TP Exploration de château, lambda-expression, et fonctions d'ordre supérieur

1. Introduction

Ici, on repart de l'activité *Un château pas très fort* de Marie Duflot-Kremer sur la vérification de formules en logique temporelle en utilisant du *model-checking*.

On cherche à vérifier des propriétés sur les trajets possibles dans le château dont le plan est représenté sur la figure suivante :

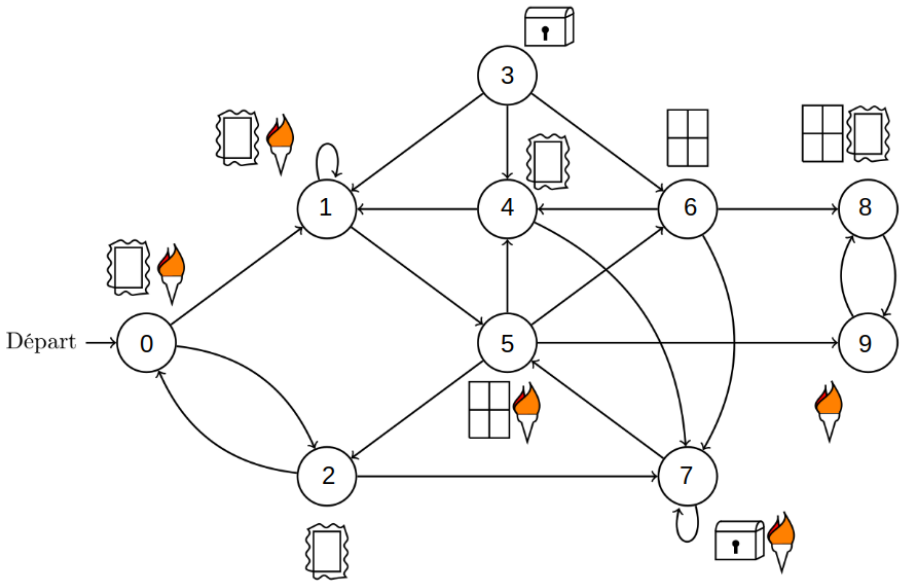

Figure 1: plan du château

On peut représenter les déplacements possibles dans le château grâce à une fonction renvoyant, pour une salle donnée la liste des salles accessibles. Voici un telle fonction définie en Haskell :

```
avancer 0 = [1, 2]
avancer 1 = [1, 5]
avancer 2 = [0, 7]
avancer 3 = [1, 4, 6]
avancer 4 = [1, 7]
avancer 5 = [2, 4, 6, 9]
avancer 6 = [4, 7, 8]
avancer 7 = [5, 7]
avancer 8 = [9]
avancer 9 = [8]
```

1. Définir la fonction `avancer(int) -> [int]` en Python

Exemple d'utilisation

```
>>> avancer(2)
[0, 7]
```

Pour représenter le fait qu'un tableau, une torche, etc. est présent ou non dans une salle, on peut définir des fonctions `tableau`, `torche`, etc. prenant un numéro de salle en paramètre et renvoyant un booléen. Voici ces fonctions en Haskell :

```
torche salle = salle `elem` [9, 7, 5, 1, 0]
coffre salle = salle `elem` [3, 7]
fenetre salle = salle `elem` [5, 6, 8]
tableau salle = salle `elem` [0, 1, 2, 8]
```

2. Définir ces fonctions en Python sous la forme de lambda-expressions.

Exemples d'utilisation

```
>>> torche(7)
True
>>> coffre(6)
False
```

Par la suite, on pourra s'intéresser au fait qu'une salle est éclairée ou non. Une salle est éclairée si elle contient une torche ou bien si elle possède une fenêtre. En Haskell, on peut écrire cette fonction ainsi :

```
lumiere salle = (torche salle) || (fenetre salle)
```

3. Définir en Python la fonction `lumiere` permettant de savoir si une salle est éclairée en utilisant une lambda-expression.

Exemples d'utilisation

```
>>> lumiere(1)
True
>>> lumiere(4)
False
>>> lumiere(6)
True
```

2. Partir en exploration

On souhaite maintenant modéliser les salles qu'il est possible d'atteindre au bout d'un nombre donné de déplacements dans le château. Pour se faire, on va utiliser une fonction qui permet, à partir d'une liste de salles, d'obtenir la listes des salles accessibles en 1 déplacement depuis une salle quelconque de la liste de départ. En Haskell, cette fonction s'écrit ainsi :

```
step positions = positions >>= avancer
```

En Python, c'est un petit peu plus compliqué :

```
from functools import reduce

def step(positions):
  return reduce(lambda l1, l2: l1 + l2, map(avancer, positions))
```

Exemple d'utilisation

```
>>> step([0, 1])
[1, 2, 1, 5]
```

Dans notre cas, on souhaite savoir où on pourra être au bout de n pas. Pour se faire, on va utiliser la récursivité. En Haskell, la fonction `distance` va s'écrire ainsi :

```
distance 0 positions = positions
distance n positions = distance (n-1) (step positions)
```

1. Écrire la fonction `distance` en Python, de façon récursive

Exemple d'utilisation

```
>>> distance(2, [0])
[1, 5, 0, 7]
>>> distance(3, [8, 9])
[9, 8]
```

On aimerait maintenant savoir s'il est possible, en exactement n déplacements, d'atteindre une salle avec une propriété donnée. Par exemple : depuis la salle 0, puis-je arriver, en exactement 2 déplacements, dans une salle contenant un coffre.

Pour le garantir, on peut calculer les salles accessibles en exactement 2 déplacement, n garder que celles contenant un coffre, et vérifier s'il la liste obtenue n'est pas vide.

En Haskell, on peut écrire cela ainsi:

```
accessible_exact propriete debut dist =
          length (filter propriete (distance dist debut)) > 0
```

On peut alors appeler cette fonction ainsi :

```
Prelude> accessible_exact coffre [0] 2
true
```

2. Écrire la fonction `accessible_exact` en Python, en utilisant la fonction `filter`.

Exemple d'utilisation

```
>>> accessible_exact(coffre, [0], 2)
True
```

3. En passant en paramètre une lambda-expression, utiliser la fonction `accessible_exact` pour vérifier s'il est possible, en 3 déplacements exactement, d'atteindre une salle avec fenêtre mais sans torche.

3. Raisonner sur les chemins

3.1 Calculs de base

Plutôt que de s'intéresser uniquement aux états d'arrivée, on pourrait s'intéresser aux chemins d'une longueur donnée. Cela permettrait de répondre à des questions comme : - y a-t-il un chemin de longueur 4 tout le temps éclairé ? - est-ce-que tous les chemins de longueur 2 me garantissent de toujours voir des tableaux ?

Un chemin est représenté par une liste de salles. La fonction `cheminer` est une fonction qui, à partir d'un chemin c de longueur n renvoie la liste des chemins de longueur $n+1$ commençant par le chemin c. En Haskell, cette fonction s'écrit ainsi :

```
poursuivre debut [] = []
poursuivre debut liste =
        (debut ++ [head liste]):(poursuivre debut (tail liste))

cheminer debut = poursuivre debut (avancer (last debut))
```

1. Écrire la fonction `cheminer(liste[int]) -> list[list[int]]` en Python

Exemple d'utilisation

```
>>> cheminer([0, 1, 3])
[[0, 1, 3, 1], [0, 1, 3, 4], [0, 1, 3, 6]]
```

De façon analogue à ce qui avait été fait dans la partie 2 pour la fonction `step`, on peut écrire la fonction `cheminer_multiple` qui permet de poursuivre une liste de chemins d'une étape. En Haskell, cela peut s'écrire ainsi :

```
cheminer_multiple chemins = cheminer >>= chemins
```

On peut alors calculer ainsi tous les chemins de longueur 2 à partir de l'entrée :

```
Prelude> cheminer_multiple [[0]]
[[0,1],[0,2]]
```

2. Écrire la fonction `cheminer_multiple` en Python

Exemple d'utilisation

```
>>> cheminer_multiple([[0, 1], [0, 2]])
[[0, 1, 1], [0, 1, 5], [0, 2, 0], [0, 2, 7]]
```

On peut maintenant écrire la fonction `prolongement` qui, à partir d'une liste de chemins, calculs toutes les prolongations de n pas supplémentaires. Cette fonction s'écrit ainsi en Hakell :

```
prolongement 0 chemins = chemins
prolongement n chemins =
                    prolongement (n-1) (cheminer_multiple chemins)
```

On peut alors obtenir ainsi tous les chemins de longueur 3 à partir du début :

```
Prelude> prolongement 2 [[0]]
[[0,1,1],[0,1,5],[0,2,0],[0,2,7]]
```

3. Écrire la fonction `prolongement(n: int, chemins: list[list[int]]) -> list[list[int]]` en Python.

Exemple d'utilisation

```
>>> prolongement(2, [[0, 1], [0, 2]])
[[0, 1, 1, 1], [0, 1, 1, 5], [0, 1, 5, 2] [0, 1, 5, 4], [0, 1, 5, 6],
[0, 1, 5, 9], [0, 2, 0, 1], [0, 2, 0, 2], [0, 2, 7, 5], [0, 2, 7, 7]]
```

3.2 Vérifications sur un chemin

On peut maintenant s'intéresser aux propriétés concernant un chemin donné. On va distinguer 2 grandes types de caractétistiques sur les chemins : - toujours p : vrai si la propriété p est vraie dans toutes les salles du chemin ; - un_jour p : vrai si la propriété p est vraie dans au moins une salle du chemin.

On peut écrire les fonctions d'ordre supérieur `toujours` et `un_jour` en Haskell ainsi :

```
toujours propriete [] = True
toujours propriete (tete:queue) =
                    (propriete tete) && toujours propriete queue

un_jour propriete [] = False
un_jour propriete (tete:queue) =
                    (propriete tete) || un_jour propriete queue
```

N.B. : on aurait aussi pu définir ces fonctions par pliage :

```
toujours2 propriete liste =
        foldl (\acc -> \elem -> acc && propriete elem) True liste
un_jour2 propriete liste =
        foldl (\acc -> \elem -> acc || propriete elem) False liste
```

Exemple :

```
Prelude> toujours torche [0, 1]
True
Prelude> un_jour fenetre [0, 2, 7, 7]
False
```

1. Écrire la fonction `toujours(propriete: int -> bool, chemin: list[int]) -> bool` en Python.

Exemples d'utilisation

```
>>> toujours(torche, [0, 1, 5]
True
>>> toujours(lambda salle: torche(salle) or tableau(salle), [0, 1, 6])
False
```

2. Écrire la fonction `un_jour(propriete: int -> bool, chemin: list[int]) -> bool` en Python.

Exemple d'utilisation

```
>>> un_jour(coffre, [0, 2, 7, 5])
True
```

3.3 Vérification sur les chemins

On peut maintenant s'intéresser à de nouvelles propriétés : - existe : existe-t-il un chemin de longueur n vérifiant une propriété (toujours p, un_jour p) donnée ? - tous : est-ce-que tous les chemins de longueur n vérifient une propriété (toujours p, un_jour p) donnée ?

Cela s'écrit alors "facilement" en Haskell :

```
tous operateur propriete longueur =
   foldl (\acc -> \chemin -> acc && (operateur propriete chemin))
         True
         (prolongement (longueur-1) [[0]])

existe operateur propriete longueur =
   foldl (\acc -> \chemin -> acc || (operateur propriete chemin))
         False
         (prolongement (longueur-1) [[0]])
```

N.B. : on a utilisé ici des pliages, mais on aurait aussi pu passer par une définition récursive.

Exemples d'utilisation : - Est-ce que tous les chemins de longueur 2 ont un tableau dans toutes les salles ?

```
Prelude> tous toujours tableau 2
True
```

- Y a-t-il un chemin de longueur 4 dont toutes les salles ont une torche ?

```
Prelude> existe toujours torche 4
True
```

- Y a-t-il un chemin de longueur 7 qui passe par une salle avec un tableau éclairé par une fenêtre ?

```
Prelude> existe un_jour (\salle -> fenetre salle && tableau salle) 7
True
```

1. Proposer une implantation en Python de la fonction `tous`.

Exemples d'utilisation

```
>>> tous(toujours, lambda salle: salle < 10, 4)
True
>>> tous(un_jour, fenetre, 3)
False
```

2. Proposer une implantation en Python de la fonction `existe`.

Exemples d'utilisation

```
>>> existe(toujours, torche, 4)
True
>>> existe(un_jour, coffre, 3)
True
```

3. Quel est le type de la fonction `tous` ?

3.4 Vérification sur tous les chemins

On peut maintenant se poser la question de réfléchir sur tous les chemins possibles :

- est-on sûr de ne jamais pouvoir atteindre se retrouver dans la salle 3 ?
- est-ce qu'il existe au moins un chemin qui mène à une salle avec un coffre et une torche ?

Le problème est qu'il existe une infinité de chemins... de longueurs potentiellemt infinies...

Mais voilà... dès l'instant qu'un chemin "boucle" (repasse par une salle déjà visitée sur ce chemin), il n'est pas nécessaire d'aller plus loin pour déterminer si `toujours` ou `un_jour` sont vérifiés.

1. Proposer des versions de "tous" et "existe" raisonnant sur tous les chemins possibles, quelle que soit leur longueur.

3.4 Liens entre les salles

Dans les différents cas traités jusqu'à présent, les propriétés analysées ne concernaient qu'une salle donnée. Mais imaginons un cas où on pourrait allumer une torche à soi (avec autonomie courte) dès qu'on rentre dans une salle avec une torche. On pourrait alors savoir si dans tous les chemins, une salle non éclairée est précédée d'une salle avec une torche...

En admettant que l'entrée soit aussi la sortie, on peourrait aussi se demander s'il existe un chemin passant permettant de prendre un coffre puis de ressortir...

Ces différents problèmes imposent de réfléchir sur différents états d'un chemin...
Voici une belle source d'idées pour de nouvelles fonctions à écrire...