TDT4205 Problem Set 1

Jan 20, 2023

Answers are to be submitted via Blackboard. Please submit your answer as an archive `username.tar.gz` containing a PDF file with answers to theoretical questions, and a code directory like the provided one, containing all files required to build your solution.

Disclaimer: The problems in this problem set are designed to be more difficult than anything you will be asked on the exam. Use any resources you can, the recitation lecture slides, asking each other, and Piazza. If you got some help from other students, remeber to make it clear where you got help, and also which tasks you did by yourself. Start early, and when the deadline approaches, **submit what you have got**, no matter how finished.

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|)`.

This regex is a bit tricky, so we made a website with tests that gives feedback as you write: `https://folk.ntnu.no/haavakro/tdt4205/regex.html`.

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 explicilty, 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.

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.

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.

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?

2 DFA for a small language

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

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.

2.3 Implementation

In the archive `ps1_skeleton.tar.gz`, you will find an implementation of this language which tracks a pair of (x, y) coordinates that are initialized to the center of a page, and two values (dx, dy). The assignment statements in our language set the (dx, dy) values respectively, and the `go` statement alters the (x, y) coordinates by (dx, dy), drawing a line from the previous position to the new one.

The main function already implements the table-based DFA simulation algorithm, but its transition table is empty. Implement the `initialize_transition_table` function found in `scanner.c` to make all finished valid statements reach the `ACCEPT` state, and all invalid statements reach the `ERROR` state. You can dimension the table to match the size of your automaton by modifying the `N_STATES` macro. You should also change the macros `ACCEPT`, `ERROR`, and `START_STATE` to match your DFA.

Once the accept state is reached, the driving code handles the statement, and resets the DFA to `START_STATE` for you, ready for the next statement. If the error state is reached, the driving code prints out the line where it happened.

When extended with a correct automaton in the table, this program will emit drawing instructions in postscript, which can be converted to a PDF document. The archive includes a sample file of commands that draw a spiral (`spiral.txt`), which can be used to verify your solution like so:

```
cat spiral.txt | ./scanner | ps2pdf - spiral.pdf
```

However! I have gone ahead and inserted an invalid statement in this ~ 7000 line file. A correctly implemented scanner will report the line number, which you can use to fix `spiral.txt` manually. You don't need to hand in the fixed file, or the `spiral.pdf` you produce.