Searching for regular expressions

We have seen in class algorithms to search for a fixed substring in a text. However in many applications one would rather like to have a broader search (look for a pattern of a certain form in a given text), for instance check if a text portion is in a correct format (phone number, date, IP address), extract lines with html tags, look for certain patterns in DNA sequences, etc. This type of request can often be formulated as the question of determining if a text matches a regular expression. In this tutorial we will implement an efficient algorithm for this task, based on the construction of a (non-deterministic) finite automaton associated to the given pattern (regular expression), and feeding the text to the automaton to check if it matches the pattern.

You have to download the 3 files : `NFA.py` (class for non-deterministic automata, where you will complete methods), `DG.py` (class for directed graphs, with already completed methods), and a test file `test.py`.

1 Regular expressions

In this tutorial the alphabet for input texts will be $\Sigma = \{$`a,...,z`$\}$. A regular expression E is here a certain type of string on the alphabet $\Sigma' = \Sigma \cup \{\text{special symbols}\}$ where the special symbols are `'.','*','(',')','|'`. A regular expression defines a pattern that encodes several words at once. This set of words (on Σ) is called the language of E and denoted $\mathcal{L}[E]$; we say that these words are *recognized* by E (we can also say that these words match E). Let us start informally with a few examples in the table below :

regular expression	recognized words
$(a\|d)(c)^*$	words starting with a or d, followed by a sequence (maybe empty) of c's
$..f.$	words of length 4 with f in the 3rd position
$ca(.)^*de$	words having ca as a prefix, and de as a suffix
$(a\|b)^*$	words with no other letter than a or b
$efrqs$	the word $efrqs$

We now give the recursive definition of a regular expression, and of the associated language. A regular expression is either :

- a single letter $\lambda \in \Sigma$, then it recognizes just the one-letter word λ.
- the single point `'.'`, then it recognizes the 26 words of one letter.
- of the form E_1E_2, the concatenation of two regular expressions E_1 and E_2, then it recognizes all words w that factor as $w = w_1w_2$, where w_1 is recognized by E_1 and w_2 is recognized by E_2.
- of the form $(E_1|E_2)$, the union of two regular expressions E_1 and E_2, then it recognizes all words recognized by E_1 or E_2, i.e., the union of these languages.
- of the form $(E)^*$, the star-closure of the regular expression E, then it recognizes all words of the form $w = w_1 \cdots w_k$, where $k \geq 0$ and each w_i is recognized by E (note that the empty word is recognized, case $k = 0$).

Each instance of the class `NFA` stores a regular expression, in the parameter `self.s`, and the attribute `self.m` gives its length (for $i \in [0..,m-1]$, `self.s[i]` gives access to the ith character of `self.s`). The attributes `self.rp` is a list of length `self.m`, such that for $i \in [0..m-1]$, if there is an opening parenthesis at position i in `self.s`, then `self.rp[i]` has to contain the position of the matching closing parenthesis ; and otherwise `self.rp[i]` contains -1. Similarly, The attributes `self.lp` is such that for $i \in [0..m-1]$, if there is a closing parentheses at position i in `self.s`, then `self.lp[i]` has to contain the position of the matching opening parenthesis ; and otherwise `self.lp[i]` contains -1.

self.s	(	a	)	∗	b	(	c	|	(	f	|	(	e	|	f	)	)	∗	)	∗	a
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
self.lp	−1	−1	0	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	11	8	−1	5	−1	−1
self.rp	2	−1	−1	−1	−1	18	−1	−1	16	−1	−1	15	−1	−1	−1	−1	−1	−1	−1	−1	−1

Question 1. Complete the method `left_right_match(self)` that has to correctly assign values in the attributes `self.lp` and `self.rp`, according to the description above (you will need a stack, in Python one can have a stack as a list and using the methods `pop` and `append`). To test your method execute `test1()`.

We will actually need a bit more (in the next section). Note that each symbol `'|'` sits between a matching opening parenthesis (to its left) and a matching closing parenthesis (to its right). If there is a symbol `'|'` at position i in `self.s`, we would like `self.rp[i]` to contain the position of the matching closing parenthesis, and `self.lp[i]` to contain the position of the matching opening parenthesis.

self.s	(	a	)	∗	b	(	c	|	(	f	|	(	e	|	f	)	)	∗	)	∗	a
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
self.lp	−1	−1	0	−1	−1	−1	−1	5	−1	−1	8	−1	−1	11	−1	11	8	−1	5	−1	−1
self.rp	2	−1	−1	−1	−1	18	−1	18	16	−1	16	15	−1	15	−1	−1	−1	−1	−1	−1	−1

Question 2. Complete the method `left_right_match_or(self)` that has to correctly assign values in the attributes `self.lp` and `self.rp`, including assignments related to the `'|'` symbols. To test your method execute `test2()`.

2 Building the NFA for a regular expression

An NFA (non-deterministic finite automaton) is a generalization of the concept of DFA (deterministic finite automaton). The differences are in two extensions (here we will only have the second one) :

— there can be several links labelled by a same letter starting from a given state,

— another type of link, so-called ϵ-links, is allowed, those links do not consume any letter of the input text.

For a directed path from the initial state to the accepting state, the consumed word is the word read along the path (the ϵ-links on the path do not contribute to the consumed word). A word w is said to be recognized by the automaton if there exists at least one path from the initial state to the accepting state whose consumed word is w (as opposed to a DFA the path may not be unique). The language of the NFA is defined as the set of words that are recognized by the automaton.

We now describe a general construction, due to Thompson, to build an NFA $\mathcal{A}[E]$ for any given regular expression E, such that the language recognized by $\mathcal{A}[E]$ is $\mathcal{L}[E]$. The construction can first be given recursively, as for the definition of regular expressions ; it is shown in Figure 1 (see also Figure 2 for an example on a given regular expression). Note that, if E has length m, then the NFA has $m+1$ states that are naturally aligned from left to right, so that the first m states match the symbols of E ordered from left to right, with the leftmost state as initial state (surrounded state in the figures), and where the last state (the accepting state, labelled capital F) has just one ingoing link, from the preceding state (which is the state for the last symbol of E).

Looking at Figure 1, you should convince yourself that the following properties are satisfied (each of these is proved by induction on the length of the regular expression) :

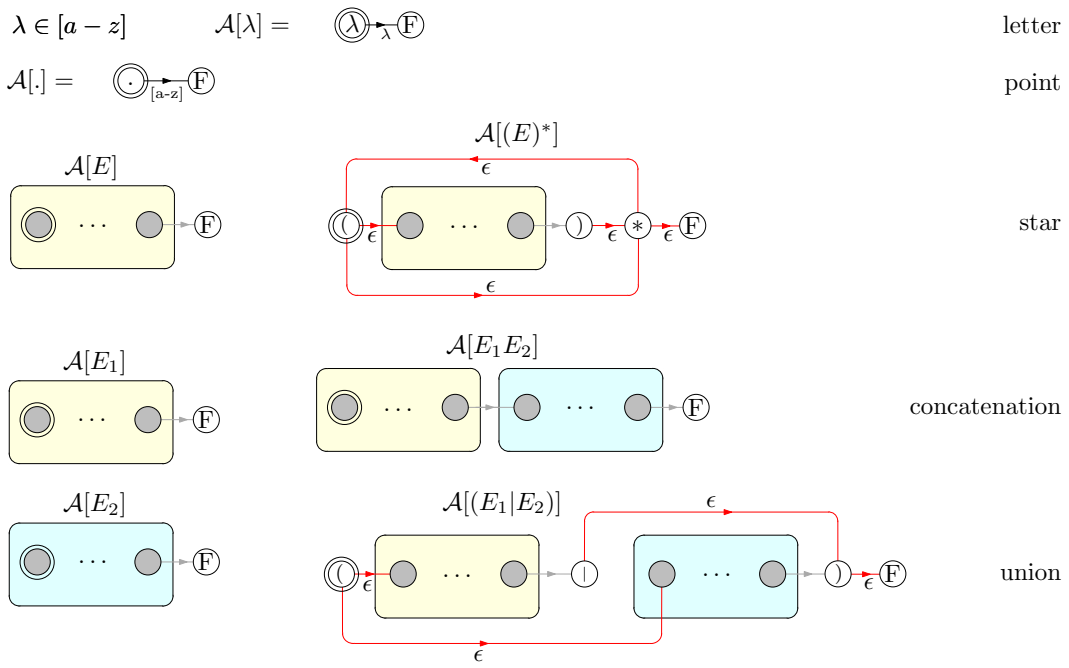

FIGURE 1 – Rules to build the NFA associated to a regular expression, for the base-cases (one-symbol expression) and the constructions (star, concatenation, union). Color rule for the links : ϵ-links are red, links with a letter are black (notation $[a-z]$ below the link in the second case means that there are in fact 26 links, one for each letter), other links whose status could be red or black (depending on the automaton) are gray.

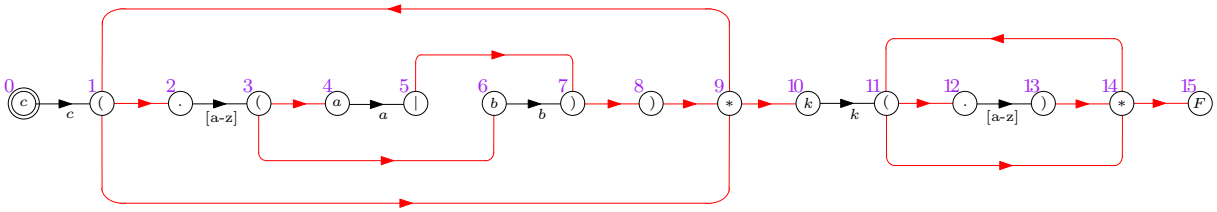

FIGURE 2 – The NFA associated to the regular expression $c(.(a|b))^*k(.)^*$ where we omit the labels on the ϵ-links (the red links).

— the language recognized by the NFA is the same as the language of words recognized by the regular expression,

— for every $i \in [0..m-1]$, there is an ϵ-link from state i to state $i+1$ iff the symbol in state i (the ith symbol of E) is in $\{$`'('`,`')'`,`'*'`$\}$,

— all the other ϵ-links are the parenthesis-links related to $\{$`'|'`,`'*'`$\}$.

— For each $\lambda \in [a-z]$, black links of letter λ are exactly those links from a state i to state $i+1$, where state i is labelled either by letter λ or by the symbol `'.'` (i.e., the ith letter of E is either λ or .).

Question 3. Complete the method `build_eps_links(self)` that constructs the ϵ-links of the NFA associated to the regular expression. These links are stored in the attribute `self.dg` (there is a method `add_link` from the class `DG`, such that `add_link(i,j)` creates a link from `i` to `j`). To test your method execute `test3()`.

Remark : We do not build the black links, since we easily know where they are (by the characterization given just before Question 3).

3 Checking if a text matches a regular expression

At first sight it seems difficult to determine if a word w is recognized by an NFA (compared to a DFA, there is not a unique way to walk along the automaton so as to be sure whether w is recognized or not). The problem can however be solved efficiently ! Let us start with the case where w is the empty word. Let K be the set of states that can be reached from the initial state using a path of ϵ-links only. Clearly the empty word is accepted iff the accepting state belongs to K. More generally, for w a word of length n, we let $w^{(i)}$ be the prefix of w of length i. For i from 0 to n we can maintain the set $K^{(i)}$ of states that can be reached from the inital state by 'consuming' $w^{(i)}$ (we have already characterized $K^{(0)}$). Indeed, for $i \geq 1$, we let $D^{(i)}$ be the set of states that can be obtained from a state in $K^{(i-1)}$ by following a black link of letter $w[i-1]$ (letter at position $i-1$ in w). Then one easily sees (take a moment to get convinced) that $K^{(i)}$ is the set of states that can be reached from a state in $D^{(i)}$ by following a path of ϵ-links.

Question 4. Complete the method `check_text(self,w)` that returns `True` if the word `w` matches the regular expression stored in `self.s`, and returns `False` otherwise (you will need the method `explore_from_subset(self,start_vertices)` in the class `DG`). Test your method for several regular expressions and words of your choice. What is the complexity order for the running time ? (in terms of the length n of w and the length m of the regular expression)

Question 5. You can then add a method `contains_pattern(s,text)` in the file `NFA.py` (at the beginning of the file, before the class), that returns `True` iff the word `text` contains a subword that satisfies the regular expression stored in `s`. Test it on examples of your choice.

Remark : In particular, if `s` contains only letters (no special symbol), the previous method checks if `s` appears as a subword of `text`, you can compare it to the KMP method seen in class, and compare experimentally its running time with those of the methods seen in TD8.

Remark : One can then add more constructions for regular expressions to formulate various constraints. (Look for instance at the functionalities of the 'grep' command of Unix, which performs pattern matching based on NFA, as in this tutorial). As an example, you can think of an implementation of the multior construction, to treat regular expressions such as $ac(d|j|t)^*ol$ (only the method `left_right_match_or` has to be updated).