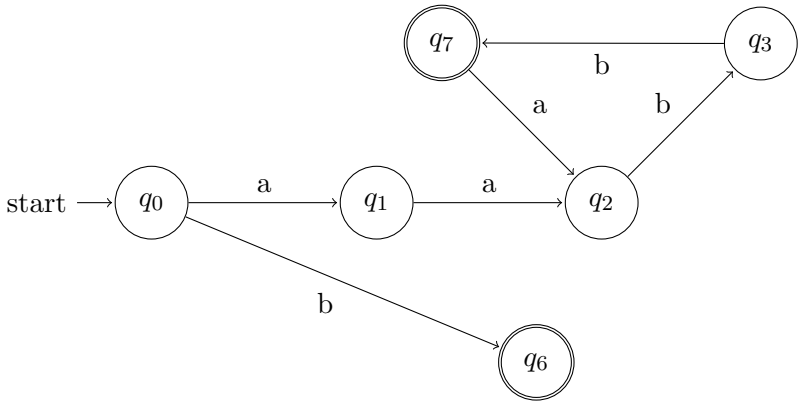
ICCS310: Assignment 2

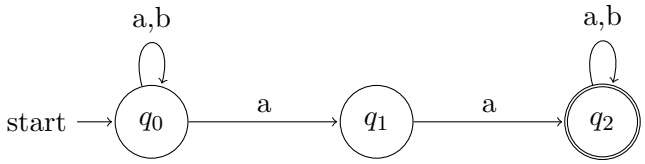
Possawat Sanorkam
possawat2017@hotmail.com
January 26, 2021

1: Regex to NFA/DFA

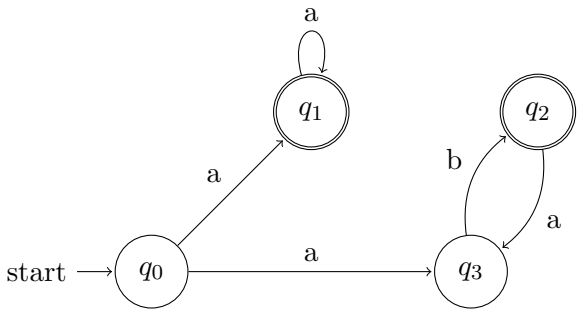
(1) $a(abb)^* + b$



(2) $(a + b)^*aa(a + b)^*$



(3) $a^+ + (ab)^+$



2: Finite-State Machines to Regex

(1) $\varnothing^*$ (Rejecting any input)

(2) $a^* + a^*b^+a^+b$ (Contains only as or any pattern of as to bs to as to bs.)

3: Binary Addition

$$A = \{w \in \Sigma^* \mid \text{ the bottom row of w is the sum of the top two rows } xy \in L_1\}$$

Prove that A is regular.

Proof: Since, A only accept column vectors of size 3 such that the bottom row of w is the sum of the top two rows, it is difficult create a machine that recognize A directly. We know that binary addition start by summing the least significant bits, including transferring a carry when the bit overflow. So, reading the string in reverse will be simpler since we can transfer the carry from the last column to the next column on the left directly.

From the lemma, if L is a regular language, then L^R is also regular. We want to show that A^R is a regular language. Let $L(M) = A^R$. So, M is a machine that recognize A^R, else we can call it the binary adder.

M is machine that translate each vector into transfer bit and send it to the next state. It only accepts the string that follow its calculation (The sum of first row, second row and carry bit must be equal to the third row). If both numbers are 1 on first and second rows, we send it the the state which refers to a carried bit included state. There are 2 states, q_1, and q_2.

q_1 is the accepting state with no carry bit transferred.

q_2 is the state with carry bit. (Not done with the addition yet)

We can mathematically define a function of LSB addition as $a \odot b = c$ and a function of carry bit as $carry(x, y, z)$. The transition of this machine follows that $x_1 \odot x_2 +$ carry bit state $= x_3$ must be satisfied and the next transition depends on the carry bit from $carry(x_1, x_2,$ carry bit state $)$ whether it will be $q_1 or q_2$. Hence, A^R is regular and that makes A regular also from the lemma.

Therefore, A is regular. □

4: Division Operation?

$$\frac{L_1}{L_2} = \{x \mid \exists \in L_2 \text{ s.t. } xy \in L_1\}$$

Prove that if L_1 and L_2 are regular, then $\frac{L_1}{L_2}$ is also regular.

Proof: From a lemma, for every regular expression R, there is a DFA that recognizes the language $L(R)$. Suppose L_1 and L_2 are regular, then there exist DFA $M_1 = (Q, \Sigma, \delta, q_0, F_1)$ which accepts L_1 and DFA $M_2 = (Q, \Sigma, \delta, q_0, F_2)$ which accepts L_2. We want to show that $L_3 = \frac{L_1}{L_2}$ where $L_1, L_2, L_3 \in \mathbb{I}$.

We have that $Q, \Sigma, \delta,$ and q_0 in M_1 and M_2 can be shared, just that the accepting states are different. L_3 then can be recognized by some DFA $M_3 = (Q, \Sigma, \delta, q_0, F_3)$. We know that Σ is a number digit alphabet (0-9). Then, each state is just an integer. So, $\forall x \in F_1, \exists y \in F_2,$ and $\exists z \in F_3, zy = x$.

From the observation, L_1, which is regular, contains accepting states that made of zy from F_2 and F_3. Also, L_2, which is regular, recognized by M_2 and we can choose any number to be an accepting state in F_2 (As long as we accept at least a number). Since F_3 can be any state (number) also, there always exist z that will satisfied $zy = x$. Thus, M_3 exists since we can design F_3.

Therefore, if L_1 and L_2 are regular, then $\frac{L_1}{L_2}$ is also regular. □

5: Does It Accept Everything?

Let $M = (Q, \Sigma, \delta, q_0, F)$.

Solution: Every finite-state machine is simply a directed graph, then we can check it by performing this algorithm.

Let $strs = \Sigma^*$, given size of Σ is a constant. Also, E = edges that we can refer to δ.

First, we design a function called *is_recognize* that check if a string can be recognized from M by forming a directed graph using M, which is the given machine, then perform a traversal to get the final state (cycles are ignored since we stopped on the last character). This would take $O(|Q| + |E|)$ to create the graph. Suppose we have infinite amount of memory, then we can store this graph to use later. If any string rejected, then we set the *accept* to false, which means machine M rejects Σ^*. Hence, assuming amortized running time for getting an item from a hash set is $O(1)$, then the time complexity would be $O(str) + O(1) + O(1) = O(str)$.

Second, we make sure that we can enumerate through $strs$. If we have infinite power of computation, we can perform *is_recognize* on each string in $strs$ at the same time. Then, the time complexity would be $O(str)$. If not, we can iterate on each string instead, which will definitely take longer time. Let ns = number of strings in $strs$, it would take $O(str * ns)$.

Finally, return whether we accept it or not. Total time complexity $O(|Q| + |E| + str)$ in parallel or else $O(|Q| + |E| + str * ns)$.

The pseudo-code is given below.

```
1  //global variables
2  G = create_graph(M) //a graph using Q as vertices and E from DFA
3  accept = true //assume that this variable is thread-safe globally
4  is_recognize(str){
5          count = 0 //First character
6          n = length of str - 1
7          current_state = str[count]
8          while(count < n){
9                  next_char = str[count+1]
10
11                 //using map of edges of current_state to go to next state
12                 current_state = go_next(current_state,next_char)
13                 count++ //increment count
14         }
15         if(current_state not in F){
16                 accept = false
17                 //if there is any rejection, reject it globally.
18         }
19 }
20
21 accept_all(strs){
22         //parallel mapping is_recognize on every string
23         strs.par_map(is_recognize)
24         return accept
25 }
```

6: All The Same?

Multiplication and power simply make them equivalent like $2 * 4 = 4 * 2 === 2^4 = 4^2$.