CPSC-406 Report

Ryan Benner
Chapman University

February 23, 2025

Abstract

Contents

1 Introduction **1**

2 Week by Week **1**
2.1 Week 1 1
2.2 Week 2 2
2.3 Week 3 3

3 Synthesis **8**

4 Evidence of Participation **8**

5 Conclusion **8**

1 Introduction

2 Week by Week

2.1 Week 1

1. Automata

- machine model which has variety of computations without memory - traversal (trees, graphs, nodes) - Turing machine - traffic light

- parking machine - mechanical calculators - streaming - regular expressions (regex): such as NLP, parsing, coding theory, file management

automata: implement algorithms, admit algorithmic instructions.

2. Computability And complexity

- universal language to analyze algorithms/problems/computations for: - efficiency/(runtime, spacetime, etc) resource/scaling - comparison between complexity, hardness of problems - complexity classes - big-o-notation - reducing problems - P vs NP, sat problems

3. Graph Algorithms

- network theory (data, mechanical, chemical, operations)

2/6

1. Automata theory

states: positions: (head) $\overrightarrow{l}$ () $\overrightarrow{t}$ (end)

ex.1 parking machine: charge/hr = 25c accepted: coins, no change, no 1, 2c coins

states: - value paid so far (0) 0 -> [5,10,25]

ex.2 valid var names: 1. index 0: letter? 2. index n $>$ 0: letter or digit or symbol

2.2 Week 2

Notes

Chapter 2.1 of ITALC discusses the basic model of computation that demonstrates the structure and components of finite automata. It explains that a finite automaton is a simple computational machine which is composed of a set of states, including a start state, and one or more end states. As the automaton reads an input, it transitions between states based on a set of rules, the transition functions. The section uses examples with simple intuition, such as an on/off switch, to show how a system can 'remember' essential information despite its simplicity. Also, it talks about the difference between deterministic and non-deterministic models. A deterministic model is one in which each state has a unique transition for each input, and a non-deterministic model is one where multiple transitions may be possible for one input. Although these two models are very different in practice, they recognize the same set of basic regular language.

Homework

Introduction to Automata Theory:

Homework: Characterize all the accepted words (i.e., describe exactly those words that get recognized).
Answer: [5, 5, 5, 5, 5], [5, 5, 5, 10], [5, 5, 10, 5], [5, 10, 5, 5], [5, 10, 10], [10, 5, 5, 5], [10, 5, 10], [10, 10, 5]

Homework: Characterize all the accepted words. Can you describe them via a regular expression?
Answer: Any word which ends in 'pay' will result in end state being unlocked

Deterministic and Non-Deterministic Finite Automata:

Homework: Determine for the following words, if they are contained in L_1, L_2, or L_3.
Answer:

word	A_1	A_2
aaa	no	yes
aab	yes	no
aba	no	no
abb	no	no
baa	no	yes
bab	no	no
bba	no	no
bbb	no	no

Homework: Consider the DFA from above: Consider the paths corresponding to the words $w_1 = 0010$, $w_2 = 1101$, $w_3 = 1100$.
Answer: w_1 and w_2 both achieve the end state

Comments and Questions

Because finite automata have limited memory, their ability to recognize more complex language patterns is restricted. Will the range of what is categorized as a finite automaton broaden over time? How could we modify automata to recognize more complex language?

2.3 Week 3

Notes

Homework

Exercise 2, 1-3:

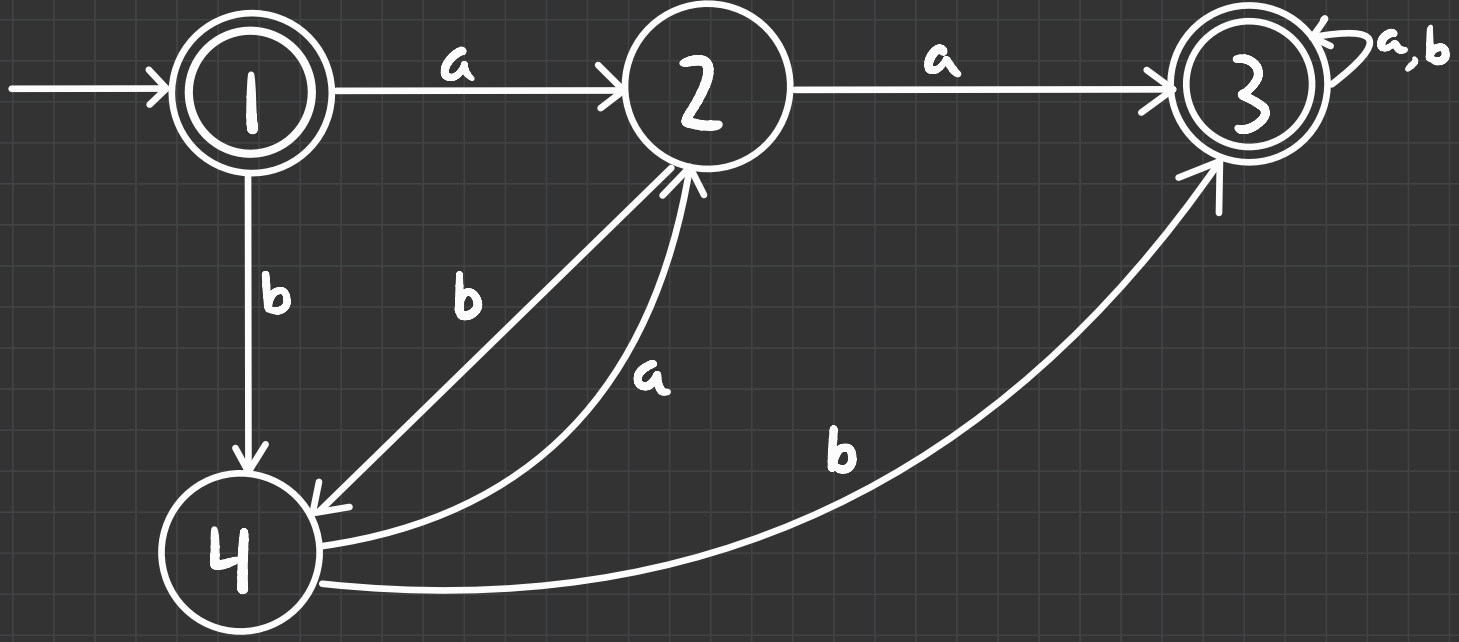
The file **dfa.py** implements a class representing a deterministic finite automaton(DFA) The automaton is defined by a set of states, inputs, transition functions, an initial state, and accepting states, all of which are initialized in the constructor function. The repr method offers a formatted string representation of the DFA for easy inspection. The run method processes an input word by iterating through its characters and transitions between states according to the defined rules. If an undefined transition is encountered or an invalid symbol is detected, the word is rejected. Finally, the refuse method generates a complementary DFA by changing the set of accepting states to accept words that the original DFA rejects.
The file **dfa − ex01.py** implements two DFAs, A1 and A2, and runs each through dfa.py to test them repsectively.

```
# dfa.py
class DFA :

    # init the DFA
    def __init__(self, Q, Sigma, delta, q0, F) :
        self.Q = Q            # set of states
        self.Sigma = Sigma    # set of symbols
        self.delta = delta    # transition function
        self.q0 = q0          # initial state
        self.F = F            # final (accepting) states

    # print the data of the DFA
    def __repr__(self) :
        return f"DFA({self.Q},\n\t{self.Sigma},\n\t{self.delta},\n\t{self.q0},\n\t{self.F})"

    # run the DFA on the word w
    # return if the word is accepted or not
    def run(self, w) :
        current_state = self.q0
        for symbol in w:
            if symbol not in self.Sigma:
                raise ValueError(f"Symbol {symbol} not in the DFA alphabet {self.Sigma}.")
            # look up transition state, if none is defined the word is rejected
            if (current_state, symbol) not in self.delta:
                return False
            current_state = self.delta[(current_state, symbol)]
        return current_state in self.F

    # returns a new DFA that accepts the words that the current DFA refuses and vice versa
    def refuse(self):
        new_F = self.Q - set(self.F)
        return DFA(self.Q, self.Sigma, self.delta, self.q0, new_F)
```

```
# dfa_ex01.py
import dfa as dfa

# generate words for testing
def generate_words():
    words = []
    alphabet = ['a', 'b']
    for first in alphabet:
        for second in alphabet:
            for third in alphabet:
                words.append(first + second + third)
    return words

def __main__() :
    Q1 = {'1', '2', '3', '4'}
    Sigma1 = {'a','b'}
    delta1 = {
        ('1', 'a'): '2',
        ('1', 'b'): '4',
        ('2', 'a'): '2',
        ('2', 'b'): '3',
        ('3', 'a'): '2',
        ('3', 'b'): '2',
        ('4', 'a'): '4',
        ('4', 'b'): '4'
    }
    q01 = '1'
    F1 = {'3'}

    Q2 = {'1', '2', '3'}
    Sigma2 = {'a','b'}
    delta2 = {
        ('1', 'a'): '2',
        ('1', 'b'): '1',
        ('2', 'a'): '3',
        ('2', 'b'): '1',
        ('3', 'a'): '3',
        ('3', 'b'): '1',
    }
    q02 = '1'
    F2 = {'3'}

    A1 = dfa.DFA(Q1, Sigma1, delta1, q01, F1)
    A2 = dfa.DFA(Q2, Sigma2, delta2, q02, F2)

    words = generate_words()
    automata = [A1, A2]

    for X in automata: # test words
         print(f"{X.__repr__()}")
         for w in words:
            print(f"{w}: {X.run(w)}")
         print("\n")

__main__()
```

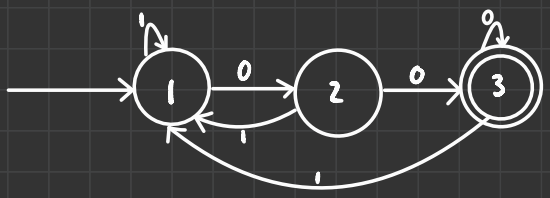
Drawings for Lab Exercise 4, ITALC Exercise 2.2.4

Lab Exercise 4:

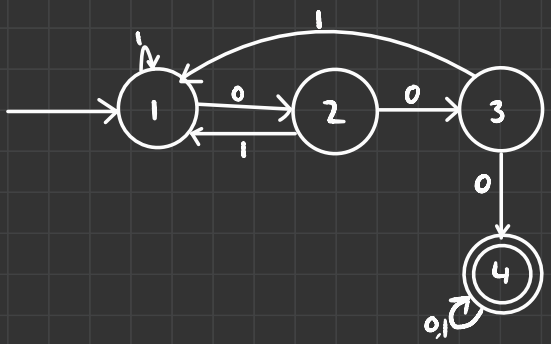


ITALC Exercise 2.2.4:

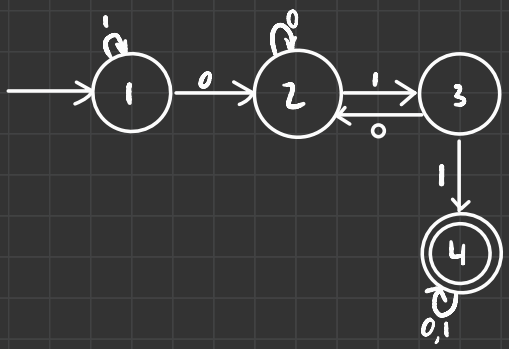
a) String ending in 00



b) all strings w/ 3 consecutive 0s



c) set of strings having substring 011



Comments and Questions

I occasionally find myself struggling to recognize the pattern when first looking at a new DFA. Is there an insightful method or trick to recognizing these patterns more easily?

3 Synthesis

4 Evidence of Participation

5 Conclusion

References

[BLA] Author, Title, Publisher, Year.