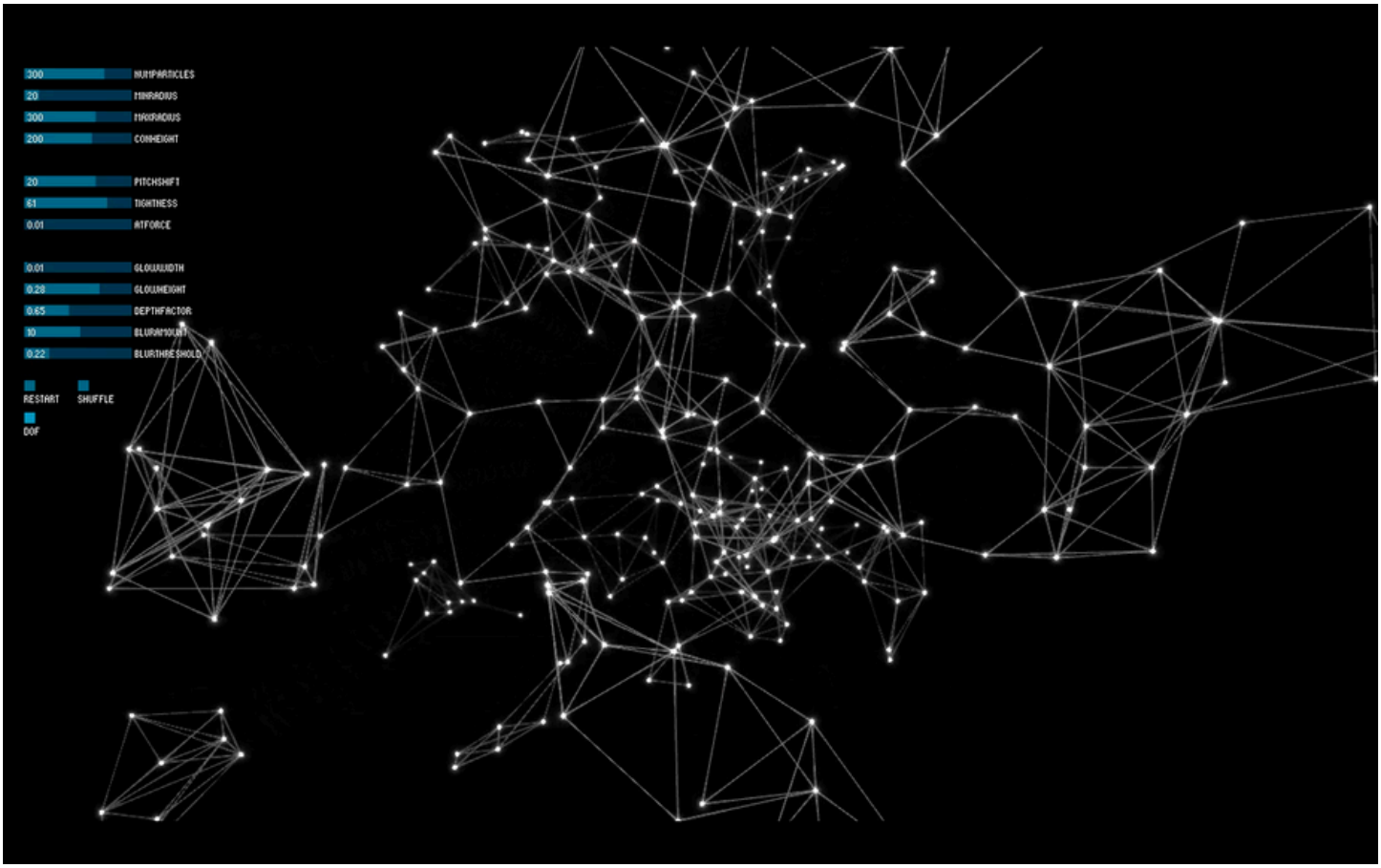
NFA to DFA Conversion in Python

By: Basant Saad El_din Mohammed | Dr: Salah El_din Shaban



Overview

This document explains the process of converting a **Nondeterministic Finite Automaton (NFA)** to a **Deterministic Finite Automaton (DFA)** using the given Python program. The program is designed to:

1. Represent an NFA using states, transitions, and other components.
2. Use the **subset construction algorithm** to transform the NFA into an equivalent DFA.
3. Display the resulting DFA, including its states, transitions, start state, and accept states.

Definitions

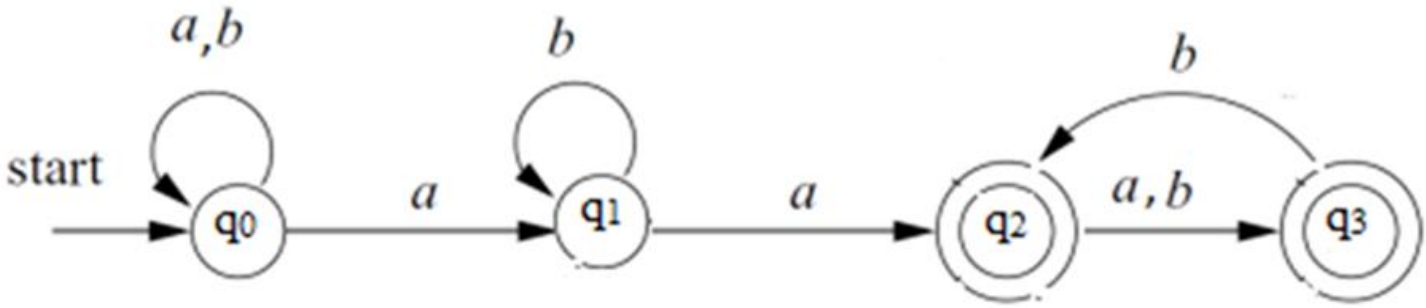
NFA (Nondeterministic Finite Automaton)

An NFA is defined as a 5-tuple ($\mathbf{Q,\Sigma,\delta,q_0,F}$) :

1. **States (Q)**: A finite set of states.
2. **Alphabet (Σ)**: A finite set of input symbols.
3. **Transition Function (δ)**:A dictionary mapping (state, symbol) to a set of states. This represents the transitions for each state and input symbol.
4. **Start State (q_0)**: The initial state of the NFA.
5. **Accept States (F)**: A subset of states where the computation can terminate successfully(accepting).

Example NFA (Defined in the Code)

Represent NFA using state diagram



1. **States(Q)**: {q0, q1, q2, q4}
2. **Alphabet(Σ)**: {a, b}
3. **Transitions(δ)**:
   - ('q0', 'a'): {'q0', 'q1'}
   - ('q0', 'b'): {'q0'}
   - ('q1', 'b'): {'q1'}
   - ('q1', 'a'): {'q2'}
   - ('q2', 'a'): {'q3'}
   - ('q2', 'b'): {'q2', 'q3'}
4. **Start State(q_0)**: q0
5. **Accept States(F)**: {q2,q3}

Represent NFA using state table

Current State	THE NEXT STAT WITH Input Symbol a	THE NEXT STAT WITH Input Symbol b
→ q0	○ {'q0', 'q1'}	○ {'q0'}
q1	○ {'q2'}	○ {'q1'}
*q2	○ {'q3'}	○ {'q2', 'q3'}
*q3	○ {}	○ {q2}

DFA (Deterministic Finite Automaton)

A DFA is defined similarly but differs in that:

1. Each (state, symbol) pair maps to exactly one state.
2. It has no epsilon (ε) transitions

DFA Conversion

Subset Construction Algorithm

The algorithm follows these steps:

1. **Start State**: Compute the **epsilon-closure** of the NFA start state to define the DFA's start state.
2. **State Exploration**: Use a queue to process each DFA state (set of NFA states), exploring all possible transitions for each input symbol.
3. **Transitions**: For each state and input symbol, compute the set of reachable NFA states and apply the epsilon-closure to form a new DFA state.
4. **Accept States**: Mark any DFA state as accepting if it includes at least one NFA accept state.
5. **Repeat**: Continue until all reachable DFA states are processed.

Breadth-First Search (BFS) Approach

- **Queue**: Used to process DFA states in a breadth-first manner.
- **Visited States**: Keeps track of states already processed to avoid redundant work.

Steps for NFA to DFA Conversion

1. **Start State**

Compute the epsilon-closure of the NFA's start state (q0):

- E(q0) = {q0} (No epsilon transitions in this example).

The DFA start state is {q0}.

2. **State Exploration**

Use the **subset construction algorithm** to generate DFA states and transitions:

1. Start with {q0} as the DFA's start state.
2. Process each symbol from the alphabet (a, b):
   - On a: {q0} → {q0,q1}
   - On b: {q0} → {q0}
3. Add the new states {q0, q1} to the DFA's states.
4. Continue exploring:
   - {'q0', 'q1'} --b--> {'q0', 'q1'}
   - {'q0', 'q1'} --a--> {'q0', 'q1', 'q2'}
   - {'q0', 'q1', 'q2'} --b-->{'q0', 'q1', 'q3'}
   - {'q0', 'q1', 'q2'} --a--> {'q0', 'q1', 'q2', 'q3'}
   - {'q0', 'q1', 'q2', 'q3'} --b--> {'q0', 'q1', 'q2', 'q3'}
   - {'q0', 'q1', 'q2', 'q3'} --a--> {'q0', 'q1', 'q2', 'q3'}
   - {'q0', 'q1', 'q3'}--b--> {'q0', 'q1', 'q2'}
   - {'q0', 'q1', 'q3'} --a--> {'q0', 'q1', 'q12}

3. **Accept States**

Any DFA state containing at least one NFA accept state (q2) is marked as an accept state:

- {'q0', 'q1', 'q3'} ,{'q0', 'q1', 'q2'} and {'q0', 'q1', 'q2', 'q3'} are DFA accept states.

Resulting DFA

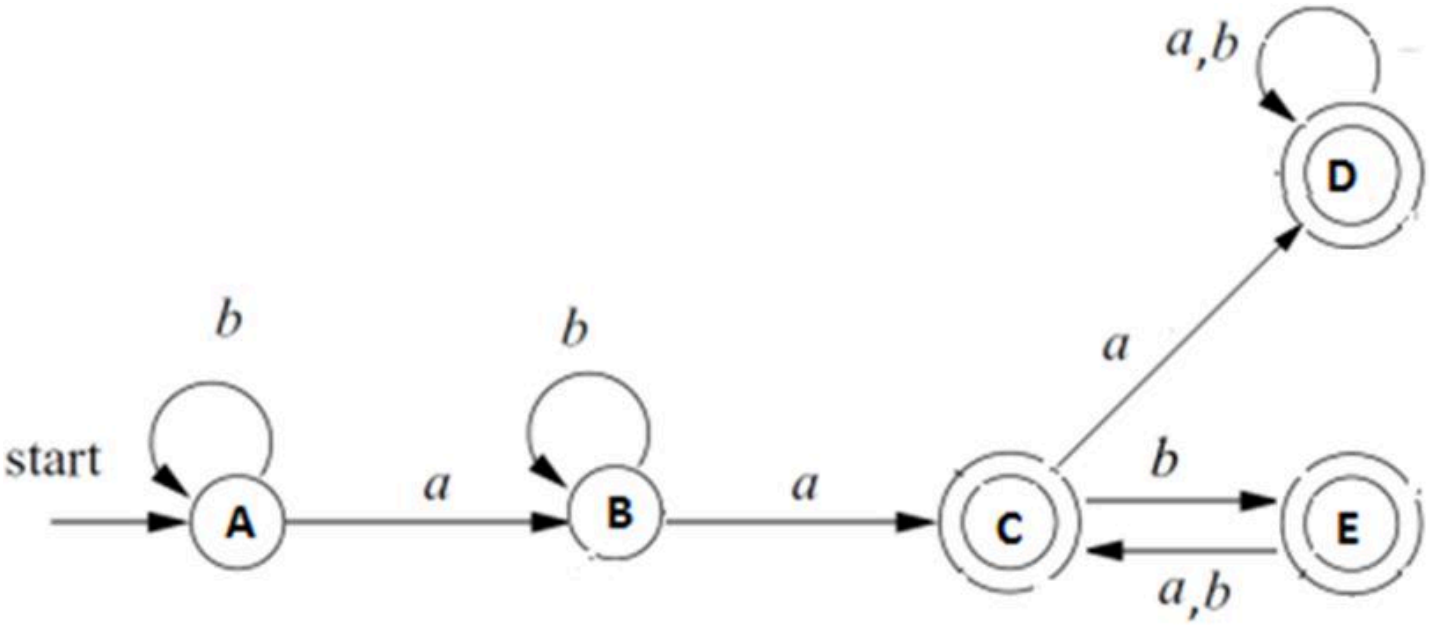
DFA States :

Q={ {q0}, {'q0', 'q1'}, {'q0', 'q1', 'q2'}, {'q0', 'q1', 'q3'}, {'q0', 'q1', 'q2', 'q3'} }

Represent DFA using state table:

DFA State	THE NEXT STATE WITH Input Symbol a	THE NEXT STATE WITH Input Symbol b
-->{q0}	{q0,q1}	{q0,}
{'q0', 'q1'}	{'q0', 'q1', 'q2'}	{'q0', 'q1'}
*{'q0', 'q1', 'q2'}	{'q0', 'q1', 'q2', 'q3'}	{'q0', 'q1', 'q3'}
*{'q0', 'q1', 'q3'}	{'q0', 'q1', 'q2'}	{'q0', 'q1', 'q2'}
*{'q0', 'q1', 'q2', 'q3'}	{'q0', 'q1', 'q2', 'q3'}	{'q0', 'q1', 'q2', 'q3'}

Represent DFA using state diagram



Python Implementation

NFA Class

The NFA class represents the nondeterministic finite automaton.

Key Methods:

- **__init__**: Initializes the NFA with states, alphabet, transitions, start state, and accept states.
- **epsilon_closure(states)**: Computes the epsilon-closure of a set of states by recursively exploring transitions labeled with '' (epsilon).

DFA Class

The DFA class represents the deterministic finite automaton constructed from the given NFA.

Key Methods:

- **__init__**: Initializes the DFA with references to the NFA and computes DFA states and transitions.
- **create_dfa()**: Implements the subset construction algorithm to build the DFA.
- **display_dfa()**: Prints the DFA states, transitions, start state, and accept states.

Here's the Python program applied to this general example:

```
from collections import defaultdict

class NFA:
    def __init__( self, states, alphabet, transitions, start_state, accept_states):
        self.states = states
        self.alphabet = alphabet
        self.transitions = transitions  # (state, symbol) -> set of states
        self.start_state = start_state
        self.accept_states = accept_states
    def epsilon_closure(self, states):
        """ Compute the epsilon closure of a set of states """
        closure = set(states)
        stack = list(states)
        while stack:
            state = stack.pop()
            for next_state in self.transitions.get((state, ''), set()):
                if next_state not in closure:
                    closure.add(next_state)
                    stack.append(next_state)
        return closure
class DFA:
    def __init__(self, nfa):
        self.nfa = nfa
        self.dfa_states = {}
        self.dfa_transitions = {}
        self.start_state = frozenset(nfa.epsilon_closure({nfa.start_state}))
        self.accept_states = set()
```

```
    self.create_dfa()

def create_dfa(self):
    unmarked_states = [self.start_state]
    self.dfa_states[self.start_state] = True
    while unmarked_states:
        current_dfa_state = unmarked_states.pop()
        for symbol in self.nfa.alphabet:
            if symbol == '': continue
            next_nfa_states = set()
            for nfa_state in current_dfa_state:
                next_nfa_states.update(self.nfa.transitions.get((nfa_state, symbol), set()))
            next_dfa_state = frozenset(self.nfa.epsilon_closure(next_nfa_states))
            if next_dfa_state not in self.dfa_states:
                unmarked_states.append(next_dfa_state)
                self.dfa_states[next_dfa_state] = True
                if any(state in self.nfa.accept_states for state in next_dfa_state):
                    self.accept_states.add(next_dfa_state)
            self.dfa_transitions[(current_dfa_state, symbol)] = next_dfa_state
def display_dfa(self):
    print("DFA States:")
    for state in self.dfa_states:
        print(f"  {state}")
    print("\nDFA Transitions:")
    for (state, symbol), next_state in self.dfa_transitions.items():
        print(f"  {state} --{symbol}--> {next_state}")
    print("\nDFA Start State:", self.start_state)
    print("DFA Accept States:", self.accept_states)
```

```
#Example NFA

states = {'q0', 'q1', 'q2','q4'}

alphabet = {'a', 'b'}

transitions = {

  ('q0', 'a'): {'q0', 'q1'},

  ('q0', 'b'): {'q0'},

  ('q1', 'a'): {'q2'},

  ('q1','b') : {'q1'},

  ('q2','a'):{'q3'},

  ('q2','b'):{'q3'},

  ('q3','b'):{'q2'} }

start_state = 'q0'

accept_states = {'q3'}

# Convert NFA to DFA

nfa = NFA(states, alphabet, transitions, start_state, accept_states)

dfa = DFA(nfa)
```

https://pastebin.com/YhAYNj90

Key Insights

1. **Efficiency**: The BFS approach ensures that each DFA state is processed exactly once.
2. **General Applicability**: This implementation can handle any NFA, including epsilon transitions.
3. **Readability**: The use of Python's high-level data structures like set and dict simplifies state management and transition logic.

Conclusion

This implementation effectively converts any NFA into a DFA using the subset construction algorithm. The code ensures that all possible transitions are processed, and it outputs the resulting DFA's states, transitions, start state, and accept states. This example highlights how deterministic behavior is derived from nondeterminism by systematically examining reachable states.