Automata Theory
Assignment-1 Report

Jivitesh Jain
2018101092

August 18, 2019

1 Overview

This bundle is a python program that converts a *non-deterministic finite state automaton (NFA)* to an equivalent *deterministic finite state automaton (DFA)*.

1.1 The Interface

The bundle consists of the following files:

- `script.py`
- `automaton.py`
- `utils.py`
- `report.pdf`

To run the program, create a file `input.json` containing the input NFA definition as a `json` object containing the following key-value pairs:

- `"states"`: Number of states
- `"letters"`: Alphabet as a list
- `"t_func"`: The transition relation
- `"start"`: The start state
- `"final"`: List of final states

Each state is represented by a number between 0 and one less than the number of states. The transition relation is a list of lists, each of which looks like:

```
[current_state, input_alphabet, list_of_next_states]
```

Then, execute:

```
$ python3 script.py
```

This will create a file `output.json` which will contain the output DFA in the same format as `input.json`.

1.2 Sample Input and Output

`input.json`

```
{
    "states": 2,
    "letters": ["a", "b"],
    "t_func": [
                [0, "a", [0, 1]],
                [0, "b", [0]],
                [1, "b", [0]]
            ],
    "start": 0,
    "final": [0]
}
```

`output.json`

```
{
    "states": 4,
    "letters": ["a", "b"],
    "t_func": [
                [[], "a", []],
                [[], "b", []],
                [[0], "a", [0, 1]],
                [[0], "b", [0]],
                [[1], "a", []],
                [[1], "b", [0]],
                [[0, 1], "a", [0, 1]],
                [[0, 1], "b", [0]]
            ],
        "start": [0],
        "final": [[0], [0, 1]]
}
```

1.3 The Algorithm

The program uses the following algorithm to generate an equivalent DFA, given an NFA:

Given an NFA $N = (Q, \Sigma, \delta, q_0, F)$, which recognizes a language A,
an equivalent DFA M can be defined as: $M = (Q', \Sigma, \delta', q_0', F')$

where,

1. $Q' = \mathcal{P}(Q)$

2. $\delta'(R, a) = \bigcup_{r \in R} \delta(r, a)$

3. $q'_0 = \{q_0\}$

4. $F' = \{R \in Q' | R \cap F \neq \phi\}$

The program does not account for NFA's with ε - transitions and does not perform state reduction

2 Implementation

2.1 Overview

The program defines and uses the following 3 classes:

The NFA Class

This class encapsulates an NFA, parsing its components from the given input dictionary. The method `make_trans` converts the transition relation from a python-list to a python-dictionary (hashmap) with the keys as tuples of current state and the input alphabet and the values as python-sets of the next states. This allows for fast lookup of a transition and makes taking set-unions easier and faster.

```
class NFA:
    def __init__(self, nfa_dict):
        self.trans = self.make_trans(nfa_dict['t_func'])
        ...

    def make_trans(self, arr):
        trans = {}
        for triplet in arr:
            trans[tuple(triplet[:2])] = set(triplet[2])
        return trans
```

The DFA Class

This class encapsulates a DFA. Although all the attributes are set from outside by a separate function, it does provide a method to return a python-dictionary containing all the components of the DFA, which can then be directly converted to `json`.

The BitString Class

This utility class simplifies conversion to and from the binary representation of the states of the DFA. This helps in mapping the states of the DFA to the states of the NFA they contain.

```
class BitString:
    def __init__(self, size, initial=0):
```

```
        ...
    def set_bit(self, pos, val):
        ...
    def get_bit(self, pos):
        ...
    def get_num(self):
        ...
    def set_num(self, num):
        ...
```

2.2 Conversion

The program uses the function `nfa_to_dfa` which accepts an NFA and returns an equivalent DFA.

```
def nfa_to_dfa(nfa):
    dfa = DFA()
    ...
```

The conversion of each component of the automaton happens as follows:

States, Letters and the Start State

```
dfa.states = 2 ** nfa.states
dfa.letters = nfa.letters
dfa.start = [nfa.start]
```

This is in accordance with the algorithm described above. Because the set of states of the DFA is the power set of the set of states of the NFA, an NFA with n states has 2^n states in its equivalent DFA, ranging from 0 to $2^n - 1$.

Final States

```
nfa_final = set(nfa.final) # for fast membership testing
final = []
for dfa_state in range(dfa.states):
    bits = BitString(nfa.states, dfa_state)
    append = False
    state_list = []
    for i in range(nfa.states):
        if bits.get_bit(i) == 1:
            state_list.append(i)
            if i in nfa_final:
                append = True
```

```
        if append:
            final.append(state_list)
    dfa.final = final
```

For each state `dfa_state` of the DFA, this piece of code loops over its binary representation to determine which all states of the NFA it contains (by checking if the corresponding bit is 1). If any such state is a final state of the NFA, `dfa_state` is added to the list of final states of the DFA.

The Transition Function

```
    trans = []
    for dfa_state in range(dfa.states):
        for a in dfa.letters:
            bits = BitString(nfa.states, dfa_state)
            in_list = []
            out_set = set()
            for i in range(nfa.states):
                if bits.get_bit(i) == 1:
                    in_list.append(i)
                    if (i, a) in nfa.trans:
                        out_set = out_set | nfa.trans[(i, a)]
            trans.append([list(in_list), a, list(out_set)])
    dfa.trans = trans
```

For each state `dfa_state` of the DFA and each input alphabet `a`, this piece of code loops over all the NFA states to determine which of those lie in `dfa_state`. Such states are then added to the list representation of `dfa_state` and their set of next states corresponding to `a` is then unioned to the set of next states for `dfa_state` and `a` in the DFA.

2.3 Bringing it all together

The script `script.py` uses the `json` module to parse `input.json` into a python-dictionary which it uses to create an object of the `NFA` class. It then calls `nfa_to dfa` on this object, and uses the `json` module again to write the output to a `json` file.