TP 2
Vcsn– 1

Version du 26 septembre 2016

Dans le cadre de ce TP, nous allons utiliser une partie de Vcsn [1]. Ce projet est une plate-forme de manipulation d'automates développée au LRDE en collaboration avec Télécom ParisTech et le Laboratoire Bordelais d'Informatique (LaBRI). Une interface avec IPython utilisant cette bibliothèque est dédiée à l'interaction avec des automates, des expressions rationnelles, etc.

La documentation de Vcsn est disponible en ligne : http://vcsn-sandbox.lrde.epita.fr/notebooks/Doc/!Read-me-first.ipynb.

Vcsn travaille sur des automates bien plus généraux que ceux de ce cours. Le concept de typage d'automate (et plus particulièrement des transitions d'un automate) se nomme « contexte » dans Vcsn. Nous utiliserons le contexte le plus simple : celui des NFAs sur l'alphabet $\{a, b, \ldots, z\}$, '`lal_char(a-z), b`'. Lorsque le support des ε-NFAs est nécessaire, Vcsn utilise alors le contexte '`lan_char(a-z), b`'.

Nous utiliserons la commande '`vcsn notebook`' qui ouvre une session interactive d'utilisation de Vcsn sous IPython. Dans cet environnement ENTER insère un saut de ligne, et SHIFT-ENTER lance l'évaluation de la cellule courante. Une fois la session interactive lancée, exécuter '`import vcsn`'.

Attention à la typographie : `automaton` doit être tapé littéralement, alors que *`automaton`* est une « méta-variable » qui désigne une expression (e.g., une variable) qui s'évalue en un automate.

Exercice 1

1. La méthode '`vcsn.context(`*`ctx`*`)`' construit un contexte à partir de sa spécification *`ctx`*, une chaîne de caractères. Définissez une variable '`b`' qui corresponde aux NFAs sur l'alphabet $\{a, b, c\}$, et affichez sa valeur.
2. La méthode '*`context`*`.expression(`*`re`*`)`' permet de construire l'objet `expression` (expression rationnelle) à partir de la syntaxe algébrique usuelle :
   — '`\z`' : le langage vide, $\emptyset$
   — '`\e`' : le mot vide, ε
   — '`a`' : le langage du mot '`a`'
   — '*`e`*`+`*`f`*' : *`e`* ou *`f`*
   — '*`ef`*' : *`e`* suivi de *`f`*
   — '*`e`*`*`' : *`e`* répété $n \geq 0$ fois
   — '`(`*`e`*`)`' : groupement

   Les priorités habituelles sont appliquées. Il existe quelques sucres syntaxiques :
   — '`[a-dmx-z]`' : '`a+b+c+d+m+x+y+z`'
   — '`[^a-dmx-z]`' : '`[efghijklnopqrstuvw]`' si l'alphabet est $\{a, \ldots, z\}$
   — '`[^]`' : '`[abcdefghijklmnopqrstuvwxyz]`' si l'alphabet est $\{a, \ldots, z\}$
   — '*`e`*`{+}`' : *`e`* répété $n \geq 1$ fois
   — '*`e`*`?`' : *`e`* optionnel
   — '*`e`*`{3,5}`' : *`e`* entre 3 et 5 fois
   — '*`e`*`{3,}`' : *`e`* au moins 3 fois
   — '*`e`*`{,5}`' : *`e`* au plus 5 fois

   Saisissez les expressions rationnelles suivantes exprimées avec la syntaxe de Perl (il faut donc les réécrire pour Vcsn). On supposera que l'alphabet est $\{a, b, c\}$.

1. http://vcsn.lrde.epita.fr

1. ‘ab*’
2. ‘ab+’
3. ‘ab*|ab+’
4. ‘abc|(bac)*|(cab)+’
5. [a-c]*[^a]
6. ‘a?bc|ab?c’

3. La méthode ‘*expression*.thompson’ permet de générer l’automate correspondant à une expression rationnelle à l’aide de l’algorithme de Thompson. Bien entendu elle produit un ε-NFA, et par conséquent, lorsque nécessaire, utilise un contexte différent pour l’automate.

   Générez l’automate de Thompson des expressions rationnelles de la question précédente.

4. La méthode ‘*automaton*.proper(*direction*="backward", *prune*=True)’ effectue l’élimination des transitions spontanées. Si l’argument *prune* est vrai (sa valeur par défaut), alors les états qui deviennent inaccessibles à cette occasion seront éliminés.

   Éliminez les transitions spontanées des automates précédents, *sans* éliminer les états devenant inaccessibles.

5. Vous pouvez émonder un automate (ce n’est pas sale) avec la méthode ‘*automaton*.trim’.

   Éliminez les états inutiles des automates précédents.

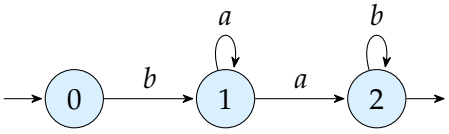
6. La méthode ‘*automaton*.format(*format*)’ permet de sérialiser un automate dans un certain format, comme ‘"daut"’.

   Lorsque Python affiche une chaîne de caractères, il n’interprète pas les retours à la ligne et affiche un résultat particulièrement illisible. N’hésitez pas à vous servir de ‘print(*expression*)’.

   Convertir un des automates précédents, et en comprendre la structure.

7. La commande ‘%%automaton’ permet de *désérialiser* un automate, i.e., de construire un automate à partir d’une description textuelle :

```
%%automaton a
context = "lal_char(abc), b"
$ -> 0
0 -> 0 [a-c]
0 -> 1 a, c
1 -> $
```



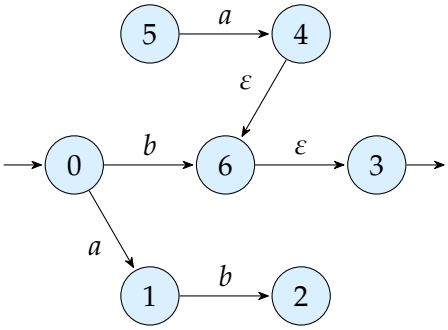
Automate 1: Un automate à saisir

   Créez l’automate 1.

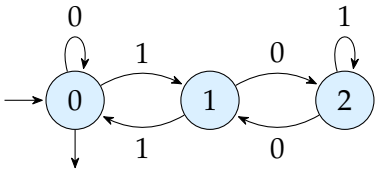
8. Émondez l’automate 2. Vérifiez que le résultat correspond bien à celui que vous attendez (c’est-à-dire l’automate privé des états qui ne sont pas co-accessibles ou pas accessibles).

   Essayez aussi les méthodes ‘*automaton*.accessible’ (pour ne garder que les états accessibles) et ‘*automaton*.coaccessible’ (pour ne garder que les états co-accessibles).

9. Construisez un automate reconnaissant les nombres *décimaux* divisibles par 3. Vous considérerez que le mot vide (‘\e’) est équivalent au nombre 0 et fait donc partie du langage reconnu par l’automate.



Automate 2: Un automate à émonder.



Automate 3: div3base2, un automate qui reconnaît les binaires divisibles par 3.

N'hésitez pas à vous inspirer de l'automate 3 qui fait la même chose sur les nombres *binaires*.
La méthode `'automaton.shortest(len=length)'` énumère tous les mots de taille $\leq$ `length` reconnus par un automate.
Vérifiez les mots de moins de trois lettres reconnus par votre automate.