The Application of Subsets Constructor in Nondeterminism

Eun Il Kim

Andre Wong

CSC 33500

12/1/16

Abstract

The nondeterministic problem, NFA (nondeterministic finite automaton), can be solved efficiently by an algorithm called subset algorithm. Although an NFA can be easily constructed, the NFA will become impractical if the machine produces multiple paths; as it will contain inconsistency and randomness. To efficiently solve a NFA, the machine must first be deterministic, specifically, a DFA (deterministic finite automaton). An algorithm that can convert a NFA to DFA would be the subset algorithm. The subset algorithm will be able to search for all possible subset of the NFA and build a new DFA. Through scheme, the subset algorithm will: construct the existing NFA, convert NFA to DFA, display a procedure required to derive the DFA. The process of the code will require several iterations, recursion, and procedures to output the correct corresponding DFA and show the effectiveness of the subset algorithm with a polynomial time complexity of O(n^3). Overall, the subset algorithm will assist in creating an operational interpreter that converts an inconsistent NFA to a uniform DFA.

The Application of Subsets Constructor in Nondeterminism

A program branches between multiple choices or paths and follow all paths to reach a desired conclusion; the reason is that program is nondeterministic. According to Robert W. Floyd's "Non-Deterministic Algorithms," nondeterminism in theory, is when the algorithm distributes different results despite having similar previous inputs in the algorithm itself. This causes inconsistencies as the results varies depending on a random factor such as sequence and timing (also known as the race condition). As a result of going through multiple paths simultaneously, the accepting paths of the nondeterministic algorithm will be different depending on the timing or series of events. As the sequence is similar and efficient between runs, a deterministic algorithm will always distribute the same results and is unaffected by the random factor. Since the theories between nondeterministic and deterministic algorithms are very similar, nondeterministic finite automata could be converted to deterministic finite automata. This can be achieved through the subset constructor algorithm.

Subset Constructor Algorithm

The subset algorithm gives an understanding of the computation that are harder to recognize as an NFA to a flexible executable DFA. The algorithm was also called the Rabin-Scott Powerset Construction which was published by Michael O. Rabin and Dana Scott in 1959. According to Rabin and Scott's "Finite Automata and Their Decision Problems," there are advantages and disadvantages of using a nondeterministic automata and why we convert them to deterministic. A nondeterministic automata shows that any set of tapes defined by such a machine could also be defined by an ordinary automata. This means that for every NFA exists a DFA in the same machine language. The advantages of these machines is the small number of

internal states needed for the many transitions, easing in which specific machines can be described. To be clear, a non-deterministic automata is not a probabilistic machine, but rather a machine with many choices in its transition. At each transition, it will choose one of several new internal states, in which some choices lead to either impossible situations with no possible movements or an accepting transition which will eventually lead to the final state. Though the structure is simple, having multiple states transition at the same time causes the build to be complex and harder to build than a deterministic automata.

A deterministic automata is uniquely determined by the table of moves, since there was one and only one way the machine would change its state in any particular situation. Since each transition has their own state, building the deterministic automata's path is simple, but having all the machines to be in this same form lead to difficulty in the creation of the construction due to the large numbers of internal states needed for even some relatively elementary operations. The large numbers of internal states represent all the possible subset of an NFA. Thus, the conversion process from an NFA to a DFA is called a subset constructor.

This is further supported by J.Howard Johnson and Derick Wood's "Instruction Computation in Subset Construction." As stated by Johnson and Wood, the subset construction computation is the method of converting a nondeterministic finite-state machine into a deterministic one of the same formal language. Converting an NFA to DFA is possible since every possible set of NFA states are the subsets of a single DFA state. Therefore, a DFA is a union of all NFA transition sets applied to every element. This is further explained within Figure 1, which are the fundamental steps of creating the subset algorithm.

Algorithm Subset Construction
On entry: A nondeterministic finite-state machine, $M = (Q, \Sigma, \delta, s, F)$.
On exit: A deterministic finite-state machine, $M' = (Q', \Sigma, \delta', s', F')$
that satisfies $L(M) = L(M')$.
begin
Let $Q_0 = \{s\}$ be the zeroth subset of Q and 0 be the
corresponding state in Q', let i be 0, and let $last$ be 0;
while $i \leq last$
do begin
for all a in Σ
do if $\{(p, a, q) : p \in Q_i\} \neq Q_j$, for some j, $0 \leq j \leq last$
then begin $last := last + 1$;
$\delta' := \delta' \cup \{(i, a, last)\}$;
$Q_{last} := \{q : p \in Q_i \text{ and } (p, a, q) \in \delta\}$
end;
$i := i + 1$
end {while};
$Q' := \{i : 0 \leq i \leq last\}$;
$F' := \{i : Q_i \cap F \neq \emptyset \text{ and } 0 \leq i \leq last\}$
end

Figure 1: An in-depth explanation of the subset algorithm by Derick Wood and J. Howard Johnson.

To sum it up, the nondeterministic finite automata (NFA) can access a plethora of states simultaneously that can transition through multiple states of the automata and accepts any of those states as an accepting state. Since NFA can accept any state as an accepting state, it is easier to develop the concept of that automata, but because there are too many states that NFA can transition to, developing the algorithm becomes much more complex (Figure 2).

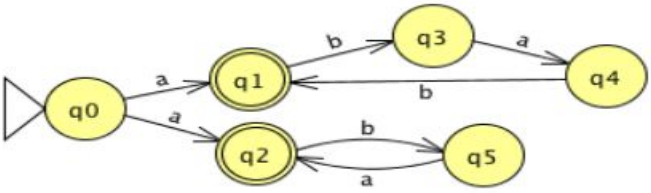

Figure 2: An example of a non-deterministic finite automaton.

The deterministic finite automata (DFA) follows restrictions unlike NFA: each input uniquely determines the resulting or source state. In other words, there is a single state for each input and can only transition after reading that input (Figure 3).

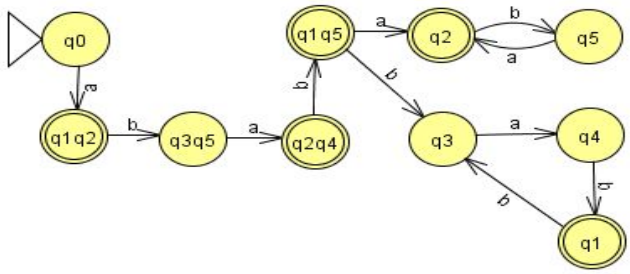

Figure 3: An example of a deterministic finite automaton.

Since each input is determined by one state, DFA can be much simpler to construct, but difficult to understand the algorithm. Overall, there are benefits that makes NFA comparatively better than DFA and vice versa. To address to this, we use the subset construction method which is applied to the scheme program.

Analyzing the Subset Scheme Code

In the DFA scheme code, there will be will 4 sections: the helper functions, simple procedures, complex procedures, and main function. The important part of this interpreter is understanding the general structure of a given input. In this case, it would be the structure of the NFA.

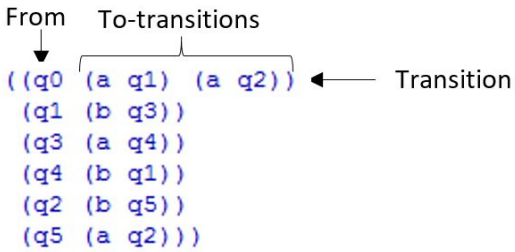

Figure 4: General NFA structure

General Structure of the FA

The general structure of the NFA, can be observed in Figure 4. The NFA as well as DFA, can be broken down into a group of sequences. This structure will always contain a list of transitions; the transition is made up of a *from-state*, a *to-transition*, and a *to-state*. Each *to-transition* is made up of an input (the condition required to go to that state) and the *to-state*.

```
(define (construct-dfa nfa) (display "\nDFA Computations\n")
   (define (iter-dfa dfa stack)
     (let ((current-state (get-test stack)))
       (map display (list dfa "\n"))
       (cond((null? current-state) dfa)
            (else (let* ((a-list (find-to-states nfa current-state 'a))
                         (b-list (find-to-states nfa current-state 'b))
                         (new-stack (update-stack dfa stack a-list b-list))
                         (new-dfa (list(append-dfa current-state a-list b-list))))
                    (if (null? dfa)
                        (iter-dfa new-dfa new-stack)
                        (iter-dfa (append dfa new-dfa) new-stack)))))))
  (iter-dfa '() (list (list (caar nfa)))))
```

Figure 5: Main function to convert DFA to NFA

Main function

The main function as seen in Figure 5, contains the construction code of the DFA called the (**construct-dfa**). This code contains the parameter: ‘nfa,’ ‘dfa,’ and ‘stack.’ The ‘nfa’ parameter will be the sequence that will be converted; the ‘dfa’ parameter will be the new deterministic sequence that will be built; and the ‘stack’ parameter will hold new possible sequences to be tested. In each run, the only values that should change in this code would be ‘dfa’ and ‘stack’ parameters. The ‘stack’ could either contain an item or be null. The general purpose of this code is to take an NFA structure and convert it to a DFA.

```
(define (get-test stack)
   (cond ((null? stack) '())
         (else (car stack))))

(define (pop stack)
  (cond ((null? stack)'())
        (else (cdr stack))))

(define (update-stack dfa stack a-list b-list)
  (let* ((current-stack (pop stack))
         (nonempty-stack? (not (null? current-stack)))
         (input-a? (not (exist-in-dfa? dfa a-list)))
         (input-b? (not (exist-in-dfa? dfa b-list))))
    (cond((and nonempty-stack? input-a? input-b?) (list current-stack a-list b-list))
         ((and input-a? input-b?) (list a-list  b-list))
         (nonempty-stack? (append current-stack
                          (if input-a? (list a-list) '())
                          (if input-b? (list b-list) '())))
         (else (list(append (if nonempty-stack? current-stack '())
          (if input-a? a-list '())
          (if input-b? b-list '()))))
        )))

(define (get-to-states nfa state input)
  (define (get-help mylist)
    (cond ((null? mylist)'())
          ((eq? (caar mylist) input) (cons (cadar mylist) (get-help (cdr mylist))))
          (else (get-help (cdr mylist)))))
  (get-help (find-to nfa state)))
```

Figure 6: Simple Procedure: Get transitions that are not null and push and update them into a temp list

Get-test

The procedure (**get-test**) will first determine whether or not there will be a state to test on; this state will be called ‘current-state.’ The ‘current-state’ is found at the ‘stack’, using the procedure (**get-test stack**) shown on Figure 6. The procedure, (**get-test)** will return the first list in the given ‘stack’. Otherwise, the procedure will return null if there is no more list within the ‘stack’ to test on. The value returned from (**get-test)** will be placed in ‘current-state’; which will be used on whether or not to halt the main function (**construct-dfa**) and return the ‘dfa’. If ‘current-state’ is not null, resume testing the ‘current-state’.

```
(define (find-to-states nfa current-state input)
    (cond ((null? current-state) '())
          (else (let ((temp-state (get-to-states nfa (car current-state) input)))
                  (cond ((null? temp-state) (find-to-states nfa (cdr current-state) input))
                        (else (append temp-state (find-to-states nfa (cdr current-state) input)))
                        )))))

(define (append-dfa current-state a-list b-list)
  (cond ((not (or (null? a-list) (null? b-list)))
         (append current-state (list (flatten(list 'a a-list))) (list (flatten(list 'b b-list)))))
        ((not (null? a-list))
         (append current-state (list (flatten(list 'a a-list)))))
        ((not (null? b-list))
         (append current-state (list (flatten(list 'b b-list)))))
        (else '())))
```

Figure 7: Find current state and append to transition list

Find-to-states

The (**construct-dfa**) will make several temporary variables, starting with the variables 'a-list' and 'b-list'. Both lists will be the newly formed *to-state*, which will be later added to the 'dfa.' Either 'a-list' or 'b-list' needs to use (**find-to-states**) to make the *to-state*. In Figure 7, the (**find-to-states**) will use 'nfa', 'current-state', and 'input' parameters. The procedure (**find-to-states**) will take each *to-state* (as 'current-state' may include a conjunction of *from-states*) from 'current-state' and test each *to-states* within (**get-to-states**); this entire procedure will be appended into the variable, 'temp-state'. The procedure (**get-to-states**), can be found in Figure 6, which will take 'nfa', 'state', and 'input'. The precondition in the (**get-to-states**) procedure is that 'state' must be an atom. The (**get-to-states**) procedure must first use the procedure (**find-to**), to find possible the *to-transition* structure within the 'nfa.' The (**find-to**) procedure which can be found in Figure 8, requires 'mylist' and 'state' parameters. The variable 'mylist' contains the 'nfa' while 'state' must be the atom *from-state*. The procedure (**find-to**) will match a single *from-state* in the NFA and return its *to-transition* structure. Then, the (**find-to**) will be utilized in the iterative code, (**get-help**), which is imbedded within the code

(**get-to-states**) in Figure 6. The (**get-help**) procedure will take the variable 'mylist' (which contains the *to-transition* structure) and append only the *to-state* that satisfies a given 'input' from the global variable in (**get-to-states**). The result will be an appended *to-state* for the given 'input' and 'state' and will be brought back to (**find-to-states**) in Figure 5, where 'temp-state' will hold the appended *to-state*s from one or more given *from-state* that was within 'current-state.' The (**find-to-states**) procedure will finally append the 'temp-state' to null and return it to 'a-list' or 'b-list' in (**construct-dfa**). The 'a-list' or 'b-list' can be null as the *to-transition* may not contain a given 'input', given in the 'current-state'. For example, ('q0 ('a 'q1) ('a 'q2)), in this case a-list will contain 'q1 and 'q2, but b-list will be null as there is no given input 'b.

```
(define (get-from mylist)
  (cond ((list? (car mylist)) '())
        (else (cons (car mylist) (get-from (cdr mylist))))))

(define (find-from? mylist state)
  (cond ((null? mylist) #f)
        ((list? state)
         (cond((equal? (get-from (car mylist)) state) #t)
              (else (find-from? (cdr mylist) state))))
        (else (cond((eq? (caar mylist) state) #t)
                   (else (find-from? (cdr mylist) state))))
        ))

(define (find-to mylist state)
  (cond ((null? mylist) '())
        ((eq? (caar mylist) state) (cdar mylist))
        (else (find-to (cdr mylist) state))))

(define (exist-in-dfa? dfa state)
  (cond((null? state) #t)
       (else (find-from? dfa state))))
```

Figure 8: Functions to find transitions between states

Update-stack

Now that both ‘a-list’ and ‘b-list’ has been updated, ‘new-stack’ must be updated in (**construct-dfa**). The ‘new-stack’ will call (**update-stack**) in Figure 6, which will push ‘a-list’ and ‘b-list’ into the ‘stack’ if it does not exist in *from-state* of the ‘dfa.’ The (**update-stack**) requires the ‘dfa’, ‘stack’, ‘a-list’, and ‘b-list’. First, the procedure will (**pop**) the ‘stack;’ then proceed to check if ‘a-list’ or ‘b-list’ exist in ‘stack’ by using the procedure (**exist-in-dfa?**). The procedure (**exist-in-dfa?**) will immediately call (**find-from?**) which both the procedures can be found in Figure 8. The (**find-from?**) procedure will use the ‘dfa’ in ‘my-list’, and a *from-state* in ‘state’. If ‘my-list’ is null, then either the ‘dfa’ was empty, so no ‘state’ can be found, or the ‘state’ still could not be found after cdr-ing ‘my-list’. If ‘my-list’ is not empty, then it must be distinguished whether the ‘state’ is an atom or a list of *from-state*. If ‘state’ is a list, then the (**get-from**) procedure found in Figure 8, will help compare to another list of *from-state* found in the ‘dfa.’ The (**get-from**) procedure will extract a list of *from-state* from a single list of transition in ‘my-list’ from the ‘dfa.’ The (**get-from**) procedure will cons each atom until it sees a list which is the start of the *to-transition*. If the list of *from-state* returned from (**get-from**) matches the list of *from-state* or ‘state’ in (**find-from?**) then #t will be returned, otherwise the procedure cdr down ‘my-list’ or the ‘dfa.’ Similarly, if the ‘state’ was an atom, it returns #t if it finds a matching *from-state* in ‘dfa’ or ‘my-list’ else it cdr’s down ‘my-list.’ The value from (**find-from?**) will then return into (**exist-in-dfa?**), which will return the value into ‘input-a?’ or ‘input-b?’ depending on whether it is a ‘a-list’ or ‘b-list’ from (**update-stack**). Although multiple conditions exist in (**update-stack**), the main resultant that will be returned will be the appended list of ‘current-stack,’ ‘a-list,’ and ‘b-list’. The idea of multiple conditions from the procedure is to exempt the null values and to return a list that contain values.

Append-dfa

Finally, a new ‘dfa’ needs to be appended in the (**construct-dfa**) procedure. The (**append-dfa**) procedure found in Figure 7, will require the *from-state* ‘current-state’ and the *to-transition* ‘a-list’ and ‘b-list’. The ‘a-list’ and ‘b-list’ can either be null, but one must exist, or it will return a null. The multiple conditions in (**append-dfa**) assesses how the list should be appended depending on whether or not one of the list is null.

Iteration of Construct-dfa

If the ‘dfa’ was null in the beginning of the (**construct-dfa**) procedure, it will run the ‘new-dfa’ and ‘new-stack’. Otherwise, the ‘new-dfa’ will be appended to the old ‘dfa’, and the appended ‘dfa’ will be sent along with ‘new-stack’ to the next iteration of (**construct-dfa**).

```
(define (new-accept-output dfa accept-states)
  (cond ((null? dfa) '())
        (else (let ((from-states (get-from (car dfa))))
                ;ormap returns the (or (map func list))
                (cond ((ormap (lambda (e) (member? e from-states)) accept-states)
                       (cons from-states (new-accept-output (cdr dfa) accept-states)))
                      (else (new-accept-output (cdr dfa) accept-states)))))))


(define nfa (list (list 'q0 (list 'a 'q1)(list 'a 'q2))
      (list 'q1 (list 'b 'q3))
      (list 'q3 (list 'a 'q4))
      (list 'q4 (list 'b 'q1))
      (list 'q2 (list 'b 'q5))
      (list 'q5 (list 'a 'q2))))
(define accept-states (list 'q1 'q2))
(display "NFA\n") nfa
(display "\nAccept State:") accept-states

(define dfa (construct-dfa nfa))
(define new-accept-states(new-accept-output dfa accept-states))
(display "\n\nDFA\n") dfa
(display "\nNew Accept States:") new-accept-states
```

Figure 9: NFA to DFA conversion

New-accept-output

After a DFA is built from an NFA, the next step is to recognize all states that are accepting states. The procedure in Figure 9 called the (**new-accept-output**), will output the new

accepting states. The (**new-accept-output**) will cdr down the 'dfa' and find all *from-state structures* that contains any of the atom within 'accept-states.' The (**get-from**) procedure must be called to extract each *from-state* from a transition in 'dfa' and place it within the variable, 'from-states'. An (**ormap**) [i.e. (or map func list)] of for-each 'accept-state' will be used to evaluate if the atom is indeed a (**member?**) of 'from-states'. If the atom of 'accept-state' is a (**member?**) of 'from-states,' the 'from-states' will be appended to the next call of (**new-accept-output**), otherwise, it will cdr down the 'dfa.'

Analyzing the NFA to DFA Conversion Output

```
NFA
((q0 (a q1) (a q2)) (q1 (b q3)) (q3 (a q4)) (q4 (b q1)) (q2 (b q5)) (q5 (a q2)))

Accept State:(q1 q2)

DFA Computations
()
((q0 (a q1 q2)))
((q0 (a q1 q2)) (q1 q2 (b q3 q5)))
((q0 (a q1 q2)) (q1 q2 (b q3 q5)) (q3 q5 (a q4 q2)))
((q0 (a q1 q2)) (q1 q2 (b q3 q5)) (q3 q5 (a q4 q2)) (q4 q2 (b q1 q5)))
((q0 (a q1 q2)) (q1 q2 (b q3 q5)) (q3 q5 (a q4 q2)) (q4 q2 (b q1 q5)) (q1 q5 (a q2) (b q3)))
((q0 (a q1 q2)) (q1 q2 (b q3 q5)) (q3 q5 (a q4 q2)) (q4 q2 (b q1 q5)) (q1 q5 (a q2) (b q3)) (q2 (b q5)))
((q0 (a q1 q2)) (q1 q2 (b q3 q5)) (q3 q5 (a q4 q2)) (q4 q2 (b q1 q5)) (q1 q5 (a q2) (b q3)) (q2 (b q5)) (q3 (a q4)))
((q0 (a q1 q2)) (q1 q2 (b q3 q5)) (q3 q5 (a q4 q2)) (q4 q2 (b q1 q5)) (q1 q5 (a q2) (b q3)) (q2 (b q5)) (q3 (a q4)) (q5 (a q2)))
((q0 (a q1 q2)) (q1 q2 (b q3 q5)) (q3 q5 (a q4 q2)) (q4 q2 (b q1 q5)) (q1 q5 (a q2) (b q3)) (q2 (b q5)) (q3 (a q4)) (q5 (a q2)) (q4 (b q1)))
((q0 (a q1 q2)) (q1 q2 (b q3 q5)) (q3 q5 (a q4 q2)) (q4 q2 (b q1 q5)) (q1 q5 (a q2) (b q3)) (q2 (b q5)) (q3 (a q4)) (q5 (a q2)) (q4 (b q1)) (q1 (b q3)))


DFA
((q0 (a q1 q2)) (q1 q2 (b q3 q5)) (q3 q5 (a q4 q2)) (q4 q2 (b q1 q5)) (q1 q5 (a q2) (b q3)) (q2 (b q5)) (q3 (a q4)) (q5 (a q2)) (q4 (b q1)) (q1 (b q3)))

New Accept States:((q1 q2) (q4 q2) (q1 q5) (q2) (q1))
```

Figure 10: NFA to DFA output

After outputting the new accepting states, the NFA to DFA conversion is complete. The code outputs the original NFA and its accepting states. During the process of conversion, the DFA computation is shown step by step until it reaches the resulting DFA and it's new accepting states. The new accepting states must contain the original accepting states. For example, NFA's accepting state contains q1 or q2, therefore DFA's accepting state can be in a different form, but must contain either q1 or q2.

Analyzing the Complexity of Subset

The DFA conversion scheme code has a polynomial time complexity of O(n^3). To find the time complexity of the code, the time complexity of each procedure must be observed. Most of the procedures are recursion or iterations, so it will result between O(n) or O(nlogn). Since the time complexity focuses on the worst-case complexity, each recursion will take O(n) or worse.

Procedures with Constant Complexity of O(1)

The (**get-test**) and (**pop-stack**) procedures will return O(n) because both procedures returns a single value.

Procedures with Linear Complexity of O(n)

The following procedures will return the time complexity of O(n). The (**flatten**), (**get-from**), (**find-to**) procedures are single recursions. The (**member?**) and (**ormap**) procedures are single iterative procedures. The (**get-to-states**) procedure will first use the value of (**find-to**) only once, then place the value into the (**get-help**)'s parameter. The (**get-help**) procedure will be recursive which will return O(n). The (**append-dfa**) will call multiple (**flatten**)s and append them.

Procedures with Polynomial Complexity of O(n^2)

The following procedures will return the time complexity of O(n^2). The (**find-from?**) procedure is an iterative calling (**get-from**). The (**exist-in-dfa?**) procedure will literally call (**find-from?**). The (**update-stack**) procedure calls (**exist-in-dfa?**) twice. The (**find-to-states**) procedure calls (**get-to-states**) on every recursion. The (**new-accept-output**) process will not only have a self-calling recursion, but will also call O(n) processes such as (**get-from**) or (**member?**).

Procedures with Polynomial Complexity of O(n^3)

The (**construct-dfa**) process will return O(n^3) because not only does it have a self-calling recursion, but it also calls O(n^2) processes such as (**find-to-states**) or (**update-stack**).

Conclusion

The subset construction is an effective algorithm that allows a nondeterministic finite automata (NFA) to become deterministic finite automata (DFA). Converting an NFA to a DFA returns a single path rather than simultaneously going through multiple paths. A single path replaces the inconsistent resultant that NFA with a consistent resultant of DFA. The subset construction algorithm chooses a state (given an initial state) and specify which possible state it traverse to given a path from a set of paths. This algorithm chooses possible paths until it reaches the accepting state (or the final state). However, if the path does not lead to the final state, then it will lead to a rejecting state.

In the scheme code, the NFA goes through many procedures by finding the transitions and state of the list, then translating each state and transition into all possible sets of the NFA, and finally inserting the new states and transitions for the newly built DFA with a time complexity of polynomial time of O(n^3).

There are several areas of the code that could have been improved, such as the use of Amb-Evaluator. Although, the Amb-Evaluator would return several paths that could either succeed or fail, it would not be practical as there are infinitely repeating sequences that could still lead to the correct path. The time complexity of the code could have also been improved, by enhancing the time complexity of the (**find-to-state**) or (**update-stack**) procedures.

Through the usage of functional program, it was possible to make a powerful interpreter that could inspect the structure of a list. An interpreter will prove to be useful when producing an algorithm and analyzing the structure of any collection of lists. Overall, it can be said that the subset algorithm was ultimately able to remove the unpredictability of nondeterminism by converting it into a deterministic result.

Citations:

Floyd, Robert W. “Non-Deterministic Algorithms.”*Repository.cmu.edu*, Carnegie Institute of Technology, Nov. 1966, http://repository.cmu.edu/cgi/viewcontent.cgi?article=2787&context=compsci.

Johnson, J Howard, and Derick Wood. “Instruction Computation in Subset Construction.”*CiteSeerX*, Natural Sciences and Engineering Research Council of Canada, from the Information Technology Research Centre of Ontario, and from the Research Grants Committee of Hong Kong., June 1996, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.49.9630&rep=rep1&type=pdf.

Rabin, Michael O, and Dana Scott. “Finite Automata and Their Decision Problem's.” *Finite Automata and Their Decision Problem's*, Apr. 1959, pp. 114–125. www.cse.chalmers.se/~coquand/AUTOMATA/rs.pdf.