CS3133 Foundations of Computer Science HW2

Keith DeSantis

8/30/21

∅ ≠ Σ λ ⊆ ⊄ ϵ δ

Collaboration with Samuel Parks and Kush Shah

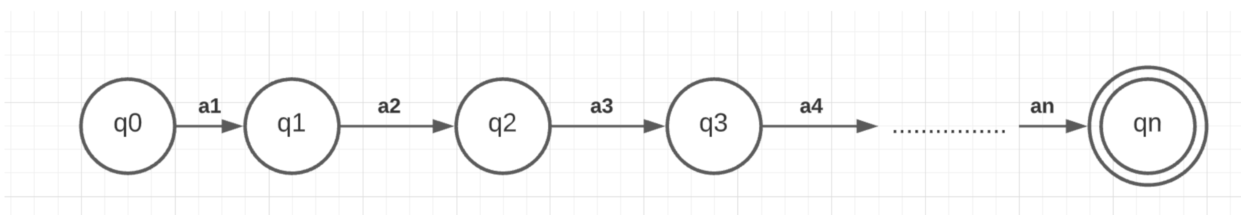
MakeNFAs)

a)
Let K consist of the string $x = a_1 \ldots a_n$ where $a_i$ is an element of Σ.

We can build an NFA M whose language is {x} as follows,

M will have a number of states equal to n+1 where n is the length of our string x, and will progress through them linearly, as shown below.

Starting at $q_0$, each state $q_i$ will have only one transition to the next state that reads $a_{i+1}$ to arrive at state $q_{i+1}$, until $q_n$ which will be the only accepting state and has no transitions out of it.




This way, any string other than x will either end its run on a non-accepting state or block the machine as the transitions are strictly formed to only allow an accepting run with x as an input.

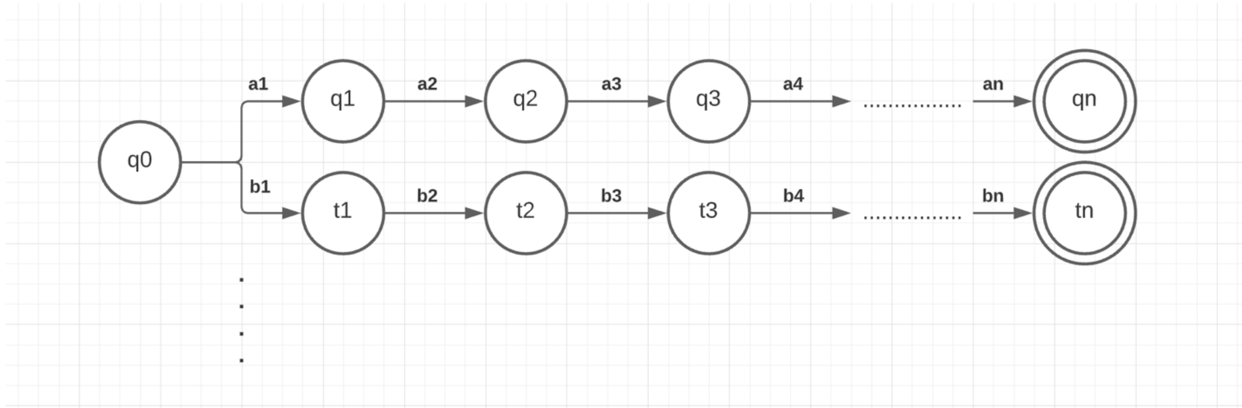
Therefore, K is regular.

b) Let K be a finite language, that is, K is a language containing finitely many strings. Show that K is regular.

By the previous part, there are k NFAs accepting precisely the individual strings $x_0$, $x_1$, $\ldots x_m$ where $x_0 - x_m$ are the strings contained in K.

Therefore, by including all k NFAs that accept $x_0, x_1, \ldots x_m$ but combining their starting states into one state, $q_0$, that has a unique transition to each smaller NFA $k_i$

that reads the first symbol of $x_i$, we are left with one NFA with a language of exactly K.




Therefore, K is regular.

c) Let K be cofinite language, that is, K is a language whose complement contains only finitely many strings. Show that K is regular.

   By theorem, if a language, A is regular then its complement, A' is regular.

   By the previous part, a language of finitely many strings, K, is regular. Therefore, K' is regular.

   By simply reversing our naming convention such that the finite language is called K' and the cofinite language is called K, it follows that K is regular.

NFAUnionBigO)

1) Algorithm A:

   Let m = number of states in M, n = number of states in N

   $O(2^m)$ for the subset construction of M, M'.
   $O(2^n)$ for the subset construction of N, N'.
   O(mn) for the product construction of M' and N'.
   So, in total the algorithm takes:

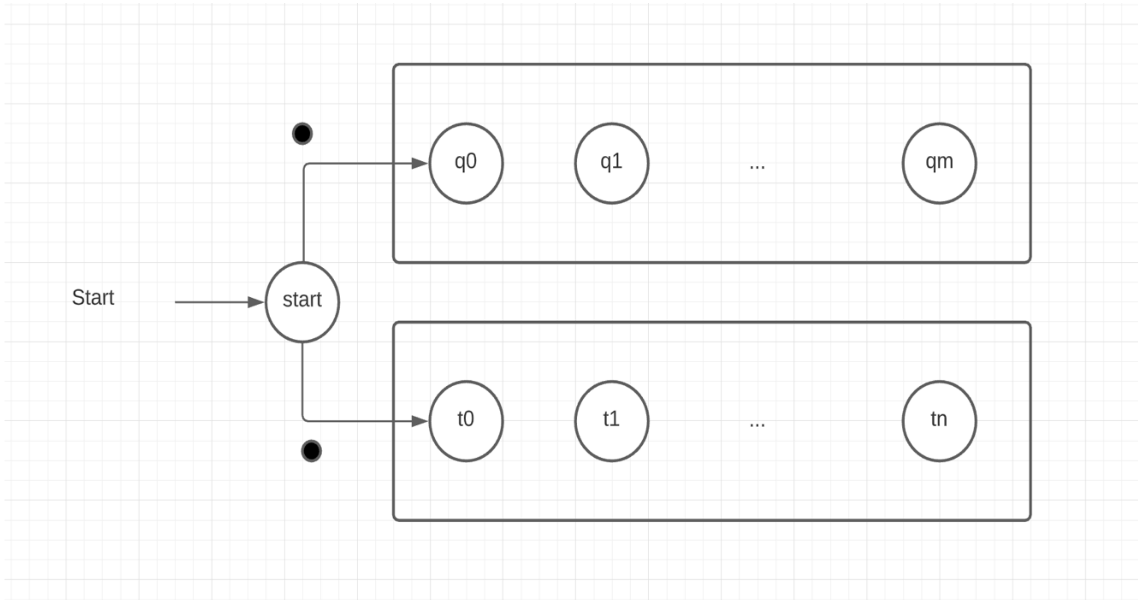
   $O(2^m + 2^n)$ time since $O(2^m)$ and $O(2^n)$ are the dominant time complexities.

2) Algorithm B:

   Let m = number of states in M, n = number of states in N

   O(m+n) for the call of algorithm 3 on M and N.

From here we have our resulting NFA$_\lambda$ for M U N, of the following form:




Where states $q_0$ to $q_m$ form M and states $t_0$ to $t_n$ form N.

From here we employ Algorithm 6 to transform this NFA$_\lambda$ to an NFA, R.

Since this is the worst-case asymptomatic analysis, we assume the most iterations of Algorithm 6 as possible, meaning there is a transition between $q_0$ and every $q_i$, as well as between $t_0$ and every $t_i$.

O(m-1) for the creation of new transitions from the start state to each q state, since a connection to each q state other than $q_0$ is needed.

O(n-1) for the creation of new transitions from the start state to each t state, for the same reason as above.

O(2) for removal of the two $\lambda$ transitions.

From here we must convert our new NFA to the final DFA (D).

O($2^r$) for the subset construction on R, where r is the number of states in R.

r = m + n + 1 since Algorithm 3's construction adds one additional state.

So, the algorithm takes:

$O(2^{m+n+1})$ time since $O(2^{m+n+1})$ is the dominant time complexity.

Based on these calculations, Algorithm 1 is a preferable algorithm in the sense of asymptomatic running time since $O(2^m + 2^n) < O(2^{m+n+1})$.

RegExpCompare)

a) **O*** and λ*

They denote the same language.

**O*** = L(**O**)* = ∅* = { λ }
λ* = { λ }
Therefore, **O*** denotes the same language as λ*.

b) (**a** + **b**)* and **a*** + **b***

**a*** + **b*** is a proper subset of (**a** + **b**)*

The string 'abbbaaba' is in (**a** + **b**)* but not **a*** + **b***.

c) (**a*b**)* and (**a*b***)*

(**a*b**)* is a proper subset of (**a*b***)*

'a' is a string that is in (**a*b***)* that isn't in (**a*b**)*.

d) (**ab** + **a**)* and (**ba** + **a**)*

Neither language is a subset of each other.

'ab' is a string in (**ab** + **a**)* but not in (**ba** + **a**)*.
'ba' is a string in (**ba** + **a**)* but not in (**ab** + **a**)*.

e) **a*ba*b**(**a** + **b**)* and (**a** + **b**)***b**(**a** + **b**)***b**(**a** + **b**)*

They are the same language.

Let E = (**a** + **b**)***b**(**a** + **b**)***b**(**a** + **b**)*
Let R = **a*ba*b**(**a** + **b**)*

Proof that R ⊆ E:

R and E can be divided as such ( | added for visual distinction):

$(a + b)^* \mid b \mid (a + b)^* \mid b \mid (a + b)^*$

$\quad a^* \qquad \mid b \mid \quad a^* \qquad \mid b \mid (a + b)^*$

The only parts that differ are $(a + b)^*$ in E and $a^*$ in R.

$a^* \subseteq (a + b)^*$, intuitively.

Therefore, $R \subseteq E$.

Proof that $E \subseteq R$:

Both expressions guarantee that every string they could produce will have two b's in them (from the only two sections of either expression without a * being two distinct b's).

Any string built from these expressions can therefore be separated into three sections as such:

Section 1 | first required b | Section 2 | second required b | Section 3

The sections are defined as such:

R: $a^*$ | first b | $a^*$ | second b | $(a + b)^*$

E: $(a + b)^*$ | first b | $(a + b)^*$ | second b | $(a + b)^*$

While it may at first look like E is a more generic form of R, and therefore may contain it as a subset, the fact that both RegExps must contain two b's means that any string from E can be interpreted in such a way that it is also an element of R.

There are two cases:

Case 1 - If a string that is in E starts with an 'a' (a…b…b…):

> The next 'b' read in the string can be defined as the first required b for R. From there continue reading symbols until another 'b' is read. This can be defined as the second required b for R, then anything beyond that symbol is valid in R since the final section is $(a + b)^*$. We know at least two b's will be read because all strings from E contain at least 2 b's as stated above.

Case 2 - If a string that is in E starts with an ‘b’ (b…b…b…):

This starting ‘b’ can be defined as the first required b for R (meaning the first section a* happens to have no concatenations on a). From there read until another ‘b’ is read, this too will be defined as the second b for R, and as above anything after this is acceptable since section 3 of R is (a + b)*.

Therefore, any element of E is also in R, or $E \subseteq R$.

Since $E \subseteq R$ and $R \subseteq E$, $E = R$.

Therefore, L(E) = L(R).

***Note: I understand that the section of this proof where I show $E \subseteq R$ is more of an explanation than a mathematical proof, I constructed a more mathematical proof but felt it was over the top and a hassle to read. If you would still like to read it, it is at the end of the file, marked with “***”

f) **a*****ba*****b**(**a** + **b**)* and **a***(**a*****ba*****ba***)*

**a*****ba*****b**(**a** + **b**)* is a subset of **a***(**a*****ba*****ba***)*

λ is a string in **a***(**a*****ba*****ba***)* but not in **a*****ba*****b**(**a** + **b**)*.

g) (**ab**)***a** and **a**(**ba**)*

They denote the same set.

Let E = (**ab**)***a**
Let R = **a**(**ba**)*

Proof that $E \subseteq R$:

Let $x \in E$.

x is of the form:

ababab… a

By dividing x up as such –

a bababa…ba

- it becomes clear that $x \in R$.

Therefore, E ⊆ R.

Proof that R ⊆ E:

Let x ϵ R.

x is of the form:

a baba…. ba

By dividing x up as such –

abababa… a

- it becomes clear that x ϵ E.

Therefore, R ⊆ E.

Since R ⊆ E and E ⊆ R, E = R, therefore L(E) = L(R).

h) (**bba**)***bb** and **bb**(**abb**)*

They denote the same set.

Let E = (**bba**)***bb**
Let R = **bb**(**abb**)*

Proof that E ⊆ R:

Let x ϵ E.

x is of the form:

bbabbba… bb

By dividing x up as such –

bb abbabb…abb

- it becomes clear that x ϵ R.

Therefore, E ⊆ R.

Proof that R ⊆ E:

Let x ϵ R.

x is of the form:

bb abbabb…abb

By dividing x up as such –

bbabbabb… bb

- it becomes clear that x ϵ E.

Therefore, R ⊆ E.

Since R ⊆ E and E ⊆ R, E = R, therefore L(E) = L(R).

i) **a**(**bca**)***bc** and **ab**(**cab**)***c**

They denote the same set.

Let E = **a**(**bca**)***bc**
Let R = **ab**(**cab**)***c**

Proof that E ⊆ R:

Let x ϵ E.

x is of the form:

a bcabcabca … bc

By dividing x up as such –

ab cabcabcab … c

- it becomes clear that x ϵ R.

Therefore, E ⊆ R.

Proof that R ⊆ E:

Let x ϵ R.

x is of the form:

ab cabcabcab … c

By dividing x up as such –

a bcabcabca … bc

- it becomes clear that x ϵ E.

Therefore, R ⊆ E.

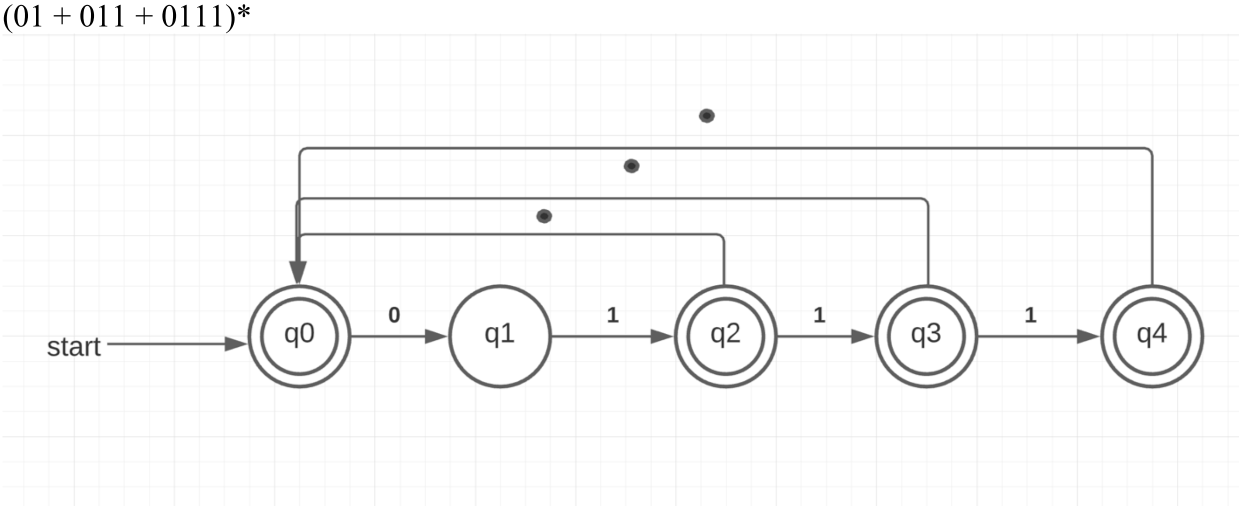
Since R ⊆ E and E ⊆ R, E = R, therefore L(E) = L(R).

MatchReqExpFA1)

$M_1$ : λ + **a(ab*b** + **aa)*ab***
$M_2$ : λ + **a(ba*b** + **aa)*a**
$M_3$ : λ + **a(ba*b** + **ba)*b**
$M_4$ : λ + **a(ab*b** + **aa)*a**
$M_5$ : λ + **a(ba*b** + **ba)*ba***

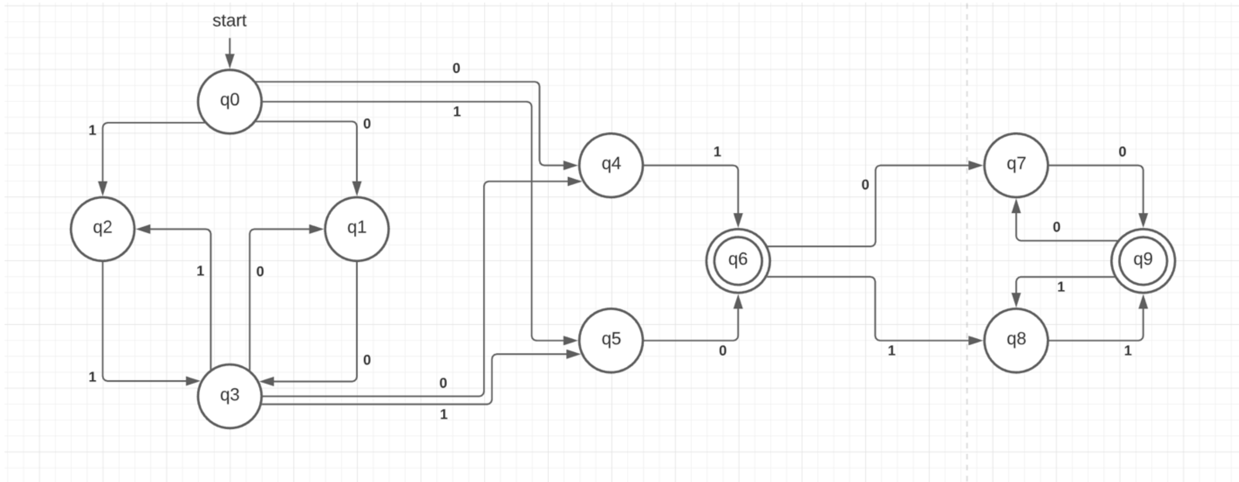
RegExpToNFA)

a) (01 + 011 + 0111)*



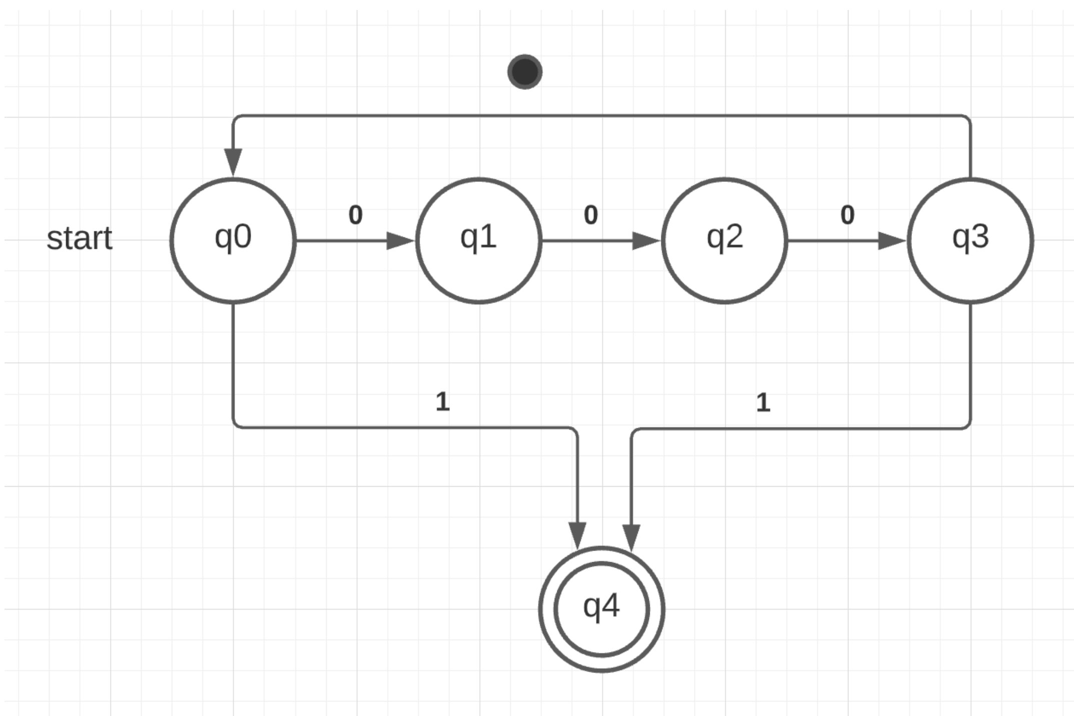

b) (00 + 11)*(01 + 10)(00 + 11)*

With some tinkering I was able to get this one:

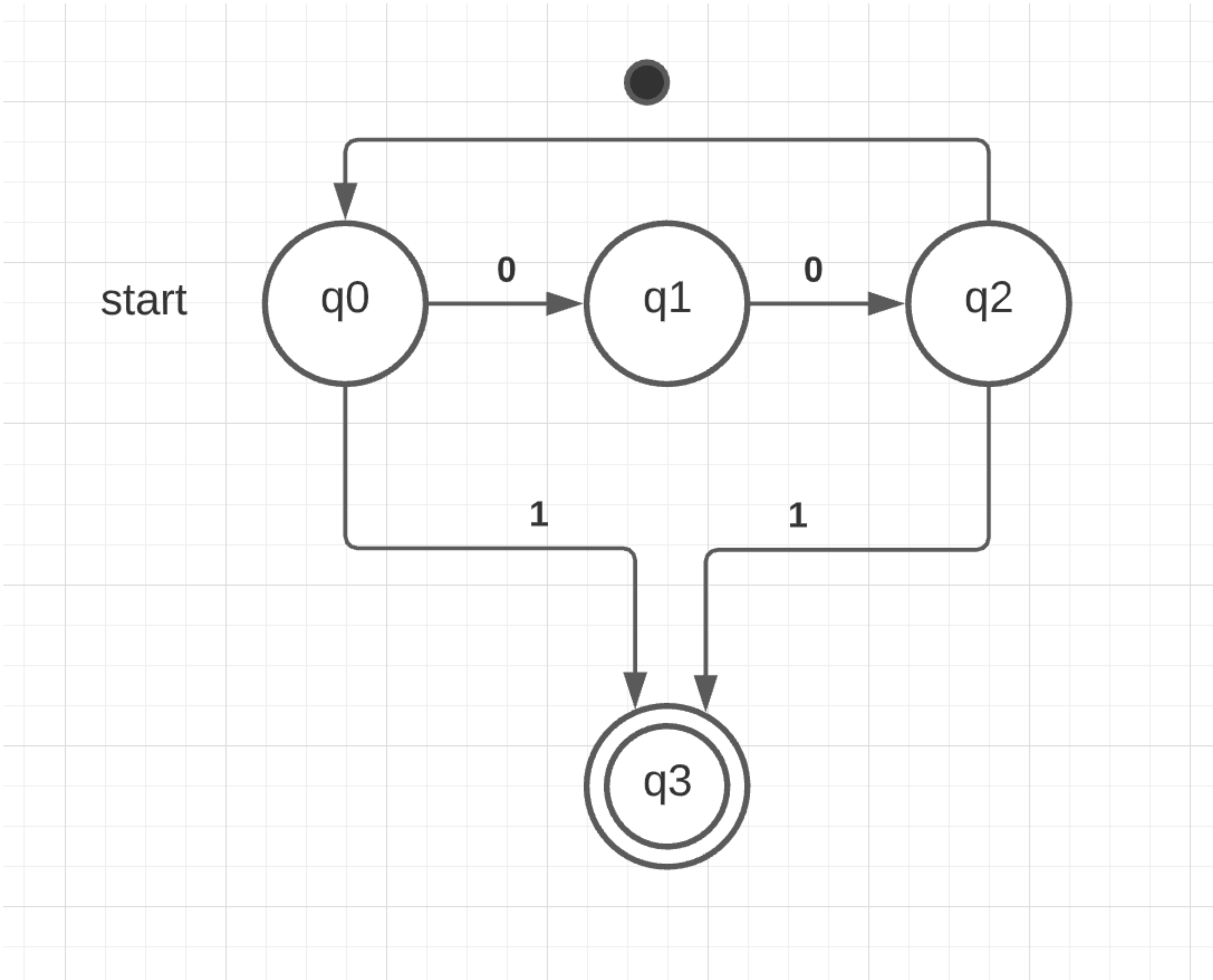



c) $(000)^*1 + (00)^*1$

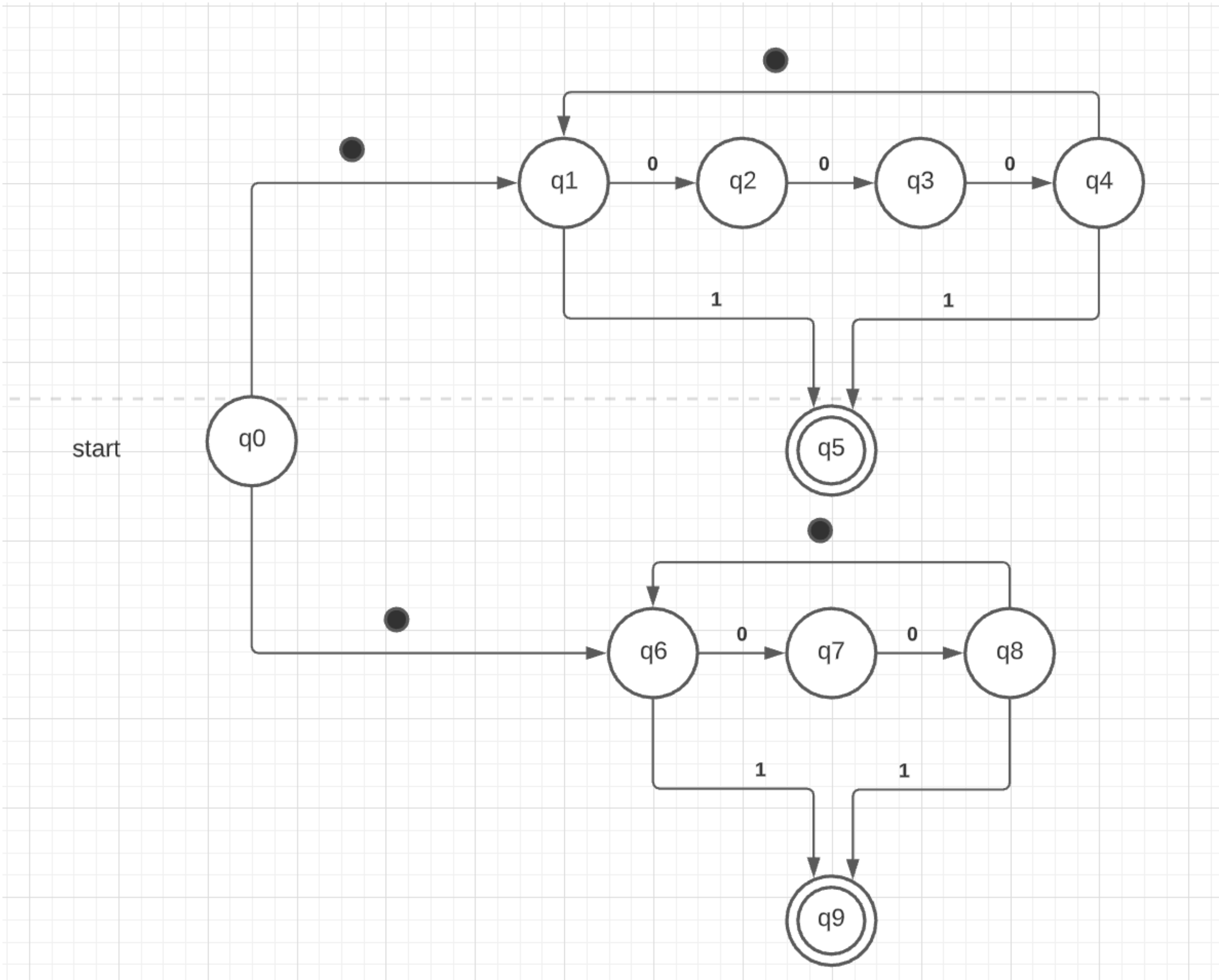
First, I construct an NFA$_\lambda$ $(000)^*1$:



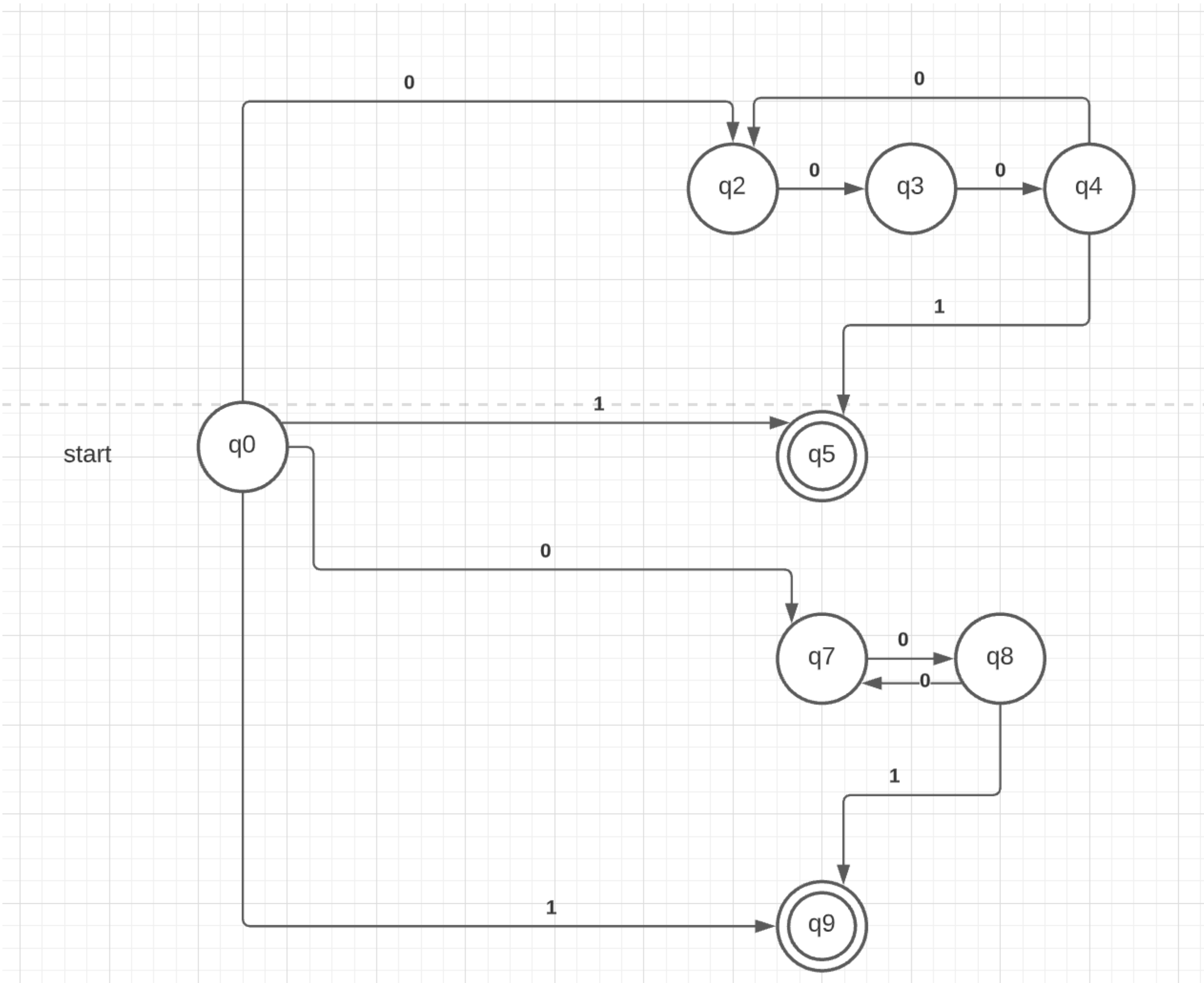

Then I construct an NFA$_\lambda$ $(00)^*1$:




Then, I combine them using a union construction for $NFA_{\lambda}$:



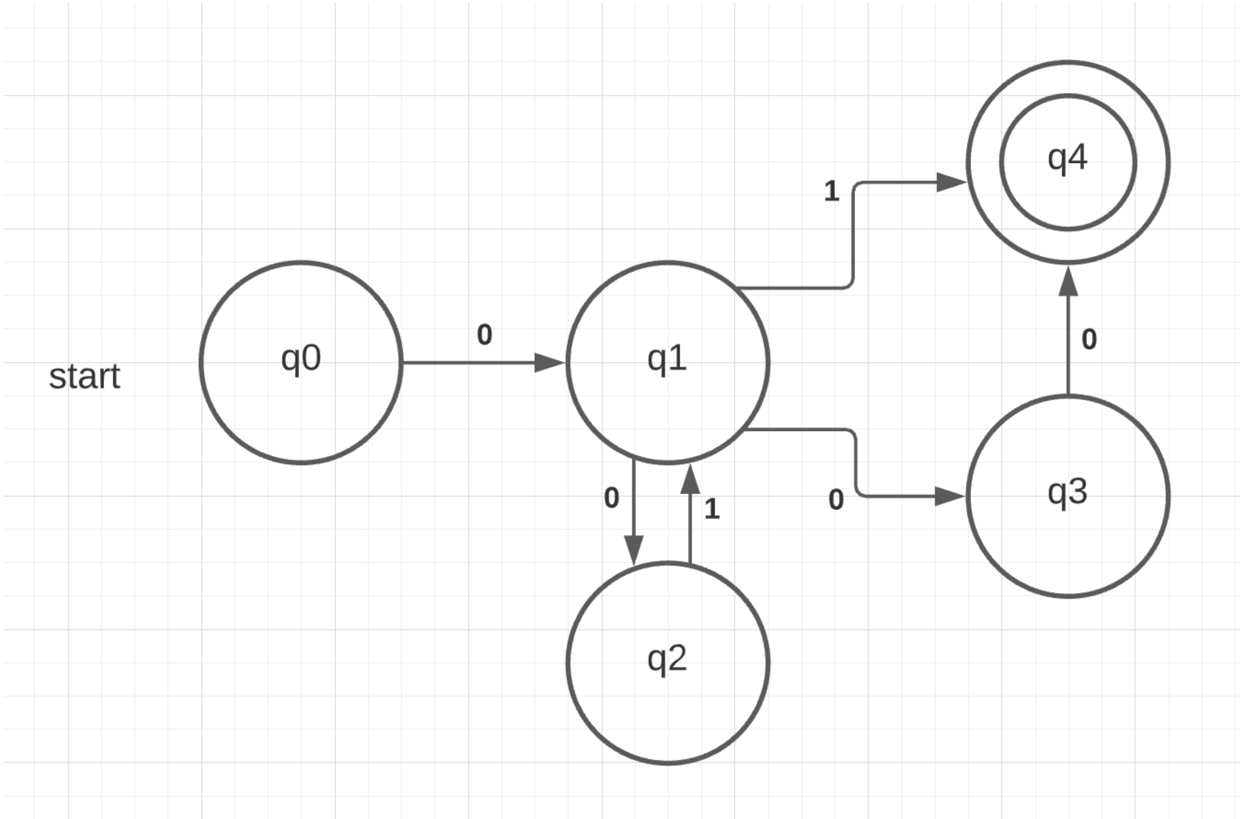

Then I perform the algorithm to convert an NFA$_\lambda$ to an NFA:



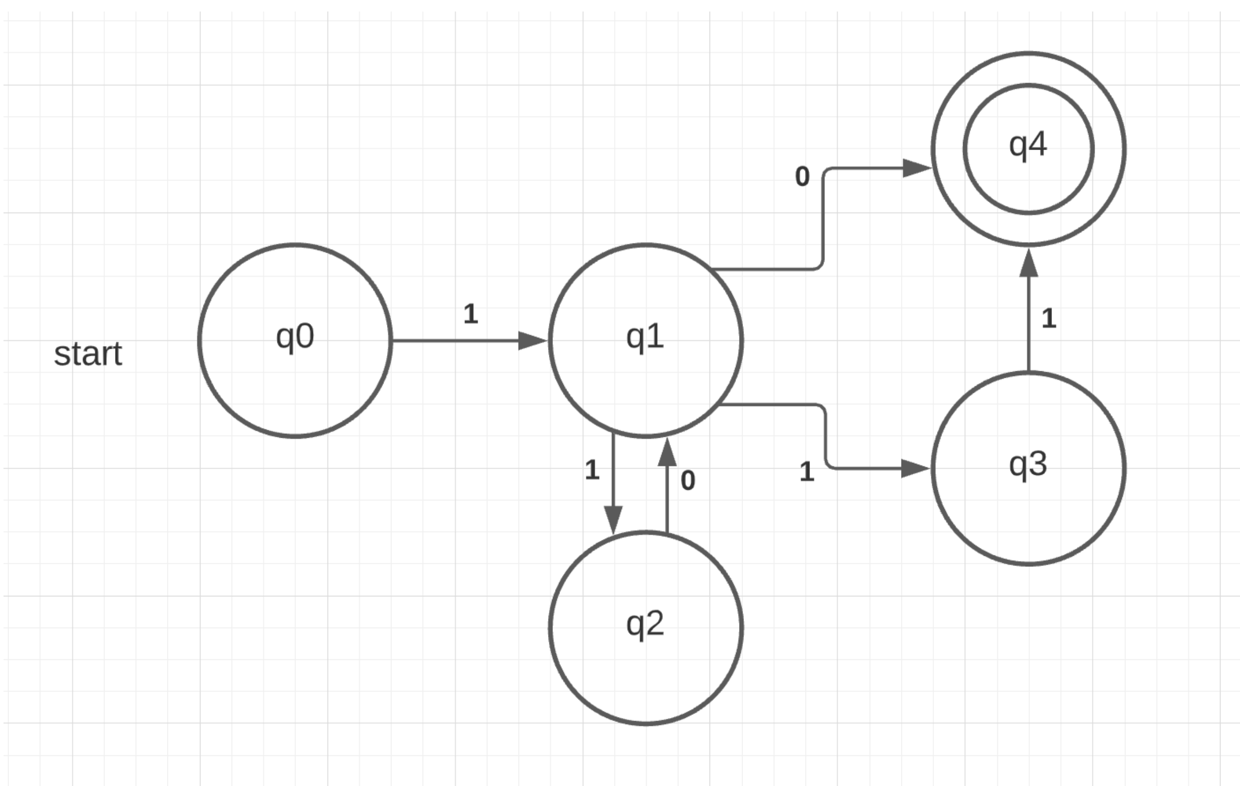

Once again, some states have been removed since they became unreachable through the union construction, but the resulting NFA accepts (000)*1 + (00)*1.

d) (0(01)*(1 + 00) + 1(10)*(0 + 11))*

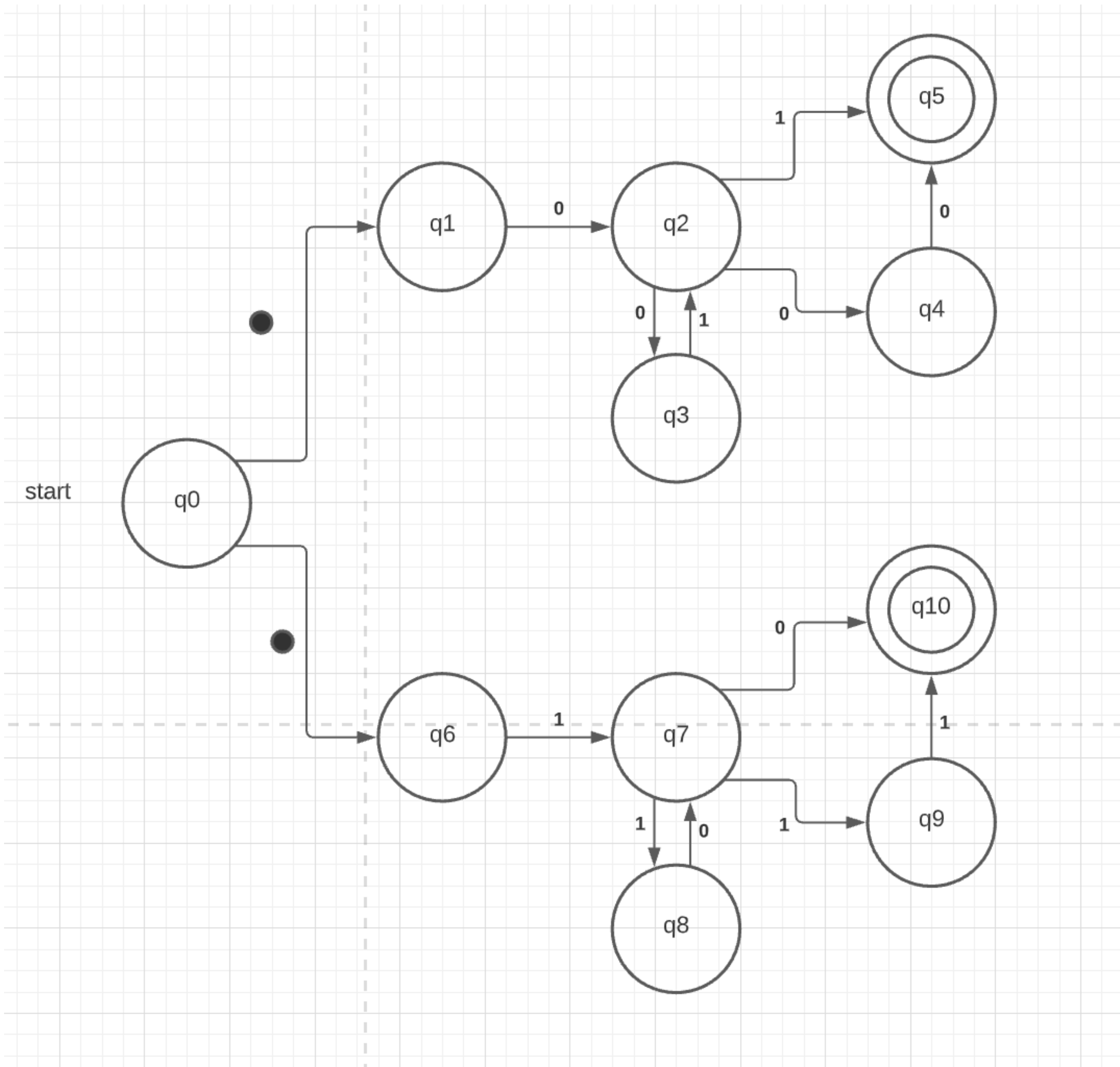
First I construct an $NFA_\lambda$ for 0(01)*(1 + 00) with some tinkering:



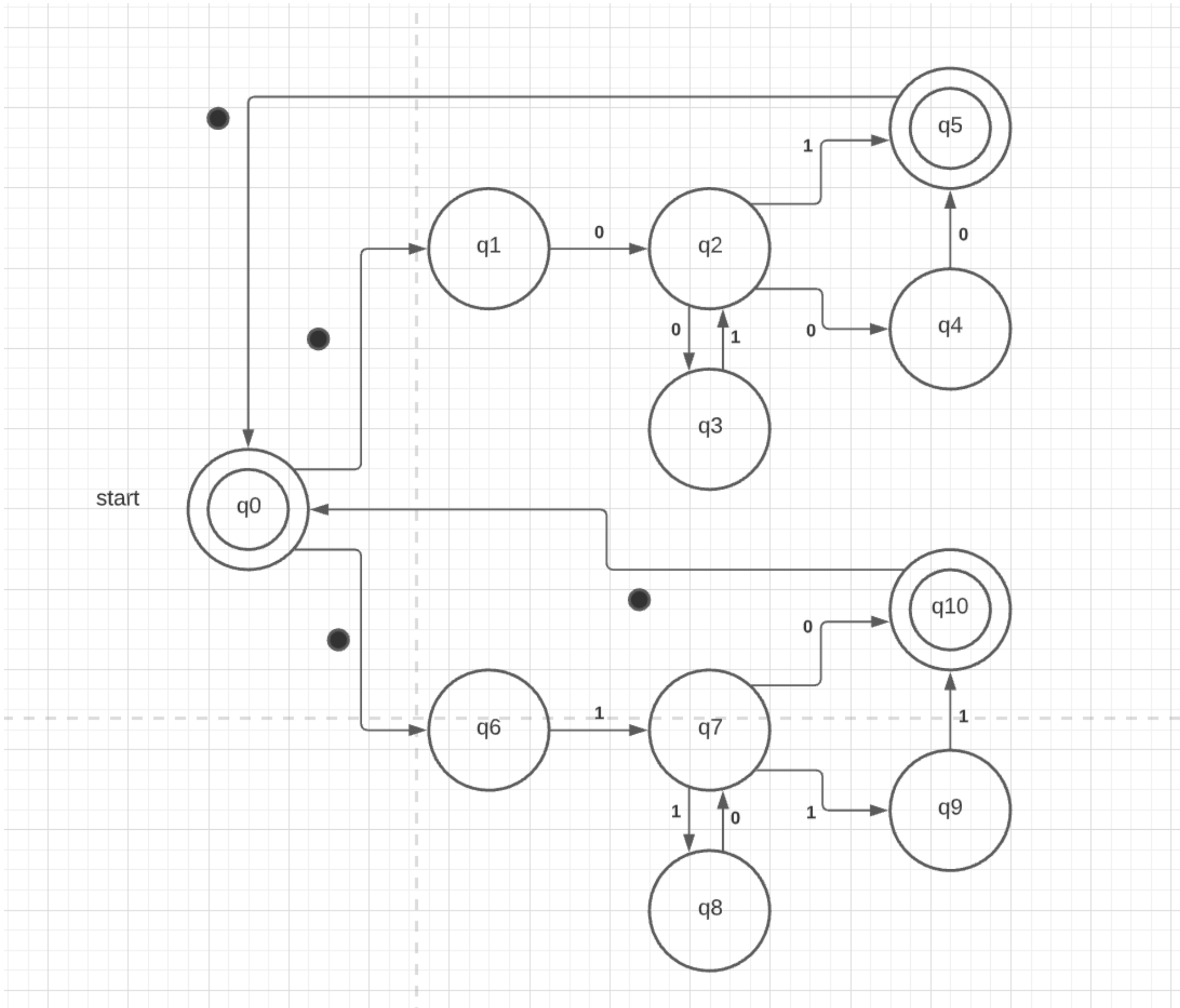

Then I construct an NFA$_\lambda$ for 1(10)*(0 + 11) with some tinkering:



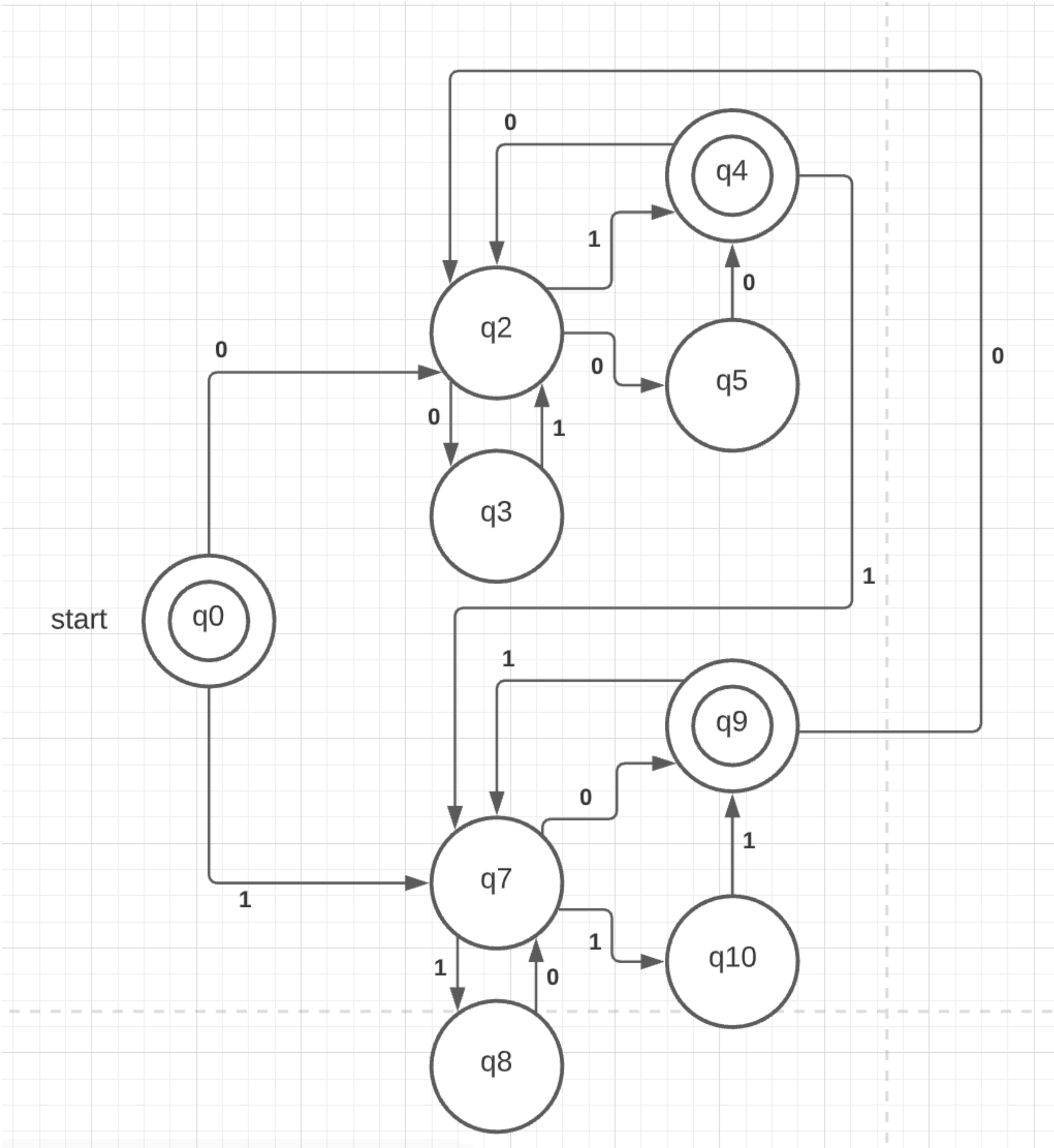
Then I perform the NFA$_\lambda$ union construction on the two NFA$_\lambda$'s:




Then I adjust the NFA$_\lambda$ to account for the outer * function:




Finally, I run the algorithm to convert the NFA$_\lambda$ to an NFA:




Once again, some states have been removed as they became unreachable after the algorithm, but the naming of states remained the same as in the NFA$_\lambda$ for consistency.

*** Note This is a mathematical proof attempt E ⊆ R for RegExpCompare part e, I felt it was overly complicated and not worth reading so I left a note by part e explaining this.

Let E = (**a** + **b**)***b**(**a** + **b**)***b**(**a** + **b**)*
Let R = **a*****ba*****b**(**a** + **b**)*

Proof by contradiction that E ⊆ R:

Let x ϵ E.
Let x not be an element of R.

By the defined regular expressions for R and E, any string in them must contain at least 2 b's, since each contains two un-starred b's.

Given this, any element of either expression can be written as:

wbybz, where w, y, and z are strings.

Therefore, x = wbybz.

There are three possible cases:

Case 1 – x begins with an 'a':

x = a w' b y b z, where w' is a substring of w.

Given that x is not an element of R, this means either w' or y must contain a 'b' since they are both denoted in the regular expression as **a*** (z is denoted as (**a** + **b**)* and therefore can be any string).

There are three possible cases to be considered:

Case 1a – w' contains one 'b':

w' can be written as a*ba*.

We can then rewrite x as: x = aa*ba*bybz.

However, we can see that x then fits the definition of R, as such:

R = **a*** **b** **a*** **b** (**a** + **b**)*
x = aa* b a* b ybz

This contradicts our assumption that x is not an element of R, therefore x ϵ R.

Case 1b – w' contains more than one 'b':

w' can be written as a*ba*b(a + b)*, (written this way in an attempt to identify the first two b's in w').

We can therefore rewrite x as: x = aa*ba*b(a + b)*ba*bybz.

However, we can see that x then fits the definition of R, as such:

R = **a*** **b** **a*** **b** **(a + b)***
x = aa* b a* b (a + b)*ba*bybz

This contradicts our assumption that x is not an element of R, there for x ϵ R.

Case 1c – y contains one or more 'b':

y can be written as a*b(a + b)*.

We can then rewrite x as: x = aw'ba*b(a+ b)*bz.

However, we can see that x then fits the definition of R, assuch:

R = **a*** **b** **a*** **b** **(a + b)***
x = aw' b a* b (a + b)*bz

This contradicts our assumption that x is not an element of R, there for x ϵ R.

Therefore, if x begins with an 'a', x must be an element of R.

Case 2 – x begins with a 'b':

x = b w' b y b z, where w' is a substring of w.

There are two possible cases to consider:

Case 2a – w' contains no b's:

x = ba*bybz

We can see that x then fits the definition of R, as such:

R = **a*** **b** **a*** **b** **(a + b)***
x = λ b a* b ybz

This contradicts our assumption that x is not an element of R, therefore x ϵ R.

Case 2b – w’ contains one or more b’s:

w’ = a*b(a + b)*.

x = ba*b(a + b)*bybz.

However, we can see that x then fits the definition of R, as such:

R = **a*** **b** **a*** **b** **(a + b)***
x = λ b a* b (a + b)*bybz

This contradicts our assumption that x is not an element of R, therefore x ϵ R.

Therefore, if x begins with an ‘b’, x must be an element of R.

Therefore, if x is an element of E it must also be an element of R.

Therefore, E ⊆ R.

Since R ⊆ E and E ⊆ R, E = R, L(E) = L(R).