From regex to NFA and back

Jules Jacobs

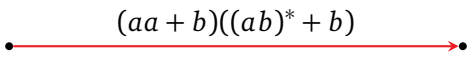
April 27, 2021

Abstract

The traditional algorithms taught for converting a regex to an NFA and back are Thompson's construction and Kleene's algorithm, respectively. In this note I'll advocate a small modification to that approach that I claim makes the algorithms simpler and easier to understand and implement, and produces smaller automata.

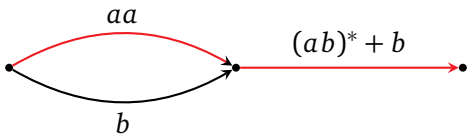
1 Regex to NFA by example

To convert the regular expression $(aa + b)((ab)^* + b)$ to an NFA, we place it on an edge between the start and end nodes:

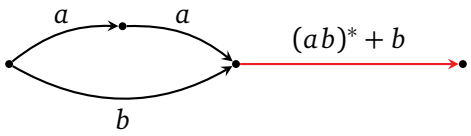


We split the sequencing into two edges by inserting a node in the middle:

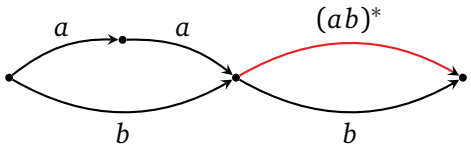
We split the sum $aa + b$ into two parallel edges:



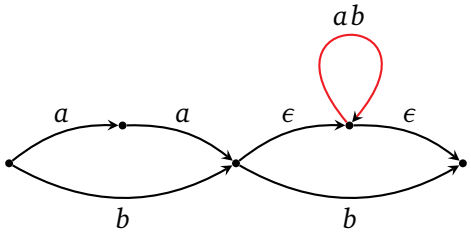
The black b edge is now complete and will be part of the final NFA.
We split the aa into two:



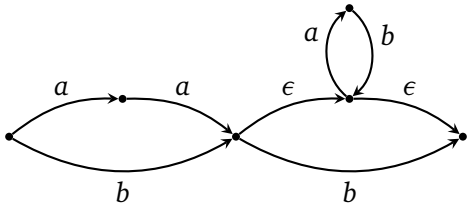
The left hand side is complete.
We continue working on the right hand side:



To eliminate the repetition $(ab)^*$, we put a new node in the middle of the edge, and add ϵ-transitions. We put what's inside of the repetition (ab in this case) on a self-loop of the new node:



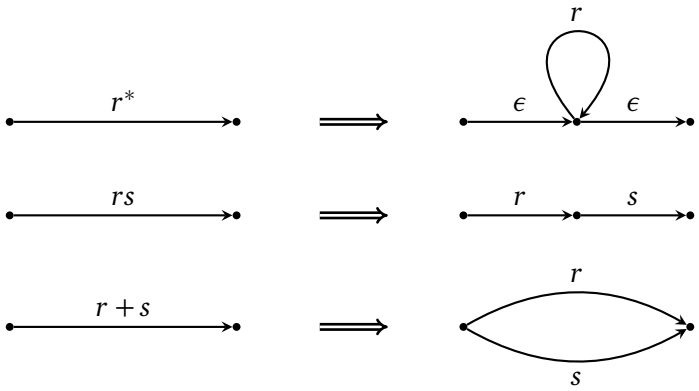
The final step is to eliminate the last remaining red ab edge:



That's it, we've converted the regex to an NFA.

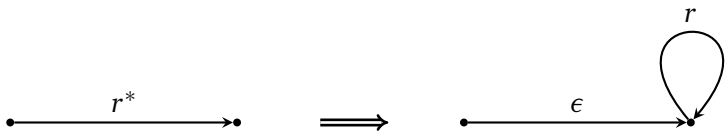
2 The rules of the game

These are the rules we used:



In each case the start and end are allowed to be the same node. You should convince yourself that **regardless of the rest of the NFA, these transformations preserve the language between the nodes.**

Question: What can go wrong if we use this rule instead?

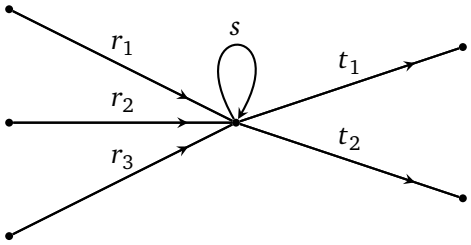


Hint: consider $a^* + b^*$.

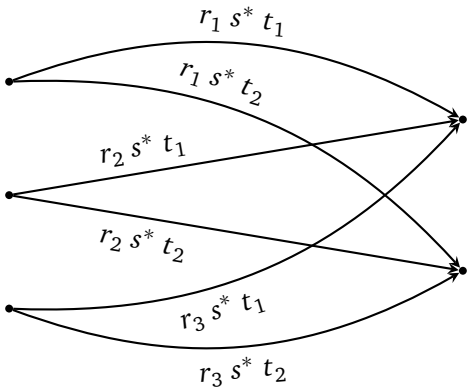
Question: Can you come up with safe rules that avoid the extra node for r^* in some cases?

3 From NFA back to regex

Take an arbitrary node inside an NFA:

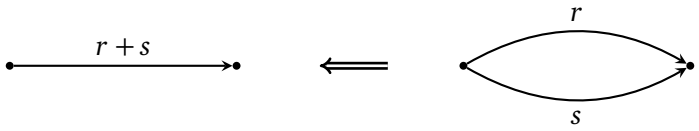


We can eliminate the middle node by inserting shortcut edges:



For each incoming edge $\xrightarrow{r}$ and outgoing edge $\xrightarrow{t}$ we insert the shortcut edge $\xrightarrow{rs^*t}$. If the node doesn't have a self loop, we take $\xrightarrow{rt}$. Alternatively, we can insert ϵ self loops everywhere.

We also need to perform the sum rule in reverse, to combine multiple parallel edges into one:



To find a regex representing the language from node x to y in an NFA:

1. Create two new **start** and **end** nodes, and insert edges **start** $\xrightarrow{\epsilon} x$ and $y \xrightarrow{\epsilon}$ **end**.
2. Eliminate all nodes from the NFA, including x and y themselves, but keep **start** and **end**.
3. Combine parallel edges whenever possible.
4. We're left with a single edge **start** $\xrightarrow{r}$ **end**, labeled with the regex we're looking for.

If we want to find a regex for multiple start nodes $\{x_i\}$ and multiple end nodes $\{y_i\}$, then we insert multiple edges **start** $\xrightarrow{\epsilon} x_i$ and $y_i \xrightarrow{\epsilon}$ **end**.

4 How to implement regex to NFA conversion

Define a data type for regular expressions:

```
class Re
object Emp extends Re
object Eps extends Re
case class Chr(a:Char) extends Re
case class Seq(a:Re, b:Re) extends Re
case class Alt(a:Re, b:Re) extends Re
case class Star(a:Re) extends Re
```

We label NFA nodes with numbers $1 \ldots n$ and store edges $i \xrightarrow{r} j$ in a finite map.

Our NFAs have three primitive operations:

- `fresh()` creates and returns a fresh node.
- `get(i,j)` looks up the regex on edge $i \to j$, and returns $\emptyset$ if the edge is not present.
- `add(i,j,r)` adds regex r on the edge $i \to j$, sums it with what was already on the edge.

```
class NFA {
  var n = 0
  var edges = Map[(Int,Int),Re]()
  def fresh() = { n += 1; n }
  def get(i:Int, j:Int) = edges.getOrElse((i,j),Emp)
  def add(i:Int, j:Int, r:Re) = edges += (i,j) → Alt(get(i,j),r)

  ...
}
```

To convert a regex to an NFA by the process described in sections 1 & 2, we define a recursive function `addRe(i,j,r)` that adds `r` to the NFA while only inserting edges with characters or ϵ on them. Note that the red edges of section 1 are never explicitly represented in the NFA; they are simply calls to `addRe`.

```
  def addRe(i:Int, j:Int, re:Re):Unit = {
    re match {
      case Emp ⟹
      case Eps ⟹
        add(i,j,Eps)
      case Chr(c) ⟹
        add(i,j,Chr(c))
      case Seq(a,b) ⟹
        val mid = fresh()
        addRe(i,mid,a)
        addRe(mid,j,b)
      case Alt(a,b) ⟹
        addRe(i,j,a)
        addRe(i,j,b)
      case Star(a) ⟹
        val mid = fresh()
        add(i,mid,Eps)
        add(mid,j,Eps)
        addRe(mid,mid,a)
    }
  }
```

That's all there's to it. What's implemented here is slightly different than in sections 1 & 2 because we never insert parallel edges. Instead, each edge will end up with characters $c_1 + c_2 + \cdots + c_n$ $(+\epsilon)$. In a practical implementation we'd want to represent this with a set data structure that can also represent character ranges `a-z` efficiently, particularly when we're using unicode.

5 How to implement NFA to regex conversion

To eliminate a node from the NFA we:

1. Collect all the self/in/out edges of the node
2. Delete all those edges from the NFA
3. Insert the shortcut edges

```
def elim(i:Int) = {
  // Find the self/in/out edges connected to i
  val self = get(i,i)
  val in = edges.collect{case ((a,b),r) if a != i && b == i ⟹ (a,r)}
  val out = edges.collect{case ((a,b),r) if a == i && b != i ⟹ (b,r)}

  // Delete all those edges
  edges -= (i,i)
  for((a,_) ← in) edges -= (a,i)
  for((b,_) ← out) edges -= (i,b)

  // Insert shortcut edges
  for((a,r) ← in; (b,s) ← out)
    add(a,b,Seq(r,Seq(Star(self),s)))
}
```

To convert an NFA with a given set of start and end nodes to a regex:

1. We add **start** and **end** nodes with ϵ-transitions to and from the set of start and end nodes.
2. We eliminate all the other nodes.
3. We return the regex r on the **start** $\xrightarrow{r}$ **end** edge.

```
def toRe(initials: Set[Int], finals: Set[Int]) = {
  // Add a new start and end node and
  // connect them to the initial and final states
  val start = fresh()
  val end = fresh()
  for(a ← initials) add(start,a,Eps)
  for(a ← finals) add(a,end,Eps)

  // Eliminate all nodes except start and end
  for(i ← 1 to start-1) elim(i)

  // Return the only edge left in the NFA
  get(start,end)
}
```

That's all.

6 Comparison with the standard approach

A representative standard text on this is, I believe, *Introduction to Automata Theory, Languages, and Computation* by Hopcroft, Motwani, and Ullman. The main differences are:

- Our NFAs have regexes on the edges from the start. Standard NFAs are simply those that happen to have only characters and ϵ on the edges.
- Our NFAs have no start and end nodes. We don't have *the* language of an NFA. We only have the language *between* two nodes.

I believe this has several advantages:

- The NFAs produced by this construction are more compact than those produced by Thompson's construction. Thompson's construction results in strictly more nodes and more ϵ-transitions. For the example of section 1, it creates 18 nodes and 15 ϵ-transitions.
- The construction is, in my opinion, conceptually easier to understand: each move of the game keeps the language between each pair of nodes the same.
- The regex to NFA algorithm makes it easier to understand the NFA to regex algorithm: it's basically the same algorithm in reverse.
- Both algorithms are very easy to implement.

Introduction to Automata Theory, Languages, and Computation says:

"The construction of a regular expression to denote the language of any DFA is surprisingly tricky"

It first gives an intricate construction spanning 5 pages of inductive proof. It then gives an alternative construction based on the elimination of states, but that is considerably complicated by the presence of multiple end nodes[1]. They run the elimination process once for every end node, and then combine the results. Furthermore, they don't eliminate all nodes, and are instead left with an automaton with two states and two self loops r and u and two edges s and t going back and forth. They then convert this to the regex $(r + su^*t)^*su^*$. They further need a special case for when the start node is the same as the end node. None of this is necessary if you insert a new **start** and **end** node with ϵ-transitions: you only do the elimination process *once* and you're left with an NFA with only a single edge, and the regex you're looking for is on that edge.

I don't claim that anything I've said here is new; it is extremely likely that it is not. Nevertheless, I couldn't find it in the standard texts or wikipedia[2]. Perhaps we could make the lives of millions of computer science students a tiny bit easier?

7 Code

The Scala code can be found at http://julesjacobs.com/notes/nfa/nfa.sc.

[1] Most presentations have a single start node and multiple end nodes; why the asymmetry?

[2] It is, of course, also possible that the reason for this is that what I wrote here is nonsense.

References

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.