Michael Clausen, **Jeffrey Chen**, **Diamond Dinh**, **Evan Fricker**, **Timothy Wiratmo**, **Bryan Nguyen**

CPSC 439-01 Spring 2023

Project 1 - DFAs and Turning Machine

Question 1:

$$L = \{w : w \text{ contains at least three 1s}\}$$

What is to be done:

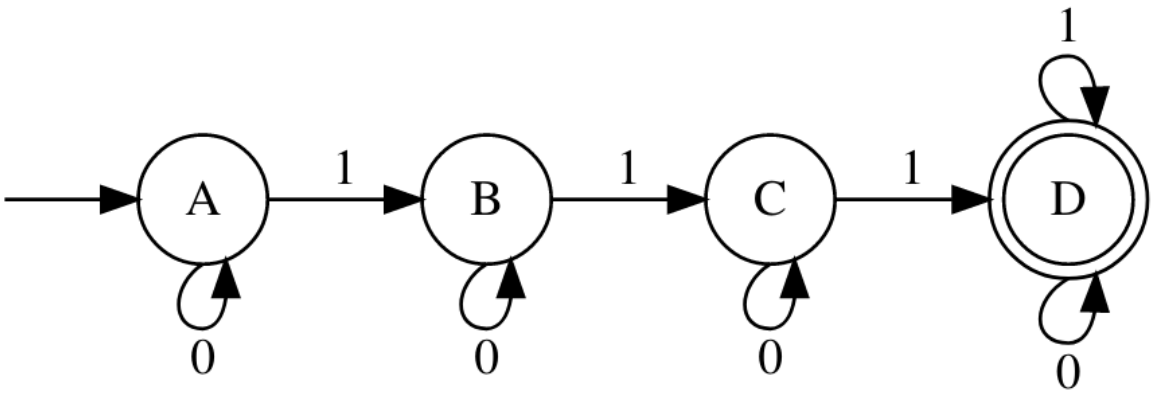
Design a DFA that will recognize the language above and describe the DFA using a transition diagram and the formal definition.

Tools used:

Graphviz

Results:

DFA Diagram and Definition



Formal Definition:

Let M = $\{Q, \Sigma, \sigma, q, F\}$ be a finite automaton where

$Q = \{A, B, C, D\}$,

$\Sigma = \{0, 1\}$,

$q_0 = \{A\}$,

$F = \{D\}$, and

the transition function σ is shown by the following table:

Current State	Next State for Input 0	Next State for Input 1
A	A	B
B	B	C
C	C	D
D	D	D

Comments on results:

The DFA was designed while using Introduction to Theory of Computation by Anil Maheshwari and Michiel Smid as reference.

Question 2

What is to be done:

Implement the DFA you described in step (1). You may use a conventional programming language such as Python or C++ or a DFA simulator such as JFLAP or David Doty's web-based Automaton Simulator.

Tools used:

(Go Playground/code link)

Results:

Implementation

```
type DFA_2_13_1_State byte

const (
    DFA_2_13_1_A DFA_2_13_1_State = iota
    DFA_2_13_1_B
    DFA_2_13_1_C
    DFA_2_13_1_D
)

var DFA_2_13_1 = MatcherDFA[DFA_2_13_1_State]{
    δ: map[DFA_2_13_1_State]StateFunc[byte, DFA_2_13_1_State]{
       DFA_2_13_1_A: func(v byte) Result[DFA_2_13_1_State] {
             switch v {
             case '0':
                   return Ok(DFA_2_13_1_A)
             case '1':
                   return Ok(DFA_2_13_1_B)
             default:
                   return None[DFA_2_13_1_State]()
             }
       },
       DFA_2_13_1_B: func(v byte) Result[DFA_2_13_1_State] {
             switch v {
             case '0':
                   return Ok(DFA_2_13_1_B)
             case '1':
                   return Ok(DFA_2_13_1_C)
             default:
                   return None[DFA_2_13_1_State]()
             }
       },
       DFA_2_13_1_C: func(v byte) Result[DFA_2_13_1_State] {
             switch v {
             case '0':
                   return Ok(DFA_2_13_1_C)
             case '1':
                   return Ok(DFA_2_13_1_D)
             default:
                   return None[DFA_2_13_1_State]()
             }
       },
       DFA_2_13_1_D: func(v byte) Result[DFA_2_13_1_State] {
             switch v {
             case '0', '1':
                   return Ok(DFA_2_13_1_D)
             default:
                   return None[DFA_2_13_1_State]()
```

```
            }
        },
    },
    q: DFA_2_13_1_A,
    F: NewSet(DFA_2_13_1_D),
}

func TestDFA_2_13_1(t *testing.T) {
    AssertMatch(t, DFA_2_13_1, "110", false)
    AssertMatch(t, DFA_2_13_1, "0", false)
    AssertMatch(t, DFA_2_13_1, "1", false)
    AssertMatch(t, DFA_2_13_1, "11", false)
    AssertMatch(t, DFA_2_13_1, "1111", true)
    AssertMatch(t, DFA_2_13_1, "111", true)
    AssertMatch(t, DFA_2_13_1, "01110", true)
    AssertMatch(t, DFA_2_13_1, "01101", true)
    AssertMatch(t, DFA_2_13_1, "01100", false)
}

/*
=== RUN   DFA_2_13_1
      dfa.go:90: "110" does not match
      dfa.go:90: "0" does not match
      dfa.go:90: "1" does not match
      dfa.go:90: "11" does not match
      dfa.go:88: "1111" matches
      dfa.go:88: "111" matches
      dfa.go:88: "01110" matches
      dfa.go:88: "01101" matches
      dfa.go:90: "01100" does not match
--- PASS: DFA_2_13_1 (0.00s)
*/
```

Question 3

Designing a Turing Machine for this language is straightforward. We retain the four states from the DFA as well as its transition function. We then define accept and reject states, as well as the start state. If the machine enters an accept state, it halts and the input is accepted. If the machine enters a reject state, it halts and the input is rejected. In this case, the input is accepted if the machine reads a blank while in state D. The input is rejected if the machine reads a blank while in any of the other states. As for the start state, the machine simply goes right one to read the start of the input.

Definition of Turing Machine M

Alphabet: {▷,∅, 0, 1}

States: {START, A, B, C, D, ACCEPT, REJECT}

Description of M:

- Begin in state START and go right one. If the input is blank, enter state REJECT. Otherwise, enter state A.
- In state A, if the input is 0, remain in state A. If it's 1, enter state B. If it's blank, enter state REJECT. If a 0 or 1 is read, go right one.
- In state B, if the input is 0, remain in state B. If it's 1, enter state C. If it's blank, enter state REJECT. If a 0 or 1 is read, go right one.
- In state C, if the input is 0, remain in state C. If it's 1, enter state D. If it's blank, enter state REJECT. If a 0 or 1 is read, go right one.
- In state D, if the input is 0 or 1, remain in state D and go right one. If the input is blank, enter state ACCEPT.
- In state ACCEPT, we halt and the input is accepted.
- In state REJECT, we halt and the input is rejected.

Question 4

What is to be done:

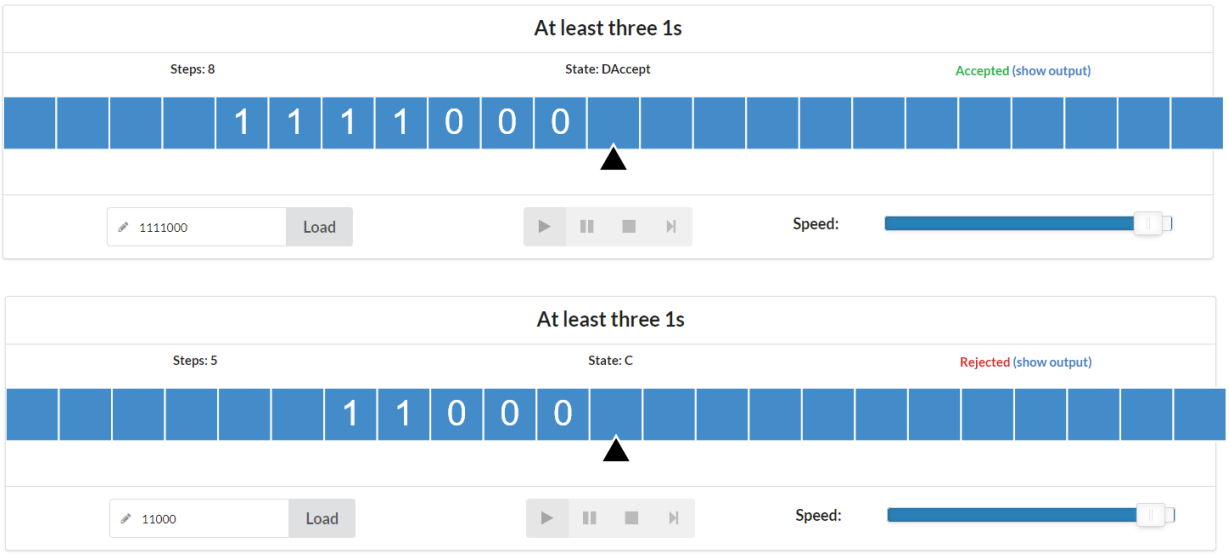
- Implement the turing machine that is designed in question 3
- Learn how to create a turing machine using the online turing machine simulator

Tools used:

Online turing machine simulator: https://turingmachinesimulator.com

Results:

[Implementation]



Comments on results:

- Should reject input if there are less than three ones.
- Should accept input if there are more than three ones

Question 5:

What is to be done:

We have to define a language similar to the one that we chose in the first step. However, the language cannot be regular. This means that the language can't be accepted by a DFA. We can show this by considering how Pumping Lemma plays a role in our new language.

Results:

$$L = \{w : w \text{ contains at least three 1s, followed by the same amount of 0s}\}$$

$$\text{Expression} = 1^n0^n \text{ for any } n >= 3$$

Comments on results:

This language is not regular because it violates the pumping rule via contradiction. When we break up this expression into xy^iz to find the pump, we encounter a problem. If we put only ones in the y slot, it will fail as it cannot be pumped (no corresponding 0s), if we put only zeros into the y slot, it will fail for the same reason (no corresponding 1s). However, if we use the combination of 1 followed by a 0, it will fail as it won't fall into the language (will turn into 1010101010... which does not meet language requirements). As there is no other possible combination, this language cannot be pumped, and is therefore non-regular.

Question 6

Defining a Turing machine for this language is more complex. We can use a TM similar to question three to check that there are at least three ones. This is our first pass through the input, after which we return to the beginning for the second pass, in which we check that we have the same number of zeroes and ones. To do this, some sort of stack must be used, wherein we push a one on the stack for every “one” that is read and pop it off the top for every “zero” read. We erase the input once it has been read in order to get back to the next input after a stack operation. If the stack is empty once we have read through the input, the input is accepted. If the stack is not empty at this point, or if we try to “pop” and the stack is already empty, then the input is rejected. We also must enforce the condition that once we have read a zero, there cannot be any more ones.

To implement these operations, we have “check” states that either push a one, pop from the stack, or check that the stack is empty, depending on the input read. We also have R# and L# states (# stands for blank), wherein we go left or right until a blank is reached. These allow us to go back and forth between the top of the stack and the next input to be read. The ERASE_ONE and OUTPUT_ONE states are self explanatory.

The TM is implemented in

Definition of Turing Machine M

Alphabet: {▷,∅, 0, 1}

States: {START, A, B, C, D, RETURN, CHECK_ONES, PUSH, R#1_PUSH, R#2_PUSH, OUTPUT_ONE, L#1_PUSH, L#2_PUSH, CHECK_ZEROES, POP, R#1_POP, R#2_POP, ERASE_ONE, L#1_POP, L#2_POP, CHECK_BLANKS, ACCEPT, REJECT}

Description of M:

1st Pass:

- Begin with state START. Go right one. Enter state A.
- In state A, if the input is 1, enter state B and go right one. Otherwise, REJECT.
- In State B, if the input is 1, enter state C and go right one. Otherwise, REJECT.
- In State C, if the input is 1, enter state D and go right 1. Otherwise, REJECT.
- In state D, if the input is 1, remain in D and go right 1. If 0, enter state RETURN and go left one. If blank, REJECT.
- In RETURN, go left until we hit the start symbol (or a blank in the TM simulator). Then enter state CHECK_ONES and go right one.

2nd Pass:

- In state CHECK_ONES, if the input is 1, enter state PUSH. If it's 0, enter state CHECK_ZEROES. If it's blank, REJECT.
- In PUSH, output a blank. Then, enter state R#1_PUSH.
- In R#1_PUSH, go right until a blank is reached. Then, enter state R#2_PUSH.
- In R#2_PUSH, go right until a blank is reached. Then, enter state OUTPUT_ONE.
- In OUTPUT_ONE, write a 1 and enter state L#1.

· In L#1_PUSH, go left until a blank is reached. Then, enter state L#2_PUSH.

· In L#2_PUSH, go left until a blank is reached. Then, enter state CHECK_ONES, go right one, and loop.

· In state CHECK_ZEROES, if the input is 0, enter state POP. If it's 1, REJECT. If it's blank, enter state CHECK_BLANKS and go right one.

· In state POP, write a blank. Then, enter state R#1_POP.

· In R#1_POP, go right until a blank is reached. Then, enter state R#2_POP.

· In R#2_POP, go right until a blank is reached. Then, enter state ERASE_ONE and go left.

· In state ERASE_ONE, if the input is 1, output a blank and enter state L#1_POP. If the input is blank, reject.

· In L#1_POP, go left until a blank is reached. Then, enter state L#2_POP.

· In L#2_POP, go left until a blank is reached. Then enter state CHECK_ZEROES, go right one, and loop.

· In state CHECK_BLANKS, if the input is blank, ACCEPT. If it's a 1, REJECT.

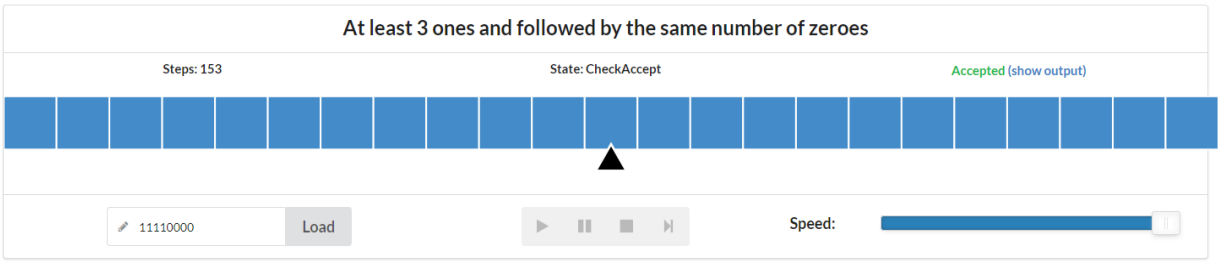
· In state ACCEPT, we halt and the input is accepted.

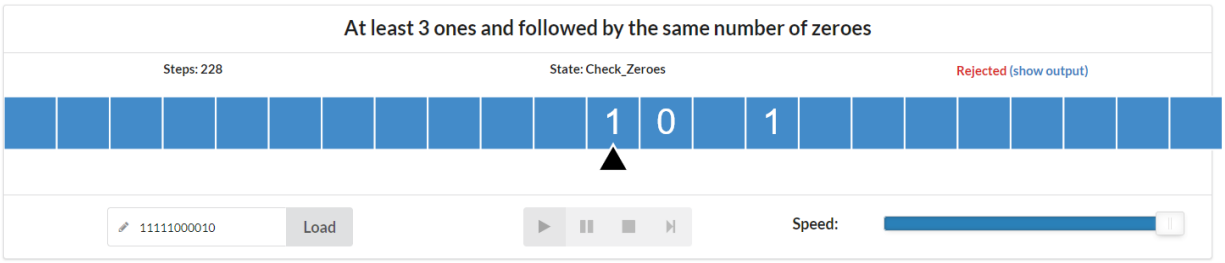
· In state REJECT, we halt and the input is rejected.

Implementation of Turing Machine M

We used Martin Ugarte's online simulator, as in question 3.

Results: Implementation


At least 3 ones and followed by the same number of zeroes
Steps: 153
State: CheckAccept
Accepted (show output)
11110000
Load
Speed:


At least 3 ones and followed by the same number of zeroes
Steps: 228
State: Check_Zeroes
Rejected (show output)
1
0
1
11111000010
Load
Speed: