# Finite Automata and Regular Expresions in Haskell

Yuanho Yao

Saturday 25th August, 2018

**Abstract**

This article investigates how finite automata and regular expressions are connected. We give an implementation of DFA(deterministic finite automata) and NFA(non-deterministic finite automata) and RE(regular expression) in Haskell. We first encode DFA and then give a code converting RE to NFA and NFA to DFA respectively. This program enables us to verify if a string is accepted in certain DFA. And we can determine if a string is an instance of certain RE.

## Contents

**1 Floyd–Warshall algorithm** **2**

**2 Deterministic Finite automata** **3**

**3 Regular expressions to finite automata** **5**

3.1 RE to NFA . . . . . 6

3.2 NFA to DFA - powerset construction . . . . . 10

**4 Future work** **12**

**Bibliography** **12**

# 1 Floyd–Warshall algorithm

In this section, we give a Haskell implementation of the Floyd–Warshall algorithm[3], which is related to Kleene's algorithm[11]: an algorithm converting DFA(deterministic finite automata) into RE(regular expressions).

The goal of this code is to find a shortest weighted path between two vertices.

We start with the definition of graphs and give an example of a graph.

```
module Mycode where

data Graph = Gr [Int] [(Int,Int)] ((Int,Int) -> Int)

instance (Show Graph) where
  show (Gr vertices edge weight) =
    "Gr " ++ show vertices ++ " " ++ show (zip edge (map weight edge))


myGraph :: Graph
myGraph = Gr vertices edge weight where
  vertices=[1,2,3,4]
  edge=[(1,3),(2,1),(4,2),(3,4),(2,3)]
  weight (1,3) = -2
  weight (2,1) = 4
  weight (4,2) = -1
  weight (3,4) = 2
  weight (2,3) = 3
  weight _     = 9999999999
```

Now we can give a `shortestPath` function, where the arguments are a graph, the start vertex, the end vertex, and the intermediate vertex respectively. If there is no intermediate vertex between the start vertex and the end vertex, then the output equals the weight between these two vertexes; if we take vertex $n$ as the intermediate vertex, the output equals the smaller value of the output taking $n-1$ as the intermediate vertex or the summation of the outputs taking $n-1$ as the intermediate vertex but $n$ as the new start vertex or end vertex respectively. In this example, if you take 4 as the intermediate vertex, the function will return the cost of the best-weighted path. For convenience, 9999999999 means there is no such path between the vertices.

Then we define the `shortestPath` function as follows:

```
shortestPath :: Graph -> Int -> Int -> Int -> Int
shortestPath (Gr _ _ weight) a b 0 = weight (a,b)
shortestPath g@Gr{}    a b c =
  minimum [ shortestPath g a b (c-1)
          , shortestPath g a c (c-1) + shortestPath g c b (c-1) ]
```

The example is as follows:

```
*MyCode> myGraph
Gr [1,2,3,4] [((1,3),-2),((2,1),4),((4,2),-1),((3,4),2),((2,3),3)]
*MyCode> shortestPath myGraph 2 3 4
2
```

# 2 Deterministic Finite automata

In this section, we give a general method to produce DFA and an implementation in Haskell.[5]

First, we define a DFA[4]. A DFA is represented formally by a 5-tuple $(Q, \sum, q_0, F, \delta)$, where:

- $Q$ : a fine set of states
- $\sum$ : a finite set of input symbols
- $q_0 \in Q$ : a start states
- $F \subseteq Q$ : a set of final state
- $\delta : Q \times \sum \to Q$ : a trasition function

The code is as follows:

```
type Alphabet = String
type TransitionFunction = [((Int, Char), Int)]

data DFA = DFA{states :: [Int]
               ,alphabet :: Alphabet
               ,start :: Int
               ,finals :: [Int]
               ,trans :: TransitionFunction}
               deriving (Show)
```

DFA work as follows:

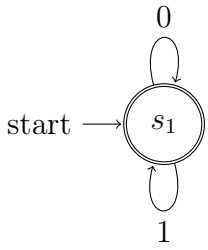
We input a string of symbols, for example, “10100110”, from the start state. We take the first digit of the string, ‘1’, and the state $q_0$ as arguments for the function $\delta$ to get to another state. The string becomes “0100110” now. We repeat the procedure. Once the string becomes “ ”, the empty string, we check the state where it stops. If the state is in the set of final(accepting) states, the string is accepted; otherwise, it is rejected.

In this article, we only use ‘0’,‘1’ as our symbols. We define the accepting function as follows:

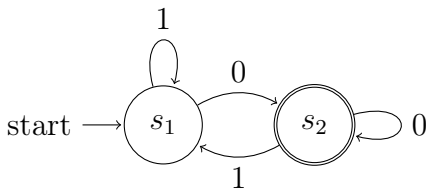
```
acceptingInDFA :: DFA -> String -> Bool
acceptingInDFA (DFA _ _ s f t) = state s
  where  state q "" = q `elem` f
         state q ('1':xs) = state (t ! (q,'1')) xs
         state q ('0':xs) = state (t ! (q,'0')) xs
         state _ _ = error "unkonwn symbol"

(!) :: Eq a => [(a,b)] -> a -> b -- suggested by Malvin
(!) rel x = y where (Just y) = lookup x rel
```

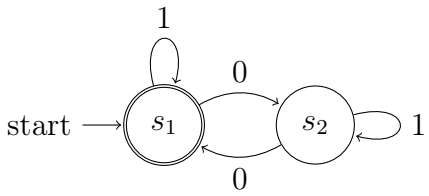
Here are some examples of DFA and their diagrams:



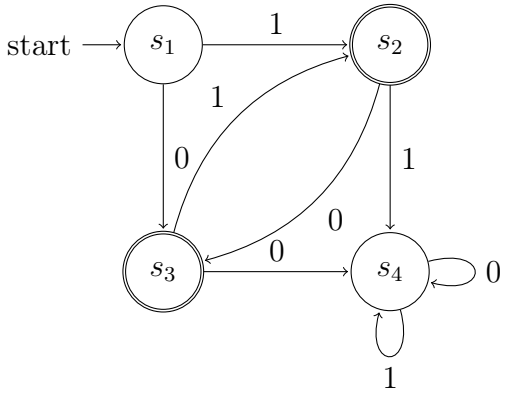

```
exTrue :: DFA
exTrue = DFA q sig s f t where
           q   = [1]
           sig = ['0','1']
           s   = 1
           f   = [1]
           t   = [((1,'1'),1),((1,'0'),1)]
```




```
exEnd0 :: DFA
exEnd0 = DFA q sig s f t where
           q   = [1,2] -- HLint Suggestion: Reduce duplication.
           sig = ['0','1'] -- I want to keep them to make the example clear.
           s   = 1
           f   = [2]
           t   = [((1,'1'),1),((1,'0'),2),((2,'1'),1),((2,'0'),2)]
```




```
exWiki :: DFA
exWiki = DFA q sig s f t where
           q   = [1,2]
           sig = ['0','1']
           s   = 1
           f   = [1]
           t   = [((1,'1'),1),((1,'0'),2),((2,'1'),2),((2,'0'),1)]
```




```
ex01or10 :: DFA
ex01or10 = DFA q sig s f t where
           q   = [1..4]
           sig = ['0','1']
           s   = 1
           f   = [2,3]
           t   = [((1,'1'),2),((1,'0'),3),((2,'1'),4),((2,'0'),3),
                  ((3,'1'),2),((3,'0'),4),((4,'1'),4),((4,'0'),4)]
```

Some example of how the accepting function works:

```
*MyCode> acceptingInDFA exTrue "1001110101"
True
*MyCode> acceptingInDFA exEnd0 "1001110101"
False
*MyCode> acceptingInDFA exEnd0 "10011101010"
True
*MyCode> acceptingInDFA exWiki "10011101010"
False
*MyCode> acceptingInDFA exWiki "100111010100"
True
*MyCode> acceptingInDFA ex01or10 "100111010100"
False
*MyCode> acceptingInDFA ex01or10 "010101"
True
*MyCode> acceptingInDFA ex01or10 "10101010"
True
```

# 3 Regular expressions to finite automata

In this section, we give a converting algorithm from RE to NFA(non-deterministic finite automaton), and from NFA to DFA. We use NFA only as an intermediate step. [1]

First we define RE used in this article as follows:

- The empty string $\epsilon$ and 0 and 1 are RE.

[1] There are already algorithms for RE to NFA and NFA to DFA. It will be more complicated if we skip NFA and write our program for RE to DFA directly.

- For $\varphi$ and $\psi$ are RE, $\varphi^*$(Kleene star), $\varphi\psi$(concatenation), and $\varphi|\psi$(alternation) are RE.

An example for RE is $(01)^*|(10)^*|0(10)^*|1(01)^*$. Our goal is to get a function that takes as input a regular expression and that outputs a DFA that accepts the same regular expression.

## 3.1 RE to NFA

Now we want to convert RE to NFA. We start from formal definiton of RE:[9]

- (empty set) $\emptyset$
- (empty string) $\epsilon$
- (literal character) `'0','1'` in $\sum$

Operations:

- (concatenation) $RS$
- (alternation) $R|S$
- (Kleene star) $R^*$

We take the following definition of `data Reg` from the reference[10].[2]

```
data Reg = E | -- Empty string
           L Char | -- Literal
           Alt Reg Reg | -- alternation
           Con Reg Reg | -- Concatenation
           Kle Reg -- Kleene star
           deriving (Eq,Show)
```

Some exaples for RE are:

```
exRE1 :: Reg
exRE1 = Alt (L '1') (Con (L '1') (L '0')) -- 1|10

exRE2 :: Reg
exRE2 = Alt (Con (L '0') (L '0')) (Con (L '1') (L '0')) -- 00|10

exRE3 :: Reg
exRE3 = Alt (L '1') (Con (L '1') (Alt (L '1') (L '0'))) -- 1|(1(1|0))

exRE4 :: Reg
exRE4 = Kle (L '1') -- 1*

exRE5 :: Reg
exRE5 = Kle (Con exRE1 exRE2) -- ((1|10)(00|10))*
```

[2]We only refer to the definition of `Reg` and use a different way to program.

Then we define NFA[6], which can access a set of states instead of one state in DFA. An NFA is represented formally by a 5-tuple $(Q, \sum, q_0, F, \Delta)$, where:

- $Q$ : a fine set of states
- $\sum$ : a finite set of input symbols
- $q_0 \in Q$ : a start state
- $F \subseteq Q$ : a set of final state
- $\Delta : Q \times \sum \to P(Q)$ : a transition function

```
type NTransitionFunction = [((Int, Char), [Int])]

data NFA = NFA{n_states :: [Int]
               ,n_alphabet :: Alphabet
               ,n_start :: Int
               ,n_finals :: [Int]
               ,n_trans :: NTransitionFunction}
               deriving Show
```

In this section, we use Thompson's construction[1] algorithm to construct NFA from RE. For $1^*$, the NFA is the following.

```
onestar :: NFA
onestar = NFA nq nsig ns nf nt where
  nq   = [1..4]
  nsig = ['e','0','1']
  ns   = 1
  nf   = [4]
  nt   = [((1,'e'),[1,2,4]),((2,'1'),[3]),((2,'e'),[2])
         ,((3,'e'),[2,3,4]),((4,'e'),[2,4])]
```

Before we define the function `regToNFA`, we define some function we will use in defining `regToNFA`. The function `replace'` replaces states in transition function.

```
replace' :: Eq a => a -> a -> [a] -> [a]
replace' x y = map (\z -> if x == z then y else z)
```

The function `toSet` sends a list to a set; the function `subset` checks subset relation.

```
toSet :: Eq a => [a] -> [a]
toSet [] = []
toSet (x:xs) = x : toSet(filter (/= x) xs)

subset :: Eq a => [a] -> [a] -> Bool
subset a b = null [x | x<-a ,x `notElem` b]
```

Here is RE to NFA[2][8]. We begin from define empty set and literals.

```
regToNFA :: Reg -> NFA
regToNFA E = NFA nq nsig ns nf nt where
  nq   = [1,2]
  nsig = ['e','0','1']
  ns   = 1
  nf   = [2]
  nt   = [((1,'e'),[2])]
```

```
regToNFA (L '1') = NFA nq nsig ns nf nt where
  nq   = [1,2]
  nsig = ['e','0','1']
  ns   = 1
  nf   = [2]
  nt   = [((1,'1'),[2])]
regToNFA (L '0') = NFA nq nsig ns nf nt where
  nq   = [1,2]
  nsig = ['e','0','1']
  ns   = 1
  nf   = [2]
  nt   = [((1,'0'),[2])]
regToNFA (L  _ ) = error "unkonwn symbol"
```

Then we define the operators `Con`, `Alt`, and `Kle` recursively. Notice that we reassign states in transition functions.

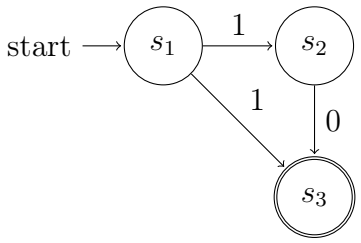
```
regToNFA (Con E y) = regToNFA y
regToNFA (Con x E) = regToNFA x
regToNFA (Con x y) = NFA nq nsig ns nf nt where
  nq   = [1..(lengthX + lengthY - 1)]
  nsig = ['e','0','1']
  ns   = 1
  nf   = [length nq]
  nt   = n_trans regToNFAX ++
         map (\((a,b),c)->((a+lengthX-1,b),map (\n->n+lengthX-1) c))(n_trans
             regToNFAY)
  lengthX = length(n_states regToNFAX)
  lengthY = length(n_states regToNFAY)
  regToNFAX = regToNFA x
  regToNFAY = regToNFA y
regToNFA (Alt x y)
  |x == y    = regToNFA x
  |otherwise = NFA nq nsig ns nf nt where
  nq   = [1..(lengthX + lengthY - 2)]
  nsig = ['e','0','1']
  ns   = 1
  nf   = [length nq]
  nt   = map (\((a,b),c)->((a,b),map (\n->n+lengthX-2) c))headRegToNFAY ++
         zip fstRegToNFAX (map replace'' sndRegToNFAX) ++
         map (\((a,b),c)->((a+lengthX-2,b),map (\n->n+lengthX-2) c))
             tailRegToNFAY
  lengthX = length(n_states regToNFAX)
  lengthY = length(n_states regToNFAY)
  regToNFAX = regToNFA x
  regToNFAY = regToNFA y
  headRegToNFAY = [head (n_trans regToNFAY)]
  tailRegToNFAY = tail (n_trans regToNFAY)
  fstRegToNFAX  = map fst (n_trans regToNFAX)
  sndRegToNFAX  = map snd (n_trans regToNFAX)
  replace'' = replace' lengthX (length nq)
regToNFA (Kle E) = regToNFA E
regToNFA (Kle x) = NFA nq nsig ns nf nt where
  nq   = [1..(lengthX + 2)]
  nsig = ['e','0','1']
  ns   = 1
  nf   = [length nq]
  nt   = [((1,'e'),[2]),((1,'e'),[lengthX+2])] ++
         map (\((a,b),c)->((a+1,b),map (+ 1) c))(n_trans regToNFAX) ++
         [((lengthX+1,'e'),[lengthX+2]),((lengthX+2,'e'),[2])]
  lengthX = length(n_states regToNFAX)
  regToNFAX = regToNFA x
```

Here is an example of how reToNFA works:

```
*Mycode> exRE1
Alt (L '1') (Con (L '1') (L '0'))
```

```
*MyCode> regToNFA exRE1
NFA {n_states = [1,2,3], n_alphabet = "e01", n_start = 1, n_finals = [3],
n_trans = [((1,'1'),[3]),((1,'1'),[2]),((2,'0'),[3])]}
```

This only represents arrows on the diagram below.




Notice that every state can transit from itself to itself with '$e$'. And we can combine arrows $((1, \text{'1'}), [2])$ and $((1, \text{'1'}), [3])$ to $((1, \text{'1'}), [2, 3])$. First, We combine $((x, \text{'}y\text{'}), [z])$ and $((x, \text{'}y\text{'}), [w])$ to $((x, \text{'}y\text{'}), [z, w])$:

```
combineNTrans :: NTransitionFunction -> NTransitionFunction
combineNTrans x
  |length x>1 = head (com x) : combineNTrans(tail (com x))
  |otherwise = x

com :: NTransitionFunction -> NTransitionFunction
com x
  |length x < 2 = x
  |fst (head x) `elem` map fst (tail x)
  = toSet(com( (fst (head x),toSet(snd (head x)
    ++ concatMap snd (filter (\n -> fst n == fst (head x))
    (tail x) ) ))
    : filter (\n -> fst n /= fst (head x))(tail x) ) )
  |otherwise = x
```

Here are some examples for test:

```
exNTF :: NTransitionFunction
exNTF = [((1,'e'),[2,8]),((1,'e'),[4,3]),((3,'0'),[4]),((4,'0'),[5])
      ,((5,'0'),[7]),((5,'1'),[6]),((6,'0'),[7]),((6,'0'),[8])
      ,((8,'e'),[2])]

exNTF2 :: NTransitionFunction
exNTF2 = [((1,'e'),[2,8]),((1,'e'),[9])]
```

This example shows how `combineNTrans` works:

```
*MyCode> combineNTrans exNTF
[((1,'e'),[2,8,4,3]),((3,'0'),[4]),((4,'0'),[5]),((5,'0'),[7])
,((5,'1'),[6]),((6,'0'),[7,8]),((8,'e'),[2])]
*Mycode> combineNTrans exNTF2
[((1,'e'),[2,8,9])]
```

However, `regToNFA` is not enough for the powerset construction[7], because it lacks reflexivity for '$e$'. Therefore, we need `regToNFA'`, which has reflexivity. We want to add transition functions $((x, \text{'}e\text{'}), [x])$, which is reflexivity, to the original NFA. We divide the process into two parts. The first part adds '$e$' to the states which already has '$e$' to some other states to themselves:

```
addE :: ((Int, Char), [Int]) -> ((Int, Char), [Int])
```

```
addE ((x,'e'),z) = ((x,'e'),x : z)
addE x = x
```

The result looks like:

```
*MyCode> map addE (combineNTrans exNTF)
[((1,'e'),[1,2,8,4,3]),((3,'0'),[4]),((4,'0'),[5]),((5,'0'),[7])
,((5,'1'),[6]),((6,'0'),[7,8]),((8,'e'),[8,2])]
```

The second part adds '$e$' to the reminded states to themselves:

```
remainsE :: [Int] -> NTransitionFunction -> NTransitionFunction
remainsE q n = n ++ [((x,'e'),[x])|x<-q, (x,'e') `notElem` map fst n]
```

Result looks like:

```
*MyCode> remainsE [1..8] (map addE (combineNTrans exNTF))
[((1,'e'),[1,2,8,4,3]),((3,'0'),[4]),((4,'0'),[5]),((5,'0'),[7])
,((5,'1'),[6]),((6,'0'),[7,8]),((8,'e'),[8,2]),((2,'e'),[2])
,((3,'e'),[3]),((4,'e'),[4]),((5,'e'),[5]),((6,'e'),[6]),((7,'e'),[7])]
```

Now combine all funcitons to get our final `regToNFA'` function:

```
regToNFA' :: Reg -> NFA
regToNFA' x = NFA nq nsig ns nf nt' where
  NFA nq nsig ns nf nt = regToNFA x
  nt' = remainsE nq (map addE (combineNTrans nt))
```

Final result looks like:

```
*MyCode> regToNFA' exRE5
NFA {n_states = [1,2,3,4,5,6,7,8], n_alphabet = "e01",
n_start = 1, n_finals = [8],
n_trans = [((1,'e'),[1,2,8]),((2,'1'),[4,3]),((3,'0'),[4])
,((4,'0'),[5]),((5,'0'),[7]),((5,'1'),[6]),((6,'0'),[7])
,((7,'e'),[7,8]),((8,'e'),[8,2]),((2,'e'),[2]),((3,'e'),[3])
,((4,'e'),[4]),((5,'e'),[5]),((6,'e'),[6])]}
```

## 3.2 NFA to DFA - powerset construction

Now we are going to convert NFA to DFA by using the powerset construction. In this section, PFA means a DFA constructed by powerset construction. PFA is basically a special case of DFA, which has sets of states from NFA as its states. We defint the data for PFA:

```
type PTransitionFunction = [(([Int], Char), [Int])]

data PFA = PFA{p_states :: [[Int]]
               ,p_alphabet :: Alphabet
               ,p_start :: [Int]
               ,p_finals :: [[Int]]
               ,p_trans :: PTransitionFunction}
               deriving Show
```

To get `nfaToPFA`, we need some fucntions below. Combining these functions, we can get the states for PFA, which are in the form $[[a,...,b],...,[c,...,d]]$.

```
nexte :: Int -> NTransitionFunction -> [Int] --'e'
nexte a ntf
  |(a,'e') `elem` map fst ntf = ntf ! (a,'e')
  |otherwise  = []

next1 :: Int -> NTransitionFunction -> [Int] --'1'
next1 a ntf
  |(a,'1') `elem` map fst ntf = ntf ! (a,'1')
  |otherwise  = []

next0 :: Int -> NTransitionFunction -> [Int] --'0'
next0 a ntf
  |(a,'0') `elem` map fst ntf =  ntf ! (a,'0')
  |otherwise  = []

ntT :: (Int -> NTransitionFunction -> [Int])
       -> [Int] -> NTransitionFunction -> [Int]
ntT t x ntf = nt_te where
  nte a = nexte a ntf
  tt  a = t a ntf
  nt_tn = toSet(concatMap nte (concatMap tt x))
  nt_te
    |subset (toSet(concatMap nte nt_tn)) nt_tn = nt_tn
    |otherwise = ntT nexte nt_tn ntf

pq :: [[Int]] -> NTransitionFunction -> [[Int]]
pq [[]] _ = [[]]
pq x ntf
  |subset (tS x ntf) x = toSet x
  |otherwise = toSet (tS x ntf ++ pq (x ++ tS x ntf) ntf)

tS :: [[Int]] -> NTransitionFunction -> [[Int]]
tS x ntf  = toSet (map nt_t1 x ++ map nt_t0 x) where
  nt_t1 s = ntT next1 s ntf
  nt_t0 s = ntT next0 s ntf
```

Now we can get the transition map from a state to another state.(The most important part.) And therefore get the `nfaToPFA` function.

```
nfaToPFA :: NFA -> PFA
nfaToPFA (NFA _ _ _ nf nt) = PFA p_q sig p_s p_f p_t where
  p_q = pq [start1] nt
  sig = ['0','1']
  p_s = start1
  p_f = [xf | xf <- p_q, (\ [x] -> x) nf `elem` xf]
  p_t = [((x,'1'),ntT next1 x nt) | x <- p_q]
        ++ [((x,'0'),ntT next0 x nt) | x <- p_q]
  start1 = ntT nexte [1] nt
```

The results looks like:

```
*MyCode> nfaToPFA (regToNFA' exRE4)
PFA {p_states = [[3,4,2],[],[1,2,4]], p_alphabet = "01"
, p_start = [1,2,4], p_finals = [[3,4,2],[1,2,4]]
, p_trans = [(([3,4,2],'1'),[3,4,2]),(([],'1'),[])
,(([1,2,4],'1'),[3,4,2]),(([3,4,2],'0'),[]),(([],'0'),[])
,(([1,2,4],'0'),[])]}
```

Finally we give a function to implement NFA. It is identical to the fucntion for DFA:

```
acceptingInPFA :: PFA -> String -> Bool
acceptingInPFA (PFA _ _ s f t) = state s
  where  state q "" = q `elem` f
```

```
        state q ('1':xs) = state (t ! (q,'1')) xs
        state q ('0':xs) = state (t ! (q,'0')) xs
        state _ _ = error "unkonwn symbol"
```

```
exRE6 :: Reg
exRE6 = Alt zero1 one0 where
  zero1  = Kle (Con (L '0') (L '1'))
  one0   = Kle (Con (L '1') (L '0'))
```

Some result:

```
*MyCode> acceptingInPFA (nfaToPFA (regToNFA' exRE6)) "01010101"
True
*MyCode> acceptingInPFA (nfaToPFA (regToNFA' exRE6)) "1010101010"
True
```

# 4 Future work

In this article, we only use '0' and '1' as our working symbols. It is left to generalize our working symbols to arbitrary many symbols. We only finished one direction that converts RE to NFA to DFA. The other direction, DFA to RE, needs more works to accomplish. PFA to simple DFA(with integers as states) is also a task which needs some extra work. This article lacks an Arbitrary instance for QuickCheck and can be added after.

# References


[1] Alfred V Aho. *Compilers: principles, techniques and tools (for Anna University), 2/e*. Pearson Education India, 2003.

[2] Barry Brown. Regular expression to nfa, 2011. `https://www.youtube.com/watch?v=RYNN-tb9WxI`.

[3] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009. See in particular Section 26.2, "The Floyd–Warshall algorithm", pp. 558–565 and Section 26.4, "A general framework for solving path problems in directed graphs", pp. 570–576.

[4] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65, 2001.

[5] Miran Lipovaca. *Learn you a haskell for great good!: a beginner's guide*. no starch press, 2011. ISBN-13: 978-1-59327-283-8.

[6] John C Martin. *Introduction to Languages and the Theory of Computation*, volume 4. McGraw-Hill NY, USA, 1991. p. 108.

[7] Michael O Rabin and Dana Scott. Finite automata and their decision problems. *IBM journal of research and development*, 3(2):114–125, 1959.

[8] Michael Shaw. Replacing items in a list, 2018. `https://codereview.stackexchange.com/questions/85703/replacing-items-in-a-list`.

[9] Michael Sipser. Chapter 1: Regular languages. *Introduction to the Theory of Computation*, pages 31–90, 1998.

[10] Simon Thompson. Regular expressions and automata using haskell. 2000. `http://cs.nyu.edu/totok/professional/software/tpcw/tpcw.html`.

[11] Jay Yellen and Jonathan L Gross. *Handbook of Graph Theory (Discrete Mathematics and Its Applications)*. CRC press, 2003. sect.2.1, remark R13 on p.65.