Théorie des langages rationnels
TP 5 - Déterminisation d'automates

Adrien Pommellet

January 18, 2023

L'objectif de ce dernier TP est d'étudier certaines propriétés des automates non-déterministes et d'implémenter l'algorithme de déterminisation vu en cours.

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_5_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_5_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.

1 Automates non-déterministes

1.1 Représentation d'un NFA

Nous allons réutiliser la bibliothèque d'automates présentée lors du TP 3. N'oubliez donc pas le préambule :

```
from thlr_automata import *
```

Les automates finis non-déterministes (NFA) sont modélisés par la classe `NFA` introduite au TP 4. La classe `NFA` est similaire dans sa manipulation et son initialisation à la classe `ENFA` si ce n'est que l'on interdit l'emploi de la lettre `""` représentant ε. Par exemple, l'automate `A` de la figure 1 est un NFA.

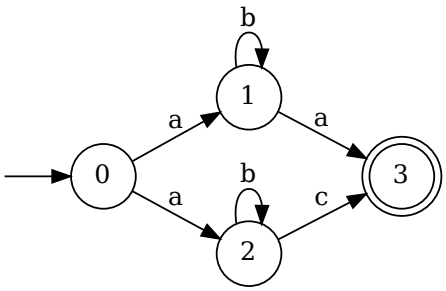

Figure 1: Un automate non-déterministe `A`.

Question 1. Définissez grâce à la classe `NFA` puis exportez une représentation graphique de l'automate `A` de la figure 1.

1.2 Propriétés d'un NFA

Un NFA est dit *complet* (resp. *déterministe*) si pour toute lettre a et état q il existe au moins (resp. au plus) une transition sortant de q étiquetée par a. L'automate `A` de la figure 1 n'est ainsi ni complet, ni déterministe.

Question 2 ($\star$). Écrivez une fonction `is_complete(nfa)` à la classe `NFA` qui renvoie `True` si l'automate est complet et `False` sinon.

Question 3 ($\star$). Écrivez une fonction `is_deterministic(nfa)` à la classe `NFA` qui renvoie `True` si l'automate est déterministe et `False` sinon.

1.3 Acceptation d'un mot par un NFA

Un état q est *accessible* en lisant un mot w dans un NFA $\mathcal{A}$ depuis un ensemble E d'états de $\mathcal{A}$ s'il y a un état $q' \in E$ tel qu'il existe dans $\mathcal{A}$ une chemin de q' à q étiqueté par w. On dit que $\mathcal{A}$ *accepte* w depuis E s'il existe un état final accessible depuis l'ensemble des états E en lisant w ; si $E = I$, l'ensemble des états initiaux de $\mathcal{A}$, alors on dit simplement que $\mathcal{A}$ accepte w.

Question 4 ($\star$). Écrivez une fonction `get_reachable_states(nfa, origins, letter)` qui renvoie l'ensemble des états accessibles dans l'automate `nfa` depuis l'ensemble `origins` par la lettre `letter`. Attention, il doit bien s'agir d'un ensemble et non d'une liste.

Question 5 ($\star$). Écrivez une fonction `accepts_from(nfa, current_states, word)` qui renvoie `True` si le `NFA nfa` accepte le mot `word` depuis l'ensemble d'états `current_states`, et `False` sinon. Notez que l'on peut utiliser une fonction récursive : si le mot `word` est vide, alors il suffit de voir si `current_states` contient un état final ; sinon, on calcule les successeurs de `current_states` par la lettre `word[0]` et on vérifie si le mot `word[1:]` est accepté depuis ces derniers.

Question 6 ($\star$). Écrivez une fonction `accepts(nfa, word)` qui renvoie `True` si le `NFA nfa` accepte le mot `word` et `False` sinon.

2 Déterminisation d'un automate

L'objectif final de ce TP est d'implémenter l'algorithme canonique de *déterminisation* d'un NFA $\mathcal{A}$, c'est-à-dire de calculer un NFA déterministe complet $\mathcal{A}'$ de même langage. Cette construction par sous-ensembles consiste à définir un ensemble Q' d'états de $\mathcal{A}'$ isomorphe aux parties 2^Q de l'ensemble d'états Q de $\mathcal{A}$. La notion de *successeur* est alors étendue aux ensembles d'états : le successeur d'un ensemble d'états $E \subseteq Q$ selon la lettre a correspond à l'ensemble des états accessibles depuis un état de E par la lettre a.

L'algorithme de déterminisation commence par explorer l'ensemble des états initiaux $I \subseteq Q$ de l'automate originel $\mathcal{A}$, c'est-à-dire détermine les successeurs de I selon les différentes lettres de l'alphabet, puis explore ces successeurs à leur tour jusqu'à ne plus avoir de nouvel ensemble à explorer. Les états de l'automate déterminisé sont alors isomorphes à ces sous-ensembles explorés ; l'unique état initial de $\mathcal{A}'$ correspond à I ; sont finaux les états de Q' qui correspondent à un ensemble d'états dans 2^Q contenant au moins un état final de l'automate originel $\mathcal{A}$. L'application de cet algorithme à l'automate `A` de la figure 1 produit l'automate déterministe `D` de la figure 2.

Pour vous faciliter la tâche, une fonction `symbolic_nfa_builder` permet de manipuler des ensembles d'états comme s'il s'agissait d'états pour créer un nouvel automate. Plus précisément, cette fonction `symbolic_nfa_builder(all_states, initial_states, final_states, alphabet, edges)` prend en argument une liste d'ensembles représentant tous les états du déterminisé `all_states`, une liste d'ensembles représentant les états initiaux `initial_states`, une liste d'ensembles représentant les états finaux `final_states`, une liste de chaînes `alphabet`, et un liste d'arêtes `edges` constitué de triplets de la forme `({0, 1}, "a", {1, 2})`. Ainsi, la suite d'instructions suivantes permet d'obtenir l'automate `B` de la figure 3 :

```
B = symbolic_nfa_builder([{0, 1}, {1, 2}], [{0, 1}], [{1, 2}], \
  ["a", "b"], [({0, 1}, "a", {1, 2}), ({1, 2}, "b", {1, 2})])
B.export("B")
```

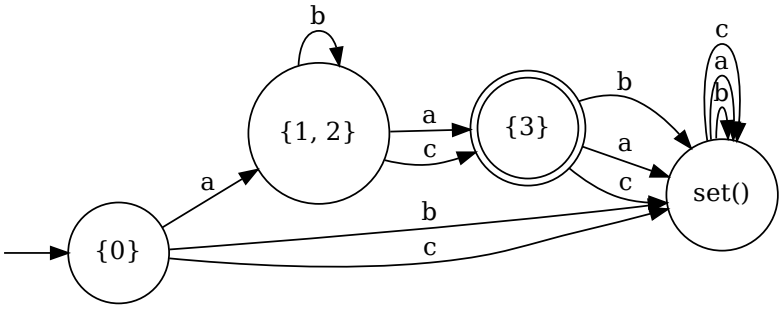

Figure 2: Le déterminisé `D` de l'automate `A`.

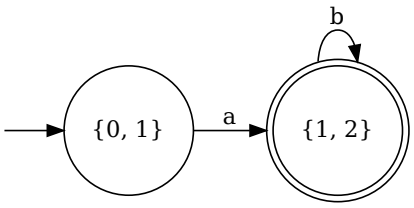

Figure 3: L'automate `B`.

Question 7 ($\star$). Écrivez une fonction `determinize(nfa)` qui renvoie le déterminisé du NFA `nfa`. La construction du déterminisé peut utiliser la fonction `symbolic_nfa_builder`. Il vous faudra également lister les différents sous-ensembles d'états et leurs arêtes associées par un algorithme de parcours proche de la fonction `get_accessible_states` du TP 3.