Compiler Report

Laxman Prabhakar **Assignment-1**

1401CS22 31/01/2017

1. Write an algorithm that converts RE to NFA. What is the complexity of the algorithm?

Answer:

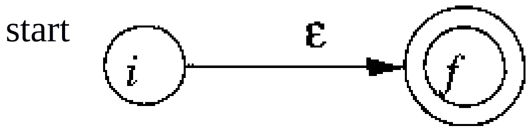
The McNaughton-Yamada-Thompson algorithm to convert a regular expression to an NFA:

INPUT: A regular expression *r* over alphabet *S*.

OUTPUT: An NFA *N* accepting L(*r*).

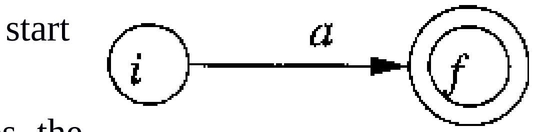
METHOD: Begin by parsing *r* into its constituent subexpressions. The rules for constructing an NFA consist of basis rules for handling subexpressions with no operators, and inductive rules for constructing larger NFA's from the NFA's for the immediate subexpressions of a given expression.

BASIS: For expression ε construct the NFA



Here, *i* is a new state, the start state of this NFA, and *f* is another new state, the accepting state for the NFA.

For any subexpression *a* in *S*, construct the NFA



where again *i* and *f* are new states, the start and accepting states, respectively.

Note that in both of the basis constructions, we construct a distinct NFA, with new states, for every occurrence of ε or some *a* as a subexpression of *r*.

INDUCTION: Suppose N(*s*) and N(*t*) are NFA's for regular expressions *s* and *t*, respectively.

a) Suppose *r* = *s*|*t*. Then N(*r*), the NFA for *r*, is constructed as in Fig.

Here, *i* and *f* are new states, the start and accepting states of N(*r*), respectively. There are ε-transitions from *i* to the start states of N(*s*) and N(*t*), and each of their accepting states have ε-transitions to the accepting state *f*. Note that the accepting states of N(*s*) and N(*t*) are not accepting in N(*r*). Since any path from *i* to *f* must pass through either N(*s*) or N(*t*) exclusively, and since the label of that path is not changed by the ε's leaving *i* or entering *f*, we conclude that N(*r*) accepts L(*s*) ∪ L(*t*), which is the same as L(*r*). That is, Fig. is a correct construction for the union operator.

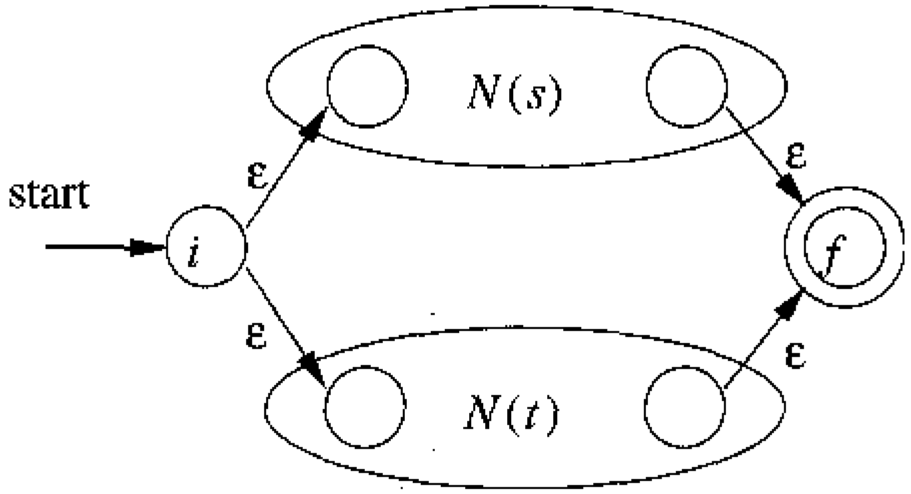


Figure: NFA for the union of two regular expressions

Suppose *r* = *st*. Then construct N(*r*) as in Fig. The start state of N(*s*) becomes the start state of N(*r*), and the accepting state of N(*t*) is the only accepting state of N(*r*). The accepting state of N(*s*) and the start state of N(*t*) are merged into a single state, with all the transitions in or out of either state. A path from *i* to *f* in Fig. must go first through N(*s*), and therefore its label will begin with some string in L(*s*).

The path then continues through N(*t*), so the path's label finishes with a string in L(*t*). As we shall soon argue, accepting states never have edges out and start states never have edges in, so it is not possible for a path to re-enter N(*s*) after leaving it. Thus, N(*r*) accepts exactly L(*s*)L(*t*), and is a correct NFA for *r* = *st*.

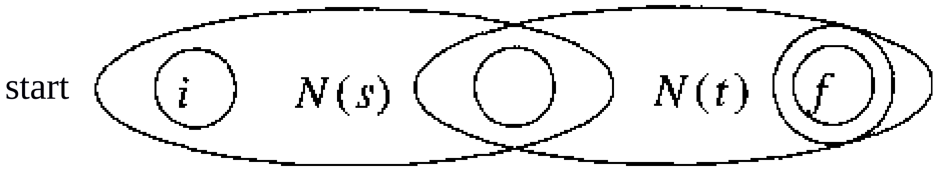
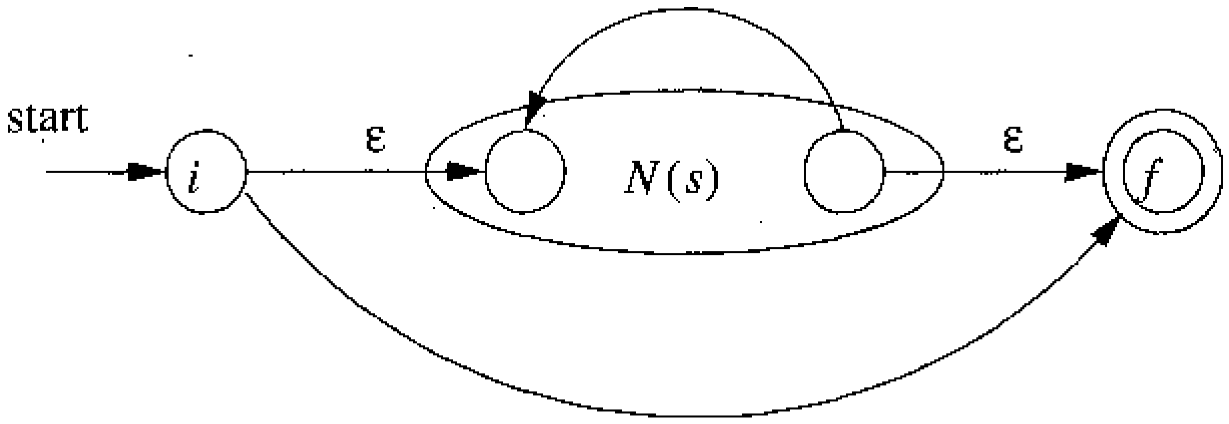


Figure: NFA for the concatenation of two regular expressions

c) Suppose *r* = *s**. Then for *r* we construct the NFA N(r) shown in Fig.

Here, *i* and *f* are new states, the start state and lone accepting state of N(*r*). To get from *i* to *f*, we can either follow the introduced path labelled ε, which takes care of the one string in L(*s*)^0, or we can go to the start state of N(*s*), through that NFA, then from its accepting state back to its start state zero or more times. These options allow N(*r*) to accept all the strings in L(*s*)^1, L(*s*)^2, and so on, so the entire set of strings accepted by N(*r*) is L(*s**).



d) Finally, suppose *r* = (*s*). Then L(*r*) = L(*s*), and we can use the NFA N(*s*) as N(*r*).

Complexity:

For Constructing NFA we have to make a syntax tree and then do an inorder traversal. Let the length of string formed from inorder traversal of *r* is *n*. Then we will construct NFA according to the String formed from inorder traversal, and as $m \geq n$ overall complexity of converting RE to NFA is O(*n*),where *m* is length of string formed from inorder traversal of syntax tree formed from regular expression *r*.

Q2. Write an algorithm to simulate the NFA in order to verify the acceptance of a string. Describe the complexity of your algorithm.

Answer:

Algorithm: Simulating an NFA.

INPUT: An input string *x* terminated by an end-of-file character *eof*. An NFA *N* with start state *SQ*, accepting states *F*, and transition function move.

OUTPUT: Answer "yes" if *M* accepts *x*; "no" otherwise.

METHOD: The algorithm keeps a set of current states *S*, those that are reached from so following a path labeled by the inputs read so far. If *c* is the next input character, read by the function *nextCha()*, then we first compute *move(S,c)* and then close that set using ε-closureQ. The **algorithm** is sketched in below:

```
1. S = ε-closure(S₀);
2. c = nextChar();
3. while ( c != e o f ) {
4.       S = e-closure(move(S,c));
5.       c = nextCharQ;
6.  }
7.  if ( S∩F != 0 ) return "yes";
8. else return "no";
```

Complexity

Complexity of the algorithm can be calcuated by analysing the complexity of each step of the algorithm. The complexity of calculating the ε-closure is $O(n^2)$ as for each state in the stack we have to traverse all the states in worst case so overall complexity of calculating ε-closure is $O(n^2)$. As there are *m* characters in the string for each character we have to compute ε-closure so the overall complexity of simulating NFA to accept the string is $O(mn^2)$.

Q3. Write an algorithm to convert NFA into DFA. What is the complexity?

Answer:

Algorithm: The subset construction of a DFA from an NFA.

INPUT: An NFA *N*.

OUTPUT: A DFA *D* accepting the same language as *N*.

METHOD: Our algorithm constructs a transition table for *D*. Each state of *D* is a set of NFA states, and we construct table 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 table, we see the definitions 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*.

OPERATION	**DESCRIPTION**
ε-closure(*s*)	Set of NFA states reachable from NFA states on ε-transitions alone.
ε-closure(*T*)	Set of NFA states reachable from some NFA state *s* in set T on ε-transitions alone; = U_s i_n T ε-closure(s).
move(*T*, *a*)	Set of NFA states to which there is a transition on input symbol a from some state *s* in *T*.

We must explore those sets of states that *N* can be in after seeing some input string. As a basis, before reading the first input symbol, *N* can be in any of the states of ε-closure(S_0), where S_0 is its start state. For the induction, suppose that *N* can be in set of states *T* after reading input string *x*. If it next reads input *a*, then *N* can immediately go to any of the states in move(*T*, *a*). However, after reading *a*, it may also make several ε-transitions; thus *N* could be in any state of ε-closure(move(*T*, *a*)) after reading input *xa*. Following these ideas, the construction of the set of states, *Dstates*, and its transition function *Dtran*, is shown below.

The start state of *D* is ε-closure(S_0), and the accepting states of *D* are all those sets of *AT*'s states that include at least one accepting state of *N*. To complete our description of the subset construction, we need only to show how initially, ε-closure(S_0) is the only state in *Dstates*, and it is unmarked;

```
while ( there is an unmarked state T in Dstates ) {
      mark T;
               for ( each input symbol a ) {
               U = e-closure(move(T,a));
                if ( U is not in Dstates )
                    add U as an unmarked state to Dstates;
                            Dtran[T, a] = U;
             }
}
```

ε-closure(T) is computed for any set of NFA states T. This process, shown, is a straightforward search in a graph from a set of states. In this case, 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;
             }
}
```

Complexity:

The complexity mainly depends on the number of state which will be formed in the output DFA. As complexity of calculating ε-closure is $O(n^2)$. So the worst case complexity of converting NFA to DFA is $O(n^2 \bullet 2^n)$ as number of states will be all the sets which can be formed from given number of states in NFA multipled by ε-closure complexity. As maximum number of states that can be formed from given NFA is of order 2^n where n is number of states in NFA overall complexity is $O(n^2 \bullet 2^n)$.

4. Write an algorithm to verify the acceptance/rejection of a string by DFA. Also compute the complexity.

Answer:

Algorithm: Simulating a DFA.

INPUT: An input string x terminated by an end-of-file character *eof*. A DFA D with start state S_0, accepting states F, and transition function move.

OUTPUT: Answer "yes" if D accepts x; "no" otherwise.

METHOD: Apply the algorithm in to the input string x. The function move(s,c) gives the state to which there is an edge from state s on input c.

The function nextChar returns the next character of the input string x.

```
S = S0
c = nextCharQ;
while ( c != e o f ) {
s = move(s,c);
5. c = nextCharQ;
6. }
7. if ( s is in F ) return "yes";
8. else return "no";
```

Complexity:

Complexity of the algorithm will be calculated by analysing the complexity of each step of the algorithm. In simulating DFA to accept string we have to traverse each character in the string. As transition function is already given in input time complexity for move(S_0, c) is O(1) . So overall complexity is O(m), as complexity for traversing the string is O(m).

5. Compute the complexity of the algorithm that converts RE to DFA directly.

Answer:

- Regular expression can be directly converted into a DFA (without creating a NFA first)
- Augment the given regular expression by concatenating it with a special symbol #

$r \rightarrow (r)$# augmented regular expression

- Create a syntax tree for this augmented regular expression
- Syntax tree
    - Leaves: alphabet symbols (including # and the empty string) in the augmented regular expression
    - Intermediate nodes: operators
- Number each alphabet symbol (including #) depending upon the positions

Algorithm:

- Create the syntax tree of (r) #
- Calculate the functions: followpos, firstpos, lastpos, nullable
- Put firstpos (root) into the states of DFA as an unmarked state
- while (there is an unmarked state S in the states of DFA) ***do***
    - mark ***S***
    - for each input symbol ***a*** do
    - let S_1, ..., S_n are positions in ***S*** and symbols in those positions are ***a***
    - ***S'*** ← followpos(S_1)∪...∪followpos(S_n)
    - **move(*S*, *a*)** ← ***S'***
    - if (***S'*** is not empty and not in the states of DFA)
        - put ***S'*** into the states of DFA as an unmarked state
- the start state of DFA is firstpos (root)
- the accepting states of DFA are all states containing the position of #

Complexity:

We will compute the complexity by computing complexity at each step. The total number of states than can be formed in worst case with help of regular expression with n symbols is of order of 2^n. We have to traverse to each input symbol of regular expression so total time complexity in worst case must be $O(n \cdot 2^n)$ as complexity of while loop is $O(2^n)$ which is order of number of states that can be formed in worst case.

6. Compute the complexity of DFA minimization algorithm.

Answer:

- partition the set of states into groups:
  - G_1: set of accepting states
  - G_2:set of non-accepting states
- For each new group G
- Partition G into subgroups such that states S_1 and S_2 are in the same group iff for all input symbols a, states S_1 and S_2 have transitions to states in the same group
- Start state: the group containing the start state of the original DFA
- Accepting states: the groups containing the accepting states of the original DFA

Complexity:

We compute the complexity of the problem by analysing the algorithm step by step. We take partition of the non-final states and for each state we have to traverse the partition,
so the overall complexity of the algorithm is $O(n^2)$. As for each state in non-final state we have to traverse all the states. We can also do it by making a matrix of size $n \times n$.