ! Il ne faut donc pas s'y prendre en retard !

2.2 La pause histoire...

Le taquin - appelé aussi 15-puzzle - se presente comme ceci :

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

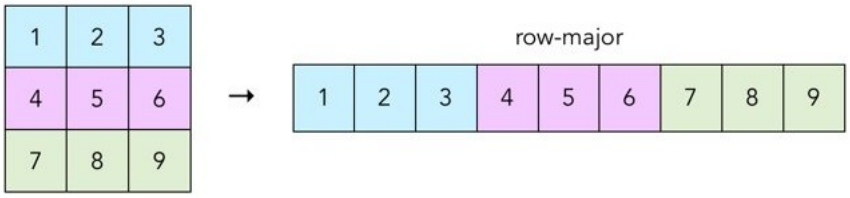
Le taquin est composé de 15 petits carreaux numérotés de 1 à 15 qui glissent dans un cadre prévu pour 16. Le but du jeu est remis dans l'ordre les 15 carreaux à partir d'une configuration initiale quelconque.
Créé vers 1870 aux États-Unis puis revendiqué plus tard en 1891 par Sam Loyd, le jeu a connu à la même époque un véritable engouement, tant aux États-Unis qu'en Europe. En effet, Loyd affirma qu'il avait « rendu le monde entier fou » avec un taquin modifié. Dans la configuration proposée, les carreaux 14 et 15 étaient inversés ; l'espace vide étant placé en bas à droite. Loyd prétendait avoir promis 1 000 USD à celui qui remettrait les carreaux dans l'ordre, mais la récompense n'aurait jamais été réclamée. L'engouement est tel que la célébre revue scientifique *"The American Journal Of Mathematics"* publia un papier de recherche appelée *"Notes ont the 15-puzzles"* ; dans lequel ai démontré a l'aide de simple concepts mathématiques qu'il existe des combinaisions sont a résoudre.

3 Cours

3.1 L'ordre Majeur de rangée/colonnes (Row- and column-major order)

L'ordre majeur de rangée est une façon de représenter les éléments d'un tableau multidimensionnel dans une mémoire séquentielle. Dans l'ordre majeur de rangée, les éléments d'un tableau multidimensionnel sont disposés séquentiellement rangées par rangées, ce qui signifie qu'il faut remplir tous les indices de la première rangée, puis passer à la rangée suivante.

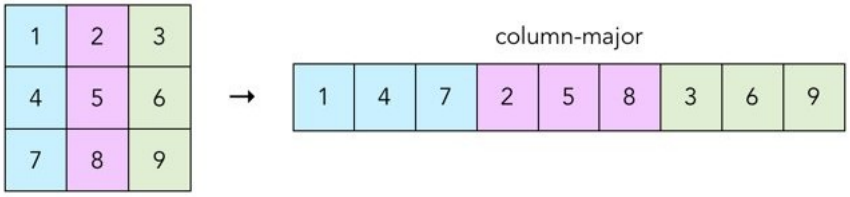
Exemple Row-Major :



Pour accéder à l'élément **table[0][1]** on applique la formule suivante en row-major :

row_major[i * width + j] => row_major[0 * width + 1]

Exemple Column-Major :



Pour accéder à l'élément **table[0][1]** on applique la formule suivante en column-major :

column_major[j * height + i] => column_major[1 * height + 0]

Attention ! Source de bug !

Quand les row- column-major sont mal implémentées, elles peuvent devenir des sources de bug difficiles a trouver ! Prenez donc votre temps !

3.2 Partial Classes

3.2.1 Definition

En C#, vous pouvez diviser l'implémentation d'une classe, d'une structure, d'une méthode ou d'une interface en plusieurs fichiers `.cs` à l'aide du mot-clé *partial*. Le compilateur combinera toutes les implémentations de plusieurs fichiers `.cs` lorsque le programme sera compilé. Cela permet d'organiser son code en plusieurs fichiers afin d'éviter les fameux fichiers de 300 lignes.

3.2.2 Exemple

File `Foo.cs`

```
1     public partial class Foo {
2         private int birthYear;
3         private string name;
4
5         public Foo(string name, int birthYear){
6             this.birthYear = birthYear;
7             this.name = name;
8         }
9
10        private int getAge(){
11            DateTime now = DateTime.Now;
12            return (now.Year - this.birthYear > 0) ?
13                now.Year - this.birthYear :
14                0;
15        }
16    }
```

File `Print.cs`

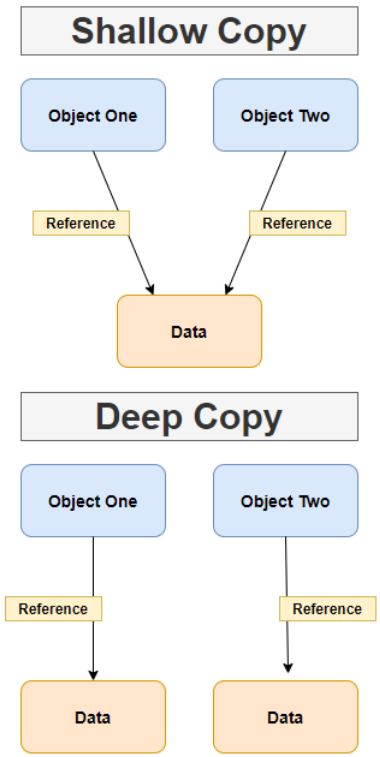
```
1     public partial class Foo {
2
3         public printName(){
4             Console.Write($"My name is {this.name} !");
5         }
6
7         public printAge(){
8             Console.Write($"I am {getAge()} year's old");
9         }
10    }
```

Pour aller plus loin...

Pour plus d'informations à ce sujet nous vous redirigeons vers la *documentation*!

3.3 DeepCopy et Clones

En général, lorsque nous essayons de faire une copie d'objet, ces derniers partagent la même adresse mémoire. Normalement, nous utilisons l'opérateur d'affectation "=" pour copier la référence, et non l'objet, sauf lorsqu'il y a un champ de type valeur. Cet opérateur copiera toujours la référence, et non l'objet réel. Par exemple : Supposons que G1 fait référence à l'adresse mémoire 5000, alors G2 fera également référence à 5000. Ainsi, si l'on modifie la valeur des données stockées à l'adresse 5000, G1 et G2 afficheront les mêmes données.



Le **ShallowCopy** consiste à copier les champs de type valeur d'un objet dans l'objet cible. Les types de références de l'objet sont copiés comme références dans l'objet cible, mais pas l'objet référencé lui-même. Elle copie les types bit par bit. Le résultat est que les deux instances sont cloné et que l'original fera référence au même objet.
Nous pouvons obtenir ce comportement en utilisant MemberwiseClone().

Le **DeepCopy** est utilisé pour faire une copie profonde complète des types de référence interne. Pour cela, nous devons configurer l'objet renvoyé par `MemberwiseClone()` . En d'autres termes, une copie profonde se produit lorsqu'un objet est copié avec les objets auxquels il fait référence.

Exemple de code pour effectuer un ShallowCopy :

```
class Program
{
    static void Main(string[] args)
    {
        Employee emp1 = new Employee();
        emp1.Name = "Adam";
        emp1.Department = "IT";
        emp1.EmpAddress = new Address() { address = "Marseille"};

        Employee emp2 = emp1.GetClone();
        emp2.Name = "Maxence";
        emp2.EmpAddress.address = "Lille"; // Changing the address

        Console.WriteLine("Employee 1: ");
        Console.WriteLine("Name: " + emp1.Name + ", Address: " +
            emp1.EmpAddress.address + ", Dept: " + emp1.Department);

        Console.WriteLine("Employee 2: ");
        Console.WriteLine("Name: " + emp2.Name + ", Address: " +
            emp2.EmpAddress.address + ", Dept: " + emp2.Department);

        Console.Read();
    }
}
public class Employee{
    public string Name;
    public string Department;
    public Address EmpAddress;

    public Employee GetClone(){
        return (Employee)this.MemberwiseClone();
    }
}

public class Address{
    public string address { get; set; }
}
```

```
# Output on STDOUT
Employee 1:
Name: Adam, Address: Lille, Dept: IT
Employee 2:
Name: Maxence, Address: Lille, Dept: IT
```

Exemple de code pour effectuer un DeepCopy :

```
class Program
{
    static void Main(string[] args)
    {
        Employee emp1 = new Employee();
        emp1.Name = "Anurag";
        emp1.Department = "IT";
        emp1.EmpAddress = new Address() { address = "BBSR"};

        Employee emp2 = emp1.GetClone();
        emp2.Name = "Pranaya";
        emp2.EmpAddress.address = "Mumbai";

        Console.WriteLine("Emplpyee 1: ");
        Console.WriteLine("Name: " + emp1.Name + ", Address: " +
        emp1.EmpAddress.address + ", Dept: " + emp1.Department);

        Console.WriteLine("Emplpyee 2: ");
        Console.WriteLine("Name: " + emp2.Name + ", Address: " +
        emp2.EmpAddress.address + ", Dept: " + emp2.Department);

        Console.Read();
    }
}
public class Employee
{
    public string Name;
    public string Department;
    public Address EmpAddress;

    public Employee GetClone(){
        Employee employee = (Employee)this.MemberwiseClone();
        employee.EmpAddress = EmpAddress.GetClone();
        return employee;
    }
}
public class Address
{
    public string address;

    public Address GetClone(){
        return (Address)this.MemberwiseClone();
    }
}
```

```
# Output on STDOUT
Employee 1:
Name: Adam, Address: Marseille, Dept: IT
Employee 2:
Name: Maxence, Address: Lille, Dept: IT
```

3.4 Surcharges

3.4.1 Définition

De nombreux langages de programmation prennent permettent d'avoir des paramètres par défaut ou optionnels. Cela permet au programmeur de rendre un ou plusieurs paramètres facultatifs en leur donnant une valeur par défaut. Cette technique est particulièrement pratique pour ajouter des fonctionnalités à un code existant.

Par exemple, vous pouvez souhaiter ajouter une fonctionnalité à une fonction existante. Admettons que cela nécessite l'ajout d'un ou plusieurs paramètres. Ce faisant, vous briseriez le code existant qui appelle cette fonction, puisqu'il ne transmettrait pas le nombre de paramètres requis. Pour contourner ce problème, vous pouvez définir les paramètres nouvellement ajoutés comme facultatifs et leur donner une valeur par défaut qui correspond à la façon dont le code fonctionnerait avant l'ajout des paramètres.

3.4.2 Exemple

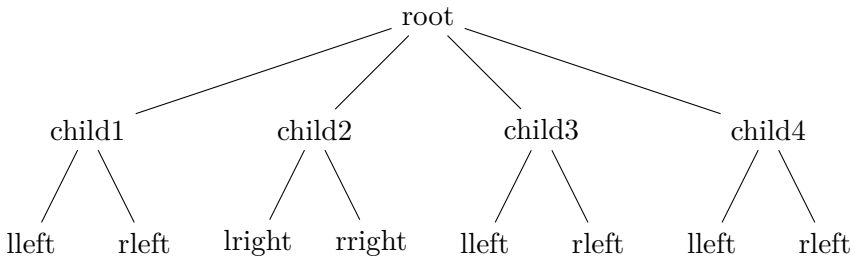
```
class SillyMath
{
    public static int Mult(int number1, int number2){
        return number1 * number2;
    }

    public static int Mult(int number1, int number2, int number3){
        return number1 * number2 * number3;
    }

    public static int Mult(int number1, int number2,
            int number3, int number4){
        return number1 * number2 * number3  * number4;
    }
}
```

3.5 Rappels sur les arbres généraux

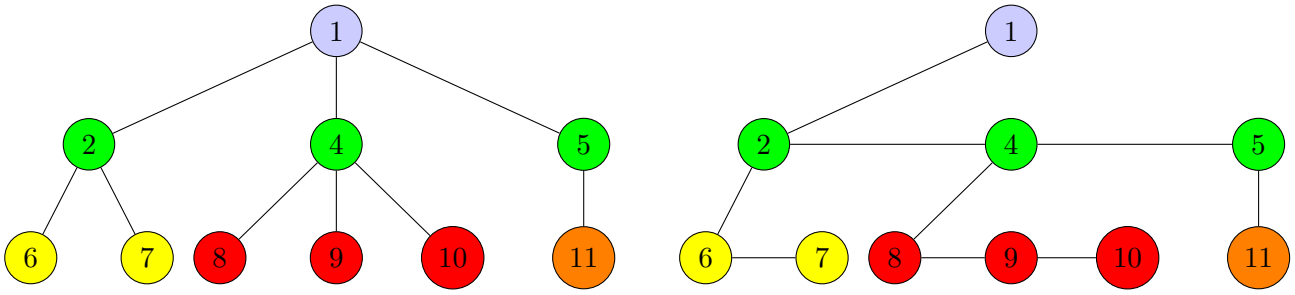
Un arbre est une collection non vide de noeud et d'arêtes possédant les propriétés suivantes :



— Un noeud est un objet simple.
— Une arête est un lien entre deux noeuds.
— Une branche de l'arbre est une suite de noeuds distincts dans laquelle deux noeuds successifs sont reliés par une arête.
— Il existe un noeud spécial appelé racine.
— La propriété qui définit un arbre est qu'il existe exactement une branche entre la racine et chacun des autres noeuds de l'arbre.
— Il est habituel de représenter les arbres avec la racine située au-dessus de tous les autres noeuds.
— Chaque noeud, excepté la racine, possède un noeud "au-dessus de lui", appelé père.
— Les noeuds directement situés sous un noeud sont appelés enfants. On peut aussi parler de frère, grand-père, etc...
— Les noeuds n'ayant pas de descendance sont appelés feuilles, noeuds terminaux ou noeud externes, par opposition aux nœuds qui ont une descendance appelés nœuds internes.

3.5.1 Implémentation

Il existe plusieurs manières d'implémenter un arbre général. Dans ce TP vous serez confronté à l'implémentation ***"Double chaîne"*** (Left-Child Right-Sibling).



(a) Representation classique (b) Representation LC-RS

FIGURE 1 – Representation d'arbre generale

Représentation du premier enfant / du frère suivant (Left Child - Right Sibling)

Dans la représentation premier enfant / frère suivant, les étapes sont les suivantes :

1. A chaque nœud, lier les enfants du même parent de gauche à droite.
2. Supprimer les liens du parent vers tous les enfants sauf le premier enfant.

Puisque nous avons un lien entre les enfants, nous n'avons pas besoin de liens supplémentaires des parents vers tous les enfants. Cette représentation nous permet de parcourir tous les éléments en

commençant par le premier enfant du parent.
Cela dit, il est possible de **labelliser chaque noeud** afin de pouvoir par la suite déterminer un chemin de la racine à un noeud donné.

Conseil

Si vous avez pour projet de faire les bonus, il vous est conseillé de consulter le fichier **GenericTree.cs** !

3.5.2 Définition d'une distance

On considère un ensemble E. On appelle distance sur E une application $d : E \times E \to \mathbb{R}_+$ vérifiant les 3 propriétés suivantes :

1. $d(x, y) = d(y, x), \forall (x, y) \in E^2$
2. $d(x, y) = 0, x = y, (x, y) \in E^2$
3. $d(x, y) \leq d(x, z) + d(z, y), \forall (x, y, z) \in E^3$

E est alors un **espace métrique**.
En informatique, la notion de distance est très importante ! En effet, par exemple elles jouent un rôle important dans l'apprentissage automatique.

Elles constituent la base de nombreux algorithmes d'apprentissage automatique populaires et efficaces, tels que les k-voisins les plus proches pour l'apprentissage supervisé et le regroupement *k-means* pour l'apprentissage non supervisé.

La réelle difficulté réside cela dit dans le choix de ces dernières puisque leur efficacité dépend tout d'abord des situations auxquelles on les confronte. En ce sens, il est important de savoir comment mettre en œuvre et calculer une série de mesures de distance différentes et comment interpréter les résultats. Pour cela, voici une liste non exhaustive de distance que vous devriez rencontrer durant votre scolarité.

Nom	Paramètre	Fonction
distance euclidienne	2-distance	$\sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$
distance de Manhattan	1-distance	$\sum_{i=1}^{n} \lvert x_i - y_i \rvert$
distance de Levenshtein	1-distance	$\text{lev(A,B)}=\begin{cases} \max(\lvert a\rvert,\lvert b\rvert) & \text{si } \min(\lvert a\rvert,\lvert b\rvert) = 0 \\ lev(a-1,b-1) & \text{si } a[0] = b[0] \\ 1 + \min \begin{cases} lev(a-1,b) \\ lev(a,b-1) \\ lev(a-1,b-1) \end{cases} & \text{sinon} \end{cases}$ (1)

Ici nous allons nous concentrer sur **la distance de Manhattan** qui est la distance entre deux points mesurée le long d'axes à angle droit. Dans un plan avec p_1 à (x_1, y_1) et p_2 à (x_2, y_2), elle est $|x_1 - x_2| + |y_1 - y_2|$.

3.5.3 Fonctions heuristiques

Une fonction heuristique est une fonction qui calcule un coût approximatif pour un problème donné (ou qui classe les alternatives).

Par exemple, le problème peut consister à trouver la distance en voiture la plus courte vers un point. Un coût heuristique serait la distance en ligne droite vers ce point. Elle est simple et rapide à calculer, une propriété importante de la plupart des heuristiques. La distance réelle serait probablement plus élevée car nous devons nous en tenir aux routes et elle est beaucoup plus difficile à calculer.

Les fonctions heuristiques sont souvent utilisées en combinaison avec des algorithmes de recherche. Vous pouvez également voir le terme admissible, qui signifie que l'heuristique ne surestime jamais le coût réel. L'admissibilité peut être une qualité importante et est requise pour certains algorithmes de recherche comme A*.

Pour aller plus loin...

Si vous êtes intéressés voici un lien qui pourrait vous être utile : *clickMeDaddy !*

3.6 MinHeap

En informatique, une structure de données de type tas est un arbre spécial qui satisfait à la propriété de tas, ce qui signifie simplement que le parent est inférieur ou égal au nœud enfant pour un tas minimum, aussi appelé tas min, et que le parent est supérieur ou égal au nœud enfant pour un tas maximum, aussi appelé tas max. Le tas binaire a été créé par J.W.J. Williams en 1964 pour **heapsort**.

Un tas binaire est un arbre binaire avec deux autres contraintes :

1. La propriété de forme : Un tas binaire est un arbre binaire complet, ce qui signifie que tous les niveaux de l'arbre sont complètement remplis, sauf éventuellement le dernier niveau. Les nœuds sont remplis de gauche à droite.
2. Propriété du tas : La valeur stockée dans chaque nœud est soit (supérieure ou égale à) OU (inférieure ou égale à) ses enfants selon qu'il s'agit d'un tas maximum ou minimum.

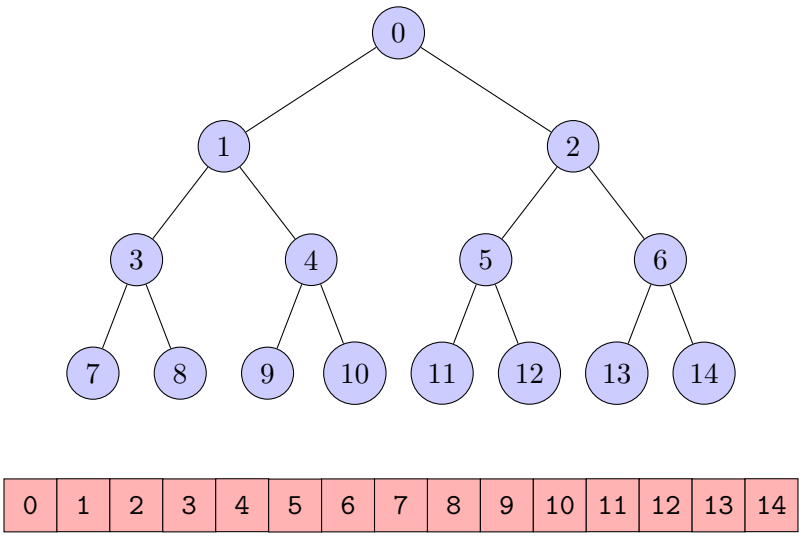
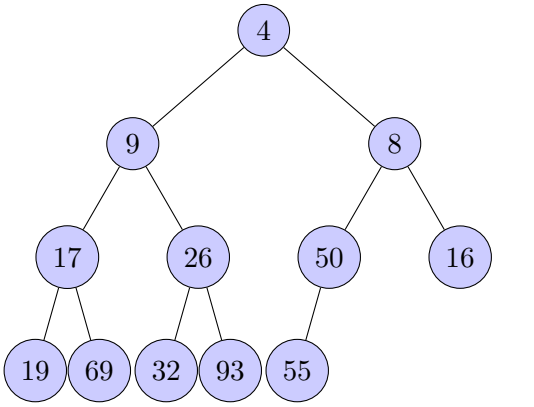


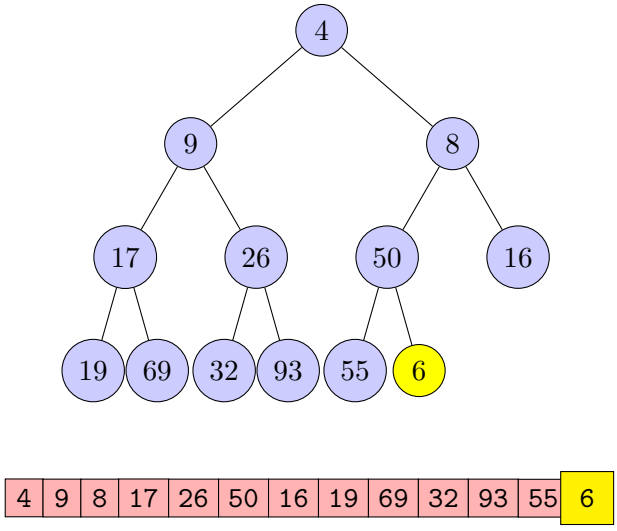
FIGURE 2 – La méthode d'Eytzinger représente un arbre binaire complet comme un tableau

3.6.1 Add

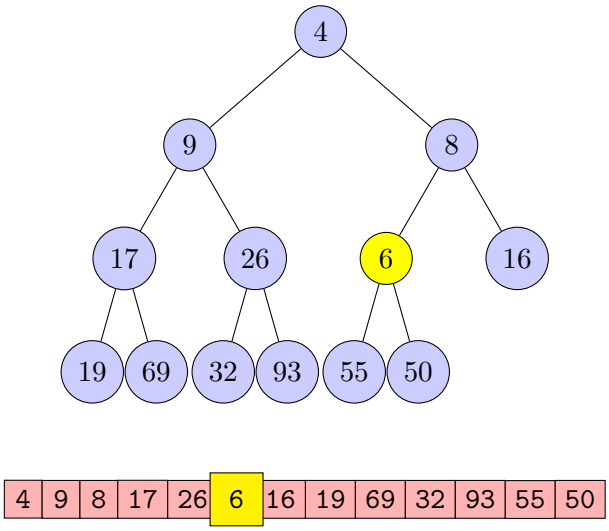
La fonction add(x) est assez simple. Comme avec toutes les structures basées sur des tableaux, nous vérifions d'abord si a est plein (en vérifiant si length(a) = n) et, si c'est le cas, nous augmentons a. Ensuite, nous plaçons x à l'emplacement $a[n]$ et nous incrémentons n. incrémenter n. À ce stade, il ne reste plus qu'à s'assurer que nous maintenons la propriété du tas. Pour ce faire, nous échangeons de manière répétée x avec son parent jusqu'à ce que x ne soit plus plus petit que son parent.



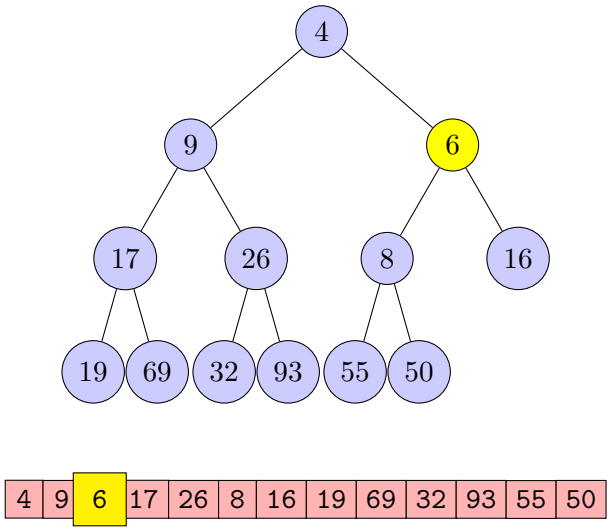
(a) L'arbre de départ



(b) On ajoute le noeud 6 à la finde la liste



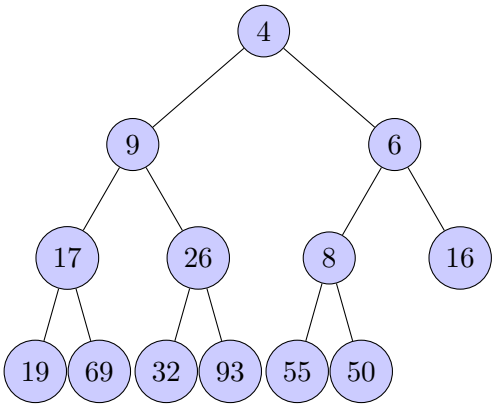
(a) On inverse les noeuds 6 et 50

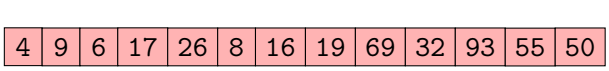


(b) On inverse les noeuds 8 et 6

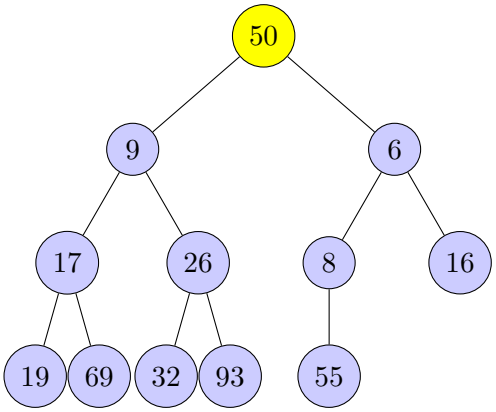
3.7 Remove

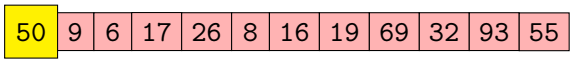
La fonction remove(), qui supprime la plus petite valeur du tas, est un peu plus délicate. Nous savons où se trouve la plus petite valeur valeur (à la racine), mais nous devons la remplacer après l'avoir supprimée et nous assurer que nous maintenons la propriété du tas. La façon la plus simple de le faire est de remplacer la racine par la valeur $a[n-1]$, supprimer cette valeur, et décrémenter n. Malheureusement, le nouvel élément racine n'est probablement pas le plus petit élément, il doit donc être déplacé vers le bas. Pour ce faire, nous comparons de manière répétée cet élément à ses deux enfants. S'il est le plus petit des trois, nous avons terminé. Sinon, nous échangeons cet élément avec le plus petit de ses deux enfants et continuons.



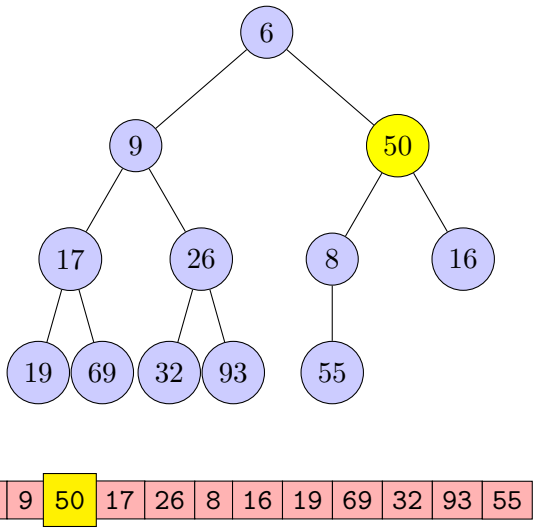


(a) L'arbre de départ

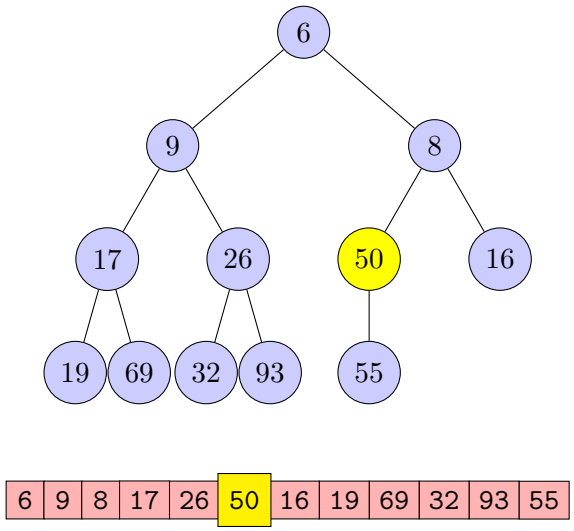




(b) On inverse les noeuds 4 et 50 puis on supprime de noeud 4



(a) On inverse les noeuds 6 et 50



(b) On inverse les noeuds 8 et 50

4 Exercises

Dans cet exercise vous allez implémenter un 15-Puzzle ainsi qu'un algorithme pour le résoudre. Vous allez devoir implémenter un tableau pour représenter le plateau de jeu. Pour ce faire, vous allez utiliser la représentation Row-Major d'une matrice .

Mise en garde

L'utilisation d'une matrice ou d'un tableau à deux dimension est interdit et sera pénalisé

Vous allez commencer par implémenter les fonctions qui vont servir à créer le tableau du 15-Puzzle ainsi que les cases qui le composent. Une fois que nous aurons un tableau à manipuler, vous allez implémenter les différentes fonctions qui nous serviront à mélanger et à résoudre le 15-Puzzle.

Implémentation

Les valeurs de retour et les exceptions que les fonctions renvoient vous seront précisées dans chaque exercice. Si rien n'est dit alors c'est que vous n'aurez pas à gérer ce cas.

Recommandation

Il est très fortement recommandé d'avoir joué au moins une fois au 15-puzzle afin de ne pas être perdu durant l'écriture des fonctions. Voici un petit *lien* !

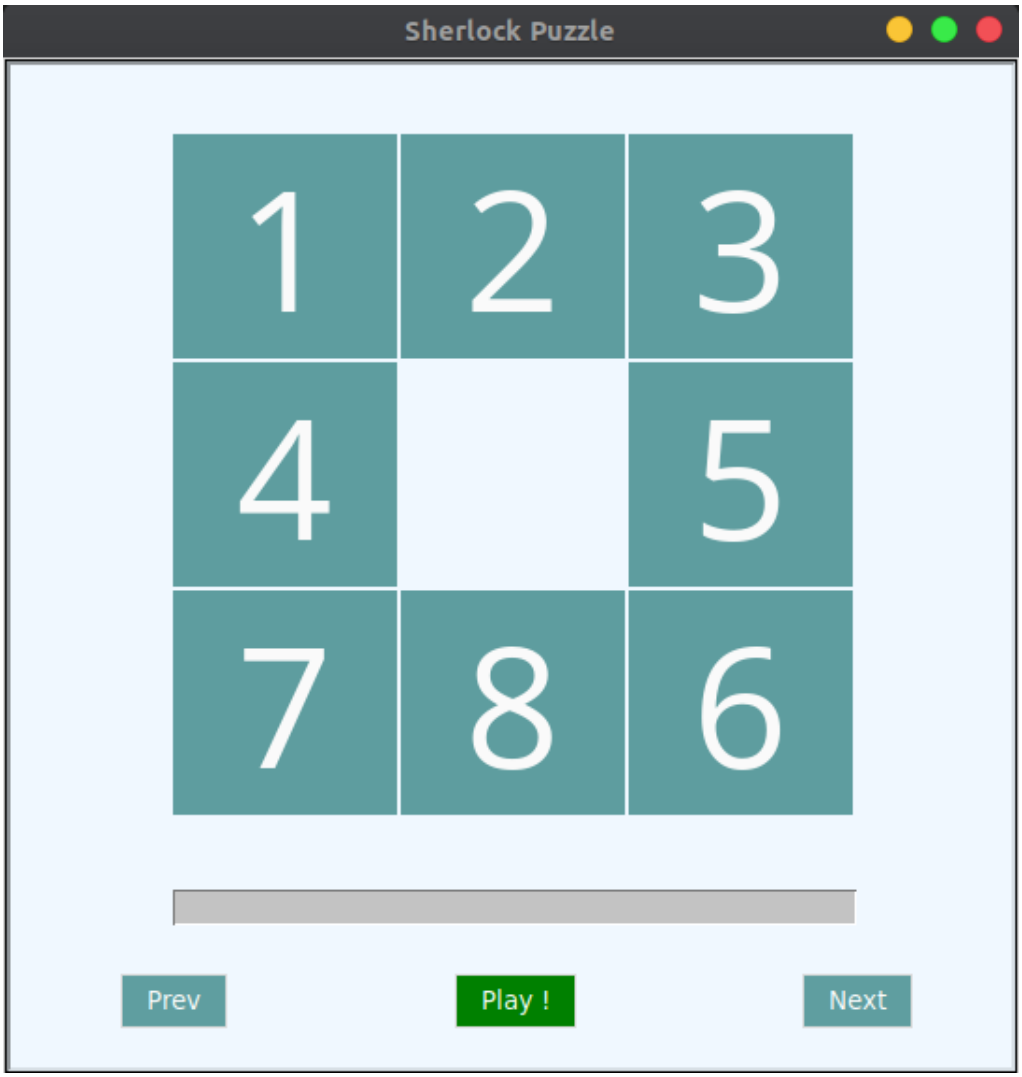
Nous vous avons fourni un puzzle viewer. Ce dernier vous servira à visualiser le puzzle et sa résolution une fois que vous aurez implémenté les fonctions obligatoires.

Attention !

Le viewer à été développé afin de fonctionner sur les machines de l'école. Nous ne pouvons pas garantir qu'il fonctionne sur d'autres machines.

Pour lancer le `Viewer` rien de plus simple !

```
1 Board board = new Board(16);
2
3 //List<Direction> steps = board.SolveBoard();
4 List<Direction> steps = board.SolveBoardBonus();
5
6 Viewer.Parse(board, steps); // Important ! Sinon ça ne fonctionnera pas !
7 Viewer.LaunchViewer();
```



— **Play** : Lance automatiquement la séquence de résolution écrite dans le fichier `Viewer/steps.out`
— **Next** : Prochaine étape.
— **Prev** : Étape précédente.

Installation

Si vous voulez lancer le `Viewer` sur vos machines, il est fortement recommandé d'installer les dépendances dans le fichier `Viewer/requirements.txt` .

```
pip install -r requirements.txt
```

4.1 Board

4.1.1 Constructeurs

Le constructeur de Tile devra suivre le prototype suivant :

```
1  public Tile(int value, bool empty)
```

Dans le constructeur de Tile vous initialiserez les membres suivants :
- type - représente l'état de la case, si elle contient une chiffre ou non.
- value - le chiffre que contient la case

Précision

La value d'une case de type `Tile.EMPTY` est un 0.

Le constructeur de Board devra suivre le prototype suivant :

```
1  public Board(int size)
```

Dans le constructeur de Board vous initialiserez les membres suivants :
- size - le nombre total de cases dans notre tableau, doit être un carré parfait [1].
- width - le nombre de cases dans une rangée
- board - la liste de Tile
- solved - un booléen qui reflète l'état du puzzle initialisé à `false`

Dans les cas suivant vous devez lancer une nouvelle exception de type `ArgumentException` :
- `size <= 0` : l'exception sera lancé avec le message "Size needs to be positive !".
- `size` n'est pas un carré parfait : l'exception sera lancé avec le message "Size needs to be a perfect square !".

4.1.2 Deep copy

Les fonction DeepCopy suivront les prototypes suivants :

```
1  public Board DeepCopy()
2
3  public Tile DeepCopy()
```

Faire des copies de nos Boards ne peux malheureusement pas se faire de manière simple. Si nous écrivions '`Board newBoard = oldBoard`' nous n'aurions pas réellement un nouveau Board car les valeurs de newBoard et de oldBoard seraient toujours liées.Pour palier à ce problème nous allons donc faire une copie profonde [2].

Dans cet exercice vous devrez donc créer un nouveau Board, y copier toutes les valeurs manuellement puis retourner ce dernier.Vous en ferez de même pour la fonction DeepCopy de Tile.

1. Pour plus d'information à propos des carrés parfaits :*https ://en.wikipedia.org/wiki/Square_number*
2. Pour plus d'information à propos des différents types de copies : *https ://en.wikipedia.org/wiki/Object_copying*

4.1.3 AreConsecutive

La fonction AreConsecutive suivra le prototype suivant :

```
1  public static bool AreConsecutive(int []arr)
```

Notre tableau de jeu va contenir des cases numérotées de 1 à 15 ainsi qu'une case vide. Ces cases ne sont pas forcément dans l'ordre mais tous les chiffres entre 1 et 15 y sont présent. Pour cette fonction vous allez donc vérifier que les éléments dans `arr` sont les entiers compris entre 0 et la taille de la liste.

Vous retournerez `True` si ces conditions sont vérifiées et `False` sinon

Exemple :

```
1  int[] array = new int[]{0,1, 2, 3, 4, 5};
2  Board.AreConsecutive(array); // True
3  array = new int[]{5, 4, 2, 1, 3, 7, 6, 0};
4  Board.AreConsecutive(array); // True
5  array = new int[]{0};
6  Board.AreConsecutive(array); // True
7  array = new int[]{1, 6, 8, 10, 12};
8  Board.AreConsecutive(array); // False
9  array = new int[]{1, 2, 3, 3, 4};
10 Board.AreConsecutive(array); // False
11 array = new int[]{1, 3, 5, 7, 9};
12 Board.AreConsecutive(array); // False
```

4.1.4 Fill

La fonction Fill suivra le prototype suivant :

```
1  public void Fill()
```

Dans cette fonction vous devez remplir le membre `board` avec des cases numérotées de 1 à 15 et une case vide. Ces cases doivent être ordonnées de manière croissante et la liste doit se terminer par une case vide.

```
1  public void Fill(int[] array)
```

Dans cette fonction vous devez remplir la board à partir de la liste `array` donné en paramètre. N'oubliez pas de vérifier que la liste est valide et qu'il existe bien une unique case vide. Attention ! si la liste donné en paramètre n'est pas consécutif alors levez une exception de type `ArgumentException`.

Rappel

N'oubliez pas que la value d'une case de type `Tile.EMPTY` est un 0.

4.2 BoardCheck

4.2.1 IsCorrect

La fonction IsCorrect devra suivre le prototype suivant :

```
1  public bool IsCorrect()
```

La fonction IsCorrect est relativement simple, vous devez vérifier que le Puzzle à bien été résolu. Nous considérons le puzzle comme étant résolu lorsque toutes les Tile contenues dans le Board sont ordonné et que seule la dernière Tile est de type `TileType.EMPTY`.

Vous retournerez `True` si le Board est résolu et `False` sinon.

4.2.2 FindEmptyPos

La fonction FindEmptyPos suivra le prototype suivant :

```
1  public int FindEmptyPos()
```

Dans cette fonction, vous devez retourner l'index de la case vide. Retournez -1 en cas d'erreur.

Exemple `board` :

```
1  Board board = new Board(16);
2  int array = new int[]
3  {
4      3, 5, 6, 4,
5      1, 2, 7, 8,
6      9, 10, 0, 12,
7      13, 14, 11, 15
8  };
9  board.Fill(array);
10
11 Console.WriteLine($"The Empty Tile is at position {board.FindEmptyPos()}");
```

```
1  The Empty tile is at position 10
```

4.2.3 IsSolvable

La fonction IsSolvable suivra le prototype suivant :

```
1  public bool IsSolvable()
```

Il est possible de savoir à l'avance si un tableau est résolvable en comptant le nombre d'inversions présent dans ce dernier. Vous allez implémenter ce test dans cette fonction.

3	5	6	4
1	2	7	8
9	10		12
13	14	11	15

(a) Solvable Grid

1	2	3	4
5	6	7	8
9	10	11	12
13	15	14	

(b) Non-Solvable Grid !

FIGURE 7 – Some grid example

Ici, N représente la taille de la grille.

1. Si N est impaire et que le nombre d'inversions est pair, l'énigme est soluble.
2. Si N est paire, le puzzle est soluble si :
   — La case vide se trouve sur une ligne paire en comptant à partir du bas (avant-dernier, quatrième dernier, etc.) et le nombre d'inversions est impair.
   — La case vide est sur une ligne impaire en comptant à partir du bas (dernier, troisième, cinquième, etc.) et le nombre d'inversions est pair.
3. Pour tout autre cas, le puzzle n'est pas soluble.

Comment compter les inversions me dites vous ?
Une paire de nombres dans un tableau est une inversion lors ce qu'ils apparaissent dans le désordre. Supposons que nous ayons le tableau suivant **{1, 2, 4, 3}**, alors la paire **(4,3)** est une inversion car $4 > 3$ et 4 apparaît avant 3. Le tableau **{3, 1, 2, 4}** a deux inversions **(3,1)** et **(3,2)**, tandis que **{1, 2, 3, 4}** n'a pas d'inversions.

Plus formellement deux éléments `arr[i]` et `arr[j]` forment une inversion si les deux conditions suivantes sont vérifiées : `arr[i] > arr[j]` et `i < j`.

4.3 BoardMove

4.3.1 GetPossibleDirections

La fonction `GetPossibleDirections` retourne un tableau contenant toutes les directions possible et valide avec lesquelles la `Empty Tile` peut échanger de place.
La fonction `GetPossibleDirections` suivra le prototype suivant :

```
public Direction[] GetPossibleDirections()
```

```
int[] array = new int[] {
    1,2,3,
    4,5,6,
    7,8,0
};
Board board = new Board(9);
board.Fill(array);
foreach (var direction in board.GetPossibleDirections()){
    Console.Write($"{direction} ");
}
```

```
UP LEFT
```

4.3.2 SwapTile

La fonction `SwapTile` suivra le prototype suivant :

```
public void SwapTile(int i1, int i2)
```

`SwapTile` échange la position de deux cellules !
Si les cellules sont de même type, vous devez lever une exception de type `ArgumentException()`

4.3.3 MoveDirection

La fonction MoveDirection suivra le prototype suivant :

```
public bool MoveDirection(Direction direct)
```

`MoveDirection` permet de déplacer la `Empty Tile` en échangeant cette dernière avec la cellule se trouvant dans l'une des directions suivantes : `UP, DOWN, LEFT, RIGHT`.

```
Board board = new Board(9);
int[] array = new int[]{
    1,2,3,
    4,5,6,
    7,8,0
};

board.MoveDirection(LEFT);
board.Print();
board.MoveDirection(UP);
board.Print();
```

```
1
2   ┌─┬─┬─┐
    │1│2│3│
3   ├─┼─┼─┤
4   │4│5│6│
5   ├─┼─┼─┤
6   │7│X│8│
7   └─┴─┴─┘
8
9   ┌─┬─┬─┐
    │1│2│3│
10  ├─┼─┼─┤
11  │4│X│6│
12  ├─┼─┼─┤
13  │7│5│8│
14  └─┴─┴─┘
```

4.3.4 Shuffle

La fonction Shuffle suivra le prototype suivant :

```
1  public void Shuffle(int nbr)
```

Mélange le Board tout en la gardant résoluble. Pour mélanger le Board vous allez devoir faire `nbr` mouvements aléatoires valide. Attention `nbr` se doit d'être positif ! Si `nbr` n'est pas valide alors levez une exception de type `ArgumentException`

4.4 BoardPrint

4.4.1 PadCenter

La fonction PadCenter suivra le prototype suivant :

```
1 public string PadCenter(string s, int width, char c)
```

`PadCenter` retourne la chaîne de caractère `s`, complétée à droite, à gauche ou dans les deux sens, avec le caractère `c` jusqu'à ce qu'elle atteigne la taille de `width`.

```
1 string result = PadCenter("Hello", 9, '\$');
2 Console.WriteLine(result);
3 result = PadCenter("Hello", 8, '\$');
4 Console.WriteLine(result);
5 result = PadCenter("Hello", 3, '\$');
6 Console.WriteLine(result);
7 result = PadCenter("Hello", -3, '\$');
8 Console.WriteLine(result);
```

```
1 $$Hello$$
2 $$Hello$
3 Hello
4 Hello
```

4.4.2 PrintLine

La fonction PrintLine suivra le prototype suivant :

```
1 public void PrintLine(int i, int width, int longest_number)
```

La fonction PrintLine prend 3 arguments :
— longest_number : La longueur du plus grand nombre. i.e 100 -> 3, 1000 -> 4, 2 -> 1.
— width : largeur de la grille
— i : l'index de la ligne de la matrice
Elle permet d'afficher les lignes intermédiaire qui compose le tableau.

```
1  // Cas ou i = 0, première ligne de la grille
2  Console.WriteLine("New_line:");
3  PrintLine(0, 4, 2); // On print pour une grille allant de 1 jusqu'à 15
4
5  // Cas ou i != 0 et != width, ligne intermédiaire
6  Console.WriteLine("New_line:");
7  PrintLine(2, 4, 2); // On print pour une grille allant de 1 jusqu'à 15
8
9  // Cas ou i = width, dernière ligne de la grille
10 Console.WriteLine("New_line:");
11 PrintLine(4, 4, 2); // On print pour une grille allant de 1 jusqu'à 15
```

```
New_line:
╔═══╦═══╦═══╦═══╗

New_Line:
╠═══╬═══╬═══╬═══╣

New_Line:
╚═══╩═══╩═══╩═══╝
```

4.4.3 Print

La fonction Print suivra le prototype suivant :

```
public void Print()
```

Elle permet de print la Board.

```
int[] sizes = new int[] {16, 9, 1};

foreach (int size in sizes)
{
    Board board = new Board(size);
    board.Fill();
    Console.WriteLine(\$"Board of size {size}:");
    board.Print();
}
```

Astuce !

L'utilisation des fonctions implémentées plus haut peut s'avérer d'une grande utilité !

Attention !

L'affichage se doit d'être identique a l'exemple ci-dessous !

```
int[] sizes = new int[] {16, 9, 1};

foreach (int size in sizes){
    Console.WriteLine($"Board of size {size}");
    Board board = new Board(size);
    board.Print();
}
```

```
Board of size 16:
╔═══╦═══╦═══╦═══╗
║ 1 ║ 2 ║ 3 ║ 4 ║
╠═══╬═══╬═══╬═══╣
║ 5 ║ 6 ║ 7 ║ 8 ║
╠═══╬═══╬═══╬═══╣
║ 9 ║ 10║ 11║ 12║
╠═══╬═══╬═══╬═══╣
║ 13║ 14║ 15║ X ║
╚═══╩═══╩═══╩═══╝
Board of size 9:
╔═══╦═══╦═══╗
║ 1 ║ 2 ║ 3 ║
╠═══╬═══╬═══╣
║ 4 ║ 5 ║ 6 ║
╠═══╬═══╬═══╣
║ 7 ║ 8 ║ X ║
╚═══╩═══╩═══╝
Board of size 1:
╔═══╗
║ X ║
╚═══╝
```

4.5 SolveBoard

4.5.1 ManhattanDistance

La fonction ManhattanDistance suivra le prototype suivant :

```
1  public int ManhattanDistance(int i1, int i2)
```

Vous allez devoir implémenter une fonction qui permet de déterminer la plus courte distance de Manhattan entre 2 cellules de la Board. La formule vous a été donné dans la partie *cours* !

La fonction possède deux arguments :
— i1 : L'index de la première cellule en Row Major
— i2 : L'index de la deuxième cellule en Row Major
Si l'index des cellules n'est pas correct, vous devez lancer l'exception suivante : `ArgumentException`.

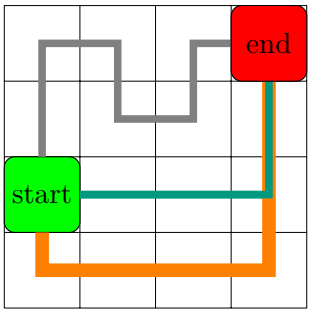


FIGURE 8 – Manhattan Distance on the grid

4.5.2 CalculateHeuristic

Pour permettre de faire un choix lors de la résolution du puzzle, il est vous est nécessaire d'implémenter une fonction qui classe les alternatives à chaque étape de branchement en fonction de l'emplacement de chaque cellule afin de décider quelle branche suivre. Ici, la fonction calcul la somme des distances de Manhattan de chaque cellule avec sa position final.
Petit exemple :

$$\alpha(x,y) = \begin{cases} d_{Manhattan}(x,y) & \text{if } goal[y] > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\beta_{Heuristic}(board, goal) = \sum_{i=0}^{N} \alpha(board[i], goal[i])$$

1	2	3	5
11	6	7	4
9	10		8
13	14	15	12

Tout d'abord sélectionnons les cellules qui ne sont pas à la bonne place (En ignorant la cellule vide biensur) !

1	2	3	5
11	6	7	4
9	10		8
13	14	15	12

1	2	3	5
11	6	7	4
9	10		8
13	14	15	12

On calcul alors pour chaque cellule mal placé sa distance de Manhattan !

			5
end			

On applique donc le même procédé pour chaque cellules et on additionne le tout !

```
Board board = new Board(9);
int[] array = new int[]
{
    4,3,1,
    7,2,5,
    0,8,6
};

board.Fill(array);
Console.WriteLine($"Heuristic Value of the board \
{board.CalculateHeuristic()}.");

array = new int[]
{
    1,2,3,
    4,5,6,
    0,8,7
};
board.Fill(array);

Console.WriteLine($"Heuristic Value of the board \
{board.CalculateHeuristic()}.");
```

```
Heuristic Value of the board 8.
Heuristic Value of the board 2.
```

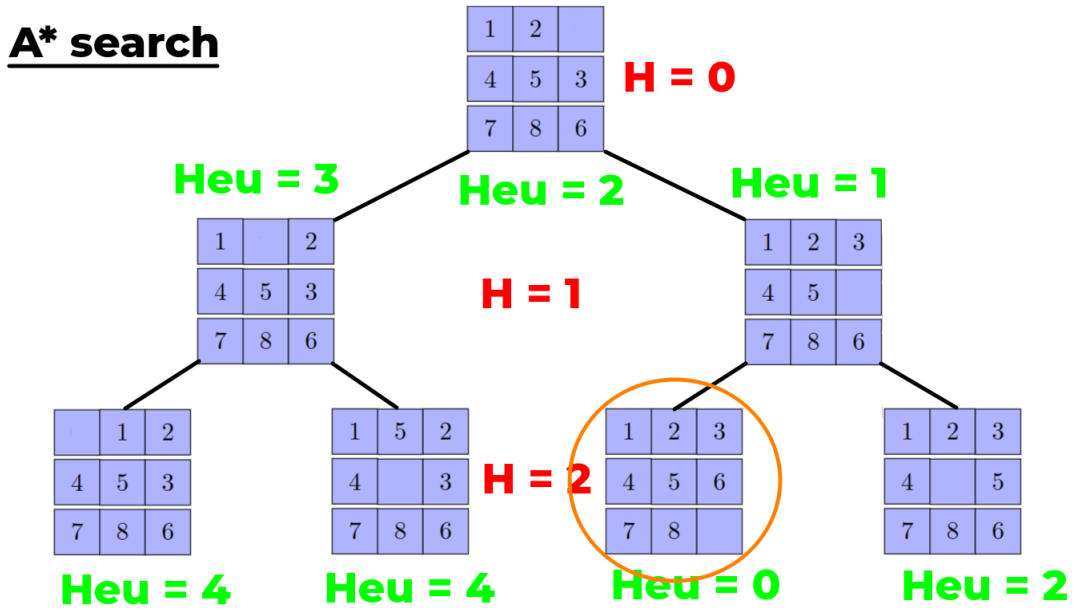
La fonction CalculateHeuristic suivra le prototype suivant :

```
public int CalculateHeuristic()
```

4.5.3 SolveBoard

Vous allez devoir implémenter la fonction qui permet de résoudre le board ! Elle renvoie une liste de `Direction` qui permet la résolution du board. Pour cela nous vous proposons un algorithme naïf qui ne marche que pour les puzzle qui ne contiennent pas de cycles ! La fonction SolveBoard suivra le prototype suivant :

```
public List<Direction> SolveBoard()
```



Pour chaque mouvements possible, déterminez et sélectionnez le fils qui permettra d'arriver en moins d'étapes possible et continuez récursivement jusqu'à tomber sur la board final ! Nous ne donnerons pas plus d'explications quant au choix du fils, vous allez devoir chercher par vous même :) .

Attention aux cycles !

Dans certains cas, les puzzles peuvent contenir des cycles. Votre algorithme n'est pour l'instant pas censé les gérer. Par exemple ce *board* la contient un cycle ! A vous de trouver où :) .

```
Board board = new Board(9);

int[] array = new int[]{
    1,2,3,
    4,0,5,
    7,8,6
};
board.Fill(array);
List<Direction> steps = board.SolveBoardBonus();

Console.WriteLine("Resolution finished");
foreach (var step in steps) {
    Console.Write($"{step} ");
}
```

```
Resolution finished
RIGHT DOWN
```

5 Bonus

Dans cette partie nous allons optimiser l'algorithme de résolution afin qu'il puisse résoudre les puzzles incluant des cycles.

5.1 MinHeap

Avant de commencer cette partie il est conseiller de lire la partie du cours sur les MinHeap. Vous allez devoir coder dans le fichier `GenericTree/MinHeap.cs`. Dans le fichiers, plusieurs fonctions non-implémenté existe. Cela dit, elles ne sont pas obligatoire mais dont l'utilisation est fortement conseillé.

5.1.1 MinHeapify

La complexité temporelle de cette opération est $O(\log n)$. Si la valeur de la clé décroissante d'un noeud est supérieure à celle du parent du noeud, on continue. Sinon, on traverse vers le haut pour corriger la violation de la propriété du tas. La fonction `MinHeapify` suivra le prototype suivant :

```
1  public void minHeapify(int pos);
```

Pour plus d'informations nous vous rédigeons vers ce *lien*.

5.1.2 Enqueue

La fonction `Enqueue` ajoute l'élément dans la MinHeap en la gardant cohérente.
La fonction `Enqueue` suivra le prototype suivant :

```
1  public void Enqueue(HeapElement<T> element)
```

Pour plus d'informations veuillez vous referez au cours *ici*.

5.1.3 Dequeue

La fonction `Dequeue` retourne le minimum tout en gardant la MinHeap cohérente.
La fonction `Dequeue` suivra le prototype suivant :

```
1  public HeapElement<T> Dequeue();
```

Pour plus d'informations veuillez vous referez au cours *ici*.

5.2 Solver

5.2.1 SolveBoardBonus

Pour la dernière fonction de ce TP, vous allez devoir implémenter l'algorithme de **A*** sur des arbres généraux.
La fonction `SolveBoardBonus` suivra le prototype suivant :

```
1  public List<Direction> SolveBoardBonus()
```

Astuce

Quand vous cherchez un algorithme, prenez l'habitude de chercher sur **http ://scholar.google.fr**. Vous comprendrez très vite pourquoi :).

There is nothing more deceptive,
than an obvious fact.