CSE 355: Module 3 Machine Design (NFAs)

Dr. Jamison W. Weber

Due Monday, September 9, 2024, 11:59pm

In this individual assignment, you will formulate nondeterministic finite automata (NFAs) that recognize target languages using a Python programming interface. The programming interface enables you to formally define NFAs, check their validity, visualize them as state machines, and trace their branching computation on test strings. When you're satisfied with your NFAs, export and upload them to the corresponding Gradescope assignment for autograding. This will provide immediate, personalized feedback if there are any mistakes. You may revise and resubmit your code as many times as you'd like until the due date.

1 Target Languages

For each of the following problems, formulate an NFA with the specified number of states (or fewer) that recognizes the target language.

Problem 1

$L_1 = \{w \in \{\texttt{0}, \texttt{1}\}^* \mid w \text{ contains } \texttt{1010} \text{ as a substring}\}$, five states.

Problem 2

$L_2 = \{\texttt{0}, \texttt{1}\}$, two states.

Problem 3

$L_3 = \{w \in \{\texttt{a}, \texttt{b}\}^* \mid w \text{ contains a pair of } \texttt{a}\text{'s separated by an odd number of } \texttt{b}\text{'s}\}$, four states.

Problem 4

$L_4 = \{\texttt{1}^a\texttt{0}^b\texttt{1}^c \mid a \geq 0, b \geq 1, c \geq 0\}$, three states.

Note. Recall that a symbol raised to a number, like $\texttt{0}^n$, is the string with n copies of the symbol `0`. So L_4 is the language of strings that are zero or more `1`s followed by one or more `0`s followed by zero or more `1`s.

Problem 5

$L_5 = \{w \in \{\texttt{0}, \texttt{1}\}^* \mid w \text{ has exactly one } \texttt{0}\text{, an odd number of } \texttt{1}\text{s, or both}\}$, five states.

2 Grading

The instructions in the following sections will show you how to use the programming interface, test your ideas, and ultimately export your solutions as a `.json` file. This file contains specifications of your NFAs, which Gradescope will instantiate and test to see if they correctly recognize the target languages.

This assignment is worth 35 points: five problems (NFAs) worth 7 points each. For each problem, you get 1 point if the NFA is valid (i.e., it follows all the rules of a formal specification), an additional 3 points if it recognizes the target language, and the final 3 points if you do so using the specified number of states (or fewer). If your NFA is invalid, Gradescope will tell you what the problem is. If your NFA recognizes a language other than the target one, Gradescope will output (1) example strings that your NFA recognizes but are outside the target language, and/or (2) example strings that are in the target language but your NFA does not recognize. This feedback should guide your resubmssions.

3 Installation

This assignment uses the same setup as the previous Machine Design assignment: `python 3.10` (or a later version) and the `cse355-machine-design` package. Detailed installation instructions can be found in the previous assignment. To ensure that you're running the latest version of our package, run:

```
> pip install cse355-machine-design --upgrade
```

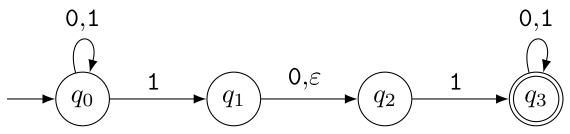
As before, complete details on the package and comprehensive documentation can be found at https://github.com/DaymudeLab/CSE355-MachineDesign.

4 Usage Guide

In this section, we demonstrate how to define and interact with NFAs using the `cse355-machine-design` package. Suppose that we are asked to formulate an NFA recognizing the language

$$L = \{w \in \{0,1\}^* \mid w \text{ contains } 101 \text{ or } 11 \text{ as a substring}\}.$$

As you may recall from Lecture 2 of this module, the below NFA N recognizes L:



4.1 Defining an NFA

To start, download the template Python source file called `module3_machinedesign.py` from Canvas. The top of this file defines our package imports:

```python
from cse355_machine_design import NFA, registry
```

Next, we have functions for defining the five NFAs for this assignment (one per problem). Copy/paste one of these functions to add a new function for our example NFA, N:

```python
def example():
    """
    L = {w in {0,1}* | w contains 101 or 11 as a substring}
    """

    Q = {}
    Sigma = {}
    delta = {

    }
    q0 = ""
    F = {}

    return NFA(Q, Sigma, delta, q0, F)
```

The five variables you need to define (`Q`, `Sigma`, `delta`, `q0`, and `F`) are the five components in an NFA's formal definition $N = (Q, \Sigma, \delta, q_0, F)$. NFA N has states $Q = \{q_0, q_1, q_2, q_3\}$, so we write these states as strings in our Python function:

```python
    Q = {"q_0", "q_1", "q_2", "q_3"}
```

Next, we define our alphabet. The input strings are defined over $\{0,1\}^*$, so:

```
    Sigma = {"0", "1"}
```

Until now, everything has been the same as in the previous Machine Design assignment for DFAs. However, an NFA's transition function δ is allowed to transition to any number of next states for a given current state and symbol, and it can contain ε-transitions. Thus, the range of δ is now sets of states (strings) instead of just one state (string). To represent ε-transitions, we use the underscore "_" symbol:

```
    delta = {
        ("q_0", "0"): {"q_0"},
        ("q_0", "1"): {"q_0", "q_1"},  # Reading a 1 in state q_0 can go to q_0 or q_1.
        ("q_1", "0"): {"q_2"},
        ("q_1", "_"): {"q_2"},         # epsilon-transition from q_1 to q_2.
        ("q_2", "1"): {"q_3"},
        ("q_3", "0"): {"q_3"},
        ("q_3", "1"): {"q_3"}
    }
```

The start state of N is q_0, so:

```
    q0 = "q_0"
```

Finally, our set of accept states is $\{q_3\}$, so:

```
    F = {"q_3"}
```

The completed function is:

```
def example():
    """
    L = {w in {0,1}* | w contains 101 or 11 as a substring}
    """

    Q = {"q_0", "q_1", "q_2", "q_3"}
    Sigma = {"0", "1"}
    delta = {
        ("q_0", "0"): {"q_0"},
        ("q_0", "1"): {"q_0", "q_1"},
        ("q_1", "0"): {"q_2"},
        ("q_1", "_"): {"q_2"},
        ("q_2", "1"): {"q_3"},
        ("q_3", "0"): {"q_3"},
        ("q_3", "1"): {"q_3"}
    }
    q0 = "q_0"
    F = {"q_3"}

    return NFA(Q, Sigma, delta, q0, F)
```

4.2 Interacting with Your NFA

Now that we've defined the NFA N, we can put it to work. In a terminal, navigate to the directory containing your `module3_machinedesign.py` file and then run the `python` command. This will start an interactive session:

```
> python
Python 3.10.13
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

Typing

```
>>> from module3_machinedesign import *
```

will import all definitions from your file into this interactive session. You can then set up one of your NFAs, like the example machine N that we defined:

```
>>> example_nfa = example()
```

Our programming interface provides tools for investigating and testing your NFAs. First, you can open a state diagram of your NFA in your web browser using:

```
>>> example_nfa.display_state_diagram()
```

Second, you can evaluate whether an input string is accepted (`True`) or rejected (`False`) by passing it to the `evaluate` function:

```
>>> example_nfa.evaluate("010110")
True
>>> example_nfa.evaluate("100")
False
```

The branching computation of NFAs can be difficult to follow, so we provide two levels of tracing. If you add the `enable_trace=1` parameter, you get a compressed version of computation tracing:

```
>>> example_nfa.evaluate("010110", enable_trace=1)
Starting in states {'q_0'}
    After epsilon closure: {'q_0'}
Reading input "0":
    Current possible states: {'q_0'}
    After epsilon closure: {'q_0'}
Reading input "1":
    Current possible states: {'q_1', 'q_0'}
    After epsilon closure: {'q_1', 'q_0', 'q_2'}
Reading input "0":
    Current possible states: {'q_0', 'q_2'}
    After epsilon closure: {'q_0', 'q_2'}
Reading input "1":
    Current possible states: {'q_1', 'q_3', 'q_0'}
    After epsilon closure: {'q_1', 'q_3', 'q_0', 'q_2'}
Reading input "1":
    Current possible states: {'q_1', 'q_3', 'q_0'}
    After epsilon closure: {'q_1', 'q_3', 'q_0', 'q_2'}
Reading input "0":
    Current possible states: {'q_3', 'q_0', 'q_2'}
    After epsilon closure: {'q_3', 'q_0', 'q_2'}
Done reading input...

Checking if at least one of the current states is a final state...
"q_3" IS a final state. We can stop looking and accept
True
```

If you add the `enable_trace=2` parameter instead, you get the full, very long version of computation tracing.

4.3 Submitting Your NFAs

At the bottom of the template file, you will see the following code:

```
if __name__ == "__main__":
    problem1().submit_as_answer(1)
    problem2().submit_as_answer(2)
    problem3().submit_as_answer(3)
    problem4().submit_as_answer(4)
    problem5().submit_as_answer(5)
    registry.export_submissions()
```

Don't change this code! Each `problem#()` function is building and returning an NFA based on how you filled out those functions above. The `submit_as_answer` function is recording those NFAs as your answers to the corresponding problems. It is *very important* that the parameter to the `submit_as_answer` function matches the problem number! If these differ, Gradescope won't be able to match up your answers to its grading code and you won't receive any points.

When you're ready to submit your answers, run `python module3_machinedesign.py` on the command line in the directory of your assignment file. This will create a `submissions.json` file that you upload to Gradescope to be graded.