# Project 1: Minimization of DFAs
# Theory of Computation (CSCI 3500), Fall 2018
# Total Points: 75

Due: Thursday, Oct. 10 by 11:59pm

**This project is to be turned in via git only.**

At this point in the semester we have covered, in depth, deterministic finite automata (DFA), as well as non-deterministic finite automata (NFA). In addition, we have proven that NFAs are equivalent to DFAs, but we noticed that the conversion process creates a number of useless states in the resulting DFA. In this project you are to implement the algorithm for minimizing DFAs. Your implementation will need to be able to parse in a description of a DFA from a file, and then minimize it into an equivalent DFA, and finally print out that corresponding DFA in the same format as the input description. This project consists of several parts.

## Preliminaries

You are allowed to use any programming language you like, and any libraries you like. However, you are not allowed to use any libraries that implement DFAs or the minimization process. You must implement those yourself.

Your project will be graded using testing, and not by examining your code. **Thus, while I will try to give partial credit when possible, if I cannot conduct the testing process, then a zero will be given.** Therefore, it is of the utmost importance that your project compiles and runs when you turn it in to get a passing grade.

You will need to turn in a zip file containing the following:

- All your source code, and
- instructions for building and running your project – this must include details of all the tools and software that I need to install to build your project.

## Part 1: Implementing DFAs

To make testing easier I recommend first implementing parsing DFAs. When choosing an internal representation make sure you keep in mind that you will need to use this representation to parse in a DFA and then minimize it into a different DFA whose states will not be the same as the states in the input. For example, you will need to be able to handle list of states. Then simply follow the formal definition, and all should be well.

One last hint. I recommend modeling the transition function as a list or an array of the type $(Q \times \Sigma) \times Q$.

## Part 2: The DFA Parser

In this section we will layout the description of DFAs that you will need to write a parser for. The parsers job is to turn the description language into a full blown DFA.

Formally, DFAs are 5-tuples, so the description language will simply describe this tuple. We now define the DFA description language. Each DFA will be described in a file consisting of the following lines:

$$\begin{array}{l}
(\mathsf{states}, (q_0, \ldots, q_i)) \\
(\mathsf{alpha}, (a_0, \ldots, a_j)) \\
(\mathsf{trans\text{-}func}, ((q_0, a_0, q'_0), \ldots, (q_n, a_n, q'_n))) \\
(\mathsf{start}, q) \\
(\mathsf{final}, (q'''_0, \ldots, q'''_j))
\end{array}$$

Finally, states can be any strings at all, but alphabet symbols must be only a single character, and your parser should issue an error if something else is given.

The file describing the DFA must be in the above order. Meaning states have to come first, then the alphabet, and so on.

Here is an example DFA described using the above language:

```
(states, (0,1,2,3,4,5))
(alpha, (1,2))
(trans-func, ((0,1,3),(0,2,4),(1,2,3),(2,1,5),(3,1,2),(4,2,1),(5,1,4)))
(start, 0)
(final, (1))
```

Here is another example:

```
(states, (0,1))
(alpha, (a))
(trans-func, ((0,a,1),(1,a,1)))
(start, 0)
(final, (1))
```

Keep in mind that after you parse in a DFA and build its data structure you will need to verify that it actually is a well-defined DFA. That is, you need to check for things like the given start state is among the set of sets, the set of final states is a subset of the set of states, and each state and alphabet symbol used in the description of the transition function are actually states and alphabet characters of the DFA. If any of these fail, so should your program.

## Part3: Pretty DFAs

In this part you have to design and implement a pretty printer for DFAs. This should be a function that takes a DFA as input, and then outputs that DFA in the DFA description language defined as follows:

$$\begin{array}{l}
(\mathsf{states}, (q_0, \ldots, q_i)) \\
(\mathsf{alpha}, (a_0, \ldots, a_j)) \\
(\mathsf{trans\text{-}func}, ((q'_0, a'_0, q''_0), \ldots, (q'''_n, a'_m, q''''_n))) \\
(\mathsf{start}, q) \\
(\mathsf{final}, (q'''''_0, \ldots, q'''''_j))
\end{array}$$

Thus, a pretty printer takes your internal representation and converts it into the language above, and then outputs it to the screen.

When you convert the DFA to a minimized DFA the states of that DFA will be lists or arrays of states. When you output this list you must output it in the format: $[q_0, \cdots, q_i]$. If $i = 0$, then output [], and if $i = 1$, then output the state without brackets. This will allow for the user to easily see the usual representation.

For example, if your program is given the following DFA:

```
(states, (1,2,3,4,5,6,7))
(alpha, (a, b))
(trans-func ((1, a, 2), (1, b, 4),
             (2, a, 3), (2, b, 2),
             (3, a, 3), (3, b, 3),
             (4, a, 7), (4, b, 5),
             (5, a, 6), (5, b, 5),
             (6, a, 6), (6, b, 6),
             (7, a, 7), (7, b, 7)))
 (start, 1)
 (final, (3,6))
```

Then it should output something equivalent to the following:

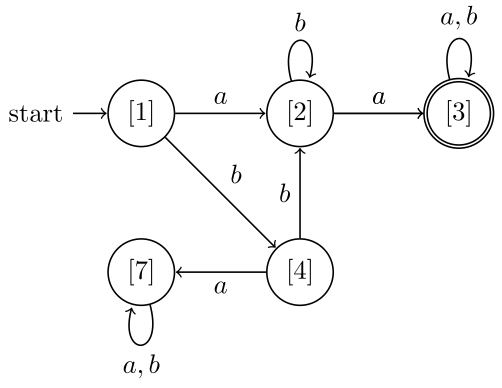
```
(states, (1,[2,5],[3,6],4,7))
(alpha, (a, b))
(trans-func ((1, a, [2,5]), (1, b, 4),
             ([2,5], a, [3,6]), ([2,5], b, [2,5]),
             ([3,6], a, [3,6]), [3,6], b, [3,6]),
             (4, a, 7), (4, b, [2,5]),
             (7, a, 7), (7, b, 7)))
 (start, 1)
 (final, ([3,6]))
```

## Part 4: Minimizing DFAs

At this point we turn to the minimization algorithm. There is a beautiful theory behind why this all works, but for this project we are only really concerned with implementing the algorithm.

A minimal DFA is one in which every state is useful to the acceptance of its language. Thus, a minimization algorithm must decide which states are useful and which are not. It will do this by determining which states are equivalent and which are distinct.

We will use the following DFA (which is actually the diagrammatic version of the example in the last section) as our running example through this section:

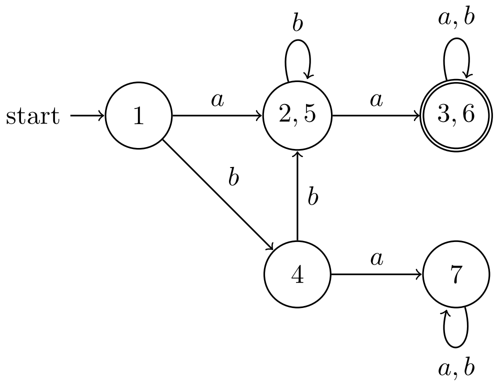


It may not be obvious, but this DFA actually is not minimal. DFAs always have a particular start state, but we could just as well start from anywhere. Suppose we call the above DFA $M$, then our current start state in M is 1, and $L(M)$ can be described by the regular expression $(ab^*a(a \cup b)^*) \cup (bbb^*a(a \cup b)^*)$. If we change the start state to, say, state 4, then $L(M)$ can be described by $(bb^*a(a \cup b)^*)$. If we start at state 7, then $L(M) = \emptyset$.

Suppose we decided to start in state 3, then $L(M)$ can be described by $(a \cup b)^*$, but the same can be said for state 6. Similarly, if we start in state 2, then $L(B)$ can be described by $b^*a(a \cup b)^*$, but the same can be said for state 5. Thus, when we get to state 3 or 6, while running $M$, it will accept or reject the exact same strings. The same can be said for states 2 and 5.

Denote by $L(M)_q$ the language accepted by $M$ when the start state of $M$ is $q$. If $L(M)_q = L(M)_r$ for two states $q$ and $r$ of $M$. Then we can actually merge the two states $q$ and $r$ into a single state, because once we reach either $q$ or $r$ the machine will accept the exact same language.

The new, and minimal DFA, is as follows:



Call two states, $q$ and $r$, of a DFA, $M$, equivalent when $L(M)_q = L(M)_r$. The minimization algorithm then has to determine which states are equivalent and which are distinct. Then we construct a new DFA by merging equivalent states. Thus, the minimization algorithm can be broken down into two phases.

## Phase 1: State Distinction

In this phase we build a table, called distinct, with an entry for each pair of states. Initially, Oevery cell in the table is set to 0. We will denote a cell in the table by $\mathsf{distinct}(p, q)$. Suppose $M = (Q, \Sigma, \delta, q_0, F)$ is a DFA. Then the following algorithm describes how to fill each cell of the table:

```
Step 1. For each pair of states $(p, q) \in Q \times Q$
    If $p \in F$ and $q \notin F$ (or vice versa), then set $\mathsf{distinct}(p, q) = 1$.
Step 2. Loop until there is no change in the table contents:
    For each pair of states $(p, q) \in Q \times Q$:
        For each alphabet symbol $a \in \Sigma$:
            If $\mathsf{distinct}(p, q) = 0$ and $\mathsf{distinct}(\delta(p, a), \delta(q, a)) = 1$, then set $\mathsf{distinct}(p, q) = 1$.
```

The algorithm works by first running step 1, and then after it completes, running step 2. When the algorithm is finished, then distinct tells us which states in $M$ are distinct. We say $p$ and $q$ are distinct iff $\mathsf{distinct}(p, q) = 1$.

The following table is distinct for our example DFA after running step 1 of the above algorithm:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | | | | | | | |
| 2 | 0 | | | | | | |
| 3 | 1 | 1 | | | | | |
| 4 | 0 | 0 | 1 | | | | |
| 5 | 0 | 0 | 1 | 1 | | | |
| 6 | 1 | 1 | 0 | 1 | 1 | | |
| 7 | 0 | 0 | 1 | 1 | 1 | 1 | |
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

The bottom of the table is labeled with the first project, and the vertical is the second projection. Thus, we can see that $\mathsf{distinct}(3,4) = 1$. Notice that by definition $\mathsf{distinct}(p,q) = \mathsf{distinct}(q,p)$, and so we do not need to track both cells. Also, notice that $\mathsf{distinct}(p,p) = 0$ for all states in the DFA, and so we do not need to track the diagonal.

The following table is after the first iteration of the second step of the algorithm above:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | | | | | | | |
| 2 | 1 | | | | | | |
| 3 | 1 | 1 | | | | | |
| 4 | 1 | 1 | 1 | | | | |
| 5 | 1 | 0 | 1 | 1 | | | |
| 6 | 1 | 1 | 0 | 1 | 1 | | |
| 7 | 1 | 1 | 1 | 1 | 1 | 1 | |
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

On the second iteration there is no change to the table, and the algorithm terminates. The table now tells us that the only equivalent states are $(2,5)$ and $(3,6)$. Thus, these will be merged in the second phase of the algorithm.

## Phase 2: Constructing the Minimal DFA

Once we have computed distinct we can define an equivalence relation:

$$p \equiv q \text{ iff } \mathsf{distinct}(p,q) = 0$$

Thus, the previous example table tells us that $2 \equiv 5$ and $3 \equiv 6$.

Given any equivalence relation like $\equiv$ we can define what are called equivalence classes. The equivalence class for the state $p$, denoted $[p]$, is the following set:

$$[p] = \{q \in Q \mid p \equiv q\}$$

Thus, $[p]$ is the set of all states that are equivalent to $p$. Thus, $[2] = [5] = \{2,5\}$ and $[3] = [6] = \{3,6\}$ for the equivalence relation induced by our example. Note that all of the other equivalence classes are singleton sets.

We now can use the equivalence relation obtained from distinct to construct a minimal DFA by constructing a new DFA called the quotient DFA. Suppose $M = (Q, \Sigma, \delta, q_0, F)$ is a DFA and we have already computed the table distinct for $M$, and hence, we have an equivalence relation, $\equiv \subseteq Q \times Q$ telling us which states are equivalent in $M$. Then we can construct the quotient DFA $M_\equiv = (Q_\equiv, \Sigma_\equiv, \delta_\equiv, q_\equiv, F_\equiv)$ as follows:

$$\begin{array}{lcl}
Q_\equiv & = & \{[p] \mid [p] \text{ is an equivalence class with respect to } \equiv\} \\
\Sigma_\equiv & = & \Sigma \\
\delta_\equiv([q], a) & = & [\delta(q, a)] \\
q_\equiv & = & [q_0] \\
F_\equiv & = & \{[p] \mid [p] \cap F \neq \emptyset\}
\end{array}$$

This construction automatically places mergable states into the same equivalence class, and hence, the new DFA will remove those, and replace them with a single state.

We can now compute the final minimal DFA for our running example. We will denote each equivalence class by its set of elements. The final DFA is as follows:

$$\begin{aligned}
Q_{\equiv} &= \{\{1\},\{2,5\},\{3,6\},\{4\},\{7\}\} \\
\Sigma_{\equiv} &= \{a,b\} \\
q_{\equiv} &= \{1\} \\
F_{\equiv} &= \{\{3,6\}\}
\end{aligned}$$

The transition function is depicted in graphical form at the end of page 4.

## Part 5: Putting it all Together

You are now ready to put all of your hard work together into one glorious function called main. The main loop should prompt the user for the path to a file containing the description of a DFA, and then parse that DFA, minimize it, and then use the pretty printer to print out the description of the equivalent DFA. That's it!

You may prompt the user in anyway you wish. You do not have to use a GUI if you do not want to, but you can if you do.