Projet TP - Automates finis

2022-2023

Présentation du projet

L'objectif de ce projet est de développer une petite bibliothèque **Python** pour définir et manipuler des automates finis. La bibliothèque `automaton.py` fournie comporte déjà certaines fonctionnalités de base. Chacune des trois parties devrait pouvoir être faite au cours d'une séance de TP. Vous pouvez bien entendu travailler entre les séances. Attention la troisième partie est plus difficile que les deux autres.

Rendu de TP

Le rendu sera fait par groupe de 2. Ce projet doit être complété et soumis sur Ametice **avant la fin du dernier TP**. Vous rendrez ce projet sous forme d'une archive `.zip`. L'archive devra contenir, en plus du fichier contenant votre code `tp-automates.py`, un dossier `tests` comportant plusieurs fichiers `.af`. Vous devrez également ajouter un fichier readme contenant le nom des membres du binôme. L'archive devra obligatoirement être nommée:
`NOMDEFAMILLE1-Prenom1_NOMDEFAMILLE2-Prenom2.zip`.

Attention: les noms des fichiers et des fonctions que vous ajouterez devront être **exactement** ceux donnés par le sujet pour pouvoir être évalués. De plus vous ne pouvez pas modifier les fonctions déjà fournies de la bibliothèque.

Présentation de la bibliothèque

0.1 Création d'un automate

Un automate est donné par quatre attributs: un nom au format `str`, un état initial (`str`), une liste d'états finaux (liste de `str`) et une liste de transitions: liste de triplets (état de départ, lettre, état d'arrivée) où la lettre doit être un caractère (`chr`).

Exécutez le `main` du fichier `automaton.py`. Vous pouvez voir dans la console un automate vide.

Ajoutez les lignes suivantes avant le `print`:

```
a.add_transition("0","a","1")
a.set_initial("0")
a.make_final("1")
```

Identifiez dans la console, l'ensemble des états, l'alphabet, l'état initial, les états finaux et la table de transitions de l'automate `a`.

Ajoutez une transition entre les états `0` et `2` avec la lettre `b`, ajoutez `0` et `2` comme état finaux puis retirez `1` des états finaux, en utilisant la fonction `unmake_final`. Cet automate devrait reconnaître le langage $\{\varepsilon, b\}$, dessinez-le sur une feuille pour vous en convaincre.

0.2 Fichier automate

La bibliothèque vous fournit une méthode pour stocker un automate fini dans un fichier texte avec l'extension `.af`. Exécutez la ligne:

```
a.to_txtfile("./tests/epsilon_b.af")
```

Vérifiez que le dossier `tests` contient bien un fichier `epsilon_b.af`, et lisez son contenu: la première ligne indique l'état initial, la seconde ligne les états finaux et les lignes suivantes les transitions de l'automate.

Exécutez:

```
a2=Automaton("aut2")
a2.from_txtfile("./tests/astar_bstar.af")
print(a2)
```

Notez que cet automate comporte un symbol spécial qui n'est pas affiché dans l'alphabet. En effet le symbole `%` est réservé pour représenter le mot vide ε. Dessinez sur une feuille l'automate `a2`.

0.3 Autres fonctionnalités

Explorez les différentes méthodes de la bibliothèque en exécutant:

```
print(a2.get_alphabet())
print(a2.get_states())
print(a2.get_transitions())
a2.make_copy(a)
print(a2)
```

0.4 Quelques fonctionnalités de Python

En **Python** vous pouvez itérer sur une liste dans une boucle `for`. Par exemple, le code suivant stocke dans la liste `state_transitions` les transitions qui partent de l'état `0`:

```
state_transitions=[]
for t in  a.get_transitions():
  if t[0] == "0":
    state_transitions.append(t)
print(state_transitions)
```

De plus vous pouvez utiliser le fait que les transitions sont des triplets pour identifier directement leurs coordonnées. Ici par exemple on stocke dans la liste `state_with_epsilon` les états à partir desquels une transition ε est possible.

```
state_with_epsilon=[]
for (source,letter,target) in  a2.get_transitions():
  if letter == EPSILON and source not in state_with_epsilon:
    state_with_epsilon.append(source)
print(state_with_epsilon)
```

Notez que dans le `if` on peut facilement tester l'appartenance d'un élément à une liste.

0.5 Création de fichiers test

Créez les 4 fichiers suivants dans le dossier `tests`:

- `det_astar_bstar.af` un automate déterministe reconnaissant $a^* + b^*$
- `abstar.af` un automate reconnaissant le langage $(ab)^*$
- `aa_factor.af` un automate non-déterministe pour $(a+b)^*aa(a+b)^*$
- `det_aa_factor.af` un automate déterministe pour $(a+b)^*aa(a+b)^*$

Il est fortement encouragé de créer d'autres fichiers `.af` et de vous aider de ces fichier pour **tester votre code régulièrement**. Par ailleurs ces fichiers seront également utilisés pour vous évaluer.

Le projet est découpé en trois parties. **Pensez à tester votre code régulièrement**.

1 Exécution d'un automate

Le but de cette partie est de définir une fonction qui exécute un automate fini déterministe sur un mot.

1.1 Automate déterministe

Une première étape est de définir une fonction `is_deterministic` telle que `is_deterministic(a)` renvoie `True` si l'automate `a` est déterministe et `False` sinon. On rappelle qu'un automate peut être non-déterministe pour deux raisons: soit il possède un état p et une lettre a tels que deux transitions différentes existent à partir de p en lisant a, soit il possède une transition ε.

1.2 Exécution

Définissez une fonction `execute` telle que `execute(a,s)` renvoie la chaîne de caractères `"ERROR"` si l'automate `a` est non-déterministe. Dans le cas contraire la fonction doit renvoyer `True` si la chaîne de caractères `s` est acceptée par l'automate et `False` sinon.

2 Opérations sur les automates

2.1 Émonder un automate

Il s'agit ici de définir une fonction `get_accessible(a)` qui renvoie la liste des états accessibles d'un automate. De même définissez une fonction `get_coaccessible(a)` qui renvoie la liste des états co-accessibles d'un automate. Enfin vous définirez une fonction `trim(a)` qui supprime les états qui ne sont pas accessibles ou co-accessibles. Notez que cette fonction doit retirer ces états des états initiaux, finaux et les transitions les contenant. Un automate dont tous les états sont accessibles et co-accessibles est appelé *émondé* (*trim* en anglais).

2.2 Complément d'un langage

Premièrement il faudra définir une fonction `complete(a)` qui rend un automate complet, s'il ne l'est pas encore, en ajoutant un état puits vers lequel seront dirigées toutes les transitions manquantes. Deuxièmement, définissez une fonction `complement(a)` qui modifie `a` pour reconnaître le langage complémentaire.

2.3 Union et intersection

Il s'agit ici de définir une fonction `union(a,b,c)` et une fonction `intersection(a,b,c)` qui définissent respectivement l'automate `c` pour l'union et l'intersection des langages reconnus par `a` et `b`.

3 Déterminisation

L'objectif est de définir une fonction `determinize(a,b)` qui définit l'automate déterministe `b` équivalent à `a`. Une première étape sera de définir la fonction `remove_epsilon(a)` qui supprime les transitions ε de `a`.

Enfin vous pourrez modifier la fonction `execute` pour prendre en compte le cas où l'automate est non-déterministe.