Automata Disambiguation

Islam Hamada

March 2021

1 Introduction

Unambiguous automata are of special interest, since some problems are easier for them, contrary to ambiguous automaton, such as the universality problem, and language containment[1]. The two problems are decidable in polynomial time for unambiguous automaton, while they are generally PSPACE-complete for the class of non-deterministic finite automata.

Luckily it's possible to convert any ambiguous automata to an unambiguous one. And I implemented 3 disambiguation algorithms in this lab course in *java*. And I'll present them in this report.

1.1 Automata

A finite automaton consists of a set of states, labeled transitions connecting the states, some initial states and some final states. An automaton accepts some word if it satisfies the acceptance condition. The set of words accepted by an automaton is called the language of the automaton. In this report, we are concerned with 3 types of automata: DFA, NFA and AFA that I'll explain next.

1.1.1 DFA

A deterministic finite automaton is a 5-tuple $\langle Q, \Sigma, \delta, I, F\rangle$ where

- Q - a finite set of states.
- Σ - the alphabet which is a set of input symbols.
- δ - a transition function $\delta : Q \times \Sigma \to Q$.
- I - is the initial state $I \in Q$.
- F - a set of accepting states $F \subseteq Q$.

1.1.2 NFA

A non-deterministic finite automaton is a 5-tuple $\langle Q, \Sigma, \delta, I, F\rangle$ where

- Q - a finite set of states.
- Σ - the alphabet which is a set of input symbols.
- δ - a transition relation $\delta : Q \times \Sigma \to 2^Q$.
- I - is the set of initial states $I \subseteq Q$.
- F - a set of accepting states $F \subseteq Q$.

NFAs differ from DFAs because NFAs can have multiple initial states and the transition relation isn't necessarily a function, i.e, NFAs have existential branching.

1.1.3 AFA

An alternating finite automaton is a 5-tuple $\langle Q, \Sigma, \delta, I, F\rangle$ where

- Q - a finite set of states.
- Σ - the alphabet which is a set of input symbols.
- δ - a transition function $\delta : Q \times \Sigma \to 2^{2^Q}$.
- I - is the set of initial states $I \subseteq Q$.
- F - a set of accepting states $F \subseteq Q$.

AFAs differ from NFAs because AFAs can also have universal branching, where a transition can lead to a conjunction of states.

1.2 Ambiguous Automata

A subset of automata that we are concerned with in this paper, is ambiguous automata. An automaton is ambiguous, if for some word w, the automaton has more than one path through its states and transitions to accept w starting from an initial state. For example consider the automaton in (1):

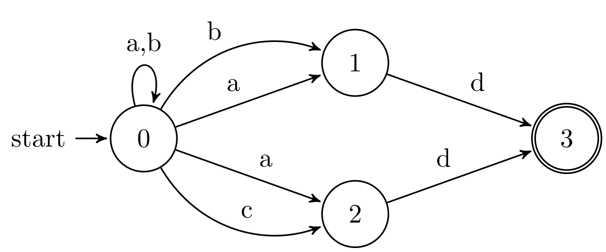

Figure 1: An NFA

For the word *ad*, it can be accepted through two paths:

- $q_0 \xrightarrow{a} q_1 \xrightarrow{d} q_3$
- $q_0 \xrightarrow{a} q_2 \xrightarrow{d} q_3$

Consequently this automaton is ambiguous. Finally an automaton is unambiguous if it's not ambiguous.

2 Disambiguation

Luckily we can convert any ambiguous automaton to unambiguous one. There are many algorithms that do so such as:

1. Determinization(subset construction)
2. Reverse Alternation Removal
3. Disambiguation Through Complementation
4. Mohris algorithm

The algorithms can be found in [2] and [3] with definitions and proofs. In this lab course, I only implemented the first 3 algorithms in Java.

2.1 Classes

2.1.1 Automaton

abstract class Automaton⟨StateCore, Alphabet, TransitionOutput, InputStateCore, InputTranOutput⟩

Automaton is a generic abstract class. It has 5 type variables:

- StateCore - the type of each state, it could be Int or a *Set⟨Int⟩*, etc.
- Alphabet - the type of letters in the transitions
- TransitionOutput - the type of the output of the transition function
- InputStateCore - the type of the state core of the input automaton, if needed
- InputTranOutput - the type of the output of the transition function of the input automaton, if needed

The class has many attributes such as:

- *alphabet* - a set of letters as the *alphabet* of type *Set⟨Alphabet⟩*
- *acc_states* - a set of *acceptance* states of type *Set⟨StateCore⟩*

- *trans* - a nested map to store the *transitions* of type *Map⟨StateCore, Map⟨Alphabet, TransitionOutput⟩⟩*

Also the class has 3 constructors:

- A regular one where only the *acceptance* states, the *alphabet* and the transitions map are provided.
- Another that uses an input automaton, and a function to expand any states forwards.
- The last is one that uses an input automaton, and a function that expands states backwards.

Also the class has many functions such as getters, setters, a function to add a new state, a function to expand forwards, another to expand backwards. And some abstract funcitons such as:

- trim() - which removes states that can never lead to acceptance
- complete_aut() - which adds the dead state, if missing, along with any missing transitions
- get_reachable_states()

2.1.2 DFA

class DFA⟨StateCore, Alphabet, InputStateCore, InputTranOutput⟩ extends Automaton⟨StateCore, Alphabet, StateCore, InputStateCore, InputTranOutput⟩

DFA is a subclass of Class *Automaton*, that has only 4 types. And the key difference is that in DFA, the transition output is a single state. Also the class has an additional attribute for the initial state of type *StateCore*.

2.1.3 NFA

class NFA ⟨StateCore, Alphabet, InputStateCore, InputTranOutput⟩ extends Automaton⟨StateCore, Alphabet, Set⟨StateCore⟩, InputStateCore, InputTranOutput⟩

NFA is a subclass of Class *Automaton*, that has only 4 types. And the key difference is that in NFA, the transition output is a set of states. Also the class has an additional attribute for the initial states of type *Set⟨StateCore⟩*

There are some additional functions such as:

- determinize() - it performs disambiguation by determinization(subset construction) and returns a DFA
- self_product() -

2.1.4 AFA

class AFA⟨StateCore, Alphabet, InputStateCore, InputTranOutput⟩ extends Automaton⟨StateCore, Alphabet, Set⟨Set⟨StateCore⟩ ⟩, InputStateCore, InputTranOutput⟩

AFA is a subclass of Class *Automaton*, that has only 4 types. And the key difference is that in AFA, the transition output is a set of sets of states. Also the class has an additional attribute for the initial states of type *Set⟨Set⟨StateCore⟩⟩*

There are some additional functions such as:

- reverseDeterminize() - it performs Reverse Alternation removal algorithm and return an unambiguous NFA
- convertToSingleInitialState()
- complement()
- diambiguateByComplement() - it performs disambiguation by complementation and returns an unambiguous AFA
- forwardAlternationRemoval()
- configTranFunction()

3 Implementation

3.1 Main Algorithms

The first 2 algorithms has a similar structure, as for them the input has 4 main components:

- An input automaton
- The initial\acceptance state of the output automaton
- A lambda function to expand a state forwards\backwards of the output automaton
- Boolean lambda function to check whether a state resulting from the expansion is an acceptance\initial state

So the algorithms will define an output automaton with an initial\acceptance state, a state expansion function, and a check. Then starting from the initial\acceptance state, we expand the state forwards\backwards by computing the states that are connected to the current state, also for each new state we perform a check to see if each new state should be marked as acceptance\initial state. And the expansion function gets applied again to all new states, until no new states are created.

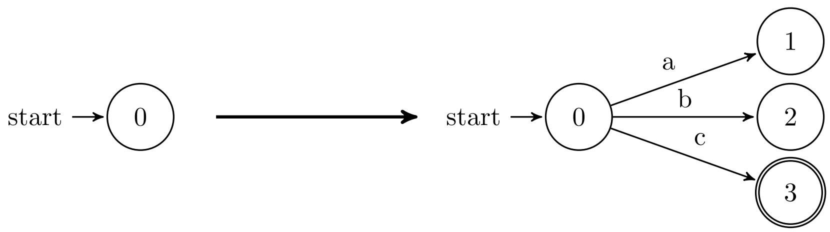

Figure 2: After a step of forward expansion

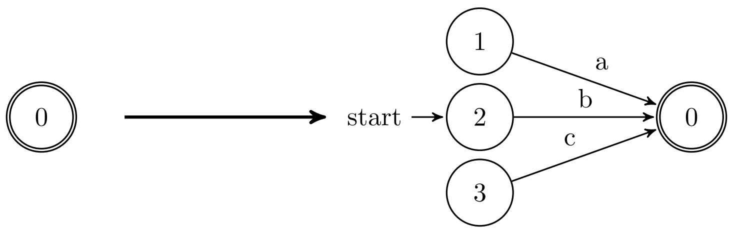

Figure 3: After a step of backward expansion

Algorithm 1 Automaton Full Forward Expansion

$Queue \leftarrow$ initialStates
$ExpandedStates \leftarrow \emptyset$
while $Queue \neq \emptyset$ **do**
  $State \leftarrow Queue$.removeTop()
  if $State \notin ExpandedStates$ **then**
    $Queue$.addAll(expandStateForward($State$))
    isAcceptState($State$)
  end if
  $ExpandedState \leftarrow ExpandedState \cup State$
end while
return

The algorithm in (1) is for automaton forward expansion. The algorithm for full backwards expansion is quite similar. Note that for the automaton to be fully expanded forwards, 3 attributes need to be defined:

- *initialStates* - the initial states
- *expandStateForward()* - a function to expand a state forwards
- *isAcceptState()* - a function that checks whether a state should be added to the set of the acceptance states

So for the first two disambiguation algorithms it suffices to:

- Describe whether the algorithm is forward or backward expansion
- Describe the function expandStateForward() or expandStateBackwards() respectively
- Describe the function isAcceptState() or isInitialState() respectively

3.1.1 Determinization

The 1st disambiguation algorithm starts with a possibly ambiguous NFA with state core of type X and uses the subset construction to build a DFA with state core of type $Set\langle X\rangle$. The algorithm is a forward expansion. The initial state of the output automaton is the set of initial states of the input automaton. expandStateForward() is defined as (2).

Note that by converting an NFA to a DFA, the automaton becomes unambiguous. Because determinism implies unambiguity, since given any state and a letter, you can only go to one state.

Algorithm 2 ExpandStateForward(Set⟨X⟩ State, Letter, InputAutTransitions)

$OutputState \leftarrow \emptyset$
for S : $State$ **do**
  $S\_Transition \leftarrow InputAutTransitions.\mathrm{get}(S).\mathrm{get}(Letter)$
  for S_2 : $S\_Transition$ **do**
    $OutputState \leftarrow OutputState \cup S_2$
  end for
end for
if $OutputState = \phi$ **then**
  return null;
else
  return OutputState;
end if

isAcceptState() is defined as in (3).

Algorithm 3 isAcceptState(Set⟨X⟩ InputAutAccStates, Set⟨X⟩ state)

return $(state \cap InputAutAccStates) \neq \emptyset$;

3.1.2 Reverse Alternation Removal

The 2nd disambiguation algorithm starts with an AFA with state core of type X and returns an NFA with state core of type $Set\langle X\rangle$, and that output automaton is reverse deterministic. The algorithm is a backward expansion that starts with the acceptance state and expands back to the initial states. The acceptance state is the set of acceptance state of the input automaton. expandStateBackward() is defined as (4).

The algorithm works because after applying it, the output automaton is reverse-deterministic, the proof can be found in [3]. An automaton is reverse-determinstic when if you reverse the direction of the transitions, given a state and a letter you can only go to one state. Hence reverse-deterministic automata are unambigous.

Algorithm 4 ExpandStateBackward(Set⟨X⟩ State,Letter,InputAutTransitions)

$OutputState \leftarrow \emptyset$
for $S : InputAutTransitions.keySet()$ **do**
  $S\_Transition \leftarrow InputAutTransitions.\text{get}(S).\text{get}(Letter)$
  for $set : S\_Transition$ **do**
    if $set \subseteq State$ **then**
      $OutputState \leftarrow OutputState \cup set$
    end if
  end for
end for
if $OutputState = \phi$ **then**
  return null;
else
  return OutputState;
end if

isInitialState() is defined as in (5).

Algorithm 5 isInitialState(Set⟨X⟩ InputAutInitStates, Set⟨X⟩ state)

return $(state \cap InputAutInitStates) \neq \emptyset$;

3.1.3 Disambiguation Through Complementation

The 3rd disambiguation algorithm is different because it starts with an AFA and performs some local disambiguations where it's necessary, until the automaton is finally unambiguous(6). The algorithm uses another called **complement**(7) to compute the complement automaton, which uses an algorithm **allSelections**(6). They are all shown below.

The algorithm applies the following equivalence $(a \vee b) \leftrightarrow (a \vee (b \wedge \neg a))$. So given a state and a letter, if there's more than one choice the equivalence is applied to makes sure that the choices lead to a disjoint sets of accepted words, hence no ambiguitiy.

References

[1] Richard Edwin Stearns and Harry B Hunt III. On the equivalence and containment problems for unambiguous regular expressions, regular grammars and finite automata. *SIAM Journal on Computing*, 14(3):598–611, 1985.

Algorithm 6 LocalDisambiguation(automaton)

```
complement ← automaton.complement()
automaton ← merge(automaton, complement)
ambiguous ← true; state_found ← false;
while ambiguous do
  state_found = false; ambiguous = false;
  \\to convert the AFA to an NFA
  nfa ← automaton.forwardAlternationRemoval();
  self_product = nfa.self_product(); self_product.trim();
  for s : self_product.state_space() do
    if (the state s is an indentical pair, and has a transition to a state s2 that is not an identical pair)
    then
      ambiguous = true;
      for AFAState : s.left do
        set ← automaton.getTrans().get(AFAState);
        for x : set do
          for y : set do
            if x ≠ y & s2.left.containsAll(x) & s2.right.containsAll(y)
            then
              state_found = true;
              set.remove(y);
              for s3 : x do
                new_states_set.addAll(y);
                new_states_set.add(s3);
                set.add(new_states_set);
              end for
              break;
            end if
          end for
          if state_found then
            break;
          end if
        end for
        if state_found then
          break;
        end if
      end for
      if ambiguous then
        break;
      end if
    end if
  end for
end while
```

Algorithm 7 complement()

```
comp_aut_trans_map ← new HashMap();
for State : transitions.keySet() do
    comp_state_map ← new HashMap();
    for Letter : transitions.get(State).keySet()) do
        sets ← transitions.get(State).get(letter);
        comp_state_transitions ←allSelections(sets);
        comp_state_map.put(Letter, comp_state_transitions);
    end for
    comp_aut_trans_map.put(State, comp_state_map);
end for
comp_acc_states ← InputAutStateSpace \ InputAutAccStates;
compAut ← new AFA(InputAutInitStates, InputAutAlphabet, comp_acc_states, comp_aut_trans_map);
```

Algorithm 8 allSelections(listOfLists)

```
numOfLists ← listOfLists.size();
loop ← true;
\\selection is the result of picking a member of each list
selections ← newList();
while loop do
    selection ← newList()
    for (i = 0 ; i < numOfLists ; i++) do
        selection[i] ← listOfLists[i][indices[i]];
    end for
    selections.add(selection);
    loop = false;

    \\update the indices
    for (i = numOfLists − 1 ; (i ≥ 0 && !loop) ; i--) do
        indices[i]++
        if (indices[i] ≥ listOfLists[i].size()) then
            indices[i] = 0;
        else
            loop = true;
        end if
    end for
end while
```

[2] Mehryar Mohri. A disambiguation algorithm for finite automata and functional transducers. In *International Conference on Implementation and Application of Automata*, pages 265–277. Springer, 2012.

[3] Yufei Liu. Disambiguation of alternating finite automata over finite words and its application to nfa.