Automates_TP1

November 15, 2021

1 Automates finis - TP 1

1.0.1 Objectif du TP

L'objectif de ce premier TP est d'implémenter un programme en **Python** avec deux arguments: un fichier `.txt` contenant la description d'un automate fini A et un mot m à reconnaître. Votre programme doit (a) lire/charger l'automate A à partir du fichier `.txt`, (b) afficher *ERROR* si l'automate est non déterministe, ou (c) si, au contraire, l'automate est déterministe, afficher *YES* si le mot peut être reconnu par l'automate, ou *NO* si le mot ne peut pas être reconnu par l'automate. En pseudo-code, cela donne :

```
1. A = automate(fichier)
2. si deterministe(A) alors
3.    si reconnait(A,m) alors
4.      afficher "YES"
5.    sinon
6.      afficher "NON"
7. sinon
8.   afficher "ERROR"
```

1.1 Environnement de travail

Énoncés: les énoncés des TPs d'automates finis sont disponibles en deux formats : PDF et notebook Jupyter `.ipynb`. Ce dernier contient du code python intégré que vous pouvez exécuter directement sur Binder : `https://mybinder.org/v2/gh/ceramisch/automatesfinis/HEAD?filepath=Automates_TP1.ipynb`.

Travailler en salle TP Vous devrez écrire un programme `tp1automates.py` en langage **Python** pour réaliser le programme décrit ci-dessus. L'énoncé au format notebook Jupyter vous permet de tester des bouts de code, mais vous ne devez pas programmer sur les notebook Jupyter directement. En salle TP, utilisez l'environnement de programmation **Spyder** disponible sur la session **Linux/Luminy**. Pour travailler à la maison, consultez la fin l'énoncé.

Dépôt de fichiers: les TP sont incrémentaux : on a besoin du TP1 pour faire le TP2, etc… Cela aboutira à la création d'un programme capable de reconnaître, à l'aide d'un automate fini, si un *mot* quelconque appartient au langage dénoté par une *expression régulière* quelconque. Prenez un moment maintenant pour préparer votre environnement de travail. Nous vous conseillons de créer un dépôt de contrôle de version `git` pour ce mini-projet, par exemple sur github.com ou gitlab.com. Vous pouvez aussi travailler sur un dossier partagé Dropbox, Google Drive ou AMUBox. Dans tous les cas, sauvegardez vos fichiers à la fin de chaque séance.

1.2 Format de fichier `.txt`

Un automate décrit dans un fichier `txt` consiste en une suite de lignes, où chaque ligne est un triplet, sauf la dernière. Les éléments du triplet sont séparés par des espaces. Le triplet `X Y a` correspond à une transition entre un état source `X` et un état cible `Y` étiquetée par le symbole `a`. Il est recommandé de représenter les états `X` et `Y` avec des nombres, et les symboles `a` avec des lettres majuscules ou minuscules non-accentuées. Le symbole spécial `%` est utilisé pour les transition- . Le premier état source de la première transition est considéré comme l'état initial de l'automate. La dernière ligne est toujours précédée de la lettre majuscule `A` indiquant les états d'acceptation. Ensuite, les états d'acceptation sont listés, séparés par des espaces.

Voici un exemple d'automate déterministe décrit dans un fichier. Cet automate reconnait a^*+b^*. Pour vous en convaincre, dessinez-le sur une feuille:

[1]:
```
source = """0 a 1
0 b 2
1 a 1
2 b 2
A 0 1 2
"""
```

1.3 Bibliothèque `automaton.py` fournie

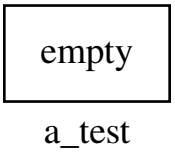
Nous vous fournissons une bibliothèque python `automaton.py`, disponible sur le github du cours, avec les fonctionnalités suivantes: * Création et manipulation d'un objet `Automaton` qui représente un automate fini * Lecture et écriture à partir d'un fichier textuel * Affichage graphique à l'aide de `graphviz`

Vous devez bien comprendre le fonctionnement de cette bibliothèque. En particulier, elle utilise de la programmation orientée objets en Python. Les prochaines sections vous permettent de découvrir la bibliothèque. Si vous êtes à l'aise en Python, vous pouvez ouvrir le code de la bibliothèque et le modifier, mais cela ne devrait pas être nécessaire.

Vous devez télécharger la bibliothèque `automaton.py` et la placer dans le même dossier où se trouve votre script/programme python du TP 1. Ensuite, vous pouvez créer un nouvel automate et lui donner un nom. Cet automate, pour le moment, sera vide. Vérifiez en exécutant le code ci-dessous :

[2]:
```
from automaton import Automaton
a_test = Automaton("a_test")
a_test
```

[2]:

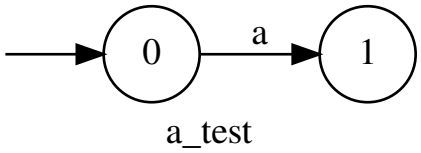


Cela affiche un automate vide. Essayez de changer son nom et exécutez à nouveau le bloc ci-dessus. Ensuite, essayez de faire la même chose en local, dans votre code `tp1automates.py`, en copiant le bout de code ci-dessus. Vous n'aurez pas l'affichage graphique, vous devriez voir quelque chose comme `<automaton.Automaton object at 0x7f2140ebd2b0>` - cela indique qu'un objet de la classe `Automaton` a été créé.

Continuons à nous familiariser avec la bibliothèque sur ce notebook. Vous pouvez ajouter des états et des transitions à votre automate vide. Attention, la source de la première transition ajoutée sera toujours considérée comme l'état initial:

[3]:
```
a_test.reset()
a_test.add_transition("0","a","1")
a_test
```

[3]:

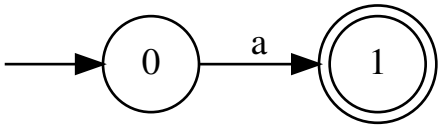


a_test

Notez que cet automate ne reconnait aucun mot, car il n'a aucun état d'acceptation/final. Vous pouvez marquer un ou plusieurs états comme finaux à l'aide de la fonction suivante:

[4]:
```
a_test.make_accept("1")
a_test
```

[4]:

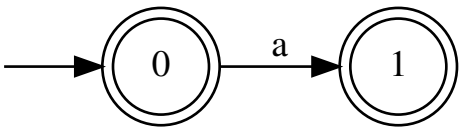


a_test

Alternativement, vous pouvez marquer plusieurs états comme états d'acceptation:

[5]:
```
a_test.make_accept(["0","1"])
a_test
```

[5]:



a_test

Vous pouvez afficher l'automate sous la forme textuelle à n'importe quel moment dans votre code, par exemple, pour déboguer. Notez que la fonction delta est donnée sous la forme d'un tableau de transitions:

[6]:
```
print(a_test)
```

```
a_test = <Q={0,1}, S={a}, D, q0=0, F={0,1}>
D =
| |a|
-----
|0|1|
-----
|1| |
-----
```

Dans cet exemple, `Q` contient la liste d'états `{0,1}`, l'alphabet `S` est automatiquement construit à partir des transitions et contient uniquement le symbole `{a}`, l'état initial est `q0=0` et les états d'acceptation `F={0,1}`. Ces valeurs sont accessibles aussi directement, sous la forme de listes ou de chaînes de caractères, via des variables de l'automate (contrairement à Java, toutes les méthodes et attributs sont publics en Python) :

[7]:
```
print(a_test.states)
print(a_test.alphabet)
print(a_test.initial)
print(a_test.acceptstates)
print(a_test.transition_table)
```

```
['0', '1']
['a']
0
['0', '1']
| |a|
-----
|0|1|
-----
|1| |
-----
```

Vous pouvez aussi afficher votre automate sous la forme de fichier `.txt`:

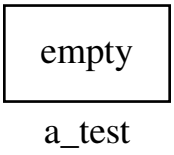
[8]:
```
print(a_test.to_txtfile("test.txt"))
```

```
0 a 1
A 0 1
```

Un automate peut être réinitialisé à tout moment:

[9]:
```
a_test.reset()
a_test
```

[9]:

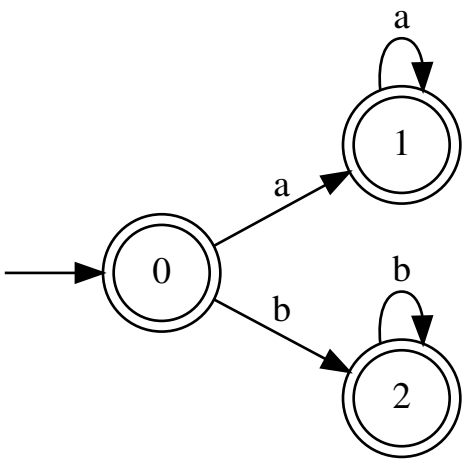


a_test

Vous pouvez aussi construire un automate à partir d'une chaîne de caractères directement à l'aide de la fonction `from_txt`:

[10]:
```
source = """0 a 1
0 b 2
1 a 1
2 b 2
A 0 1 2
"""
astarbstar = Automaton("astarbstar")
astarbstar.from_txt(source)
astarbstar
```

[10]:



astarbstar

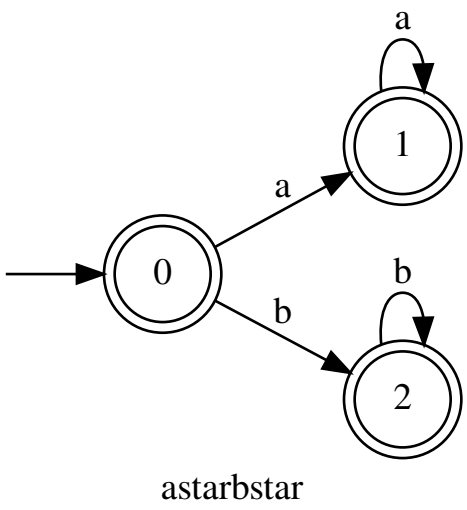
Essayez de modifier la variable `source` et observez le résultat dans l'automate. Ajouter ou supprimez des transitions, des états, des états d'acceptation…

Vous pouvez aussi lire un automate en passant directement le nom du fichier à la fonction `from_txtfile`:

[11]:
```
astarbstar.from_txtfile("test/astarbstar.af")
astarbstar
```

```
WARNING: Automaton astarbstar not empty: content will be lost
```

[11]:



astarbstar

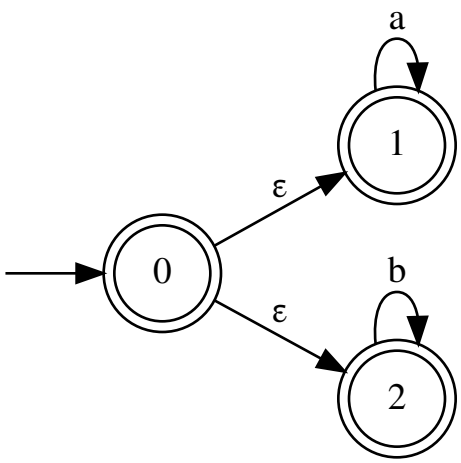
Remarque: la bibliothèque vous préviendra si quelque chose d'étrange se passe à l'aide des `WARNING`, comme ci-dessus lors que vous écrasez le contenu existant par le chargement d'un nouvel automate. Parfois ces messages vous aideront à ne pas faire de bêtise, d'autres fois vous pouvez les ignorer sans problèmes.

Les transitions- sont représentées par le symbole spécial `%` (constante `automaton.EPSILON`). Vous pouvez créer un automate non-déterministe avec des transitions- comme ceci:

[12]:
```
source = """0 % 1
0 % 2
1 a 1
2 b 2
A 0 1 2
"""
astarbstar.from_txt(source)
astarbstar
```

```
WARNING: Automaton astarbstar not empty: content will be lost
```

[12]:



astarbstar

Les automates sur ce sujet s'affichent graphiquement à l'aide de la fonction `to_graphviz()` (qui est executée automatiquement en coulisses). Vous pouvez toujours copier-coller des automates sur ce notebook pour les visualiser, mais il est peut-être plus pratique de les créer directement en local. Pour cela, il suffit d'appeler la fonction `to_graphviz()` avec un nom de fichier. Un nouveau fichier PDF avec la représentation graphique de l'automate sera alos créé (l'extension `.pdf` sera automatiquement ajoutée). La fonction renvoie l'objet `graphviz` au format DOT. Il n'est pas nécessaire de le comprendre, mais cela peut vous aider à comprendre comment fonctionne la bibliothèque :

[13]:
```
astarbstar_gv = astarbstar.to_graphviz("test/astarbstar.gv")
print(astarbstar_gv.source)
```

```
digraph finite_state_machine {
  rankdir=LR;
  size="8,5"  label="astarbstar"  node [shape = doublecircle]; "0" "1" "2";
  node [shape = circle];
  __I__ [label="", style=invis, width=0]
  __I__ -> "0"
  "0" -> "1" [label = ];
  "0" -> "2" [label = ];
  "1" -> "1" [label = a];
  "2" -> "2" [label = b];
}
```

À certains moments, vous aurez besoin d'accéder à la liste des **états** et de **transitions** à l'intérieur de l'automate. Ainsi, il faut comprendre comment ces éléments sont représentés en interne. Prenez le temps de bien comprendre la structure, faites un schéma sur une feuille si besoin. Vous pouvez aussi regarder le code-source de la bibliothèque `automaton.py` directement. Cela vous aidera à mieux vous servir de la bibliothèque lors de ce TP et des suivants.

Chaque état est représenté comme un objet de la classe `State` et stocké dans un dictionnaire nommé `statesdict`. Les clés de ce dictionnaire sont les noms des états (`str`), et les valeurs sont les objets de type `State` :

[14]:
```
astarbstar.statesdict
```

[14]:
```
OrderedDict([('0', <automaton.State at 0x7f71b825be50>),
             ('1', <automaton.State at 0x7f71b8280430>),
             ('2', <automaton.State at 0x7f71b8280370>)])
```

Par exemple, pour accéder à l'objet `State` correspondant à l'état numéro `"1"`, vous pouvez accéder à `astarbstar.statesdict["1"]`. Chaque état contient une liste de transitions :

[15]:
```
astarbstar.statesdict["1"].transitions
```

[15]:
```
OrderedDict([('a',
              OrderedDict([(<automaton.State at 0x7f71b8280430>, None)]))])
```

Nous voyons ci-dessus, par exemple, que l'état nommé `"1"` contient une transition pour le symbole `a` menant vers un autre dictionnaire d'état. Ce dictionnaire permet de prendre en compte les transitions vers plusieurs états dans les automates non déterministes. Pour accéder à l'état destination de la transition, on doit d'abord transformer le dictionnaire en liste, puis récupérer le premier élément de cette liste, et finalement afficher cet état comme une chaîne de caractères `str`. Mis bout à bout, cela donne :

[16]:
```
# source state 1 (from dict of states), transition on "a", first (and only)␣
 ↪destination state "1"
str(list(astarbstar.statesdict["1"].transitions["a"])[0])
```

[16]:
```
'1'
```

Ouf ! C'est bien complexe, mais cela permet de représenter n'importe quel automate fini déterministe ou non déterministe.

Alternativement, utilisez l'attribut `transitions` de l'automate, qui donne la liste de transitions sous la forme d'une `list` python simple. Chaque transition est une tuple de trois éléments de type `str` : source, symbole, destination. Cela peut être pratique, si vous voulez manipuler des objets plus simples que les dictionnaires et les objets `State` :

[17]:
```
for (source,symb,dest) in astarbstar.transitions : # et non pas a.transitions()
  print( "{} --{}--> {}".format(source,symb,dest))
```

```
0 --%--> 1
0 --%--> 2
1 --a--> 1
2 --b--> 2
```

Vous trouverez d'autres exemples d'usage de la bibliothèque dans `automaton.py` à la toute fin du code-source.

2 Travail à effectuer

Votre travail consiste à implémenter deux fonctionnalités:

1. Écrire une fonction qui prend un automate fini en entrée et qui renvoie un booléen pour indiquer si, oui ou non, l'automate est déterministe
2. Écrire une fonction qui prend un automate fini déterministe et un mot en entrée, et qui renvoie un booléen pour indiquer si, oui ou non, le mot est reconnu par l'automate

De plus, vous devez écrire un script/programme qui peut être exécuté sur un terminal et qui prend en entrée (`sys.argv`) deux arguments: un fichier texte contenant un automate et un mot à reconnaître, dans cet ordre. Le script doit :

1. Charger l'automate depuis le fichier texte (à l'aide de la bibliothèque fournie)
2. Vérifier si l'automate est déterministe (fonctionnalité 1 ci-dessus) et, sinon, afficher *ERROR*
3. Si l'automate est déterministe, vérifier s'il reconnaît le mot (fonctionnalité 2 ci-dessus) et afficher *YES* si le mot est reconnu, *NO* sinon

Quelques fichiers de test vous sont fournis dans le dossier `test`. Voici quelques exemples d'exécution pour votre programme `tp1automates.py` :

```
$ ./tp1automates.py test/astarbstar.af a
YES
$ ./tp1automates.py test/astarbstar.af b
YES
$ ./tp1automates.py test/astarbstar.af bbb
YES
$ ./tp1automates.py test/astarbstar.af %
YES
$ ./tp1automates.py test/astarbstar.af aaaaaaa
YES
$ ./tp1automates.py test/astarbstar.af bbaa
NO
$ ./tp1automates.py test/astarbstar.af aba
NO
$ ./tp1automates.py test/astarbstar.af aabbb
NO
$ ./tp1automates.py test/astarbstar.af abc
NO
$ ./tp1automates.py test/astarbstar-epsilon.af abb
ERROR
$ ./tp1automates.py test/astarbstar-nfa.af abb
ERROR
```

Une version vide de `tp1automates.py` est fournie sur le git du cours. Utilisez-la comme point de départ de votre TP.

Vous devez non seulement implémenter le programme mais aussi le tester. Pour cela, écrivez 3 automates finis au format textuel pour augmenter la base de tests au délà des deux fichiers fournis dans le dossier `test`. Vous pouvez utiliser des automates vus en cours ou en TD, par exemple. Ces

automates doivent être placés dans le dossier `test` et formeront votre base de tests, qui grandira au fur et à mesure des TPs. Vous pouvez générer les `.pdf` correspondants à chaque automate test à l'aide de la fonction `to_graphviz()`

Pour déboguer votre programme, n'oubliez pas que vous pouvez à tout moment appeler `print(a)` pour afficher l'automate sur le terminal, ou appeler `a.to_graphviz("out.pdf")` pour le visualiser graphiquement dans un fichier `out.pdf`. En python, vous pouvez/devez aussi utiliser la bibliothèque `pdb` pour le débogage, et vous pouvez aussi typer votre programme et vérifier les types à l'aide de `mypy`.

3 Travailler à la maison

Si vous avez l'habitude de travailler sur une IDE telle que intelliJ, eclipse, pycharm ou Spyder, utilisez-la. Sinon, installez Anaconda+Spyder, comme en salle TP. Alternativement, vous pouvez programmer avec un simple éditeur de texte comme atom, SublimeText, geany ou Notepad++, ou utiliser l'environnement 100% en ligne repl.it (mais celui-ci ne permet pas l'affichage graphique).

Pré-requis à installer : Pour les affichages graphiques ci-dessous, vous devez installer l'outil `graphviz` et la bibliothèque python correspondante. Sur **Linux**, une fois python 3 installé, vous pouvez exécuter:

```
sudo apt install graphviz
pip3 install graphviz # ou tout simplement pip install graphviz, selon votre configuration
```

Si vous êtes sur **Windows**, le plus simple est d'installer Anaconda. Une fois que vous l'avez téléchargé et installé, allez sur Anaconda Navigator > Environement > Update index. Affichez les packages 'Not installed' au lieu de 'Installed' puis cherchez *graphviz* dans la barre de recherche. Installez les packages *graphviz* et *python-graphviz* (les deux sont nécessaires). Ensuite, vous pouvez revenir à la page d'accueil (Home) et lancer l'IDE Spyder. Une vidéo sur Ametice vous explique comment installer Anaconda+Spyder et les pré-requis graphviz sur Windows.

[ ]: