Kompilatorteknikk TDT4205 - Assignment 1

Ola Kristoffer Hoff

27. January 2022

1 Regular languages, NFAs and DFAs

Let the formal language $\mathcal{L}$ be all strings over the alphabet {a, b}, with no more than two 'a's in a row. Examples include abba, baab, baabbba, and the empty string ϵ.

1.1 Show that $\mathcal{L}$ is a regular langauge, by writing a regular expression for it. You only need operators described in slideset 03: |, * and grouping with (). You may also use a? as a shorthand for (a|).

Here is the expression I made: (b*a?a?b)*(a|aa)?.

The thought behind it is that the left zero-closure part takes care of any string containing at least one b and any number of a's, or the empty string. Then the last part takes care of the not covered edge case of a single or double a, also can be nothing to get the empty string or not end in an a.

1.2 Convert the regex from 1.1 into a non-deterministic finite automata (NFA) using the McNaughton–Yamada–Thompson algorithm. Remember to number the states, indicate the starting state, and mark states as either accepting or non-accepting.

When performing the algorithm explicitly, you often get a lot of trivial paths consisting of only ϵ-edges. You are allowed to use some human intuition to avoid introducing obviously unnecessary states and ϵ-edges.

Since the logic that follows in each step is quite simple, I did not make a graph for each "iteration" of the construction. However, below you can see the complete NFA from the regex.

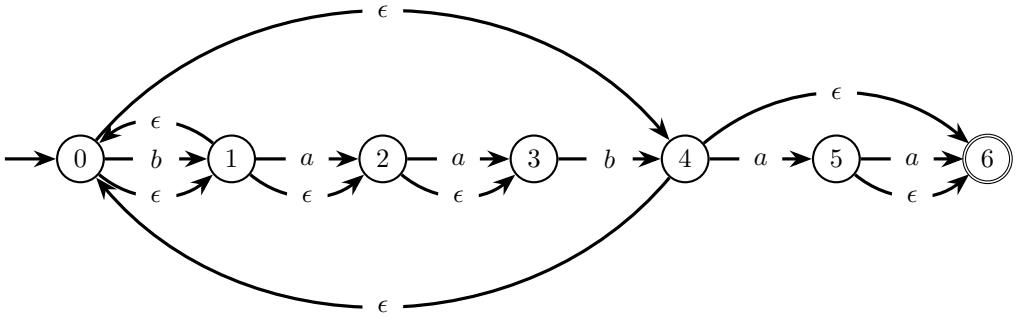

Figure 1: NFA from regex: (b*a?a?b)*(a|aa)?.

1.3 Convert the NFA from 1.2 into a deterministic finite automata (DFA), using the subset construction method described in slideset 04. The Recitation Lecture shows a more complete example of NFA-to-DFA conversion, where the NFA states can't be partitioned into disjoint subsets.

Remember that a DFA may not contain ϵ-edges, and that every state in a DFA must have exactly one out-edge per symbol in the alphabet. If a symbol doesn't lead to any states in the NFA, you must create a "dead end" state in the DFA, and direct any lost cause inputs there.

Once you have the DFA, give each DFA state a number, independent of the NFA state numbering. Again, remember to indicate the starting state, and which states are accepting.

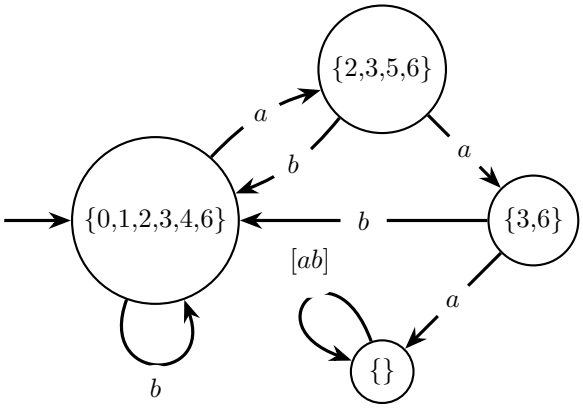

Figure 2: NFA subsets for converting NFA to DFA.

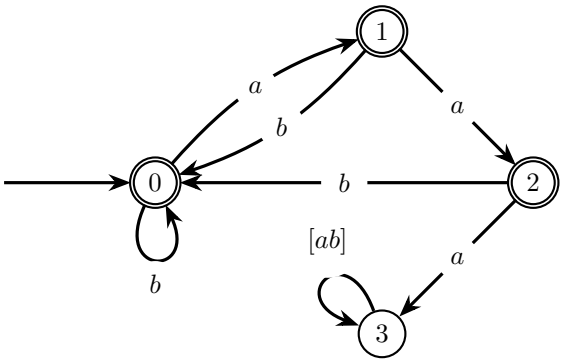

Figure 3: DFA constructed from NFA. Happens to be the minimised DFA too.

1.4 Minimizing a DFA means creating a new DFA with the minimum number of states that still matches the exact same language. Slideset 04 shows Moore's algorithm, which is our recommended way of minimizing DFAs. If the DFA is already minimal, the algorithm will prove it by not producing a smaller DFA.

Minimize the DFA you created in 1.3.

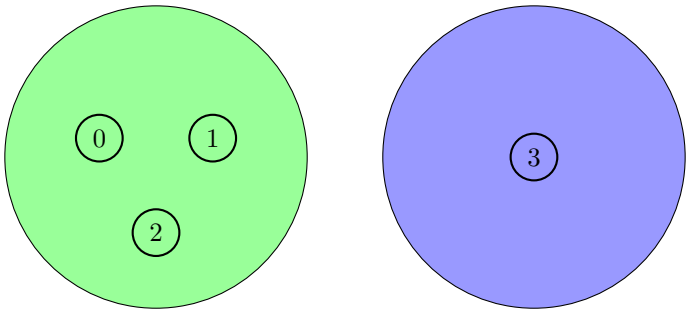

Figure 4: States grouped in accepting (left/green) and non-accepting (right/blue).

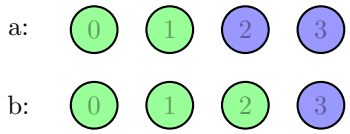

Figure 5: State transitions to group on input "a" or "b".

As we can see state three is already a single state, which means we can exclude it from further review. We will split the green group ({0,1,2}) into two new groups.

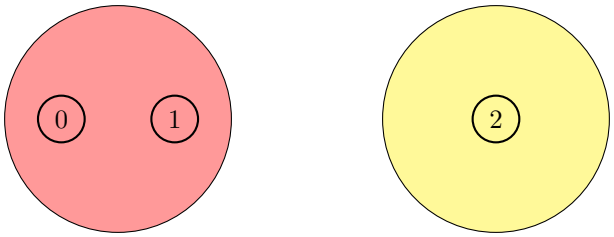

Figure 6: States grouped in staying in green (left/red) and leaving green (right/yellow).

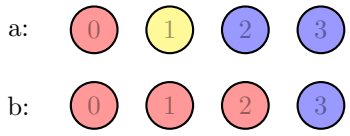

Figure 7: State transitions to group on input "a" or "b".

As we can tell from the last diagram, states 0 and 1 will now be split into separate groups as well. Then all states are in their own group, meaning there is no minimisation to be done. Somehow I have managed to create the minimum DFA directly from my original regex expression (through a NFA of course).

1.5 Finally, consider the "opposite" langauge, $\mathcal{L}^*$, consisting of every string over the alphabet {a, b} that has to have more than two consecutive 'a's occur somewhere in the string.

How would you go about creating a DFA matching $\mathcal{L}^*$, given that we already have a DFA matching $\mathcal{L}$?.

Produce a regular expression that matches $\mathcal{L}^*$. (Try using the website from 1.1 and make every test red).

Between DFAs and regexes, which was the easiest to invert to the "opposite" language?

I understood the language as any sequence of "a"'s and "b"'s that has at least one consecutive sequence of three or more "a"'s in it. With this understanding I created the regex: (b*a*)*aaa+(b*a*)*. Simply put it does what I described above, any number of "a" and/or "b" then at least three "a"'s then again any number of "a" and/or "b".

Creating the opposite DFA we could "toggle" the states acceptance. This would turn all accepting states to non-accepting and vice-versa. We can see this DFA below. The DFA can be described as it being non-accepting until it has a path of three "a"'s then it is absorbed by the accepting state 3, where any sequence of $[ab]$ is valid.

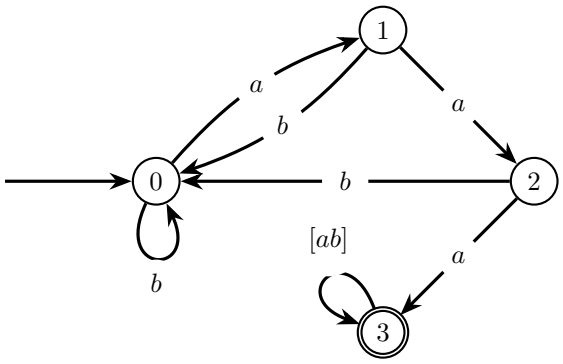

Figure 8: DFA inverted to support opposite language.

The DFA was much easier if I am correct in my method of toggling the states. The regex was fairly easy to create since it had an inclusive constraint and not an exclusive one from earlier, but in the general sense I think DFA's would always be easier to invert.

2 DFA for small language

In this exercise, we will create a scanner to recognise a minimalistic language for line drawings. It consists of these three types of statements, which are all terminated by a newline character ($'\backslash n'$):

dx=(integer)
dy=(integer)
go

The character sequences **dx=**, **dy=**, and **go** are fixed, integers consist of a **non-empty** sequence of digits with an optional '-' prefix for negative values.

2.1 Write a regular expression matching exactly one statement, including the newline character at the end.

The regex I created: "((d(x|y)=-?(0|([1-9][0-9]*)))|(go))\n".

The thought behind it is quite simple we either have a "dx" or "dy" statment or a "go". It must end in a "\n". Then the minus is optional and an integer is either zero, or starts with one through nine then any number of zero through nine after that.

2.2 Create a DFA matching this regular langauge. All complete valid statements should lead to a single accepting state, while anything invalid should lead to a single error state.

Draw this DFA, and number the states. You don't have to prove the DFA minimal, but know that it's possible to do it with 10 states. Since our alphabet is now 256 characters big, you can omit drawing the edges to the "dead end" invalid state, but know that they are there.

I will find the DFA in the same way we did in task 1. So first I convert the regex to a NFA.

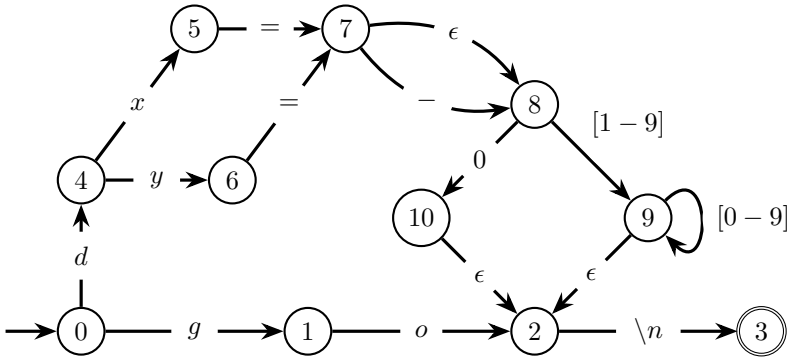

Figure 9: NFA from regex: "((d(x|y)=-?(0|([1-9][0-9]*)))|(go))\n".

Then I convert the NFA into a DFA.

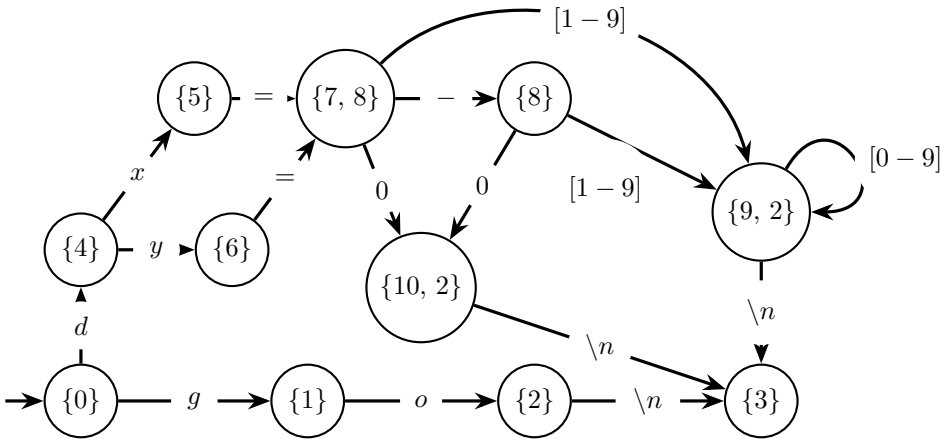

Figure 10: DFA groupings from NFA.

We note that state group 5 and 6 both transition to group $\{7, 8\}$ on =, this can then be shortened. Same with state group $\{10, 2\}$ and 2 to 3, on "\".

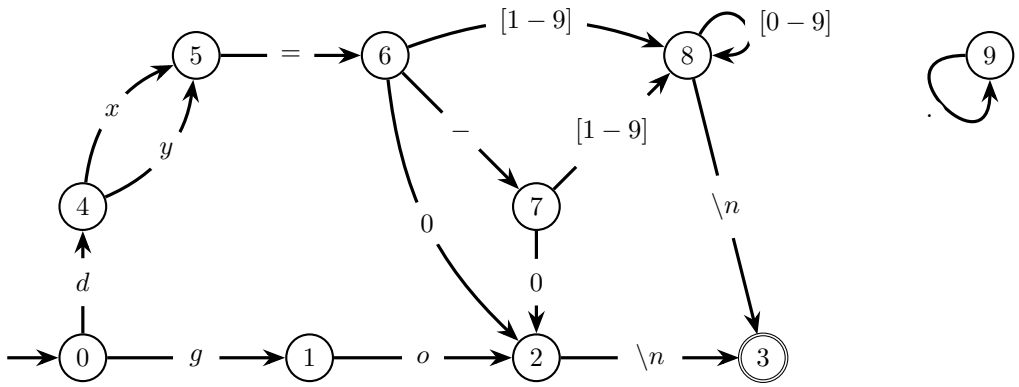

Figure 11: DFA from DFA groupings, with observed optimisations.

I added the "dead state" (state 9), but have omitted the edges going to it from all other states.

As we can see, the final DFA has ten states, counting the "dead state". If my logic is correct and the optimal DFA had ten states, then this is the optimal DFA.

2.3 Implementation

The error was at line 5890, and had the incorrect statement: "dx==". Below you can see the PDF generated from running the script.

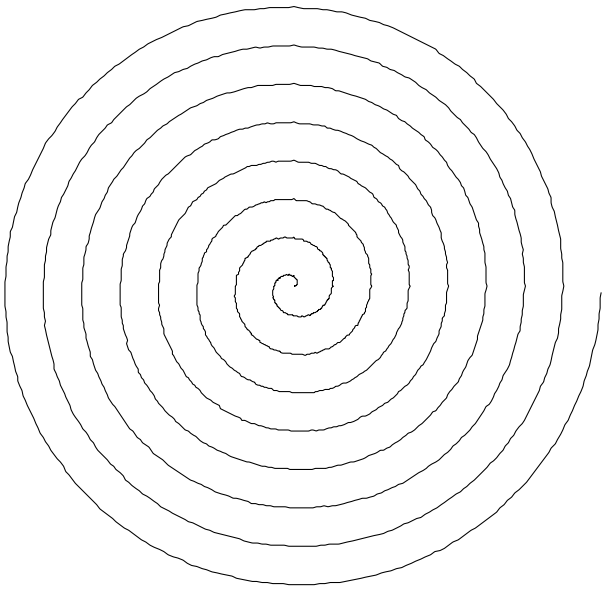

Figure 12: The generated PDF from the script.