# Design and Analysis of Algorithms

## NFA-DFA Conversion

|  |  |  |
|---|---|---|
| Shashank MG | - | PES1201700298 |
| Sparsha P | - | PES1201700226 |

## ● Abstract

It is important for converting easier-to-construct NFAs into more easily executable DFAs. We consider the number of states of a DFA that is equivalent to an n-state NFA accepting a finite language. It shows that O(2^n) is the (worst-case) optimal upper-bound on the number of states of a DFA that is equivalent to an n-state NFA accepting a finite language.
If a language is recognized by an NFA, then it is also accepted by equivalent DFA.

## ● Problem Definition

Finite state automata are abstract models of computation. In other words, they are systems that take input (in form of symbols) and produce a result (or, perform a computation). To do this, they read symbols and move internally between states.

There are two classes of Finite state automata: Deterministic automata (DFA) and Non-deterministic automata (NFA).

Finite state automata are described by < Q, ∑, q0, δ, F > where:

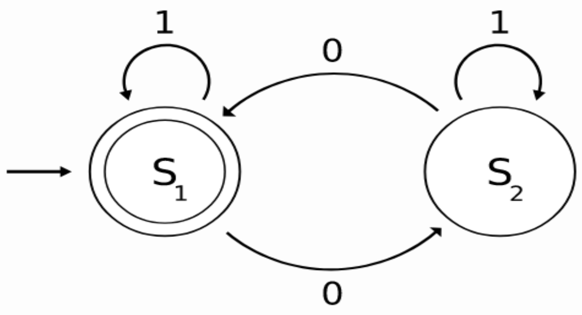
- q0: Start state
- Q: List of states
- δ: Transition functions between states
- F: Final states
- ∑: Input symbols

The main distinction between the two is based on their transitions and input symbols. A DFA is not allowed to have multiple transitions for a given input symbol where as a NFA is allowed to do so. There exists a special input symbol for NFA that is λ(lambda) which allows the state of a NFA to change without consumption of a input symbol.

The main objective of our project is to convert a given NFA to a DFA.

## ● Introduction

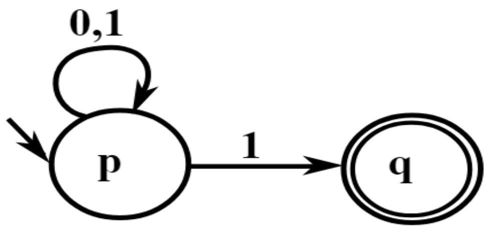
DFA stands for Deterministic Finite Automaton. Its job is to read in a string of symbols (input) and move (transition) between its states based on the input and ultimately accept or reject the string of symbols. The transitions are deterministic: for a given symbol string and a given DFA, the DFA will go through the same states every time the same symbol string is fed into it. In other words, the path between the states is guaranteed to be unique for a given symbol string. The way this is enforced is by specifying that for every state, for a given input symbol, there's only one transition. So this is allowed:



For example, in the figure shown above, if the DFA is in state S1 and the input symbol is 0, then on consuming 0 it goes into state S2 and if the input symbol is 1 it stays in S1. Similarly, for each state on a given input symbol there is only one transition possible.

NFA is non-deterministic finite automaton. In a NFA:

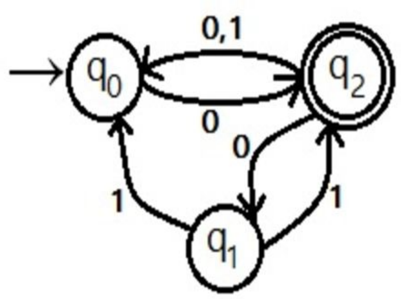
- In a state, for one input, the automaton can transition to more than one states
- Automaton can transition using empty input

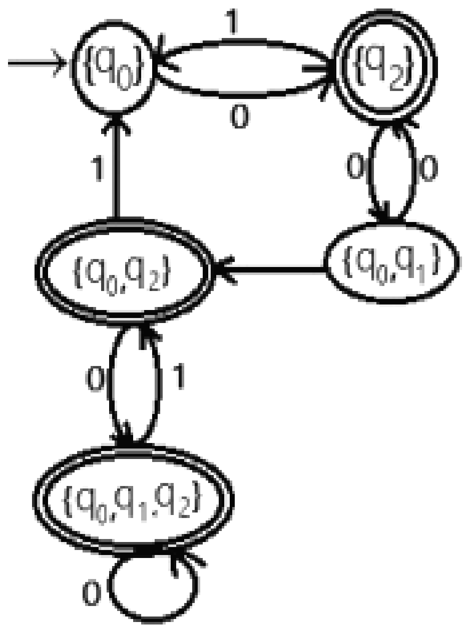


For example, if the NFA in the figure above is in state p, then if it reads in symbol 1, it could either stay in p or transition to q. It’s not deterministic (as the name implies).

NFAs are smaller than DFA, in terms of number of states. However, an NFA can always be converted into a DFA which will just have more states. This should be intuitive - if I am restricted to only one transition per symbol per state, then to encode the same behaviour as an NFA, we will need more states.

Example of a Conversion from NFA to a DFA.





The first image is a NFA which needs to be converted to a DFA. We can see the equivalent DFA in the second image. We observe in the NFA that for the state q2 on the symbol 0 we go to q1 and q0. In its equivalent DFA q2 on 0 goes to a state {q0,q1} which is a new state generated in the process of conversion and so on. We also see that the total number of states in the NFA is lesser than the number of states in its equivalent DFA.

- ## Design

There are several algorithms that allow you to convert a NFA to a DFA. One such is the epsilon closure.
An epsilon closure of a given state q is defined as the states which can be be reached from q without actually consuming any input symbol.
This can be regarded as a Transformation of the given input where we find the closure of all the states before hand and use it to simplify the algorithm further.

Algorithm in detail:
Step 1 : Take epsilon closure for the states of NFA. Closure of the start state of NFA will become state of DFA.

Step 2 : Find the states that can be traversed from the present for each input symbol

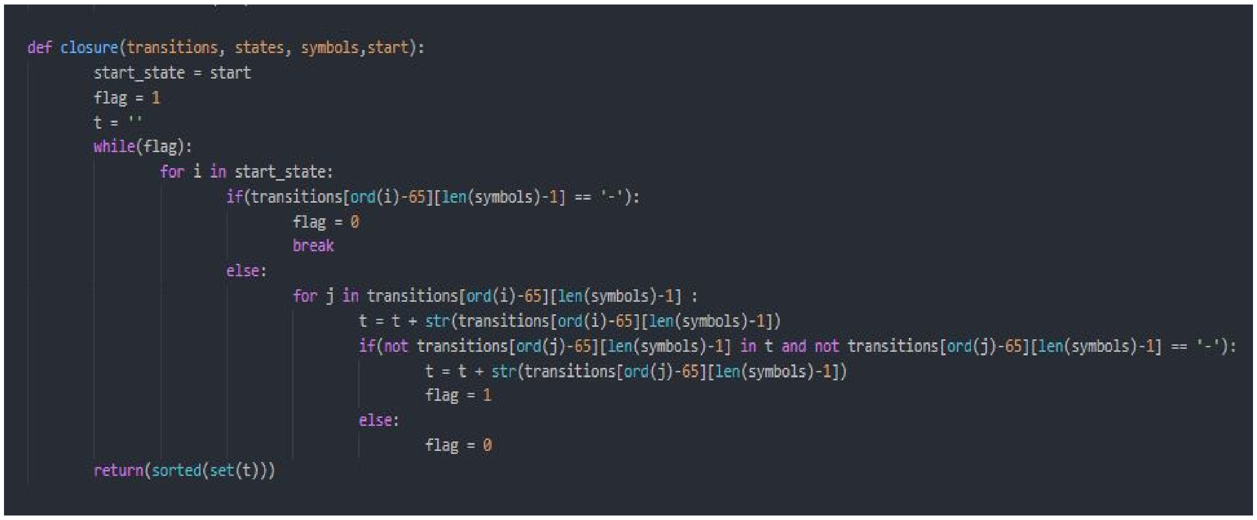
(union of transition value and their closures for each states of NFA present in current state of DFA).

Step 3 : If any new state is found take it as current state and repeat step 2.
Step 4 : Do repeat Step 2 and Step 3 until no new state present in DFA transition table.
Step 5 : Mark the states of DFA which contains final state of NFA as final states of DFA.

## ● Implementation

The closure function and transition function was implemented separated in the following way :

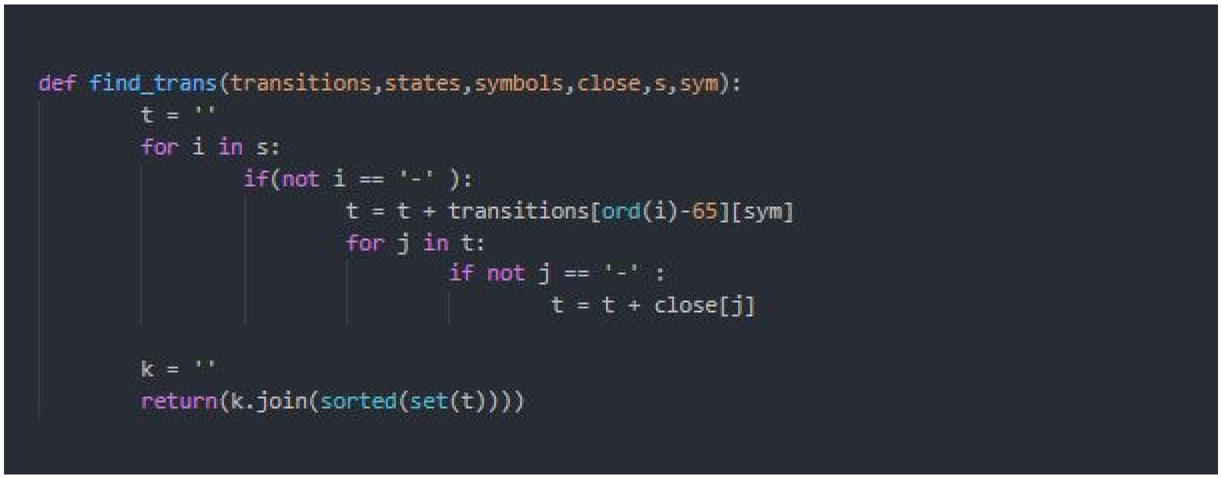
### 1) Closure function



```python
def closure(transitions, states, symbols,start):
	start_state = start
	flag = 1
	t = ''
	while(flag):
		for i in start_state:
			if(transitions[ord(i)-65][len(symbols)-1] == '-'):
				flag = 0
				break
			else:
				for j in transitions[ord(i)-65][len(symbols)-1] :
					t = t + str(transitions[ord(i)-65][len(symbols)-1])
					if(not transitions[ord(j)-65][len(symbols)-1] in t and not transitions[ord(j)-65][len(symbols)-1] == '-'):
						t = t + str(transitions[ord(j)-65][len(symbols)-1])
						flag = 1
					else:
						flag = 0
	return(sorted(set(t)))
```

This above function takes the `transitions, states, symbols,start` an input the function and returns a set with the closures of all the states in the states list.

### 2) Transition function

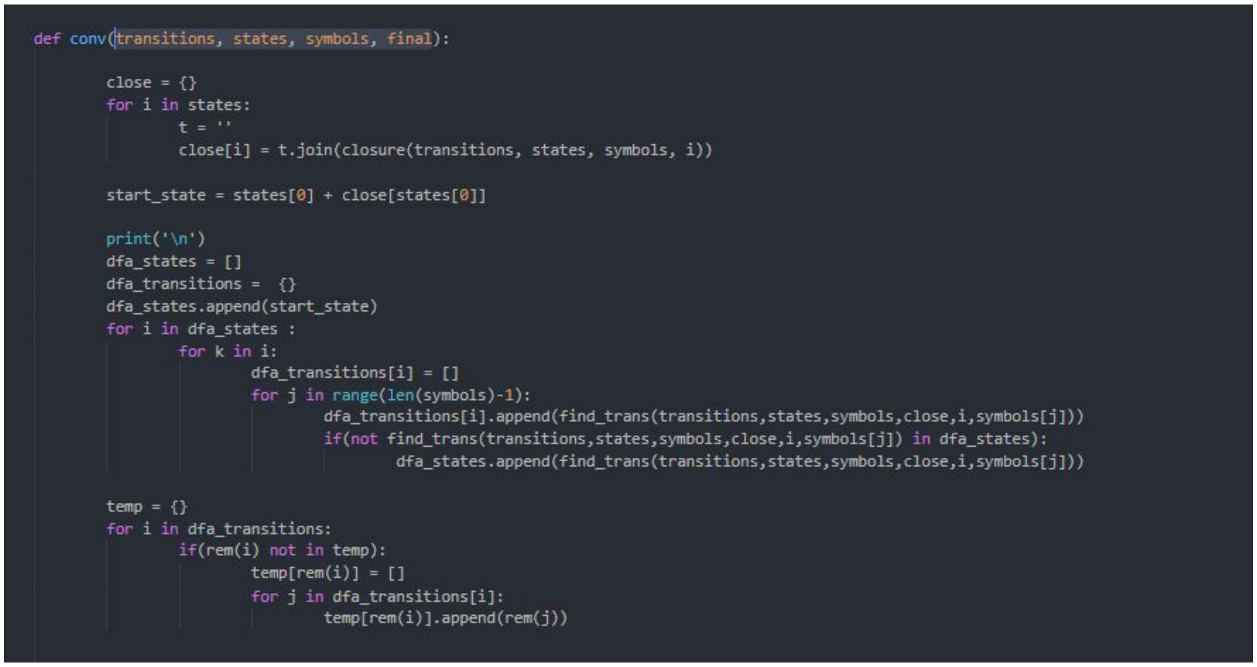


```python
def find_trans(transitions,states,symbols,close,s,sym):
	t = ''
	for i in s:
		if(not i == '-' ):
			t = t + transitions[ord(i)-65][sym]
			for j in t:
				if not j == '-' :
					t = t + close[j]

	k = ''
	return(k.join(sorted(set(t))))
```

The above transition function takes parameters `transitions,states,symbols,close,s,sym` and returns the new transition for the deterministic graph.

## 3) Conversion function



```python
def conv(transitions, states, symbols, final):

        close = {}
        for i in states:
                t = ''
                close[i] = t.join(closure(transitions, states, symbols, i))

        start_state = states[0] + close[states[0]]

        print('\n')
        dfa_states = []
        dfa_transitions =  {}
        dfa_states.append(start_state)
        for i in dfa_states :
                for k in i:
                        dfa_transitions[i] = []
                        for j in range(len(symbols)-1):
                                dfa_transitions[i].append(find_trans(transitions,states,symbols,close,i,symbols[j]))
                                if(not find_trans(transitions,states,symbols,close,i,symbols[j]) in dfa_states):
                                        dfa_states.append(find_trans(transitions,states,symbols,close,i,symbols[j]))

        temp = {}
        for i in dfa_transitions:
                if(rem(i) not in temp):
                        temp[rem(i)] = []
                        for j in dfa_transitions[i]:
                                temp[rem(i)].append(rem(j))
```

The above function takes the parameters `transitions, states, symbols, final` and gives the final converted DFA as the output.

## ● Conclusion

The algorithm described is thus used to convert a given NFA into into its equivalent DFA. This can be helpful while solving many problems where creating a NFA is more easier.
The complexity of the above algorithm is O(n^3). The 2-D array of string is used to implement the NFA structure. The DFA structure is represented using a list of strings. We have adopted an input enhancement technique which can be regarded as a transform and conquer design approach to find the closure the state beforehand. This eases the process of the transition.

## ● References

1. https://www.researchgate.net/publication/280561964_A_General_Approach_to_DFA_Construction
2. https://www.geeksforgeeks.org/theory-of-computation-conversion-from-nfa-to-dfa/
3. https://www.dennis-grinch.co.uk/tutorial/enfa-to-dfa