TP C#12 : MyFlappyAI

Consignes de rendu

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

```
csharp-tp12-prenom.nom
    |-- README
    |-- Bases/
        |-- Bases.sln
        |-- Bases/
            |-- Tout sauf bin/ and obj/
    |-- Flappy/
        |-- Flappy.sln
        |-- Flappy/
            |-- Tout sauf bin/ and obj/
```

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

— Remplacez `prenom.nom` par votre propre login.
— Le `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 bonii que vous aurez implémentés. Un `README` vide sera considéré comme une archive invalide (malus).

Table des matières

1 Introduction

1.1 Objectifs

L'objectif de ce TP est de concevoir une intelligence artificielle primitive capable de jouer au FlappyBird. Vous pouvez trouver à travers ce cours une initiation à l'intelligence artificielle.

2 Cours

2.1 Rappels sur les délégués

Les délégués représentent des *types* de fonction. C'est-à-dire qu'on va pouvoir associer un type à une fonction. Une fonction est d'un *délégué/type* donné si et seulement si leurs prototypes correspondent.

Un exemple parlera sûrement plus que des mots :

```
1  public class Program
2  {
3      public delegate void TestFunc(string message);
4
5      public static void MyFunc1(string message)
6      {
7          Console.WriteLine("F1 : {0}", message);
8      }
9
10     class MyClass
11     {
12         public void MyFunc2(string message)
13         {
14             Console.WriteLine("F2 : {0}", message);
15         }
16     }
17
18     public static void Main(string[] args)
19     {
20         TestFunc f1 = MyFunc1;
21         MyClass my_obj = new MyClass();
22         TestFunc f2 = my_obj.MyFunc2;
23
24         f1("FOO"); // Affiche : F1 : FOO
25         f2("BAR"); // Affiche : F2 : BAR
26     }
27 }
```

Dans ce cas-ci, notre délégué est *TestFunc*. Il se comportera donc comme un type. Ensuite, on a une fonction *MyFunc1* dont le prototype est le même que celui du délégué *TestFunc*. On peut donc faire une variable de type *TestFunc* donc la valeur est notre fonction *MyFunc1*. Cela fonctionne de manière équivalente avec les méthodes (*MyFunc2*).

On pourra donc réécrire nos opérations mathématiques :

```
1  public class Program
2  {
3      public delegate int Operation(int a, int b);
4
5      public static int Add(int a, int b)
6      {
7          return a + b;
8      }
9
10     public static int Mult(int a, int b)
11     {
12         return a * b;
13     }
14
15     public static void Main(string[] args)
16     {
17         Operation first_operation = Add;
18         Operation second_operation = Mult;
19         Console.WriteLine("Result : {0}",
20             second_operation(first_operation(1, 2),
21                 first_operation(3, 4)));
22         // Affiche : 21 ((2+1) * (3+4))
23     }
24 }
```

2.1.1 En paramètre

Une utilisation des délégués est de passer des fonctions en paramètre. Imaginons qu'on veuille afficher nos opérations. Plutôt que de faire un *ShowAdd*, un autre *ShowMult*... on va utiliser les délégués comme ceci :

```
1  public class Program
2  {
3      public delegate int Operation(int a, int b);
4
5      public static int Add(int a, int b)
6      {
7          return a + b;
8      }
9
10     public static int Mult(int a, int b)
11     {
12         return a * b;
13     }
14
15     public static int Calculate(Operation operation,
16                                 int a, char symbol, int b)
17     {
18         int res = operation(a, b);
19         Console.WriteLine("{0} {1} {2} = {3}", a, symbol, b, res);
20         return res;
21     }
22
23     public static void Main(string[] args)
24     {
25         Calculate(Mult, Calculate(Add, 1, '+', 2), '*',
26             Calculate(Add, 3, '+', 4));
27         // Affiche :
28         // 1 + 2 = 3
29         // 3 + 4 = 7
30         // 3 * 7 = 21
31     }
32 }
```

2.1.2 Dans une structure de données

Un deuxième cas d'utilisation est de stocker des fonctions/méthodes dans des structures de données (comme un tableau, une liste, un dictionnaire...). Par exemple :

```
1  public class Program
2  {
3      public delegate int Operation(int a, int b);
4
5      public static int Add(int a, int b)
6      {
7          return a + b;
8      }
9
10     public static int Mult(int a, int b)
11     {
12         return a * b;
13     }
14
15     public static int Calculate(Dictionary<char, Operation> operations,
16                                 int a, char symbol, int b)
17     {
18         int res = operations[symbol](a, b);
19         Console.WriteLine("{0} {1} {2} = {3}", a, symbol, b, res);
20         return res;
21     }
22
23     public static void Main(string[] args)
24     {
25         Dictionary<char, Operation> operations
26             = new Dictionary<char, Operation>();
27         operations['+'] = Add;
28         operations['*'] = Mult;
29         Calculate(operations, Calculate(operations, 1, '+', 2), '*',
30             Calculate(operations, 3, '+', 4));
31         // Affiche :
32         // 1 + 2 = 3
33         // 3 + 4 = 7
34         // 3 * 7 = 21
35     }
36 }
```

2.1.3 Moins de verbosité

Il existe deux types de fonctions :

— Les fonctions dont le type de retour est *void*. On pourra remplacer leurs délégués à l'aide de *Action<type du premier paramètre, type du deuxième paramètre,..., type du dernier paramètre>*

— Les fonctions dont le type de retour n'est pas *void*. On pourra remplacer leurs délégués à l'aide de *Func<type du premier paramètre, type du deuxième paramètre,..., type du dernier paramètre, type de retour>*.

Exemple :

```
1  public class Program
2  {
3      public static void NoParametersAndNoReturnValue()
4      {
5          Console.WriteLine("0 V");
6      }
7      public static void OneParametersAndNoReturnValue(string s)
8      {
9          Console.WriteLine("1 V");
10     }
11     public static void ThreeParametersAndNoReturnValue(char c, string s,
12                                                         int i)
13     {
14         Console.WriteLine("3 V");
15     }
16     public static bool NoParametersAndAReturnValue()
17     {
18         Console.WriteLine("0 R");
19         return true;
20     }
21     public static int OneParametersAndAReturnValue(string s)
22     {
23         Console.WriteLine("1 R");
24         return 0;
25     }
26     public static string ThreeParametersAndAReturnValue(char c, string s,
27                                                          int i)
28     {
29         Console.WriteLine("3 R");
30         return "";
31     }
32
33     public static void Main(string[] args)
34     {
35         Action a0 = NoParametersAndNoReturnValue;
36         Action<string> a1 = OneParametersAndNoReturnValue;
37         Action<char, string, int> a3 = ThreeParametersAndNoReturnValue;
38         Func<bool> f0 = NoParametersAndAReturnValue;
39         Func<string, int> f1 = OneParametersAndAReturnValue;
40         Func<char, string, int, string> f3 = ThreeParametersAndAReturnValue;
41         a0(); // Affiche : 0 V
42         a1(""); // Affiche : 1 V
43         a3('0', "", 0); // Affiche : 3 V
44         bool r0 = f0(); // Affiche : 0 R
45         int r1 = f1(""); // Affiche : 1 R
46         string r2 = f3('0', "", 0); // Affiche : 3 R
47     }
48 }
```

2.1.4 Lambdas

Les lambdas sont des fonctions anonymes. C'est une syntaxe *(params) => res* (les parenthèses ne sont pas obligatoires s'il n'y a qu'un seul paramètre) permettant de créer des fonctions facilement et sans devoir écrire ni le prototype ni le nom (d'où anonymes). Vous ne pouvez utiliser cette syntaxe que lorsque le type attendu est connu. C'est-à-dire qu'on peut l'utiliser en paramètre ou dans l'affectation d'une variable dont le type est spécifié. Exemple :

```
1  public class Program
2  {
3      public static void Example(Func<int, int, int> func)
4      {
5          Console.WriteLine(func(1, 2));
6      }
7
8      public static void Main(string[] args)
9      {
10         // Compile :
11         Func<int, int> func1 = e => e + 1;
12         Console.WriteLine(func1(4));
13         // Compile :
14         Func<int, int> func2 = (e) => e + 1;
15         Console.WriteLine(func2(4));
16         // Compile :
17         Example((e, f) => e + f);
18         // Ne compile pas : il manque les parenthèses
19         // Example(e, f => e + f);
20         // Ne compile pas : le type est inconnu
21         // Console.WriteLine((e => e + 1)(4));
22         // Ne compile pas : le type est inconnu
23         // var func = e => e + 1;
24     }
25 }
```

2.1.5 En résumé

En résumé, une *Action* est un *type de fonction* sans valeur de retour et une *Func* est un *type de fonction* avec valeur de retour. Ce sont des délégués génériques. Entre les <>, on trouve de gauche à droite les types des paramètres. Dans le cas des *Func*, le dernier type générique est le type de retour. Dans le cas d'une fonction sans paramètres et sans valeur de retour, on a juste *Action*, sans les <>. Les lambdas sont une syntaxe permettant d'écrire des fonctions rapidement sans les nommer.

2.2 LINQ

LINQ[1], *Language-Integrated Query*, est un ensemble de méthodes et de syntaxes permettant de faire du traîtement sur des listes, tableaux... (nous ne traîterons pas les "..." dans le cadre de ce TP). Il existe deux syntaxes : les requêtes et les méthodes.

1. https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/getting-started-with-linq

Pour pouvoir l'utiliser, il faudra ajouter le *using* :

```
1  using System.Linq;
```

2.2.1 Valeur de retour

Les requêtes et méthodes de LINQ retournent *in fine* toujours un *IEnumerable<T>*. C'est-à-dire qu'on peut soit faire un *foreach* dessus ou alors le transformer en liste via *.ToList()*, en tableau via *.ToArray()* et encore d'autres. On peut aussi compter directement son nombre d'éléments via *.Count()*.

Note : ces méthodes sont ajoutées par LINQ et ne sont donc pas disponibles sans.

2.2.2 Les requêtes

Les requêtes LINQ ont une syntaxe un peu à part. La base est la suivante :

```
1  from element in list select fun(element);
```

C'est-à-dire : pour tout *element* de *list*, on ajoute *fun(element)* au résultat.

Note : *fun* est une simple fonction qui associe *element* à une valeur.

Ensuite, on peut ajouter des conditions :

```
1  from element in list where cond(element) select element
```

C'est-à-dire : pour tout *element* de *list* tel que *cond(element)*, on ajoute *element* au résultat. Mettre plusieurs *where* est déconseillé, on préférera utiliser des &&.

Note : *cond* est juste une condition booléenne comme on mettrait dans la condition d'un *if* par exemple.

Ensuite, on peut ajouter un ordre :

```
1  from element in list orderby val(element) select element
```

C'est-à-dire : pour tout *element* de *list* ordonné par *val(element)*, on ajoute *element* au résultat. Pour trier selon plusieurs ordres, on peut les séparer par des virgules.

Note : *val* est une fonction qui associe un élément à sa clé et les éléments seront traités dans l'ordre des clés.

Voici un exemple de requêtes à plusieurs conditions et ordres de tri :

```
1  from element in list
2      where cond1(element) && cond2(element)
3      orderby val1(element), val2(element)
4      select element
```

C'est-à-dire : pour tout *element* de *list* tel que *cond1(element)* *cond2(element)* ordonné d'abord par *val1(element)* et ensuite par *val2(element)*, on ajoute *element* au résultat.

Un avantage des requêtes est qu'on peut appliquer directement nos opérations sur les éléments (voir l'exemple ci-dessous).

2.2.3 Les méthodes

Les méthodes LINQ sont une syntaxe plus commune permettant de représenter des choses équivalentes. On va donc traduire les requêtes en méthodes :

```
1  from element in list select element
2  list // On voit que le select element peut être implicite
3
4  from element in list select fun(element)
5  list.Select(fun)
6
7  from element in list where cond(element) select element
8  list.Where(cond)
9
10 from element in list orderby val(element) select element
11 list.OrderBy(val)
12
13 from element in list
14     where cond1(element) && cond2(element)
15     orderby val1(element), val2(element)
16     select element
17 list.Where(e => cond1(e) && cond2(e)).OrderBy(val1).ThenBy(val2)
```

Dans les méthodes, on aura souvent tendance à, lorsque les fonctions sont petites, utiliser des lambdas.

2.2.4 Exemple

Imaginons qu'on veuille sélectionner le double de tous les éléments positifs en ordre décroissant d'une liste, on aura :

```
1  List<int> list = new List<int> { -4, -3, -2, -1, 0, 1, 2, 3, 4 };
2  // Code fait main (en utilisant la fonction sort des listes)
3  List<int> res1 = new List<int>();
4  foreach (int e in list)
5  {
6      if (e >= 0)
7      {
8          res1.Add(e * 2);
9      }
10 }
11 res1.Sort();
12 res1.Reverse();
13
14 // Requête
15 List<int> res2 = (from element in list
16                        where element >= 0
17                        orderby -element
18                        select 2 * element).ToList();
19 // Méthodes
20 List<int> res3 = list.Where(e => e >= 0)
21                       .OrderBy(e => -e)
22                       .Select(e => 2 * e).ToList();
23
24 Console.WriteLine("Counts {0} {1} {2}",
25                    res1.Count, res2.Count, res3.Count);
26 for (int i = 0; i < res1.Count; ++i)
27     Console.WriteLine("Values [{0}] {1} {2} {3}", i,
28                        res1[i], res2[i], res3[i]);
```

On voit bien que les trois codes donnent le même résultat, mais sont écrits sous trois formes différentes. Il semble aussi évident que les deux dernières formes soient plus rapides à écrire que la première.

2.2.5 Encore plus

LINQ contient encore beaucoup plus de fonctionnalités que nous n'aborderons pas dans ce TP. Mais vous pouvez vous renseigner dans la documentation [2].

2. https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/

2.3 Intelligences Artificielles

2.3.1 A*

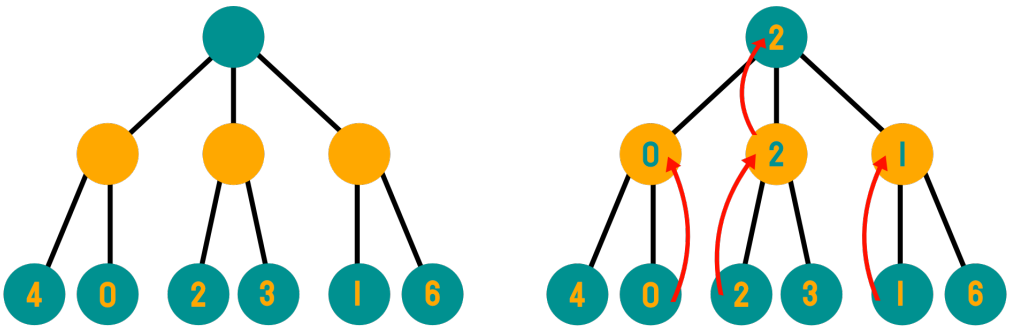
A* est un algorithme de recherche de plus court chemin. Son objectif est assez similaire à ce que vous avez vu lors du TP C# 9 avec Dijkstra. Tous deux servent à trouver le plus court chemin, avec un exception sur de A* : il ne cherche pas **LE** plus court chemin, mais **l'un DES** plus courts chemin.

	Dijkstra	A*
Objectif	Trouver le meilleur chemin dans un graphe	Trouver l'un des meilleurs chemins dans un graphe
Complexité	Passe sur tous les nœuds	Passe sur une partie des nœuds
Exigences	Un graphe suffit en entrée	Un graphe dont on peut connaître facilement la distance pour tout nœud distinct deux à deux

A* utilise la distance entre les nœuds pour déterminer les chemins se rapprochant le plus de la sortie.
A* constitue les débuts de l'intelligence artificielle sur le domaine de la planification.

2.3.2 Minimax

L'algorithme Minimax (appelé aussi MinMax) est un algorithme de décision. Son but est de prendre la meilleure décision quand il oppose un adversaire. Il peut être utilisé dans une IA qui jouerait au morpion, aux échecs ou encore au Go (avec des subtilités). Son principe prend en compte que l'adversaire tentera de prendre la meilleure décision pour le battre. Son implémentation se repose sur un arbre de décision.



Arbre de décision Minimax

Ainsi, le choix que votre IA doit faire est représenté par un noeud en bleu, et le choix "supposé" de l'adversaire en jaune.
Le noeud racine (tout en haut) correspond au choix que doit faire l'IA actuellement. Les sous-couches correspondent aux prévisions de l'IA.
Le but est alors d'aller explorer les sous-couches pour choisir la meilleure décision.
En feuilles d'arbre, on attribue un score aux noeuds grâce à une fonction d'évaluation. Plus le score est élevé, pour l'avantage de l'IA est grand. On remonte ainsi d'un cran, pour se retrouver sur la décision de l'adversaire.
On suppose alors que l'adversaire va choisir dans ces noeuds fils la décision qui vous rapporte le moins de points (Min). Ici, dans le premier noeud de l'adversaire (en jaune de gauche vers la

droite), on suppose que l'adversaire va prendre la décision d'aller sur le fils droit, car ce noeud vous rapporte le moins de point. On effectue cela ainsi de suite pour mettre à jours tous les noeuds de l'adversaire du même niveau.
On met alors notre noeud à jour en prenant la décision qui nous rapportera le plus de points. Ici, parmi [0, 2, 1], on choisit 2.
Ainsi, la meilleure décision à prendre est celle qui permet d'aller sur le noeud ayant pour nombre de points 2.

Le problème principal du Minimax est qu'il est très coûteux, et explorer toutes les possibilités jusqu'à la fin peut vite devenir impossible. Dans le cas d'un morpion ce n'est pas très gênant car les possibilités ne sont pas immenses, mais dans le cadre d'un jeu d'échecs, les possibilités sont beaucoup plus grandes, et encore plus dans le jeu du Go, où la taille de l'arbre de décision est de 10^{600}, soit bien plus que le nombre d'atomes dans l'univers (estimé entre 10^{80} et 10^{85}). Il convient donc de trouver des combines pour palier à ces problèmes.
La première solution est de limiter la profondeur de l'arbre. Ainsi on prévoit un certain nombre de coups d'avance, puis on évalue les coups les plus avancés avec une bonne fonction d'évaluation pour donner un score à à cette prévision.
La deuxième solution est d'ignorer des sous-arbres qui pourraient être jugés inintéressants. Il est possible pour cela d'utiliser un algorithme d'élagage alpha-bêta, ou encore des réseaux de neurones. [3]

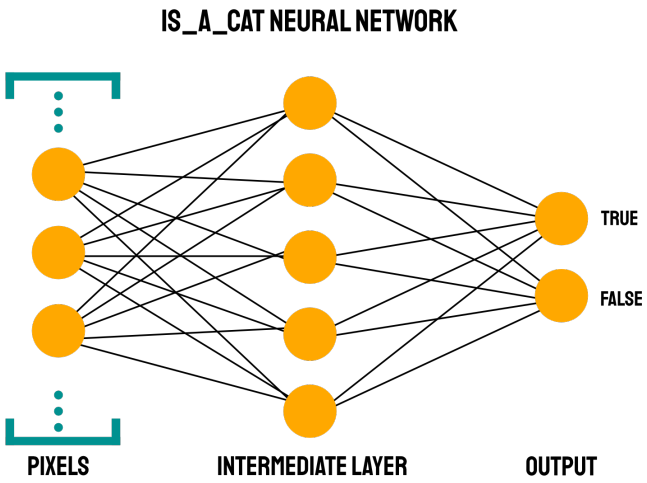
2.3.3 Réseau de neurones

Les réseaux de neurones constituent la base du fonctionnement de l'intelligence artificielle. Le but d'un réseau de neurones est d'approximer au mieux une application.

$$f : \begin{array}{c} E \to F \\ u \mapsto f(u) \end{array}$$

Le but est ainsi de trouver (au mieux) la relation qui permet de passer de l'ensemble E vers F. Prenons plus concrètement un algorithme qui se charge de reconnaître la présence d'un chat dans une image.
On a donc en entrée de l'algorithme une image, qui, à partir de cette image, permet de renvoyer soit *Vrai* si l'image contient un chat, soit *Faux* si l'image n'en contient pas.
E correspond donc à l'ensemble des pixels d'une image, et F à $\{Vrai, Faux\}$.
De plus, on sait qu'il existe une application *(≈ fonction)* f qui permet de reconnaître un chat dans une image (la preuve, on sait distinguer une image qui contient un chat d'une autre qui n'en contient pas).
Le but est donc de créer un réseau de neurones qui a pour but d'approximer l'application f "Reconnaître un chat dans une image". Pour parvenir à cette fin, nous allons utiliser l'apprentissage sur un réseau de neurones.

3. `https://fr.wikipedia.org/wiki/%C3%89lagage_alpha-b%C3%AAta`



Ainsi à partir d'images que l'on sait à l'avance si elles contiennent un chat ou non, on demande au réseau de neurones d'analyser l'image et de déduire si oui ou non elle contient un chat. Si le résultat est faux, alors on *"blâme"* le réseau de neurones, ou alors on le *"récompense"* s'il répond juste. [4]
Ainsi plus le réseau de neurones est entrainé avec des images, plus il sera apte à savoir si une image qu'il n'a jamais analysée contient un chat. Pour les explications techniques du fonctionnement d'un réseau de neurones, je vous réfère à ce lien : `https://youtu.be/aircAruvnKk`

4. En réalité, il n'est pas question de blâmer ni de récompenser, il s'agit plutôt de calculer la différence entre la réponse donnée et la réponse attendue et de la faire rétropropager à travers les couches. Si cela vous intéresse, je vous laisse chercher cela sur Google. Et pour les plus motivés : `http://neuralnetworksanddeeplearning.com/index.html` (Ça pourrait vous servir pour l'année prochaine, comme par exemple, le proj... O_O

3 Mise en bouche

3.1 Introduction

Comme petite mise en bouche, voici quelques petits exercices sur les délégués et sur LINQ qui vous rappelleront peut-être un peu le CAML. Le premier exercice a pour but de vous permettre de tester simplement votre code, les trois suivants de vous familiariser avec LINQ et les deux derniers de vous le faire utiliser réellement. Cela peut paraître long, mais ce sont en réalité majoritairement des petites fonctions.

Les fonctions sont à écrire dans la class Program de la solution Bases.

3.1.1 Le début

Cette première fonction a pour but de faire une base pour tester votre code.

3.1.1.1 Apply

```
1  public static void Apply<T>(List<T> list, Action<T> fun);
2  // Apply(new List<int> { 1, 2, 3 }, Console.WriteLine);
3  // Affiche :
4  // 1
5  // 2
6  // 3
7  // Apply(new List<int> {}, Console.WriteLine);
8  // N'affiche rien
```

Cette fonction applique *fun* à chaque élément de la liste.

3.1.2 LINQ - 1

Ces trois exercices ont pour but de vous familiariser avec les requêtes et les méthodes de LINQ. Vous devez donc écrire trois fonctions par exercice :

1. La première avec le suffixe *None*, pour lequel vous ne pouvez pas utiliser de fonction extérieure.
2. La seconde avec le suffixe *Method*, pour lequel vous devez utiliser les méthodes LINQ.
3. La troisième avec le suffixe *Query*, pour lequel vous devez utiliser une requête LINQ (vous pouvez tout de même utiliser *.ToList()*).

Note : les ? dans les exemples sont à remplacer par un des suffixes.

3.1.2.1 Map

```
1  public static List<T2> MapNone<T1, T2>(List<T1> list, Func<T1, T2> fun);
2  public static List<T2> MapMethod<T1, T2>(List<T1> list, Func<T1, T2> fun);
3  public static List<T2> MapQuery<T1, T2>(List<T1> list, Func<T1, T2> fun);
4
5  // Apply(Map?(new List<int> { 1, 2, 3, 4 }, e => e * 2),
6  //       Console.WriteLine);
7  // Affiche :
8  // 2
9  // 4
10 // 6
11 // 8
```

Cette fonction crée une nouvelle liste contenant l'application de la fonction *fun* à chaque élément de la liste de départ.

3.1.2.2 Filter

```
1  public static List<T> FilterNone<T>(List<T> list, Func<T, bool> fun);
2  public static List<T> FilterMethod<T>(List<T> list, Func<T, bool> fun);
3  public static List<T> FilterQuery<T>(List<T> list, Func<T, bool> fun);
4
5  // Apply(Filter?(new List<int> { 1, 2, 3, 4 }, e => e \% 2 == 0),
6  //       Console.WriteLine);
7  // Affiche :
8  // 2
9  // 4
```

Cette fonction crée une nouvelle liste contenant tous les éléments validant la fonction *fun*.

3.1.2.3 Sort

```
1  public static List<T> SortNone<T, S>(List<T> list, Func<T, S> fun);
2  public static List<T> SortMethod<T, S>(List<T> list, Func<T, S> fun);
3  public static List<T> SortQuery<T, S>(List<T> list, Func<T, S> fun);
4
5  // Apply(Sort?(new List<int> { 1, 2, 3, 4 }, e => -e),
6  //       Console.WriteLine);
7  // Affiche :
8  // 4
9  // 3
10 // 2
11 // 1
```

Cette fonction crée une nouvelle liste contenant tous les éléments triés dans l'ordre croissant des applications de la fonction *fun*.

Explication de l'exemple : par la fonction *e => -e*, on associe les clés { -1, -2, -3, -4 } aux valeurs { 1, 2, 3, 4 }. En triant les clés et gardant l'association, on obtient l'exemple ci-dessus.

Pour *SortNone*, vous pouvez utiliser :

```
1  Comparer<T>.Default.Compare(e1, e2);
```

Dont sa valeur de retour est strictement négative si *e1 < e2*, nulle si *e1 == e2* et strictement positive si *e1 > e2*.

L'algorithme de tri est laissé au choix entre *Bubble Sort*, *Insertion Sort* et *Selection Sort*. *Heap Sort*, *Quick Sort* et *Merge Sort* peuvent aussi être utilisés, mais sont plus compliqués à implémenter. Les performances n'étant pas notées, nous vous recommandons de choisir un des trois premiers. Vous pouvez visualiser tous les tris ici [5].

L'ordre d'éléments égaux n'est pas important ici.

3.1.3 LINQ - 2

Les fonctions suivantes ne consistent plus qu'à utiliser LINQ. Vous pouvez choisir si vous préférez utiliser les requêtes ou les méthodes. Vous pouvez utiliser des fonctions tierces mais pas de boucles (ni fonctions récursives...).

5. https://visualgo.net/en/sorting

3.1.3.1 LINQ1

```
1 public static List<int> LINQ1(List<int> list);
2 // Apply(LINQ1(new List<int> { -4, -3, -2, -1, 0, 1, 2, 3, 4 }),
3 //       Console.WriteLine);
4 // Affiche :
5 // 4
6 // 2
7 // 0
8 // 2
9 // 4
```

Cette fonction affiche la valeur absolue des éléments pairs triés en ordre décroissant.

3.1.3.2 LINQ2

```
1 public static List<string> LINQ2(List<string> list);
2 // Apply(LINQ2(new List<string> { "FOO", "BAR", "FOOBAR", "BARFOO", "BAZ" }),
3 //       Console.WriteLine);
4 // Affiche :
5 // BARBAR
6 // FOOFOO
7 // BARFOOBARFOO
8 // FOOBARFOOBAR
```

Cette fonction affiche la répétition de chaque élément soit étant "BAR" soit contenant "FOO" trié d'abord en ordre croissant de longueur et ensuite en ordre alphabétique.

4 Flappy Bird

4.1 Introduction

Vous voici enfin à votre intelligence artificielle.

4.2 Consigne

Vous devez créer une intelligence artificielle dans la classe *MaximaxController*. Le principe que vous devez utiliser est celui du maximax. C'est un algorithme très proche du minimax. Cependant, le minimax est utilisé pour prendre une décision avec des opposants, alors que le maximax fonctionne aussi sans opposant.

4.2.1 Maximax

Le principe est assez simple à comprendre : chaque fois qu'on doit prendre une décision, on associe un score à chaque série d'actions possibles. (Dans notre cas, on va associer 1 si on arrive à passer le prochain tuyau et -1 si on entre en collision avec). En général, on associera un nombre négatif à une série qui finit mal et un score positif à une série qui finit bien. Ensuite, on prend la série qui a le score maximal et on retourne sa première action.

4.2.2 Exemple

Une explication sur une version simplifiée :

4.2.2.1 Conditions initiales Considérons une grille de 5 de hauteur (et on considère la largeur "infinie") :

- Notre oiseau est en (x=0, y=2)
- Le prochain tuyau est en (x=3) et a dans sa partie haute les cases {(x=3, y=0), (x=3, y=1)} et dans sa partie basse la case (x=3, y=4)

4.2.2.2 Algorithme On doit faire notre premier choix, appliquons l'algorithme (faites attention à l'indentation, ce sont des appels récursifs) :

- On considère que l'oiseau saute, il se retrouve en (x=1, y=1) {S}.
  - On considère que l'oiseau saute, il se retrouve en (x=2, y=0) {S, S}.
    - On considère que l'oiseau saute, il se retrouve en (x=3, y=0) {S, S, S}. Collision avec le tuyau, le score de cette série est de -1.
    - On considère que l'oiseau tombe, il se retrouve en (x=3, y=1) {S, S, T}. Collision avec le tuyau, le score de cette série est de -1.
  - On considère que l'oiseau tombe, il se retrouve en (x=2, y=2) {S, T}.
    - On considère que l'oiseau saute, il se retrouve en (x=3, y=1) {S, T, S}. Collision avec le tuyau, le score de cette série est de -1.
    - On considère que l'oiseau tombe, il se retrouve en (x=3, y=3) {S, T, T}.
      - On considère que l'oiseau saute, il se retrouve en (x=4, y=2) {S, T, T, S}. On a passé le tuyau, le score de cette série est de 1.
      - On considère que l'oiseau tombe, il se retrouve en (x=4, y=4) {S, T, T, T}. On a passé le tuyau, le score de cette série est de 1.
- On considère que l'oiseau tombe, il se retrouve en (x=1, y=3) {T}.
  - On considère que l'oiseau saute, il se retrouve en (x=2, y=2) {T, S}.
    - On considère que l'oiseau saute, il se retrouve en (x=3, y=1) {T, S, S}. Collision avec le tuyau, le score de cette série est de -1.
    - On considère que l'oiseau tombe, il se retrouve en (x=3, y=3) {T, S, T}.
      - On considère que l'oiseau saute, il se retrouve en (x=4, y=2) {T, S, T, S}. On a passé le tuyau, le score de cette série est de 1.
      - On considère que l'oiseau tombe, il se retrouve en (x=4, y=4) {T, S, T, T}. On a passé le tuyau, le score de cette série est de 1.
  - On considère que l'oiseau tombe, il se retrouve en (x=2, y=4) {T, T}.
    - On considère que l'oiseau saute, il se retrouve en (x=3, y=3) {T, T, S}.
      - On considère que l'oiseau saute, il se retrouve en (x=4, y=2) {T, T, S, S}. On a passé le tuyau, le score de cette série est de 1.
      - On considère que l'oiseau tombe, il se retrouve en (x=4, y=4) {T, T, S, T}. On a passé le tuyau, le score de cette série est de 1.
    - On considère que l'oiseau tombe, il se retrouve en (x=3, y=4) {T, T, T}. Collision avec le tuyau, le score de cette série est de -1.

4.2.2.3 Scores On a donc les séries suivantes, associées aux scores suivants :

- {S, S, S} : -1
- {S, S, T} : -1
- {S, T, S} : -1
- {S, T, T, S} : 1
- {S, T, T, T} : 1

— {T, S, S} : -1
— {T, S, T, S} : 1
— {T, S, T, T} : 1
— {T, T, S, S} : 1
— {T, T, S, T} : 1
— {T, T, T} : -1

4.2.2.4 Choix de l'action Finalement, le score maximal est de 1. On a le choix entre les séries :

— {S, T, T, S}
— {S, T, T, T}
— {T, S, T, S}
— {T, S, T, T}
— {T, T, S, S}
— {T, T, S, T}

En regardant la première action, on voit qu'on a le choix. Vous pouvez choisir arbitrairement. (Une version qui choisit l'action qui apparaît le plus est en bonus.) On considérera qu'on choisit de tomber dans cet exemple.

4.2.2.5 Suite Maintenant, le jeu avance, on se retrouve en (x=1, y=3) et on recommence tout l'algorithme.

4.2.3 Implémentation

Il vous est demandé de coder votre intelligence artificielle dans la classe *MaximaxController*.

4.3 Le jeu

Afin d'implémenter votre intelligence artificielle et vos bonii, vous aurez besoin de comprendre le fonctionnement global du jeu. Vous pouvez commencer par y jouer, la touche pour le joueur étant la barre espace. Ensuite, n'hésitez pas à lire le code et les commentaires. Si vous ne comprenez pas une partie, n'hésitez pas à poser une question à vos ACDCs.

4.3.1 Game

Cette classe contient le jeu. Vous n'aurez pas nécessairement besoin d'interagir avec directement.

4.3.2 Bird

Cette classe représente un oiseau. Vous aurez possiblement besoin des méthodes :
— *Update*
— *DeepCopy*

4.3.3 Pipe

Cette classe représente un tuyau. Vous aurez possiblement besoin des méthodes :
— *Collides*
— *DeepCopy*

4.3.4 Drawer & ConsoleDrawer

La classe abstraite *Drawer* représente une sortie (en général) et la classe *ConsoleDrawer* la sortie console. Vous n'aurez pas nécessairement besoin d'interagir avec directement.

4.3.5 Manager & KeyboardManager

La classe abstraite *Manager* représente une entrée (en général) et la classe *KeyboardManager* l'entrée clavier. Vous n'aurez pas nécessairement besoin d'interagir avec directement.

4.3.6 Controller, MaximaxController, BestController & KeyboardController

La classe abstraite *Controller* représente un preneur de décision (en général) et la classe *KeyboardController* la prise de décision liée à l'entrée clavier. Vous n'aurez pas nécessairement besoin d'interagir avec directement. La classe *MaximaxController* contiendra votre intelligence artificielle. Il y a plus d'informations sur le *BestController* à la fin du sujet. Il n'est utile que si vous faites des bonii.

4.3.7 Deque

La classe *Deque* contient une implémentation d'une file à deux extrémités. Vous aurez surêment besoin d'utiliser plusieurs de ses méthodes. Pensez bien à lire les commentaires associés. Si vous avez une question, n'hésitez pas à la poser à un de vos ACDCs.

4.4 Bonii

On va pouvoir identifier trois sortes de bonii :

4.4.1 L'amélioration du maximax

Vous pouvez améliorer votre maximax de plusieurs façons différentes :
- Modifier la fonction qui associe le score à chaque série (en regardant par exemple la différence entre la hauteur de l'oiseau et le milieu de l'espace libre du prochain tuyau).
- Ne pas se concentrer uniquement sur le prochain tuyau, mais aussi sur les suivants.
- ...

4.4.2 Une autre intelligence

Vous pouvez aussi décider de créer plusieurs intelligences artificielles différentes, par exemple pour tester plusieurs versions de votre algorithme...
- Une première version de votre maximax avant amélioration...
- Pour les plus courageux, vous pouvez utiliser un algorithme différent (par exemple ceux présentés en haut). Mais n'oubliez pas de rendre votre maximax, sans quoi vous ne pourrez être noté !

Pour les bonii de ce genre, vous devez donc créer de nouveaux *Controller*. Pensez à lire la partie *Scores* ci-dessous si vous voulez être classés en fonction de votre meilleure IA.

4.4.3 Modifier le jeu

Vous pouvez aussi modifier le jeu, ajouter des fonctionnalités... **Faites juste attention, votre intelligence artificielle doit savoir tourner sur le jeu sans modification.**

4.4.4 Autres

Vous pouvez aussi créer d'autres bonii auquels nous n'aurions pas pensé, n'oubliez pas de les mettre dans votre README.

4.5 Scores

Pour rendre ce TP plus interactif, un classement a été mis en place sur l'intranet ACDC. Ce classement ne représente pas la note finale et est à but ludique uniquement.

4.5.1 Une autre IA

Le *Controller* qui est appelé est le *BestController*. Cette classe étend par défaut de votre *MaximaxController* et est donc prête pour l'emploi. Cependant, si vous décidez de faire une autre IA, vous pouvez modifier directement *BestController* ou tout étendre *BestController* de votre nouveau *Controller*.

I see no point in coding if it can't be beautiful.