# Seminarpaper

Mario Wagner, 0730223

Graz, am October 7, 2016

## Contents



## List of Figures

# List of Tables

# 1 Introduction

Supervised learning is the task of inferring knowledge from a fully labeled training data. In contrast, unsupervised learning is the task of inferring knowledge from an unlabeled dataset. The common ground of supervised and unsupervised learning, is the task of inferring knowledge from a small amount of labeled data and a huge amount of unlabeled training data; i.e., *semi-supervised learning*. In other words, semi-supervised learning is making use of that small amount of data (i.e., supervised learning) to do unsupervised learning considerably more accurate.

*Active learning* is a special case of semi-supervised learning in which the labeled data is incrementally provided by a continuous interaction between the learner and an *Oracle*. The oracle can be a user, a system, or a teacher that provides a certain level of insight to supervise the learner by letting it to actively query for labels of unknown data. When the oracle is a system, actively querying it only means the learner is iteratively providing the system with input alphabet and asking for the output alphabet.

This method is commonly used to learn models of reactive systems by actively providing them with external events (i.e., inputs) and observe how they react to them (i.e., outputs). Though, sometimes active learning can be as expensive as manual labeling since not all runs of all systems are inexpensive or repeatable; to exemplify, consider a scenario in which an input sequence triggers a safety critical behaviour of the system (e.g. self-termination).

[* ME: here goes a brief history about the $L^*$ algorithm of Angluin (2 or 3 paragraphs). *]

Although $L^*$ is a quite useful automata learning approach, there are also issues with it; first of which is while using $L^*$ the learner either learns the correct system or nothing. Next issue with $L^*$ is actually stated earlier, that is, when it is impossible for some systems to perform some queries for various reasons (e.g. not being cost-effective, safety critical behaviour, expensive overheads, and etc.). Because of these reasons and many others that we may not know beforehand, passive learning (i.e., unsupervised learning) is used to learn the somewhat correct model of the system under learn. AALERGIA[1] is an interesting approach to learn stochastic automata from a data-set of traces of a system.

[* ME: here you have to reword parts of introduction of AALERGIA [1] that are relate to the incentives of devising such method; two or three paragraphs suffice. *]

# 2 Preliminaries

## 2.1 Markov Models

### 2.1.1 Markov Decision Procedure

### 2.1.2 Deterministic Labeled Markov chain

### 2.1.3 Discrete-time Markov chain

## 2.2 Prism Model Checker

### 2.2.1 Probabilistic Linear Temporal Logic

[* ME: That's on ME. *]

### 2.2.2 Dice Programs

The first and simpler example we are going to use is a PRISM model of a simple probabilistic algorithm, the dice model[2]. Figure 1 shows the states and the probability of the state transitions. We start at an initial state 0 and we toss a coin at each step, giving us a 50 percent chance of transitioning to each of the two following states. The algorithm ends at the leafs of the tree, which represent the values of the dice (d1-d6).

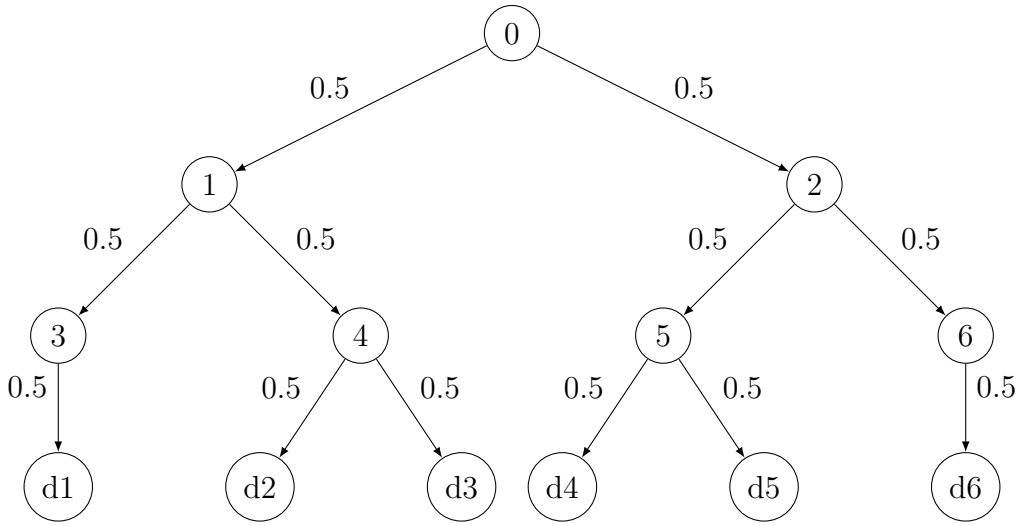


Figure 1: The Dice Example

The source code of the dice PRISM model is shown in figure 2. At the beginning, the code indicates that it is describing a discrete-time Markov chain (DTMC). The source code shows a single PRISM module called dice. The variable $s$, which represents the state of the system, can take the values from 0 to 7 and is initialised with the value 0 (line 6). The variable $d$, which represents the value of the dice, can range from 0 to 6 and is also initialised with the value 0, meaning that the dice has no value yet (line 8).

Each line depicts a state and the probabilities of the transitions, e.g. line 11 shows there is a probability of 0.5 of transitioning to state 3 or state 4 if we are currently in state 1. The model ends if a value is assigned to the dice, as there are no further transitions from that point on.

```
1  dtmc
2
3  module dice
4
5    // local state
6    s : [0..7] init 0;
7    // value of the dice
8    d : [0..6] init 0;
9
10   [] s=0 -> 0.5 : (s'=1) + 0.5 : (s'=2);
11   [] s=1 -> 0.5 : (s'=3) + 0.5 : (s'=4);
12   [] s=2 -> 0.5 : (s'=5) + 0.5 : (s'=6);
13   [] s=3 -> 0.5 : (s'=1) + 0.5 : (s'=7) & (d'=1);
14   [] s=4 -> 0.5 : (s'=7) & (d'=2) + 0.5 : (s'=7) & (d'=3);
15   [] s=5 -> 0.5 : (s'=7) & (d'=4) + 0.5 : (s'=7) & (d'=5);
16   [] s=6 -> 0.5 : (s'=2) + 0.5 : (s'=7) & (d'=6);
17   [] s=7 -> (s'=7);
18
19 endmodule
```

Figure 2: The Prism Dice Model

### 2.2.3 Herman's Self-Stabilisation Algorithm

[* ME: (1) describe the problem; (2) how they handled the problem with prism model checker? (3) what are the findings? and be specific that we are going to use this case study from this point onward for the learning algorithm; otherwise we state that we are referring to the dice programs. *]

## 2.3 AALERGIA

[* ME: (1) Who is responsible for ALERGIA; (2) Why they implemented this tool? (3) What are the issues with the tool? *] [* ME: (1) Who is responsible for AALERGIA; (2) How they addressed the issues with ALERGIA in this novel implementation? (3) Is there any room for improvement? (The answer is yes and you will refer to the PAutomataC contest + related work) *]

In this paper, the algorithm AALERGIA[1] and the provided MATLAB implementation[1] are used as a starting point for our work. In order to get a better idea of the way the implementation works, we will describe it in detail and a running example for

[1] http://mi.cs.aau.dk/code/aalergia

better replicability is used throughout this section. The implementation of the algorithm consists of several parts, which are depicted in the figure 3.


Figure 3: The Flow Diagram AALERGIA

### 2.3.1 Training Data

The data used to learn the Deterministic Labeled Markov Chain (DLMC) is supplied in the form of a comma separated values file, which contains the training set and the alphabet to be used. The alphabet consists of a finite 1×N-cell array, where each column contains one letter in the alphabet. The training set consist of a finite N×1-cell array, where each row contains sequences of symbols generated from the model, separated by commas.

**Dataset Generator** [* ME: we need to talk about how we are capable of generating dataset from a prism model. *]

The running example we are going to use is a self-stabilizing ring network with 3 processes. Table 1 shows the beginning of the trainings set, which contains 50 Sequences in total. In our case, each symbol in the training set represents the states in the ring at a given moment. For example, the symbol 001 shows that process 3 is in the state 1 and process 1 and 2 are in the state 0.

| |
|---|
| 000,001, |
| 000,011,100,010,101,010,001,100,010,101, |
| 000,101, |
| 000,111,010,001,110,011,101,010,001,110,011, |
| 000,111,011,101,110,001, |
| 000,101,010,101,110,011,100,011,101,110, |

Table 1: The Trainings set

Table 2 shows the alphabet, which consists of 8 symbols from 000 to 111.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |

Table 2: The Alphabet

### 2.3.2 Frequency Prefix Tree Acceptor

After loading the trainings set and the alphabet into the workspace, the Frequency Prefix Tree Acceptor (FPTA) is created. A deterministic frequency finite automaton (DFFA) is a tuple $A = \langle \Sigma, Q, I_{\text{fr}}, F_{\text{fr}}, \delta_{\text{fr}}, \delta \rangle$ where $\Sigma$ = the finite alphabet, $Q$ = the finite set of states, $I_{\text{fr}}$ = the initial state frequencies, $F_{\text{fr}}$ = the final state frequencies, $\delta_{\text{fr}}$ = the frequency transition function and $\delta$ = the transition function.

Figure 4 shows how a simplified FPTA for the in section 2.2.2 described dice model looks like. The sample S is a multiset of size 20. S = {(SH,1), (SHHH,1), (SHHH$D_1$,1), (SHH$D_1$,2), (SHH$D_1D_1$,1), (SHT,2), (SHT$D_2$,1), (SHT$D_2D_2$,1), (SHT$D_3$,1), (ST,1), (STH,2), (STH$D_4$,1), (STH$D_5$,1), (STT,2), (STT$D_6$,2)}. The letter H stands for heads, the letter T stands for tails and D1 - D6 stands for the different dice values. The numbers in the state indicates the number of the occurrence of the event, whereas the number on the transitions illustrates the count of all events below it.

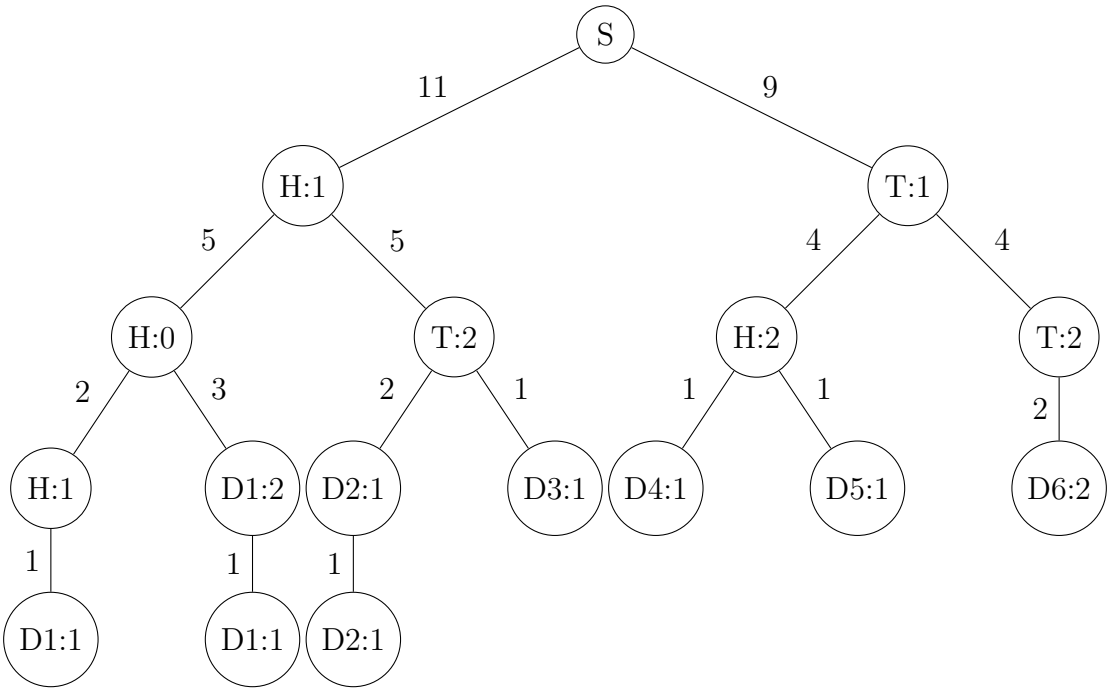


Figure 4: The FPTA for the DICE model

Algorithm 1 shows how the creation of the FPTA is implemented in the AALERGIA package. The code starts by sorting the trainings set and by removing double occurrences (line 3). After that, a for-loop iterates through the sorted strings and splits them at each comma (line 6 - 15). This creates all the prefixes from the trainings set and adds them to the cell array named *prefix*, as shown in Table 3.

| |
|---|
| 000,000,010,001,100,011,101,110,001,110,001,110,011,101,010,001,100,011,100,010,001, |
| 000,000,010,001,100,011,101,110,001,110,001,110,011,101,010,001,100,011,100,010, |
| 000,000,010,001,100,011,101,110,001,110,001,110,011,101,010,001,100,011,100, |
| 000,000,010,001,100,011,101,110,001,110,001,110,011,101,010,001,100,011, |
| 000,000,010,001,100,011,101,110,001,110,001,110,011,101,010,001,100, |
| 000,000,010,001,100,011,101,110,001,110,001,110,011,101,010,001, |
| 000,000,010,001,100,011,101,110,001,110,001,110,011,101,010, |
| 000,000,010,001,100,011,101,110,001,110,001,110,011,101, |
| 000,000,010,001,100,011,101,110,001,110,001,110,011, |
| 000,000,010,001,100,011,101,110,001,110,001,110, |
| 000,000,010,001,100,011,101,110,001,110,001, |

Table 3: The Prefixes

**Algorithm 1** Create the FPTA

**Input:** the $\sum$ and a set of strings S (training data)
**Output:** a DFFA A and a sorted set of strings U
$U \leftarrow \text{sort}(S)$
*prefixes* $\leftarrow U$
$F_{fr} \leftarrow \text{string\_count}(U)$
**for** `u` $\in$ `U` **do**
    *index* $\leftarrow$ find positions of "," in $u$
    **while** *index* $> 0$ **do**
        *substr* $\leftarrow$ *substring(1:index)*
        **if** *substr* $\notin$ *prefix* **then**
            *prefixes(end+1)* $\leftarrow$ *substr*
        **end if**
        *index* $\leftarrow$ *index-1*
    **end while**
**end for**
*prefixes* $\leftarrow$ width_first_sort(*prefixes*)
*predecessor* $\leftarrow$ find_predecessor(*prefixes*)
**for** `p` $\in$ `prefixes` **do**
    *sym* $\leftarrow$ *get_symbol(p)*
    *pre* $\leftarrow$ *predecessor(p)*
    *freq_trans_matrix*$^1$ *(pre, sym)* $\leftarrow$ *predecessor(p)*
    *frequency_transition* $\leftarrow$ *freq_trans_matrix*$^2$ *(pre, sym)*
    *frequency_transition* $\leftarrow$ *frequency_transition* + *frequency(p)*
    **while** *pre* **do**
        update_freq_trans_matrix($p$, *pre*)
        *pre* $\leftarrow$ *predecessor(p)*
    **end while**
**end for**
*init_states* $\leftarrow$ *freq_trans_matrix*$^1$*(1, all)*
**return** $A$

In our example, the dimension of *prefix* is 494x1 and contains all unique prefixes of the trainings set for further evaluation. The cell array *prefix* is sorted by width-first-sort (line 16). The shortest strings are at the beginning of the array and the largest at the end of the array, as shown in Table 4.

| |
|---|
| 000, |
| 000,000, |
| 000,001, |
| 000,010, |
| 000,011, |
| 000,100, |
| 000,101, |
| 000,110, |
| 000,111, |
| 000,000,010, |
| 000,000,100, |
| 000,000,111, |
| 000,001,100, |
| 000,001,110, |

Table 4: The Sorted Prefix Array

The function *find_predecessor* searches through the *prefixes* array and saves all the predecessors in the corresponding array (line 16). In the end, a for-loop over all the prefixes is executed in order to find all transitions and their frequencies respectively. The values are saved into the *freq_trans_matrix* (line 18 - 28). This Matrix contains 2 elements: the first element is the transition matrix, which contains all the transition between the states. The second element is a matrix, which contains all the frequencies between the states.

The returned object DFFA (line 30) has the following members:

- a finite set of states
- the labels of the states
- the alphabet
- the initial state
- the initial state frequency
- the final state frequency
- the frequency transition matrix
- RED (later used for merging) [* **ME**: be more comprehensive about RED set *]

- BLUE (later used for merging) [* ME: likewise *]

Table 5 shows how the first element of the frequency transition matrix (the transition matrix) is structured. The columns represent the 8 symbols, the rows the different states. For example, row number 2, column 1 shows the number 3. This indicates that state 2 and symbol 1 lead to state 3. The second element of the frequency transition matrix has the same dimensions as the first element and follows the same logic, but instead of transitions it contains the frequencies.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 0 | 0 | 11 | 0 | 12 | 0 | 0 | 13 |
| 0 | 0 | 0 | 0 | 14 | 0 | 15 | 0 |
| 0 | 16 | 0 | 0 | 0 | 17 | 0 | 0 |

Table 5: The Transition Matrix

### 2.3.3 Normalization

The algorithm 2 shows how a DFFA is converted into a deterministic probabilistic finite automaton (DPFA). The computation is pretty straightforward. As input, the algorithm takes a well defined DFFA, that means that the number of strings entering and leaving a given state is identical. The probabilistic transition matrix (line 5) is identical to the frequency transition matrix, but its second element contains the probabilities of the transitions instead of the frequencies of the transitions.

The algorithm searches through the frequency transition matrix, sums up the frequencies of the transitions and adds them to the frequency of the state itself in order to get the total number of frequencies (line 7). Following that, the frequency of each transition and of the state is divided by the total number of total frequencies, which results in the corresponding probabilities for the transitions. (line 9).

**Algorithm 2** Create a DPFA from a DFFA

1: **Input:** a well defined DFFA
2: **Output:** corresponding DPFA
3: **for** $q \in Q$ **do**
4: $freq\_state \leftarrow dffa.finalStateFrequency(q)$
5: $ptm^1 \leftarrow dffa.frequencyTransitionMatrix^1$
6: $freq\_trans \leftarrow dffa.frequencyTransitionMatrix^2(q,\ all)$
7: $freq\_total \leftarrow freq\_trans + freq\_state$
8: **if** $freq\_total > 0$ **then**
9: $ptm^2(node,\ index) \leftarrow \frac{freq\_trans}{freq\_total}$
10: $fsp(q) \leftarrow \frac{freq\_state}{freq\_total}$
11: **end if**
12: **end for**
13: **return** $DPFA(dffa,\ fsp,\ ptm)$

### 2.3.4 Golden Section Search

First, we need to search for a left and right border of $\varepsilon$ in order to get a region for our golden section search. This is done by the following algorithm 3. In order to save space, the algorithm only explains how to find the left border of the region. The right border is found the same way, with some minor modifications. The algorithm iteratively calculates BIC-scores for the values of $\varepsilon$. For each run of the while-loop, the $\varepsilon$ is multiplied by 0.5 until a left border is found. If the new $\varepsilon$ produces a Bayesian Information Criterion(BIC)-score larger than the previous $\varepsilon$ (line 8), it essentially means that we found a right border, but the left border is further to the left and we need to search again. If the new score is the same as the old score (line 13), we search again. If the value of the score doesn't change for 5 consecutive tries, we break the loop and take the last value of $\varepsilon$ as our left border. Last but not least, if the new score is smaller than the old score (line 17), we found the left border and can continue searching for the right border (not shown in the pseudo code).

**BIC Score** This is a measure that can be used to evaluate the learned model. It creates a score that is calculated on the likelihood minus a penalty term for the model complexity. A larger BIC score (x > y) means a better score and is preferred. Further details are given at [1]. [* ME: we need the formula *]

**Algorithm 3** Find $\varepsilon$-region

```
1: Input: the DFFA
2: Output: the region for ε, scores, epsilon_values
3: ε ← 1
4: score ← calculate_BIC_Score(ε, dffa)
5: new_ε ← ε * 0.5
6: while new_ε > 0 do
7:     new_score ← calculate_BIC_score(new_ε, dffa)
8:     if new_score > score then
9:         region_right ← ε
10:        ε ← new_ε
11:        new_ε ← new_ε * 0.5
12:        score ← new_score
13:    else if new_score = score then                ▷ do this max. 5 times
14:        ε ← new_ε
15:        new_ε ← new_ε * 0.5
16:        score ← new_score
17:    else
18:        region_left ← new_ε
19:        break
20:    end if
21: end while
22: ...
23: ...
24: return region_right, region_left
```

Table 6 shows the values of the left and right border of $\varepsilon$. The algorithm ran only once for the left border ($\varepsilon$ startes with 1 and is halfed every run) and 5 times for the right border, since the scores always stayed the same. Table 7 shows the BIC scores and table 8 shows the corresponding $\varepsilon$ values. For example, we calculated the BIC score for $\varepsilon = 0.5$ and the result was $-1.5769e + 03$.

| | |
|---|---|
| 0.5 | 64 |

Table 6: The Search Region

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| -840.6080 | -1.5769e+03 | -840.6080 | -840.6080 | -840.6080 | -840.6080 | -840.6080 | -840.6080 |

Table 7: The BIC scores

After having found the region, the golden section search can begin. Algorithm 4 shows how it is implemented. Basically, the algorithm uses the golden ratio on our 2 borders a1 and a2 and tries to find a maximum (the highest BIC score) between these 2 values.

| 1 | 0.5 | 2 | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|---|

Table 8: The Epsilon values

If it exists, the section that contains the maximum is selected as new region and the search is started again, until the recursion reaches depth 3 or the condition at line 11 is fulfilled. At line 7, 8 we calculate the 2 values a1,a2 for our golden ratio, given the region r1,r2. After that, the corresponding BIC scores f1,f2 are calculated(line 9, 10). If the distance between the 2 values a1,a2 is small enough, we stop the recursion and the average score between the 2 values is taken as a good $\varepsilon$ (line 12). If f1 is smaller than f2, the maximum has to be between a1 and r2 and a new golden section search is started. If it is the other way around and f1 is larger than f2, the maximum is between r1 and a2 and we start the search with these values. If it happens that the score of f1 and f2 is exactly the same, we need to make sure, that the recursion is not endless (line 24). We check if f1 is larger than the largest score we already calculated. Should this be the case, the maximum is between a1 and a2 (line 27). Lastly, if we do not have any luck and we have not found a maximum so far, we simply start 2 new golden searches for the left (r1, a1) and the right (a2, r2) region respectively (line 34, 35). In the end, the algorithm returns a suggestion for a good $\varepsilon$ as shown in Table 9, which we use as input parameter ($\alpha$) for our merging algorithm.

| 64 |
|---|

Table 9: The Epsilon

**Algorithm 4** Golden_section_search

```
1: Input: region, dffa, scores
2: Output: good_eps, scores, epsilon
3: good_eps ← 0
4: r1 ← region(1,1)
5: r2 ← region(1,2)
6: while true do
7:     a1 ← r1 + 0.382(r2-r1)
8:     a2 ← r2 + 0.618(r2-r1)
9:     f1 ← calculate_BIC_score(a1,dffa)
10:    f2 ← calculate_BIC_score(a2, dffa)
11:    if |a1-a2| < 0,00001 then
12:        good_eps ← (a1 + a2) / 2
13:        return
14:    end if
15:    if f1 < f2 then
16:        region ← region(a1, r2)
17:        good_eps, scores, epsilon ← Golden_section_search(region, dffa, scores)
18:        return
19:    else if f1 > f2 then
20:        region ← region(r1, a2)
21:        good_eps, scores, epsilon ← Golden_section_search(region, dffa, scores)
22:        return
23:    else
24:        if recursion_depth = 3 then
25:            return
26:        end if
27:        if f1 > max(scores) then
28:            new_region ← a1, a2)
29:            good_eps, scores, epsilon ← Golden_section_search(new_region, dffa,
    scores)
30:            return
31:        else
32:            regionL ← region(r1, a1)
33:            regionR ← region(a2, r2)
34:            good_eps, scores, epsilon ← Golden_section_search(regionL, dffa, scores)
35:            good_eps, scores, epsilon ← Golden_section_search(regionR, dffa, scores)
36:            return
37:        end if
38:    end if
39: end while
```

### 2.3.5 Merging

After having calculated a good $\alpha$ value, we can finally start the AALERGIA algorithm. RED is a set of states, that is already revised, whereas BLUE is the set of states that we need to look at. The RED set starts with the initial state (line 3) and BLUE gets all its successors (line 4). After that, we loop through all the elements of BLUE and check if we can merge it with the AALERGIA_compatible criterion (line 14). If the states are compatible, we can merge them and the tree becomes smaller, if not, we promote the state to the RED set (line 21) and exclude it from the BLUE set (line 24). In the end, we get a merged DFFA with a good BIC score. Further details on the AALERGIA_compatible criterion can be found at [1]

**Algorithm 5** AALERGIA

```
1:  Input: a DFFA, a DPFA, the alpha
2:  Output: the merged DFFA
3:  RED ← dffa.initial_state
4:  BLUE ← freq_trans_matrix^1(dffa.initial_state, all)
5:  promote ← 1
6:  while BLUE is not empty do
7:      BLUE ← sort(BLUE)
8:      q_b ← BLUE(1)
9:      BLUE.remove(q_b)
10:     same_label_ind ←find_same_labels(RED, q_b)
11:     for ind ∈ same_label_ind do
12:         q_r ← RED(ind)
13:         threshold ← calculate_compatible_params(dffa, q_r, q_b, alpha)
14:         if AALERGIA_compatible(dffa, dpfa, q_r, q_b, alpha, threshold) then
15:             dffa_merged ← AALERGIA_merge(RED, BLUE, q_r, q_b)
16:             promote ← 0
17:             break
18:         end if
19:     end for
20:     if promote then
21:         RED ← q_b and RED
22:     end if
23:     successors ← dffa_merged.freq_trans_matrix^1(RED, all)
24:     BLUE ← all successors not in RED
25: end while
```

### 2.3.6 Perplexity

# 3 Related Work

[* ME: The contest you were looking at. write three paragraphs each of which dedicated to one of the top three approaches; cite the publications (Worst to best). *] [* ME: The continue the discussion on SHIBATA and why it is winning every major contest in this field PAutomataC 2012, CONTEST 2016. *]

# 4 Meat

[* ME: a case study on wireless sensory networks. *]

# 5 Evaluation

[* ME: compare the findings about the case study with real-world scenario and the knowledge of an expert in the field of wireless sensory networks. *]

# 6 Conclusions

[* ME: the approach is good; but! *]

# References

[1] H. Mao, Y. Chen, M. Jaeger, T. D. Nielsen, K. G. Larsen, and B. Nielsen, "Learning probabilistic automata for model checking," in *2011 Eighth International Conference on Quantitative Evaluation of Systems (QEST)*, pp. 111–120.

[2] D. Knuth and A. Yao, *Algorithms and Complexity: New Directions and Recent Results*, ch. The complexity of nonuniform random number generation. Academic Press, 1976.