Faculty of Computer Science, Dalhousie University *2-Oct-2025*

CSCI 4152/6509 — Natural Language Processing

Lecture 3: Finite Automata Review

Location: Studley LSC-Psychology P5260 Instructor: Vlado Keselj
Time: 14:35 – 15:55

Previous Lecture

- Why is NLP hard?
  - ambiguous, vague, universal
- 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**
- Finite state automata (start)

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 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 $\Sigma = \{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$ 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

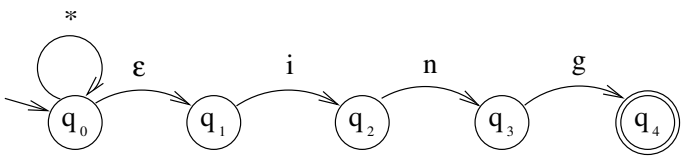
- Translating into a DFA
  - Sets of reachable NFA states become states of new DFA

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)
$\rightarrow \{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$
$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_3\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$

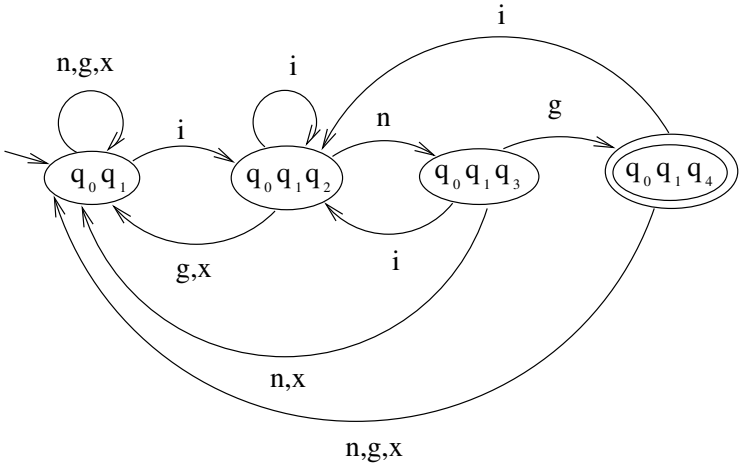
We now add any new sets of states to the first column and repeat the process until now new sets of states appear. We end up with the following table:

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.

Finite Automata in NLP

Slide notes:

Finite Automata in NLP

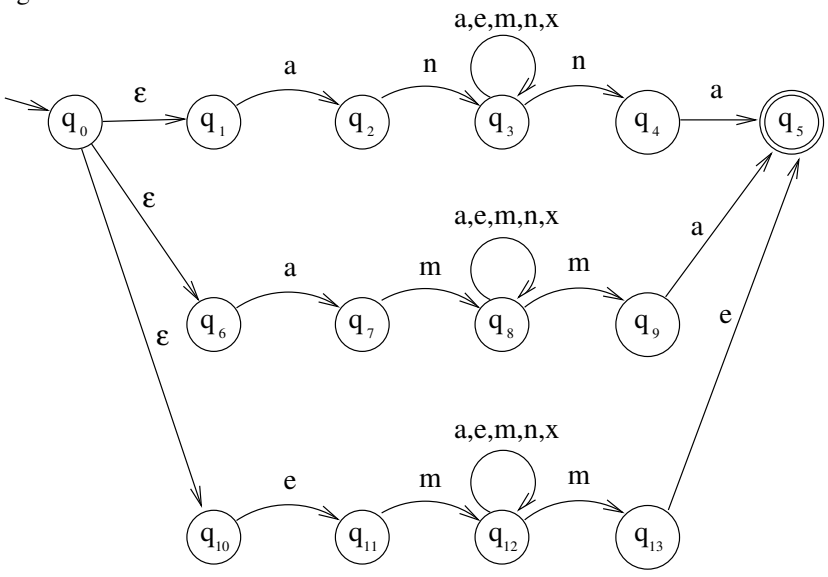
- Useful in data preprocessing, cleaning, transformation and similar low-level operations on text
- Useful in preprocessing and data preparation
- Efficient and easy to implement
- Regular Expressions are equivalent to automata
- Used in Morphology, Named Entity Recognition, and some other NLP sub-areas

Finite Automata—DFA, NFA, and their various variations—are very useful models for describing different kinds of language and string processing. They are easy to implement and they run efficiently. They are a frequently used model in morphological processing and in describing morphological processes of a language.

In addition to being useful as direct models for many NLP tasks, they are also an excellent way to implement regular expressions. Regular expressions are a convenient way of representing language patterns, but it is not obvious how to implement them in an algorithm. A solution is to translate a regular expession to NFA, then translate NFA to DFA, and then implement a DFA. All these steps are well-understood and straightforward.

Similarly to regular expressions, automata are useful in many text preprocessing tasks, such as text filtering, cleaning, transformation, and similar.

Exercise Problem: Let us suppose that we want to implement a recognizer of all words that either:
1) start with ‘an’ and end with ‘na’,
2) start with ‘am’ and end with ‘ma’, or
3) start with ‘em’ and end with ‘me’. We will denote any other letters with ‘x’. An easy way to describe these words is with the following NFA:



Translate this NFA into a DFA.

Answer:

State	a	e	m	n	x
$\rightarrow \{q_0, q_1, q_6, q_{10}\}$	$\{q_2, q_7\}$	$\{q_{11}\}$	$\emptyset$	$\emptyset$	$\emptyset$
$\{q_2, q_7\}$	$\emptyset$	$\emptyset$	$\{q_8\}$	$\{q_3\}$	$\emptyset$
$\{q_{11}\}$	$\emptyset$	$\emptyset$	$\{q_{12}\}$	$\emptyset$	$\emptyset$
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\{q_8\}$	$\{q_8\}$	$\{q_8\}$	$\{q_8, q_9\}$	$\{q_8\}$	$\{q_8\}$
$\{q_3\}$	$\{q_3\}$	$\{q_3\}$	$\{q_3\}$	$\{q_3, q_4\}$	$\{q_3\}$
$\{q_{12}\}$	$\{q_{12}\}$	$\{q_{12}\}$	$\{q_{12}, q_{13}\}$	$\{q_{12}\}$	$\{q_{12}\}$
$\{q_8, q_9\}$	$\{q_5, q_8\}$	$\{q_8\}$	$\{q_8, q_9\}$	$\{q_8\}$	$\{q_8\}$
$\{q_3, q_4\}$	$\{q_3, q_5\}$	$\{q_3\}$	$\{q_3\}$	$\{q_3, q_4\}$	$\{q_3\}$
$\{q_{12}, q_{13}\}$	$\{q_{12}\}$	$\{q_5, q_{12}\}$	$\{q_{12}, q_{13}\}$	$\{q_{12}\}$	$\{q_{12}\}$
F: $\{q_5, q_8\}$	$\{q_8\}$	$\{q_8\}$	$\{q_8, q_9\}$	$\{q_8\}$	$\{q_8\}$
F: $\{q_3, q_5\}$	$\{q_3\}$	$\{q_3\}$	$\{q_3\}$	$\{q_3, q_4\}$	$\{q_3\}$
F: $\{q_5, q_{12}\}$	$\{q_{12}\}$	$\{q_{12}\}$	$\{q_{12}, q_{13}\}$	$\{q_{12}\}$	$\{q_{12}\}$

5 Regular Expressions and Perl

5.1 Regular Expressions

Regular Expressions

- Review of regular expressions (for some of you, it was covered in earlier courses as well)
- Used as patterns to match parts of text
- Equivalent to automata, although this may not be obvious
- Provide a compact, algebraic-like way of writing patterns
- Example: `/Submit (the )?file [A-Za-z.-]+/`

Slide notes:

Some References on Regular Expressions

You can find many references on Regular Expressions, including:

- Chapter 2 of the textbook [JM]
- Perl "Camel book" or many resources on Internet
- On timberlea server: '`man perlre`' and '`man perlretut`'
- The same effect: '`perldoc perlre`' and '`perldoc perlretut`'
- Or on the web: `http://perldoc.perl.org/perlre.html` and `http://perldoc.perl.org/perlretut.html`

Regular Expressions are an important topic in Computer Science, both from a theoretical perspective and as a practical tool. It is expected that it was studied by most students taking this course before, so we will do only a brief review of them here. If you are not familiar with regular expressions, there are many available resources to catch up and learn more. For example, the regular expressions are covered in Chapter 2 of the main course textbook (Jurafsky and Martin), and the Perl "Camel book" includes this topic. If you work on the `timberlea` server, or any other Linux computer, the manual pages command '`man perlre`' includes a lot of information about Perl regular expressions, and '`man perlretut`' presents an excellent tutorial on the topic.

Some interesting developments in the history of Regular Expressions are:

- Research by Stephen Kleene: regular sets, and the name of regular sets and regular expressions (1951),
- Implementation in QED by Ken Thompson (1968),

- Open-source implementation by Henry Spencer (1986),
- Use in Perl by Larry Wall (1987),
- Perl-style Regular Expressions in many modern programming languages.

Example Regular Expressions

- Literal: `/woodchuck/ /Buttercup/`
- Character class: `/./` (any character),
`/[wW]oodchuck/`, `/[abc]/`, `/[12345]/`
(any of the characters)
- Range of characters: `/[0-9]/`, `/[3-7]/`, `/[a-z]/`, `/[A-Za-z0-9_-]/`
- Excluded characters and repetition: `/[^()]+/`
- Grouping and disjunction: `/(Jan|Feb) \d?\d/`
- Note: `\d` is same as `[0-9]`
- Another character class: `\w` is same as `[0-9A-Za-z_]` ('word' characters)
- Opposite: `\W` same as `[^0-9A-Za-z_]`

RegEx Examples: Anchors, Grouping, Iteration

```
/^This is a/   # use of anchor
/This^or^that/ # not an anchor
/woodchucks?/
/\bcolou?r\b/       # anchor \b
/is a sentence\.$/  # end of string anchor

# Grouping and iteration:
/This sentence goes on(, and on)*\.$/
/cat|dog/           # disjunction (alternation)
/The (cat|dog) ate the food\./
```