TPCS12 : What The Hex

1 Consignes de rendu

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

```
– rendu-TPCS12-login_x.zip
    |– rendu-TPCS12-login_x/
        |– AUTHORS
        |– README
        |– WhatTheHex/
            |– WhatTheHex.sln
            |– WhatTheHex/
                |– Tout sauf bin/ et obj/
```

Vous devez remplacer **login_x** par votre propre login. Vérifiez ces consignes avant de rendre :

Fichier AUTHORS

Le fichier nommé `AUTHORS` doit respecter le format habituel, une étoile '*', un espace ' ', votre login_x et un retour à la ligne. AUTHORS est un fichier sans extension.

Remarques

Lisez le sujet dans son intégralité avant de commencer. Une fois la lecture terminée, relisez-le. Réalisez les exercices dans l'ordre. Avant de rendre votre travail, Vérifiez les points suivants :

- Les fonctions demandées respectent le prototype du sujet
- Les dossiers bin/ et obj/ sont supprimés
- Le fichier AUTHORS est présent.
- **Votre code compile**

2 Le Jeu de Hex

2.1 Principes

Le jeu de Hex est un jeu de société, se jouant sur un plateau en forme de losange composé de cases hexagonales. Ce jeu oppose deux joueurs, chacun dispose de jetons. La partie est terminée lorsqu'un joueur a formé un chemin contiguë de jetons reliant les deux bords opposés qui lui sont associés. Un joueur doit tenter de relier deux bords du plateau alors que l'autre doit tenter de relier les deux autres bords. Les joueurs jouent chacun leur tour. Il est possible de poser un jeton sur n'importe quelle case non occupée, et il est impossible de déplacer un jeton une fois posé. Plus d'info sur : https ://fr.wikipedia.org/wiki/Hex

La partie commence avec un plateau vide. Sur notre implémentation, le joueur de couleur rouge engage la partie. Le joueur rouge doit relier le bord en bas à gauche avec le bord en haut à droite, tandis que le joueur de couleur bleu doit relier le bord en haut à gauche et le bord en bas à droite. Nous vous conseillons de jouer sur un plateau de taille 6x6 et non un 11x11 comme recommandé par les règles, cela vous permet de tester le jeu et votre code. Vous pouvez modifiez les variables `gameInfo.row` et `gameInfo.column`

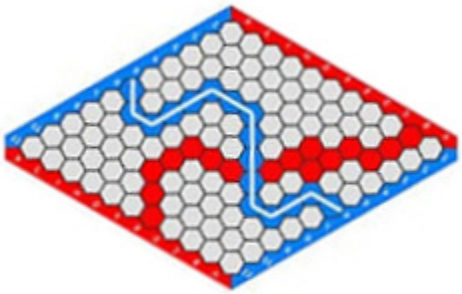

FIGURE 1 – Partie de Hex gagné par les bleus

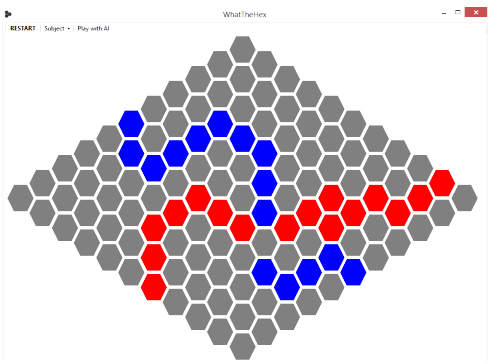

FIGURE 2 – Notre plateau de jeu

2.2 Bilan et analyse logique sur le jeu de Hex

Le jeu de Hex à l'intéressante particularité de **ne pas pouvoir donner de match nul**. Explication rapide.

Lorsque toutes les cases sont remplies par les jetons des joueurs, il y a forcément un gagnant. Montrons cela rapidement : un plateau entièrement rempli ne contient que 2 couleurs. Si l'un des joueurs n'a pas réussi à former de chemin entre la bordure et son opposée, cela veut dire que les jetons sont au moins coupés en deux groupes de jetons adjacents. Pour couper un groupe, sur toute la longueur du plateau, il faut un groupe adjacent de la couleur adverse qui effectue un tracé sur les deux autres bordures. Donc, si joueur 1 n'a pas gagné, c'est que joueur 2 a gagné et vice-versa.

Autre particularité notable : **le joueur 1 a forcément une stratégie gagnante**.

Partons du fait qu'il y a forcément un gagnant. Cela induit que le joueur 1 peut gagner à coup sûr s'il joue correctement. On dit que le joueur 1 a une stratégie gagnante. Par l'absurde, si joueur 1 n'a pas de stratégie gagnante, joueur 2 en a une car il y a forcément un gagnant. Or, si joueur 2 a une stratégie gagnante, alors joueur 1 en a une aussi, car c'est joueur 1 qui commence à jouer et que poser un jeton ne peut pas servir le joueur adverse. On a donc :

J1 n'a pas de stratégie gagnante => J1 a une stratégie gagnante.
Absurde. On conclut que J1 a forcément une stratégie gagnante.

2.3 Exercice : Déterminer la fin du jeu

Pour cet exercice, vous allez devoir remplir la fonction déterminant si un joueur à terminé une partie. La fonction prendra en paramètre le tableau d'entier représentant une partie. Ouvrez le fichier **gameInfo.cs**. C'est ici que vous devez coder cette fonction qui est la seule fonction qu'il manque pour finir le jeu.

```
1 public static int isFinished(int[,] board)
```

Vous pouvez faire une ou deux fonction(s) intermédiaire(s), par exemple pour tester si l'un des deux joueurs en particulier a gagné la partie. Dans tous les cas, vous allez devoir faire une fonction récursive qui s'appellera sur toutes les cellules voisines jusqu'à un cas final (cellules pas occupées par le même joueur, cellules en dehors des bords, etc). La fonction récursive renvoie vrai si elle atteint le bord opposé.

Conseils :

— Itérez sur la première colonne pour les rouges, et sur la première ligne pour les bleus. À chaque fois que vous trouvez une cellule de la couleur voulue, appelez la fonction récursive.
— Vous pouvez afficher les coordonnées des cases du plateau dans le menu 'Subject' du Winform.
— Vous pouvez utiliser la fonction `AllocConsole()` dans le constructeur du plateau, se trouvant dans **Board.cs** cette fonction ouvre une console, utile pour les `Console.Write()` !

Pour éviter d'appeler plusieurs fois la fonction récursive sur une même case lors d'un parcours et ainsi éviter une boucle infinie, vous pouvez vous servir de la variable `visited` qui est un entier stocké dans `Cell`. Vous avez accès au tableau d'entiers `board[,]` mais aussi au tableau `cells[,]` qui contient des informations supplémentaires au sujet de chaque cellule.

3 La Généricité

3.1 Introduction

Les types génériques, ou types templatés, permettent de factoriser votre code. Le principe de généricité permet d'écrire des classes ou méthodes dont la spécification des types n'est pas donnée jusqu'à ce que la classe ou la méthode soit déclarée et instanciée. Leur usage permet de gagner en performance et de sécuriser les types lors de la compilation (les génériques évitent d'utiliser les casts).

L'utilisation la plus courante des classes génériques est avec des collections comme les listes triées, les tables de hachage, les piles d'attente, les arborescences et autres collections où les opérations telles que l'ajout et la suppression d'éléments sont exécutées de façons similaire indépendamment du type des données. Voici un exemple de type que vous utilisez depuis longtemps, utilisant les génériques, le type `List` :

```
1  // Declaration d'une liste, contenant des entiers
2  List<int> myList = new List<int>();
3
4  // Declaration d'une liste, contenant des courants alternatifs, parfois continus
5  List<ACDC> deadlineIsComing = new List<ACDC>();
```

Maintenant, regardons comment implémenter une classe qui accepte n'importe quel type en paramètre. Prenons l'exemple d'un point qui contient deux paramètres : son abscisse et son ordonnée.

```
1  // Point.cs
2  public class Point<Coordinate>
3  {
4      Coordinate x;
5      Coordinate y;
6
7      public Point(Coordinate x, Coordinate y)
8      {
9          this.x = x;
10         this.y = y;
11     }
12     public Coordinate getX()
13     {
14         return x;
15     }
16     public Coordinate getY()
17     {
18         return y;
19     }
20 }
```

```
// Program.cs
public static void Main (string[] args)
{
    Point<int> a = new Point<int>(5, 7);
    Point<int> b = new Point<int>(5, 7);
    Point<float> c = new Point<float>(2.5f, 3.42f);
    Point<Point<int>> d = new Point<Point<int>>(a, b);
}
```

Sachez que l'on peut donner plusieurs type en paramètre d'une classe : un Tuple reçoit deux types : le type du premier paramètre et le type du deuxième. Pour reproduire ce comportement, rien de plus simple :

```
public class Tuple<Type1, Type2>
{
    Type1 item1;
    Type2 item2;

    public Tuple(Type1 item1, Type2 item2)
    {
        this.item1 = item1;
        this.item2 = item2;
    }
}
```

3.2 Exercice : Arbres généraux

Le but de ce TP est de coder une intelligence artificielle capable de jouer au jeu de Hex de manière intelligente. Cette IA sera susceptible de vous battre si vous y réussissez. Une telle technologie s'appuie évidemment sur des algorithmes, et ces algorithmes ont besoin d'une structure que vous allez coder dans ce chapitre : un arbre général !

Vous allez devoir coder un arbre général templaté, c'est-à-dire que cet arbre aura des noeuds qui contiendront des éléments de n'importe quel type ! Cela peut être un arbre d'entiers, un arbre de chaîne de caractère, un arbre d'arbre...

Commençons par le commencement. Créez un fichier **Tree.cs**, dans lequel vous déclarez une classe `Tree<Element>`.

`Element` est donc le nom donné au type que l'on reçoit en paramètre. La classe `Tree` contient deux variables privées : une liste de fils (qui sont eux-mêmes des arbres) et un noeud du type `Element` ;

Créez un constructeur de la classe `Tree` qui prend en paramètre unique le noeud de l'arbre que l'on est en train de créer.

```
1 public Tree (Element node)
```

Créez maintenant un getter/setter de la classe qui permettra de modifier ou d'obtenir le noeud actuel.

```
1 public Element Value
```

Pour les trois fonctions suivantes, le prototype n'est pas donné, à vous de réfléchir à la meilleure manière d'implémenter un arbre général à partir de ce que vous savez maintenant.

Implémentez :

— une fonction `addChild` qui ajoute un enfant à l'arbre actuel.

— une fonction `getChild` qui renvoit l'enfant correspondant à un index donné en paramètre.

— une fonction `nbChild` qui renvoit le nombre d'enfants de l'arbre actuel.

Pour finir, et pour vérifier que votre arbre fonctionne correctement, écrivez une fonction `print()` qui va afficher l'arbre de la manière suivante : (1, 1.1, 1.2, 1.2.1 sont des noeuds, 1.1 et 1.2 étant les fils de 1, 1.2.1 étant le fils de 1.2)

1 < 1.1, 1.2 < 1.2.1 > >

1, 1.1, 1.2 [...] représentent les noeuds, vous devez afficher la valeur des noeuds (par exemple un noeud contenant un float avec comme valeur 2.5 devra afficher 2.5). Si le type ne permet pas à Console.Writeline d'afficher correctement la valeur, ce n'est pas grave. Ce cas n'est pas à traiter.

À partir de cette fonction, testez votre arbre sur différents types en paramètres. Soyez sûr que votre arbre fonctionne correctement avant de passer à la partie suivante.

4 Une IA SUPérieure

On arrive à la dernière partie, la plus difficile mais aussi la plus intéressante. Cette partie s'écrit en peu de ligne de code (environ 100 lignes) mais demande d'avoir compris les algorithmes que nous allons vous présenter.

4.1 Initialisations

Regardez du côté du fichier **AI.cs**. La classe `AI` contient quatre variables :
— `max_game_analysed`, qui servira de limite pour l'algorithme de Monte-Carlo
— `max_depth`, qui servira pour la profondeur max de notre arbre
— `color`, qui va servir à savoir la couleur pour laquelle l'IA joue
— `rand`, qui va servir à générer de l'aléatoire par la suite
Complétez les deux constructeurs de la classe `AI`. N'oubliez pas d'initialiser la variable `rand`.

4.2 Minimax

Commencez par aller regarder du côté de Wikipédia ce qu'est l'algorithme minimax si vous ne le connaissez pas :
https ://fr.wikipedia.org/wiki/Algorithme_minimax

L'algorithme minimax est très utilisé pour résoudre les jeux de plateaux récursivement.

Pour reprendre son fonctionnement, on construit un arbre dont les noeuds représentent le "gain" du joueur concerné pour un plateau donné. Chaque noeud correspond à un plateau différemment rempli, et le gain est donc différent à chaque fois. Ici, notre arbre sera rempli plus tard avec les bonnes valeurs, pour l'instant nous nous contenterons d'imaginer que l'arbre est rempli.

$$minimax(p) = \begin{cases} f(p) & \text{si p est une feuille de l'arbre, avec f une fonction de gain} \\ max(minimax(c1), ..., minimax(cn)) & \text{si p est un noeud amical, avec c1, ..., cn les fils de p} \\ min(minimax(c1), ..., minimax(cn)) & \text{si p est un noeud adverse, avec c1, ..., cn les fils de p} \end{cases}$$

Lorsque l'on est sur un noeud amical, c'est-à-dire que c'est un mouvement que va effectuer l'IA, elle cherche à maximiser son gain : on calcule le maximum.

Lorsque l'on est sur un noeud adverse, c'est-à-dire que c'est un mouvement que va effectuer l'adversaire de l'IA, elle prend le minimum de gain qu'elle peut obtenir : elle prend en compte le "pire" des cas, celui où l'adversaire joue le meilleur coup.

Lorsque l'on est sur une feuille, on retourne le gain. Vous n'avez pas besoin de calculer le gain ici, vous devez juste retourner la valeur du noeud correspondant au gain, ce gain étant calculé ultérieurement.

Dans ce cas de figure, on choisit d'attribuer aux noeuds de l'arbre un type bien particulier :

```
1 Tuple<float , Tuple<int , int>>;
```

Le premier item est un flottant, contenant une valeur entre 0 et 1 qui représente le pourcentage de gain. Le deuxième item du Tuple est un Tuple contenant deux entiers : c'est la position (abscisse, ordonnée) du dernier mouvement effectué pour arriver à ce noeud.

Ce type est donc le type que nous utiliserons pour construire notre arbre, et c'est aussi le type de noeud que nous allons manipuler dans la fonction `minimax` (pour calculer le minimum ou le maximum, on pensera bien à prendre le premier item uniquement car il représente le gain).

Parce que ce type est lourd et long à réécrire, voici une nouvelle notion triviale : les alias. Vous pouvez donner un nom simple à un type compliqué grâce à la syntaxe suivante :

```
1 using typeElement = Tuple<float , Tuple<int , int>>;
```

Cette ligne apparaît donc en haut de votre fichier, elle permet d'appeler ce type compliqué par le simple alias : `typeElement`.

Voici maintenant le prototype de la fonction `minimax` :

```
1 public typeElement minimax(Tree<typeElement> tree , int depth)
```

On remarque qu'elle prend un arbre en paramètre. Encore une fois, cet arbre sera construit plus tard dans le TP, vous pouvez tester en construisant à la main des arbres en attendant si besoin. Le deuxième paramètre, `depth`, sert à indiquer la profondeur du coup réaliser. Si la profondeur est de 0, c'est que l'on cherche à savoir les coups que peut faire l'IA face à la situation courante, donc on veut maximiser notre gain. Si la profondeur est de 1, c'est que l'on est en train de prévoir les coups que peut répondre l'adversaire face aux coups que l'on peut jouer, donc on veut minimiser notre gain. On peut donc à partir de la profondeur savoir si l'on est sur un coup de l'IA ou de l'adversaire.

Servez-vous de la variable max_depth pour gérer le cas d'arrêt (`depth == max_depth`)

N'oubliez pas de retourner un noeud du type `typeElement` : notre but va être de retourner la position optimale que peut jouer l'IA, et le noeud retourné contient le gain maximum que peut faire l'IA ainsi que la position qui correspond à ce gain.

4.3 Monte-Carlo

Voilà, c'est fini pour **MiniMax** ! On passe à la fonction qui va calculer le gain à partir d'un plateau.

Commencez par aller regarder du côté de Wikipédia ce qu'est la méthode de Monte-Carlo si vous ne la connaissez pas :
https ://fr.wikipedia.org/wiki/Méthode_de_Monte-Carlo

L'explication de Wikipédia peut paraître compliquée, mais le principe est simple : Monte-Carlo génère un nombre connu de tests sur une partie donnée, cela pour obtenir un gain, calculé à partir du résultat des tests sur la partie.

Dans notre cas de figure, on veut obtenir un gain à partir d'un plateau. On va appliquer la méthode de Monte-Carlo sur un tableau donné en paramètre, c'est-à-dire que l'on va calculer au hasard, et jusqu'au bout, un nombre choisi de parties à partir d'un plateau de jeu. On pourra ainsi calculer le ratio de parties gagnées sur le nombre de parties jouées et savoir le gain que l'on peut espérer sur un plateau !

Implémentez la fonction suivante :

```
1 float monteCarlo(int[,] board, int color)
```

Cette fonction prend un tableau en paramètre, et va devoir générer des parties aléatoires en partant de ce tableau, et c'est le joueur de la couleur donnée qui joue le prochain coup.

Pour cela, vous avez la variable `rand` et la variable `max_game_analysed`.

Pour connaître la taille du plateau, on peut se servir des variables `gameInfo.row` et `gameInfo.column`

Faites attention lorsque vous manipulez l'aléatoire pour générer des coups aléatoirement : il faut éviter de se retrouver avec une boucle qui est potentiellement infinie !

Exemple :

```
1 // Cette boucle peut tourner longtemps
2 do
3 {
4     rnd = rand.Next(gameInfo.row * gameInfo.column);
5 } while board[rnd / gameInfo.column, rnd % gameInfo.row] != 0;
```

Ici la boucle peut tourner longtemps, par exemple si le plateau est grand et n'a plus qu'une case valide (non occupée). À vous de ruser pour éviter ce problème !

Faites également attention, lorsque vous manipulez les cases du tableau, à avoir fait une copie intégrale de celui-ci, pour chacune des parties générées : on ne veut pas changer les valeurs à l'intérieur du terrain pour de vrai !

4.4 Build Tree

C'est la dernière fonction du mécanisme de l'IA. Vous avez de quoi calculer un gain à partir d'un plateau de jeu, vous avez de quoi sélectionner le noeud correspondant à la position optimale à partir d'un arbre, il reste maintenant à créer cet arbre.

On veut remplir un arbre de manière en lui ajoutant autant d'enfant qu'il n'y a de possibilités. On va devoir appeler la fonction récursivement jusqu'à une profondeur maximum, représentée par `max_depth`.

Voici le prototype de la fonction récursive que vous devez implémenter :

```
1 void build_tree(Tree<typeElement> tree, int[,] board, int depth)
```

Le pseudo-code vous sera d'une grande aide pour comprendre comment marche la fonction :

```
Si on a atteint la profondeur max
    On appelle monteCarlo pour modifier la valeur du gain sur le noeud actuel
Sinon
    Pour chacune des cases VALIDES du terrain, faire :
        On met la case à la couleur qui correspond au joueur qui doit jouer
        On ajoute un enfant contenant 0 comme gain et la position de la case actuelle
        On appelle récursivement build_tree sur le fils qui vient d'être créé
        On remet la case actuelle à zéro
```

4.5 Play

La fonction principale de l'IA :

```
1 public Tuple<int, int> play(int[,] board)
```

Cette fonction enveloppe les trois autres. Elle reçoit un plateau de jeu et va renvoyer une position correspondant au coup choisi par l'IA. Elle doit, dans l'ordre :

- Créer un arbre
- Le remplir avec `build_tree`
- appeler `minimax` sur cet arbre
- renvoyer la position qui correspond au noeud retourné par `minimax`

Lorsque vous testerez votre IA, faites attention à lui donner des valeurs pas trop grandes, sinon il risque de passer plusieurs minutes à réfléchir. Les valeurs par défaut (profondeur de 2, nombre de tests 150, tableau de 6x6) sont là pour vous faire tester sans passer trop de temps à calculer.

4.6 BONI

Maintenant, c'est l'heure du duel ! Ajustez vos paramètres `max_game_analysed` et `max_depth` pour pouvoir affronter des joueurs en temps réel (évitez de faire réfléchir l'IA 5 minutes par coups, mais ne la rendez pas trop faible non plus, un juste milieu peut être 20 secondes par coup).

Ce n'est pas tout ! vous pouvez aider votre IA par exemple en lui indiquant quel coup jouer si elle joue la première (si le plateau est vide) : il est difficile dans ce cas de choisir un coup en effectuant de l'aléatoire sur un plateau vide, vous pouvez donc décider de lui faire jouer directement au milieu ou pas très loin du centre si le plateau est vide (ces coups sont en général plus efficaces et vous évitez à votre IA de réfléchir pour pas grand chose).

Si vous êtes tentés par une IA qui calcule plus vite, et que vous voulez pousser le sujet à fond, vous pouvez tenter d'optimiser votre algorithme `minimax` (ou son associé `build_tree`) : c'est lui le plus lent. Pour un plateau sur lequel il reste 10 cases non jouées, il contient, en nombre total de noeuds :
`10 + 10x9 + 10x9x8 [...] jusqu'à la profondeur max_depth`
Il existe une technique d'optimisation appelée **l'élagage alpha-bêta**. Libre à vous de tenter de l'implémenter.

Vous pouvez aussi utiliser des **threads** pour paralléliser le calcul.