Devoir maison 2018 – version avec exemples

1 Impératifs

Travail à rendre par mail à `marc-michel.corsini@u-bordeaux.fr` en utilisant une adresse officielle de la forme `prenom.nom@etu.u-bordeaux.fr`, vous indiquerez dans le corps du mail les noms et prénoms des membres du groupe et vous prendrez soin de mettre en copie vos camarades **CC**, afin que tous reçoivent l'accusé de réception du mail. Date limite du rendu le **Lundi 03 décembre à 08h00 (matin)**, tout retard sera assorti d'une pénalité. Pour ce DM vous travaillerez par groupe de 2 **exceptionnellement** 3[1], évidemment la note sera adaptée « plus on est de fous, moins il y a de riz » (Coluche).
Le rendu est constitué d'un document au format **PDF** pour la partie théorique (TdA, axiomes, complexité) et un code **python 3.x**.

2 Réalisation d'un type de données abstrait : automates finis

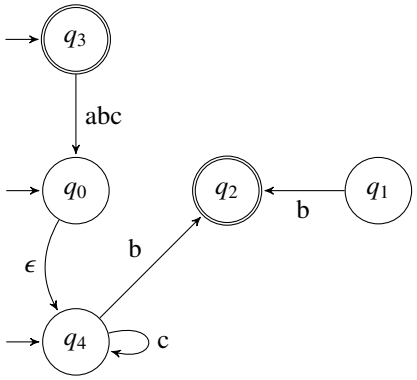
Effectuer les opérations classiques sur les automates finis, appliquer les algorithmes vus en cours (complétion, détermination, minimisation, ...). Un automate peut être lu à partir d'un fichier suffixé « .aut », la syntaxe est simple. Les états sont des entiers, le symbole # est la marque d'un commentaire qui s'étend jusqu'à la fin de la ligne. Sur une ligne on peut avoir,

- des états initiaux le premier caractère de la ligne est un « > »,
- des états terminaux, dans ce cas le premier caractère de la ligne est un « < »
- une transition, la syntaxe est « int str int », le premier entier est l'état de départ de la transition, le second entier est l'état d'arrivée, la chaîne, éventuellement vide est l'étiquette de la transition (si la chaîne est vide, il s'agit d'une ϵ-transition

Il n'y a pas d'ordre sur les informations

```
> 0 4
< 2 3
> 3
3 abc 0
1 b 2
0 4
4 b 2
4 c 4
```

Ce fichier correspond à l'automate suivant :



(a) automate $A_1 = (\{q_0, q_1, q_2, q_3, q_4\}, \{a, b, c\}, \Delta, \{q_0, q_3, q_4\}, \{q_2, q_3\})$

Vous trouverez dans `automata.zip` un code **python** permettant de lire un automate depuis un fichier, et de sauvegarder un automate dans un fichier.

[1] Dans ce cas veuillez me contacter par mail **au plus tard le mercredi 22 oct. 2017 12h00** afin que je donne mon aval.

1. `automata_0.aut` un fichier sans problème
2. `automata_1.aut` un fichier avec information incorrecte
3. `rwAutomata.py` définit deux classes `BasicReader` et `BasicWriter` qui sont testées sur les deux fichiers.

2.1 Présentation du problème

Un automate fini est une 5-uplet $A = (Q, \Sigma, \Delta, S, F)$ où Q est l'ensemble des états, Σ est l'alphabet du langage, Δ est l'ensemble des transitions, $S \subseteq Q$ l'ensemble des états initiaux, $F \subseteq Q$ l'ensemble des états terminaux. $L(A)$ est le langage reconnu par l'automate A.

2.2 Travail à effectuer

Pour chaque méthode, j'attends la signature, les axiomes ad-hoc ainsi qu'un calcul de complexité dépendant de **vos** algorithmes où $n = |\ Q\ |$ représente le cardinal des états, $m = |\ \Delta\ |$ le cardinal des transitions et $s = |\ \Sigma\ |$ le cardinal de l'alphabet.

2.2.1 Services

Il n'y a pas d'attributs publiques ou protégés – c'est-à-dire de valeur modifiable – dans la classe à réaliser, un petit programme test **test_dmAutomates.py** vérifie l'existence des méthodes demandées et l'absence d'attributs non protégés en écriture. Le nom de la classe et celui des méthodes à réaliser sont imposés.

1. La classe **Automaton**, n'hérite que de la classe **object**.
2. Le constructeur `__init__` :
   - Sans argument, il crée un automate reconnaissant le langage vide (ne pas confondre avec le mot vide).
   - Un argument qui est un fichier traité par **BasicReader** si le fichier n'existe pas, un automate reconnaissant le langage vide sera créé.
   - Cinq arguments :
     - Le premier est un itérable[2] d'entiers (les états), tout ce qui n'est pas entier est ignoré
     - Le second est un itérable de caractères alpha numérique (l'alphabet) – lettre a-zA-Z, chiffre 0-9 – tout ce qui n'est pas alphanumérique est ignoré
     - Le troisième est un itérable de triplets (les transitions) de la forme (entier, chaine, entier), les entiers sont des états, la chaine est constituée de lettres de l'alphabet, tout ce qui n'est pas conforme est ignoré
     - Le quatrième est un itérable d'entiers inclus dans le premier paramètre – tout ce qui n'est pas conforme est ignoré – ce sont les états initiaux.
     - Le cinquième est un itérable d'entiers inclus dans le premier paramètre – tout ce qui n'est pas conforme est ignoré – ce sont les états terminaux.

```
>>> a = Automaton({0,1}, "abc", [(0, 'a', 0), (0, 'b', 1), (2, 'c', 3)], [0], [1])
rejecting (2, c, 3)
>>> print(a)
Etats :
 0 1
Alphabet :
 a b c
Transitions :
 0 b 1
 0 a 0
Etats initiaux :
 0
Etats terminaux :
 1
>>> a = Automaton(range(4), "abc", [(0, 'a', 0), (0, 'b', 1), (2, 'c', 3)], [0], [1])
>>> print(a)
Etats :
 0 1 2 3
Alphabet :
```

[2] x est itérable si, en **python** on peut écrire « for v in x : ».

```
 a b c
Transitions :
 2 c 3
 0 b 1
 0 a 0
Etats initiaux :
 0
Etats terminaux :
 1
>>> a = Automaton(range(4), "abc", [(0, 'a', 0), (0, 'b', 1), (2, 'cc', 3)], [0], [1])
>>> print(a)
Etats :
 0 1 2 3
Alphabet :
 a b c
Transitions :
 2 cc 3
 0 b 1
 0 a 0
Etats initiaux :
 0
Etats terminaux :
 1
>>> a = Automaton()
>>> a
Automaton(2, 0, 0, 1, 1)
>>>
```

3. Les imprimables `__repr__` et `__str__`. En supposant l'automate A_1 stocké dans la variable `a`. On aura accès au cardinal de chaque ensemble Q, Σ, Δ, S, F (`repr`), ou au détail de chaque information (`str`).

```
>>> a = Automaton('automata_0')
>>> a
Automaton(5, 3, 5, 3, 2)
>>> print(a)
Etats :
  0 1 2 3 4
Alphabet:
  a b c
Transitions:
  0 4
  1 b 2
  3 abc 0
  4 b 2
  4 c 4
Etats initiaux:
  0 3 4
Etats terminaux:
  2 3
```

4. 3 booléens `afd, afdc, afn` (**property**) qui renvoient **True** ou **False** en fonction de la nature de l'automate.
5. des attributs en lecture seule `property` (**non modifiable**)
   (a) `automata` renvoie le 5-uplet non modifiable de l'automate
   (b) `Q` : renvoie la liste triée des états
   (c) `Sigma` : renvoie la liste triée des symboles de l'alphabet
   (d) `Delta` : renvoie la liste triée des transitions
   (e) `S` : renvoie la liste triée des états initiaux
   (f) `F` : renvoie la liste triée des états terminaux

```
>>> a
Automaton(5, 3, 5, 3, 2)
>>> a.automata
([0, 1, 2, 3, 4], ['a', 'b', 'c'], [(0, '', 4), (1, 'b', 2), (3, 'abc', 0), (4, 'b', 2), (4, 'c', 4)], [0, 3, 4], [2, 3])
```

```
>>> a.automata[0].append(5)
>>> a.automata
([0, 1, 2, 3, 4], ['a', 'b', 'c'], [(0, '', 4), (1, 'b', 2), (3, 'abc', 0), (4, 'b', 2), (4, 'c', 4)], [0, 3, 4], [2, 3])
>>> a.automata[2][0][1] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> a.automata[2][0] = 'x'
>>> a.automata
([0, 1, 2, 3, 4], ['a', 'b', 'c'], [(0, '', 4), (1, 'b', 2), (3, 'abc', 0), (4, 'b', 2), (4, 'c', 4)], [0, 3, 4], [2, 3])
>>> a.Q
[0, 1, 2, 3, 4]
>>> a.Sigma
['a', 'b', 'c']
>>> a.Delta
[(0, '', 4), (1, 'b', 2), (3, 'abc', 0), (4, 'b', 2), (4, 'c', 4)]
>>> a.S
[0, 3, 4]
>>> a.F
[2, 3]
>>> a.Q
[0, 1, 2, 3, 4]
>>> type(a.Q)
<class 'list'>
>>> a.Q.append(5)
>>> a.Q
[0, 1, 2, 3, 4]
>>> a.Q.extend([4, 5])
>>> a.Q
[0, 1, 2, 3, 4]
>>> a.Q.pop(0)
0
>>> a.Q.pop(3)
3
>>> a.Q
[0, 1, 2, 3, 4]
>>> a
Automaton(5, 3, 5, 3, 2)
>>>
```

6. Opérations sur les automates, renvoyant un nouvel **Automaton** et ne modifiant pas les automates initiaux
   (a) `access()` renvoie un nouvel **Automaton** ne contenant que les états accessibles.
   (b) `deterministe()` renvoie un nouvel **Automaton** qui est la version déterministe de l'automate.
   (c) `minimal()` renvoie un nouvel **Automaton** qui est l'automate obtenu après minimisation.
   (d) `complete`() renvoie un nouvel **Automaton** qui est la version déterministe et complet de l'automate
   (e) `complement`() renvoie un nouvel **Automaton** qui est l'automate complémentaire de l'automate initial.

$$L(\overline{A}) = \overline{L(A)}$$

   (f) `union`$(B_1, \ldots, B_n)$ $n \geq 1$, l'automate résultant est non déterministe.

```
>>> a = Automaton('automata_0')
>>> b = Automaton()
>>> c = Automaton('automata_0')
>>> d = a.union(b)
>>> e = a.union(a, c, b)
```

   (g) `inter`(B), l'automate résultant est déterministe, vous utiliserez la propriété suivante :

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

   (h) `concat`(B), l'automate résultant est non déterministe
   (i) `fermeture`(), l'automate résultant est non déterministe

7. Opérations de comparaison entre automates, renvoie **True** ou **False**
   (a) `__eq__`, $A == B$ si $L(A) = L(B)$
   (b) `__lt__`, $A < B$ si $L(A) \cap \overline{L(B)} = \emptyset$, il s'agit de l'inclusion des langages reconnus par les automates.

```
>>> a = Automaton('automata_0')
>>> b = Automaton()
>>> c = Automaton('automata_0')
>>> d = a.union(b)
>>> e = a.union(a, c, b)
>>> a == b
False
>>> a == c
True
>>> a == d
True
>>> a == e
True
```

8. Opérations sur les mots (**facultatif**, points bonus pour la programmation)

   (a) `accepte`(w) savoir si un mot w est reconnu par l'automate, renvoie True ou False

   (b) `execution`(w) savoir s'il existe une dérivation **maximale** $(q_0, w) \vdash (q', \epsilon), q_0 \in S$, si elle n'existe pas renvoie une liste vide, si elle existe (et est unique) renvoie la liste des états, si elle n'est pas unique renvoie une liste de liste d'états – chaque sous-liste correspondant une exécution **maximale** particulière.

   (c) `derivation`(q, w), dans le cas déterministe, renvoie une liste d'états de longueur **maximale** et le mot résiduel, dans le cas non déterministe renvoie la liste de toutes les dérivations maximales possibles accompagnées du mot résiduel.

   Ces opérations, sont plus complexes puisqu'il faut faire tous les cas possibles, ce qui nécessite l'utilisation d'une pile (ou d'une file). En prenant, l'automate A_1 voici ce que l'on obtient :

```
>>> a.accepte('a')
False
>>> a.accepte('b')
True
>>> a.execution('b')
[[0, 4, 2], [4, 2]]
>>> a.execution('a')
[]
>>> a.execution('abc')
[3, 0, 4]
>>> a.derivation(3, 'abc')
[([3, 0, 4], ''), ([0, 4], 'abc'), ([4], 'abc')]
```

2.2.2 Cas de méthodes non demandées

Vous avez toute liberté pour rajouter des méthodes que vous pensez utiles pour la manipulation des automates. Vous devrez, dans le cas où ces méthodes sont non privées – donc accessibles à l'utilisateur – fournir : signature, axiomes, complexité et justifier leurs raisons d'être.

3 Évaluation, remarques

L'évaluation globale du travail sera sur **12** points pour la partie « signatures, axiomes, compléxité » et sur **10** points pour la partie implémentation « codage et tests ». Le code sera impérativement fait avec une approche orientée objets pour le TdA Les tests mettent en évidence la validité de votre code, et permettent de certifier que l'application fournie est conforme à la demande (et à vos axiomes).
Il vaut mieux implémenter quelques fonctionnalités de manière adéquate que tout faire n'importe comment.
Si votre code ne tourne pas, l'évaluation est 0 pour la partie code.
Tout cas de tricherie sera durement sanctionné

4 Ressources

Le support de cours, et particulièrement le chapitre « Introduction à la calculabilité » vous servira de point de départ.