The Automata Learning Tool

Master Thesis Project

Authors: Morten Stig Mortensen
Supervisor: Ulrik Mathias Nyman

Department of Computer Science
Aalborg University

Abstract

This report is an extension of a prior semester project. At first we conduct an experiment in order test our hypothesis, the experiment yielded interesting results. Then, the setup of the experiment is evaluated and reflected upon. Based on the results, we decided to make further additions to the learning tool. Before adding the new additions the learning tool was able to handle DFA's, but with the new additions it can now also handle NFA's and PDA's. Then the implementations are discussed. Finally, a conclusion is drawn and ideas for future work are presented.

Contents

Chapter 1

Introduction

The work documented in this report is based upon some of the results of the prior semester. In the prior semester to this, the authors of this report wrote a paper called Developing and Evaluating an Automata Learning Tool [1], which did mention preparations of an experiment that had to be conducted. The experiment has been conducted in this report and showed interesting results. The paper from the prior semester can be found in the Aalborg University Project Library.

The results of the experiment indicated that the participants gained more from doing the paper and pencil exercises, rather than using the learning tool which we have created. This of course contradicted the hypothesis that we had, which stated *is the learning tool we have created better or as good as the use of traditional paper and pencil*. The results of the experiment can be seen in subsection 2.2 or in the Appendix for more details.

Even though the results of the experiment contradicted our hypothesis, we still believed in the learning tool and the vision which we had in mind for this project. We started to work on this learning tool because we felt, at the time we was introduced to automata theory, that we needed something which was able to visualize the different types of fundamental automata. By having the automata visualized it would be easier to understand how an automaton worked, and the learning tool would also make sure that the constructed automaton is correct according to its semantic definition. We felt that all these things could enhance the understanding of students who are for the first time being introduced to automata theory. Thus, our vision for this learning tool was to deliver the above.

With the vision for this learning tool in mind, we kept implementing new features. Before implementing the new features, the learning tool was capable of

autogenerating exercises which had something to do with deterministic finite automata (DFA). The goal of this report was to implement new features that would further improve the learning tool. This report deals with the theory and implementation of two other fundamental automata: non-deterministic finite automata (NFA), and pushdown automata (PDA).

Chapter 2

The Experiment

In this chapter we will present the findings of the experiment which we have conducted. The purpose of this experiment was to test the hypothesis of whether the learning tool we have created is better or as good as the use of traditional paper and pencil. With this experiment we were also interested in collecting feedback from the involved participants expressing whether or not, if they preferred to use the learning tool.

2.1 The Experiment Setup

This experiment has been conducted on students who were for the first time being introduced to automata theory. All of the students who participated in the experiment all attended the introductory course *Syntax and Semantics* at Aalborg University. It should also be noted that the students who were involved with the experiment only had their first lecture in the introductory course, at the time of conducting this experiment. The number of students that participated in this experiment was 11. In Table 2.1 the setup for the experiment is shown.

At first we divided the students into two groups, A and B, where the size of group A was 6 and the size of group B was 5. Each row in Table 2.1 represents an activity with the amount of time in minutes allocated shown in the parenthesis. At first both groups were given a test, this test was the same throughout the whole experiment. We wanted to measure the students initial knowledge and understanding of automata theory, and therefore we started out with a test. After the first test we gave group A exercises that had to be solved using traditional paper and pencil, P, and group B got exercises which had to be solved using the learning tool, C. Then another test was conducted, such that, we could measure if the students had improved. After the second test we

	A	B
	Test(10)	Test(10)
Session 1:	P(15)	C(15)
	Test(10)	Test(10)
Session 2:	C(15)	P(15)
	Test(10)	Test(10)

Table 2.1: The Experiment Setup Visualized

switched the type of medium that the students had to use. Then at last, a final test was conducted to see if any further improvements had been made.

The test consisted of 4 different exercises. In the first exercise the students had to perform set operations, such as, intersection, difference, union, and concatenation given two sets. In the second exercise the students were given a DFA and had to answer questions, such as, what is the initial state, what is the accept state, does this DFA accept a certain input string, and does it accept the empty string ϵ. In the third and fourth exercise the students had to construct a DFA which accepted a language that was given in the exercise.

We chose to keep the time span for each activity short. By keeping the time span short, we made sure that the students were engaged with the given activity. This also meant that the number of exercises that the students had to solve during the exercise sessions was limited to a small set of exercises. In order to solve all of the exercises during the allocated time, the student had to through all of the exercises at a good phase.

The participants were also anonymous. We were only interested in tracking their progress throughout the experiment and knowing whether they started out by doing exercises by pencil and paper or if they started out by using the learning tool. Each participant got an id which consisted of 4 characters, for example, *B302*. The first character of the id was either *A* or *B*, and indicated which medium this participant started out with. In this case since it was a *B*, then that participant started out by doing exercises using the learning tool. The second character of the id indicated what test the participant had performed. In this case *B3*, told us that it was the third and final test. The last two remaining characters of the id indicated which participant it was. In this case *B302*, told us that it was participant 2 of group B.

2.2 The Results

In this section we present the interesting results of the experiment. The complete tables of results can be seen in the Appendix.

Here, we were only interested in the columns of P and C, which we have visualized using a histogram. These two columns indicates the improvements of the participants. It should also be noted that the green column in the histograms represents the average points gained.

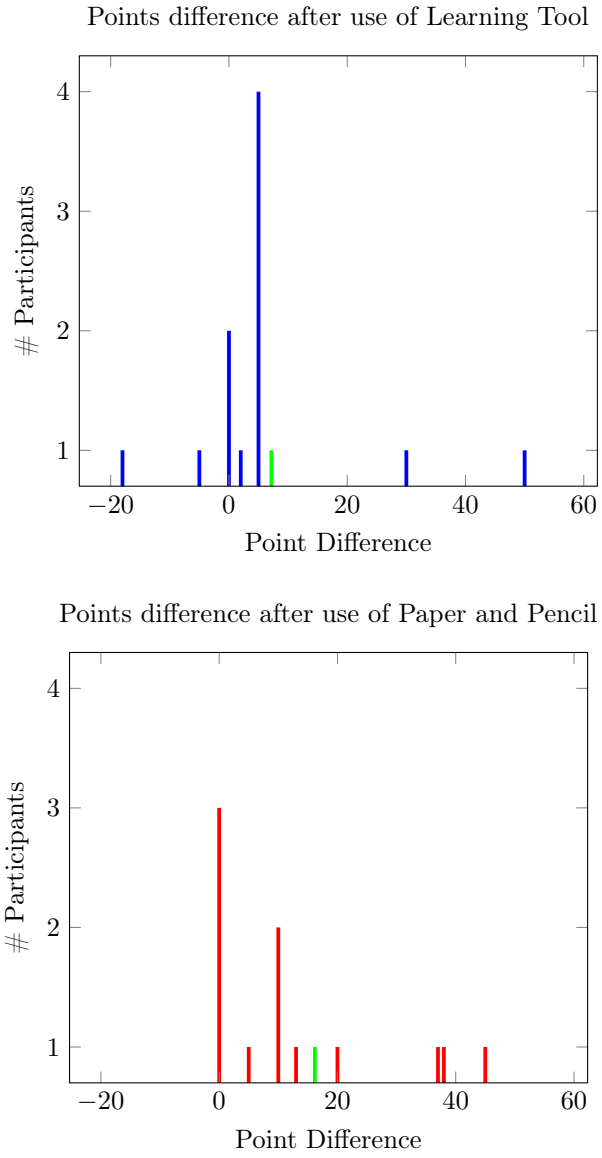

Table 2.2: Points difference after use of Learning Tool

2.3 Reflections & Discussion

In this section we discuss the results gathered from the experiment and the experiment in general.

The setup we chose for the experiment, which can be seen in Table 2.1, was chosen due to the fact that it was robust since it interleaves the two types of exercise sessions. The main objective of the experiment was to measure the improvements of the participants from one test to another. By interleaving the two types of exercise sessions, we made sure that we gathered solid results that could tell the improvements from one test to another. From the interleaving we were also able to tell which of the two mediums was most beneficial.

The results we gathered indicated that paper and pencil yielded a more positive change in points, than compared to the use of the learning tool. It might be the case that the participants which started out doing paper and pencil exercises, did not improve as much after the use of the learning tool, as they did after the initial exercise session with paper and pencil. It might also be case that the learning tool is lacking a bit of functionality and perhaps a bit more user-friendliness, in order to gather as much positive change in points, as the paper and pencil session.

Some of the limitations we had while conducting this experiment was that, the group size of participants was not too big. If we had more participants, the results of the experiment would have been even stronger and the gap in points difference might have been smaller. Also, we would have been able to introduce a control group. With a control group we could have conducted a whole other experiment, where the control group only were allowed to perform the tests.

With these results we can say for sure that there are improvements to be made to the learning tool. The results also indicated that we are heading in the right direction. The reason why we can state that we are heading in the right direction even though the results might indicate otherwise is that, it was not the learning tool itself that caused the difficulties. The participants did not complain about the learning tool being too challenging to use. We can see that it caused learning for some of the participants, this can be seen in the Table 6.1 and Table 6.2 in the Appendix. We still need to look more into the design of the learning tool, the information contained in each exercise, and the feedback each exercise provide. If we take a step back and look at these different aspects, then it is our belief that the learning tool would be able to compete better against the traditional use of paper and pencil.

Chapter 3

Theory

In this chapter we present the theory for two fundamental types of automata, i.e. NFA and PDA. Both NFA and PDA have been implemented in the learning tool according to their formal definitions.

It should be noted that all of the definitions in this chapter is from the book *Introduction to the Theory of Computation* by Michael Sipser [2].

3.1 Non-deterministic Finite Automata

Definition 1 (NFA)**.** A Non-deterministic Finite Automaton is a 5-tuple $(\mathcal{Q}, \Sigma, \delta, q_0, F)$.

1. $\mathcal{Q}$ is a finite set of states,
2. Σ is a finite alphabet,
3. $\delta : \mathcal{Q} \times \Sigma_\epsilon \longrightarrow \mathcal{P}(\mathcal{Q})$ is the transition function,
4. $q_0 \in \mathcal{Q}$ is the start state, and
5. $F \subseteq \mathcal{Q}$ is the set of accept states.

For any alphabet Σ we write Σ_ϵ to be $\Sigma \cup \{\epsilon\}$.

Definition 2 (Words)**.** Given a finite alphabet, Σ, a word on Σ, w, is a finite sequence

$$w = a_0 a_1 ... a_n$$

where $n \in \mathbb{N}$, and $\forall i \leq n . a_i \in \Sigma$. Let Σ^* denote set of all words on Σ.

Definition 3 (Accept)**.** Given a NFA $M = (\mathcal{Q}, \Sigma, \delta, q_0, F)$ and a word, $w = a_0a_1...a_n \in \Sigma^*$, we say that M accepts w iff a sequence of states $s_0, s_1, ..., s_n$ in S exists such that:

1. $s_0 = q_0$
2. $s_{i+1} \in \delta(s_i, a_{i+1})$, for $i = 0, ..., n-1$, and
3. $s_n \in F$.

Condition 1 states that M start in the start state. Condition 2 states that state s_{i+1} is one of the allowable next states when M is in state s_i and reading a_{i+1}. Finally, condition 3 states that M accepts its input if the last state is an accept state.

3.2 Pushdown Automata

Definition 4 (PDA)**.** A Pushdown Automaton is a 6-tuple $(\mathcal{Q}, \Sigma, \Gamma, \delta, q_0, F)$, where $\mathcal{Q}, \Sigma, \Gamma$ and F are all finite sets, and

1. $\mathcal{Q}$ is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta : \mathcal{Q} \times \Sigma_\epsilon \times \Gamma_\epsilon \longrightarrow \mathcal{P}(\mathcal{Q} \times \Gamma_\epsilon)$ is the transition function,
5. $q_0 \in \Sigma$ is the start state, and
6. $F \subseteq \mathcal{Q}$ is the set of accept states.

Definition 5 (Accept)**.** A Pushdown Automaton $M = (\mathcal{Q}, \Sigma, \Gamma, \delta, q_0, F)$ computes as follows. It accepts input w if w can be written as $w = w_1, w_2, ..., w_m$, where each $w_i \in \Sigma_\epsilon$ and sequences of states $r_0, r_1, ..., r_m \in \mathcal{Q}$ and string $s_1, s_2, ..., s_m \in \Gamma^*$ exist that satisfy the following three conditions. The string s_i represents the sequence of stack contents that M has on the accepting branch of the computation.

1. $r_0 = q_0$ and $s_0 = \epsilon$,
2. For $i = 0, ..., m-1$, we have $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, where $s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\epsilon$ and $t \in \Gamma^*$, and
3. $r_m \in F$.

Condition 1 states that M start out in the start state with an empty stack. Condition 2 states that M moves according to the state, stack and next input symbol. Condition 3 states that an accept state occurs at the input end.

3.3 Context Free Grammar

Definition 6 (CFG)**.** A context-free-grammar is a 4-tuple (V, Σ, R, S)

1. V is a finite set called the *variables*,
2. Σ is a finite, disjoint from V, called the *terminals*,
3. R is a finite set of *rules*, with each rule being a variable and a string of variables and terminals, and
4. $S \in V$ is the start variable.

Definition 7 (CNF)**.** A context-free-grammar is in **Chonsky normal form** if every rule is of the form

1. $A \longrightarrow BC$
2. $A \longrightarrow a$

where a is any terminial and A, B, and C are any variables - except that B and C may not be the start variable. In addition, we permit the rule $S \longrightarrow \epsilon$, where S is the start variable.

The following definition is from the book *Languages and machines: an introduction to the theory of computer science*[3].

Definition 8 (GNF)**.** A context-free-grammar G = (V, Σ, R, S)
is in **Greibach normal form** if each rule is of the form:

1. $A \longrightarrow aA_1A_2 \ldots A_n$
2. $A \longrightarrow a$
3. $S \longrightarrow \lambda$

where $a \in \Sigma$, and $A_i \in V - \{S\}$ for $i = 1, 2, \ldots n$.

Chapter 4

The Learning Tool

In this chapter we present the features we implemented in the learning tool after the experiment.

4.1 Extensions to the Learning Tool

Before extending the learning tool with the new features, the learning tool only contained exercises which had something to do with DFA's. After implementing the new features the learning tool was able to deal with NFA's and PDA's.

Usually when we implemented new features, we kept in mind the learning aspects of things. But, during this implementation phase we switched our focus to the actual learning tool itself. We switched focus because we wanted the learning tool to contain small sandboxes, which we like to refer to as *playgrounds*, such that the user had the opportunity to experiment with different creations of the newly implemented features. Then, in another iteration of the implementation phase the focus could switch back to the learning aspect to make the *playgrounds* more intuitive and user-friendly.

4.1.1 The Implementation of NFA

When we began to work on the implementation of the NFA class we re-used the internal representation of the DFA class, and modified it to fit with the formal definition of NFA, which can be found in Definition 1 in Section 3.1.

We had to extend the DFA class, such that, it was capable of dealing with epsilon transitions. Not only did we add epsilon transitions, we also had to take

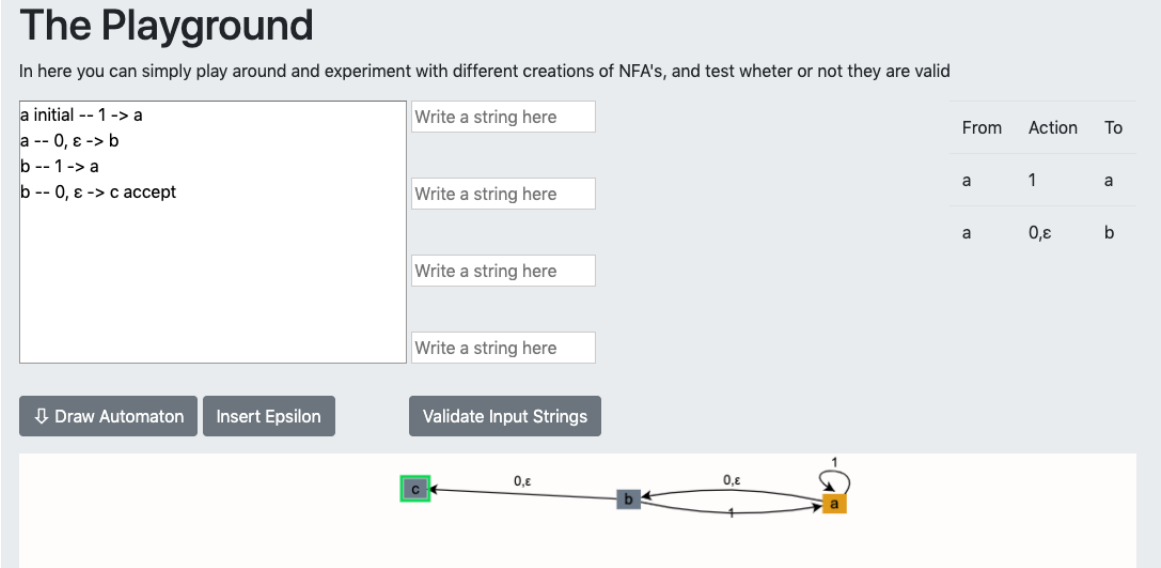

Figure 4.1: NFA - Playground

into consideration non-determinism. Non-determinism is in this case whenever we encountered an epsilon transition, when encountered we had to "split" the automaton. One automaton where we followed the epsilon transition and another automaton where we followed the transition value from that edge. In order to coupe with non-determinism we had to implement a model checker. We will go into more details about the model checker in Section 3.2.

During this process we also came up with the idea of a transition table. The transition table is a table that tells the user that, from the current state which we are in, we can perform a certain transition, and end in another state. We also had to create a new parser which was able to handle epsilon transitions.

In Figure 4.1 the result of the implementation can be seen.

4.1.2 The Implementation of PDA

When we began to work on the implementation of the PDA class we had to start from scratch. There was not any prior existing code which we could re-use, like we did with the NFA's. This time we also had to modify the graphical library, Springy [4], which we used to display the automata on the canvas. We had to modify the way Springy presented the edges between states. The edges had to be presented as they usually are on a PDA, like those that can be found in

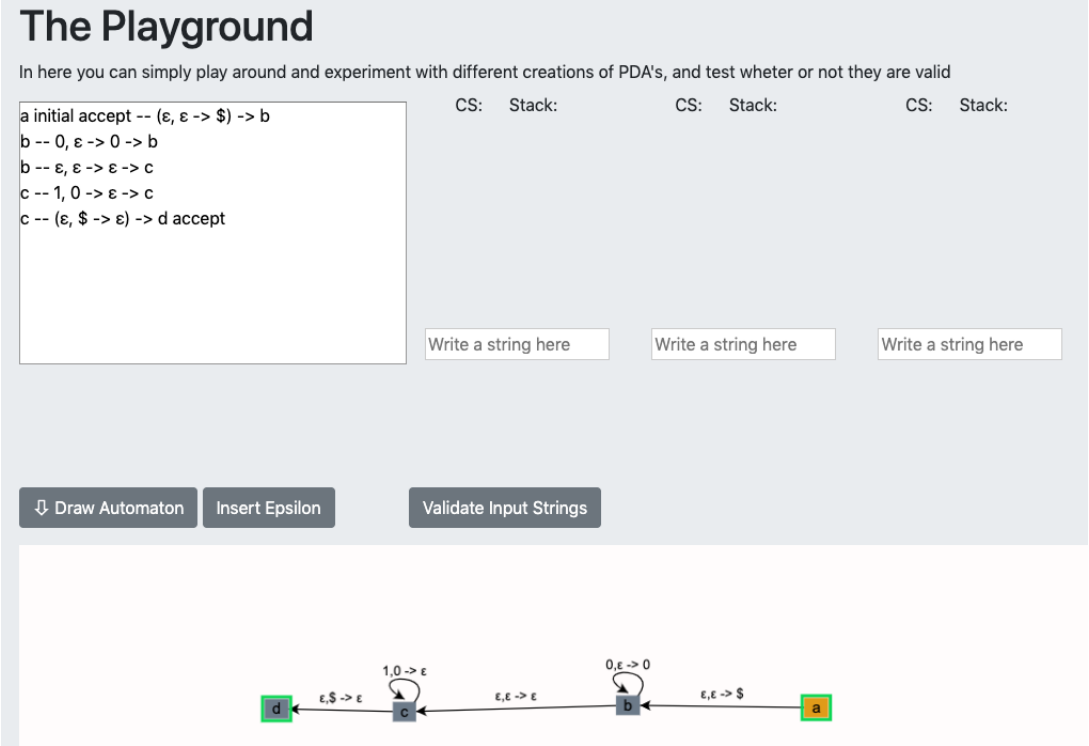

Figure 4.2: Simple PDA

Introduction to the Theory of Computation.

When it came to the creation of the PDA parser, we simply had in mind that we wanted to have the same or similar syntax to the one in *Introduction to the Theory of Computation*. We ended up with a mix of our syntax and the syntax from the book.

The initial PDA class was able to work with small and simple PDA's defined according to its formal definition, which can be found in Section 3.2.

An example of a simple PDA can be seen in Figure 4.2, this PDA recognizes the language $\{0^n1^n | n \geq 0\}$. Meaning that there has to be an equal amount of $0's$ and $1's$.

After the initial implementation, we extended the PDA class to be able to recognize and accept more interesting languages, such as
$\{a^ib^jc^k \mid i, j, k \geq 0 \text{ and } i = j \text{ or } i = k\}$. Meaning that, either it is the case that there is an equal amount of $a's$ and $b's$ or there is an equal amount of $a's$ and $c's$. An example of this language can be seen in Figure 4.3.

In order to implement this extended PDA class, we first had to implement something which was able to deal with non-determinism.

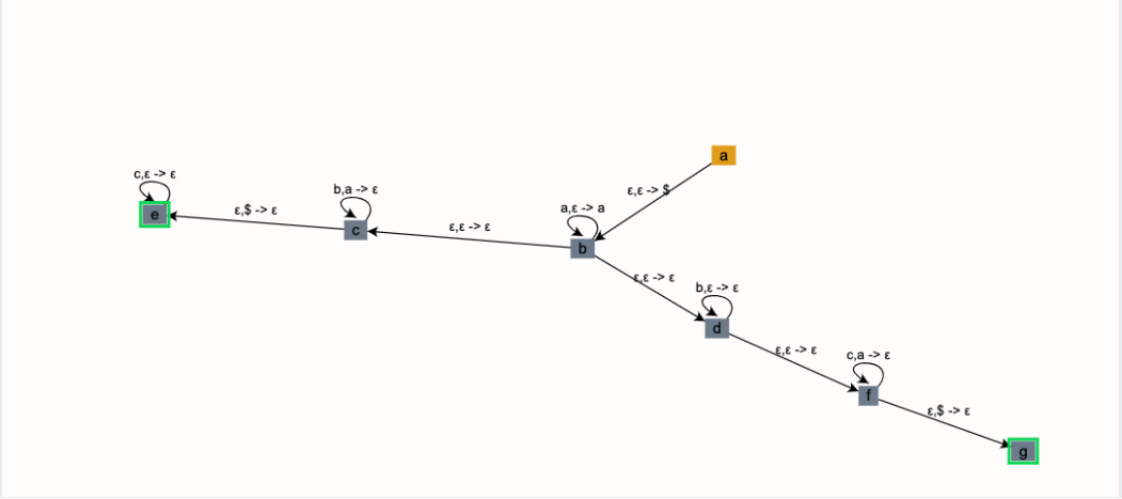

Figure 4.3: A more interesting PDA

We dealt with non-determinism in the same manner as we did with the NFA's, we used a model checker. Non-determinism in PDA's occurs whenever a state has more than one enabled outgoing transition. We wanted to explore all of the outgoing transition to see which of the transitions ended in a potentially accept state. We also defined an epsilon transition, that is whenever the transition had the form: $\epsilon, \epsilon \rightarrow \epsilon$.

There were four cases where epsilons occurred in a transition. Case 1, the transition $a, \epsilon \rightarrow a$. Here we simply added the input value a to the stack without consulting the stack itself. Case 2, the transition $b, \epsilon \rightarrow \epsilon$. Here we read the input value and did nothing with it. Case 3, the transition $b, a \rightarrow \epsilon$. Given that the input value was b and the top stack element was a then we proceeded to pop that element from the stack. Case 4, the transition $\epsilon, \epsilon \rightarrow \epsilon$. This is a "free" transition.

Whenever we encountered a state which had more than one enabled outgoing transition we would "split" the automaton, and explore each of the outgoing transitions. If it was the case that one of the outgoing transitions ended up in an accept state, then it was reported back that there existed a valid path from the state which we were in that lead to an accept state. The way we performed explorations when traversing through a PDA is defined according to Definition 5 in Chapter 3.

The pseudo code for the modified implementation of breadth-first-search (BFS)

can be seen in Algorithm 1

Algorithm 1 Modified breadth-first-search

```
1:  procedure BFS(automaton, state, word, stack)
2:      maxCall ← 0
3:      S ← new Queue
4:      S.push(id : 0, CS : state, word : word, stack : stack)
5:      while S.length > 0 & maxCall < 1000 do
6:          V ← S.pop
7:          if V.CS is accept state & V.word empty & V.stack empty then
8:              return V.CS
9:          for all edge in edges do
10:             if edge can be followed then
11:                 Update stack according to word and edge
12:                 S.push(edge)
13:                 maxCall ← maxCall + 1
```

It should also be noted that after we created and implemented this Algorithm. We then modified it to work with the NFA's, the difference between those two versions is that the stacking handling has been removed.

A trace is defined as a list of objects, and the data structure of each object can be seen in Listing 4.1.

```
interface ObjStructure {
  id: number
  CS: string    // CurrentState
  word: string
  stack: string
  previousID: number | null
}
```

Listing 4.1: The Data Structure of each Object

Now let us assume that we are about to traverse a PDA, like the one in Figure 4.3. In order to traverse a PDA we have to use the *main* method. This method is defined on the PDA class, and can be seen in Listing 4.2.

```
main = (word: string, state: string) => {
  ....
  const BFSResult = this.BFS(automaton, state, word, stack)

  tracePath.push(vistedPath[vistedPath.length - 1])

  this.getTrace(vistedPath)
  this.displayTrace(tracePath)
  ....
}
```

Listing 4.2: Main Method

The *main* method makes use of the modified BFS approach, which can be seen in Algorithm 1, in order to explore the automaton. When the exploration has been performed and an answer has been found, it then extracts the generated trace and displays that trace to the user.

4.2 Testing

In this section we present the testing of the newly implemented features. During the conduction of the tests we chose to use a White-box testing approach, the reason for that, was that we were interested in the behavior of the newly implemented features. The only way we could grantee that the behavior of the features was correct, was by looking underneath the hood while performing the tests and validations.

4.2.1 NFA Testing

The primary automaton we used to test and validate the NFA class which was the one in Figure 4.1. We chose this automaton because we could test different properties, such as, epsilon walks and regular input strings. If we gave this automaton the empty input string, ϵ, we expected that the outcome of traversing this would yield in an accept state. Also, if we provided the automaton with input strings, such as, 0, 1, 01, 00, 11 then we would also expect it to end in an accept state, and this was mainly due to the epsilon walks that could be performed no matter what state we were in. In order for the automaton to reject an input string it would have to violate Definition 3, and the shape of the input strings had to be in a specific manner. A rejected input string would have to contain 00 followed by any arbitrary 0's or 1's, examples of such strings could be 01001, 000 and 001. The results can be seen in Figure 4.4.

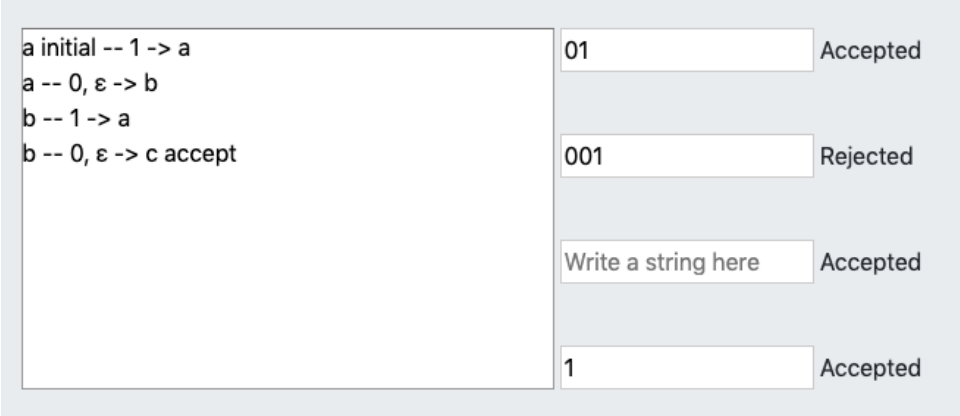

Figure 4.4: NFA Test Results

4.2.2 PDA Testing

The test and validation of the PDA class was a bit more exhaustive than of the NFA class. This time around we had two primary automata, those can be seen in Figure 4.2 and Figure 4.3. The automaton in Figure 4.2 was used to test and validate our implementation of the basic *push* and *pop* functionality and to make sure that the stack also functioned properly as well. If we provided the automaton with the input strings, such as, 01 and, ϵ we then would expect that the outcome would yield in an accept state. On the other hand, if we provided the automaton with an input string 001. We then would expect the outcome to yield in an error state. The results of those input strings can be seen in Figure 4.5.

We used Figure 4.3 to test and validate the behavior of the model checker we implemented. This automaton had two different epsilon branches that it could explore at state *b*, therefore we chose this automaton as our primary for testing and validating the model checker. If we gave this automaton the input strings of *aabbc*, *aabcc*, and *abc* then the expected outcome of the three input strings would yield in an accept state. The results of the input strings can be seen in Figure 4.6.

While testing and validating the two automata, we also had to test and validate other functionality within the *main* method (Listing 4.2). We needed to make sure that the methods *getTrace()* and *displayTrace()* functioned as anticipated. The *getTrace()* method extracted the generated trace. A generated trace could exist of a path that went from the initial state to either an accept state or to an error state. Then the *displayTrace()* method took the generated trace and displayed that to the user on the website. In the following we present two

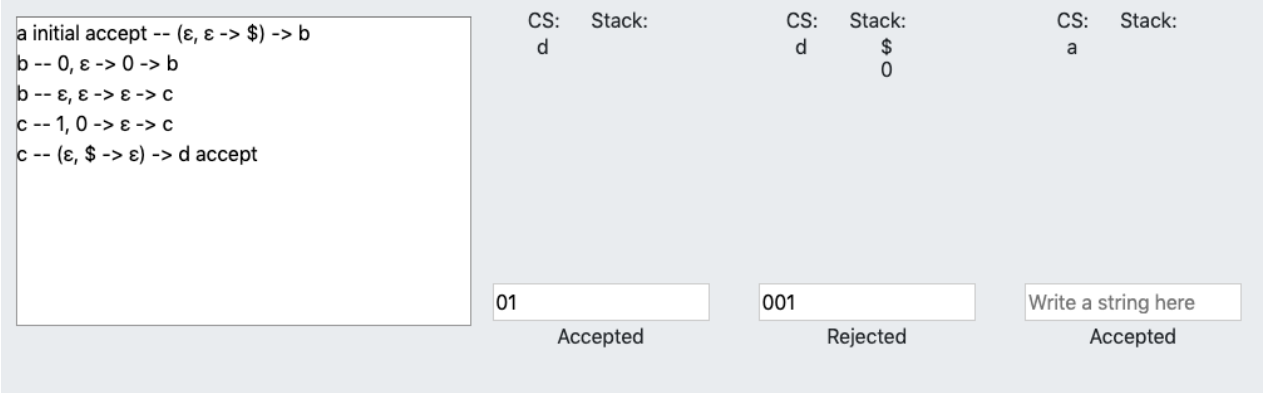

Figure 4.5: PDA Simple Results

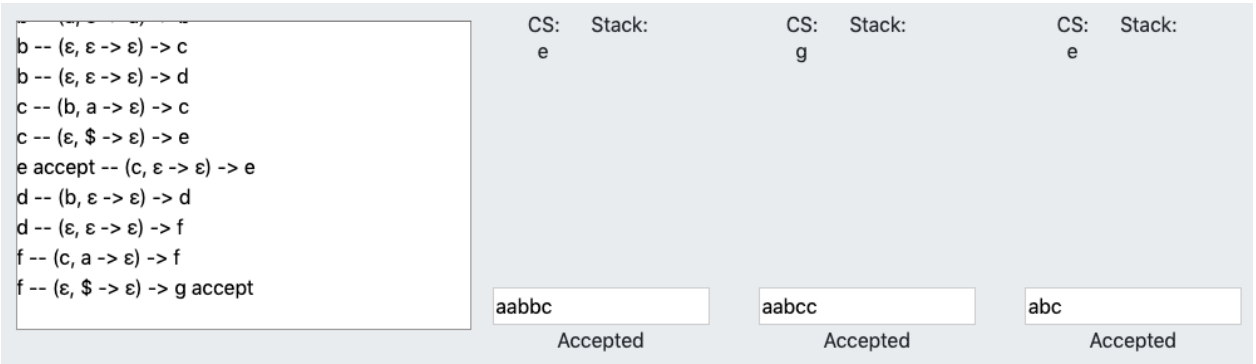

Figure 4.6: PDA Model Checker Results

generated traces, one trance which lead to an error state, and another trace which lead to an accept state.

In Listing 4.3 the generated trace for the input string *aabccc* is shown, this input string was rejected by the automaton since there was not an equal number of $a's$ and $c's$ or $a's$ and $b's$.

```
{id: 0, CS: "a", word: "єaaєbєcccє", stack: "", previousID: null}
{id: 1, CS: "b", word: "aaaєbєcccє", stack: "$", previousID: 0}
{id: 2, CS: "b", word: "aaєbєcccє", stack: "$a", previousID: 1}
{id: 3, CS: "b", word: "aєbєcccє", stack: "$aa", previousID: 2}
{id: 4, CS: "b", word: "єbєcccє", stack: "$aa", previousID: 3}
{id: 5, CS: "c", word: "bєcccє", stack: "$a", previousID: 4}
{id: 6, CS: "d", word: "bєcccє", stack: "$aa", previousID: 4}
{id: 7, CS: "c", word: "", stack: "$a", previousID: 5}
{id: 8, CS: "d", word: "єcccє", stack: "$aa", previousID: 6}
{id: 9, CS: "e", word: "", stack: "$a", previousID: 7}
{id: 10, CS: "f", word: "cccє", stack: "$a", previousID: 8}
{id: 11, CS: "f", word: "ccє", stack: "$", previousID: 10}
{id: 12, CS: "f", word: "", stack: "$", previousID: 11}
{id: 13, CS: "f", word: "", stack: "$", previousID: 12}
```

Listing 4.3: Trace Fail - aabccc

In Listing 4.4 we have shown the generated trace for the input string *abc*. This was of course a valid input string, and one which got accepted by the automaton.

```
{id: 0, CS: "a", word: "єaєbєcє", stack: "", previousID: null}
{id: 1, CS: "b", word: "aaєbєcє", stack: "$", previousID: 0}
{id: 2, CS: "b", word: "aєbєcє", stack: "$a", previousID: 1}
{id: 3, CS: "b", word: "єbєcє", stack: "$a", previousID: 2}
{id: 4, CS: "c", word: "bєcє", stack: "$", previousID: 3}
{id: 5, CS: "d", word: "bєcє", stack: "$a", previousID: 3}
{id: 6, CS: "c", word: "єcє", stack: "", previousID: 4}
{id: 7, CS: "d", word: "єcє", stack: "$a", previousID: 5}
{id: 8, CS: "e", word: "cє", stack: "", previousID: 6}
{id: 9, CS: "f", word: "cє", stack: "$", previousID: 7}
{id: 10, CS: "e", word: "є", stack: "", previousID: 8}
```

Listing 4.4: Trace Success - abc

With the tests and traces presented in this subsection and with their outcome. We can confidently state that our implementation of the PDA and model checker functioned as expected.

Chapter 5

Discussion & Conclusion

In this chapter we discuss the results gathered throughout this report, then we conclude and present ideas of what could further be implemented to this learning tool.

5.1 Discussion

At the start of this report we conducted an experiment, and the result of the experiment indicated that the traditional use of paper and pencil was better than the use of our learning tool. This was of course not the result we wanted, but it only motivated us to make the learning tool even better. We do believe that after all of the work implemented throughout this report, that the learning tool is in a much better position than before. The reflections and discussions of the conducted experiment can be seen in Section 2.3.

After the experiment had been conducted we began to implement the new features, i.e. NFA and PDA. We made a choice to implement those as playgrounds only, the reason for that was we wanted the users to be able to play around and experiment with those newly added automata and not necessarily be able to do exercises in the first iteration of the implementation.

During the implementation process we also had to take into consideration how we wanted to deal with non-determinism. It was quite challenging at first, since we did not have any prior experience with implementing a model checker. The model checker we ended up implementing was capable of performing a BFS when it had to explore the various automata.

For future work it would be a good idea to look more into the design of the learning tool. Make it look more sharp and clean, and take into consideration

the use of a framework when creating this web-based solution. Frameworks that might be of interest are: Express[5], and Angular[6]. Besides of migrating to a framework, the visual aspects of the learning tool could also be improved. It might be a good idea to make more use of the canvas, which display the automata. Make it such that, whenever the user is testing an input string on an automata then visualize the progress of exploration on the canvas as well. This could potentially increase the users understanding of how the automaton explores the given input string.

By making everything more visual, we might then reduce the complexity of certain concepts. It might become easier to grasp difficult concepts, which could appear alien at first, when concepts become visualized. It therefore seems to be ideal to find the right balance of visualization and theory.

Regarding the model checker, then it could be beneficial to implement further exploration methods, such as, depth-first-search (DFS). By having more exploration methods implemented the user could visually see the difference of how an automaton gets explored. This could potentially also increase the users understanding.

There are pros and cons of the two exploration methods, BFS and DFS. With BFS we are guarantied to get an answer if the input string is accepted or rejected. The same can not be said for DFS, since it might be the case that we could encounter an epsilon loop and keep exploring that forever. There are ways to deal with infinite epsilon loops by taking the given PDA and then convert it into a context-free-grammar (CFG), and then convert that into a Greibach-Normal-Form (GNF). The conversion from a CFG to a GNF can be found in the book *Languages and machines: an introduction to the theory of computer science*[3] on page 122 under Theorem 5.6.3. The form of a GNF can be seen in Definition 8.

5.2 Conclusion

We have in this report succeeded in further developing the learning tool, we achieved the goal which we had in mind for this report. We succeeded in implementing two other fundamental automata, i.e. NFA and PDA. By implementing those two fundamental automata, we also succeeded in implementing a model checker which helps with the exploration of the two new automata. This implementation have further increased the strength and robustness of the learning tool.

There is still work that needs to be done that could further increase the users experience with the learning tool and hopefully decrease the complexity of the various concepts.

Chapter 6

Appendix

6.1 The Experiment Results

It should be noted that the two columns, P and C, indicates the difference in points gained or lost after the use of either pencil and paper or the learning tool. The amount of total points that could be gathered was 100.

	Exercise 1	Exercise 2	Exercise 3	Exercise 4	Total Points	P	C
A106	8	15	0	15	38		
A206	8	15	0	15	38	0	
A306	8	15	0	20	43		5
A105	8	15	0	0	23		
A205	8	15	5	5	33	10	
A305	8	15	0	0	33		0
A104	10	12	5	10	37		
A204	10	12	5	30	57	20	
A304	10	12	10	30	62		5
A103	6	12	10	0	28		
A203	8	12	45	0	65	37	
A303	8	12	45	30	95		30
A102	6	12	0	0	18		
A202	6	15	5	5	31	13	
A302	6	15	5	5	31		0
A101	10	12	0	30	52		
A201	10	15	35	30	90	38	
A301	10	15	40	30	95		5
Total						118	45
Avg						19.7	7.5

Table 6.1: Results of the Experiment for Group A

	Exercise 1	Exercise 2	Exercise 3	Exercise 4	Total Points	P	C
B105	10	15	0	20	45		
B205	10	15	40	30	95		50
B305	10	15	45	30	100	5	
B104	10	15	40	30	95		
B204	10	15	35	30	90		-5
B304	10	15	45	30	100	10	
B103	10	15	0	10	35		
B203	10	15	0	15	40		5
B303	10	15	0	15	40	0	
B102	8	15	0	0	23		
B202	10	15	0	0	25		2
B302	10	15	0	0	25	0	
B101	8	15	0	20	43		
B201	10	15	0	0	25		-18
B301	10	15	25	20	70	45	
Total						60	34
Avg						12	6.8

Table 6.2: Results of the Experiment for Group B

Bibliography

[1] M. S. Mortensen, "Developing and evaluating an automata learning tool," 2019. `https://projekter.aau.dk/projekter/files/294873768/P9_final.pdf`.

[2] M. Sipser, *Introduction to the Theory of Computation*. Boston, MA: Course Technology, third ed., 2013.

[3] T. A. Sudkamp and A. Cotterman, *Languages and machines: an introduction to the theory of computer science*, vol. 2. Addison-Wesley Reading, Mass., 1988.

[4] D. Hotson *et al.*, "Springy.js." `https://pegjs.org`, 2013.

[5] S. TJ Holowaychuk *et al.*, "Expressjs." `https://expressjs.com`, November 16, 2010.

[6] Google, "Angular." `https://angular.io`, September 14, 2016.