# DD2372 Lab 1 report

Forrest Timour

April 29, 2018

## Implementation

I decided to implement DFAs and NFAs using `structs` in C. Ignoring fields used for memory management in C, my representation was the following:

$$\text{NFA} : \begin{cases} \texttt{startstate} \\ \texttt{states}[] \end{cases} \quad \text{NFA state} : \begin{cases} \texttt{accepting} \\ \texttt{active} \\ \texttt{transitions}[] \end{cases} \quad \text{NFA transition} : \begin{cases} \texttt{destination} \\ \texttt{symbol} \end{cases}$$

$$\text{DFA} : \begin{cases} \texttt{startstate} \\ \texttt{activestate} \\ \texttt{alphabet} \\ \texttt{states}[] \end{cases} \quad \text{DFA state} : \begin{cases} \texttt{accepting} \\ \texttt{transitions}[] \end{cases}$$

(DFA transitions are just the states transitioned to in order of the alphabet)

I implemented all of the necessary subroutines for managing DFAs and NFAs in an object-oriented way, which I deemed not interesting enough to include in this report. I handled the "character" $\varepsilon$ by replacing defining it as the non-printable ascii characer '
001'.

## $\varepsilon$-NFA Construction

First, I implemented constructors for the 6 cases of the inductive definition for the regex-to-$\varepsilon$-NFA construction. This was fairly straightforwards, and not worth explaining here as it was described exactly in class how to do the constructions. The more interesting part of the $\varepsilon$-NFA construction was parsing the regular expression in the first place. I decided to implement my own parser, and generate the $\varepsilon$-NFA structurally as I went.

First, I checked to make sure any parenthesis were properly balanced, as this would cause my parser to fail and probably seg-fault. Then, I iterated over the regular expression four times, focusing on a different task each time:

1. Create singleton NFAs to accept a single letter from the alphabet, and handle anything in parenthesis recursively
2. Evauluate `[*+?]` from left to right, binding it to whatever came before
3. Concatenate everything together from left to right, except for `|`
4. Evaluate the remaining `|`'s, treating missing operands on either side as $\varepsilon$

## $\varepsilon$-Removal

I broke this down into two steps. First, I added $\varepsilon$-edges to the $\varepsilon$-NFA to create the $\varepsilon$-closure (guaranteeing that if $B \in \delta(A, \varepsilon)$ and $C \in \delta(B, \varepsilon)$, it must also be the case that $C \in \delta(A, \varepsilon)$). I also propagated accepting states backwards (guaranteeing that if some $B \in \delta(A, \varepsilon)$ is an accepting state, $A$ is also an accepting state).

After computing the closure, I made the $\varepsilon$-edges redundant by copying all non-$\varepsilon$ edges (say edges on $a \in \Sigma$) from $\delta(A, \varepsilon)$ to $\delta(A, a)$. Then, I removed all of the $\varepsilon$-edges from the NFA, having already guaranteed that the result would be equivalent.

## Conversion from NFA to DFA

I will refer to the NFA as $N$ and the DFA as $D$. I performed the lazy construction for creating $N$ that was discussed in class. I started by copying the first state $n_0$ into $D$ (not the transitions). Then, I iterated over the constantly resizing buffer of states $d_n$ in $D$ until I had covered all of them. For each state, I checked for each symbol $a \in \Sigma$ what the union of states in $N$ was that was reached by the substates (from $n$) in the composite state in $d_n$ being handled. If this union did not exist yet, I created a new composite state in $D$. Either way, I then had a state to set as the destination on that particular symbol. The garbage state $\varnothing$ was just handled a special case when there were no outgoing edges.

## DFA Minimization

I performed the table filling algorithm here, which I will not discuss in detail. I constructed an array of size $(|D| - 1)^2$, and treated it as a lower-left-triangular table. After computing the table, I iterated over it once, creating a new state each time an entire row (reprenting a state) was distinbuishable from all previous states. The details of the implementation in C are not so important here, suffice it to say that I made a mapping from states in $D$ to the new states in $D/_{\equiv}$ and copied the transitions from $D$.

## DFA Simulation

When I created the NFA and DFA "objects" originally, I included functions for resetting and running them on characters/strings. I read input from `stdin` line by line, first getting the alphabet and regular expression, then iterating over all possible substrings of each line to search for a match. There is not much to say here.

## Findings

The inductive construction of $\varepsilon$ from regular expression is very inefficient with regards to the final $\varepsilon$-NFA size, and had a large number states that served no purpose other than to propagate $\varepsilon$-edges. The NFAs that I printed with $\varepsilon$-closure and $\varepsilon$-removal had so many transitions everywhere that they were unreadable. However, the DFA construction removed the large majority of unnecessary edges by combining redundant composite states, and the minimization (of course) combined any remaining equivalent states.

I did not notice any difference in performance based on the size of the two inputs, but I wrote it in C (a language with very little overhead) and did not test on huge inputs. The vast majority of running time was spent compiling LaTeX code in a separate process, for displaying the automata. However, because I did not have access to a standard library containing sets, the running time on huge regular expressions and alphabets is expected to take a big performance hit. With a hashset, many of he $\mathcal{O}(|Q|)$ and $\mathcal{O}(|\Sigma|)$ loops could be reduced to $\mathcal{O}(1)$.

Simulating the $\varepsilon$-NFA instead of converting to a minimal DFA would give the same result, but with a much larger runtime. In order to read a symbol into a DFA, I had to perform two array accesses and update the index of the current active state, altogether a constant time operation $\mathcal{O}(1)$. However, for the much-larger-than-necessary NFAs, the worst case is that every state is active, has a transition for every symbol in $\Sigma$ to every other state, and that you have to read every transition to update the entire NFA. The running

time of reading a single symbol from such an NFA is then $\mathcal{O}(|N|^2 * |\Sigma|)$. This absolute worst-case scenario will of course not happen with the $\varepsilon$-construction, but the number of states will be a constant factor larger than the length of the regular expression, and the asymptotic bound still holds. Clearly, DFAs win despite the overhead of creating them.