TD 4 : Traits

Université Paris Diderot – Master 2 Informatique

21 septembre 2015

Ces travaux dirigés servent d'introduction aux "traits", qui ont été brièvement mentionnés en cours et seront présentés en détails lors du cours 5. Pour le moment, figurez-vous qu'un trait est une classe dont la classe mère peut être définie plus tard.

Ce sujet s'intéresse au développement d'une bibliothèque de résolution de problèmes par exploration d'un domaine de recherche modélisé par un graphe implicite. Ainsi, en SCALA, un problème est représenté par une instance de la classe abstraite `Problem` suivante :

```
abstract class Problem {

  trait Checkable { def ok :Boolean }
  trait Movable { def moves :Set[Move] }
  type State <:Checkable with Movable

  trait Mover { def moves (s:State) :State }
  type Move <:Mover

  abstract class Path { def add (m :Move) :Path }
  val emptyPath :Path

  type Answer = (State, Path)
}
```

Dans cette définition, on distingue 4 types associés à la classe `Problem` :

1. `State` : Il s'agit d'un état `s` du problème. Si cet état est tel que « `s.ok == true` » alors `s` est une solution admissible. L'état `s` peut aussi être un état intermédiaire du problème (le problème est en cours de résolution). Dans ce cas, on peut passer à une autre étape de résolution en utilisant la méthode `moves` qui fournit l'ensemble des étapes de résolution applicables depuis `s`. Si cet ensemble est vide et que `s` n'est pas une solution admissible alors `s` est une impasse et il faut revenir sur des étapes de résolution choisies précédemment.
2. `Move` : Une valeur de type `Move` représente une étape de résolution. On peut l'appliquer à un état du problème pour passer à un autre état.
3. `Path` : Un chemin dénote une séquence d'étape de résolution menant de l'état initial du problème à un état donné. On construit un chemin incrémentalement en partant du chemin vide et en utilisant la méthode `add` qui construit un nouveau chemin auquel on a rajouté une nouvelle étape `m`.
4. `Answer` : Une solution à un problème est un couple d'une solution admissible et d'un chemin qui y mène depuis l'état initial du problème.

L'objectif de notre bibliothèque est de fournir des instances concrètes de la classe abstraite suivante :

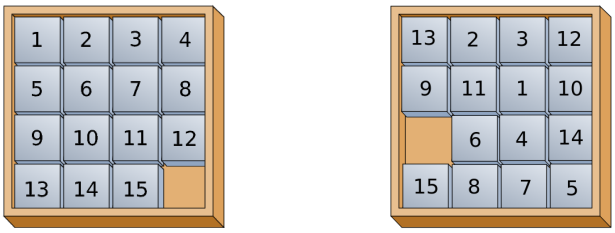

FIGURE 1 – À gauche la configuration finale désirée, à droite une configuration mélangée.

```
abstract class SolvedProblem extends Problem {
  class NoAnswerFound extends Throwable
  def apply (s :State) :Answer
}
```

Plus précisément, nous allons successivement *dériver* plusieurs algorithmes de recherche : la recherche en profondeur, la recherche en largeur, la recherche en profondeur avec heuristique, la recherche en largeur avec heuristique et l'algorithme $A^\star$. Nous appliquerons enfin cette bibliothèque à la résolution du jeu du taquin.

Exercice 1 (Taquin) *Le jeu du taquin est une grille carrée dont les cases sont numérotées. Une de ses cases est manquante, ce qui permet de permuter de proche en proche ce trou avec une case adjacente. Une configuration initiale du jeu est une permutation quelconque des cases. L'objectif du jeu est de replacer la grille dans un état où les numéros des cases sont ordonnées de gauche à droite et du haut vers le bas, la case vide se trouvant tout en bas à droite. Les figures suivantes illustrent ces deux types de configurations. Pour plus détails sur ce jeu, veuillez consulter :*

`http://fr.wikipedia.org/wiki/Taquin`

1. Définissez une classe `Puzzle` qui implémente la classe `Problem` en modélisant le jeu du taquin. □

Exercice 2 *(Mémoire) Un algorithme de recherche a besoin d'une mémoire lui indiquant si il a déjà croisé un certain état du problème. On se donne donc un trait représentant la propriété du problème :*

```
trait Memory extends Problem {
  def worth_it (s :State) :Option[State]
  def will_worth_it (s :State) :Boolean
}
```

Un problème qui a une mémoire donne la capacité au processus de résolution de se souvenir si l'exploration d'un état du problème est utile. En particulier, explorer de nouveau un état déjà rencontré est inutile. La méthode « `will_worth_it (s)` » renvoie **`false`** *lorsque `s` est inutile et* **`true`** *dans le cas contraire. La méthode « `worth_it (s)` » renvoie `None` lorsque `s` est inutile et `Some (s)` dans le cas contraire. De plus, elle mémorise dans une table le fait que `s` a maintenant été rencontré.*

1. Lisez la documentation de la classe `HashMap` de la bibliothèque standard de SCALA. *Quelles méthodes sont nécessaires au type `State` pour instancier la classe `HashMap[State, State]` ?*
2. Implémenter une classe fille `MPuzzle` qui étend la classe `Puzzle` avec le trait `Memory`. □

Exercice 3 (Algorithme de recherche) *Un algorithme de recherche est un parcours du graphe formé par le domaine des états du problème. Il maintient une structure de donnée `WaitingLine` qui stocke l'ensemble des états restant à*

explorer. La mémoire permet d'éviter de rajouter des états inutiles dans cet ensemble. L'algorithme s'exprime abstraitement de la façon suivante :

```
Recherche () =
  SI WaitingLine est vide ALORS il n'y a pas de solution
  SINON
    Extrait N le premier état de WaitingLine
    SI l'exploration de N est inutile ALORS Appelle récursif à Recherche
    SINON SI N est une solution admissible ALORS Retourne N et son chemin.
          SINON
             1. Rajoute dans WaitingLine tous les états accessibles en une étape depuis N
             2. Appelle récursivement Recherche.
```

On étend donc la classe `SolvedProblem` à l'aide du trait `Memory` en utilisant cette algorithme. Le squelette du code est :

```
abstract class ExploringSearch extends SolvedProblem with Memory {
  type WaitingLine[T] <:Seq[T] with Growable[T]
  def pop [T] (s :WaitingLine[T]) :WaitingLine[T]
  var next :WaitingLine[(State, Path)]

  final def run :(State, Path) = // À compléter

  def apply (s :State) :(State, Path) = {
    next += ((s, emptyPath))
    run
  }
}
```

1. En vous référant à la documentation de la bibliothèque standard de SCALA*, quelles sont les méthodes que l'on peut attendre du type `WaitingLine[T]` ?*
2. Complétez la méthode `run`. □

Exercice 4 (Un première instanciation concrète) *Pour implémenter une exploration en profondeur des solutions, il suffit d'utiliser une structure de données de pile pour représenter les éléments restants à explorer :*

```
trait DFS {
  type Answer
  type WaitingLine[T] = MutableList[T]
  def pop [T] (s:WaitingLine[T]) = s.tail
  var next :MutableList [Answer] = new MutableList
}
```

Ensuite, on peut se lancer de la résolution d'une instance (facile) du problème :

```
object Puzzle1 {
  def main (args :Array[String]) = {
    val grid = Array (
          Array (1, -1, 3, 4),
          Array (5, 2, 6, 8),
          Array (9, 10, 7, 12),
          Array (13, 14, 11, 15)
    )
    val solver = new ExploringSearch with DFS with MPuzzle
    val (final_state, path) = solver (new solver.State (grid, 0, 1))
    println (path)
}}
```

Si vous lancez l'exécution du programme, vous devez obtenir une erreur à l'exécution nommée `ClassCastException`*. En effet, sauf si vous êtes malins, vous créez une instance de* `State` *dans la méthode* `moves` *de la classe* `Move` *du trait* `Puzzle`*. Or, cette instance utilise le type* `State` *de* `Puzzle` *et non pas celui de* `MPuzzle`*. On aimerait que* `State` ***soit un type virtuel*** *de* `Puzzle`*.*

1. Trouvez une solution à ce problème. Indice : utilisez une déclaration de type et non une définition de type pour introduire le type `State`*.*

2. Dessinez la chaîne d'héritage correspondant à la composition mixin finale « `new DFS with MPuzzle` »

3. Définissez le trait correspondant à un parcours en largeur en utilisant non pas une liste mais une file pour implémenter l'ensemble des états restants à traiter.

4. Essayez maintenant la grille :

```
val grid = Array (
      Array (-1, 1, 2, 3),
      Array (4, 5, 6, 7),
      Array (8, 9, 10, 11),
      Array (12, 13, 15, 14))
```

Votre programme ne termine pas sur cette entrée et pourtant, il existe une solution à cette grille (voir la page WIKIPEDIA *référencée plus haut). Quel est le problème d'après vous ?* □

Exercice 5 (Utilisation d'une heuristique) *Une heuristique est une évaluation de la pertinence d'un état du problème pour atteindre la solution. Dans le cas du jeu du taquin, une bonne heuristique est la somme des distances des chiffres à leur position dans la configuration finale. (La distance choisie ici est la somme des valeurs absolues de différences des coordonnées.)*

Une heuristique peut servir à ***diriger*** *une recherche, par exemple, on peut s'en servir pour trier les successeurs d'un état du plus pertinent (à explorer en premier) au moins pertinent (sans grand intérêt a priori).*

1. Définir un trait correspondant à la capacité d'un état à calculer son image par la fonction d'évaluation heuristique.

2. Étendre l'algorithme de recherche générique en un algorithme de recherche dirigé par l'heuristique.

3. Étendre le trait `MPuzzle` *pour y inclure le calcul de l'heuristique décrite plus haut.* □

Même avec cette heuristique, les algorithmes de parcours ne sont pas efficaces. Il faut utiliser l'algorithme $A^\star$ qui consiste à utiliser une `file de priorité` qui est capable à tout moment de proposer l'état le plus pertinent vis-à-vis de l'heuristique.

Exercice 6 ($A^\star$) *1. Définissez un nouveau type d'algorithme de recherche avec heuristique qui utilise une file de priorité de la bibliothèque standard pour représenter l'ensemble des états restants à traiter.*

2. On peut modifier cet algorithme pour qu'il calcule aussi la solution la plus courte. Il suffit d'effectuer un simple raffinement de la classe `Memory`*. Saurez-vous le trouver ?* □