Faculty of Computer Science, Dalhousie University *11-Sep-2024*

CSCI 4152/6509 — Natural Language Processing

Lecture 3: Finite Automata Review

Location: Carleton Tupper Building Theatre C   Instructor: Vlado Keselj
Time: 16:05 – 17:25

Previous Lecture

- Ambiguities at different levels of NLP
- About course project
  - Deliverables: P0, P1, P, R
  - Project report structure
  - Choosing project topic

Part II

Stream-based Text Processing

Slide notes:

- Considering text as a stream of characters, words, and lines of text
- Review of Finite Automata and Regular Expressions
- Review of Unix-style text processing
- Introduction to Perl
- Morphology fundamentals
- N-grams
- Reading: Chapter 2, Jurafsky and Martin

In this part, we will consider language and text as a stream of characters, words, and lines of text, and look into some processing models that are applicable in this environment. We will first refresh out knowledge about finite automata and regular expressions, and some common Unix-based tools that can be used for basic text processing. We will then introduce the Perl programming language as an extension of these tools. Following this, we will introduce elements of morphology, and introduce character and word n-grams.

4 Review of Automata and Regular Expressions

4.1 Finite-State Automata

Finite-State Automata

- Regular Expressions and Regular Languages
- Regular Languages can be described using
  - Regular Expressions
  - Regular Grammars
  - Finite-State Automata (DFA and NFA)

- DFA = Deterministic Finite Automaton
- NFA = Non-deterministic Finite Automaton
- also referred to as Finite-State Machines

Typical Low-level NLP Tasks

- Pre-processing text
- Tokenization
- Sentence Segmentation
- Morphological Processing (e.g., Stemming)
- "Vectorizing" Text
- Information Extraction (simpler cases)
- *and so on*

Example Task: Removing HTML Tags

Deterministic Finite Automaton

- Formally defined as a 5-tuple: $(Q, \Sigma, \delta, q_0, F)$
  - Q is a set of states
  - Σ is an input alphabet
  - $\delta : Q \times \Sigma \to Q$ is a transition function
  - $q_0 \in Q$ is the start state
  - $F \subset Q$ is a set of final or accepting states
- Graph representation is frequently used
- Consider finite automata for sets of strings:
  `baaa...a!  ha-ha-...-ha  up-up-down-up-down-up-up-...down`

You should notice that we define a DFA (Deterministic Finite Automaton) in such way that for each state and each character from the input alphabet, there is exactly one transition to another state. In some definitions of DFA, such as the one given in the book, this is relaxed by having at most one transition, and if a transition does not exist, it is assumed to lead to a non-listed "dead" state. We are going to explicitly show this "dead" state in our examples.

Representing DFA

- Formally, as sets and functions (mappings)
- As a transition table
- As a graph
- Consider the DFA for the language: `baaa...a!`

Non-deterministic Finite Automaton

- Formally: $(Q, \Sigma, \delta, q_0, F)$
- However, the transition function is different: $\delta : Q \times \Sigma_\varepsilon \to P(Q)$
  where $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$, and $P(Q)$ is the set of all subsets of Q (powerset)
- A string is accepted if there is $\underline{\text{at least}}$ one path leading to an accepting state
- Consider: `/.*ing/` or `/jan|jun|jul/`

Another NFA and DFA Example

- Write a DFA that accepts any sequence over alphabet $\Sigma = \{\texttt{a}, \texttt{b}, \ldots, \texttt{z}\}$ that ends with '`eses`', like 'theses' or 'parentheses'.
- Write an NFA that accepts the same language.

You can consider more examples, such as HTML tags and comments.

Implementing NFAs

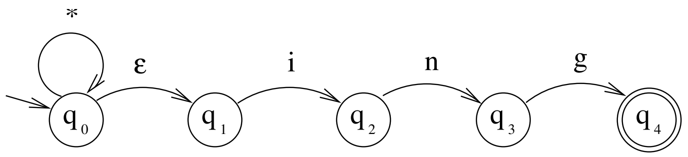
- DFA — easy to implement, NFA — not straightforward
- Two approaches for NFA: backtracking and translation to DFA
- Using backtracking — usually inefficient solution
- Translating into a DFA
  - Sets of reachable NFA states become states of new DFA

NFA to DFA Translation

- Start with NFA and create new equivalent DFA
- DFA states are sets of NFA states
- If q_0 is the start NFA state, then the start DFA state is **Closure**(q_0)
- **Closure**(A) of a set of NFA states A is a set A with all states reachable via ε-transitions from A
- Fill DFA transition table by keeping track of all states reachable after reading next input character
- Final states in DFA are all sets that contain at least one final state from NFA

Example of Translating an NFA to DFA

Let us consider a language of words consisting of all lowercase English letters ending with 'ing', i.e., of all words that can be described with the regular expression `/.*ing/`. These words would be recognized by the following NFA:



In the NFA shown above, we use ε to denote an ε-transition, i.e., a transition that happens without reading any input symbol. The asterisk symbol ('`*`') is used to denote all letters. The shown NFA illustrates how we can conveniently express this language with an NFA. An issue is how to implement an NFA as they are more difficult to implement than DFAs. One solution would be to use backtracking to explore all possible states while reading a string. This is not so easy to implement and it may also be a very inefficient solution depending on an NFA. A more efficient solution usually is to translate this NFA into a DFA. Let us see how we can do this.

The states of an DFA produced from an NFA are sets of states from the NFA. The start state of the DFA is a set containing the start state of the NFA. In this case, it is the state q_0. We also add all states in the so-called 'closure' of q_0, which are all states that can be reached from q_0 via ε-transitions. In our particular case, it implies that q_1 is also a state in the closure of q_0. Hence, the start state of the DFA is the set $\{q_0, q_1\}$. We can start the DFA transition table as follows:

State	`i`	`n`	`g`	other letters (not `i`, `n`, or `g`)
$\rightarrow \{q_0, q_1\}$				

We now look at the states q_0 and q_1 and look at the states reachable from these states after reading '`i`'. These states are q_0, because it is reachable from q_0 via '`*` transition, and the state q_2, which is reachable from q_1. So the states reachable from $\{q_0, q_1\}$ after reading '`i`' character are the states $\{q_0, q_2\}$. We also need to include closure of this set, i.e., all states reachable via ε transitions from these states, so we finally get the set $\{q_0, q_1, q_2\}$. That is how we fill the first entry of the transition table:

State	`i`	`n`	`g`	other letters (not `i`, `n`, or `g`)
$\rightarrow \{q_0, q_1\}$	$\{q_0, q_1, q_2\}$			

We fill the rest of the row of the table in a similar way. These entries are less interesting since the corresponding letters lead only to the state q_0, and we add its closure to get $\{q_0, q_1\}$:

State	`i`	`n`	`g`	other letters (not `i`, `n`, or `g`)
$\rightarrow \{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$

After finishing a row in the transition table, we need to check that all added table entries, which are sets of NFA states, appear in the first column of the transition table. We can notice that $\{q_0, q_1\}$ is in included in the column, while we have one new set $\{q_0, q_1, q_2\}$, which needs to be added to the table, in the first column. We may sometimes encounter more of these sets and we need to add them all. After adding this new set to the first column, we get the table:

State	`i`	`n`	`g`	other letters (not `i`, `n`, or `g`)
$\rightarrow \{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$
$\{q_0, q_1, q_2\}$				

We repeat the process with the new row of the table and obtain:

State	`i`	`n`	`g`	other letters (not `i`, `n`, or `g`)
$\rightarrow \{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$
$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_3\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$

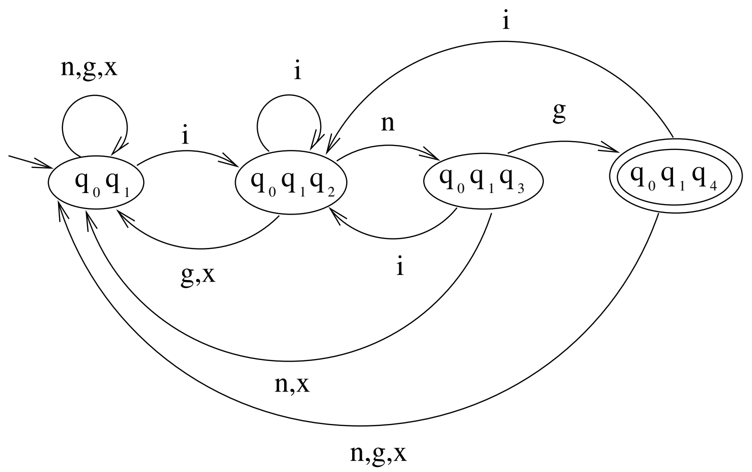
We now add any new sets of states to the first column and repeat the process until now new sets of states appear. We end up with the following table:

State	`i`	`n`	`g`	other letters (not `i`, `n`, or `g`)
$\rightarrow \{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$
$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_3\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$
$\{q_0, q_1, q_3\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_1, q_4\}$	$\{q_0, q_1\}$
$\{q_0, q_1, q_4\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$

We now go through the first column and we mark all sets of states that include at least one final state from the NFA as final states in the DFA. The NFA had only one final state q_4, so we mark only the state $\{q_0, q_1, q_4\}$ as the final state in DFA, and finally obtain the following transition table:

State	`i`	`n`	`g`	other letters (not `i`, `n`, or `g`)
$\rightarrow \{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$
$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_3\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$
$\{q_0, q_1, q_3\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_1, q_4\}$	$\{q_0, q_1\}$
F: $\{q_0, q_1, q_4\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$

If we want to represent this DFA as a graph, we would obtain:



It the above DFA we used ‘x’ to represent any letter other than ‘i’, ‘n’, or ‘g’.

It is now relatively easy to implement DFA in any programming language, and it would be a very efficient solution. A possible issue with this approach is that the new DFA could have exponentially more states that the original NFA. One solution would be to apply DFA minimization, which is a well-known procedure that we will not study here in any detail. Another approach is not to represent the DFA explicitly, as we did, but to implement NFA by keeping track of all reachable states after reading input, and whether any final state is included in this set of reachable states at the end.