Convert Nondeterministic Finite Automata to Minimized Deterministic Finite Automata

XIMIN YAN, United International College

This project study aims to develop a program implemented by Python which can convert *Nondeterministic Finite Automata*(NFA) to a minimized *Deterministic Finite Automata*(DFA) automatically. For a better capability of reusing, I defined two classes with respect to NFA and DFA. With predefined NFA, the program can execute directly without any input.Then interior implemented functions will process and get the initial DFA first.Next,the initial DFA will be processed again by another functions and procure minimized DFA eventually. In this program, users can modify the predefined NFA by themselves and obtain correct results as they expect.

Categories and Subject Descriptors: C.3.8 [**Finite Automata**]: Principle of Programming

Additional Key Words and Phrases: NFA, DFA, Finite Automata, Python, Classes

1. INTRODUCTION

Programming languages are tools for programmers to construct the real world, and all the software and programs running on all devices were written in some programming languages. However, programming languages people utilized such as C, C++ or Java were high level languages which cannot be executed directly by machines.Actually, before a program can be run, it first must be translated into a form in which it can be executed by a machine and this process is conducted by the compiler.

Compiler is a sort of software system whose function is to convert modified source program to target assembly program. During the converting process, actually, the compiler experiences seven different phases and *figure 1* illustrates the procedure clearly.

For the whole compile system, lexical analysis is the fundamental. The lexical analyzer reads the stream of characters making up the source program and groups the character into meaningful sequences called lexemes. When it comes to lexical identity, Finite automata is no doubt a efficient approach to solve it. From holistic level, Finite automata is just like a transition diagram with a few difference:

(1) Finite automata are recognizers;they simply say ”yes” or ”no” about each possible input string.

(2) Finite automata come is two flavors:
   (a) *Nondeterministic finite automata*(NFA) have no restriction on the labels of their edges. A symbol can label several edges out of the same state, and ϵ, the empty string, is a possible label.
   (b) *Deterministic finite automata*(DFA)have, for each state, and for each symbol of its input alphabet exactly one one edge with that symbol leaving that state.

Both nondeterministic finite automata and deterministic finite automata are capable of recognizing the same language which can be represented by *regular languages*, but now our project just focuses on NFA and DFA.

2. FINITE AUTOMATA

2.1. Nondeterministic Finite Automata

A *nondeterministic finite automaton* (NFA) consists of:

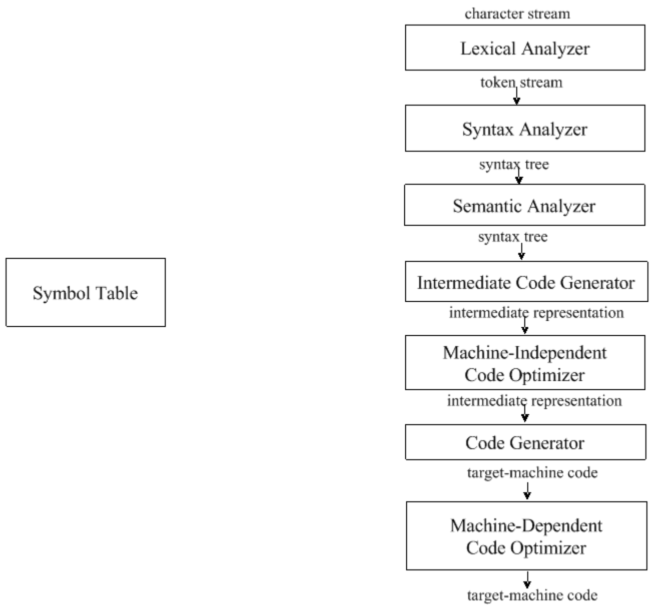


Fig. 1. Phrase of a Compiler

(1) A finite set of states S

(2) A set of input symbol $\sum$, the *input alphabet*. We assumes that ϵ,which stands for the empty string, is never a member of $\sum$.

(3) A *transition function* that gives, for each function, and for each states in $\sum \cup \epsilon$ a set of *next states*

(4) a state *S0* from s that is distinguished as the *start state*(*or initial state*)

(5) A set of states F, a subset of S, that is distinguished as the *accept states*(*final states*)

We can represent either an NFA or DFA by a *transition graph*, where the nodes are states and the labeled edges represent the transition function. There is an edge labeled a from state s to state t if and only if t is one of the next states for the state s and input a. This graph is very much like a transition diagram,except:

(1) The same symbol can label edges from one state ro several different states, and

(2) An edge may be labeled by ϵ, the empty string, instead of ,or in addition to, symbols from the input alphabet.

2.1.1. The expression of Nondeterministic Finite Automata. The most transparent way to represent NFA is states graph which looks like state transition diagram. Now, here is an example to illustrate all strings of a's and b's ending in the particular string *abb* by *Figure 2*.

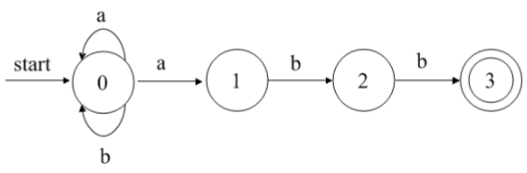
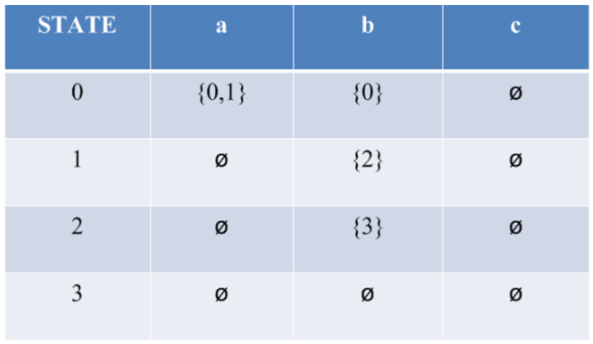


Fig. 2. A nondeterministic finite automaton

In *Figure 2*, the double circle around state 3 indicates that this state is accepting. Notice that the only ways to get from the start state 0 to the accepting state is to follow some path that states in state 0 for a while, then goes to state 1,2, and 3 by reading *abb* from the input. Thus, the only strings getting to the accepting state are those that end in *abb*

We can also represent an NFA by a *transition table*, whose rows correspond to states, whose columns correspond to the input symbol and ϵ. The entry for a given state and input is the value of the transition function applied to those arguments. If the transition function has no information about that state-input pair, we put $\emptyset$ in the table for the pair. *Figure 3* is an NFA transition table.



STATE	a	b	c
0	{0,1}	{0}	ø
1	ø	{2}	ø
2	ø	{3}	ø
3	ø	ø	ø

Fig. 3. Transition table for the NFA

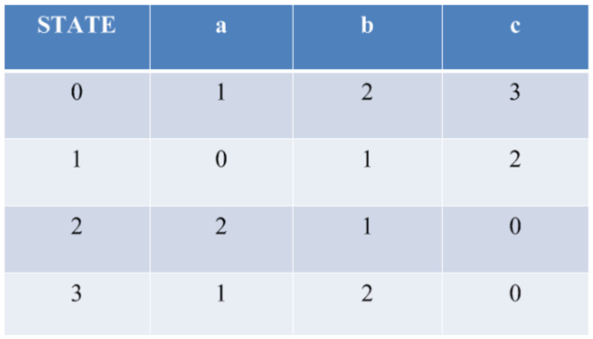
2.2. Deterministic Finite Automata

A *deterministic finite automaton* (DFA) is a special case of an NFA where:

(1) There are no moves on input ϵ, and

(2) For each state s and input symbol a, there is exactly one edge out of s labeled a.

If we are using transition table to represent a DFA, then each entry is a single state. We may therefor represent this state without the curly brace that we use to form sets. As *Figure 4* shows:



STATE	a	b	c
0	1	2	3
1	0	1	2
2	2	1	0
3	1	2	0

Fig. 4. Transition table for the DFA

And transition diagram as *Figure 5* shows.

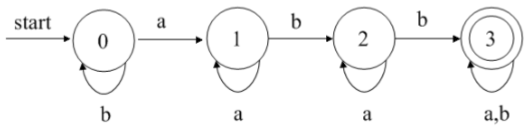


Fig. 5. A nondeterministic finite automaton

While the NFA is an abstract representation of an algorithm to recognize the strings of a certain language, the DFA is a simple, concrete algorithm for recognizing strings. It is fortune indeed every NFA can be converted to a DFA accepting the same language, because it is the DFA that we really implemented or simulate when building lexical analyzers. And it is also,in part, the target of this project.

3. IMPLEMENTATION BY PYTHON

The target of this project is to convert an NFA to a minimized DFA automatically. The implementation is in two main steps:

(1) Convert an NFA to a DFA

(2) Minimize the DFA

Ahead of commencing the core part, I intend to introduce Python which is a programming language utilized in this project. I have used several concepts of Python, such as *list*,*dictionary*,and *tuple*,etc.

3.1. Conversion of an NFA to a DFA

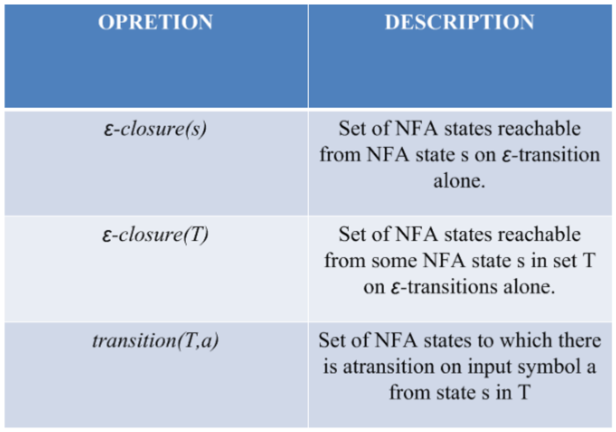
The method I used is the subset construction whose general idea is that each state of the constructed DFA corresponds to a set of NFA states. After reading input a_1,a_2,...a_n, the DFA is tin that state which correspond to the set of states that the NFA can reach, from its start state, following paths labeled a_1,a_2,...a_n.

ALGORITHM 1: The subset construction of a DFA from an NFA

Input: An NFA N

Output: A DFA D accepting the same language as N

Method: My algorithm constructs a transition table $Dtran$ for D. Each state of D is a set of NFA states, and we construct $Dtran$ so D will simulate ”in parallel” all possible moves N can make on a given input string. Our first problem is to deal with ϵ-transitions of N properly. In *Figure 6*, you can see the definition of several functions that describe basic computations on the states of N that are needed in the algorithm. Note that s is a single state of N, while T is a set of states of N.



OPRETION	DESCRIPTION
ε-closure(s)	Set of NFA states reachable from NFA state s on ε-transition alone.
ε-closure(T)	Set of NFA states reachable from some NFA state s in set T on ε-transitions alone.
transition(T,a)	Set of NFA states to which there is atransition on input symbol a from state s in T

Fig. 6. Operations on NFA states

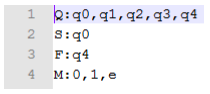
3.1.1. Define NFA in files. In this project, I utilized two *.txt* files to define my own NFA. They were consisted of five items, such as the start states like q_0, the moving like *0,1,*ϵ, the accepted states like q_4 and all the states like q_0, q_1, q_2, q_3. In the first file named *transitionTable.txt*, I defined the transition function; and in the *other4.txt* file, the rest of element were defined. It is really convenient and performed, because clients can easily modify any element of the five ones. However, clients should comply with several rules as following:

(1) Clients cannot add the empty lines in the two files. For example, between first character Q and S in *other4.txt*,there are no empty lines. Otherwise, the system will show you some errors.

(2) Clients can only modify the content after colon obeyed some format in *other4.txt*. Similarly, in the *transitionTable.txt* file, only the items after the colon can be modified by users freely, ahead of it, users cannot be permitted to modify it, such as deleting. But users can add items, for instance, q_0 has three moves such as *0,1,*ϵ, based on them, clients can add the fourth move, that is 2 which format is the same as previous three ones.

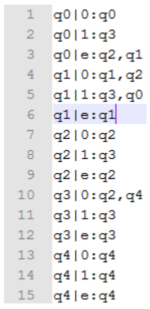
(3) If one or some states come to another states without consuming some moves, users should add empty set which I defined as *'o'* in *other4.txt*.Above all, users should modify two files if he or she wants to add empty set, first, you need to add *o* states in *other4.txt* file, then you need to append empty set after colon if this state does not consume particular moves.

Actually, using file as an input of NFA is a better choice, users can modify the content conveniently and input any NFA they want on the condition that following the rules.



```
Q:q0,q1,q2,q3,q4
S:q0
F:q4
M:0,1,e
```

Fig. 7. The predefined NFA in *other4.txt*



```
q0|0:q0
q0|1:q3
q0|e:q2,q1
q1|0:q1,q2
q1|1:q3,q0
q1|e:q1
q2|0:q2
q2|1:q3
q2|e:q2
q3|0:q2,q4
q3|1:q3
q3|e:q3
q4|0:q4
q4|1:q4
q4|e:q4
```

Fig. 8. The predefined NFA in *transitionTable.txt*

In *Figure 7*, Q indicated how many states the does an NFA have; S meant the start state, such as q_0; F meant the acceptance and M was the consuming bit when states translation. And in *Figure 8*, I defined the transition function independently, vertical bar was the separated notation to divide the line into two parts, one was the state, the other were moves and target states. Then I divided the right part into another two parts too, separating moves and target states. Finally, these information can be analyzed into *dictionary* and *list* of python.

3.1.2. ϵ-NFA to NFA. Here I give the Pseudo code for computing *ϵ-closure* (T), and this process is a straightforward search in a graph from a set of states, and in this case I give, imagine that only the ϵ-labeled edges are available in the graph:

```
push all states of T onto stack
initialize e_closure(T) to T
While stack is not empty:
        pop t, the top element, off stack
        for each state u with an edge from t to u labeled e
            if u is not in e_closure(T):
                add u to e_closure(T)
                push u onto stack
```

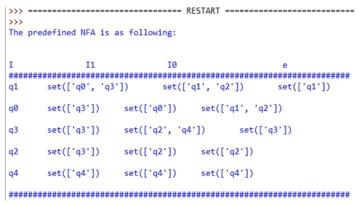
Before I paste the a little python code, I intend to introduce my structure of program for you. I defined two classes refer to others work in web-sites. One is called *NFA*, the other is *DFA*. And in these two classes, I defined many functions to implement the project.

Following is the python code to implement *ϵ-closure* (T):

```
def eclosure(self,state):
                equal = self.deltaHat(state,self.epsilon)
                return equal


def eclosureSet(self,state):
                equal = self.deltaHatSet(state,self.epsilon)
                return equal


def pop(self,temp,inputString):
          tempo = temp
          temps = temp.copy()
          temps = frozenset(temps)
          temps = set([temps])
          while len(temps) > 0:
               xSet = temps.pop()
               tempx = set([])
               print(xSet)
               for a in inputString:
                    for state in xSet:
                         tempx = self.delta[state][a]
                         tempx1 = frozenset(tempx)
                         if tempx1 <= xSet:
                              continue
                         else:
                              temps.add(tempx1)
                              tempo |= tempx
          return tempo
```

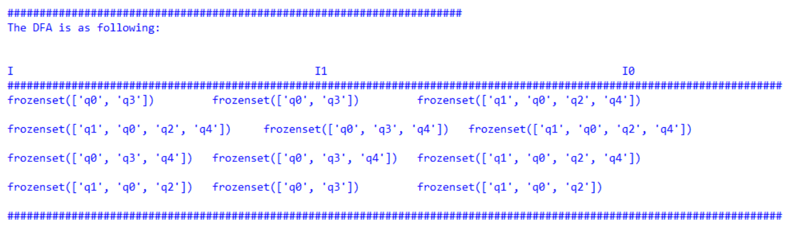
In these three functions, *eclosure(self,state)* and *eclosureSet(self,state)* are used to computing the next state from an exact state consumed ϵ. And *pop(self,temp,inputString)* is for iteration purpose to find all the next states from the set we just get,which consumed ϵ.

3.1.3. NFA to DFA. The algorithm for computing DFA from NFA just like I written before in *Algorithm 1*, here I will not give any code for you, but you can see it in my code function called *convertNFAtoDFA(N)*. Here I just show the NFA in console(*Figure 9*) and the DFA of result(*Figure 10*):



```
>>> ================================ RESTART ============================
>>>
The predefined NFA is as following:


I               I1                I0                        e
########################################################################
q1      set(['q0', 'q3'])       set(['q1', 'q2'])       set(['q1'])

q0      set(['q3'])     set(['q0'])     set(['q1', 'q2'])

q3      set(['q3'])     set(['q2', 'q4'])       set(['q3'])

q2      set(['q3'])     set(['q2'])     set(['q2'])

q4      set(['q4'])     set(['q4'])     set(['q4'])

########################################################################
```

Fig. 9. The predefined NFA in Python



```
########################################################################
The DFA is as following:


I                                                I1                                               I0
##########################################################################################################################
frozenset(['q0', 'q3'])         frozenset(['q0', 'q3'])         frozenset(['q1', 'q0', 'q2', 'q4'])

frozenset(['q1', 'q0', 'q2', 'q4'])     frozenset(['q0', 'q3', 'q4'])  frozenset(['q1', 'q0', 'q2', 'q4'])

frozenset(['q0', 'q3', 'q4'])   frozenset(['q0', 'q3', 'q4'])   frozenset(['q1', 'q0', 'q2', 'q4'])

frozenset(['q1', 'q0', 'q2'])   frozenset(['q0', 'q3'])         frozenset(['q1', 'q0', 'q2'])

##########################################################################################################################
```

Fig. 10. The unminimized DFA

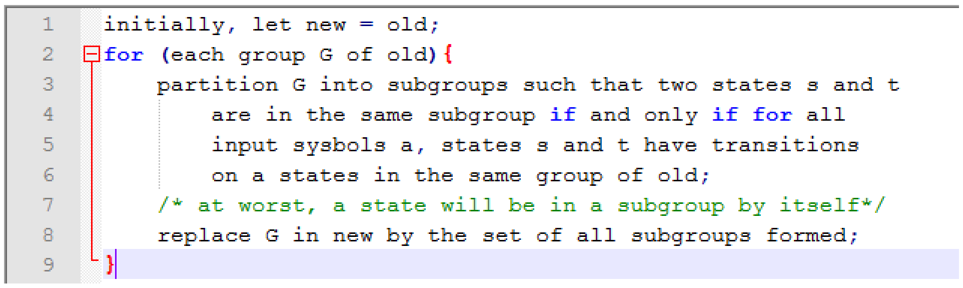
4. MINIMIZE DFA

There can be many DFA's that recognize the same language. When we implement a lexical analyzer as a DFA, we would generally prefer a DFA with as few states as possible, since each state requires entries in the table that describes the lexical analyzer. In this part, I implemented the function to minimize DFA.

METHOD:

(1) Start with an initial partition *old* with two groups, *F* and *S-F*, the accepting and nonaccepting states of *D*.

(2) Apply the procedure *Figure 9* to construct a new partition *new*



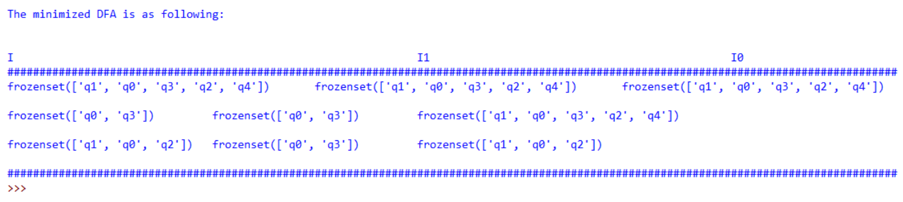
```
initially, let new = old;
for (each group G of old){
    partition G into subgroups such that two states s and t
        are in the same subgroup if and only if for all
        input sysbols a, states s and t have transitions
        on a states in the same group of old;
    /* at worst, a state will be in a subgroup by itself*/
    replace G in new by the set of all subgroups formed;
}
```

Fig. 11. Construction of new

(3) If *new* = *old*, let *final* = *old* and continue with step (4).

(4) Otherwise, repeat step (2) with *new* in place of *old*

Here is the result of this algorithm:



```
The minimized DFA is as following:


I                                                I1                                               I0
##########################################################################################################################
frozenset(['q1', 'q0', 'q3', 'q2', 'q4'])       frozenset(['q1', 'q0', 'q3', 'q2', 'q4'])       frozenset(['q1', 'q0', 'q3', 'q2', 'q4'])

frozenset(['q0', 'q3'])         frozenset(['q0', 'q3'])         frozenset(['q1', 'q0', 'q3', 'q2', 'q4'])

frozenset(['q1', 'q0', 'q2'])   frozenset(['q0', 'q3'])         frozenset(['q1', 'q0', 'q2'])

##########################################################################################################################
>>>
```

Fig. 12. Minimized DFA

5. CONCLUSION

This project perfectly implements the initial target, which is converting an ϵ- NFA to a minimized DFA automatically. And important functions as I finished this project are: *minimizeDFA*,*convertNFAtoDFA* and *eclosure*,etc. NFA to DFA and get minimized DFA is the first step when we want to implement more complicated function, like syntax analyzer, and semantic analyzer, even a compiler. So this project is really important as a fundamental to others. After implementing this automata, we can analyze a input character string by using it. In general, this project is decent, although the procedure is tough, yet I've got a decent result and users can modify the predefined NFA in two files whatever they want(*but need to follow the rules I defined*), implementing all the requirements.It is plausible that I can do better if there is enough time.

6. REFERENCE

Principles,Techniques,Tools,made by Alfred V. Aho Monica S.Lam Ravi Sethi Jeffrey D. Ullman.

Course Project http://www-bcf.usc.edu/~breichar/teaching/2011cs360/NFAtoDFA.py

MICHAEL ALTFIELD, ON APRIL 7TH, 2011 http://tech.michaelaltfield.net/2011/04/07/regex-2-dfa-in-python/