TP 2
Vcsn– 1

Version du 21 septembre 2020

For this practical, we will be using Vcsn[1]. This project is a platform for the manipulation of finite automata, developped at LRDE in collaboration with Télécom ParisTech and LaBRI (Laboratoire Bordelais de Recherche en Informatique). An IPython interface is dedicated to the interactive manipulation of finite automata, rational expressions, etc.

Vcsn documentation is available online :
`http://vcsn-sandbox.lrde.epita.fr/notebooks/Doc/!Read-me-first.ipynb`.

Vcsn manipulates a much more general class of automata than the one seen in class. The concept of an automaton type (in particular transition type) is called « context » in Vcsn. We will use the simplest context : the context of NFAs on the alphabet $\{a, b, \ldots, z\}$, '`lal_char(a-z), b`'. If support for ε-NFAs is needed, Vcsn then uses the context '`lan_char(a-z), b`'.

The command '`vcsn notebook`' opens an interactive Vcsn session with IPython. In this environment ENTER inserts a newline, and SHIFT-ENTER evaluates the current cell. Once the interactive session has been launched, execute '`import vcsn`'.

Beware of typography : `automaton` must be typed in litteraly, whereas *`automaton`* is a « meta-variable » denoting an expression (e.g., a variable) that evaluates to an automaton.

Exercice 1

1. The method '`vcsn.context(`*`ctx`*`)`' builds a context from the string *`ctx`*. Define a variable '`b`' that corresponds to NFAs on the alphabet $\{a, b, c\}$ and print its value.
2. The method '*`context`*`.expression(`*`re`*`)`' builds an object `expression` (rational expression) from the usual algebraix syntax :
   — '`\z`' : the empty language, $\emptyset$
   — '`\e`' : the empty word, ε
   — '`a`' : the language of the word '`a`'
   — '`e+`*`f`*' : `e` or *`f`*
   — '*`ef`*' : `e` followed by *`f`*
   — '`e*`' : `e` repeated $n \geq 0$ times
   — '`(e)`' : grouping
   The usual priority rules apply. There is some syntactical sugar :
   — '`[a-dmx-z]`' : '`a+b+c+d+m+x+y+z`'
   — '`[^a-dmx-z]`' : '`[efghijklnopqrstuvw]`' if the alphabet is $\{a, \ldots, z\}$
   — '`[^]`' : '`[abcdefghijklmnopqrstuvwxyz]`' if the alphabet is $\{a, \ldots, z\}$
   — '`e{+}`' : `e` repeated $n \geq 1$ times
   — '`e?`' : optionally `e`
   — '`e{3,5}`' : `e` between 3 and 5 times
   — '`e{3,}`' : `e` at least 3 times
   — '`e{,5}`' : `e` at most 5 times
   Rewrite the following Perl rational expressions in Vcsn syntac). We assume that the alphabet is $\{a, b, c\}$.
   1. '`ab*`'

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

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

3. The method *‘expression.thompson’* generates the automaton corresponding to a rational expression, as per Thompson algorithm. Of course, it produces an ε-NFA, and may thus use a different context for the automaton.

   Compute the Thomsom automata of the rational expressions from the previous question.

4. The method *‘automaton.proper(direction="backward", prune=True)’* removes ε-transitions. If the argument *prune* is true (which is the default), then states that become unreachable will be deleted.

   Remove ε-transitions from the previous automata, but *do not* remove unreachable states.

5. *‘automaton.trim’* trims an automaton.

   Trim the useless states in the previous automata.

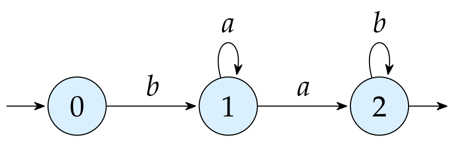
6. The method *‘automaton.format(format)’* serializes an automaton to a given format, such as ‘"daut"’.

   When Python prints a string, it does not interpret newline characters and the result is often unreadable. Use ‘print(*expression*)’.

   Convert one of the previous automata, and try to understand its structure.

7. The command ‘%%automaton’ *deserializes* an automaton, i.e. it builds an automaton from a textual description :

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



Automate 1: An automaton to type in

   Build the automaton automate 1.

8. Trim the automaton automate 2. Check that the result fits your expectations (i.e. the automaton from which states that are neither accessible nor co-accessible have been removed).

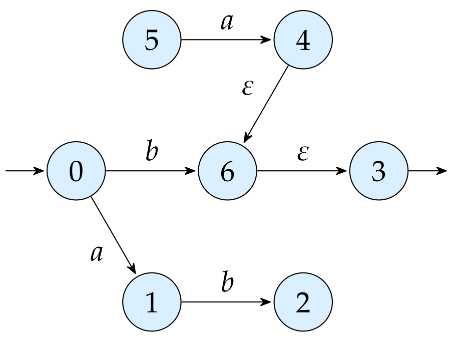
   Also try the method *‘automaton.accessible’* (to keep accessible states only) and *‘automaton.coaccessible’* (to keep co-accessible states only).

9. Build an automaton that recognizes all multiples of 3, written in base 10. Assume that the empty word (‘\e’) represents the number 0 and is thus part of the language.

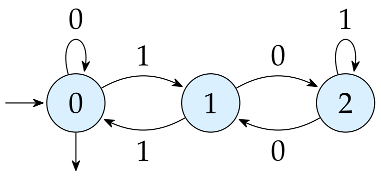
   You can draw your inspiration from automate 3 that does the same thing on numbers written in binary.

   The method *‘automaton.shortest(len=length)’* enumerates all words of size $\leq$ *length* recognized by an automaton.

   Check the words of less than 3 letters recognized by your automaton.



Automate 2: An automaton to trim.



Automate 3: div3base2, an automaton that recognizes all multiples of 3 written in binary.