Searching for regular expressions

In the lecture, we have seen algorithms that search for a fixed substring in a text. However, in many applications, one would rather like to look for a certain pattern, for instance, checking if a text portion is in a correct format (phone numbers, dates, IP addresses), extracting lines with HTML tags, or looking for specific patterns in DNA sequences. This type of request can often be formulated as determining whether a text matches a regular expression. In this TD, we implement an efficient algorithm for this task, based on the construction of a non-deterministic finite automaton (NFA), characterizing the given pattern (a regular expression), and feeding the text to the NFA in order to check if it matches the pattern.

Download the files `nfa.py`, `dg.py`, and `test.py` from Moodle. The file `nfa.py` contains a class for NFAs as well as skeleton code that you need to complete. The file `dg.py` provides a class for directed graphs and does not need to be edited. Last, the file `test.py` can be used for testing your code.

1 Regular expressions

In this TD, the alphabet for input texts is the set of all 26 lowercase letters of the Latin alphabet, denoted by $[a-z]$. We define a regular expression E to be a certain type of string over the alphabet $[a-z] \cup \{$`.`, `*`, `(`, `)`, `|`$\}$. A regular expression defines a pattern that encodes several words at once, called the *language* of E and denoted by $\mathcal{L}(E)$. We say that these words are *accepted* by E. The accepted language of a regular expression E is defined inductively as follows.

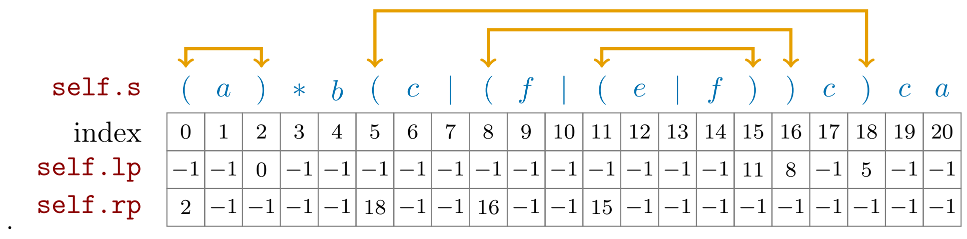
- If $E = \lambda \in [a-z]$, then E accepts exactly the one-letter word λ.
- If $E =$ `.`, then E accepts any of the 26 words of one letter.
- If $E = E_1E_2$—the concatenation of two regular expressions E_1 and E_2—, then E accepts all words $w = w_1w_2$, where w_1 is accepted by E_1 and w_2 is accepted by E_2.
- If $E =$ `(`E_1`|`E_2`)`, then E accepts all words accepted by E_1 or E_2, that is, the union of these languages.
- If $E =$ `(`E_1`)*`, then E accepts all words of the form $w = (w_i)_{i\in[k]}$ where $k \in \mathbf{N}_{\geq 0}$ and each w_i is accepted by E. Note that the empty word is recognized via the case $k = 0$.

Table 1: Some exemples of regular expressions and the words they accept.

Regular expression	Accepted words
`(a|d)(c)*`	Words starting with `a` or `d`, followed by a sequence of `cs` (maybe none)
`..f.`	Words of length 4 with `f` at the 3rd position
`ca(.)*de`	Words with `ca` as prefix and `de` as suffix
`((a|b))*`	Words with no other letter than `a` or `b`
`efrqs`	The word `efrqs`

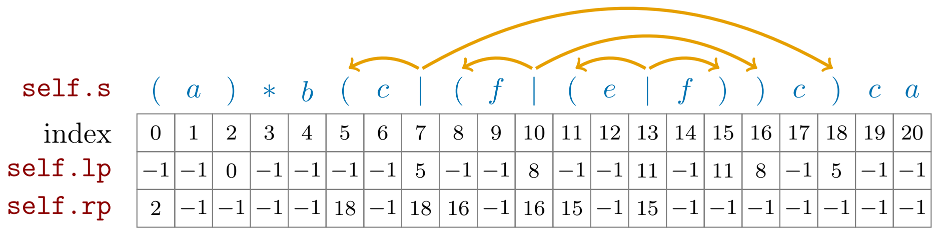
We now consider the file `nfa.py`, which defines the class `NFA` with many attributes explained in the following. Please refer to Figure 1 for an example of the different attributes.

Each instance of `NFA` stores a regular expression as a string in the attribute `self.s`, and the attribute `self.m` provides its length. The attribute `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`]` contains the position of the matching closing parenthesis. If `self.s[`i`]` is not an opening parenthesis, `self.rp[`i`]` contains -1. Similarly, the attributes `self.lp` is such that for $i \in [0..m-1]$, if there is a closing



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

Figure 1: An example of the different attributes of an instance of the class `NFA`.



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

Figure 2: The augmented lists `self.lp` and `self.rp`, which also store indices for each `|`.

parenthesis at position i in `self.s`, then `self.lp[`i`]` has to contain the position of the matching opening parenthesis. Otherwise, it contains -1. Please note that we augment these lists during this TD.

The class `NFA` further contains an `__init__` method that takes a string of a regular expression and, for the moment, initializes the lists `lp` and `rp` with `-1`s. Last, the class implements the method `__str__`, which prints the regular expression as well as the content of the lists `lp` and `rp`, useful for debugging purposes. We will always assume that the regular expressions that we consider are well-formed.

Question 1. Complete the method `left_right_match(self)`, which assigns the correct values in the attributes `self.lp` and `self.rp`, according to the description above. To this end, it is useful to use a stack, which can be easily simulated in Python by using a list and its methods `pop` and `append`.

When constructing an NFA in the next section, we need a bit more information than just the indices of matching parentheses. Note that each `|` sits between a matching opening parenthesis (to its left) and a matching closing parenthesis (to its right). If there is a `|` 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. Please refer to Figure 2 for an example.

Question 2. Complete the method `left_right_match_or(self)`, which assigns the correct values to the attributes `self.lp` and `self.rp`, including assignments related to `|`s.

2 Building the NFA for a regular expression

An NFA is a generalization of a deterministic finite automaton (DFA) with respect to their transitions—NFAs are still exactly as powerful as DFAs. The differences are in the following two extensions, one of which is relevant for us:

- there can be several links labeled by a same letter starting from a given state, and
- another type of links, so-called ε-links, is allowed, which do not consume any letter of the input.

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 consume letters). A word w is said to be *accepted* by the automaton if and only if there exists at least one path from the initial state to the accepting state whose consumed word is w. The language of the NFA is defined as the set of words that are accepted by the automaton.

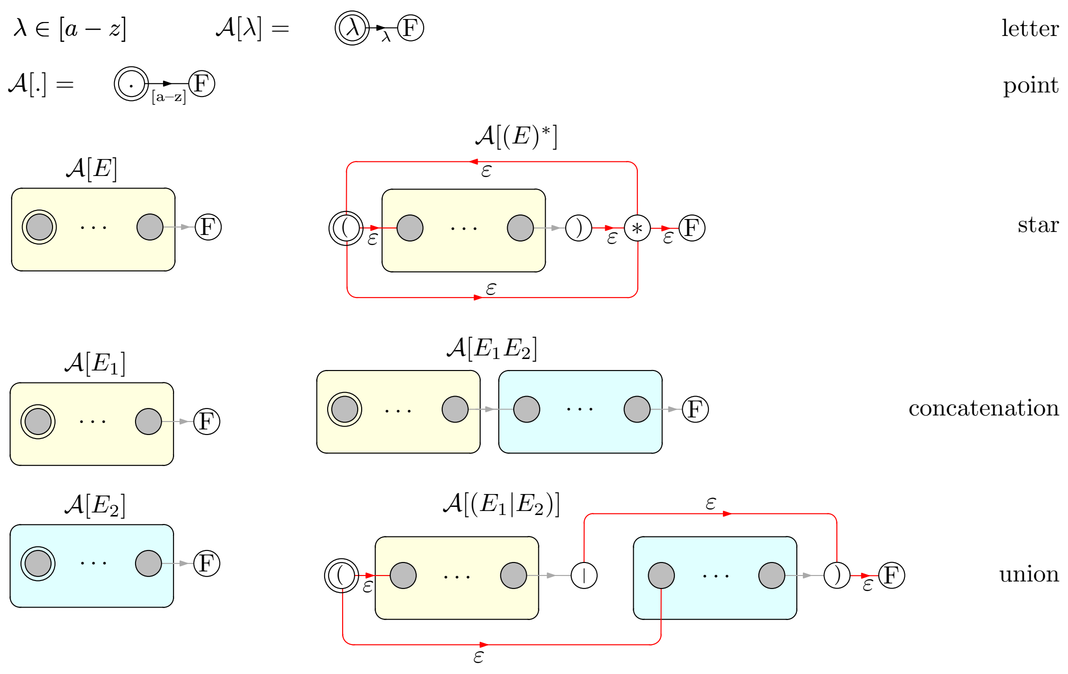


Figure 3: The rules to build the NFA associated with a regular expression, for the base cases (one-symbol expressions) and the induction steps (star, concatenation, union). The links are colored as follow: ε-links are red, links with a letter are black (the notation $[a-z]$ below the link in the second case means that there are actually 26 links, one for each letter), other links, whose status could be red or black (depending on the automaton), are gray.

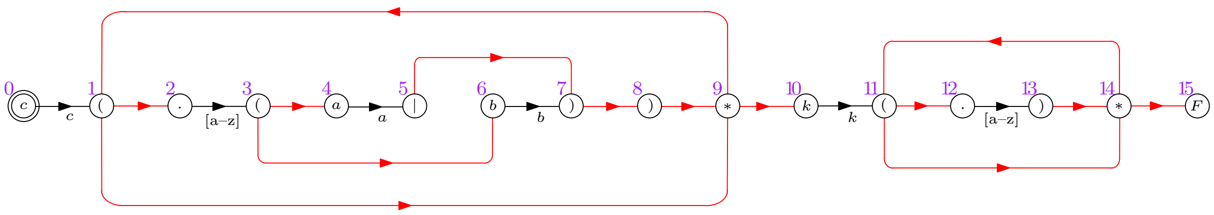


Figure 4: The NFA associated with the regular expression `c(.(a|b))*k(.)*`, where we omit the labels on the ε-links (the red links).

We describe a general construction, due to Thompson, for building an NFA $\mathcal{A}[E]$ for any given regular expression E such that the language accepted by $\mathcal{A}[E]$ is $\mathcal{L}(E)$. We define the construction inductively, as for the definition of regular expressions. Figure 3 illustrates this definition, and Figure 4 provides an example of an NFA. 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).

Referring to Figure 3, you should convince yourself that the following properties are satisfied (each of these is proven by induction on the length of the regular expression):

- The language recognized by an NFA is the same as the language of words recognized by the respective regular expression.
- For every $i \in [0..m-1]$, there is an ε-link from state i to state $i+1$ if and only if 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 $\lambda \in [a\text{–}z]$, black links of letter λ are exactly those links from a state i to state $i+1$ where the symbol in state i is in $\{\lambda, \texttt{.}\}$.

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

Remark: We do *not* build the black links, since we easily know where they are.

3 Checking if a text matches a regular expression

At first sight, it might seem difficult to determine if a word w is accepted 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 accepted or not. However, the problem can be solved efficiently!

We start with the case where w is the empty word. Let $K^{(0)}$ 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 if and only if the accepting state belongs to $K^{(0)}$. More generally, for a word w of length n, we let $w^{(i)}$ be the prefix of w of length i. For i from 0 to n, we maintain the set $K^{(i)}$ of states that are reachable from the initial state by consuming $w^{(i)}$. For $i \geq 1$, we let $D^{(i)}$ be the set of states that are obtained from a state in $K^{(i-1)}$ by following a black link that uses letter $w[i-1]$. Note that $K^{(i)}$ is then the set of states that are reachable from a state in $D^{(i)}$ by following a path of ε-links.

Example. Given the regular expression `c(.(a|b))*k(.)*` (see Figure 4) and the word `ck`, we have that $K^{(0)} = \{0\}$, $D^{(1)} = \{1\}$, $K^{(1)} = \{1, 2, 9, 10\}$, $D^{(2)} = \{3, 11\}$, $K^{(2)} = \{4, 6, 11, 12, 14, 15\}$. As `ck` has two letters and $K^{(2)}$ contains the final state 15, this word belongs to the language of the regular expression `c(.(a|b))*k(.)*`.

Question 4. Complete the method `check_text(self, w)`, which returns `True` if the word `w` matches the regular expression `self.s`, and it returns `False` otherwise. To this end, make use of the method `explore_from_subset(self, start_vertices)` (of `self.dg`), which takes a list of indices of `w` and returns a list of indices of all states that are reachable via ε-links. What is the complexity order of the run time in terms of the length n of `w` and the length m of the regular expression?

Question 5. Complete the method `contains_pattern(s, text)` at the top of `NFA.py`, which returns `True` if and only if the word `text` contains a subword that satisfies the regular expression stored in `s`. You should create only one instance of `NFA` and call its method `check_text()` only once.
Hint: Modify your input regular expression.

Remark: For `contains_pattern(s, text)`, if `s` contains only letters (no special symbols), the method checks if `s` appears as a subword of `text`. You can compare it to the KMP method seen in the lecture and compare experimentally its run time with those of the methods seen in TD6.

Further remark: One can add more constructions for regular expressions in order to formulate various constraints. Look for instance at the functionalities of the `grep` command in Unix, which performs pattern matching based on NFAs, as in this TD. As an example, you can think of an implementation of the `multior` construction for treating regular expressions such as `ac(d|j|t)*ol` (only the method `left_right_match_or` needs to be updated).