Construction of a Minimal Deterministic Finite Automaton from a Regular Expression

Sanjay Bhargava
Department of Computer Science
Banasthali University
C-62, Sarojini Marg, C-Scheme, Jaipur - 302001

G. N. Purohit
Centre for Mathematical Science
Banasthali University
Banasthali, Rajasthan - 304022

ABSTRACT

This paper describes a method for constructing a minimal deterministic finite automaton (DFA) from a regular expression. It is based on a set of graph grammar rules for combining many graphs (DFA) to obtain another desired graph (DFA). The graph grammar rules are presented in the form of a parsing algorithm that converts a regular expression R into a minimal deterministic finite automaton M such that the language accepted by DFA M is same as the language described by regular expression R.

The proposed algorithm removes the dependency over the necessity of lengthy chain of conversion, that is, regular expression → NFA with ε-transitions → NFA without ε-transitions → DFA → minimal DFA. Therefore the main advantage of our minimal DFA construction algorithm is its minimal intermediate memory requirements and hence, the reduced time complexity. The proposed algorithm converts a regular expression of size n in to its minimal equivalent DFA in $O(n.\log_2 n)$ time. In addition to the above, the time complexity is further shortened to $O(n.\log_e n)$ for $n \geq 75$.

General Terms

Algorithms, Complexity of Algorithm, Regular Expression, Deterministic Finite Automata (DFA), Minimal DFA.

Keywords

Alphabet, Automaton Construction, Combined State, Union, Concatenation, Kleene Closure, Minimization, Transition.

1. INTRODUCTION AND BACKGROUND

Regular expressions and finite automata are two dissimilar representations for regular languages: regular expressions on one hand generate regular languages while on the other hand finite automata accept regular languages. It is well known that each regular expression can be transformed into a nondeterministic finite automaton (NFA) with or without ε-transitions, and finally this NFA can be converted into a DFA. In the literature related to the conversion problem, it has been found that there exist many different algorithmic approaches for converting a regular expression into some variant of a finite automaton; Watson [34] enumerated various algorithmic approaches for the conversion problem. Algorithmic approaches to convert a regular expression into some variant of a finite automaton include:

- The algorithms to convert regular expression into NFA with or without ε-transitions (see, e.g. [1], [5], [6], [12], [13], [16], [18], [20], [21], [23], [28], [33], [36] and [37]) and
- The algorithms to convert regular expression into DFA using intermediate NFAs (as in various studies like [2], [3], [4], [9], [10], [15], [19], [24], [25], [29], [30], [31], [32] and [35]).

Daciuk *et al.* [11] discussed a parsing algorithm to convert a set of strings into a minimal, deterministic, acyclic finite-state automaton. Carrasco and Forcada [8] presented another algorithm to modify any minimal finite-state automaton so that a string is added to or removed from the language accepted by it. Recently, Carrasco *et al.* [7] presented another algorithm that allowed the incremental addition or removal of unranked order trees to a minimal frontier-to-root deterministic finite-state tree automaton. All the above studies had limitations as they represented only a finite set of strings.

The traditional methods, discussed above, to convert a regular expression R into a minimal DFA M consist of four phases: (i) to convert R into a non-deterministic finite automaton (NFA) with ε-transitions, (ii) to convert the above NFA into an NFA without ε-transitions, (iii) to convert this NFA (without ε-transitions) into a DFA, and finally (iv) to minimize this DFA. This paper presents a parsing algorithm ***Construct*** which converts a regular expression (finite as well as infinite set of strings) into a minimal DFA using the following two phases, recursively.

(i) Constructing a DFA (for ε / ϕ / a single alphabet symbol / kleene closure of a DFA / union of two DFA / concatenation of two DFA) by applying the rules for automaton construction and joining operations of DFA (Cohen [10]), and

(ii) Minimizing the obtained DFA (Hopcroft and Ullman [19]).

In the proposed algorithm, parsing is performed by (i) obtaining the information from regular expression in a precise manner and then (ii) converting this information into DFA using the graph grammar rules defined for this conversion. Following Johnson *et al.* [22], Möhring [27] and Mayr *et al.* [26], we have prescribed a set of graph grammar rules for minimization, union, concatenation, and kleene closure operations over DFA and our parsing algorithm works as a genuine folder containing all these rules. Throughout the conversion process, graph operations needed to join DFA take place using an appropriate graph grammar rule from this folder.

The contents of this paper are arranged as follows.

Section 2 describes the proposed implementation procedure, followed by the equivalent algorithm to convert a regular expression into a minimal DFA in the next section. This section also depicts the instances of our algorithm's implementation initially by graphs and followed by tables. Finally, this section

ends with an evaluation of the algorithm. Last Section 4 deals with the conclusions of the present research paper.

2. IMPLEMENTATION PROCEDURE

Following Brzozowski and Cohen [6] and Antimirov [1], the proposed algorithm ***Construct*** first constructs a minimal DFA (graph) for the deepest positioned element of the given regular expression. In the next step, ***Construct*** combines this DFA with the DFA based on the surrounding part of deepest element's position, to obtain another minimal DFA. The process of combining DFA is continued till the entire regular expression is converted into the resultant DFA. The implementation procedure is described in detail as follows.

Construct first stores the given regular expression R as (R) in an array X. Therefore the start and end markers of R are '(' and ')' respectively. Then, following Berry and Sethi [3], ***Construct*** obtains the address of innermost parenthesis string say R_1 of R. ***Construct*** then scans R_1 for some x^* $\forall$ x $\in \Sigma \cup$ {ε, ϕ, M_z such that z ≥ 1 *(M_z will not exist for R_1 and can exist only for R_2 onwards)*}, and if any such x^* exists, ***Construct*** constructs the minimal DFA for those entire x^* *(**Construct** first constructs an intermediate DFA say I for x, and then for x^* **it** obtains the kleene closure DFA of I, and finally **it** stores the so obtained DFA in some L_i)*, and replaces all x^* by their equivalent minimal DFA name L_i for i ≥ 1 in R_1. ***Construct*** also stores all those L_i's in XL. Thereby, in the string R_1 all x^* are replaced by some L_i for i ≥ 1.

Construct again scans R_1 for some string s $\forall$ s $\in \Sigma^* \cup \{\phi\} \cup \{L_i, M_z \text{ for } i, z \geq 1\}^*$, and if any such s exists, ***Construct*** constructs the minimal DFA for all those s recursively *(**Construct** first constructs a DFA N_j for the 1st character of string s; then if s contains another character, **Construct** concatenates N_j by the DFA of that character and stores the resultant minimal DFA back in N_j, and this process continues till the whole s is consumed)* and replaces all s by their equivalent minimal DFA name N_j for j ≥ 1. ***Construct*** also stores all those N_j's in XN. Hence, in the string R_1, all s are replaced by some N_j for j ≥ 1.

R_1 is again scanned for a '+' operator. If a '+' does not exist, the only N_j value in R_1 is stored as M_1. However in the presence of a '+' operator, ***Construct*** assumes that it is between x and y $\forall$ x, y $\in$ {N_j for j ≥ 1}. ***Construct*** stores the left side operand of '+' as M_1 and joins this M_1 to the right side operand of '+' using union operation, and stores the result *(minimal DFA)* back in M_1; then ***Construct*** again searches for the next '+' operator in R_1, and if found ***Construct*** again joins M_1 to the right side operand of '+' using union operation, and stores the result *(minimal DFA)* back in M_1; ***Construct*** continues till the entire string R_1 is replaced by M_1. ***Construct*** also stores M_1 in XM.

Construct then replaces the entire (R_1) ("*(", followed by string R_1, followed by ")"*") by M_1. As the innermost left and right parentheses are removed, ***Construct*** has, possibly, a new innermost parenthesis *(the next higher parenthesis after the parenthesis for R_1)*. Let the innermost parenthesis string be R_2. ***Construct*** performs the same operation with R_2 as ***it*** had done with R_1 to get the next string R_3. Continuing like this, ***Construct*** will replace the entire regular expression R by an equivalent minimal DFA name M_z for z ≥ 1. Finally, the last M_z is printed and stored as the output M.

In the implementation procedure, inside the regular expression ***Construct*** performed all the replacement operations by replacing strings L_i, N_j, or M_z *(for i, j, z ≥ 1)* into the regular expression (as in Ben-David *et al.* [2]). The exact values of these L_i, N_j, and M_z were stored on XL, XN, and XM respectively. Whenever ***Construct*** replaced anyone of L_i, N_j, or M_z into the regular expression, the corresponding value of L_i, N_j, or M_z was either appended to the sets XL, XN, or XM respectively *(if L_i, N_j, or M_z was new)* or was updated in XL, XN, or XM respectively *(if L_i, N_j, or M_z was already existing)*. Similarly if ***Construct*** read anyone of L_i, N_j, or M_z inside a regular expression, the corresponding value of L_i, N_j, or M_z was extracted from XL, XN, or XM respectively, some join operation *(union, concatenation, or kleene closure)* was performed over them, and the resultant value was stored back in XL, XN, or XM respectively along with storing the same string name into the regular expression (*For example if L_3 was DFA for a^*, then 'L_3' was over-written in place of a^* in regular expression and the exact value of L_3 was also stored in XL. Similarly, if we wanted to concatenate N_3 followed by DFA for symbol 1, we first extracted the current value of N_3 from XN, concatenated it with the DFA for symbol 1, stored the resultant DFA back as N_3, stored the value of N_3 in XN at the previous place of N_3, and rewrote '$N_3$1' as 'N_3' inside the regular expression.*).

Following Rytter [28], ***Construct*** used a high-speed processor that performed all the graph operations in almost insignificant time. Whenever any of the graph grammar function (*Symbol, Union, Concat, Star, and Min*) was called inside the algorithm, the processor was activated and after activation it performed the following sequence of operations in a constant and insignificant time:

(i) It obtained the information from array X and used the function that activated it.

(ii) It executed the associated function (the function that activated the processor) with the input, as received from X. If this input was an already stored string name (L_i, N_j, or M_z), then the processor extracted the value of that string name from XL, XN, or XM and used that extracted value as input, otherwise the processor used the same read input.

(iii) It stored the output (L_i, N_j, or M_z) of the associated function by overwriting that output name (L_i, N_j, or M_z) in array X at the place from where the input was taken. It also stored the full description of L_i, N_j, or M_z in XL, XN, or XM respectively.

3. ALGORITHM

Algorithm Construct takes a regular expression R and a finite set of alphabet symbols Σ as input and prints the minimal DFA M as output such that L(M) = L(R). Figure 1 shows the algorithm Construct.

Algorithm Construct

Input: a regular expression R and a set Σ of alphabet symbols.

Output: a minimal DFA M such that L (M) = L (R).

```
type automata = (Q, Σ, δ, Q[1], F);
M : automata;
begin
        M ← Converter (R, Σ);
            [M stores the output sent by function Converter]
        print M;                          [output M is printed]
end
```

Figure 1. Algorithm Construct to print the minimal DFA equivalent to a regular expression.

Algorithm Construct calls the function Converter, which maintains all the replace operations in array X (*array X contains R*) for converting R into a minimal DFA M. Figure 2 shows the function Converter.

```
Function Converter (R, Σ)
begin
if R = ϕ OR R = 'ε' OR length(R) = 1 then
begin
return Symbol(R);      [function Symbol returns min. DFA
                        for ϕ / ε / reg. exp. of length 1]
end;
endif;
int n, n_1, i, j, z, inner, outer, left_paren, right_paren,
position_star, inter, count;
n ← length(R);
X : array [1..n+2] of Σ ∪ {(, ), #, L_i, N_j, M_z for i, j, z ≥ 1};
z ← 1; X[1] ← '(';
Store R in array X from 2^nd to (n+1)^th position; X[n+2] ← ')';
n_1 ← n+2;
10  i, j ← 1;
find the innermost bracket in X;
inner ← in such that X[in] contains the left parenthesis of
        innermost bracket;
outer ← out such that X[out] contains the right parenthesis of
        innermost bracket;
left_paren ← inner; right_paren ← outer;
scan X from X[inner] to X[outer] for all x* where x ∈ Σ ∪ {ε, ϕ,
M_z for z ≥ 1}
 if x* found then for every x* such that X[position_star] = x
  if X[position_star] = M_z then L_i ← Min(Star(X[position_star]))
  else L_i ← Min(Star(Symbol(X[position_star])))
  endif;
  X[position_star] ← L_i;     [X[position_star] stores the
                               automata name for x*]
 store L_i in XL;             [automata L_i for x* is stored in XL]
 i ← i +1;
shift the values of X[position_star+2] to X[n_1] to 1 position left
 in X;
 X[n_1] ← '#'; outer ← outer - 1;
 endif;
 if no x* found then exit endif;
endscan;
inter ← inner;
15  scan X from X[inter] to X[outer]
count ← inter + 1;
if X[count] ∈ Σ ∪ ε ∪ ϕ then N_j ← Symbol(X[count]) else N_j ←
X[count]
endif;
X[count] ← N_j;        [X[count] stores the automata name N_j]
store N_j in XN;               [automata N_j is stored in XN]
count ← count+1;
20  if X[count] ∈ Σ ∪ ε ∪ ϕ then
      N_j ← Min(Concat(N_j, Symbol(X[count])));
      X[count-1] ← N_j;    store N_j in XN;
      shift the values of X[count+1] to X[n_1] to 1 position left;
      X[n_1] ← '#';  outer ← outer-1;  go to 20;
  endif;
  if X[count] ∈ {L_i, M_z for i, z ≥ 1} then
      N_j ← Min(Concat(N_j, X[count]));
      X[count-1] ← N_j;    store N_j in XN;
      shift the values of X[count+1] to X[n_1] to 1 position left;
      X[n_1] ← '#'; outer ← outer-1; go to 20;
  endif;
  if X[count] = '+' then
      j ← j+1; inter ← count; go to 15;
  endif;
  if X[count] = ')' then
      j ← j+1;
  endif;
endscan;
scan X from X[inner] to X[outer]
M_z ← X[inner+1];     [Some N_j at X[inner+1] is stored in
                       name M_z]
X[inner+1] ← M_z; [X[inner+1] stores the automata name M_z]
store M_z in XM;          [automata M_z is stored in XM]
30  if X[inner+2] = '+' then
M_z ← Min(Union(M_z, X[inner+3]));
X[inner+1] ← M_z;
store M_z in XM;
shift the values of X[inner+4] to X[n_1] to 2 positions left;
X[n_1-1], X[n_1] ← '#'; outer ← outer-2;
go to 30;
endif;
if X[inner+2] = ')' then endif;
endscan;
z ← z+1;
shift the values of X[inner+1] to X[n_1] to 1 position left;
shift the values of X[outer] to X[n_1] to 1 position left;
replace '#' from X[n_1+1-(right_paren-left_paren)] to X[n_1];
n_1 ← n_1 – (right_paren-left_paren);
if n_1 = 1 then return X[n_1] else go to 10 endif;
                        [value of X[n_1] i.e. M_z is returned]
end function;
```

Figure 2. Function Converter to convert a regular expression R into minimal DFA.

Function **Converter** stores regular expression R in an array X from 2nd to (n+1)th position, where n = length(R). X[1] and X[n+2] are initialized as '(' and ')' respectively. Then **Converter** constructs the DFA using the following three phases recursively.

(i) During the first phase, **Converter** captures the innermost parenthesis string, say R_1. Inside R_1, it replaces all x^* (*x can be a symbol of Σ, or ϕ, or 'ε', or an already existing DFA name M_z*) by their equivalent minimal DFA name L_i in R_1 and also stores these L_i's in XL.

(ii) During the second phase, **Converter** again scans R_1 from left to right in search of some s $\forall$ s $\in \Sigma^* \cup \{\phi\} \cup$

$\{L_i, M_z \text{ for } i, z \geq 1\}^*$, and replaces all such s by their equivalent minimal DFA name N_j in R_1 and also stores N_j's in XN.

(iii) During the third phase, **Converter** again scans R_1 for '+' operators, and using 'union operation of DFA' it replaces the whole string R_1 by an equivalent DFA name M_1 and also stores M_1 in XM.

So, after the first recursion, the entire string R_1 is converted into some M_1. Then **Converter** replaces (R_1) by M_1 in array X. It again scans X for an innermost parenthesis string, say R_2. Now, **Converter** deals with R_2 in exactly the same way as it converted R_1 into M_1, to get M_2. This process continues until **Converter** is unable to find an innermost parenthesis, at which time it comes out from the recursive process, and sends the last DFA M_z to the calling algorithm. Function Converter makes use of some other functions depicting graph grammar rules for various graph operations. These functions are Shiftleft, Symbol, Union, Concat, and Min. Figure 3 shows the function Shiftleft.

Function Shiftleft (X, d_1, d_2)
begin
for d = d_1 to d_2
    begin
      X[d] ← X[d+1];
    end;
X[d_2+1] ← '#'; return X;
end function; **[cell's contents of X are shifted to one position left from d_1 to d_2 and last cell's content is made #]**

Figure 3. Function Shiftleft.

Function **Shiftleft** shifts the contents of X to one position left, starting from d_1, and also places a '#' sign to the last cell of X. Figure 4 shows the function Symbol.

Function Symbol (a)
begin
if a = 'ε' then
    begin
    return M ← ($\{q_0, q_1\}$, Σ, $\{\delta_\varepsilon(\{q_0, q_1\}, s) \leftarrow \{q_1\}\ \forall\ s \in \Sigma\}$, q_0, $\{q_0\}$);
    end; **[returns the minimal DFA for 'ε']**
endif;
if a = ϕ then
    begin
    return M ← ($\{l_0\}$, Σ, $\{\delta_\phi(l_0, s) \leftarrow \{l_0\}\ \forall\ s \in \Sigma\}$, l_0, ϕ);
    end; **[returns the minimal DFA for ϕ where ϕ contains only one state q_0]**
endif;
return M ← ($\{p_0, p_1, p_2\}$, Σ, $\{\delta_a(p_0, a) \leftarrow \{p_1\}, \delta_a(p_0, b) \leftarrow \{p_2\} \forall\ b \in \Sigma - \{a\}, \{\delta_a(\{p_1, p_2\}, c) \leftarrow \{p_2\}\ \forall\ c \in \Sigma\}$, p_0, $\{p_1\}$); **[returns the min. DFA for a single alphabet symbol a]**
end function;

Figure 4. Function Symbol obtaining min. DFA for 'ϕ' or 'ε' or alphabet symbol of length 1.

Function **Symbol** converts ε into a minimal DFA; the minimal DFA contains two states q_0 and q_1 out of which q_0, the start state, is the only final state and q_1 is a non-final state. There is no transition from q_0 enters back to q_0, and all the transitions from q_0 and q_1, for all alphabet symbols, enter to state q_1.

Function **Symbol** also converts ϕ into a minimal DFA; the minimal DFA contains only one state l_0. l_0 is the start state and is also a non-final state, and all the transitions from l_0, for all alphabet symbols, enter back to state l_0.

Function **Symbol** also converts an alphabet symbol 'a' into a minimal DFA; the minimal DFA contains three states p_0, p_1, and p_2 out of which p_0 is the start state and p_1 is the only final state. There is a transition that enters from p_0 to p_1 on the alphabet symbol 'a', and all other possible transitions from all the states enter to p_2 which is a non-final state. Figure 5 shows the function Union.

Function Union ($M_l(Q_l, \Sigma, \delta_l, l_0, F_l)$, $M_r(Q_r, \Sigma, \delta_r, m_0, F_r)$)
begin
l_length ← cardinality(Q_l); r_length ← cardinality(Q_r);
rename the states of Q_l as l_u and store corresponding δ_l for $0 \leq u \leq$ l_length–1;
rename the states of Q_r as m_v and store corresponding δ_r for $0 \leq v \leq$ r_length–1;
M ← (Q ← $\{[l_u m_v] \mid 0 \leq u \leq \text{l_length}-1, 0 \leq v \leq \text{r_length}-1\}$, Σ, δ_{union}, $[l_0 m_0]$, F);
F ← ϕ;
u ← 0;
while (u ≤ l_length–1)
begin
    v ← 0;
    while (v ≤ r_length–1)
    begin
        $\delta_{union}([l_u m_v], a) \leftarrow \{[l_g m_h] \mid \delta_l(l_u, a) = l_g \text{ and } \delta_r(m_v, a) = m_h\ \forall\ a \in \Sigma, \text{ and } g, h \geq 0\}$;
        if $l_u \in F_l$ OR $m_v \in F_r$ then
        F ← F ∪ $\{[l_u m_v]\}$
        endif;
        v ← v+1;
    end;
    u ← u+1;
end;
rename $[l_0 m_0]$ by r_0 and rename other states of Q as r_u for $1 \leq u <$ l_length x r_length and also rename all these states in F and store corresponding δ_{union};
return M ← (Q, Σ, δ_{union}, r_0, F);
end function;

Figure 5. Function Union obtaining union of two DFA.

Function **Union** joins two DFA with respect to the union operation by combining the states. **Union** first combines the start states of the two DFA and then makes this combined state as the start state of the resultant DFA. Let this combined state be $[l_0 m_0]$, where l_0 and m_0 are the start states of two inputs DFA respectively. The transition from state $[l_0 m_0]$ for an alphabet symbol 'a' enters into state $[l_g m_h]$ provided that transition from state l_0 for 'a' enters into state l_g, and transition from state m_0 for 'a' enters into state m_h in the respective DFA. **Union** continues this process of generating and connecting new states, by means of transitions, until it has no more new states to connect and all

the possible transitions enter in to some state. Finally, all those combined states in this resultant DFA become final, which have any of their component states as a final state in the inputs DFA. Figure 6 shows the function Concat.

```
Function Concat (Ml(Ql, Σ, δl, l0, Fl), Mr(Qr, Σ, δr, m0, Fr))
begin
l_length ← cardinality(Ql); r_length ← cardinality(Qr);
rename the states of Ql as lu and store corresponding δl for 0 ≤ u
≤ l_length–1;
rename the states of Qr as mv and store corresponding δr for 0 ≤
v ≤ r_length–1;
Σ° ← addlast(Σ, '#');   [adds # after all the elements of set Σ
                         to get Σ°]
jc, tc, t ← 1, ktemp, jtemp, w : integer;
l, kc : string; a : char;
S : array[variable length] of strings;
if l0 ∈ Fl then
    begin
    kc ← l0m0; S[jc] ← l0m0;
    end
else
    begin
    kc ← l0; S[jc] ← l0;
    end
endif;
a ← first element of Σ°;        [combines l0 with m0 if l0 is a
                                 final state]
while (jc > 0)
    begin
    if length(kc) = 2 then      [length(s) gives the length of
                                 string s i.e. length(l1m2m4) = 6]
        begin
        l ← δl(kc, a);
        end
    else
        begin
        l ← δl(substring(kc, 1, 2), a);
        w ← 2;
        while (w ≤ length(kc)-1)
          begin
          l ← append(l, δm(substring(kc, w+1, 2), a));
          w ← w+2;
          end;                          [append(a, b) = ab]
        end;
    δ(kc, a) ← l;
    if substring(l, 1, 2) ∈ Fl then
        begin
        jc ← jc+1;
        if m0 in l then remove m0 from l;
        δ(kc, a) ← append(l, m0);
        S[jc] ← δ(kc, a);
        l ← δ(kc, a);                   [append m0 at the end of l]
        end
    else
        begin
        jc ← jc+1; δ(kc, a) ← l; S[jc] ← l;
        end;
    rewrite l such that suffixes of states are in non-decreasing
    order;
    rewrite l such that it does not contain duplicate occurrence
    of states;    [l0m1m3m2→ l0m1m2m3, l0m1m1m3→ l0m1m3]
        S[jc] ← l; δ(kc, a) ← l;
        for ktemp = 1 to jc-1
          begin
          if S[ktemp] = S[jc] then
            begin
            jc ← jc-1;
            end;
          end;
    [does not allow duplicate state names to enter in array S]
        t ← t+1;
        a ← next element of Σ°;
        if a = '#' then
          begin
          a ← first element of Σ°; ic ← ic+1;
          for jtemp = ic to jc
          begin
          restore the strings in array S such that length(S[jtemp])
          is in non-decreasing order;
          end;        [if S[1]=l0m0m1 & S[2]=l0m2 they will be
                       restored as S[1]=l0m2 & S[2]=l0m0m1]
          end
        else
          begin
          t ← t-1;
          end;
        kc ← S[t];
        if t > jc then
          begin
          exit;
          end;
end;                                    [of while (jc > 0)]
F ← ϕ;
for ktemp = 1 to jc
    begin
    Q[ktemp] ← S[ktemp];
    if mi in S[ktemp] such that mi ∈ Fr then
        begin
        F ← F ∪ S[ktemp];
        end;
    end;
rename the states of Q as sc and store corresponding δcon for c ≥
0 and also rename the corresponding states in F;
return M ← (Q, Σ, δcon, s0, F);
end function;
```

Figure 6. Function Concat obtaining concatenation of two DFA.

Function **Concat** joins two DFA, M_1 followed by M_2, using the following rule. To construct M_1M_2, **Concat** first constructs that part of M_1, which does not contain any final state of M_1. For the remaining construction, whenever **Concat** reaches to a final state p of M_1, it clubs this final state p with the start state q of M_2 to get a combined state [pq], and also renames state p as the combined state [pq]. Now **Concat** obtains the remaining states of M_1M_2 using the following rule. If a transition enters into a combined state [lg] for some alphabet symbol 'a', which contains any of its component states as a final state of M_1, **Concat** clubs this combined state with the start state q of M_2 to get a combined

state [lgq] *(new or already existing)*, and that transition, for the same alphabet symbol 'a', now enters into this combined state [lgq]. Then **Concat** constructs the remaining part of M_1M_2 using the same logic which was applied in function **Union** for combined states. Finally, all those combined states in M_1M_2 become final which have any of their components as a final state of M_2. Figure 7 shows the function Star.

```
Function Star (M(Q, Σ, δ, q_0, F))
begin
length ← cardinality(Q);
rename the states of Q as q_i and store corresponding δ for 0 ≤ i ≤ length–1;
Σ° ← addlast(Σ, '#');
is ← 0, k, current, i_star, t_star, j_star, k_temp, j_temp, w, clos : integer;
P, S, F', Q : array[variable length] of strings;
q, k_star : string;
a : char;
a ← first element of Σ°;
while (a ≠ '#')              [while loop creates states p_i for
                              transition(s) from q_0 back to q_0]
    begin
    if δ(q_0, a) = q_0 then
          begin
          is ← is+1; P[is] ← p_is-1; δ'(q_0, a) = p_is-1;
          end;
    a ← next element of Σ°;
    end;
∀ a ∈ Σ   begin
          if δ(q_0, a) = q_j, such that j ≠ 0 then
            begin
            δ'(q_0, a) ← q_j;
            ∀ 0 ≤ k ≤ cardinality(P)-1   begin
                                         δ'(p_k, a) ← q_j;
                                         end;
            end
          else
            begin
            if δ(q_0, a) = q_0 then
                begin
                ∀ 0 ≤ k ≤ cardinality(P)-1          begin
                                                    δ'(p_k, a) ← p_k;
                                                    end;
                end;
            end;
        end;        [this block creates self transition(s) at p_i's
                    and other transition(s) for q_0 and p_i's]
∀ a ∈ Σ   begin
          for current = 1 to cardinality(Q)-1
                    begin
                    δ'(q_current, a) ← δ(q_current, a);
                    end;
          end;
       [remaining transitions for states other than q_0 and p_i's]
append the states of P and Q, after renaming p_i's into q_j's where i
≥ 0 and j = cardinality(Q)+i and restore δ' for these qj's;
      [adds states to DFA if there is a transition from q_0 to q_0]
i_star, t_star, j_star ← 1; k_star ← q_0; S[j_star] ← q_0;
a ← first element of Σ°;
while (j_star > 0)
    begin
    if length(k_star) = 2          then
                                   begin
                                   q ← δ'(k_star, a);
                                   end
    else
                                   begin
                                   q ← δ'(substring(k_star, 1, 2), a);
                                   w ← 2;
                                   while (w ≤ length(k_star)-1)
                                     begin
                                     q ← append(q, δ'(substring(k_star,
                                     w+1, 2), a));
                                     w ← w+2;
                                     end;
                                   end;
    δ*(k_star, a) ← q;
    if q_i in q such that q_i ∈ F then
            begin
            if q_0 in q then
                    begin
                    remove q_0 from q;
                    q ← append(q, q_0);
                    end
            else
                    begin
                    q ← append(q, q_0);
 [append q_0 at the end of q if q contains a final state of F]
                    end;
            end;
    j_star ← j_star+1; δ*(k_star, a) ← q; S[j_star] ← q;
    rewrite q such that suffixes of states are in non-
    decreasing order;                  [q_0q_1q_3q_2 → q_0q_1q_2q_3]
    rewrite q such that it does not contain duplicate
    occurrence of states;              [q_0q_1q_1q_3 → q_0q_1q_3]
    δ*(k_star, a) ← q;
    S[j_star] ← q;
    for k_temp = 1 to j_star-1
            begin
            if S[k_temp] = S[j_star] then
                    begin
                    j_star ← j_star-1;
                    end;
            end;
 [does not allow duplicate state names to enter in array S]
    t_star ← t_star+1;
    a ← next element of Σ°;
    if a = '#' then
            begin
            a ← first element of Σ°;
            i_star ← i_star+1;
            for j_temp = i_star to j_star
                begin
                restore the strings in array S such that
                length(S[j_temp]) is in non-decreasing order;
                    [if S[1]=q_0q_2 & S[2]=q_1, they will be
                    restored as S[1]=q_1 & S[2]=q_0q_2]
                end;
            end
    else
            begin
            t_star ← t_star-1;
```

```
            end;
        k_star ← S[t_star];
        if t_star > j_star then
            begin
            exit;
            end;
end;                                    [of while (j_star > 0)]
F' ← {q0};
for k_temp = 1 to j_star
        begin
        Q[k_temp] ← S[k_temp];
        if q_i in S[k_temp] such that q_i ∈ F then
            begin
            F' ← F' ∪ S[k_temp];
            end;
        end;
rename the states of Q as t_clos and store corresponding δ_closure for
clos ≥ 0 and also rename the corresponding states in F';
return M ← (Q, Σ, δ_closure, t0, F');
end function;
```

Figure 7. Function Star obtaining kleene closure of a DFA.

Function **Star** finds the kleene closure of a DFA M using the following rule. To construct M^*, **Star** first checks if there is any transition from start state q_0, back to q_0, and if any such transition exists, then **Star** adds new state(s) equal to the number of such transitions. The nature *(in terms of transitions)* of these newly added states is same as that of the start state q_0. So now **Star** has an intermediate DFA M', which contains no transition from start state q_0, back to q_0. Now to construct M^*, **Star** first constructs that part of M' which does not contain any final state. For the remaining construction of M^*, whenever **Star** reaches to a final state p of M', it clubs this final state with the start state q of M' to get a combined state [pq] and also renames state p as the combined state [pq]. Now **Star** obtains the remaining states of M^* using the following rule. If a transition enters into a combined state [lg] for some alphabet symbol 'a', which contains any of its components states as a final state of M', **Star** clubs this combined state [lg] with the start state q of M' to get a combined state [lgq] *(new or already existing)*, and that transition, for the same alphabet symbol 'a', now enters into this combined state. Then **Star** constructs the remaining part of M^* using the same logic as was applied in function **Union** for combined states. All those combined states in M^* become final which have any of their component as a final state of M'. Finally in M^*, **Star** converts the start state q_0 also to a final state. Figure 8 shows the function Min.

```
Function Min (M(Q, Σ, δ, q0, F))
begin
Σ° ← addlast(Σ, '#');
k_min, temp, temp1, temp2 : integer;
S, F' : array[variable length] of strings;
find the equivalent states in set Q and store them as combined
states;
store all the states (combined as well as single) of Q in array S
such that all the component states of S[k_min] are equivalent for k
≥ 1, and set S[1] such that it contains q0 as one of its component;
a ← first element of Σ°;
while (a ≠ '#')
    begin
    for temp = 1 to cardinality(S)
        begin
        δ_min(S[temp], a) = S[temp1] such that δ(q_i, a) = q_j such
        that q_i in S[temp] and q_j in S[temp1] for some q_i, q_j;
        end;
    a ← next element of Σ°;
    end;
F' ← ϕ;
for temp = 1 to cardinality(S)
    begin
    if q_i in S[temp] such that q_i ∈ F then
        begin
        F' ← F' ∪ S[temp];
        end;
    end;
rename the states of S as m_temp2 and store corresponding δ_min for
0 ≤ temp2 ≤ cardinality(S)-1 and also rename the corresponding
states in F';
return M ← (S, Σ, δ_min, m0, F');
end function;
```

Figure 8. Function Min obtaining minimal DFA equivalent to the given DFA.

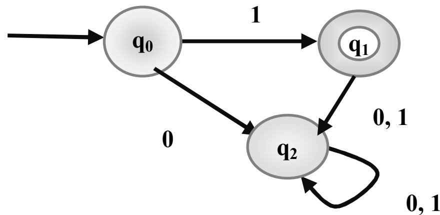
Function **Min** converts a DFA M into an equivalent DFA with minimum number of states. **Min** first finds the equivalent states of M, and then clubs those equivalent states to make some combined states (Hopcroft and Ullman, [19]). All those states which are not equivalent to any other state do not form any combined state and hence, will be written as single states. Therefore now **Min** has, possibly, some combined states and some single states for minimal DFA M_{min}. The start state of the resultant DFA M_{min} is either the start state q_0 of M *(if q_0 is not equivalent to any other state)* or a combined state containing q_0 as one of its component. Now, **Min** connects the start state of M_{min} to some other state *(combined or single)* by means of transitions for all alphabet symbols, using the transitions of M. **Min** continues for connecting the other states also by means of transitions for all alphabet symbols. Finally, all those states *(combined or single)* in M_{min} become final if they have anything common with the set of final states F of M.

3.1 Algorithm Implementation

A detailed demonstration on how the algorithm ***Construct*** works is shown as successive instances of the algorithm's implementation using two different representations: firstly by graphs and secondly by tables.

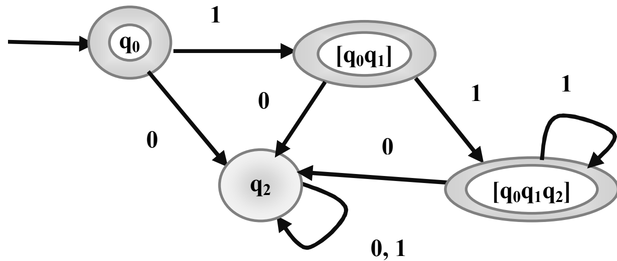
3.1.1 Using Graphs

To convert a randomly chosen regular expression 01^* into a minimal DFA M, ***Construct*** first places 01^* in a set of parenthesis; therefore, the input will look like (01^*). Then ***Construct*** scans for the innermost parenthesis and since, in this case only one parenthesis exists, the same is the innermost. Thereby, 1^* is the substring of regular expression for which ***Construct*** draws an initial DFA L_1. ***Construct*** constructs L_1 using the function Min(Star(Symbol(1))) and the successive steps of this construction are as shown by Figures 9 - 11.



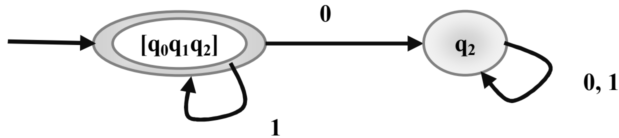
$K_1 = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\})$

Figure 9. DFA K_1 designed using Symbol(1).



$K_2 = (\{q_0, [q_0q_1], [q_0q_1q_2], q_2\}, \{0, 1\}, \delta, q_0, \{q_0, [q_0q_1], [q_0q_1q_2]\})$

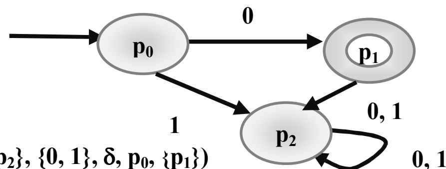
Figure 10. DFA K_2 designed using Star(Symbol(1)).



$K_3 = (\{[q_0q_1q_2], q_2\}, \{0, 1\}, \delta, [q_0q_1q_2], \{[q_0q_1q_2]\})$

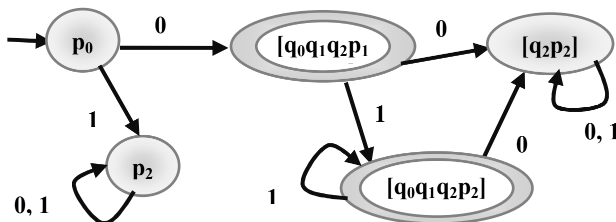
Figure 11. DFA K_3 designed using Min(Star(Symbol(1))).

Construct still runs in the same parenthesis and finds a DFA N_1 equivalent to $0L_1$ (value of K_3 is stored in L_1) using the following sequential construction steps (Figure 12 - 13).



$N_1 = (\{p_0, p_1, p_2\}, \{0, 1\}, \delta, p_0, \{p_1\})$

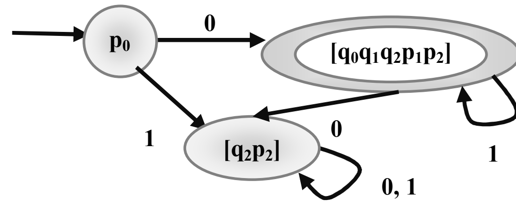
Figure 12. DFA N_1 designed using Symbol(0).



$N_1 = (\{p_0, p_2, [q_2p_2], [q_0q_1q_2p_1], [q_0q_1q_2p_2]\}, \{0, 1\}, \delta, p_0, \{[q_0q_1q_2p_1], [q_0q_1q_2p_2]\})$

Figure 13. DFA N_1 designed for N_1L_1.

Finally, ***Construct*** minimizes the obtained DFA N_1 to get minimal DFA N_1 as shown in Figure 14.



$N_1 = (\{p_0, [q_2p_2], [q_0q_1q_2p_1p_2]\}, \{0, 1\}, \delta, p_0, \{[q_0q_1q_2p_1p_2]\})$

Figure 14. DFA N_1 designed for its minimal version.

Therefore, the regular expression string looks like (N_1). Then ***Construct*** stores N_1 as M_1 and the string will look like (M_1). Finally, ***Construct*** replaces (M_1) by M_1, which is the minimal DFA for the given regular expression.

In the next section, we'll present the successive instances of our algorithm's implementation over another random regular expression using table representation.

3.1.2 Using Tables

In this section, the algorithm ***Construct*** is applied for converting a randomly chosen regular expression ((0+1(01*+0*)*1+1)*) into a minimal DFA. Here n = 20, so ***Construct*** initializes an array X of size 22 *(20+2)*. Next, ***it*** places the regular expression into X from 2nd to 21st place and also assigns X[1] = '(' and X[22] = ')' as shown in table 1. The successive instances of the algorithm's implementation are shown by the following sequence of stages of X in table 1.

Table 1. Array X showing the successive instances of conversion process of ((0+1(01*+0*)*1+1)*) into DFA M_4.

(	(	(	0	+	1	(	0	1	*	+	0	*	)	*	1	+	1	)	*	)	)
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

The innermost parenthesis is from 7th to 14th positions, and inside this parenthesis first star is at 10th position, and the character at (10-1) i.e. 9th position is a 1. So ***Construct*** replaces 9th position by L_1 (the min. DFA for 1^*), shifts all the cell's content from 11th position onwards to 1 position left, and finally replaces the 22nd position by #; next stage of X is

(	(	(	0	+	1	(	0	L_1	+	0	*	)	*	1	+	1	)	*	)	)	#
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

Still in the same innermost parenthesis, which is now from 7th to 13th positions, one more * exists at position 12. So ***Construct*** replaces 11th position by L_2 (the min. DFA for 0^*), shifts all the cell's content from 13th position onwards to 1 position left, and finally replaces the 22nd position by #; next stage of X is

(	(	(	0	+	1	(	0	L_1	+	L_2	)	*	1	+	1	)	*	)	)	#	#
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

Construct still runs in the same innermost parenthesis, which is now from 7th to 12th position. ***It*** reads the character at 8th position and replaces it by N_1 (the min. DFA for 0); next stage of X is

ARRAY X (Contd.)

(	(	(	0	+	1	(	N_1	L_1	+	L_2	)	*	1	+	1	)	*	)	)	#	#
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

In the same innermost parenthesis, i.e. from 7th to 12th positions, ***Construct*** finds a concatenation between 8th and 9th position. So ***it*** replaces the 8th position by N_1 (the min. DFA of concatenation of previous value stored in N_1 by L_1), shifts all the cell's content from 10th position onwards to 1 position left, and finally replaces the 22nd position by #; next stage of X is

(	(	(	0	+	1	(	N_1	+	L_2	)	*	1	+	1	)	*	)	)	#	#	#
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

In the same innermost parenthesis, i.e. from 7th to 11th positions, ***Construct*** reads a + followed by L_2, which is at 10th position. So ***it*** replaces 10th position by N_2; next stage of X is

(	(	(	0	+	1	(	N_1	+	N_2	)	*	1	+	1	)	*	)	)	#	#	#
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

All the possible concatenation operations are performed inside the innermost parenthesis, so now ***Construct*** checks the innermost parenthesis for the union operation. In the same innermost parenthesis, i.e. from 7th to 11th positions, ***it*** reads the character at 8th position and replaces it by M_1 (the DFA N_1 is stored as name M_1); next stage of X is

(	(	(	0	+	1	(	M_1	+	N_2	)	*	1	+	1	)	*	)	)	#	#	#
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

In the same innermost parenthesis, i.e. from 7th to 11th positions, ***Construct*** finds a union between 8th and 10th position. So ***it*** replaces the 8th position by M_1 (the min. DFA of union of previous value stored in M_1 by N_2), shifts all the cell's content from 11th position onwards to 2 positions left, and finally replaces each of 21st and 22nd positions by #; next stage of X is

(	(	(	0	+	1	(	M_1	)	*	1	+	1	)	*	)	)	#	#	#	#	#
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

In the same innermost parenthesis, i.e. from 7th to 9th positions, ***Construct*** reads a ')' at 9th position. So ***it*** removes that innermost parenthesis, by first shifting 8th position onwards to 1 position left, and then shifting 9th position onwards to 1 position left, and each time replacing the 22nd position by #; next stage of X is

(	(	(	0	+	1	M_1	*	1	+	1	)	*	)	)	#	#	#	#	#	#	#
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

So now ***Construct*** has a new innermost parenthesis from 3rd to 12th positions. ***Construct*** scans this parenthesis for a *, which appears at 8th position. ***It*** replaces the contents of 7th position by L_1 (the min DFA for M_1^*), shifts all the cell's content from 9th position onwards to 1 position left, and finally replaces the 22nd position by #; next stage of X is

(	(	(	0	+	1	L_1	1	+	1	)	*	)	)	#	#	#	#	#	#	#	#
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

Construct still runs in the same innermost parenthesis, which is now from 3rd to 11th position. ***It*** reads the character at 4th position and replaces it by N_1 (the min. DFA for 0); next stage of X is

ARRAY X (Contd.)

(	(	(	N_1	+	1	L_1	1	+	1	)	*	)	)	#	#	#	#	#	#	#	#
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

In the same innermost parenthesis, i.e. from 3rd to 11th positions, ***Construct*** reads a + followed by a 1, which is at 6th position. So ***it*** replaces 6th position by N_2; next stage of X is

(	(	(	N_1	+	N_2	L_1	1	+	1	)	*	)	)	#	#	#	#	#	#	#	#
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

In the same innermost parenthesis, i.e. from 3rd to 11th positions, ***Construct*** finds a concatenation between 6th and 7th position. So ***it*** replaces the 6th position by N_2 (the min. DFA of concatenation of previous value stored in N_2 by L_1), shifts all the cell's content from 8th position onwards to 1 position left, and finally replaces the 22nd position by #; next stage of X is

(	(	(	N_1	+	N_2	1	+	1	)	*	)	)	#	#	#	#	#	#	#	#	#
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

In the same innermost parenthesis, i.e. from 3rd to 10th positions, ***Construct*** finds a concatenation between 6th and 7th position. So ***it*** replaces the 6th position by N_2 (the min. DFA of concatenation of previous value stored in N_2 by the DFA for symbol 1), shifts all the cell's content from 8th position onwards to 1 position left, and finally replaces the 22nd position by #; next stage of X is

(	(	(	N_1	+	N_2	+	1	)	*	)	)	#	#	#	#	#	#	#	#	#	#
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

Construct still runs in the same innermost parenthesis, which is now from 3rd to 9th position. ***It*** reads the character at 8th position and replaces it by N_3 (the min. DFA for 1); next stage of X is

(	(	(	N_1	+	N_2	+	N_3	)	*	)	)	#	#	#	#	#	#	#	#	#	#
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

All the possible concatenation operations are performed inside the innermost parenthesis, so now ***Construct*** checks the innermost parenthesis for the union operation. In the same innermost parenthesis, i.e. from 3rd to 9th positions, ***it*** reads the character at 4th position and replaces it by M_2 (the DFA N_1 is stored as name M_2); next stage of X is

(	(	(	M_2	+	N_2	+	N_3	)	*	)	)	#	#	#	#	#	#	#	#	#	#
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

In the same innermost parenthesis, i.e. from 3rd to 9th positions, ***Construct*** finds a union between 4th and 6th positions. So ***it*** replaces the 4th position by M_2 (the min. DFA of union of previous value stored in M_2 by N_2), shifts all the cell's content from 7th position onwards to 2 positions left, and finally replaces each of the 21st and 22nd positions by #; next stage of X is

(	(	(	M_2	+	N_3	)	*	)	)	#	#	#	#	#	#	#	#	#	#	#	#
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

In the same innermost parenthesis, i.e. from 3rd to 7th positions, ***Construct*** finds a union between 4th and 6th positions. So ***it*** replaces the 4th position by M_2 (the min. DFA of union of previous value stored in M_2 by N_3), shifts all the cell's content from 7th position onwards to 2 positions left, and finally replaces each of the 21st and 22nd positions by #; next stage of X is

ARRAY X (Contd.)

(	(	(	M_2	)	*	)	)	#	#	#	#	#	#	#	#	#	#	#	#	#	#
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**

In the same innermost parenthesis, i.e. from 3rd to 5th positions, ***Construct*** reads a ')' at 5th position, so ***it*** removes that innermost parenthesis, by first shifting 4th position onwards to 1 position left, and then shifting 5th position onwards to 1 position left, and replacing each time 22nd position by #; next stage of X is

(	(	M_2	*	)	)	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**

So now a new innermost parenthesis exists from 2nd to 5th positions. ***Construct*** scans this parenthesis for a *, which appears at 4th position. ***It*** replaces the contents of 3rd position by L_1 (the min DFA for M_2^*), shifts all the cell's content from 5th position onwards to 1 position left, and finally replaces the 22nd position by #; next stage of X is

(	(	L_1	)	)	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**

Construct still runs in the same innermost parenthesis, which is now from 2nd to 4th position. ***It*** reads the character at 3rd position and replaces it by N_1 (the DFA L_1 is stored as name N_1); next stage of X is

(	(	N_1	)	)	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**

In the absence of any possible concatenation operation, ***Construct*** checks the innermost parenthesis for the union operation. In the same innermost parenthesis, i.e. from 2nd to 4th positions, ***Construct*** reads the character at 3rd position and replaces it by M_3 (the DFA N_1 is stored as name M_3); next stage of X is

(	(	M_3	)	)	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**

In the same innermost parenthesis, i.e. from 2nd to 4th positions, ***Construct*** reads a ')' at 4th position, so ***it*** removes that innermost parenthesis, by first shifting 3rd position onwards to 1 position left, and then shifting 4th position onwards to 1 position left, and replacing each time 22nd position by #; next stage of X is

(	M_3	)	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**

Now the innermost parenthesis is from 1st to 3rd position. The innermost string has no *, so no replacement is possible by an L_i. Next, ***Construct*** scans the innermost parenthesis for concatenation operation. ***It*** reads the character at 2nd position and replaces it by N_1 (the DFA M_3 is stored as name N_1); next stage of X is

(	N_1	)	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**

In the absence of any possible concatenation operation, ***Construct*** checks the innermost parenthesis for the union operation. In the same innermost parenthesis, i.e. from 1st to 3rd positions, ***it*** reads the character at 2nd position and replaces it by M_4 (the DFA N_1 is stored as name M_4); next stage of X is

ARRAY X (Contd.)

(	M_4	)	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**

In the same innermost parenthesis, i.e. from 1st to 3rd positions, ***Construct*** reads a ')' at 3rd position, so ***it*** removes that innermost parenthesis, by first shifting 2nd position onwards to 1 position left, and then shifting 3rd position onwards to 1 position left, and replacing each time 22nd position by #; next stage of X is

M_4	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**

Now, ***Construct*** does not find any innermost parenthesis and thus shows the final output as M_4.

3.2 Evaluation of Algorithm

In this section, we obtain the time complexity of the proposed algorithm ***Construct***. Because of the varied nature of regular expressions (*variable positions of left parenthesis, right parenthesis, *, +, and concatenation operator*), we obtain the average-case time complexity by applying the algorithm over a lot of different-sized regular expressions. First of all, the algorithm ***Construct*** is applied over a regular expression $(011)^*+10^*+(1101^*0)^*$ of size 20 and the time taken for this conversion is obtained as shown below.

Here n = 20, R = $(011)^*+10^*+(1101^*0)^*$.
Therefore array X is $((011)^*+10^*+(1101^*0)^*)$.
The transitions needed to obtain innermost parenthesis=5+4=**9**.
The transitions needed in search of a '*' operator inside the innermost parenthesis=**4**.
The transitions needed in search of a concatenation operator inside the innermost parenthesis=**4**.
The transitions needed in search of a '+' operator inside the innermost parenthesis=**2**.
Therefore array X is $(M_1^*+10^*+(1101^*0)^*)$.
Continued from the same point after replaced M_1, the transitions needed to obtain the next innermost parenthesis=14+7=**21**.
The transitions needed in search of a '*' operator inside the innermost parenthesis=**7**.
The transitions needed in search of a concatenation operator inside the innermost parenthesis=**6**.
The transitions needed in search of a '+' operator inside the innermost parenthesis=**2**.
Therefore array X is $(M_1^*+10^*+M_2^*)$.
Continued from the same point after replaced M_2, the transitions needed to obtain the next innermost parenthesis=2+10=**12**.
The transitions needed in search of a '*' operator inside the innermost parenthesis=**10**.
The transitions needed in search of a concatenation operator inside the innermost parenthesis=**7**.
The transitions needed in search of a '+' operator inside the innermost parenthesis=**6**.
Therefore array X now contains the resultant DFA M_3.
Hence, the total number of transitions required for converting R into DFA M_3 = 9+4+4+2+21+7+6+2+12+10+7+6 = **90**.

Next, the algorithm ***Construct*** is applied over 150 regular expressions of 15 different sizes n, and for this, 10 different and random regular expressions are taken of each size n. Then, the average time taken in the conversion for each value of n is obtained and is shown in table 2.

Table 2. Comparison Table between n, $n.log_e n$, $n.log_2 n$ and time taken by proposed algorithm.

n	"n"	$n.log_e n$	$n.log_2 n$	n^2	Average Time taken by proposed algorithm
1	1	0	0	1	**1**
5	5	8.05	11.61	25	**26.2**
10	10	23.03	33.22	100	**46**
15	15	40.62	58.60	225	**65.8**
20	20	59.92	86.44	400	**92.4**
25	25	80.47	116.10	625	**110.6**
30	30	102.04	147.21	900	**127.4**
35	35	124.44	179.53	1225	**148.4**
40	40	147.56	212.88	1600	**167**
45	45	171.30	247.13	2025	**190**
50	50	195.60	282.19	2500	**221**
75	75	323.81	467.16	5625	**318.6**
100	100	460.52	664.39	10000	**426.4**
150	150	751.60	1084.32	22500	**608.2**
200	200	1059.66	1528.77	40000	**804**

As shown in table 2, the proposed algorithm takes a little more time than $n.log_2 n$ for $1 \leq n \leq 10$; it coincides with the time $n.log_2 n$ for $10 \leq n \leq 20$; and then it becomes better by taking less time than $n.log_2 n$ for $n > 20$. In addition, the algorithm's time complexity becomes better than $n.log_e n$ when $n \geq 75$. Hence, the proposed algorithm takes $O(n.log_2 n)$ time. Besides, for larger values of n ($n \geq 75$) the proposed algorithm becomes more time-efficient and shows a time complexity of $O(n.log_e n)$ as shown in figure 15.

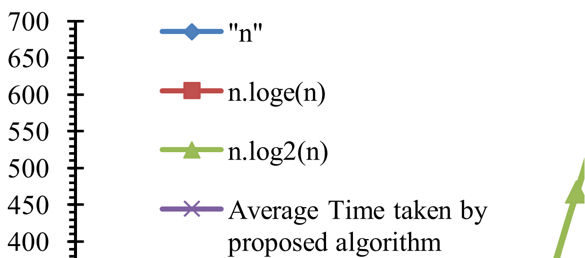


Figure 15. Comparison Graph between n, $n.log_e n$, $n.log_2 n$ and time taken by proposed algorithm.

4. CONCLUSION AND FUTURE WORK

From the basic results of formal language parsing, graph grammars and automata theory we have derived a simple novel method to construct a minimal deterministic finite automaton from a regular expression. We have applied our method to a lot of regular expressions and obtained the desired and exact results all the times showing thereby the applicability of our method for getting the minimal DFA from a regular expression. This method removes the dependency over the necessity of lengthy chain of conversion, that is, regular expression → NFA with ε-transitions → NFA without ε-transitions → DFA → minimal DFA. Therefore the main advantages of our minimal DFA construction algorithm are its minimal intermediate memory requirements and hence, it's reduced time complexity. This algorithm converts a regular expression of size n in to its minimal equivalent DFA in $O(n.log_2 n)$ time. In addition to this, the time complexity is further shortened to $O(n.log_e n)$ for $n \geq 75$.

Glushkov [14] presented a method to convert a regular expression into an ε-free NFA with $O(n^2)$ transitions while Hagenah and Muscholl [17] performed the same conversion with $O(n.log^2(n))$ transitions. Later, Hromkovic *et al.* [20] gave a method for the above conversion where the time complexity was $O(n.log_2 n)$ which was the least amongst all the above methods. However, all the above methods were inadequate as they converted a regular expression into only an ε-free NFA and not into minimal DFA. Therefore, additional time was required to convert ε-free NFA into minimal DFA. Hence, the overall time complexity for the required conversion was more than $O(n.log_2 n)$. However, our method converted a regular expression into a minimal DFA directly in $O(n.log_2 n)$ time hence, showing the supremacy of our method over the above methods.

Furthermore, Rytter [28] presented a method for converting a regular expression of size n into an NFA in *log n* time using (*n/log n*) parallel processors. As compared to Rytter's method [28], our method converted a regular expression of size n into a DFA in $O(n.log_2 n)$ steps using a high-speed processor. Thereby, the number of processors was reduced to 1 by our method. However, the time complexities were not comparable as the two methods produced different outputs; Rytter's method [28] produced NFA while our method produced minimal DFA. Hence, results by our algorithm are an improvement over Rytter's method [28].

Most researches attempted hitherto are based on the use of intermediate NFA for the above conversion. However, the present algorithm uses intermediate DFA in place of NFA, and still shows a time complexity which is shorter as compared to other available methods, thus motivating the use of DFA in place of NFA for similar studies. In addition to this, the above algorithm also inspires a further study for producing a more time-efficient algorithm for the above conversion.

5. REFERENCES

[1] Antimirov, V. [1996]. "Partial derivatives of regular expressions and finite automata constructions". *Theoretical Computer Science*. vol. 155, no. 2, pp. 291-319.

[2] Ben-David, S., D. Fisman, and S. Ruah [2008]. "Embedding finite automata within regular expressions". *Theoretical Computer Science*. vol. 404, no. 3, pp. 202-218.

[3] Berry, G. and R. Sethi [1986]. "From regular expressions to deterministic automata". *Theoretical Computer Science*. vol. 48, no. 1, pp. 117-126.

[4] Berstel, J., D. Perrin, and C. Reutenauer [2009]. *Codes and Automata*. *Encyclopedia of Mathematics and its*

Applications no. 129. Cambridge University Press. Cambridge.

[5] Bruggemann-Klein A. [1993]. "Regular expressions into finite automata". *Theoretical Computer Science.* vol. 120, no. 2, pp. 197-213.

[6] Brzozowski, J. A. and R. Cohen [1969]. "On decompositions of regular events". *Journal of the ACM (J. ACM)*. vol. 16, no. 1, pp. 132-144.

[7] Carrasco, R. C., J. Daciuk, and M. L. Forcada [2009]. "Incremental construction of minimal tree automata". *Algorithmica.* vol. 55, no. 1, pp. 95-110.

[8] Carrasco, R. C. and M. L. Forcada [2001]. "Incremental construction and maintenance of minimal finite-state automata". *Computational Linguistics*. vol. 28, no. 2, pp. 207-216.

[9] Chang, C. H. and R. Paige [1992]. "From regular expressions to DFAs using compressed NFAs". In *Proceedings of the 3rd Annual Symposium on Combinatorial Pattern Matching*. Lecture notes in Computer Science no. 644. Springer-Verlag, London. pp. 90-110.

[10] Cohen, D. I. A. [1991]. *Introduction to Computer Theory*. 2nd edn. John Wiley & Sons, Inc. New York.

[11] Daciuk, J., S. Mihov, B. W. Watson, and R. E. Watson [2000]. "Incremental construction of minimal acyclic finite-state automata". *Computational Linguistics*. vol. 26, no. 1, pp. 3-16.

[12] Geffert, V. [2003]. "Translation of binary regular expressions into nondeterministic ε-free automata with o(nlogn) transitions". *Journal of Computer and System Sciences.* vol. 66, no. 3, pp. 451-472.

[13] Ginzburg, A. [1968]. *Algebraic Theory of Automata*. Academic Press. New York.

[14] Glushkov, V. M. [1961]. "The abstract theory of automata". *Uspekhi Mathematicheskikh Nauk (UMN)*. vol. 16, no. 5(101), pp. 3-62.

[15] Greenlaw, R. and H. Hoover [1998]. *Fundamentals of the Theory of Computation: Principles and Practice.* Morgan Kaufmann Publishers, Inc. Elsevier, San Francisco, USA.

[16] Gurari, E. [1989]. *An Introduction to the Theory of Computation*. Computer Science Press. Ohio State University, Columbus, Ohio.

[17] Hagenah, C. and A. Muscholl [1998]. "Computing epsilon-free NFA from regular expressions in $o(n.\log^2(n))$ time". In *Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science*. Lecture Notes in Computer Science no. 1450. Springer-Verlag, London. pp. 277-285.

[18] Hein, J. L. [1996]. *Theory of Computation.* Jones & Bartlett Publishers, Inc. Sudbury, MA.

[19] Hopcroft, J. E. and J. Ullman [1979]. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley Longman Publishing Company, Inc. Boston, MA, USA.

[20] Hromkovic J., S. Seibert, and T. Wilke [2001]. "Translating regular expressions into small ε-free nondeterministic finite automata". *Journal of Computer and System Sciences.* vol. 62, no. 4, pp. 565-588.

[21] Ilie L. and S. Yu [2003]. "Follow automata". *Information and Computation.* vol. 186, no. 1, pp. 140-162.

[22] Johnson, W. L., J. H. Porter, S. I. Ackley, and D. T. Ross [1968]. "Automatic generation of efficient lexical processors using finite state techniques". *Communications of the ACM.* vol. 11, no. 12, pp. 805-813.

[23] Leiss, E. [1980]. "Constructing a finite automaton for a given regular expression". *ACM Special Interest Group on Algorithms and Computation Theory (ACM SIGACT News).* vol. 12, no. 3, pp. 81-87.

[24] Lewis, H. R. and C. H. Papadimitriou [2001]. *Elements of the Theory of Computation*. 2nd edn. Pearson Education Asia. Delhi.

[25] Martin, J. [2004]. *Introduction to Languages and the Theory of Computation.* 3rd edn. Tata McGraw Hill. New Delhi.

[26] Mayr, Ernst W., G. Schmidt, and G. Tinhofer (eds.) [1995]. *Graph-Theoretic Concepts in Computer Science.* Lecture notes in Computer Science no. 903. Springer-Verlag, Berlin/Heidelberg, New York.

[27] Möhring, R. H. (ed.) [1991]. Graph-Theoretic Concepts in Computer Science, 16th International Workshop, WG '90, Berlin, Germany, June 20-22, 1990, Proceedings. Lecture Notes in Computer Science no. 484. Springer. London, UK.

[28] Rytter, W. [1989]. "A note on optimal parallel transformations of regular expressions to nondeterministic finite automata". *Information Processing Letters.* vol. 31, no. 2, pp. 103-109.

[29] Singh, A. [2009]. *Elements of Computation Theory.* Springer-Verlag. London.

[30] Sipser, M. [2006]. *Introduction to the Theory of Computation.* 2nd edn. PWS Publishing.

[31] Stefano, C. R. [2009]. *Formal Languages and Compilation.* Springer-Verlag. London.

[32] Taylor, R. G. [1998]. *Models of Computation and Formal Languages.* Oxford University Press. New York.

[33] Thompson, K. [1968]. "Regular expression search algorithms". *Communications of the ACM.* vol. 11, no. 6, pp. 419-422.

[34] Watson, B. [1995]. "Taxonomies and toolkits of regular language algorithms". Ph.D. Thesis. Eindhoven University of Technology, CIP-DATA Koninklijke Bibliotheek, Den Haag.

[35] Wood, D. [1987]. *Theory of Computation: A Primer*. Addison-Wesley Longman Publishing Company, Inc. Boston, MA, USA.

[36] Yamamoto, H. [2005]. "New finite automata corresponding to semiextended regular expressions". *Systems and Computers in Japan.* vol. 36, no. 10, pp. 54-61.

[37] Ziadi, D. and J. M. Champarnaud [1999]. "An optimal parallel algorithm to convert a regular expression into its Glushkov automaton". *Laboratoire d'Informatique de Rouen*. vol. 215, no. 1-2, pp. 69-87.