Algorithms and Computability PROJECT

Patryk Abramczyk
Andrzej Frankowski
Filip Nasiadko
Karol Mielnicki
Warsaw Univercity of Technology

1. Introduction

Goal of the program is to find an optimal Deterministic Finite Automata accepting a regular language **L**, such that **L** ⊂ Σ*, where Σ* is the set of all words over an input alphabet Σ. Program is given a tool s.t. (∀ x, y ε Σ*) the tool answers question: *for two given words x and y, does the relation induced by the language* $\boldsymbol{R_L}$ *hold between them, i.e.* $\boldsymbol{x}\ R_L\ \boldsymbol{y}$*?*

The tool is given in the form of automata that accepts the same language **L** and can be imported into the program from a text file. Optimal solution is an automata with the smallest number of errors compared to the given tool, where error is the number of words accepted or rejected contrary to the given tool.

2. Input

Input is given as a text file. User has a possibility to load it by selecting a direct path file dialog, or by dragging file onto program frame. Loaded data is the definition of a specific deterministic finite automaton. It is represented as a sequence of natural numbers separated with commas, where:

- First number "N" denotes number of states, where N is >= 1, and 1 is the initial state
- Second number "K" denotes number of symbols in alphabet
- Sequence of consecutive N numbers denotes each state in the transition function table
- Sequence of consecutive K number denotes each symbol in the alphabet
- Additionally there can be M number of accepting states where M <= N

After loading a file we need to define parameters for the particle swarm optimisation. Those parameters are:

- Maximal number of iteration.
- Maximal number of states.
- Tolerance of error.
- Maximal word length.
- Coefficients of direction for global, local and personal best sample position

Program will be launched with a default data set. User will be able to override data by specifying path to a text file containing unified DFA representation and providing PSO parameters in Settings panel. Computations start on a button click. Upon launch of the calculations input data and algorithm parameters are validated. Proper information is displayed in case of invalid fields and input data. Calculations happen in background, informing user about current progress and potential optimal automata.

3. Calculations

Particle Swarm Optimization

PSO is a computational optimization method that is inspired by the group behavior of animals. As genetic algorithms, it is a population-based method, that is, it represents the state of the algorithm by a population, which is iteratively modified until a termination criterion is satisfied.

In PSO algorithms, the population $P = \{p_1,\dots,p_n\}$ of the feasible solutions is often called a **swarm** and the feasible solutions $p_1,\dots,p_n$ are called **particles**. The PSO method views the set R^n of feasible solutions as a “space” where the particles “move” towards best solution. Each particle i has its neighborhood N_i (a subset of P). The structure of the neighborhoods is called the **swarm topology**, which can be represented by a graph.

Each of particles i at given iteration t has:

- x_i (t) -> the position
- p_i (t) -> the “historically” best position, called **personal best**
- l_i (t) -> the best position of the neighboring particles, called **local best**; for the fully connected topology it is the “historically” best known position of the entire swarm
- v_i (t) -> the velocity

At the beginning of the algorithm, the particle velocities and positions are randomly initialized within set bounds. Termination happens after a given number of iterations, or once the ideal solution is obtained.

a. Parameters

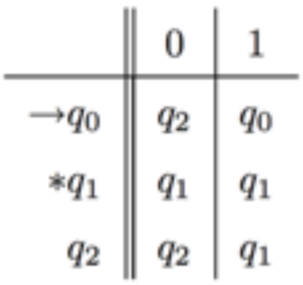
Following parameters can be provided to optimize PSO algorithm:

- MAXITER - number of maximal iterations
- PARTICLES_COUNT - number of particles in the swarm, i.e. how many automatas we define in the search space.
- PERSONAL_WEIGHT - moving towards personal best position
- GLOBAL_WEIGHT - moving towards global best position
- VEL_WEIGHT - preserving current velocity
- SPACE_SIZE - size of the solution space

b. Representation of DFA in PSO algorithm

Deterministic Finite Automata transition table is commonly represented as $\delta: \Sigma \times Q \rightarrow Q$. This cannot be easily represented in the solution space for PSO algorithms, therefore we needed to find another form that will allow us to store information about transition between states. Such form is $\delta: Q \times \Sigma \times Q \rightarrow \{0, 1\}$, i.e. a matrix representing transition function δ per each symbol from the alphabet. Rows are represented as input states, columns as output states and values are either 0, or 1, representing whether there is a transition from the input state to the output state.

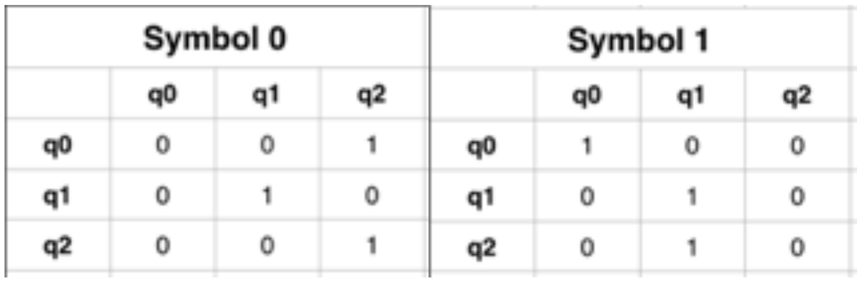
Example of converting table representing function $\delta: \Sigma \times Q \rightarrow Q$ to transition table representation in our program $\delta: Q \times \Sigma \times Q \rightarrow \{0, 1\}$ for a given DFA:



	0	1
$\rightarrow q_0$	q_2	q_0
$*q_1$	q_1	q_1
q_2	q_2	q_1

Transition table. The $\rightarrow$ indicates the start state: here q 0 The $*$ indicates the final state(s) (here only one final state q 1)
For this example $Q = \{ q_0, q_1, q_2 \}$ start state q_0 $F = \{ q_1 \}$ $\Sigma = \{ 0, 1 \}$ δ is a function from $Q \times \Sigma$ to Q $\delta : Q \times \Sigma \rightarrow Q$ $\delta (q\,0, 1) = q_0$ $\delta (q\,0, 0) = q\,2$

Program representation:



Symbol 0				Symbol 1			
	q0	q1	q2		q0	q1	q2
q0	0	0	1	q0	1	0	0
q1	0	1	0	q1	0	1	0
q2	0	0	1	q2	0	1	0

We do not consider non-determinism, so each row must have at most one 1 in each row.

c. Cost function

First of all we need to define what exactly is an error. In our case it’s the difference, given as percentage value, between answers of a given tool and current particle. Cost function will calculate error, based on current position of every particle automaton. Thanks to such solution it will be possible to compare two automatons with different number of states and then decide which one is closer to ideal solution.

d. Preliminary operations

Word sets

In order to achieve best results we create two separate sets of words - testing and training set. Training set is used to create the tool in form of DFA. Testing set is used to test for relationship of two words during computation.

Creating a tool

We load a text file to the program and based on the data inside it we create transition table. Thanks to this we can generate a tool that will take two words as an arguments and rather they are in relation or not.

Generating word sets

In the beginning, based on the input alphabet Σ we generate finite sets of words accepted by our “tool” automaton. In order to create this set of words, we select randomly a sample population, taking into account size restrictions. We need to remember that those values are crucial for the optimization purposes, so they can be modified by user. The idea of obtaining words works in a such way that firstly we create all possible combinations of short words. Next we increase length of word and generate all new combinations from all symbols for a given length. In the meantime, after each iteration we check if the new word is in relation with any of other words and based on this we can define new equivalence class or add word to the existing one.

For longer words we check all the combinations and compare them with generated equivalence classes, that means: we create a word, check if it belongs to any equivalence class. If so, then we do not add it to the set of generated words and continue computation.

Definitions of local, personal and global best

During computation we move our particles closer and closer to perfect solution. Direction is calculated based on three vectors. They start in position of current particle and ends in different points, which are:

- Personal Best - Position of the best version of specific particle.
- Local Best - Position of the best particle in the nearest neighborhood.
- Global Best - Currently the best particle in set of all particles.

Discretization

Although the algorithm operates on floating point values in order to perform precise moves of the particles, we have to use integers to represent actual DFA models. Values which are used to construct automatas are rounded to the nearest integer value.

Pseudocode

```
Start
    for (i=1 to PARTICLES_COUNT)
        $P_{Velocity}$ = uniformly random vector
        $P_{Position}$ = uniformly random position bounded by solution space
        $P_{Personal\_Best}$ = $P_{Position}$
        if (Cost ($P_{Personal\_Best}$)  Cost ($P_{global\_Best}$))
            $P_{Global\_Best}$ = $P_{Personal\_Best}$
        End if
    End for
        While (i < MAX_ITER && ideal solution is not found
            for (P PARTICLES)
                for u1 and u2 uniformly (0,1) distributed random vectors of
                length nvars
                $P_{Velocity}$ = VEL_WEIGHT * $P_{velocity}$ + PERS_WEIGHT *u1 *
                ($P_{Personal\_Best}$ - $P_{Position}$) + GLOB_WEIGHT * u2 *
                (Global_Best_Position - $P_{Position}$)
                $P_{Velocity}$ = Normalize ($P_{Velocity}$)
                $P_{Position}$ = ShiftPosition($P_{Position}$, $P_{Velocity}$)
                if(Cost($P_{Position}$)  Cost($P_{Personal\_Best}$))
                    $P_{Personal\_Best}$ = $P_{Position}$
                    If(Cost( $P_{Personal\_Best}$)  Cost($P_{Global\_Best}$)
                        $P_{Global\_Best}$ = $P_{Personal\_Best}$
                    End if
                End if
            End for
        End while
End
```

Unreachable states

After the algorithm has finished calculations, some automatas may have unreachable states. If there are two automatas with the same number of errors, the one with less number of unreachable states is considered to be better.

After each iteration we decide where and how to move. One of the main problems is to change transition table according to new position. New coordinates gives us information about positions of 1's in the table. We change 0's to 1's in the proper cells and the way around. If the automaton turns out to be non-deterministic we have to convert it to deterministic one by splitting table to two.

Program behaviour during computations

During iterations, which can take a long time program has to somehow inform user about ongoing computation. Time complexity of the calculation is hard to define, so program will display number of current iteration and number of maximum iterations in the simplest form i.e. "Computation progress: 236/1000".

Computation interruptions
In case of unexpected shutdown program will not store any progress information. According to project requirements it is not a crucial functionality, thus it is better to focus on improving computation than additional functions. As a extra feature it can be implemented if, and only if the rest of the project will be done before deadline.

4. Output

Output of a successful computation is given as a list of best five tested automatons with the smallest error. They are sorted descending according to the error value. User can compare them and interpret given results easily.

Output can be interpreted in terms of number of errors compared to the original DFA. Unreasonably poor results may indicate that user will have to reconsider PSO parameters.

5. Changes

The original idea did not change too drastically. Most of the code is as described in the above content. The changes that were made are as follows

- Solution representation includes state number and acceptable states

As an additional feature, to improve our solution we decided to add information about the state's number and list of acceptable states. Thanks to this, the solution representation is more readable.

- Another important one is dynamic limits for each solution (next state 0-5 for 5 state dfa, 0-10 for 10 state)

In the beginning we decided that the maximum number of possible states for each solution will be 20. During our work we came up with an idea that it can be optimized. For each solution we dynamically change the maximum number of possible states. Thanks to such a solution we do not generate unnecessarily very big automatons.

- Space_size is removed as it is obsolete and determined dynamically for each vector element in a particle.

- The max execution time was added as a parameter to control the algorithm run time, along with the number of iterations

DFA learning system

Architecture overview

DFO optimisation consists of 3 stages:

1. DFA loading
2. Test set generation
3. PSO optimisation

The purpose of the step 3 is to demonstrate the optimisation capabilities of the PSO and it's ability to be used for machine learning purposes. The objective is to minimise the error of accepting/rejecting input words on a learning set of input words.

After learning, the learned DFA can be tested on a separate testing set.

Architecturally, the application is divided into the optimisation library and the GUI. Gui uses the optimisation library to run the optimisation algorithm and display the results. The optimisation library consists of the DFA support that loads DFA-s, represents them in memory and can compute their output for a given input, the word set generator and the PSO.

DFA support

The automaton is represented by the **DFA.java** class. The automatons are loaded from file using the **parseFromFile(File path)** method in the **DFOFactory.java** class. Each automaton consists of 5 element: (1) set of states, (2) set of inputs, (3) transition table, (4) initial state and (5) set of acceptable states. Both states and inputs are encoded as integers. The transition table is implemented in **TransitionTable.java** class, as a Map such that the next states correspond to pairs of initial states and inputs.

The transitions are computed using the **DFAComputer.java** class. The main method to do this is **compute(List<Integer> inputs)**. After computing the output, it is returned as a **ComputeResults.java** object, containing the final state and a flag that indicates if the input word is accepted or not (available using the **isAccepted()** method)**.**

Word set generator

To generate the word set that will be used in the PSO, **WordSetGenerator.java** is used. It uses the method **generateWordSet()** to generate words with configurable maximum word length and maximum allowed number of words in the set. During the generation, the generator will check if the word is accepted in the loaded DFA. If it is, it will be included to the accepted words list in the word set, if not, it will be included in the non accepted word list.

The results of the generator is the **WordSet** class, that consists of two lists of words: accepted and unaccepted. Word set is the basis for evaluation of the automata learned by the PSO. If the optimized DFA accepts a word from the non accepted list and vice versa, this is counted as an error. After all the errors are counted, the overall evaluation is equal to errors/totalWordNumber.

Functionality of solution evaluation is implemented in the **Evaluator.java** class, method **evaluate(Solution solution).**

PSO

PSO is an optimization algorithm that produces solution of an optimization problem.

Our solution is implemented in the **Solution** class. It is a representation of a deterministic finite automaton in a form suitable for modification and optimization by the PSO algorithm.

Our solution consists of deciding on (1) number of states, (2) transition table and (3) acceptable states. Number of states is assigned as a double value. DFA state number is later calculated by rounding that value. Transition table is represented as a series of **DoubleTransitionTable** objects, a different one for each input. **DoubleTransitionTable** class keeps a map of double values, one number for each state. Next state for a given state and input is represented as a rounded value of the DFA. Acceptance of the states is represented as **acceptedStates** field in the Solution class. If a double value for a state is less than 0.5, state is not accepted, if it's higher it is accepted.

Example solution:

```
Transition table for input 1
1:        4.913842754193902
2:        1.6391503715281601
3:        3.2068347375936925
4:        3.8047906163550587
5:        6.25727762227946
6:        1.9636804224451467
7:        4.311926385920067

Transition table for input 2
1:        3.9314463262959998
2:        6.27847817573301
3:        3.405339311956868
4:        3.597471468265049
5:        5.566492749375323
6:        6.241197393303487
7:        4.334229611708858

Accepted:
1:        0.9297334759357836
2:        0.0
3:        0.49712971341624906
4:        0.9456179237223038
5:        0.631889449453432
6:        0.4897191302016699
7:        0.48346460172456746
```

It has 7 states and two inputs. For input 1, from state 1 it goes to state 5 since the line 1 in Transition Table for input 1 has 4.913842754193902. For input 2, it goes to state 2 (1.63 rounded) etc. Also, state 1 is accepted as 0.92 from the acceptance table is rounded to 1. State 2 is not acceptable nor is 3 as 0.0 and 0.4971 are rounded to 0.

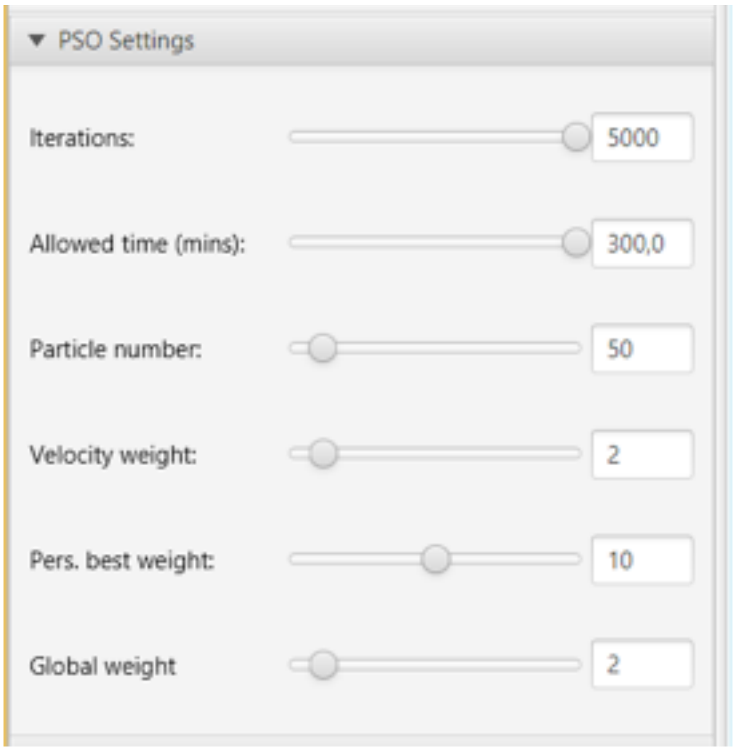
When converted to the usual representation of an automaton, it looks like this:

```
States (7): [1, 2, 3, 4, 5, 6, 7]
Inputs: (2): [1, 2]
Transition table:
        1    2
--------------------
q1  |  q5   q4
q2  |  q2   q6
q3  |  q3   q3
q4  |  q4   q4
q5  |  q6   q6
q6  |  q2   q6
q7  |  q4   q4
Initial state: 1
Accepted states: (3): [1, 4, 5]
```

Conversion from the Solution objects to DFA objects are done using the method **convertFromSolution()** in the **DFAFactory class.**

Solutions are produced using the PSO algorithm, implemented in the PSO class. PSO consists of a swarm (list) of particles that explore the search space. The method that does the optimization is **search()**, which takes the input states and learning word set as arguments. PSO is constructed based on the **PSOParams** object. It takes all the input parameters needed for the optimization algorithm (can be configured in the GUI).

Iterations parameter determines the maximum number of algorithm iterations. *Allowed time* determines maximum allowed time the algorithm will execute. *Particle number* defines the number of particles. *Velocity weight* determines the relative influence of the current particle velocity. If it is high, it will be difficult for the particles to change their direction, if it is low, particles will more likely be able to change direction. *Pers. Best weight* is a parameter that determines the likelihood that the particle will change direction towards the best position of this particle during the search. *Global best weight* determines the likelihood that the particle will turn in the direction of the best solution found by all particles together (best so far found by the entire algorithm).



Iterations:	5000
Allowed time (mins):	300,0
Particle number:	50
Velocity weight:	2
Pers. best weight:	10
Global weight	2

PSO main loop consists of

```
updateSpeeds();
moveParticles();
evaluateParticles(results);
```

methods. It updates speeds of particles, moves them, evaluates them all then repeats until the end condition.

During entire algorithm, best solution is kept in the bestSoFar variable of the PSO class.

Particle is implemented in **Particle** class. It consists of a current solution, best solution discovered by that particle and the velocity vector. Velocity vector is analogous to the solution class and the key method is **update()** which updates speed vectors according to the formula from the pseudocode (pg. 5 of the documentation):

$$P_{Velocity} = VEL\_WEIGHT * P_{velocity} + PERS\_WEIGHT * u1 * (P_{Personal\_Best} - P_{Position}) + GLOB\_WEIGHT * u2 * (Global\_Best\_Position - P_{Position})$$

Particle movement is done using the **performMovement()** in the particle class, which does the calculation from the pseudocode on pg. 5:

$$P_{Position} = ShiftPosition(P_{Position}, P_{Velocity})$$

Tests

PSOTest.java, PSOTestEven.java, PSOTestNonAccepting.java

Loads a DFA, prints it, creates word set, runs PSO, displays best found solution.

SolutionTest.java

Tests the Solution class

WordGeneratorTest.java

Tests the word generator class

GUI

Implemented in JavaFX 2.0, the GUI form is defined in **DFALearner.fxml.**

Main class loads and displays the form. The form is handled by the class **DFALearnerController.java** which dispatches events across application and takes care behavior is consistent, for example it loads, and displays DFAs in text panes, makes sure that buttons are disabled during processing and enables them back when it's done, handles errors such as I/O errors or file format errors. Actions for three buttons: load, learn and test are implemented in the following methods of **DFALearnerController.java:**

> **loadAutomatonAction(),**
>
> **learnAutomatonAction()**
>
> **performTest()**

DoubleOnlyChangeListener.java, SliderChangeListener.java, TextFieldDoubleChangeListener.java and TextFieldIntegerChangeListener.java are auxiliary listener classes that respond to events such as changes in the values of the parameters by sliders or directly by writing the parameters into text boxes. They handle those events by updating the GUI elements to provide consistent values in all elements (slider change changes the text field, changing the text field moves the slider).

PSOLearner.java is a **Runnable** class being used in a separate thread to do long computation of the PSO algorithm. Running it in a separate thread allows the user interface to be responsive during long computation and not hang and stop responding.

PSOGuiListener.java handles optimization events – each time a better solution is found or after a certain number of iterations elapses, it updates the status text and the progress bar in the status bar. After the PSO algorithm is finished it updates all the results and displays them.

Using the application

Loading DFAs

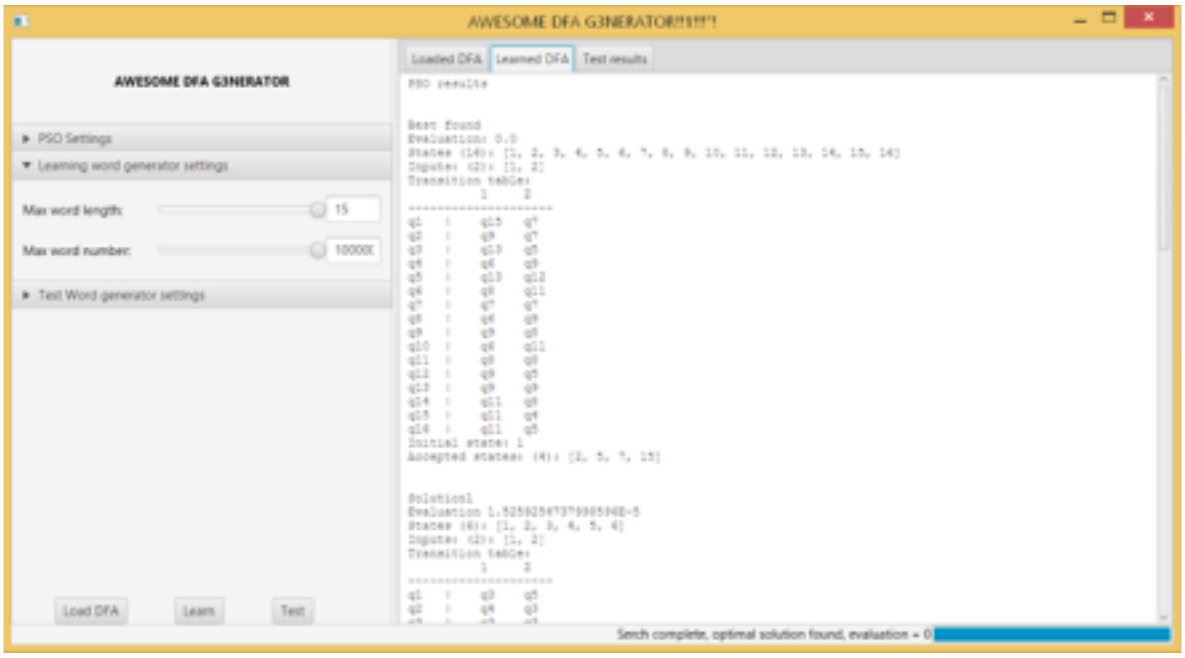
Click on load, by default it loads “exemple.dfa” in the current directory during startup.

Learning DFAs

Click learn. Adjust settings of PSO and Learning word set previously.

Learning progress is displayed in the progress bar.

Solution is displayed in Learned DFA tab after (1) optimal DFA with evaluation 0 is found, (2) all iterations have been done or (3) allowed time expires, whichever comes first.


AWESOME DFA G3NERATOR!!1!!"!
AWESOME DFA G3NERATOR
PSO Settings
Learning word generator settings
Test Word generator settings
Load DFA
Learn
Test
Loaded DFA
Learned DFA
Test results

Max word length: 15

Max word number: 10000

```
PSO results


Best found
Evaluation: 0.0
States (16): [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
Inputs: (2): [1, 2]
Transition table:
          1     2
--------------------
q1   |    q15   q7
q2   |    q9    q7
q3   |    q13   q5
q4   |    q6    q9
q5   |    q13   q12
q6   |    q8    q11
q7   |    q7    q7
q8   |    q6    q9
q9   |    q9    q8
q10  |    q6    q11
q11  |    q8    q8
q12  |    q9    q5
q13  |    q9    q9
q14  |    q11   q8
q15  |    q11   q4
q16  |    q11   q5
Initial state: 1
Accepted states: (4): [2, 5, 7, 15]


Solution1
Evaluation 1.5259254737998596E-5
States (6): [1, 2, 3, 4, 5, 6]
Inputs: (2): [1, 2]
Transition table:
          1     2
--------------------
q1   |    q3    q5
q2   |    q4    q3
```

Serch complete, optimal solution found, evaluation = 0

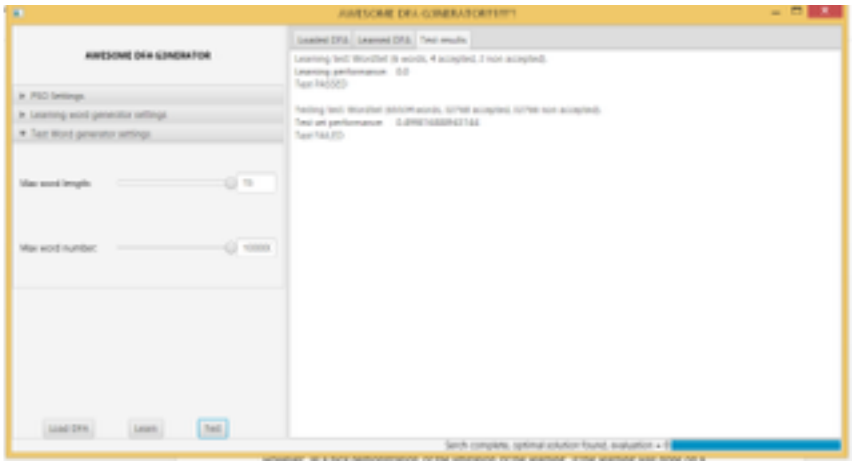
Testing the learned DFAs

After learning, adjust the test word generator settings and click test.

Note that if the test generator is set up to generate less and shorter words than the learning, the test results will always pass.

However, as a nice demonstration of the limitation of the learning, if the learning was done on a small learning set and testing is done on large words, it is likely that the results will not be good.

In our example, after learning the DFA on words up to 2 in length, it's not realistic the results of learning will be good enough to work on words up to 15 in length. Small learning set just doesn't give enough information for successful learning.



AWESOME DFA GENERATOR!!!!!
AWESOME DFA GENERATOR
Loaded DFA
Learned DFA
Test results
PSO Settings
Learning word generator settings
Test Word generator settings
Load DFA
Learn
Test

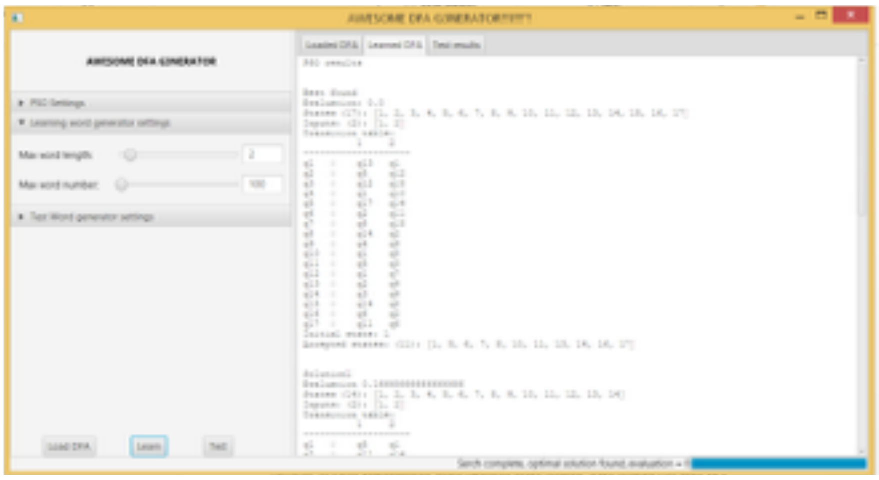
Max word length: 15

Max word number: 100000

```
Learning test: WordSet (6 words, 4 accepted, 2 non accepted)
Learning performance:   0.0
Test PASSED

Testing test: WordSet ([illegible] words, [illegible] accepted, [illegible] non accepted)
Test set performance:   [illegible]
Test FAILED
```

Serch complete, optimal solution found, evaluation = [illegible]



AWESOME DFA GENERATOR!!!!!
AWESOME DFA GENERATOR
Loaded DFA
Learned DFA
Test results
PSO Settings
Learning word generator settings
Test Word generator settings
Load DFA
Learn
Test

Max word length: 2

Max word number: 100

```
PSO results

Best found
Evaluation: 0.0
States (17): [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]
Inputs (2): [1, 2]
Transition table:
            1     2
--------------------
[illegible]
Initial state: 1
Accepted states: (11): [illegible]


Solution1
Evaluation: [illegible]
States (14): [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
Inputs (2): [1, 2]
Transition table:
            1     2
--------------------
[illegible]
```

Serch complete, optimal solution found, evaluation = [illegible]

Tests

Reconstruction

Test description

All of tests were made following to description posted in **MinimalRequirements.docx**
classes of automata having 4, 6, 10, 15 states were chosen and 10 of each class were randomly generated.

Document containing all randomly generated automatas is attached as: **Inputs.xlsx**, in format ready to run in **DFA Application**

As required all automatas were tested in 2 modes:

Mode 1:

Words with length c = 6
Number of words limited to n=5,000
Number of iterations I=5,000
Time limit to t=3 min

Test module for comparison:
Words with length c = 7
Number of words limited to n=10,000

Mode 2:

Words with length c = 7
Number of words limited to n=6,000
Number of iterations I=5,000
Time limit to t=3 min

Test module for comparison:
Words with length c = 8
Number of words limited to n=10,000

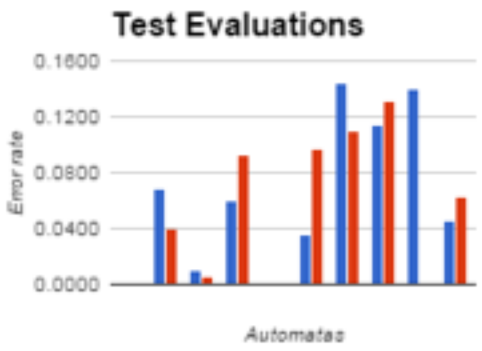
Test results

Test results to every single test are attached as: **tests.xlsx**

4-states

Caused by low level of complexity of 4-states automatas, DFA Generator has indisputably great results.

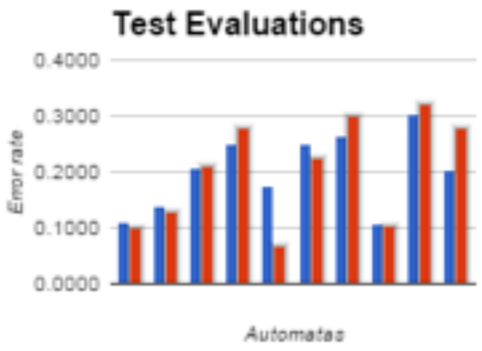
Average evaluation error rate: 0.0578
Best found automata evaluation: 0.000
Average difference between learn and test module: 0.0013
Best Automata(s): 4,1 and 4,5
Attached input file: 4-Best.dfa



Application caped to find two ideal solutions, evaluation rate is very insignificant and Average difference between learn and test module is almost unnoticeable. Low level of complexity, also affect huge difference between accuracy in calculations in two modes (blue - mode1, red mode2)

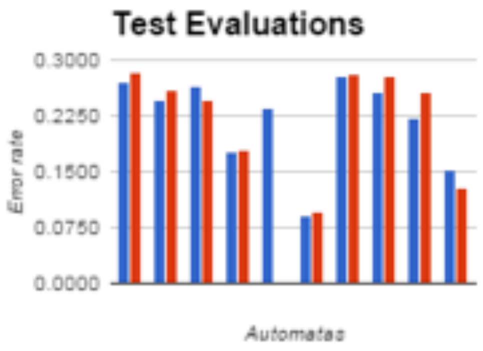
6-states

With rising complexity of automatas, precision of solutions is noticeable decreasing



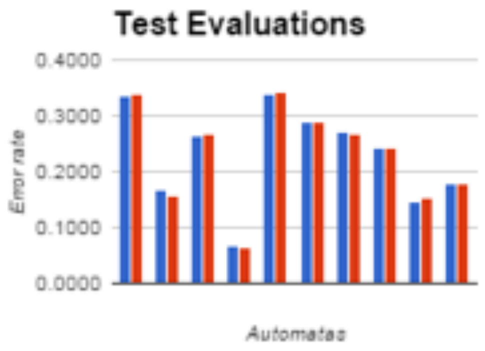
Average evaluation error rate: 0.2011
Average difference between learn and test module: 0.0084
Best found automata evaluation: 0.0672
Best Automata(s): 6,5, for mode 2
Attached input file: 6-Best.dfa

10-states



Average evaluation error rate: 0.2103
Average difference between learn and test module: 0.0061
Best found automata evaluation: 0.0000
Best Automata(s): 10,5 for mode 2 Attached input file: 10-Best.dfa

In 10-stages automatas DFA generator has surprisingly good results in comparison to 6-stages automatas. Avare error is slightly higher but difference between learn and test module is smaller and what is more important, it managed to find perfect solution. High complexity in 10 stages automatas, affects that for exception with 5th test, all of them have very close accuracy in both modes.



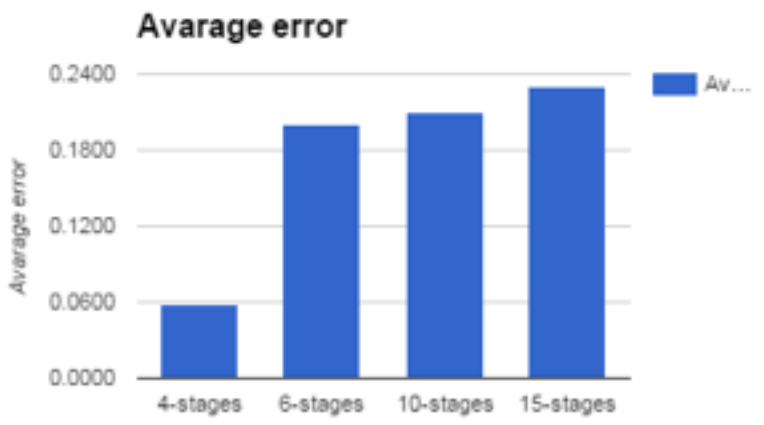
15-states

Average evaluation error rate: 0.2304
Average difference between learn and test module: 0.0040
Best found automata evaluation: 0.0648Best Automata(s): 15,4 for mode 2
Attached input file: 15-Best.dfa

15-stages automatas are the most complex in this chapter. Significant difference in complexity affects results directly. Error rate is higher then it was in previous tests. Surprisingly average difference between learn and test module is very low and only 4-stages automatas has lower rate. Best found automata has undoubtedly satisfactory results. It is notable, that 15-stages automatas are complex enough to almost have no difference in accuracy of solutions for both modes.

Conclusion

In each of 4 class of automatas, 10 were tested in two modes for learning and test module



Team's expectations were met. With rising complexity of automatas, logaritmicly rised avarage error of solutions found by DFA generator. Graph shows how much differ accuracy for 4 and 6-stages automatas. Then with rising number of stages difference is significantly lower.

Approximation

Test description

All of tests were made following to description posted in **MinimalRequirements.docx**
classes of automata having 20, 30, 50, 80 states were chosen and 5 of each class was randomly generated.

Tests:

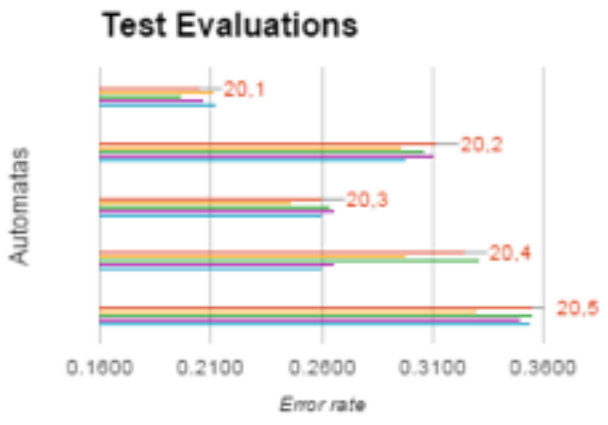
Words with length c = 6
Number of words limited to n=5,000
Number of iterations I=5,000
Time limit to t=3 min

Test module for comparison:
Words with length c = 7
Number of words limited to n=10,000

Algorithm as required, was forced to find solutions only for given nr of states: 4, 6, 8, 10, 12.
Each automaton of each class was tested for all of the states' limitations.

20-states

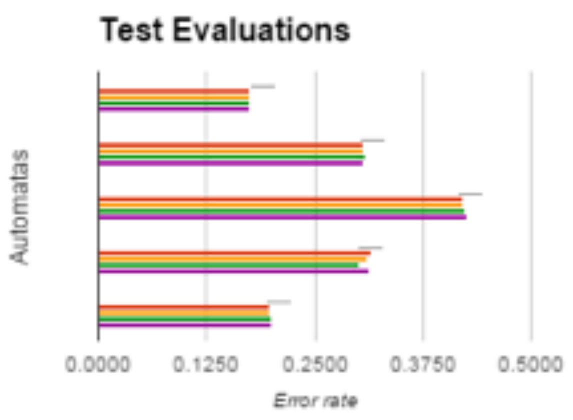
Average error rate, 4-states:	0.2917
Best found 4-states automata:	0.2056
Average error rate, 6-states:	0.2767
Best found 6-states automata:	0.2112
Average error rate, 8-states:	0.2909
Best found 8-states automata:	0.1966
Average error rate, 10-states:	0.2801
Best found 10-states automata:	0.2072
Average error rate, 12-states:	0.2771
Best found 12-states automata:	0.2124



Average evaluation error rate: 0.2833
Average difference between learn and test module:0.0080
Best Automata(s): 0.1966
Attached input file: 20-Best.dfa

30-states

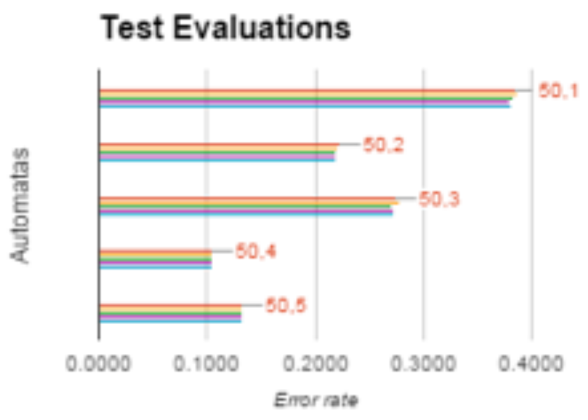
Average error rate, 4-states:	0.2794
Best found 4-states automata:	0.1768
Average error rate, 6-states:	0.2835
Best found 6-states automata:	0.1750
Average error rate, 8-states:	0.2816
Best found 8-states automata:	0.1746
Average error rate, 10-states:	0.2820
Best found 10-states automata:	0.1754
Average error rate, 12-states:	0.2841
Best found 12-states automata:	0.1754



Average evaluation error rate: 0.2821
Average difference between learn and test module: 0.0065
Best Automata(s): 0.1746
Attached input file: 30-Best.dfa

50-states

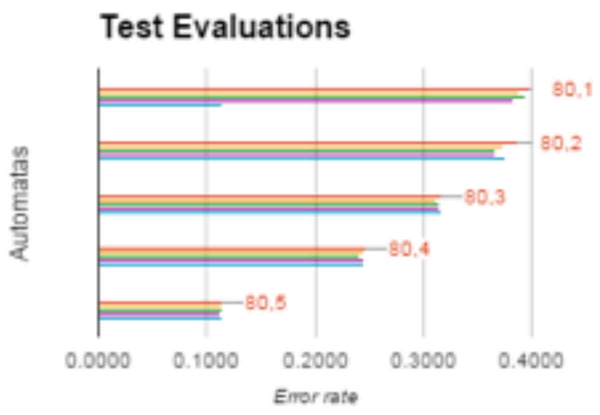
Average error rate, 4-states:	0.2241
Best found 4-states automata:	0.1056
Average error rate, 6-states:	0.2249
Best found 6-states automata:	0.1056
Average error rate, 8-states:	0.2222
Best found 8-states automata:	0.1050
Average error rate, 10-states:	0.2225
Best found 10-states automata:	0.1056
Average error rate, 12-states:	0.2222
Best found 12-states automata:	0.1048



Average evaluation error rate: 0.2232
Average difference between learn and test module: 0.0011
Best Automata(s): 0.1048
Attached input file: 50-Best.dfa

80-states

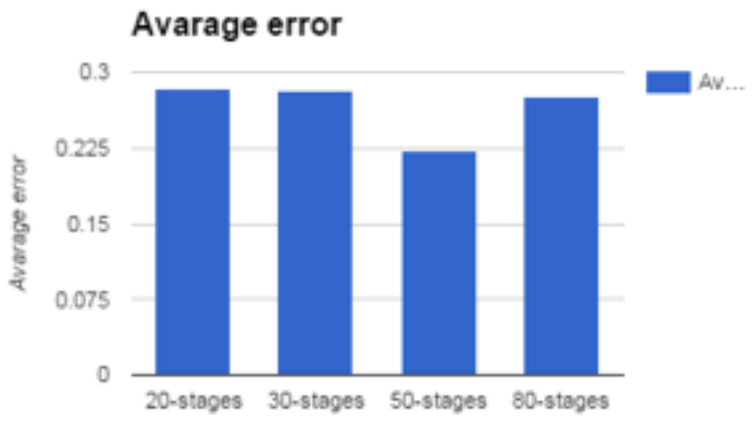
Average error rate, 4-states:	0.2930
Best found 4-states automata:	0.1148
Average error rate, 6-states:	0.2869
Best found 6-states automata:	0.1144
Average error rate, 8-states:	0.2866
Best found 8-states automata:	0.1142
Average error rate, 10-states:	0.2846
Best found 10-states automata:	0.1132
Average error rate, 12-states:	0.2332
Best found 12-states automata:	0.1140



Average evaluation error rate: 0.2769
Average difference between learn and test module: 0.0080
Best Automata(s): 0.1132
Attached input file: 80-Best.dfa

Conclusion

In each of 4 class of automatas, 5 were tested in in 5 different output states bounds for learning and test module



As expected, number of complexity did not make a huge impact on accuracy of algorithm because all of 4 classes are very complex. Surprisingly as posted in previous graphs, number of output automatas stages did not make significant changes either. Examination on each class shows that the only significant change on precision of solutions has number of accepting states of each automata.