Afonso das Neves Fernandes

Degree of Ambiguity

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Junho de 2019

Afonso das Neves Fernandes

Degree of Ambiguity

Relatório de Projeto

Orientadores: Nelma Resende Araújo Moreira e Rogério Ventura Lages dos Santos Reis

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Junho de 2019

Contents

List of Figures

Algorithms

Chapter 1

Introduction

One important subject in computer science is theory of finite automata because automata are a robust model of computation that has widespread applications from compilers to bioinformatics, image recognition and computer networks.

One of such applications is in pattern matching and in particular regular expression pattern matching. For manipulation, expressions are usually converted into non-deterministic finite automata. One of the difficulties is the ambiguity of the automata because there may exist several possible matches and thus several syntactic trees. To prevent these problems, normally, greedy strategies are implemented to select one [3].

In this report, we will study the ambiguity of non-deterministic finite automata. In particular we will use the library of FAdo [4] to implement the criteria for ambiguity classification and run experiments to study the distribution of the classes of ambiguity.

We start by defining ambiguity of an automaton and the classes of ambiguity. Then we present the criteria to classify the ambiguity and their implementations.

Finally, we will present the experiments done in order to study the distribution of the classes of ambiguity and discuss the results.

1.1 Definitions and Notations

In this section we present some definitions used in this report. In particular the definition of an NFA and a DFA, and the definition of a path.

Given a finite set of symbols Σ, called an alphabet, and a word as a finite sequence of elements of Σ, we define a language to be a set of words over an alphabet and Σ^* to be the language of all words that can be formed using symbols of Σ.

Throughout this report, we use some simple regular expressions with the following operations: concatenation, + as disjunction and $*$ as the Kleene star.

Definition 1. A non-deterministic finite automaton (NFA) M is defined as a 5-tuple (Q,Σ,δ,I,F) where Q is the set of states, Σ the set of input symbols, $\delta \subseteq Q \times \Sigma \times Q$ the finite set of transitions and I,F are the set of initial and final states, respectively.

The size of an NFA is defined as $|Q| + |\delta|$, the number of states plus the number of transitions.

Definition 2. A deterministic finite automaton (DFA) is an NFA where for each state s and symbol x there is at most one pair (s,x,s') in δ, with s' in Q.

Definition 3. A path π on an NFA is a sequence of states, bigger than one, such that between two consecutive states of π there is at least one transition. We denote by $i[\pi]$ and by $f[\pi]$ the origin state and destination state of π. If $i[\pi] \in I$ and $f[\pi] \in F$ the path π is an accepting path. A path π is labeled by a set of words, each one resulting from the concatenation of symbols of the transitions between consecutive states of π ($l[\pi]$). A path π spells a word $w \in \Sigma^*$ if $w \in l[\pi]$. A path accepts a word if it is an accepting path and if the path spells the word.

(a) NFA (b) DFA

Figure 1.1: Example of an NFA and a DFA

In figure 1.1 are shown two automata diagrams. The circles represent states and the arrows represent transitions.

In Figure 1.1a we have the following paths:

- $\pi_1 = 0, 0, 1$
- $\pi_2 = 0, 1, 1$

The path π_1 is an accepting path and is labeled only by the word aa. On the other hand, the path π_2 is not an accepting path and is labeled by $\{aa, ab\}$

Definition 4. The language of an NFA M, $L(M)$, is the union of $l[\pi]$ for all accepting paths π on M.

All languages that can be represented by finite automata are called regular languages. If two automata accept the same language they are equivalent.

It is known for each positive integer m exists an NFA with m states such that the smallest equivalent DFA will have 2^m states [9].

For DFA's there is at most one accepting path for each word. This leads to more efficient algorithms for the membership problem, this is to decide if a word belongs to the language defined by the automaton.

Although the decision problems on a DFA are computational more efficient than on an NFA with the same size, NFA's are normally used due to the exponential growth in size in the conversion to the equivalent DFA.

Definition 5. A state of an automaton is useful if it is in some accepting path. If all states of an automaton are useful then the automaton is trim.

Definition 6. The product automaton, $M_1 \times M_2$, is an automaton where: the states are of the form (s_1, s_2), where both s_1 and s_2 are states of M_1 and M_2, respectively; there is a transition $((s_1, s_2), x, (s'_1, s'_2))$ if in M_1 there is the transition (s_1, x, s'_1) and in M_2 there is the transition (s_2, x, s'_2); and the state (s_1, s_2) is in the initial set if both s_1, s_2 are in the initial set of M_1 and M_2, respectively.

In this work, the library of FAdo is used to implement the algorithms and make the experiments [4]. In the library there is a structure for an NFA, where (Q,Σ,δ,I,F) are respectively represented by the following attributes: States(list), Sigma(set), delta(dict), Initial(set), Final(set).

Chapter 2

Ambiguity

In this chapter we will study a metric of ambiguity of an NFA and the underlying algorithms to classify it.

2.1 Degree of Ambiguity

An NFA M is *ambiguous* if there are at least two distinct paths of M that accept a word $w \in \Sigma^*$. Furthermore, the number of paths that accepts w in the automaton M is defined as the degree of the word w in the automaton M, $da_M(w)$.

The degree of ambiguity on an NFA M is defined as $sup\{da_M(w)|w \in \Sigma^*\}$. This degree can be either finite or infinite. The degree is finite if there is a positive integer k such that, for all words w, the $da_M(w) \leq k$ otherwise is infinite.

There are 4 classes of ambiguity to classify an NFA [12]: *unambiguous* (UFA), *finitely ambiguous* (FNFA), *polynomially ambiguous* (PNFA) and *exponentially ambiguous* (ENFA).

Definition 7. An NFA is *unambiguous* if, for each word w, there is at most one accepting path.

Figure 2.1a shows an example of an NFA diagram that is *unambiguous*. All accepted words have only one accepting path. Note that a DFA is always *unambiguous* (but not all *unambiguous* automata are DFA).

Definition 8. An NFA is *finitely ambiguous* if there is a $k \in Z$ such that, for each word w, the number of paths that accepts w is at most k.

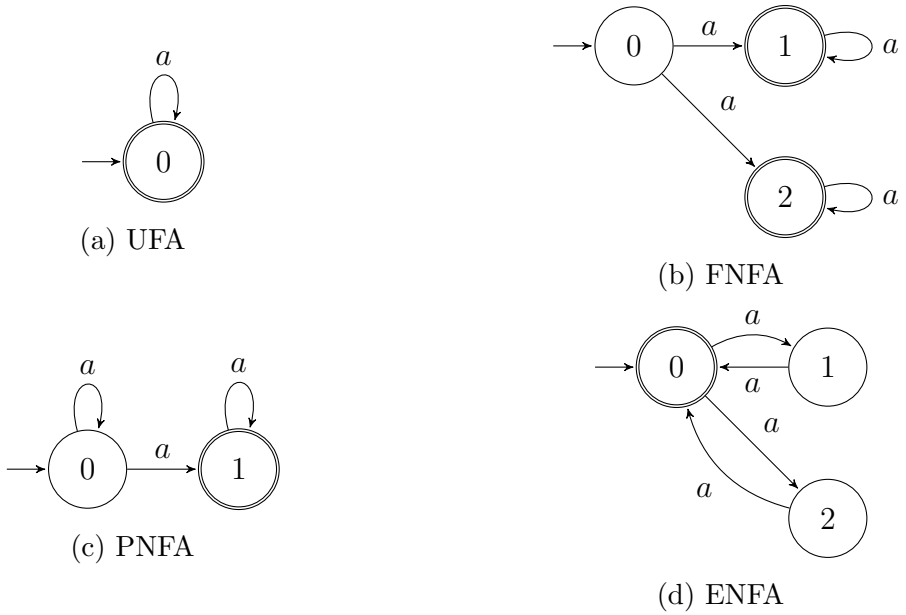


Figure 2.1: Classes of Ambiguity

Figure 2.1b shows an example of an NFA diagram that is *finitely ambiguous*, where the accepted language is the language of the regular expression aa^*. The degree of ambiguity of the NFA is two.

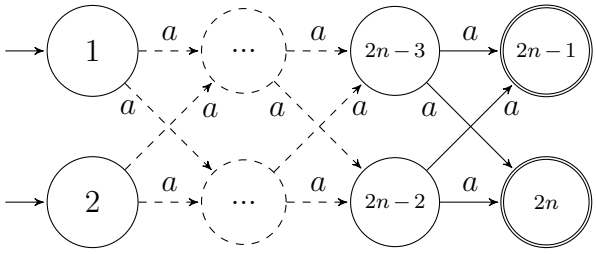


Figure 2.2: Family of FNFA where the degree of ambiguity grows exponentially with the size of the automata

Lemma 1. **[12]** The degree of an FNFA of size m is at most of the order $2^{O(m \times \log_2 m)}$.

In Figure 2.2 we can see an example of a family of FNFA diagram where the degree grows exponentially with the size of the automata.

An automaton of that family with $2n$ nodes, where $n > 1$, will have (finite) degree of ambiguity of $2^{n/2}$ and will accept the language a^{n-1}.

Definition 9. An NFA is *polynomially ambiguous* if there is a polynomial p such that, for each word w, the number of paths that accepts w is at most $p(|w|)$.

Figure 2.1c shows an example of a PNFA diagram where the word $a^i, i \geq 1$, has i accepting paths. An example of a polynomial that bounds the degree of ambiguity of the NFA mentioned is the identity polynomial $p(k) = k$.

Definition 10. An NFA is *exponentially ambiguous* if, for each word w, the number of paths that accept w is bounded by some exponential function on the size of w.

An example of an ENFA diagram is shown in Figure 2.1d where the degree of the word $a^{2(i+1)}$ has 2^i accepting paths.

2.2 Criteria for Ambiguity Classification

To classify an automaton there are three criteria that can be checked. A criterion to check if an automaton is ambiguous, the Infinite Degree of Ambiguity (IDA) criterion to check if the degree is finite and the Exponential Degree of Ambiguity (EDA) criterion to check if an automaton has exponential ambiguity. There is an algorithm to check each criterion and they can be computed in polynomial time, therefore the classification of the ambiguity of an NFA can be decided in polynomial time.

In this section we present the criteria above and the newly written code to test the criteria of NFA's. The newly written auxiliary functions used in the code can be consulted in the appendix, except for the functions *trim*, *dup* and *addTransition* that can be found in the documentation of FAdo [4].

2.2.1 Criterion for Exponential Degree of Ambiguity - EDA

An NFA is ENFA if and only if it complies with the following EDA criterion [12].

EDA: there exists a state q with at least two distinct cycles labeled by some $v \in \Sigma^*$.

The algorithm presented in Algorithm 2.1 is based on the following lemma.

Lemma 2. **[1]** An NFA M satisfies EDA if and only if there exists a strongly connected component of $M^2 = M \cap M$, obtained from the product, that contains two states of the form (p, p) and (q, q') with p, q, q' states of M and $q \neq q'$.

For an NFA M, the algorithm presented in Algorithm 2.1, is as follows. First, in lines 2-7, M is trimmed and M^2 is computed. Then we get the connected components of

M^2 (line 9), and, for each one, we check if there are two states $(s_1, s_2), (s'_1, s'_2) \in M^2$ such that $s_1 = s_2 \wedge s'_1 \neq s'_2$ (line 16).

```
1  def testEDA(nfax):
2      nfa = nfax.dup()
3      nfa.trim()
4
5      conj = myProduct(nfa,nfa)
6      addFinals(conj, nfa, nfa)
7      conj.trim()
8
9      comp = stronglyConnectedComponents(conj)
10     for l in comp:
11         for s1 in l:
12             for s2 in l:
13                 if s1==s2: continue
14                 (p1,p2) = literal_eval(conj.States[s1])
15                 (q1,q2) = literal_eval(conj.States[s2])
16                 if p1==p2 and q1!=q2:
17                     return True
18     return False
```

Algorithm 2.1: Algorithm to test EDA criterion

2.2.2 Criterion for Infinite Degree of Ambiguity - IDA

An NFA M is PNFA if and only if it complies with the following IDA criterion [12] and not satisfies the EDA criterion.

IDA: There are distinct useful states $p, q \in Q$ such that for some word $v \in \Sigma^*$ $(p, v, p), (p, v, q), (q, v, q) \in \delta$.

The algorithm presented in Algorithm 2.2 is based on the following lemma.

Lemma 3. [1] An NFA M satisfies IDA if and only if there exist two different states p and q in M with a path in $M^3 = M \cap M \cap M$, obtained from the product, from state (p, p, q) to state (p, q, q).

For an NFA M, the algorithm presented in Algorithm 2.2, is as follows. First, in lines 2-11, M is trimmed and M^3 is computed. Then, in lines 13-19, we add the transitions $((p, p, q), \#, (p, q, q))$ for all states p, q of M ($\#$ is a symbol that is not in the alphabet of M). Finally, we get the connected components of M^3 (line 21), and, for each one,

check if there is a transition by the new symbol # and by another in the original alphabet (lines 30-31).

```
1  def testIDA(nfax):
2      nfa = nfax.dup()
3      nfa.trim()
4
5      sz = len(nfa.States)
6
7      conj = myProduct(nfa,nfa)
8      addFinals(conj, nfa, nfa)
9
10     conj2 = myProduct(conj,nfa)
11     addFinals(conj2, conj, nfa)
12
13     for p in xrange(0,sz):
14         for q in xrange(0,sz):
15             if p==q: continue
16             a = (p*sz+p,q)
17             b = (p*sz+q,q)
18             conj2.addTransition(a[0]*sz+a[1],'#',b[0]*sz+b[1])
19     conj2.trim()
20
21     comp = stronglyConnectedComponents(conj2)
22     for l in comp:
23         tcard = False
24         tother = False
25         for i in l:
26             if i not in conj2.delta: continue
27             for symb in conj2.delta[i]:
28                 for t in conj2.delta[i][symb]:
29                     if t in l:
30                         if symb=='#': tcard = True
31                         else: tother = True
32                         if tcard and tother: return True
33
34     return False
```

Algorithm 2.2: Algorithm to test IDA criterion

2.2.3 Criterion for Having Ambiguity

An NFA M is ambiguous if and only if in $M \times M$ there is a useful state that is not in the diagonal [10].

An automaton is FNFA if and only if it is ambiguous and not satisfies the IDA criterion. If it is not ambiguous then it is UFA.

For an NFA M, the algorithm presented in Algorithm 2.3, is as follows. First, in lines 2-6, M is trimmed and M^2 is computed. Then for each useful state of M^2 we test if it is in the diagonal (lines 10 and 11).

```
1  def testAmbiguity(nfax):
2      nfa = nfax.dup()
3      nfa.trim()
4
5      conj = myProduct(nfa,nfa)
6      addFinals(conj, nfa, nfa)
7
8      useful = conj.usefulStates()
9      for i in useful:
10         (x,y) = literal_eval(conj.States[i])
11         if x != y:
12             return True
13
14     return False
```

Algorithm 2.3: Algorithm to test Ambiguity

Chapter 3

Results and their Discussion

In this chapter we present the results of the experiment performed in order to evaluate the incidence of each class of ambiguity in NFA'a obtained from converting regular expressions. We considered two standard methods: Glushkov (also known as Position) [7] and Antimirov (also known as Partial Derivatives or PD) [2].

To perform the experiment 10000 regular expressions were generated for each $n \in \{20, 35, 50, 75, 100\}$ and $k \in \{2, 5, 10\}$, where n is the number of symbols in the expression excluding parentheses and k is the size of the alphabet. Each regular expression was converted into two NFA's, one for each method of conversion.

The average size of the automata resulting from the conversion using the *Position* method is $n/2$ and using the *PD* method is $n/4$, where n is the size of the regular expression [5].

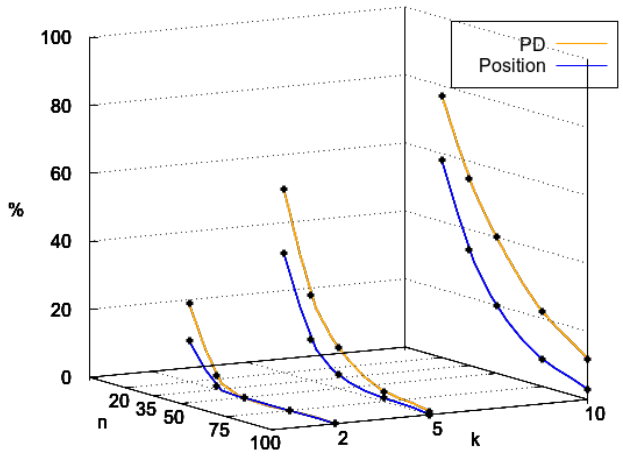
We classified the NFA's using the algorithms explained in the last chapter. The results are summarized in Figure 3.1.

In the graphics of Figure 3.1 we can see that both methods have similar distributions. That was expected because it was proven that the *PD* automaton is isomorphic to a quotient of *Position* automaton [8].

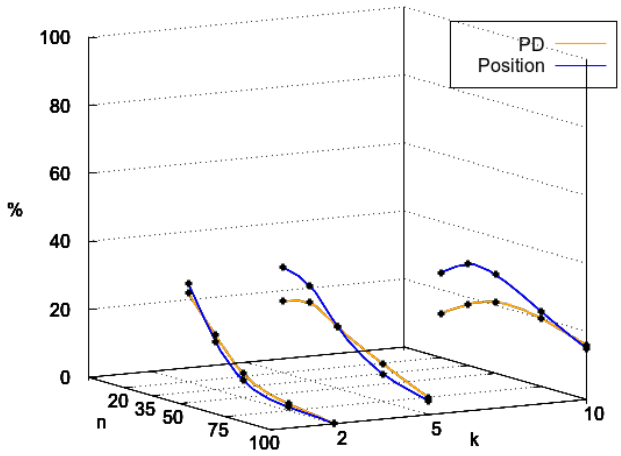
The number of *unambiguous automata* increases if the k is large or when the n is small.

In our results, PNFA is the class of ambiguity with the lowest presence.

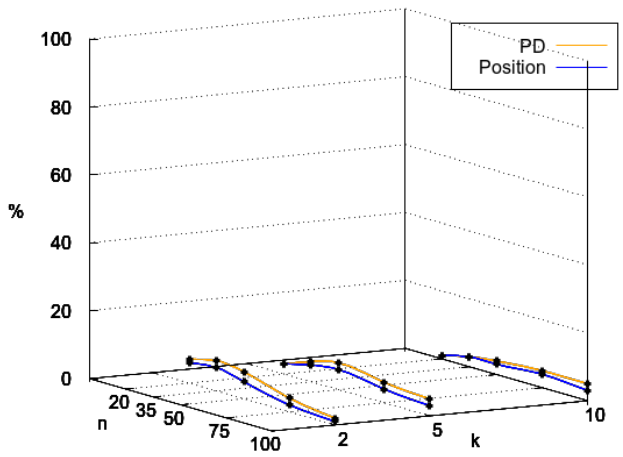
Contrary to the *unambiguous* automata, the number of *exponential ambiguous* automata increases if the n is large or when the k is small.



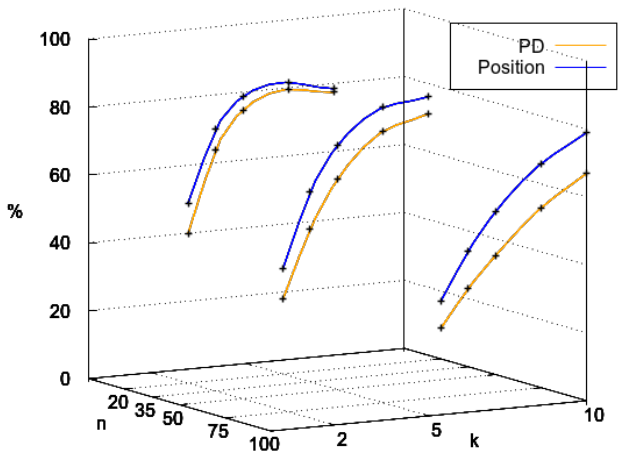
(a) Unambiguous Automata



(b) Finitely Ambiguous Automata



(c) Polinomially Ambiguous Automata



(d) Exponentially Ambiguous Automata

Figure 3.1: Results

Chapter 4

Conclusion

The ambiguity of an automaton is an important characteristic because if an NFA is *unambiguous* many decision problems that are only solvable in exponential time become solvable in polynomial time (e.g equivalence of two automata [11]).

There are well-defined algorithms to classify the ambiguity class of an automaton that was implemented and used in this work. Besides the algorithms presented, there is also one to determine the smallest degree of the polynomial that bounds the ambiguity degree of a PNFA automaton [12].

However, there is no algorithm to determine the degree of an FNFA automaton except for finding the degree by some kind of brute force exploration of all words. It is also PSPACE-complete to decide if an FNFA automaton has degree $k, k > 0$ [6].

Furthermore, the degree of ambiguity of an FNFA can be exponential in the size of the automaton [12]. An example of a family of FNFA where the degree grows exponentially with the size of the automaton was shown in Figure 2.2.

Finally, analyzing the results, we can see that most of the automata classified are *exponentially ambiguous*. Which means that the most common class of ambiguity for automata (if it is the result from the conversion of a regular expression) is ENFA, and, since the ENFA is the one where the number of paths growths faster (exponentially), this leads to a more inefficient analysis of automata.

In order to get a more clear understanding of the ambiguity of automata, it is necessary more work to find an algorithm to compute the degree of ambiguity of an FNFA. An interesting line of work would be also to try to decrease and remove the ambiguity of an automaton.

Appendix A

Code for Auxiliary Functions

```
def addFinals(conj, nfa1, nfa2):
    """Adds finals states to nfa1 x nfa2"""
    sz = len(nfa2.States)
    for x in [(a, b) for a in nfa1.Final for b in nfa2.Final]:
        if str(x) in conj.States:
            conj.addFinal(x[0]*sz+x[1])

def myProduct(nfa1, nfa2):
    """Computes nfa1 x nfa2"""
    nfa = NFA()

    sz1 = len(nfa1.States)
    sz2 = len(nfa2.States)

    #state (i,j) is in index i*sz2+j
    for i in xrange(0,sz1):
        for j in xrange(0,sz2): nfa.addState(str((i,j)))
    for i in nfa1.Initial:
        for j in nfa2.Initial: nfa.addInitial(i*sz2+j)

    for s1 in nfa1.delta:
        for symb in nfa1.delta[s1]:
            for s2 in nfa2.delta:
                if symb not in nfa2.delta[s2]: continue
                states = [i*sz2+j for i in nfa1.delta[s1][symb] for j in
    nfa2.delta[s2][symb]]
                for state in states: nfa.addTransition(s1*sz2+s2,symb,
   state)
    return nfa
```

```
28
29 def stronglyConnectedComponents(nfa):
30     """Computes the strongly connected components of nfa"""
31     def dfs(graph, node, vis, list):
32         stack = [node]
33         while len(stack) > 0:
34             n = stack.pop()
35             if vis[n] == 2: continue
36             if vis[n] == 1:
37                 list.append(n)
38                 vis[n] = 2
39                 continue
40
41             vis[n] = 1
42             stack.append(n)
43             for child in graph[n]:
44                 if not vis[child]: stack.append(child)
45
46     sz = len(nfa.States)
47     graph  = [set() for i in xrange(0,sz)]
48     graphT = [set() for i in xrange(0,sz)]
49
50     for i in nfa.delta:
51         for symb in nfa.delta[i]:
52             for j in nfa.delta[i][symb]:
53                 graph[i].add(j)
54                 graphT[j].add(i)
55
56     stack = []
57     vis = [0 for i in xrange(0,sz)]
58     for node in xrange(sz-1,-1,-1):
59         if not vis[node]: dfs(graph, node, vis, stack)
60
61     strongComp = []
62     vis = [0 for i in xrange(sz-1,-1,-1)]
63     while len(stack) > 0:
64         node = stack.pop()
65         if vis[node]: continue
66         comp = []
67         dfs(graphT, node, vis, comp)
68         strongComp.append(comp)
69
70     return strongComp
```

Algorithm A.1: Auxiliary functions

Bibliography

[1] Cyril Allauzen, Mehryar Mohri, and Ashish Rastogi. General algorithms for testing the ambiguity of finite automata and the double-tape ambiguity of finite-state transducers. *Int. J. Found. Comput. Sci.*, 22(4):883–904, 2011.

[2] Valentin M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.*, 155(2):291–319, 1996.

[3] Angelo Borsotti, Luca Breveglieri, Stefano Crespi-Reghizzi, and Angelo Morzenti. From ambiguous regular expressions to deterministic parsing automata. In *Implementation and Application of Automata - 20th International Conference, CIAA 2015, Umeå, Sweden, August 18-21, 2015, Proceedings*, pages 35–48, 2015.

[4] Sabine Broda, Markus Holzer, Eva Maia, Nelma Moreira, and Rogério Reis. A mesh of automata. *Information and Computation*, 265:94 – 111, 2019.

[5] Sabine Broda, António Machiavelo, Nelma Moreira, and Rogério Reis. On the average state complexity of partial derivative automata: an analytic combinatorics approach. *Int. J. Found. Comput. Sci.*, 22(7):1593–1606, 2011.

[6] Tat-hung Chan and Oscar H. Ibarra. On the finite-valuedness problem for sequential machines. *Theor. Comput. Sci.*, 23:95–101, 1983.

[7] Victor M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16(5):1–53, 1961.

[8] Lucian Ilie and Sheng Yu. Follow automata. *Inf. Comput.*, 186(1):140–162, 2003.

[9] Hing Leung. Descriptional complexity of nfa of different ambiguity. *Int. J. Found. Comput. Sci.*, 16(5):975–984, 2005.

[10] Jacques Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, 2009.

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

[12] Andreas Weber and Helmut Seidl. On the degree of ambiguity of finite automata. *Theor. Comput. Sci.*, 88(2):325–349, 1991.