

Chapter 2

State Merging Algorithms

2.1 Preliminaries

Before we start analyzing how the state merging algorithms work, some basic functions on automata as well as functions on the sets of words have to be defined. We assume that below given routines are available throughout the whole book. Please refer to Appendixes A, B, and C in order to familiarize with the Python programming language, its packages relevant to automata, grammars, and regexes, and some combinatorial optimization tools. Please notice also that we follow the docstring convention. A docstring is a string literal that occurs as the first statement in a function (module, class, or method definition). Such string literals act as documentation.

```
from FAdo.fa import *

def alphabet(S):
    """Finds all letters in S
    Input: a set of strings: S
    Output: the alphabet of S"""
    result = set()
    for s in S:
        for a in s:
            result.add(a)
    return result

def prefixes(S):
    """Finds all prefixes in S
    Input: a set of strings: S
    Output: the set of all prefixes of S"""
    result = set()
    for s in S:
        for i in xrange(len(s) + 1):
            result.add(s[:i])
    return result

def suffixes(S):
    """Finds all suffixes in S
    Input: a set of strings: S
    Output: the set of all suffixes of S"""
```

```

result = set()
for s in S:
    for i in xrange(len(s) + 1):
        result.add(s[i:])
return result

def catenate(A, B):
    """Determine the concatenation of two sets of words
    Input: two sets (or lists) of strings: A, B
    Output: the set AB"""
    return set(a+b for a in A for b in B)

def ql(S):
    """Returns the list of S in quasi-lexicographic order
    Input: collection of strings
    Output: a sorted list"""
    return sorted(S, key = lambda x: (len(x), x))

def buildPTA(S):
    """Build a prefix tree acceptor from examples
    Input: the set of strings, S
    Output: a DFA representing PTA"""
    A = DFA()
    q = dict()
    for u in prefixes(S):
        q[u] = A.addState(u)
    for w in iter(q):
        u, a = w[:-1], w[-1:]
        if a != '':
            A.addTransition(q[u], a, q[w])
        if w in S:
            A.addFinal(q[w])
    A.setInitial(q[''])
    return A

def merge(q1, q2, A):
    """Join two states, i.e., q2 is absorbed by q1
    Input: q1, q2 state indexes and an NFA A
    Output: the NFA A updated"""
    n = len(A.States)
    for q in xrange(n):
        if q in A.delta:
            for a in A.delta[q]:
                if q2 in A.delta[q][a]: A.addTransition(q, a, q1)
        if q2 in A.delta:
            for a in A.delta[q2]:
                if q in A.delta[q2][a]: A.addTransition(q1, a, q)
    if q2 in A.Initial: A.addInitial(q1)
    if q2 in A.Final: A.addFinal(q1)
    A.deleteStates([q2])
    return A

def accepts(w, q, A):
    """Verify if in an NFA A, a state q recognizes given word
    Input: a string w, a state index (int) q, and an NFA A
    Output: yes or no as Boolean value"""
    ilist = A.epsilonClosure(q)
    for c in w:

```

Fig. 2.1 A PTA accepting aa , aba , and bba

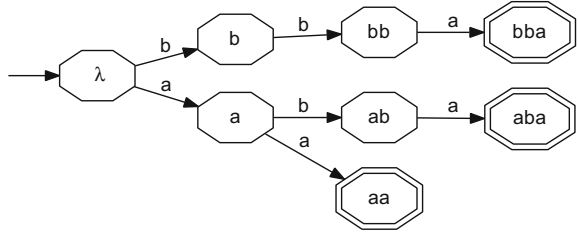
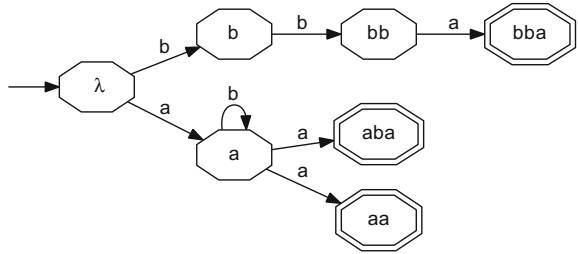


Fig. 2.2 An NFA after merging a and ab in PTA



```

ilist = A.evalSymbol(ilist, c)
if not ilist:
    return False
return not A.Final.isdisjoint(ilist)

```

There are two fundamental functions that are present in every state merging algorithms given in this book: `buildPTA` for constructing a prefix tree acceptor and `merge` which performs the merging operation.

Definition 2.1 A *prefix tree acceptor* (PTA) is a tree-like DFA built from the learning examples S by taking all the prefixes in the examples as states and constructing the smallest DFA A which is a tree which holds $L(A) = S$. The initial state is a root and all remaining states q have exactly one ingoing edge, i.e., $|\{q' : q \in \delta(q', a)\}| = 1$.

An exemplary PTA is depicted in Fig. 2.1.

The merging operation takes two states from an NFA and joins them into a single state. As we can see from the definition of the `merge` function, the new state (which inherits a label after `q1`) shares the properties, as well as ingoing and outgoing arcs of both states that have been merged. Consider for instance automaton from Fig. 2.1. If states a and ab are merged, resulting automaton is as in Fig. 2.2.

It should be noted that after this operation the PTA lost the determinism property and—what is more attractive—the new automaton represents an infinite language.

2.2 Evidence Driven State Merging

The idea behind this algorithm is fairly straightforward. Given a sample, we start from building a PTA based on examples, then iteratively select two states and do

merging unless compatibility is broken. A heuristic for choosing the pair of states to merge, can be realized in many ways. We propose the following procedure. A score is given to each state pair, and the state pair with the best score is chosen. In order to explain the score in the simplest way (and for further investigations in the present book), we ought to define the right and the left languages of a state q .

Definition 2.2 For the state $q \in Q$ of an NFA $A = (Q, \Sigma, \delta, s, F)$ we consider the two languages:

$$\vec{L}(q) = \{w \in \Sigma^*: \delta(q, w) \cap F \neq \emptyset\}, \quad \overleftarrow{L}(q) = \{w \in \Sigma^*: q \in \delta(s, w)\}.$$

Thus, the right language of a state q , $\vec{L}(q)$, is the set of all words spelled out on paths from q to a final state, whereas the left language of a state q , $\overleftarrow{L}(q)$, is the set of all words spelled out on paths from the initial state s to q .

Now we can define the score of two states $q, r \in Q$ for an NFA A and the set U of the suffixes of S_+ :

$$\text{score}(q, r) = |U \cap \vec{L}(q) \cap \overleftarrow{L}(r)|.$$

Finally, we have got the following form of the EDSM algorithm:

```
def makeCandidateStatesList(U, A):
    """Build the sorted list of pairs of states to merge
    Input: a set of suffixes, U, and an NFA, A
    Output: a list of pairs of states, first most promising"""
    n = len(A.States)
    score = dict()
    langs = []
    pairs = []
    for i in xrange(n):
        langs.append(set(u for u in U if accepts(u, i, A)))
    for i in xrange(n-1):
        for j in xrange(i+1, n):
            score[i, j] = len(langs[i] & langs[j])
            pairs.append((i, j))
    pairs.sort(key = lambda x: -score[x])
    return pairs

def synthesize(S_plus, S_minus):
    """Infers an NFA consistent with the sample
    Input: the sets of examples and counter-examples
    Output: an NFA"""
    A = buildPTA(S_plus).toNFA()
    U = suffixes(S_plus)
    joined = True
    while joined:
        pairs = makeCandidateStatesList(U, A)
        joined = False
        for (p, q) in pairs:
            B = A.dup()
            merge(p, q, B)
            if not any(B.evalWordP(w) for w in S_minus):
```

```

    A = B
    joined = True
    break
return A

```

2.3 Gold's Idea

The central structure of the present algorithm is a table, which during the run is expanded vertically. Its columns are indexed by all suffixes (called EXP) of a sample $\Sigma^* \supset S = (S_+, S_-)$ and its rows are indexed by a prefixed closed set starting from the set $\{\lambda\} \cup \Sigma$ (as usual Σ denotes an alphabet). The rows of the table correspond to the states of the final deterministic automaton A . The indexes (words) of the rows are divided into two sets: RED and BLUE. The RED indexes correspond to states that have been analyzed and which will not be revisited. The BLUE indexes are the candidate states: they have not been analyzed yet and it should be from this set that a state is drawn in order to consider merging it with a RED state. There are three types of entries in the table, which we will call observation table (OT). $OT[u, e] = 1$ if $ue \in L(A)$; $OT[u, e] = 0$ if $ue \notin L(A)$; and $OT[u, e] = *$ otherwise (not known). The sign $*$ is called a hole and corresponds to a missing observation.

Definition 2.3 Rows indexed by u and v are *obviously different* (OD) for OT if there exists such $e \in EXP$ that $OT[u, e], OT[v, e] \in \{0, 1\}$ and $OT[u, e] \neq OT[v, e]$.

Definition 2.4 An observation table OT is *complete* (or has no holes) if for every $u \in RED \cup BLUE$ and for every $e \in EXP$, $OT[u, e] \in \{0, 1\}$.

Definition 2.5 A table OT is *closed* if for every $u \in BLUE$ there exists $s \in RED$ such that for every $e \in EXP$ $OT[u, e] = OT[s, e]$.

The algorithm is divided into four phases. We will illustrate its run by an example. Let $S = (\{\lambda, ab, abab\}, \{a, b, aa, ba, bb, aab, bab, bbb\})$ be a sample for which we want to find a consistent DFA.

Building a table from the data

```

def buildTable(S_plus, S_minus):
    """Builds an initial observation table
    Input: a sample
    Output: OT as dictionary and sets: Red, Blue, EXP"""
    OT = dict()
    EXP = suffixes(S_plus | S_minus)
    Red = {''}
    Blue = alphabet(S_plus | S_minus)
    for p in Red | Blue:
        for e in EXP:
            if p+e in S_plus:
                OT[p, e] = 1
            else:
                OT[p, e] = 0 if p+e in S_minus else '*'
    return (Red, Blue, EXP, OT)

```

This phase is easy. For a sample S we get the following table:

	' '	a	b	aa	ab	ba	bb	aab	bab	bbb	abab
' '	1	0	0	0	1	0	0	0	0	0	1
a	0	0	1	*	0	*	*	*	1	*	*
b	0	0	0	*	0	*	0	*	*	*	*

Throughout the rest of the example run, RED rows occupy the upper part, while BLUE rows occupy the lower part of the table.

Updating the table

This phase is performed through the while loop that we can see in the `synthesize` function given at the end. The aim of this phase is to bring about the table to be closed. To this end, every BLUE word (index) b that is obviously different from all RED words becomes RED and rows indexed by ba , for $a \in \Sigma$, are added to the BLUE part of the table. In the example, this operation has been repeated two times (for a and then for b):

	' '	a	b	aa	ab	ba	bb	aab	bab	bbb	abab
' '	1	0	0	0	1	0	0	0	0	0	1
a	0	0	1	*	0	*	*	*	1	*	*
b	0	0	0	*	0	*	0	*	*	*	*
aa	0	*	0	*	*	*	*	*	*	*	*
ab	1	*	*	*	1	*	*	*	*	*	*

	' '	a	b	aa	ab	ba	bb	aab	bab	bbb	abab
' '	1	0	0	0	1	0	0	0	0	0	1
a	0	0	1	*	0	*	*	*	1	*	*
b	0	0	0	*	0	*	0	*	*	*	*
aa	0	*	0	*	*	*	*	*	*	*	*
ab	1	*	*	*	1	*	*	*	*	*	*
ba	0	*	0	*	*	*	*	*	*	*	*
bb	0	*	0	*	*	*	*	*	*	*	*

Filling in the holes

Now, the table is closed. The next phase is in order to make the table complete.

```

def fillHoles(Red, Blue, EXP, OT):
    """Tries to fill in holes in OT
    Input: rows (Red + Blue), columns (EXP), and table (OT)
    Output: true if success or false if fail"""
    for b in ql(Blue):
        found = False
        for r in ql(Red):
            if not any(OT[r, e] == 0 and OT[b, e] == 1 \
                or OT[r, e] == 1 and OT[b, e] == 0 for e in EXP):
                found = True
                for e in EXP:
                    if OT[b, e] != '*':
                        OT[r, e] = OT[b, e]
        if not found:
            return False
    for r in Red:
        for e in EXP:
            if OT[r, e] == '*':
                OT[r, e] = 1
    for b in ql(Blue):
        found = False
        for r in ql(Red):
            if not any(OT[r, e] == 0 and OT[b, e] == 1 \
                or OT[r, e] == 1 and OT[b, e] == 0 for e in EXP):
                found = True
                for e in EXP:
                    if OT[b, e] == '*':
                        OT[b, e] = OT[r, e]
        if not found:
            return False
    return True

```

This routine, first fills the rows corresponding to the RED states by using the information included in the BLUE rows which do not cause any conflict. In our example, the table has not been changed. Then all the holes in the RED rows are filled by 1s:

	' '	a	b	aa	ab	ba	bb	aab	bab	bbb	abab
' '	1	0	0	0	1	0	0	0	0	0	1
a	0	0	1	1	0	1	1	1	1	1	1
b	0	0	0	1	0	1	0	1	1	1	1
aa	0	*	0	*	*	*	*	*	*	*	*
ab	1	*	*	*	1	*	*	*	*	*	*
ba	0	*	0	*	*	*	*	*	*	*	*
bb	0	*	0	*	*	*	*	*	*	*	*

Finally, the routine again visits BLUE rows, tries to find a compatible RED row and copies the corresponding entries. This results in the following table:

	' '	a	b	aa	ab	ba	bb	aab	bab	bbb	abab
' '	1	0	0	0	1	0	0	0	0	0	1
a	0	0	1	1	0	1	1	1	1	1	1
b	0	0	0	1	0	1	0	1	1	1	1
aa	0	0	0	1	0	1	0	1	1	1	1
ab	1	0	0	0	1	0	0	0	0	0	1
ba	0	0	0	1	0	1	0	1	1	1	1
bb	0	0	0	1	0	1	0	1	1	1	1

Building a DFA from a complete and closed table

```

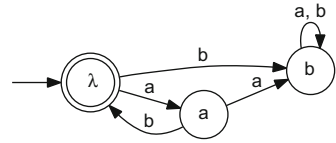
def buildAutomaton(Red, Blue, EXP, OT):
    """Builds a DFA from closed and complete observation table
    Input: rows (Red + Blue), columns (EXP), and table (OT)
    Output: a DFA"""
    A = DFA()
    A.setSigma(alphabet(Red | Blue | EXP))
    q = dict()
    for r in Red:
        q[r] = A.addState(r)
    for w in Red | Blue:
        for e in EXP:
            if w+e in Red and OT[w, e] == 1:
                A.addFinal(q[w+e])
    for w in iter(q):
        for u in iter(q):
            for a in A.Sigma:
                if all(OT[u, e] == OT[w+a, e] for e in EXP):
                    A.addTransition(q[w], a, q[u])
    A.setInitial(q[''])
    return A

def OD(u, v, EXP, OT):
    """Checks if rows u and v obviously different for OT
    Input: two rows (prefixes), columns, and table
    Output: boolean answer"""
    return any(OT[u, e] in {0, 1} and OT[v, e] in {0, 1} \
        and OT[u, e] != OT[v, e] for e in EXP)

def synthesize(S_plus, S_minus):
    """Infers a DFA consistent with the sample
    Input: the sets of examples and counter-examples
    Output: a DFA"""
    (Red, Blue, EXP, OT) = buildTable(S_plus, S_minus)
    Sigma = alphabet(S_plus | S_minus)
    x = ql(b for b in Blue if all(OD(b, r, EXP, OT) for r in Red))
    while x:
        Red.add(x[0])
        Blue.discard(x[0])
        Blue.update(catenate({x[0]}, Sigma))
    for u in Blue:
        for e in EXP:
            if u+e in S_plus:
                OT[u, e] = 1

```


Fig. 2.3 The result



```

else:
    OT[u, e] = 0 if u+e in S_minus else '*'
    x = ql(b for b in Blue if all(OD(b, r, EXP, OT) for r in Red))
if not fillHoles(Red, Blue, EXP, OT):
    return buildPTA(S_plus)
else:
    A = buildAutomaton(Red, Blue, EXP, OT)
    if all(A.evalWordP(w) for w in S_plus) \
        and not any(A.evalWordP(w) for w in S_minus):
        return A
    else:
        return buildPTA(S_plus)

```

In the last phase, the table is transformed into a DFA. RED words constitute the set of states, whereas the transition function is defined using the entries. The algorithm returns a DFA depicted in Fig. 2.3:

```

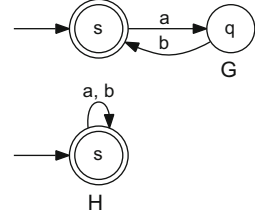
>>> A = synthesize({"", "ab", "abab"}, \
...   {"a", "b", "aa", "ba", "bb", "aab", "bab", "bbb"})
>>> print A.dotFormat()
digraph finite_state_machine {
    node [shape = doublecircle]; "";
    node [shape = circle]; "a";
    node [shape = circle]; "b";
    "" --> "a" [label = "a"];
    "" --> "b" [label = "b"];
    "a" --> "" [label = "b"];
    "a" --> "b" [label = "a"];
    "b" --> "b" [label = "a, b"];
}

```

2.4 Grammatical Inference with MDL Principle

The minimum description length (MDL) principle is a rule of thumb in which the best hypothesis for a given set of data is the one that leads to the best compression of the data. Generally, searching for small acceptor compatible with examples and counter-examples is a good idea in grammatical inference. MDL principle is the development of this line of reasoning in case of the absence of counter-examples.

Fig. 2.4 Two automata compatible with a sample S



2.4.1 The Motivation and Appropriate Measures

Suppose that we are given the sample $S = \{\lambda, ab, abab, ababab\}$. Let us try to answer the question what makes an automaton G better than H (see Fig. 2.4) in describing the language represented by S . The idea is to measure an automaton along with the size of encoding all words of the sample. In this way, not always is the smallest automaton recognized as the most promising. The process of parsing is also at stake here.

Let $A = (Q, \Sigma, \delta, s, F)$ be an NFA all of whose non-final states have outgoing edges. The number of bits required to encode the path followed to parse a word w can be assessed by the below given function ch . We associate with each state $q \in Q$ the value $t_q = \sum_{a \in \Sigma} |\delta(q, a)|$ if $q \notin F$. If $q \in F$ then $t_q = 1 + \sum_{a \in \Sigma} |\delta(q, a)|$, since one more choice is available. We are now in a position to define $ch(q, w)$. For the empty word we have $ch(q, \lambda) = \log(t_q)$ if $q \in F$; otherwise $ch(q, \lambda) = \infty$. For $w = au$ ($a \in \Sigma, u \in \Sigma^*$) ch depends on the recursive definition: $ch(q, w) = \log(t_q) + \min_{r \in \delta(q, a)} ch(r, u)$ and $ch(q, w) = \infty$ if $\delta(q, a) = \emptyset$.

We can now, given a sample S and an NFA A , measure the score sc of A :

$$sc(A, S) = |Q| + \|\delta\|(2 \log |Q| + \log |\Sigma|) + \sum_{w \in S} ch(s, w),$$

where $\|\delta\|$ is the number of transitions of A .

Coming back to automata G, H and a sample S , the exact computations of the scores can be found below. Notice that $t_s = 2$ and $t_q = 1$ for G , while $t_s = 3$ for H . Thus, we have $sc(G, S) = 2 + 2(2 \log(2) + \log(2)) + 10 \log(2) = 18.0$ and $sc(H, S) = 1 + 2(2 \log(1) + \log(2)) + 16 \log(3) = 28.36$. Therefore, we need more space to encode both the automaton and the data in case of H and S than in case of G and S .

2.4.2 The Proposed Algorithm

The idea is as follows. We start from the PTA and iteratively merge a pair of states as long as this operation decreases the score. The order in which states are merged can be scheduled in many ways. We base on the quasi-lexicographic order of the labels of

states. After building the PTA, every state in an automaton has a label corresponding to the path from the initial state to that state.

```

from math import log
from FAdo.fa import *

def sc(A, S):
    """Measures the score of an NFA A and words S
    Input: an automaton A and the set of words S
    Output: a float"""

    @memoize
    def ch(i, w):
        """Calculates the size of encoding of the path
        followed to parse word w from the ith state in A
        Input: state's index, word
        Output: a float"""
        if w == '':
            return log(t[i], 2) if i in A.Final else float('inf')
        else:
            if i in A.delta and w[0] in A.delta[i]:
                return log(t[i], 2) + min(ch(j, w[1:]) \
                    for j in A.delta[i][w[0]])
            else:
                return float('inf')

    s = list(A.Initial)[0]
    t = dict()
    for i in xrange(len(A.States)):
        t[i] = 1 if i in A.Final else 0
        if i in A.delta:
            t[i] += sum(map(len, A.delta[i].intervalvalues()))
    return len(A.States) + sum(ch(s, w) for w in S) \
        + A.countTransitions()*(2*log(len(A.States), 2) \
        + log(len(A.Sigma), 2))

def synthesize(S):
    """Finds a consistent NFA by means of the MDL principle
    Input: set of positive words
    Output: an NFA"""
    A = buildPTA(S).toNFA()
    Red = {''}
    Blue = set(A.States)
    Blue.remove('')
    current_score = sc(A, S)
    while Blue:
        b = ql(Blue)[0]
        Blue.remove(b)
        for r in ql(Red):
            M = A.dup()
            merge(M.States.index(r), M.States.index(b), M)
            new_score = sc(M, S)
            if new_score < current_score:
                A = M
                current_score = new_score
            break
        if b in A.States:
            Red.add(b)
    return A

```

2.5 Bibliographical Background

The evidence driven state merging algorithm given in Sect. 2.2 is based in part on concepts published in Coste and Fredouille (2003). Gold’s algorithm (Gold 1978) was the first GI algorithm with convergence properties. We have presented the algorithm according to its description in de la Higuera (2010). The MDL principle in the context of deterministic automata was described in de la Higuera (2010). We have adapted it to a non-deterministic output in Sect. 2.4. More thorough theoretical investigations in this regard were carried by Adriaans and Jacobs (2006) for DFAs, and by Petasis et al. (2004) for CFGs.

It is also worth to note three state of the art tools for heuristic state-merging DFA induction: the Trakhtenbrot-Barzdin state merging algorithm (denoted Traxbar) adapted by Lang (1992), Rodney Price’s Abbadingo winning idea of evidence-driven state merging (Blue-fringe) described by Lang et al. (1998), and Rlb state merging algorithm (Lang 1997).

Trakhtenbrot and Barzdin (1973) described an algorithm for constructing the smallest DFA consistent with a complete labeled training set. The input to the algorithm is the PTA. This tree is squeezed into a smaller graph by merging all pairs of states that represent compatible mappings from word suffixes to labels. This algorithm for completely labeled trees was generalized by Lang (1992) to produce a (not necessarily minimum) machine consistent with a sparsely labeled tree.¹

The second algorithm that starts with the PTA and folds it up into a compact hypothesis by merging pairs of states is Blue-fringe. This program grows a connected set of red nodes that are known to be unique states, surrounded by a fringe of blue nodes that will either be merged with red nodes or be promoted to red status. Merges only occur between red nodes and blue nodes. Blue nodes are known to be the roots of trees, which greatly simplifies the code for correct merging. The only drawback of this approach is that the pool of possible merges is small, so occasionally the program has to do a low scoring merge.

The idea that lies behind the third algorithm, Rlb, is as follows. It dispenses with the red-blue restriction and is able to do merges in any order. However, to have a practical run time, only merges between nodes that lie within a distance ‘window’ of the root on a breadth-first traversal of the hypothesis graph are considered. This introduction of a new parameter is a drawback to this program, as is the fact that its run time scales very badly with training string length. However, on suitable problems, it works better than the Blue-fringe algorithm. In Lang (1997) one can find the detailed description of heuristics for evaluating and performing merges.

¹The reader can use implementations from the archive <http://abbadingo.cs.nuim.ie/dfa-algorithms.tar.gz> for the Traxbar and for the two remaining state-merging algorithms.

References

- Adriaans PW, Jacobs C (2006) Using MDL for grammar induction. In: *Proceedings of grammatical inference: algorithms and applications*, 8th international colloquium, ICGI 2006. Tokyo, Japan, 20–22 Sept 2006, pp 293–306
- Coste F, Fredouille D (2003) Unambiguous automata inference by means of state-merging methods. In: *Proceedings of machine learning: ECML 2003*, 14th European conference on machine learning. Cavtat-Dubrovnik, Croatia, 22–26 Sept 2003, pp 60–71
- de la Higuera C (2010) *Grammatical inference: learning automata and grammars*. Cambridge University Press, New York, NY, USA
- Gold EM (1978) Complexity of automaton identification from given data. *Inf Control* 37:302–320
- Lang KJ (1992) Random DFA's can be approximately learned from sparse uniform examples. In: *Proceedings of the fifth annual workshop on computational learning theory*. ACM, pp 45–52
- Lang KJ (1997) Merge order count. Technical report, NECI
- Lang KJ, Pearlmutter BA, Price RA (1998) Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In: *Proceedings of the 4th international colloquium on grammatical inference*. Springer, pp 1–12
- Petasis G, Paliouras G, Karkaletsis V, Halatsis C, Spyropoulos CD (2004) E-GRIDS: computationally efficient grammatical inference from positive examples. *Grammars* 7:69–110
- Trakhtenbrot B, Barzdin Y (1973) *Finite automata: behavior and synthesis*. North-Holland Publishing Company

Grammatical Inference
Algorithms, Routines and Applications
Wieczorek, W.
2017, XI, 145 p. 18 illus., Hardcover
ISBN: 978-3-319-46800-6