Théorie des langages rationnels
TP 3 - Manipulation d'automates

Adrien Pommellet

18 janvier 2023

Ce TP est votre premier contact avec la manipulation d'automates en Python. Nous commencerons donc par programmer des algorithmes simples.

Téléchargez au préalable le fichier `thlr_automata.py` sur Moodle. Ce TP nécessite une installation préalable de la bibliothèque `graphviz` ; si elle ne s'avère pas disponible sur votre poste, utilisez la suite d'instructions suivante :

```
python -m venv ~/envthlr
. ~/envthlr/bin/activate
pip install graphviz
```

Les questions marquées d'une étoile `(*)` seront **notées** et font l'objet d'un rendu dont **la rédaction doit obligatoirement respecter les règles suivantes** :

— Les réponses au TP doivent être regroupées dans un unique fichier nommé `thlr_3_code.py` (tout autre fichier sera ignoré). Inutile d'uploader les bibliothèques externes que l'on vous fournit telle que `thlr_automata.py`.

— Il est impératif que le fichier soit interprétable, sans la moindre erreur de syntaxe. Si ce n'est pas le cas, le rendu entier sera ignoré. Testez votre code sous Linux directement depuis la console avec l'instruction `python3 thlr_3_code.py` depuis un répertoire contenant ses éventuelles dépendances, sans passer par un IDE.

— Les noms de fonctions donnés dans l'énoncé doivent être respectés.

— La réponse à la question numéro `i` et les éventuelles sous-fonctions utilisées doivent être placées entre des balises de la forme `#[Qi]` et `#[/Qi]` (sans espace après `#`, avec un `/` et non un \). Ainsi, si l'on vous demande d'écrire une fonction `f`, votre réponse doit être de la forme :

```
#[Qi]
def f():
  # Insert your code here
#[/Qi]
```

— Testez vos fonctions sur des exemples pour vous assurer de leur correction ; n'incluez toutefois pas ces tests et ces exemples entre les tags `#[Qi]` et `#[/Qi]`.

— Certaines fonctions sont interdépendantes : une erreur sur la première fonction peut compromettre les suivantes, soyez donc vigilants.

Le non-respect de ces critères peut mettre en défaut le système de notation automatique et conduire à une note de 0 pour le rendu dans son intégralité, sans contestation possible.

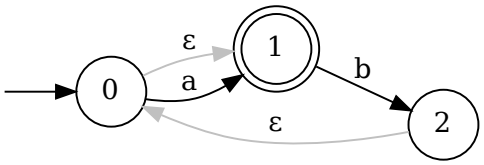

FIGURE 1 – Une représentation graphique de l'automate `A`.

1 Une introduction aux automates

En préambule, n'oubliez pas d'importer la bibliothèque d'automates `thlr_automata.py` :

```
from thlr_automata import *
```

1.1 Le constructeur

Les automates finis non-déterministes avec ε-transitions (ε-NFA) sont modélisés par la classe `ENFA`. Pour créer un nouvel automate, on doit utiliser un *constructeur*, c'est-à-dire une fonction `ENFA(all_states, initial_states, final_states, alphabet, edges)` qui prend en argument les listes d'entiers `all_states`, `initial_states`, et `final_states`, une liste de chaînes `alphabet`, et une liste de triplets `edges` représentant les arêtes : le triplet `(0, "a", 1)` représente l'arête $0 \xrightarrow{a} 1$. La chaîne `""` représente ε dans les automates. L'automate `A` de la figure 1 est obtenu en exécutant les instructions suivantes :

```
A = ENFA([0, 1, 2], [0], [1], ["a", "b"], [(0, "a", 1), (0, "", 1),
(1, "b", 2), (2, "", 0)])
A.export("A")
```

La commande `A.export("A")` exporte dans le répertoire par défaut une représentation graphique de l'automate `A` dans un fichier `A.pdf`.

Question 1. Créez l'automate `B` de la figure 2 grâce à la classe `ENFA` puis exportez-en une représentation graphique.

1.2 Opérations simples

Une fois l'automate `A` défini, il est possible d'accéder à ses attributs ; `A.all_states` correspond à l'ensemble de tous les états de `A` ; `A.initial_states`, à l'ensemble de tous les états initiaux de `A` ; `A.final_states`, à l'ensemble de tous les états finaux de `A` ; et `A.alphabet`, à l'ensemble des lettres (représentées par des chaînes) de l'alphabet de `A` auquel on adjoint la chaîne vide `""` représentant ε. Pour accéder aux arêtes, on utilise une méthode `A.get_successors` ; par exemple, `A.get_successors(0, "a")` renvoie l'ensemble des états pointés par des arêtes sortant de l'état `0` étiquetées par `a`, aussi appelés *successeurs immédiats* de `0` par `a`. Notez que cet ensemble peut être vide ou contenir plusieurs éléments (en cas de non-déterminisme).

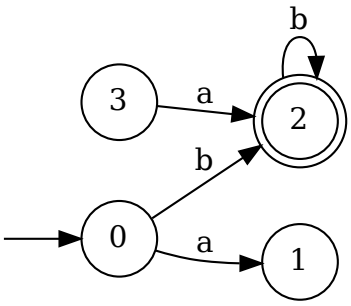

FIGURE 2 – Un automate `B` à représenter.

Il est possible de modifier un automate `A` existant avec différentes méthodes. Par exemple, `A.add_letter("x")` ajoute la lettre `x` à l'alphabet de `A` ; `A.add_edge(0, "x", 1)` ajoute une arête étiquetée par `x` entre les états `0` et `1` de l'automate ; `A.add_state()` ajoute un nouvel état à `A`, et renvoie le numéro assigné à cet état ; `A.remove_state(1)` retire l'état `1` de `A` ainsi que toutes les arêtes qui en viennent ou en sortent.

Question 2. Modifiez l'automate `B` à l'aide des opérations décrites précédemment, sans le redéfinir, de manière à obtenir l'automate de la figure 3.

2 Émondage d'un automate

Nous allons introduire dans cette partie une méthode de simplification des automates.

2.1 Descendants d'un état

Un état q' est un *descendant* d'un état q dans un automate s'il existe un chemin (peu importe son étiquette) allant de q à q'. On dit aussi que q' est *accessible* depuis q. Notons que q est naturellement accessible depuis q par le chemin vide. Par exemple, dans l'automate `C` de la figure 3, les états `0`, `1`, `2`, et `3` sont accessibles depuis `0`.

On souhaite désormais déterminer tous les états accessibles depuis un état q. Nous allons pour ce faire utiliser un algorithme de *parcours* sur l'automate. Son principe est le suivant :

— On maintient deux ensembles, un ensemble `incoming` des états à visiter (à l'origine ne contenant que q), et un ensemble `visited` (à l'origine vide) des états déjà visités.

— Dans une boucle, on retire un état de l'ensemble `incoming`, on l'ajoute à `visited`, puis on ajoute tous ses successeurs immédiats (peu importe la lettre de l'alphabet) qui ne sont pas déjà dans `visited` à `incoming`.

— Cette boucle prend fin quand `incoming` est vide ; on renvoie alors l'ensemble `visited`.

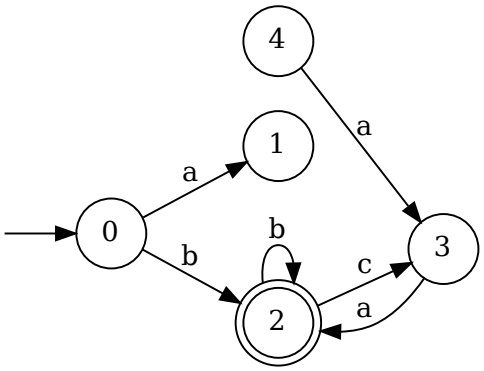

FIGURE 3 – L'automate `C` à obtenir à partir de `B`.

Question 3 (⋆). Écrivez une fonction `get_accessible_states(automaton, origin)` qui renvoie un ensemble contenant tous les états accessibles depuis l'état `origin` dans `automaton`. La méthode `get_successors` décrite dans la section précédente vous ainsi que la méthode `pop()` sur les ensembles vous seront utiles.

2.2 États utiles

Un état q est dit *accessible* (tout court) dans un automate donné s'il est accessible depuis un état initial ; il est dit *co-accessible* s'il existe un état final accessible depuis q ; il est *utile* s'il est à la fois accessible et co-accessible. Ainsi, dans l'automate `B` de la figure 2, l'état `1` est accessible, `3` est co-accessible, mais seuls `0` et `2` sont utiles.

Question 4 (⋆). Écrivez une fonction `is_accessible(automaton, state)` qui renvoie `True` si l'état `state` de l'automate `automaton` est accessible, et `False` sinon. Il vous faudra naturellement réutiliser la fonction `get_accessible_states`.

Question 5 (⋆). Écrivez une fonction `is_co_accessible(automaton, state)` qui renvoie `True` si l'état `state` de l'automate `automaton` est co-accessible, et `False` sinon. Il vous faudra naturellement réutiliser la fonction `get_accessible_states`.

Question 6 (⋆). Écrivez une fonction `is_useful(automaton, state)` qui renvoie `True` si l'état `state` de l'automate `automaton` est utile, et `False` sinon.

2.3 Simplification d'un automate

L'*émondage* d'un automate consiste à éliminer ses états inutiles. Retirer les états inutiles d'un automate ne change pas le langage accepté. Ainsi, l'automate `D` de la figure 4 obtenu en retirant les états inutiles `1` et `4` accepte un langage égal à celui de l'automate `C` de la figure 3.

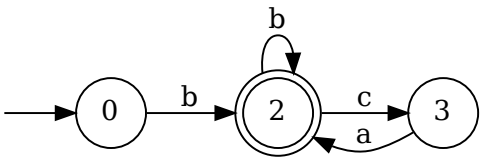

FIGURE 4 – L'automate `D` obtenu en émondant `C`.

Question 7 ($\star$). Écrivez une fonction `prune(automaton)` qui retire tous les états inutiles de l'automate `automaton`.