Implementation and visualization of the procedure NFA into DFA

Jordano Baer
Computer Science and Engineering
University of South Florida
Tampa, United States
jordanobaer@mail.usf.edu

John Cameron
Computer Science and Engineering
University of South Florida
Tampa, United States
jcameron2@mail.usf.edu

***Abstract*—This is the report for the Automata Theory/Formal Languages term project. The goal was to develop an algorithm that converts a Non-deterministic Finite Automata (NFA) to a Deterministic Finite Automata (DFA). This report describes the design process, implementation, and visualization of the algorithm developed.**

***Index Terms*—NFA, DFA, Visualization, Automata, Algorithm**

I. Introduction

The objective of this project was to develop a visualized procedure that converts an NFA into its equivalent DFA. The conversion process will never fail because if a language can be recognized by an NFA, there will always be an equivalent DFA. Because our procedure should be able to handle a NFA containing λ-productions (i.e. null-productions or ϵ-productions), the final DFA should have the ability to contain multiple states within a single state. In addition, a trap state will also be included in the resulting DFA if any state in the given NFA has an undefined transition.

II. Background

The first step when developing this algorithm, or in other words, the NFA-to-DFA visualization procedure, the differences between an NFA and a DFA had to be known. Unlike a DFA, an NFA is capable of being in multiple states at once, having undefined or λ transitions, and multiple transitions when given only a single input.

A. Differences between NFA and DFA

An NFA is similar to a DFA, except that there are three main differences that are allowed in a NFA but not allowed in a DFA:

1) NFA permits λ-transitions
2) NFA is able to have undeclared or undefined transitions
3) NFA may contain transitions to different states after receiving a single input (can be in multiple states at once)

Two automatas are equivalent if and only if they both accept the same language. As a result, it can be concluded that if both automatas accept the same language, then they are equal. It is also important to know that the NFA-to-DFA procedure will always terminate because the number of states in a NFA

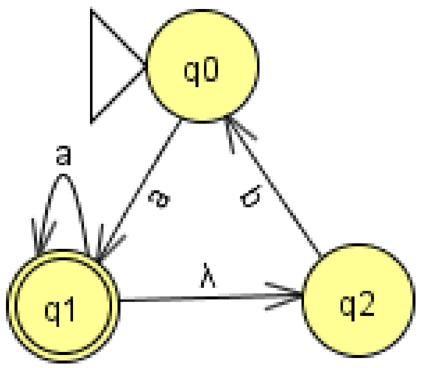


Fig. 1. Original NFA

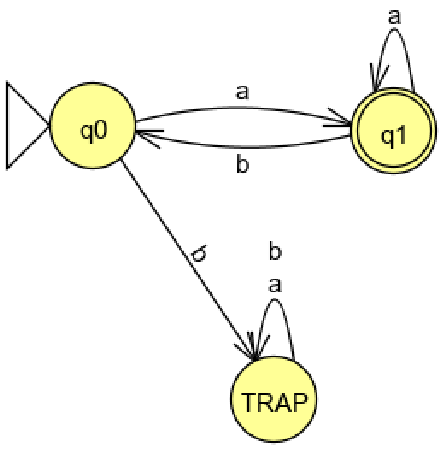


Fig. 2. Equivalent DFA from NFA in Fig. 1

and its equivalent DFA is always finite. The final DFA may have 2^n states, where n is the number of states of the original NFA.

B. Important factors during the conversion process

During the process, there are important factors that the algorithm should track during the conversion process:

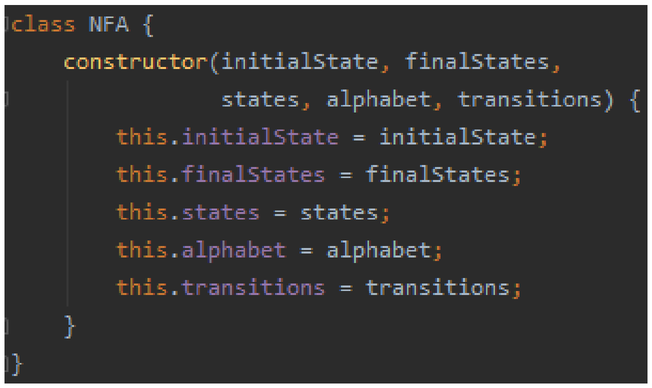
1) Is a trap state required?
2) What is the initial state?
3) What are the final states?
4) Does the original NFA contain λ-transitions?
5) How will the algorithm know when it is done?
6) Is there a λ-transition from initial state to final state?

III. Methodology

The implementation of the NFA-to-DFA algorithm was developed using Javascript. The basic concept was that users would be able to input an NFA in a web interface (HTML) and then the site, using Javascript, would convert the NFA provided to an equivalent DFA. The algorithm itself does not rely on any third-party libraries; however, Viz.js was used for the purpose visualizing finite automatas.

A. The Automata Format

An NFA and DFA both use the same data structure internally because of the similarities between a DFA and NFA. The most prominent characteristic is that a DFA is also an NFA, but an NFA is not a DFA. This data structure used to represent an NFA/DFA is simply a class named NFA. As shown in Fig. 3, the data structure follows the formal definition of an NFA. The formal definition states that an NFA has an initial state, a set of final states, a set of all states in the automata, the alphabet set, and the transitions. The transitions variable uses its own data structure, named Transition, which also follows the formal definition of a transition in an NFA ($\Delta : Q \times \Sigma \rightarrow P(Q)$).

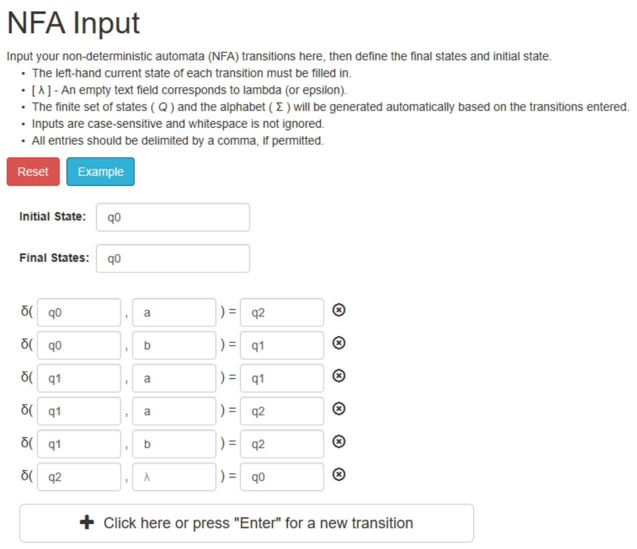


```
class NFA {
    constructor(initialState, finalStates,
                states, alphabet, transitions) {
        this.initialState = initialState;
        this.finalStates = finalStates;
        this.states = states;
        this.alphabet = alphabet;
        this.transitions = transitions;
    }
}
```

Fig. 3. Code Snippet of the NFA Class

B. User Input

The user input was designed in such a way that would be the most intuitive for the end user. To achieve this, the HTML page was made to be clutter-free and to only prompt the user for necessary input values. For example, the algorithm would automatically generate the alphabet set from the transitions provided by the user instead of prompting the user to define the alphabet set. By looking at user input form, it can be pointed out that the most prominent aspect are the textboxes acting as placeholders for user inputs. Notice that the textboxes are aligned and positioned in such a way so that the user would be able to clearly understand how the algorithm would process the input provided. The overall goal was to make sure that the user would be able to easily understand how to enter data, read the results, and make modifications.



NFA Input

Input your non-deterministic automata (NFA) transitions here, then define the final states and initial state.

- The left-hand current state of each transition must be filled in.
- [λ] - An empty text field corresponds to lambda (or epsilon).
- The finite set of states (Q) and the alphabet (Σ) will be generated automatically based on the transitions entered.
- Inputs are case-sensitive and whitespace is not ignored.
- All entries should be delimited by a comma, if permitted.

Initial State: q0

Final States: q0

δ(	q0	, a	) =	q2	⊗
δ(	q0	, b	) =	q1	⊗
δ(	q1	, a	) =	q1	⊗
δ(	q1	, a	) =	q2	⊗
δ(	q1	, b	) =	q2	⊗
δ(	q2	, λ	) =	q0	⊗

Fig. 4. User Input HTML Form

C. Create the new states

After the user enters the information, the NFA is created and displayed to the user so that he or she can confirm its correctness. Then, the first state of the DFA is created by calculating the closure of the initial state in the NFA. Also, the initial state of the DFA is stored in a stack. When the algorithm creates a new state, that new state will be added to the stack for further processing. While the stack is not empty, the algorithm will check all outgoing transitions for the current state and then check to see if a new state is created from these transitions based on its closure. The new transitions for the DFA are created by the union operation with the closure of the transitions from the original NFA for the current state in the DFA. If a new state is created, it is added to the stack. If the new transition is empty, a trap state is created instead. A trap state is simply a state that only contains transitions to itself. Because of this, once the automata enters the trap state in a DFA, there is no way out of that state. The reason a trap state is required in some cases is because a DFA requires a transition for every possible input in the alphabet for every state unlike an NFA.

D. Relevant Data Structures

When creating a new state and its transitions, it is critical to check the original NFA and search for its original states,

Algorithm 1: NFA to DFA conversion algorithm. δ references NFA transitions.

```
Input: An NFA without lambda transitions
Output: An equivalent DFA
1  Function NFAtoDFA($NFA$):
2      $new\_states$ = stack()
3      $new\_states$.push($initial\ state$)
4      while $new\_states \neq \emptyset$ do
5          $current\_state = new\_states$.pop()
6          for $s \in \Sigma$ do
7              if $\delta(current\_state, s) \neq NIL$ then
8                  $new\_state$ = a new state in DFA based
                     on the union of next states in NFA
9                  $\delta(current\_state, s) = new\_state$
10                 $new\_states$.push($new\_state$)
11             else
12                 $\delta(current\_state, s) = trap\ state$
                     $(create\ one\ if\ needed)$
13             end
14         end
15     end
16     return $new\ dfa$
```

Algorithm 2: DFA Reduction Algorithm

```
Input: Any DFA
Output: A reduced DFA equivalent to original DFA
1  Function reduceDFA($dfa$):
2      for $state1 \in Q$ do
3          for $state2 \in Q$ do
4              if $state1 \neq state2$ then
5                  $states\_equal$ = true
6                  for $s \in \Sigma$ do
7                      if $\delta(state1, s) \neq \delta(state2, s)$ then
8                          $states\_equal = false$
9                      end
10                 end
11                 if $states\_equal$ then
12                     $remove = state1$
13                     $replace = state2$
14                     if $state1 \in F$ then
15                         $remove = state2$
16                         $replace = state1$
17                     end
18                     $dfa$.delete($remove$)
19                     for $state \in Q$ do
20                         for $s \in \Sigma$ do
21                             if $remove \in \delta(state, s)$
                                 then
22                                 $\delta(state, s)$.sub($remove$)
23                                 $\delta(state, s)$.add($replace$)
24                             end
25                         end
26                     end
27                 end
28             end
29         end
30     end
31     return $dfa$
```

their transitions, and closure. A stack was used along with a loop to keep track of newly generated states in the DFA. As a result, all states in the DFA would be considered and populated with their proper transitions. The loop terminates when the stack becomes empty because all of the new states would have been created. Generally, it did not matter what order the data structure processed elements in. The only reason a stack was used was because of its simplicity; the algorithm would behave no different if the stack was replaced with a FIFO, FILO, LIFO, or LILO data structure.

E. Reduction

DFA reduction was not necessary for this project. Regardless, a DFA minimization algorithm was implemented as a bonus and as a way to have cleaner results. The minimization works as followed: after the DFA is created, the code calls the reduction function. This function goes through all the states of the DFA and searches for two states which have the same transitions for every possible $s \in \Sigma$ and that are final or non-final, i.e., if one state is final, the other one must also be final for them to be considered equal. If the states are equal, the algorithm then decides which state should be removed. If one state is the initial state, that means that this state should not be removed and, instead, the other state should be removed. After the algorithm decided which state to remove, the code then searches for all the instances of that state in the DFA and replaces it with the other equivalent state.

F. Visualization

When the original NFA is generated and every time a new state for the DFA is created, the program will save its current state and make its display viewable by the user. That way, the user can select each step of the conversion process to see how the algorithm generated the final DFA. In addition, the HTML page will also render the original NFA so that the user can confirm their input.

The Viz.js library uses the dot graph format to serialize and deserialize finite automatas. An example serialization in the dot text format is shown in the first Listing.

```
digraph fsm {
    rankdir=LR;
    size="8,5";
    node [shape = doublecircle]; q1;
    node [shape = point]; INITIAL_STATE
    node [shape = circle];
    INITIAL_STATE -> q0;
    q0 -> q1 [label=a];
    q1 -> q1 [label=a];
    q1 -> q2 [label=(lambda)];
    q2 -> q0 [label=b];
}
```

Listing 1. Graphviz DOT Language Example for Fig. 1. (lambda) is λ.

IV. Results

A. Running the Program

Because this project was implemented using web technologies, users only need a browser capable of executing Javascript ES2016. The only step involved is to open the HTML file in said web browser.

B. Demonstration

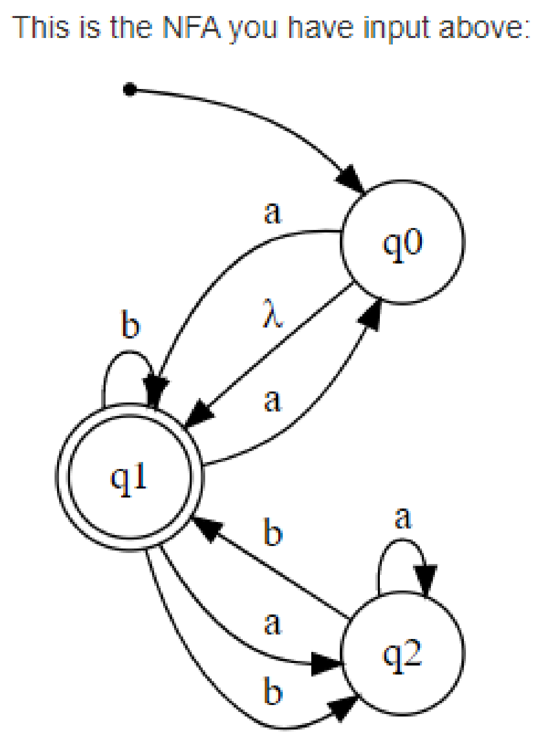


Fig. 5. Original NFA

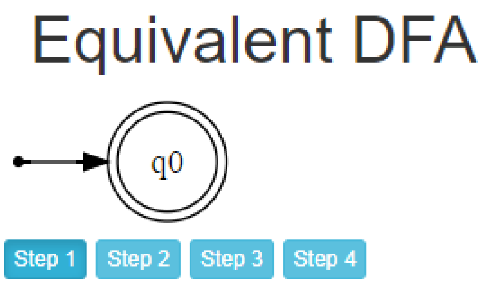


Fig. 6. DFA Step 1

Notice that, step 4 contains the complete DFA and Step 5 reduces the DFA. All three states have the same transitions for ’a’ and for ’b’. Additionally, they are all final states, therefore, the three states are equal. Since they are equal, the algorithm will remove the duplicates and replace them in the minimized

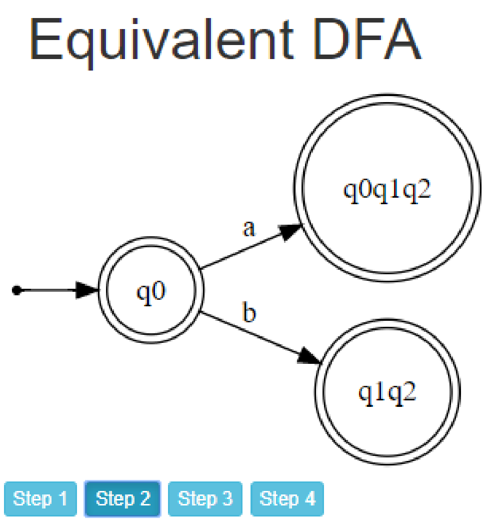


Fig. 7. DFA Step 2

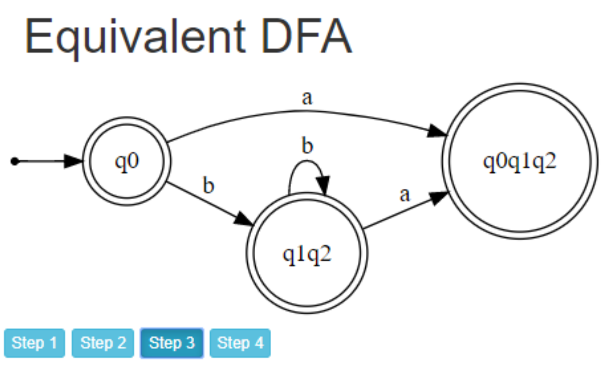


Fig. 8. DFA Step 3

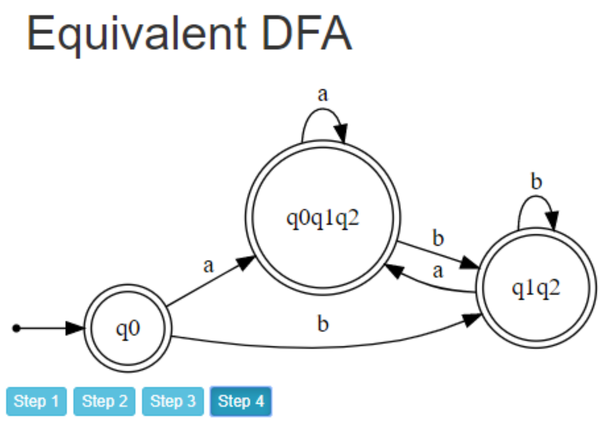


Fig. 9. DFA Step 4

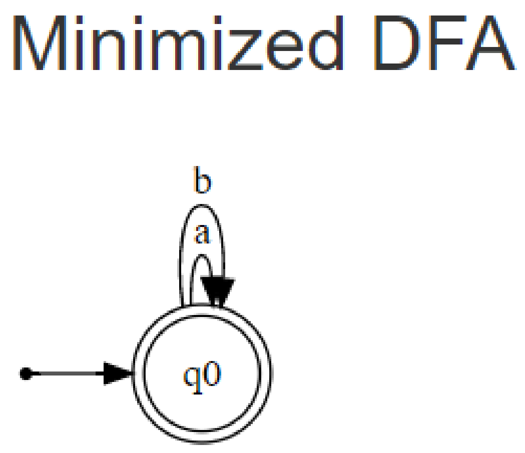


Fig. 10. DFA Step 5 (Final Step, Minimized DFA)

DFA. State q0 is the initial state and, because of that, it should not be deleted. States are considered equal if they have the same transitions for all possible inputs in the alphabet and both must be final or not final.

C. Testing

Multiple NFAs were used to test this project. In order to evaluate the results, many different types of NFAs were converted to DFAs. More than ten NFAs, some with lambda transitions and some without, were all converted to DFAs successfully. Additionally, the converted DFAs were also verified with the procedure provided by JFLAP to convert NFAs to DFAs.
It has been already proved that if a language L is accepted by some NFA, then there will be some equivalent DFA which also accepts L [1]. With this in mind, it can be reasonably concluded that our algorithm is implemented correctly when the equivalent DFA created accepts and rejects the same strings recognized by the original NFA. An example is provided to show how a string can be checked to see if it would be accepted or rejected by either an NFA or DFA. If the string is accepted or rejected by both the NFA and equivalent DFA, then our conclusion of the implementation being correct becomes more confident. The algorithm is known to be correct because it is based off the NFA and DFA equivalence proof.

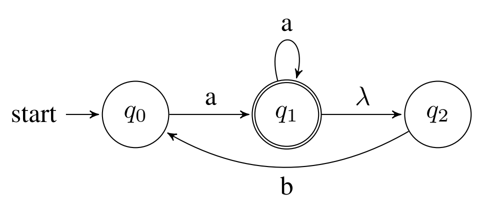


Fig. 11. This is the NFA that will be used to test the correctness of our implementation.

$$\begin{tikzcd}
q0 \arrow[r, "a"] & q1 \arrow[r, "\lambda"] \arrow[dr, "b"'] & q2 \arrow[r, "b"] & q0 \arrow[r, "a"] & q1 \\
 & & \emptyset & &
\end{tikzcd}$$

Fig. 12. Visualization of how input string aba would be processed by the NFA in Fig. 11. After the input has been fully consumed, $q1$ remains as an active state in the NFA. Because $q1 \in F$, this NFA accepts the string aba.

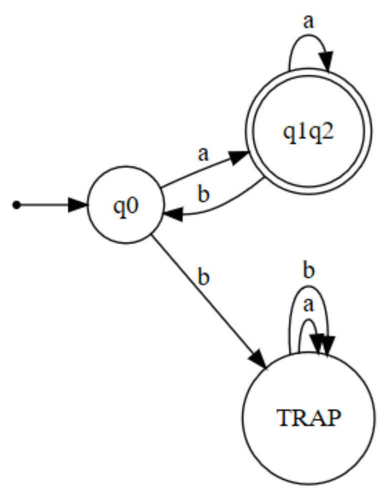


Fig. 13. Equivalent DFA generated by our implementation when given the NFA shown in Fig. 11. Notice that a trap state had been created to handle the case for when state $q0$ receives an input b.

$$q0 \xrightarrow{a} q1 \xrightarrow{b} q0 \xrightarrow{a} q1q2$$

Fig. 14. Visualization of how input string aba would be processed by the DFA in Fig. 13. When the input been fully consumed, $q1q2$ is the active state in the DFA. Because $q1q2$ is a final state ($q1q2 \in F$), this DFA accepts the string aba.

Because the NFA in Fig. 11 and its equivalent DFA in Fig. 13 both accept the string aba, it can be reasonably be assumed that the two automatas accept the same language. Please keep in mind that performing a brute-force test of all possible strings in the alphabet is time consuming. Because of this, it difficult to truly prove whether two automatas are equivalent by only testing and using examples.

D. Surprising Discoveries

1) Web-based NFA to DFA conversion implementations: During development, it was surprising to see that there was a shortage of existing NFA to DFA implementations readily available for the general public. Because of this, there was nothing to reference for what the graphical interface could look like. Overall, the final interface satisfies the requirement of allowing the user to easily an NFA and view its equivalent DFA intuitively.

2) Incomplete NFA to DFA conversion process in JFLAP: JFLAP is software for experimenting with formal languages topics, such as Non-deterministic finite automatas (NFA), which was developed by Susan H. Rodger, Duke University using Java. In order to verify the correctness of the algorithm created in this paper, JFLAP was used to cross-check. During the cross-checking process, it became apparent that JFLAP's NFA to DFA conversion algorithm is incomplete because it does not generate trap states. Therefore, our implementation can be argued to be more complete because of this missing feature.

V. Conclusion

Although the process of converting an NFA to a DFA seems simple when doing it manually, it is much more complicated to convert this process into code. One of the most challenging parts during development was to determine the transitions for the new state. For this part, it was necessary to look at every state from the original NFA that compose the new state, then search for all the states in the transition table and finally join the closure for those states. The use of data structures such as classes and stacks facilitated the development.

There are a lot of details involved in creating the DFA. For example, the algorithm must keep track of the initial and final states, check to see if a trap state is necessary, and perform the closure of any lambda transitions present in the NFA, and so on. The algorithm needed to handle all these details in order to create the correct DFA, for example, there was one bug that we had where the initial state of the DFA was not final, but in the correct answer, the initial state should also be final. That's because when there's a λ-transition from the initial state to a final state in the original NFA, the initial state has to be a final state in the DFA, we noticed and fixed this bug during the testing phase. Additionally, after the DFA is created, there was also the reduction procedure which removed useless or repetitive states and transitions.

References

[1] Linz, Peter. An Introduction to Formal Languages and Automata. Jones & Bartlett Learning, 2012.

[2] "Vis.js Community Edition *." Visjs.org, https://visjs.org/

[3] "JFLAP." JFLAP, http://www.jflap.org/