BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

VIZUALIZATION OF AUTOMATA ALGORITHMS

VIZUALIZACE AUTOMATOVÝCH ALGORITMŮ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. JIŘÍ KUCHYŇKA**
AUTOR PRÁCE

SUPERVISOR **doc. Mgr. LUKÁŠ HOLÍK, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

144816

Institut:	Department of Intelligent Systems (UITS)
Student:	**Kuchyňka Jiří, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Application Development
Title:	**Vizualization of Automata Algorithms**
Category:	Algorithms and Data Structures
Academic year:	2022/23

Assignment:

Advanced algorithms for finite automata are sometimes complicated, getting a good feel for their strengths and weaknesses and understanding them in general is often difficult. You will therefore develop a generic system for visualizing the algorithms.

1. Learn about the MATA automata library and the basic automata algorithms.
2. Design a generic system that allows to easily visualize runs automata algorithms.
3. Implement your system in the MATA library.
4. Demonstrate the use of the system on basic algorithms, ideally also on some advanced algorithms (such as computing a simulation relation, anti-chain inclusion testing, ...).

Literature:

- Parosh Aziz Abdulla, Yu-Fang Chen, Lukás Holík, Richard Mayr, Tomás Vojnar: When Simulation Meets Antichains. TACAS 2010: 158-174
- https://github.com/VeriFIT/mata
- HOLÍK Lukáš a ŠIMÁČEK Jiří. Optimizing an LTS-Simulation Algorithm. *Computing and Informatics*, roč. 2010, č. 7, s. 1337-1348. ISSN 1335-9150. Dostupné z: http://www.cai.sk/ojs/index.php/cai/article/view/147

Requirements for the semestral defence:
1,2, ideally a part of 3

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Holík Lukáš, doc. Mgr., Ph.D.**
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	24.5.2023
Approval date:	3.11.2022

Abstract

The goal of this thesis is to design and implement a generic system to visualize algorithms that operate over automata. The resulting system completely separates the part that is dedicated to generating data for visualization and the part that is dedicated to visualizing it. The system only specifies their communication interface. This thesis focuses on integrating such system into existing libraries in a way to minimize the requirements on the programmer to start visualizing the state of their algorithm. The thesis also briefly discusses the possibilities of using such a system to visualize the state of an algorithm while stepping through the program during debugging. The proposed system can be used for teaching, research, and practical applications in automata theory. In the future, the system could be extended with tools to visualize Turing machines and algorithms operating on them.

Abstrakt

Tato práce se zabývá návrhem a implementací generického systému k vizualizaci algoritmů, které pracují nad automaty. Výsledný systém zcela odděluje část, která se věnuje generování dat k vizualizaci a část, která se věnuje vizualizování. Systém pouze určuje jejich komunikační rozhraní. Práce se zaměřuje na integraci takovéhoto systému do existujících knihoven takovým způsobem, aby byly minimalizovány požadavky na programátora, k tomu aby mohl vizualizovat stav svého algoritmu. Práce se také zkráceně věnuje možnostem využití tohoto systému k vizualizaci stavu algoritmu při krokování programem během ledění. Navržený systém může být použit pro výuku, výzkum a praktické aplikace v oblasti teorie automatů. V budoucnu by mohl být systém rozšířen o nástroje k vizualizaci turingových strojů a algoritmů pracujících nad nimi.

Keywords

automata, automaton, NFA, DFA, visualization, debugging, algorithms, automata algorithms, automata visualization, graphs, GraphViz, data streaming, Jupyter Notebook, realtime visualization, automata algorithm debugging

Klíčová slova

automaty, NFA, DFA, vizualizace, ladění programu, algoritmy, automatové algoritmy, vizualizace automatů, grafy, GraphViz, streamování dat, Jupyter Notebook, vizualizace v reálném čase, debuggování automatových algoritmů

Reference

KUCHYŇKA, Jiří. *Vizualization of Automata Algorithms.* Brno, 2023. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. Mgr. Lukáš Holík, Ph.D.

Vizualization of Automata Algorithms

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of doc. Mgr. Lukáše Holíka, Ph.D. I have listed all the literary sources, publications, and other sources that were used during the preparation of this thesis.

.....
Jiří Kuchyňka
May 23, 2023

Acknowledgements

I would like to express my deepest gratitude to everyone who has supported me throughout my studies. A special thank you goes to Mgr. Lukáš Holík, Ph.D., for his invaluable advice and feedback during the completion of this thesis.

Contents

Chapter 1

Introduction

Automata have long served as a fundamental basis for solving a variety of computational problems. Their versatility has made them indispensable in the design and analysis of algorithms in numerous domains, including pattern matching, data compression, artificial intelligence, and formal language theory. The ability to effectively represent and manipulate the states and transitions of automata is essential for the development of efficient algorithms and a deeper understanding of their properties.

With the increasing complexity of algorithms operating on automata, it is becoming increasingly difficult to understand how these algorithms work, and there is a greater need to be able to describe how these algorithms work in different ways so that everyone can understand them. One way to describe these algorithms is to visualize their state. For example, we may want to visualize the state of an algorithm during its execution. This can be achieved by running the algorithm in a debugger and stepping through it. The debugger will show us the variables in the computer's memory, and we can get an idea of how the algorithm's code is being executed. But what if, in addition to the code itself and the variables in memory, it would also be possible to display a visualization of the automata in the form in which the algorithm is currently working with it? This is the question that this thesis is trying to address.

There are already tools to visualize automata. These tools are purely dedicated to visualizing automata or, for example, animating the execution of an automata. However, with the need to visualize behavior of an algorithm which is modifying an automata, the visualization needs to include annotations with respect to the state of the code, and one needs to take into account that the automata states will change, split, merge, etc., and these changes need to be represented in an understandable way.

This thesis proposes a standardizable interface for logging the state of the automata algorithm to facilitate its visualization. It also presents a reference implementation of the `GraphViz`-based visualizer and integration for the `mata` library to generate visualization data. The reference implementation employs several custom techniques designed to simplify the process of visualizing any automata algorithms, including special rendering adjustments to ensure consistent positioning of states and edges across all visualized steps of an algorithm, and a user-friendly `print`-like function for logging the state of automata for visualization. By building the system in this manner, it becomes extensible and adaptable, allowing users to create their own version of library integration for their favorite automata library or develop their own visualizer tailored to their specific needs. This flexibility not only fosters innovation in the field of automata visualization, but also encourages the de-

velopment of new techniques and tools, ultimately benefiting researchers and developers working with automata algorithms.

The visualization of automata algorithms provides numerous benefits not only to researchers and developers working in the field, but also to teachers and educators seeking to effectively explain these concepts to students. By visually representing the state transitions, structure, and annotations of an automata during the execution of an algorithm, it becomes easier to comprehend and analyze the underlying mechanics. This fosters a better understanding of the algorithm's behavior, which can lead to more efficient debugging, optimization, and improvement of the algorithm itself. Moreover, this enhanced comprehension allows for clearer communication and collaboration between researchers, as well as a more engaging and intuitive learning experience for students studying automata theory. As a result, the proposed visualization system has the potential to significantly improve the study and development of automata algorithms, as well as enhance educational practices in the field, ultimately leading to more powerful and efficient solutions for complex computational problems and a better-educated generation of computer scientists.

This thesis first discusses the theoretical aspects of automata, their foundations, common models, and corresponding algorithms in chapter 2 and chapter 3. These theoretical foundations will pave the way for understanding of automata algorithms visualization. Following this, chapter 4 and chapter 5 introduce the proposed design for an automata visualization system and a standardizable interface for communication between the two designed sub-systems. The reference implementation of this design is described in chapter 6, followed by an exploration of potential future enhancements in chapter 7. The system is then evaluated by visualizing various automata algorithms in chapter 8. Finally, the thesis concludes in chapter 9 with results and outlines the potential for future developments.

Chapter 2

Automata Background and Foundations

Automata theory is a fundamental area of study within theoretical computer science, providing the foundation for understanding the power and limitations of various computational models. The study of formal languages, automata, and computational complexity allows us to classify and solve real-world problems efficiently. The importance of automata theory can be seen in the development of compilers, natural language processing, pattern recognition, and many other areas of computer science.

In this chapter, we will dive into the basics of automata theory, starting with the Chomsky Hierarchy, which provides a classification of formal languages based on their expressiveness. We will then introduce various automata models, such as finite automata, pushdown automata, and Turing machines, and discuss their computational capabilities. Following this, we will compare deterministic and non-deterministic automata, exploring their similarities and differences. Finally, we examine formal language operations and closure properties, which are essential in determining the properties of languages generated by automata.

2.1 Chomsky Hierarchy

The Chomsky Hierarchy as described by paper [15] and study materials [45], is a classification of formal languages based on their expressiveness and the types of grammars that can generate them. It was proposed by Noam Chomsky in 1956. The hierarchy consists of four levels, each with its corresponding grammar and automaton. As we move up the hierarchy, the languages become less restrictive and more expressive, while the automata become more powerful. The four levels of the Chomsky Hierarchy are as follows:

1. **Type 0:** Recursively enumerable languages, generated by unrestricted grammars. These languages can be recognized by Turing machines.
2. **Type 1:** Context-sensitive languages, generated by context-sensitive grammars. These languages can be recognized by linear-bounded automata.
3. **Type 2:** Context-free languages, generated by context-free grammars. These languages can be recognized by pushdown automata.

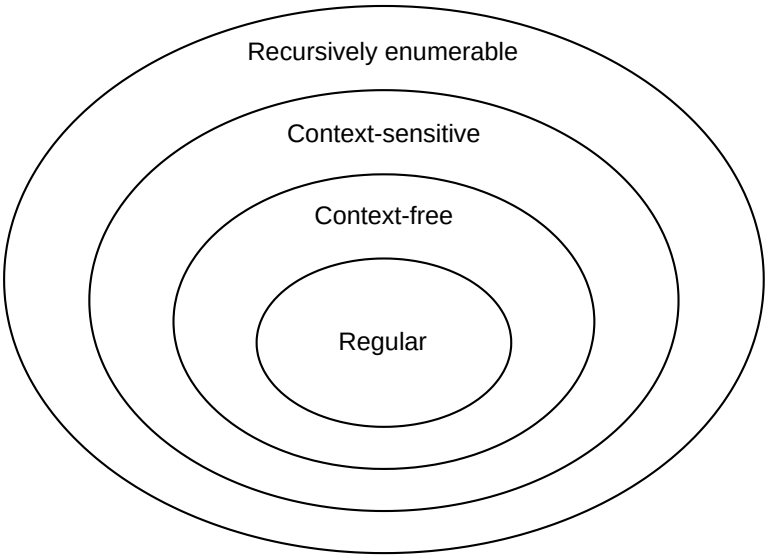

Figure 2.1: The Chomsky Hierarchy, showing the relationship between the four levels.

4. **Type 3:** Regular languages, generated by regular grammars. These languages can be recognized by finite automata.

Figure 2.1 illustrates the Chomsky Hierarchy, showing the relationship between the four levels.

At the lowest level of the Chomsky Hierarchy are the regular languages, which can be expressed using regular expressions and generated by finite automata. These languages are relatively simple and have limited expressive power, but can be processed efficiently. Examples of regular languages include the set of binary strings with an even number of zeros or the set of strings over an alphabet that do not contain a specific substring.

Moving up the hierarchy, we encounter context-free languages that can be generated by context-free grammars and recognized by pushdown automata. Context-free languages are more expressive than regular languages and are commonly used to describe the syntax of programming languages. They can model nested structures, such as parentheses in arithmetic expressions or nested blocks in programming languages.

Next, we have context-sensitive languages, which are generated by context-sensitive grammars and can be recognized by linear-bounded automata. These languages are more expressive than context-free languages, allowing for the representation of more complex structures and dependencies. An example of a context-sensitive language is the set of strings in the form $a^n b^n c^n | n \geq 1$, where the number of a's, b's, and c's are equal.

At the highest level of the Chomsky Hierarchy are the recursively enumerable languages, which can be generated by unrestricted grammars and recognized by Turing machines. These languages represent the most general and expressive class of languages, encompassing all languages that can be recognized by a Turing machine, and are considered to be equivalent to the class of languages that can be computed by an algorithm.

Understanding the Chomsky Hierarchy is crucial for identifying the limitations and capabilities of various computational models, as well as for determining which automata is most suitable for a given problem. In the following sections, we will explore different automata models and their relationships to the Chomsky Hierarchy.

2.2 Automata Models

In this section, we will introduce and discuss various automata models that play a significant role in the study of theoretical computer science. These models include finite automata, pushdown automata, and Turing machines. Each automata model represents a different level of computational power and is used to recognize a particular class of formal languages. This chapter was inspired by book [18] and study materials [45].

Finite Automata

A finite automata (FA) is a simple computational model that is used to recognize regular languages. It consists of a finite set of states, a finite set of input symbols, a transition function, an initial state, and a set of final or accepting states. Formally, a finite automata is defined as a 5-tuple:

$$M = (Q, \Sigma, \delta, q_0, F), \tag{2.1}$$

where:

- Q is a finite set of states,
- Σ is a finite set of input symbols (called the alphabet),
- $\delta : Q \times \Sigma \to Q$ is the transition function,
- $q_0 \in Q$ is the initial state, and
- $F \subseteq Q$ is the set of accepting states.

Finite automata can be deterministic (DFA) or non-deterministic (NFA). The main difference between these two types lies in the transition function, where NFA can transition into multiple states simultaneously while DFA can transition into one state only as can be seen in Figure 2.2.

Pushdown Automata

Pushdown automata (PDA) are a more powerful computational model than finite automata, capable of recognizing context-free languages. A PDA extends the finite automata model by including a stack, a last-in, first-out (LIFO) data structure. The formal definition of a PDA is a 7-tuple:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z, F), \tag{2.2}$$

where:

- Q is a finite set of states,
- Σ is a finite set of input symbols (called the alphabet),
- Γ is a finite set of stack symbols,
- $\delta : Q \times (\Sigma \cup \epsilon) \times \Gamma \to \mathcal{P}(Q \times \Gamma^*)$ is the transition function,
- $q_0 \in Q$ is the initial state,

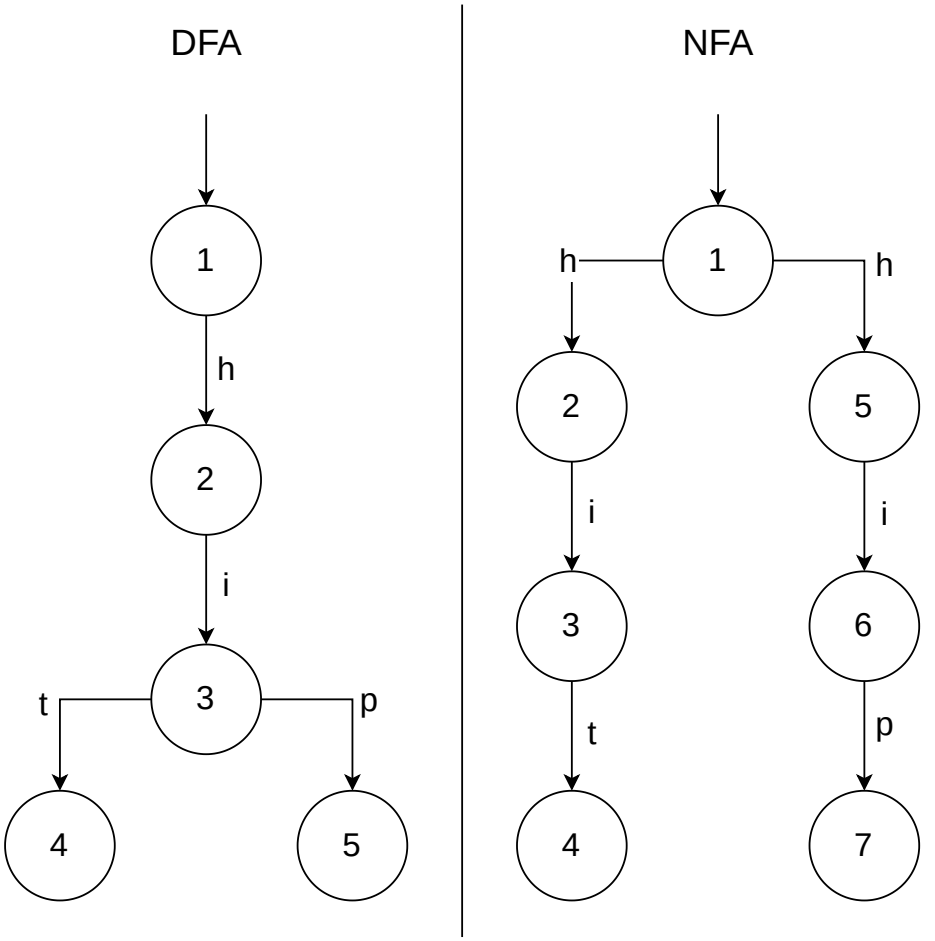

Figure 2.2: An example of a DFA and NFA recognizing the same language.

- $Z \in \Gamma$ is the initial stack symbol, and
- $F \subseteq Q$ is the set of accepting states.

Turing Machines

Turing machines (TM) represent a highly expressive computational model capable of simulating any algorithm. They are used to recognize recursively enumerable languages. A Turing machine consists of a finite set of states, a finite set of input symbols, a tape divided into cells, a tape head that can read and write symbols, and a transition function that defines the machine's behavior. Formally, a Turing machine is defined as a 7-tuple:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject}), \tag{2.3}$$

where:

- Q is a finite set of states,
- Σ is a finite set of input symbols (called the alphabet) not containing the blank symbol $\square$,

- Γ is a finite set of tape symbols, such that $\Sigma \subset \Gamma$ and $\square \in \Gamma$,
- $\delta : (Q \setminus q_{accept}, q_{reject}) \times \Gamma \to Q \times \Gamma \times L, R$ is the transition function,
- $q_0 \in Q$ is the initial state,
- $q_{accept} \in Q$ is the accepting state, and
- $q_{reject} \in Q$ is the rejecting state, with $q_{accept} \neq q_{reject}$.

The Turing machine starts in the initial state q_0 with the input on the tape and the tape head positioned on the leftmost symbol. The machine reads the symbol under the tape head, updates its state according to the transition function, writes a new symbol on the tape, and moves the tape head one position to the left (L) or right (R). The computation continues until the Turing machine reaches the accepting or rejecting state.

Turing machines are considered the most powerful computational model because they can simulate any algorithm, provided that the algorithm can be represented as a sequence of instructions and executed on a computer with unlimited memory. They serve as a theoretical basis for understanding the limits of computation and computability theory.

2.3 Deterministic and Non-deterministic Automata

Deterministic and non-deterministic automata are two distinct ways of modeling computation in automata theory. In this section, we will explore the differences between deterministic and non-deterministic automata, focusing on their definitions and properties. Additionally, we will discuss the implications of these distinctions on the computational power of various automata models, as well as the conversion between deterministic and non-deterministic automata when applicable.

Deterministic Automata

Deterministic automata are characterized by having at most one possible transition for a given state and input symbol. In other words, the automata behavior is fully determined by its current state and the input symbol that it reads. The most common deterministic automata is the deterministic finite automata (DFA), which is defined as a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where:

- Q is a finite set of states,
- Σ is a finite set of input symbols (the alphabet),
- $\delta : Q \times \Sigma \to Q$ is the transition function,
- $q_0 \in Q$ is the initial state, and
- $F \subseteq Q$ is the set of accepting (final) states.

Figure 2.3 shows an example of a deterministic finite automata (DFA) that recognizes the language of strings over the alphabet $\{a, b, c\}$ that match regular expression $(ab) * ac$.

Deterministic pushdown automata (DPDA) and deterministic turing machines (DTM) are other examples of deterministic automata.

```
M = (Q, Σ, δ, q0, F)

Q = {1,2,3}
Σ = {a,b,c}
q0 = 1
F = {3}

δ = {
  (1, a) -> 2,
  (2, b) -> 1,
  (2, c) -> 3,
}
```

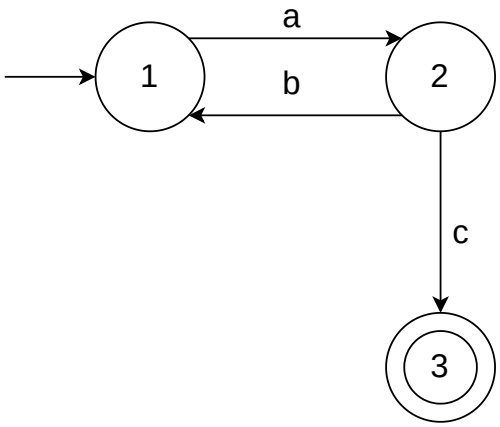

Figure 2.3: An example of a deterministic finite automata (DFA)

Non-deterministic Automata

In contrast to deterministic automata, non-deterministic automata can have multiple transitions for a given state and input symbol. This means that the automata can „choose“ which transition to follow, allowing it to explore multiple possibilities simultaneously. The most common non-deterministic automata is the non-deterministic finite automata (NFA), which is defined as a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where:

- Q, Σ, q_0, and F have the same definitions as in a DFA, and
- $\delta : Q \times \Sigma \to 2^Q$ is the transition function.

Figure 2.4 shows an example of a non-deterministic finite automata (NFA) that recognizes the language of strings over the alphabet $\{a, b, c\}$ that match regular expression $(ab) * (ac|a)$. Note that the NFA has two possible transitions for the state q_1 and input symbol a, allowing it to transition into either q_1 or q_4.

Non-deterministic pushdown automata (NPDA) and non-deterministic Turing machines (NTM) are other examples of non-deterministic automata.

Computational Power and Conversions

The distinction between deterministic and non-deterministic automata has implications for their computational power. For instance, while DFAs and NFAs are equivalent in terms of the languages they can recognize (regular languages), deterministic and non-deterministic pushdown automata are not equivalent in their capabilities. NPDAs are strictly more powerful than DPDAs, as they can recognize context-free languages, whereas DPDAs can recognize only a proper subset of context-free languages.

In cases where deterministic and non-deterministic automata are equivalent in computational power, such as DFAs and NFAs, it is possible to convert between the two models.

```
M = (Q, Σ, δ, q0, F)

Q = {1,2,3,4}
Σ = {a,b,c}
q0 = {1,2}
F = {3,4}

δ = {
  (1, a) -> {2,4},
  (2, b) -> {1},
  (2, c) -> {3},
}
```

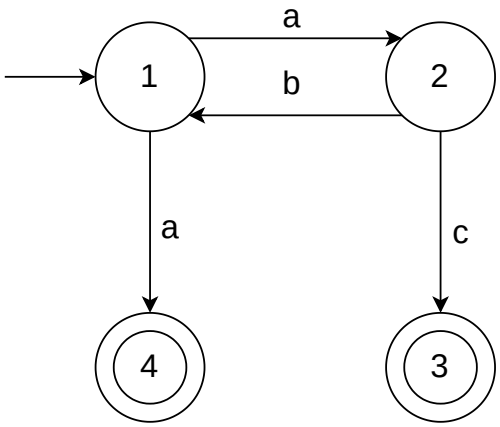

Figure 2.4: An example of a non-deterministic finite automata (NFA)

For example, an NFA can be transformed into an equivalent DFA using the powerset construction, which creates a new DFA where each state corresponds to a subset of the original NFA states. The transition function for the new DFA is defined on the basis of the original NFA's transitions and the subsets of states. The conversion process can be computationally expensive, since the number of states in the resulting DFA may be exponential to the number of states in the original NFA.

However, for more powerful automata, such as pushdown automata and Turing machines, the equivalence between deterministic and non-deterministic models does not always hold. As mentioned above, DPDAs and NPDAs differ in their computational capabilities, with NPDAs being strictly more powerful. Consequently, there is no general conversion method for transforming an NPDA into an equivalent DPDA.

2.4 Formal Language Operations

Formal language operations are essential to understand the interactions between languages and how they can be combined or manipulated to create new languages. This section will discuss the most common formal language operations: union, concatenation, intersection, complement, and Kleene star, as well as providing formal definitions for each operation and illustrations of their differences with figures.

Union

The union operation takes two languages L_1 and L_2 as input and produces a new language L_3 consisting of all the strings that are in either L_1 or L_2. The formal definition of the union operation is as follows:

$$L_3 = L_1 \cup L_2 = \{w | w \in L_1 \text{ or } w \in L_2\} \tag{2.4}$$

Concatenation

The concatenation operation takes two languages L_1 and L_2 and produces a new language L_3 consisting of all possible strings formed by concatenating a string from L_1 with a string from L_2. The formal definition of the concatenation operation is as follows:

$$L_3 = L_1 \cdot L_2 = \{xy | x \in L_1 \text{ and } y \in L_2\} \tag{2.5}$$

Figure 2.5 shows example of the concatenation operation between two languages L_1 and L_2. The resulting language L_3 contains all strings formed by concatenating a string from L_1 with a string from L_2.

$$\begin{aligned} L_1 &= \{asdf, uvw\} \\ L_2 &= \{ghjk, xyz\} \\ L_3 &= L_1 \cdot L_2 = \{asdfghjk, asdfxyz, uvwghjk, uvwxyz\} \end{aligned} \tag{2.6}$$

Figure 2.5: Example of the concatenation operation between two languages L_1 and L_2.

Intersection

The intersection operation takes two languages L_1 and L_2 and produces a new language L_3 consisting of all the strings present in both L_1 and L_2. The formal definition of the intersection operation is as follows:

$$L_3 = L_1 \cap L_2 = \{w | w \in L_1 \text{ and } w \in L_2\} \tag{2.7}$$

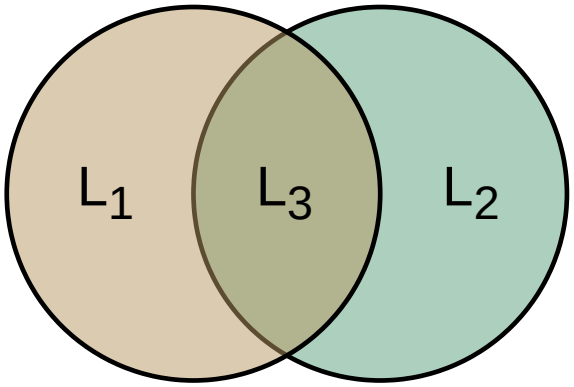

Figure 2.6: Illustration of the intersection operation between two languages L_1 and L_2.

Figure 2.6 illustrates the intersection operation between two languages L_1 and L_2. The resulting language L_3 contains all the strings present in both L_1 and L_2.

Complement

The complement operation takes a language L and an alphabet Σ and produces a new language L' consisting of all the strings over Σ that are not in L. The formal definition of the complement operation is as follows:

$$L' = \overline{L} = \{w \in \Sigma^* | w \notin L\} \tag{2.8}$$

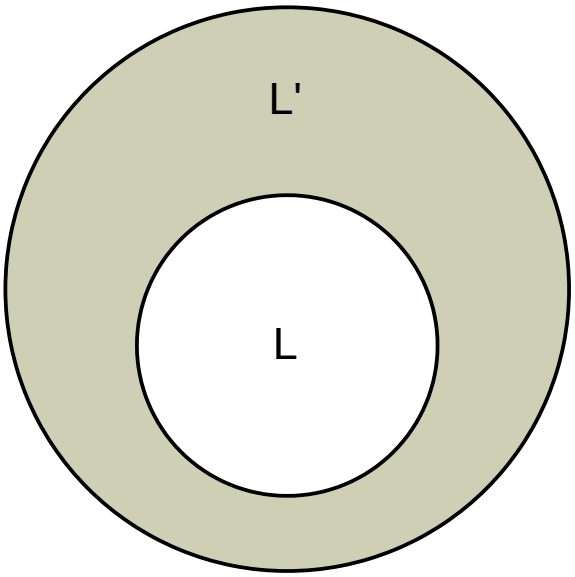

Figure 2.7: Illustration of the complement operation on language L with respect to alphabet Σ.

Figure 2.7 illustrates the complement operation on language L with respect to the alphabet Σ. The resulting language L' contains all the strings over Σ that are not present in L.

Kleene Star

The Kleene star operation takes a language L and produces a new language L^* consisting of all strings that can be formed by concatenating zero or more strings from L. The formal definition of the Kleene star operation is as follows:

$$L^* = \bigcup_{n=0}^{\infty} L^n \tag{2.9}$$

where $L^0 = \{\epsilon\}$, $L^1 = L$, and $L^n = L^{n-1} \cdot L$ for $n > 1$.

Figure 2.8 shows example of the Kleene star operation on language L. The resulting language L^* contains all strings that can be formed by concatenating zero or more strings from L.

$$\begin{aligned} L &= \{a, b\} \\ L^* &= \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, ...\} \end{aligned} \tag{2.10}$$

Figure 2.8: Example of the Kleene star operation on language L.

Chapter 3

Automata Algorithms and Applications

In the realm of computer science, algorithms form the heart of problem solving. They provide systematic approaches for tackling computational tasks, transforming abstract problems into concrete solutions. When it comes to automata, the application of well-crafted algorithms can reveal the full potential of automata, allowing us to modify, analyze, and compare them with efficiency.

Algorithms designed for automata are the unseen engines that drive some of the most crucial processes in computer systems. They enable us to realize complex functionalities, optimize resources, and construct effective solutions to a wide range of computational problems. The power of automata algorithms is derived from their ability to manipulate and adapt automata structures to meet specific needs, making them a versatile tool in a variety of fields.

This chapter will dive into the world of automata algorithms, focusing on their practical applications and the software libraries that support their implementation. We will explore various libraries, such as Mata, Spot, Greenery, AutomataLib, OpenFst, Grail, Pyformlang, Fado, and PySimpleAutomata, discussing their characteristics, advantages, and limitations.

We will also highlight common use cases for automata algorithms in real-world scenarios, including pattern matching, data compression, compiler and interpreter design, artificial intelligence, and formal verification. Within these domains, we will dive into specific algorithms and techniques, such as regular expression matching, Aho-Corasick algorithm, Huffman coding, Lempel-Ziv compression, lexical and syntax analysis, state-space search, reinforcement learning, model checking, and equivalence checking.

Finally, we will go over common automata algorithms used for intersection, union, negation, inclusion, finding reachable automata states, and detecting emptiness of automata.

3.1 Libraries for Working with Automata

Automata libraries form a crucial aspect of automata theory applications, serving as the backbone that enables the construction, modification, and analysis of automata. Numerous libraries catering to different programming languages have been developed over the years, providing a diverse range of functionalities to meet various needs.

However, it is important to note that not all libraries are created equal. Although some remain actively maintained and developed, offering a comprehensive set of features and

capabilities, many others have been left unmaintained or incomplete. This discrepancy can pose significant challenges when it comes to finding a library that is suitable for a particular application or study.

This section contains a list of libraries for working with automata, covering various programming languages such as Python, C++, Java, and others. This list aims to provide a comprehensive overview, highlighting the active development status, the language of implementation, and the key features of each library.

The goal is not only to provide a reference point for future studies, but also to enable a thorough comparison between the different libraries. This comparison can provide insight into their strengths and weaknesses, helping in the selection of the most appropriate library for a particular use case or research purpose.

In the following subsections, we briefly introduce each library, providing an overview of its key features, advantages, and limitations. Note that while the list is extensive, it is not exhaustive, and there are other libraries available that are not covered by this thesis.

Mata for C++ and Python

Mata [43] is an actively developed open-source library that provides an interface for various types of automata, including, but not limited to, non-deterministic finite automata (NFA) and alternating finite automata (AFA). It is available for Python and C++ developers. The library is in active development. The library stands out with its continuous development, trying to catch up to the latest advancements and needs in the field of automata theory. It provides a common ground for different types of automata, offering a wide range of functionality that promotes flexibility and efficiency in working with automata.

Spot for C++ and Python

Spot [39], shown in Figure 3.1 is a library primarily designed for C++17, but also offers bindings for Python. It specializes in the manipulation of Linear Temporal Logic (LTL) and ω-automata, as well as model checking. Being actively maintained and developed, it ensures its relevance and effectiveness in the rapidly evolving field of automata theory.

Figure 3.1: The official logo [40] of the Spot library.

Spot's strength lies in its focus on the LTL and ω-automata, providing robust support for manipulating these types of automata and conducting model checking. This makes it a valuable tool for projects that heavily rely on these specific areas of automata theory.

Automata for Python by caleb531

Automata [12], also known as `automata-lib`, is a Python library currently under active development. Provides support for the design and manipulation of finite automata, pushdown automata, and Turing Machines.

The primary strength of the Automata library lies in its broad coverage of automata types, including finite automata, pushdown automata, and Turing machines. The library presents a well-structured and user-friendly interface for creating and manipulating these automata, making it a powerful tool for educational and research purposes.

Greenery for Python

Greenery [33] is a Python library that offers a set of tools to parse and manipulate regular expressions. It is maintained actively, ensuring its usability and relevance in the field of automata and regular expressions.

The primary focus of Greenery is regular expressions, providing a solid foundation for parsing and manipulating them. Its unique focus and commitment to this area of automata theory make it an essential tool for projects that require in-depth work with regular expressions.

AutomataLib for Java

AutomataLib [27], is a library built for Java that focuses on automata, graphs, and transition systems modeling. It is under active development.

AutomataLib excels in enabling users to model various automata, graphs, and transition systems, providing a comprehensive platform for exploring these structures. Its ongoing development ensures its adaptability and relevance in the rapidly progressing field of computer science and automata theory.

OpenFst for C++

OpenFst [32], is a C++ library known for its functionalities related to the construction, combination, optimization, and searching of weighted finite-state transducers (FSTs). Despite being one of the older libraries, it is still under active maintenance, thus ensuring its functionality and reliability.

OpenFst's focus on weighted finite-state transducers (FSTs) makes it a powerful tool for tasks that involve the manipulation and optimization of these structures.

Grail for C++

Grail [14] (also [13]), is a C++ library that provides functionalities for symbolic computation. It is designed for the manipulation of objects in formal language theory, such as automata, languages, and regular expressions. Grail continues to be maintained, ensuring its readiness for contemporary applications.

Grail's primary strength is its extensive support for symbolic computation, making it a potent tool for projects that need to manipulate and analyze formal language theory objects. Its ongoing maintenance makes it a reliable choice for researchers and developers working in this domain.

Pyformlang for Python

Pyformlang [8] (also [38]), is a Python library designed to handle formal grammars. It includes abstractions for regular expressions, finite automata (deterministic, non-deterministic, with/without epsilon transitions), finite state transducers, context-free grammar, pushdown automata, indexed grammar, and recursive automata. Pyformlang is a still mostly maintained library.

The key advantage of Pyformlang is its broad coverage of formal grammars, providing a comprehensive toolkit for manipulating various types of automata and grammars. Its active maintenance and wide array of features make it a suitable choice for projects that require extensive manipulation of formal grammars.

Fado for Python

Fado [37] (also [36]) is a versatile and high-performance library dedicated to the symbolic manipulation of automata and other computational models. It is designed for Python and is in active maintenance.

The uniqueness of Fado comes from its performance in handling symbolic manipulation of various computation models, including automata. Its extensibility adds to its appeal, making it a good choice for automata manipulation in Python-based projects.

PySimpleAutomata for Python

PySimpleAutomata [31] is a Python library that provides robust functionalities for managing deterministic finite automata (DFA), non-deterministic finite automata (NFA) and alternative finite state automata in Word (AFW). Although it may not show significant activity, it is still maintained, making it a reliable option for automata manipulation.

PySimpleAutomata offers the advantage of being specialized in handling DFA, NFA, and AFW, making it a valuable resource for automata-related tasks that focus on these specific models. Despite its relatively quiet activity, the library is maintained and can be a reliable choice for projects that require automata manipulation in Python.

Automata for Python by bniemczyk

Automata by bniemczyk [10], is a Python library that provides abstractions for non-deterministic finite automata (NFA). It was developed as a tool to simplify and streamline the process of working with NFAs in Python.

Although it provides a handy interface for working with NFAs, it is important to note that it is no longer actively maintained. Despite its stagnant development status, it can still provide a valuable starting point for projects that deal with NFAs.

Nfa for Python

Nfa [35], is a pure Python library that facilitates the creation and manipulation of non-deterministic finite automata (NFA). It provides a set of functions and methods that simplify the process of building and operating on NFAs.

However, similar to Automata by bniemczyk, this library is mostly unmaintained. While this poses challenges for users seeking to find a library with ongoing support and development, Nfa's simplicity makes it an option worth considering for projects that need to construct and manipulate NFAs.

DFA for C++

DFA [24], is a library exclusively designed for C++. It is specifically designed to facilitate the definition and evaluation of deterministic finite automata (DFA) models. Unfortunately, it is currently unmaintained, but can still be useful for static projects or as a learning resource.

The primary use case of DFA is in projects that require a straightforward way to define and work with deterministic finite automata. Despite the lack of ongoing maintenance, it has a well-structured design that makes it easy to understand and use.

Automata for Haskell

Automata [6] (also [5]) is a comprehensive library for Haskell. Provides the ability to work with several different types of automata, including deterministic and non-deterministic finite-state automata (DFSA and NFSA), deterministic finite-state transducers (DFST), and non-deterministic finite-state transducers (NFST).

Automata for Haskell is a versatile tool with a wide scope, accommodating several types of automata. It is particularly useful for projects that require a diverse range of automata types, providing an array of functionalities to tackle complex problems. Furthermore, it includes various optimizations, such as a read-only run-length-encoded variant that aids performance when the described grammar includes long static runs of characters.

Automaton for Java

Automaton [9] is an abstraction library dedicated to automata theory in the Java programming language. Its principal aim is to provide a simplified interface for working with different types of automata. However, it should be noted that the development of this library is no longer active, which may imply outdated or missing features compared to other actively maintained libraries.

Automata for Rust

The Automata library for Rust [20] provides a programming framework for various types of automata and related algorithms. Offers good support for deterministic finite automata (DFA) and non-deterministic finite automata (NFA). However, similar to the Automaton library for Java, it is not currently under active development, which might limit its utility in contemporary applications.

Despite stagnation in the development of Automata for Rust, it is still a valuable resource for those working in the field of automata theory. The library offers tools and methods that aid in implementing automata algorithms. However, limitations point to the necessity of continuous development and maintenance in order to keep libraries relevant and beneficial to the evolving field of automata theory.

Thompson for Clojure

Thompson [1] is a simple library specifically designed for creating and manipulating finite-state automata (FSA). It is written in Clojure, a modern, functional and dynamic language on the Java platform. Thompson's simplicity makes it a suitable choice for beginners who are just starting out with automata theory or for projects that require lightweight and uncomplicated solutions.

However, it is important to note that Thompson is currently unmaintained. While the library might be still functional, the lack of active development means that it may not be compatible with newer versions of Clojure.

Package dk.brics.automaton for Java

The package dk.brics.automaton [11] is a Java library with comprehensive support for deterministic and non-deterministic finite automata (DFA and NFA). It provides robust capabilities for handling the Unicode alphabet (UTF16) and offers extensive support for regular expression operations, including concatenation, union, Kleene star, intersection, and complement.

Although the library is not currently under active development, it remains a powerful tool due to its comprehensive feature set and its implementation in Java, which is a widely used language in both academia and the industry.

It should be noted that despite its extensive capabilities, the dk.brics.automaton package may not be suitable for all projects, particularly those that require cutting-edge features or active developer support. Nevertheless, its wide range of functions and mature design make it a worthy consideration for projects involving finite automata and regular expressions.

3.2 Common Cases for using Automata Algorithms

Automata algorithms find diverse applications in several fields and are often indispensable for solving specific computational problems. This section explores such instances, where these algorithms play a vital role in enhancing the efficiency and capability of systems.

Understanding the functionality and implementation of automata algorithms in these domains provides a broader perspective on their significance and potential. It also helps identify the underlying principles that can be applied to devise novel algorithms for new applications.

Pattern Matching

Pattern matching is one of the fundamental applications of automata algorithms. It refers to the process of locating specific sequences, or „patterns“, within larger bodies of text or data. Automata-based pattern matching algorithms are used in many areas of computer science, including search engines, bioinformatics, text editors, and programming languages.

These algorithms utilize automata models to represent the desired patterns. Once the model is constructed, it is then traversed with the input data. Successful traversal to an accepting state of the automata indicates that the pattern is present in the data.

Two of the most commonly used pattern matching algorithms are Regular Expression Matching and the Aho-Corasick algorithm. Both algorithms are built upon the principles of finite automata.

This section was inspired by the book *Flexible pattern matching in strings* [28] and article *Efficient String Matching* [4].

Regular Expression Matching

Regular expressions, or regex, are a powerful tool for specifying patterns of text. They are often used in text processing tasks, such as searching for and replacing strings in a document.

The regular expression matching algorithm utilizes finite automata to represent the pattern described by the regular expression. Specifically, it uses an automata called deterministic finite automata (DFA).

The process begins by converting the regular expression into a non-deterministic finite automata (NFA) using Thompson's construction algorithm. More about this algorithm can be found in article *Programming Techniques: Regular Expression Search Algorithm* [41]. This NFA is then transformed into a DFA.

Once the DFA is constructed, the algorithm uses the input string to traverse the DFA. If the end of the string is reached while in an accepting state, the algorithm reports a match.

Aho-Corasick Algorithm

The Aho-Corasick algorithm is a string matching algorithm that is efficient in finding all occurrences of a set of patterns within an input text. It constructs a finite-state machine that resembles a trie[1] with additional „fail“ transitions.

Unlike the regular expression matching algorithm that matches a single pattern, the Aho-Corasick algorithm is capable of matching multiple patterns simultaneously. This makes it particularly useful in applications where multiple patterns need to be matched in a large body of text or data, such as in virus scanning and bioinformatics.

The Aho-Corasick algorithm first constructs an automata using the given patterns. This automata contains a „fail“ transition for each state, which determines the next state to move to when the current input character does not have an outgoing edge in the current state.

During the pattern matching phase, the algorithm traverses the automata using the input text, following the „fail“ transitions when necessary. Whenever it reaches an accepting state, it reports a match of the corresponding pattern.

Data Compression

Data compression is a fundamental aspect of computer science that involves reducing the volume of data used for storage or transmission. Compression techniques often leverage patterns in data to achieve these reductions. Automata algorithms can play a crucial role in implementing data compression techniques, with their ability to recognize and exploit these patterns. Two well-known algorithms used in data compression are Huffman coding and Lempel-Ziv compression.

This section was inspired by the book *Introduction to algorithms* [16] and article *A Technique for High-Performance Data Compression* [44].

Huffman Coding

Huffman coding is a well-known method for lossless data compression. The principle of Huffman coding involves the use of variable-length bit codes for different characters. Characters that occur more frequently are assigned shorter codes, whereas less frequent characters receive longer codes. The algorithm uses a priority queue to create a binary tree, where the most frequent characters are placed closer to the root of the tree.

Huffman coding can be represented as a finite automata, where the transitions correspond to the binary code associated with each character.

[1] A trie, also known as a prefix tree, is a type of search tree data structure.

Lempel-Ziv Compression

The Lempel-Ziv compression algorithms, also known as LZ77 and LZ78, are fundamental for lossless data compression. These algorithms are dictionary-based, meaning they work by creating a dictionary of phrases that have previously appeared in the text.

When a new phrase is encountered, it is added to the dictionary and a reference to the phrase (a pair of a position and length in LZ77, or a single index in LZ78) is placed in the output. This makes these methods particularly effective for compressing large files with repetitive sequences.

Automata can be used in the implementation of Lempel-Ziv algorithms to efficiently search the dictionary and find matches for the current phrase.

Compiler and Interpreter Design

Automata play a crucial role in the field of compiler and interpreter design, as they form the backbone of lexical and syntax analysis phases. Both lexical analyzers (also known as scanners) and syntax analyzers (also known as parsers) use different types of automata to fulfill their duty.

Following section was inspired by the book *Crafting interpreters* [30].

Lexical Analysis

Lexical analysis is the first phase of a source code compilation, in which the source code is converted into a series of tokens. Each token is a string of characters that constitutes a logically cohesive sequence, such as identifiers, keywords, and operators. Finite automata, specifically deterministic finite automata (DFAs), are used in the design of lexical analyzers.

The DFA is designed in such a way that each state represents a possible stage in the recognition of a token, and each transition represents the processing of an input character. The final states represent the completion of a token. A typical example would be a DFA designed to recognize identifiers in a programming language, where identifiers can be a sequence of letters and digits, starting with a letter.

Syntax Analysis

Following lexical analysis, the resulting stream of tokens is fed into the syntax analyzer or parser. This stage involves checking the tokens against the grammatical rules of the programming language to ensure their correct ordering, which results in a parse tree showing the syntactic structure of the input.

For this process, pushdown automata (PDA) are used. The PDA is capable of handling the nested structure of programming languages, which basic deterministic finite automata cannot handle. This is due to the stack data structure, which allows PDAs to store an arbitrary amount of information.

An example would be the handling of matching parentheses or matching begin-end symbols in a block-structured language. The PDA pushes each opening symbol onto the stack and pops it when a matching closing symbol is encountered.

Artificial Intelligence

Artificial intelligence (AI) is a field that attempts to create algorithms capable of intelligent behavior. It encompasses a broad range of subfields, from general-purpose areas such as ma-

chine learning and perception, to specific tasks such as game playing and theorem proving. Automata theory and automata algorithms find their applications in various aspects of AI, including state-space search and reinforcement learning. This section will dive into these two areas, explaining how they operate and the role automata play in their operations.

This section was inspired by the book *Automata-Driven Partial Order Reduction and Guided Search for LTL Model Checking* [22] and article *Learning Task Automata for Reinforcement Learning using Hidden Markov Models* [2].

State Space Search

State space search is a process used in AI to search a problem space, i.e., the collection of all possible configurations of a problem, in order to find a solution. The problem space can be represented as a graph or a state machine, where each node represents a state, and each edge represents a transition from one state to another. Automata can be used to represent the state space and the transitions between states.

Automata-based algorithms can be used to navigate this state space, searching for a path from the initial state to a goal state. Examples of these algorithms include Depth-First Search (DFS), Breadth-First Search (BFS), and A* search algorithm, all of which can be adapted to work with automata.

Reinforcement Learning

Reinforcement learning (RL) is a type of machine learning in which an agent learns to make decisions by interacting with its environment. The agent receives feedback in the form of rewards or punishments and aims to maximize the total reward over time. The process can be modeled as a Markov Decision Process (MDP), a mathematical framework that is used to describe an environment for RL.

Automata, particularly finite-state machines, can be used to represent the policy of an agent, which is a map from states to actions. The agent's policy can be updated and optimized over time using reinforcement learning algorithms like Q-learning or Policy Gradient methods.

Formal Verification

Formal verification represents one of the vital application areas for automata algorithms. As the complexity of software and hardware systems increases, ensuring their correctness becomes a challenge. Formal verification is a technique to prove or disprove the correctness of systems with respect to certain specifications or properties using formal methods of mathematics.

Automata algorithms play a fundamental role in formal verification by modeling the behavior of systems as state machines, which can be formally analyzed. Two main approaches are prevalent in formal verification, model checking and equivalence checking, both of which extensively use automata theory and algorithms.

This section was inspired by the explanation of formal verification in the article *Introduction to formal verification* [26].

Model Checking

Model checking is a form of formal verification where the system's model, often represented as an automata or a state-transition system, is exhaustively analyzed to check whether it meets a given specification. This specification is usually expressed in formal logic, such as temporal logic, which can describe properties about the system's behavior over time.

The process of model checking involves systematically exploring all states and transitions of the system model to check if the desired property holds. If the property does not hold, a counterexample is typically produced, illustrating a sequence of transitions that leads to an error.

Automata algorithms are crucial in model checking, especially for handling large or infinite state spaces. Techniques such as state space reduction, symbolic representation, and abstraction use automata algorithms to make the model checking process tractable.

Equivalence Checking

Equivalence checking is another important application of automata algorithms in formal verification. It involves comparing two systems (or two versions of a system) to determine if they are equivalent with respect to a certain criterion. In other words, it checks whether the two systems behave identically under all possible inputs.

Automata algorithms are central to this process. Systems are modeled as automata, and an equivalence algorithm is applied to these automata. If the algorithm determines that the automata are equivalent, it means that the systems have the same behavior. If not, the algorithm often provides a counterexample, showing an input sequence where the systems behave differently.

Automata-based equivalence checking is commonly used in hardware verification, compiler optimization validation, and protocol verification.

3.3 Common Automata Algorithms

Automata, in their many forms, serve as powerful computational models for a wide range of tasks within computer science. However, the utilization of these models requires the use of specialized algorithms that allow the manipulation and analysis of automata. There are many automata algorithms in the wild, they are vast and varied, each tailored to operate on a specific form of automata or designed to achieve a particular objective.

In the context of this thesis, we will focus on a select few algorithms, chosen for their relevance to this thesis. We will discuss algorithms for performing intersection, union, and negation operations on automata. We will also look into the inclusion algorithm using antichains. In addition, we will discuss algorithms for finding reachable states, a vital part of automata analysis, and detecting the emptiness of automata, an important problem in formal language theory.

Although these algorithms only represent a small fraction of all existing automata algorithms, they are foundational and widely used in the field. Understanding them will offer valuable insight into the broader field of automata theory and its applications. In the following sections, we will dive into the mechanics of these algorithms and offer a detailed exploration of their workings and applications. This section was inspired by the study materials [45], book [18] and paper [3].

Intersection Algorithm

The intersection of two automata, A_1 and A_2, produces a new automata A_{12} that accepts a word if and only if both A_1 and A_2 accept the word. This operation corresponds to the intersection operation on the languages accepted by the two automata.

The intersection algorithm for non-deterministic finite automata (NFA) is based on the construction of a new automata whose states correspond to pairs of states (q_{01}, q_{02}), where q_{01} is a state of A_1 and q_{02} is a state of A_2.

Let $A_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$ be two non-deterministic finite automata. The intersection automata $A_{12} = (Q_{12}, \Sigma, \delta_{12}, q_{012}, F_{12})$ is defined as follows:

- $Q_{12} = Q_1 \times Q_2$, i.e., the set of states of A_{12} is the Cartesian product of the states of A_1 and A_2.
- Σ is the input alphabet, common to both A_1 and A_2.
- $q_{012} = (q_{01}, q_{02})$ is the initial state of A_{12}.
- $F_{12} = F_1 \times F_2$, i.e., the set of final states of A_{12} is the Cartesian product of the final states of A_1 and A_2.
- The transition function δ_{12} is defined by $\delta_{12}((q_1, q_2), a) = \delta_1(q_1, a) \times \delta_2(q_2, a)$ for all $(q_1, q_2) \in Q_{12}$ and $a \in \Sigma$. This means that, given an input symbol a, the new automata A_{12} transitions from state (q_1, q_2) to a set of states that is the Cartesian product of the transition results of A_1 and A_2.

Union Algorithm

The Union Algorithm is a core method in automata theory, used to combine two automata into a single automata that accepts the union of the languages of the original automata.

Formally, given two NFAs $A_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$, we construct a new NFA $A_{12} = (Q_{12}, \Sigma, \delta_{12}, q_{012}, F_{12})$ that accepts the union of the languages accepted by A_1 and A_2.

The components of A_{12} are defined as follows:

- $Q_{12} = Q_1 \cup Q_2$, the set of states of A_{12} is the union of the states of A_1 and A_2.
- Σ is the input alphabet, common to both A_1 and A_2.
- $q_{012} = \{q_{01}, q_{02}\}$, the set of initial states of A_{12} is the union of the initial states of A_1 and A_2.
- $F_{12} = F_1 \cup F_2$, the set of accept states of A_{12} is the union of the accept states of A_1 and A_2.
- δ is defined as follows: for each state $q \in Q_{12}$ and each symbol $a \in \Sigma$,
  - If $q \in Q_1$, then $\delta(q, a) = \delta_1(q, a)$,
  - If $q \in Q_2$, then $\delta(q, a) = \delta_2(q, a)$.

The Union Algorithm thereby constructs a new NFA A_{12} that accepts any string accepted by A_1 or A_2. This is achieved by creating an NFA that simultaneously simulates A_1 and A_2 on each input string.

Thus, the Union Algorithm enables us to create an automata that captures the combined language of two distinct automata, a crucial operation in many applications of automata theory.

Negation Algorithm

The negation operation on an automata results in a new automata that accepts exactly the complement language of the original automata language. More formally, given an automata A that recognizes a language $L(A)$, the negation operation produces a new automata A' such that $L(A') = \Sigma^* - L(A)$, where Σ^* denotes the set of all possible strings over the alphabet Σ.

When dealing with non-deterministic finite automata (NFA), the concept of negation is not as straightforward as it is with deterministic finite automata (DFA). This is mainly due to the fact that an NFA can transition to multiple states for a given input, leading to potential ambiguity when attempting to define the complement.

A common approach to negating an NFA is to first convert it into a DFA and then apply the negation operation on the resulting DFA. Once the DFA is obtained, the negation operation is straightforward and involves swapping the accepting and non-accepting states.

Consider a DFA $M = (Q, \Sigma, \delta, q_0, F)$, where:

Negation of M, denoted as $\neg M$, is a new DFA $M' = (Q, \Sigma, \delta, q_0, F')$, where $F' = Q - F$.

This method of negating an NFA by converting it to a DFA and swapping the accepting and non-accepting states guarantees that the resulting automata will recognize the complement language of the original NFA's language.

Inclusion Algorithm using Antichains

Automata inclusion is a critical operation in automata theory. Given two automata A and B, the inclusion problem aims to decide whether the language recognized by A is a subset of the language recognized by B. In other words, we want to determine whether every word accepted by A is also accepted by B. While this problem is decidable for finite automata, it becomes undecidable for more complex types of automata, such as Turing machines.

One of the techniques that provides a practical solution for the automata inclusion problem is the use of antichains. Antichains are sets of incomparable elements under a given partial order. The idea behind the antichain inclusion algorithm is to compare states of the two automata not individually, but in groups. These groups, or antichains, are created based on a simulation relation, a pre-order relationship that captures the idea of one state simulating the behavior of another.

Given two non-deterministic finite automata (NFA) $A = (Q_A, \Sigma, \delta_A, q_{0A}, F_A)$ and NFA $B = (Q_B, \Sigma, \delta_B, q_{0B}, F_B)$. The antichain inclusion algorithm checks if $L(A) \subseteq L(B)$ by using a simulation relation $\preceq$ defined over $Q_A \times Q_B$. This relation is reflexive and transitive. It is used to group the states of A and B into antichains. The algorithm keeps track of pairs of antichains that have been visited before and only explores new ones.

The antichain inclusion algorithm provides an efficient and effective solution to the automata inclusion problem. Avoids the state explosion problem that arises in the traditional subset construction method for checking inclusion and is particularly useful in practical applications, such as model checking and formal verification.

Finding Reachable Automata States

In the realm of automata theory, the concept of *reachability* is of great importance. In the context of non-deterministic finite automata (NFA), reachability refers to the ability to transition from the start state to a particular state in the automata through a sequence of transitions. The set of reachable states of an automata can provide us with valuable insight into its structure and behavior.

Let $A = (Q, \Sigma, \delta, q_0, F)$ be a non-deterministic finite automata. The *reachability problem* for A asks for the set of states $Q' \subseteq Q$ that can be reached from the start state q_0 for some input string $w \in \Sigma^*$. Formally, a state $q \in Q$ is said to be *reachable* if there exists a string $w \in \Sigma^*$ such that $q \in \delta(q_0, w)$.

To compute the set of reachable states, we can use a simple depth-first search (DFS) or breadth-first search (BFS) starting from the start state q_0 and following the transitions defined by the function δ. The pseudocode for the algorithm using DFS is presented in Figure 3.2.

```
function reachable_states_recursive(S, A, q):
    add q to set S (mark q as visited)
    for each state p in A.delta(q, w) for some w in A.Sigma*:
        if p is not visited:
            reachable_states_recursive(S, A, p)

function reachable_states(A):
    create empty set of visited states S
    reachable_states_recursive(S, A, A.q0)
    return S
```

Figure 3.2: Pseudocode for the algorithm for finding reachable states of an automata.

This algorithm marks all states as unvisited at the beginning. Then the DFS starts from the start state. Whenever a state is visited, it is marked as such. The DFS procedure visits all states that can be reached from the current state by following the transitions. Once the DFS procedure has been applied to the start state, the set of visited states constitutes the set of reachable states of the automata.

Having the set of reachable states of an automata is crucial in many contexts. For example, in the process of automata minimization. States that are not reachable from the start state can be eliminated without affecting the language recognized by the automata.

Detecting Emptiness of Automata

Determining whether a given automata accepts at any word is a fundamental problem in automata theory. This is known as the *emptiness problem*. For a non-deterministic finite automata (NFA), the emptiness problem can be solved using a depth-first search (DFS) or a breadth-first search (BFS) from the initial state to find an accepting state.

An NFA $M = (Q, \Sigma, \delta, q_0, F)$ is empty if and only if there does not exist a word $w \in \Sigma^*$ such that M accepts w.

Mathematically, the above definition can be represented as:

$$\forall w \in \Sigma^*,\ \delta^*(q_0, w) \cap F = \emptyset$$

where $\delta^*(q, w)$ denotes the set of states that can be reached from state q after processing the word w.

The pseudocode for the emptiness check algorithm using DFS is presented in Figure 3.3.

```
function is_empty_recursive(S, M, q):
    if q is in M.F:
        return False
    add q to set S (mark q as visited)
    for each state p in A.delta(q, w) for some w in A.Sigma*:
        if p is not visited:
            if not is_empty_recursive(S, A, p):
                return False
    return True

function is_empty(M):
    create empty set of visited states S
    return is_empty_recursive(S, A, A.q0)
```

Figure 3.3: Pseudocode for the algorithm for detecting emptiness of an automata.

This algorithm works by starting from the initial state and traversing all reachable states using DFS. If it reaches an accepting state, it returns `False`, meaning that the NFA is not empty. If it has traversed all reachable states without finding an accepting state, it returns `True`, indicating that the NFA is empty.

Chapter 4

Proposed Design of Automata Visualization System

This chapter dives into the high-level design of the proposed Automata Visualization System. Design is focused on trying to effectively visualize the state and behavior of automata algorithms, contributing to a more comprehensive understanding of their functionality. The following text does not dive into the specifics of the reference implementation, but provides a broader outlook on the structure and functionality of the system.

The proposed Automata Visualization System is split into two separate parts, namely, the visualizer and the library integration. These two sub-systems communicate through a simple json stream-based communication interface that could potentially be standardized in the future. This design choice aims to provide flexibility, extensibility, and user-friendliness to the system.

The design acknowledges the diverse user environments in which the system could operate and aims to accommodate these differences. With a focus on usability, the design prioritizes flexibility, allowing for usage across a wide array of applications. Furthermore, to promote widespread adoption, the design aims to simplify integration with various libraries.

The system's architecture, including its sub-systems and their intercommunication, is discussed in detail. The functionality of the library integration sub-system and the visualization sub-system is explained, providing an overview of how they contribute to automata visualization.

Lastly, the chapter explores different use cases of the system. This includes visualizing the behavior of an algorithm and visual debugging of an algorithm. In all, this chapter provides a comprehensive insight into the proposed design of the Automata Visualization System, highlighting its potential to significantly contribute to automata algorithm understanding and development.

4.1 Expected User Environment for the System

The proposed system is designed with the expectation that its primary users will be developers or researchers working in the field of automata algorithms. The system can operate on any computer system that supports the dependencies of the chosen implementations of library integration and visualizer. Although the system was designed with *Linux* and *Windows* environments in mind, it is not inherently limited to these operating systems. The

system should also function effectively on other platforms, such as *MacOS* and *FreeBSD*, as long as the dependencies are supported.

As developers, users are expected to have a basic understanding of the underlying technologies and programming principles. This allows the system to be designed with a certain level of complexity, as users are expected to be capable of navigating and troubleshooting any potential issues that may arise. While the system is designed to be as user-friendly and intuitive as possible, it is not designed to be completely unbreakable. Instead, it prioritizes extensibility, adaptability, and the ability to facilitate efficient visualization of automata algorithms.

The resulting visualizations generated by the system are not limited to be used by developers alone. These visualizations can be used by non-developer users, such as teachers or students, to understand the workings of automata algorithms better. These users may not interact directly with the system to generate the visualizations; instead, they can use pre-generated visualization results, like images, animations, etc.

4.2 Goals of the System Design

Understanding the inner workings of an algorithm is often a requirement for developers, particularly, when the algorithm is complex, not well documented, or requires debugging. To gain insight, developers often rely on tools such as interactive debuggers such as *GDB*, memory analysis tools such as *Valgrind*, or even straightforward *print* statements. While these tools provide valuable information about variable states, they may not offer a comprehensive view of the algorithm's behavior in a broader context.

One way to address this issue is through the visualization of the algorithm's execution, which can offer a more intuitive understanding of its inner workings. This is particularly relevant in the context of automata algorithms, where the state and transitions of the automata being manipulated by the algorithm can be visualized for better understanding.

There are different ways to implement such a visualization system. One approach is to offer a tool that allows the user to have full control over the visualization process, specifying details such as layout, node and edge relations, and annotations. This is the approach taken by libraries like *GraphViz*, which offers great flexibility but might be more demanding for users seeking a quick overview of the algorithm's operation.

This thesis proposes a different approach: a visualization system that requires minimal effort from the user while offering a clear and intuitive view of the algorithm's behavior. This system aims to provide easy integration with existing algorithms and automate the visualization process as much as possible.

To achieve this, the proposed system design follows the principles of *print* debugging. Developers can insert a simple statement into their code, similar to a *print* statement, and the system takes care of visualizing the data at that point in the algorithm execution. This design has several specific goals, which are detailed in the following subsections.

Flexibility for Versatile Usability

The visualization system aims to be flexible enough to accommodate a wide range of use cases. This includes different types of automata algorithms, various programming languages, and diverse user requirements. The system should be easy to integrate into existing code and should not impose constraints on the design or implementation of the algorithm.

Simplifying Integrations with Libraries for Better Extensibility

The system should facilitate integration with existing libraries for working with automata. This includes providing a clear and straightforward interface for sending data to the visualizer and offering guidelines for developers, who wish to add support for new libraries. The goal is to make the system extensible and encourage the development of additional integrations.

User-Friendliness for an Effortless Experience

The visualization system should be easy to use for developers at all levels of expertise. This means providing a clear and intuitive interface for inserting visualization commands into the code and offering meaningful visualizations that help users understand the algorithm's operation. The system should require minimal configuration and should automate the visualization process as much as possible.

4.3 System Architecture

The architecture of the proposed system is designed to be modular and adaptable to support a wide variety of needs and use cases. It is divided into two primary components: the library integration sub-system and the visualization sub-system (see Figure 4.1).

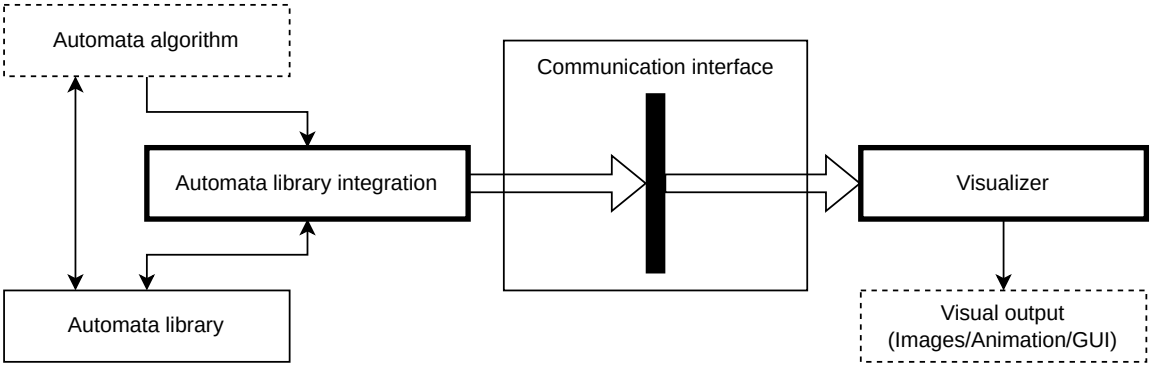

Figure 4.1: Overview of the proposed system's architecture, showing the two primary system components and their interaction.

This architectural design is inspired by the principles of *UNIX*, separating each part of the system into its own self-contained component, ensuring that each component of the system focuses on a specific function. The library integration sub-system focuses on data extraction, while the visualization sub-system focuses on rendering and visual presentation. The standardizable interface facilitates communication between these two components, ensuring the operation of the overall system.

Library Integration Sub-system

The library integration sub-system serves as the interface between the existing automata library and the proposed system. It is responsible for collecting data from the automata library and transforming it into a standardized format that can be understood and processed by the visualization sub-system. This design enables the integration of the proposed system

with virtually any library for working with automata, broadening its applicability and usability.

Visualization Sub-system

The visualization sub-system is tasked with taking the standardized data provided by the library integration sub-system and generating the corresponding visualizations. It is responsible for all visualization-related operations, including layout management, rendering, and generation of the final visual output. This approach centralizes all visualization logic within a single component, simplifying the overall design and facilitating future enhancements or modifications to the visualization functionality.

Communication Interface

To enable communication between the library integration sub-system and the visualization sub-system, a standardizable interface is employed. This interface represents the standardized format into which the library integration sub-system transforms the data from the automata library. Through this interface, the visualization sub-system can accept and process the transformed data to generate the required visualizations. More information about the communication interface can be found in the chapter *Proposed Design of Standardizable Interface* 5.

4.4 Use Cases

The primary motivation behind the design and development of the proposed automata visualization system is to make it easier for developers to understand automata algorithms. By visually representing the behavior of these algorithms, it can help with understanding and debugging such algorithms. With this in mind, it is possible to identify two primary use cases for the system as can be seed in Figure 4.2.

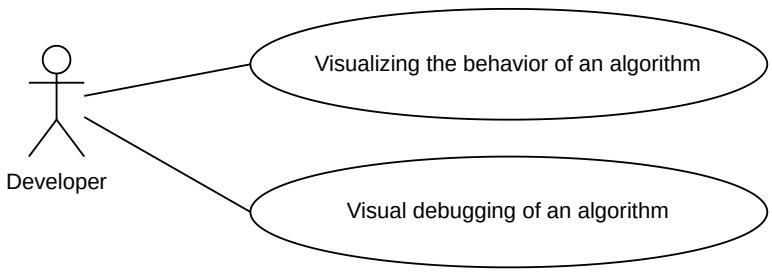

Figure 4.2: Use case diagram of the Automata Visualization System

Visualizing the behavior of an algorithm

This use case covers scenario where a user seeks to enhance their understanding of an existing algorithm by visualizing its behavior. In this context, the user utilizes the print-like interface provided by the library integration to generate the necessary visualization data. This data can then be processed to produce a visual representation of the algorithm's behavior over time, enabling the user to gain insights into the intricate workings of the algorithm. This could be particularly useful in educational settings or when sharing and presenting

algorithmic concepts with others, as visualizations can often communicate complex ideas more intuitively than textual descriptions or pseudocode.

Visual debugging of an algorithm

The second use case revolves around the debugging process during algorithm development. In this scenario, the user aims to identify, understand, and fix issues in the algorithm execution by visually inspecting its state transitions. As the user steps through the algorithm, the print-like interface's output is redirected to the chosen visualizer. This visualizer then displays the current state of the algorithm, including the state of the automata it is working with. This additional visual information, alongside traditional debugging tools displaying variable values, can provide more context and insight, aiding the user in resolving issues more efficiently.

Chapter 5

Proposed Design of Standardizable Interface

The purpose of this chapter is to outline the concept and design of a standardizable interface for the automata visualization system. This interface serves as a bridge between the two main sub-systems: the automata library integration and the visualization component. It is responsible for facilitating a seamless transfer of data to be visualized, thereby acting as an integral part of the visualization process.

The use of a standardizable interface brings several benefits. It decouples the two subsystems, allowing each to be developed, maintained, and updated independently. This design supports greater extensibility, as different library integration modules or visualization modules can be developed and integrated without affecting the rest of the system. Furthermore, it provides a clear and consistent means of representing data, simplifying the process of integrating new libraries, or developing new visualization techniques.

Throughout this chapter, we shall dive into the various aspects of the interface design, beginning with its objectives and basic use cases that the system is designed to address. Subsequently, we will examine the internal workings of the system in more detail, providing a comprehensive understanding of the system architecture and design.

5.1 Goals of the Interface

The primary purpose of the interface design lies in facilitating a seamless transmission of the algorithm state represented by annotated automata, generated during the algorithm's execution, to the visualization component. Such an interface should satisfy several key criteria to ensure effectiveness and flexibility:

- **Simplicity**: The data should be structured in a format that is straightforward to comprehend, manipulate, and visualize. This includes maintaining a consistent organization of data elements, using well-defined and intuitive data types, and providing clear documentation and guidelines for interface usage.
- **Flexibility**: The interface should be designed with extensibility and adaptability in mind. It should be able to accommodate various types of automata and different visualization needs. This means providing mechanisms for handling diverse data requirements.

- **Standardization**: The interface should be built around widely-accepted standards to ensure broad compatibility and interoperability. By aligning with recognized standards, the interface can better integrate with existing tools and systems, facilitate data exchange and collaboration, and enable future enhancements and extensions.

In essence, the design of the interface aims to strike a balance between simplicity and flexibility. It seeks to offer a simple, easy-to-use standard for developers to log the state of automata for visualization, while also being adaptable enough to support a broad spectrum of automata algorithms and visualization requirements. This vision guides the subsequent sections detailing the design and implementation of the proposed interface.

5.2 Data Models and Structures

The data transmitted via the interface are encapsulated in a form of a JSON object, where each object represents a particular step of an algorithm. As we can see in Figure 5.1, this JSON object consists of three main elements: „topic“, „graphs“ and „texts“.

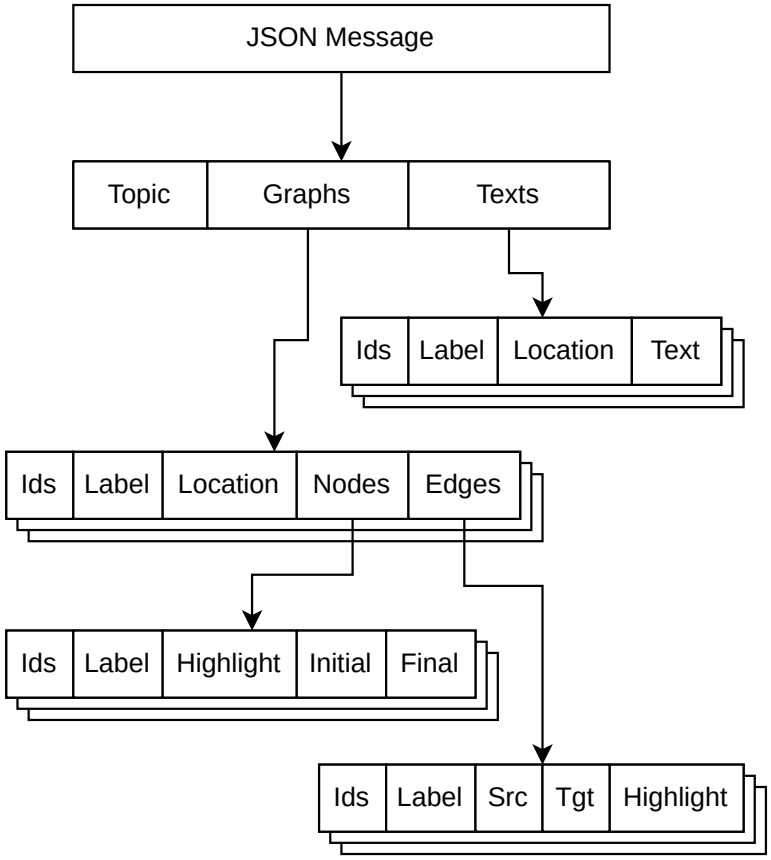

Figure 5.1: Structure of a JSON message

The „topic“ serves as an identifier for the stream to be updated. It provides a way to differentiate between different algorithmic threads, enabling the simultaneous visualization of multiple automata algorithms or different automata being processed alongside each other, enabling the visualization of more complex automata algorithms.

The „graphs“ array holds a set of graph objects that embody the state of the automata at a particular algorithm step. Each graph object consists of:

- `ids`: An array holding identifiers for the automata.
- `label`: A label specifying the name of the automata.
- `location`: An optional annotation for specifying the graph's spatial positioning.
- `nodes`: An array of all graph nodes. Each node object can have attributes such as „ids“, „label“, „highlight“, „initial“, „final“ and more to represent the state, name, color, and type (initial or final) of the node.
- `edges`: An array of all graph edges. Each edge object can have attributes like „ids“, „label“, „src“, „tgt“, „highlight“ and others to specify the edge's state, name, source, target nodes and color.

The „texts“ array comprises of various text objects to be displayed alongside the graphs. Each text object consists of:

- `ids`: An array of identifiers for the text.
- `label`: A label specifying the name of the text.
- `location`: An annotation for specifying the text's spatial positioning.
- `text`: The actual text content to be displayed.

Through the design of this structure, the interface provides a flexible and efficient way to log and visualize the state of automata algorithms. It facilitates the dynamic representation of algorithmic steps, adapting to changes in automata states, and enhancing the user's understanding of the algorithm's behavior.

5.3 Interface Model

The proposed interface serves as a bridge for the flow of data between the algorithm using automata library integration and the visualizer, ensuring an standardizable medium for the transmission of information. At its core, the interface can be represented as a pipe, where one end receives a stream of visualization frames from the algorithm, and the other end feeds this data to the visualizer that, in turn, generates graphical representations for the user.

The design of this interface leans towards versatility, enabling it to accommodate a variety of visualization tools, including potential web-based visualizers and other novel visualization techniques that may emerge in the future. This ensures that the interface remains robust and adaptable to evolving requirements in the field of algorithm visualization.

This design approach to the interface provides a robust foundation for the visualization system, enabling seamless integration and communication between the algorithm through automata library integration and the visualizer.

5.4 Transmission Options

The interface is designed as a pipe, which can be utilized in various ways for transferring visualization data. This versatile approach allows for a broad spectrum of options for data transmission, depending on the user's needs and context. The following subsections discuss several potential methods of utilizing this interface for transmitting data.

Direct Transmission

In a scenario, where an automata algorithm is *„logging“* its state to its standard output using the automata library integration, one straightforward method of transferring visualization data is by directly connecting the standard output of the automata algorithm to the standard input of the visualizer. This creates a pipeline of data from the algorithm directly into the visualizer, allowing for immediate visualization. This is especially useful for debugging purposes where developer can directly see an updated visualization with each step the algorithm *prints* out.

File-Based Transmission

Another common scenario is to save the log data to a file, which can be processed by the visualizer at a later time. This approach provides flexibility, allowing users to generate visualizations when needed, without having to run the algorithm simultaneously. This method is particularly useful when debugging or analyzing the behavior of an algorithm over time. It is useful for inspecting and modifying the logged data, which is ideal for developing a middleware for modifying the data before they are passed to the visualizer.

Integrated Visualization

With some additional integration, it is possible to redirect the output of the log data back to another library within same application instead of outputting it to the standard output. This library can then visualize the data as part of the application running the algorithm. This technique is used to provide support for Jupyter notebooks, enabling the interactive visualization of automata algorithms within the notebook environment.

Network-Based Transmission

The interface also supports network-based transmission of the data stream, which can be achieved using various protocols. For example, the log data can be sent to a WebSocket server, which can forward the data to a web page for visualization. Alternatively, an HTTP polling method can be utilized, wherein the web page polls the server for new messages and visualizes them. This approach can be useful in scenarios where WebSocket connections are not feasible. This enables the creation of web-based visualizers, expanding the potential environments where the visualization can be consumed.

Compression and Encryption

The interface is compatible with compression and encryption mechanisms. Since the data structure of the interface does not inherently optimize for space usage (it is stream of JSON objects), employing stream compression when transmitting log data over the internet can be advantageous. However, given the generally manageable size of logs for typical automata,

compression is not mandatory. Likewise, stream encryption can be used to secure the transmission of visualization data, depending on the confidentiality requirements of the use case.

Chapter 6

Reference Implementation of Visualizer and Library Integration

In this thesis, a reference implementation of the proposed system as detailed in Chapter 4 has been carried out. The purpose of this chapter is to dive into the specifics of the implementation process and to discuss the decisions made during the development of the reference implementation.

To begin with, a survey of currently available solutions was conducted. In this chapter, we discuss the range of these options and the choices that were ultimately made.

Subsequently, we explore the development of the visualization sub-system reference implementation using GraphViz as a visualization backend.

Finally, the integration with the Mata library is discussed. This library integration plays an important role in producing data that feeds into the visualization sub-system. The chapter provides an insight into how this integration has been designed and developed.

6.1 Existing Solutions and Selection Process

As part of the process of creating the reference implementation of this system, we needed to evaluate and select solutions for two primary concerns: a library for visualizing graphs, which forms the basis for the visualizer sub-system, and an automata library to serve as the target for library integration. These decisions were critical in guiding the implementation of the system.

There is also an important choice of programming language for the visualizer sub-system. While the programming language of an automata library integration is determined by programming language used by automata library, the programming language of visualizer had to be selected from wide range of available options.

Graph Visualization Libraries

To visualize the state of automata algorithms, we needed to choose a graph visualization library. After conducting an evaluation of several popular libraries, we decided on GraphViz [19]. It met our needs perfectly because of several key features. GraphViz uses a language, namely the DOT language, to describe the graph. This language is both machine and human-readable, allowing users to manipulate the graph representation as needed. Moreover, GraphViz offers significant versatility in that it allows a wide range of param-

eters to be tweaked for visualization purposes, thus providing the user with more control over the visualization output.

Other options such as NetworkX for Python [29], G6 for JavaScript [7] or D3 for JavaScript [17], and ReGraph for Python [25] were also considered. However, they didn't quite match requirements in the same way that GraphViz did. For instance, while NetworkX and ReGraph offer extensive functionality, their integration would have required considerably more development effort and may have introduced unnecessary complexity to the system. The JavaScript libraries G6 and D3, while powerful, would have necessitated additional considerations such as browser compatibility and client-server communication, detracting from primary goal of automata visualization.

Automata Libraries

For the library integration component of the system, our task was to select a suitable automata library. While the scope of the thesis required the integration to be designed for the Mata library, a review of other automata libraries was still performed to better understand the landscape and challenges of library integration.

Among the libraries looked into were Spot, AutomataLib for Python, OpenFst, and Grail. Each of these libraries offers a unique set of features, capabilities, and API designs. However, given the requirements of the assignment the integration of the Mata library was implemented.

Programming Language for the Visualizer

In order to implement the GraphViz visualizer, an appropriate programming language must be chosen. The main task of the GraphViz visualizer is to generate a description of the graph in the DOT language. Hence, there are no strict language requirements.

Python was chosen as the language for developing the visualizer for several reasons. Python is a high-level language known for its simplicity and readability, making it suitable for development tasks where readability and maintainability of the codebase are of high importance.

Moreover, the author of this thesis is familiar with Python, which significantly simplifies the development process. In addition, Python has a rich ecosystem of libraries and frameworks that can be leveraged to further speed up the development process and improve the functionality of the application.

One such library is Graphviz, a Python interface to GraphViz that can create, read, write, and draw graphs using Python using GraphViz DOT language in process. By using this library, the process of generating DOT code is greatly simplified, thus reducing the complexity of the visualizer implementation.

6.2 GraphViz Visualizer

The GraphViz visualizer forms an integral part of the reference implementation. Its main task is to convert the data received through the standardizable interface into the DOT language format for the purpose of graphically representing automata. The visualizer supports both Command Line Interface (CLI) and Python library interfaces, each with their unique advantages and use-cases.

In this chapter, we will also dive into a key aspect of our visualization approach: maintaining consistent state and edge positioning throughout different steps of the algorithm's operation. This approach significantly improves readability of resulting visualizations.

CLI Interface

The CLI tool is designed to be simple and easy to use. It reads input data, either from a file or standard input, and generates DOT code representing the corresponding automata. This code is either written to standard output, or it can be processed per frame by a user-specified command. The latter is done by adding a special flag, with the DOT code being passed via standard input, and the topic and index information provided in environment variables to the user-specified command.

The CLI tool also provides two modes for reading the input data: a stream mode and a normal mode. In the stream mode, the visualizer regenerates all previous visualizations after each message to ensure that the layout of the new step matches the previous ones. This approach ensures that the positions of nodes and edges remain consistent across all visualizations, enhancing readability and user understanding. On the other hand, in the normal mode, the visualizer reads the entire input first and then generates all visualizations in one go.

One of the great advantages of the CLI tool is that it can be combined with a few other shell commands to accommodate a wide variety of use-cases, making it a versatile tool for generating and manipulating automata visualizations. For example usage see Figure 6.1.

```
./my-script-which-logs-automata-states-to-standard-output | \
    automataviz \
        --no-refresh-with-stream \
        --exec 'dot -Tpng | display' 2>/dev/null
```

Figure 6.1: Example usage of the CLI interface.

Python Library Interface

The Python library interface provides an alternative way to generate automata visualizations. It allows for direct integration of the visualizer into the code of any Python application, thereby providing more control over the visualization process.

This interface is built using Pydantic models. Developers can instantiate a Graph model from JSON data, a dictionary, or programmatically fill it. Afterward, it's possible to create a DOT code representation of that model for visualization.

This approach enables developers to tightly integrate the automata visualization process into their application or algorithm. It offers the flexibility to customize and manipulate the visualization process based on the specific requirements of the application or algorithm, thus enabling the creation of more accurate and insightful visualizations. For example usage see Figure 6.2.

Consistent State and Edge Positioning

One of the significant features of the GraphViz visualizer is the ability to maintain the consistent positioning of states and edges across different steps of the visualization. To

```
import automataviz.model as model

view = model.ViewData(topic="my-topic")

# For each logged frame
# Create view for the frame
frame = model.View(...)
view.add_frame(frame)

dot = view.compile()
```

Figure 6.2: Example usage of the library interface.

ensure this consistency, the visualizer uses an approach where it renders all nodes and edges in each image, regardless of their active status in the current step of the algorithm.

However, those nodes and edges which are not part of the current step are marked as hidden. This technique of pre-rendering ensures that the GraphViz layout algorithm arranges the nodes and edges consistently across the different visualization steps, leading to a more coherent and easily trackable representation of the automata evolution.

This consistent positioning greatly improves the readability of the visualizations. It helps user with following the changes of states and transitions in the automata and results in a more understandable way. By visually maintaining the structural layout of the automata, users can easily focus on the changes and behaviors of the algorithm without getting lost in the shifting positions of states and edges.

6.3 Mata Library Integration

The reference implementation for the proposed visualization system also contains the integration for the Mata library. Mata is a robust automata library that supports both Python and C++ interfaces. This flexibility in interfaces offers the opportunity to tailor the design of the integration to best suit the features and conveniences offered by each respective language.

In both the Python and C++ interfaces, the integration module provides helper functions to enable 'print'-like logging of the automata state at any point during the execution of an algorithm. This makes it easy to generate visualization data for algorithms operating on automata.

Python Interface

The Python integration is implemented as a Python module that can be imported in the same way as the Mata library itself. It provides a simple interface to generate visualization data that can be used in conjunction with the GraphViz-based visualizer or any other visualizer implementing the standardizable interface discussed in previous chapter.

An example of usage is shown in the figure 6.3 below.

The `printviz` function provided by the module takes an instance of an automata as its primary argument. It also supports additional parameters to modify the automata visualization, such as marking certain states or transitions as highlighted, changing labels,

```
import libmata as mata
from mataviz import printviz

# Create an automata
nfa = mata.Nfa()

# Add some states and transitions
# ...

# Visualize the automata
printviz(nfa)
```

Figure 6.3: Example usage of the Python interface for Mata library integration.

and so on. These features allow users to tailor the visualization to their specific needs and improve the comprehensibility of the automata behavior.

C++ Interface

The C++ integration for the Mata library operates in a similar manner to the Python interface, but it is designed to leverage the particular advantages of C++ for better performance and ease of use. The interface is defined in a header file that can be included in any C++ program utilizing the Mata library.

An example of usage is shown in the figure 6.4 below.

```
#include <mata/nfa.hh>
#include "matacviz.h"

// Create an automata
Nfa nfa;

// Add some states and transitions
// ...

// Visualize the automata
printviz(nfa);
```

Figure 6.4: Example usage of the C++ interface for Mata library integration.

The `printviz` function in the C++ interface accepts a reference to an instance of an automata. Similar to the Python interface, additional arguments can be provided to adjust the appearance of the visualization. This allows for fine-tuned control over how the automata behavior is visualized.

6.4 Jupyter Notebook Integration

Jupyter Notebook [23], being an open-source web application that allows creation and sharing of documents containing live code, equations, visualizations and narrative text, serves as an excellent platform for the visualization of automata algorithms. To facilitate the integration of the automata visualization system in Jupyter Notebooks, a separate Python library, referred to as `matavizbook`, was developed.

The `matavizbook` library is specifically designed to handle the output of the `libmata` library integration. It utilizes the Python interface of the GraphViz visualizer, alongside `ipywidgets` [21], a collection of interactive widgets for the Jupyter notebook. With these components, the library is able to present interactive visualizations of automata. Users can view different states of the algorithm execution using a slider, offering a intuitive interface to follow the timeline of the algorithm.

```
import libmata as mata
from mataviz import printviz
from matavizbook import setup_output_to_notebook, process_and_display

# Set up the output to Jupyter Notebook
setup_output_to_notebook()

# Create an automata
nfa = mata.Nfa()

# Add some states and transitions
# Or do some algorithmic magic
# ...

# Visualize the automata
printviz(nfa)

# display all visualizations
process_and_display()
```

Figure 6.5: Example usage of the Jupyter Notebook integration.

The integration of the system within a Jupyter Notebook is relatively straightforward, as illustrated in the Python code example in Figure 6.5. In the example, the `setup_output_to_notebook` function from the `matavizbook` library configures the output handler of the `mataviz` library to output data to the Jupyter Notebook. The `printviz` function, a part of the `mataviz` library, is used to generate visualizations of the automata at different stages of the algorithm execution. These visualizations are logged and are ready to be displayed within the notebook.

Finally, the `process_and_display` function from the `matavizbook` library is called. This function processes all logged visualizations and displays them interactively in the Jupyter Notebook. The result is an interactive frame with slider that the user can manipulate to visualize different stages of the algorithm execution, effectively creating an execution timeline of the algorithm.

Chapter 7

Possible Future Extensions

While this thesis has covered the foundation and initial design of the automata visualization system, it is by no means exhaustive of its full potential. The system has been designed with extensibility and adaptability as its central goals, opening up numerous possibilities for enhancement and customization.

The advantage of such an approach is the ability to implement an extensive range of use-cases without significant alterations to the core system. These could include more specific integrations with existing tools, rendering improvements, real-time visualization enhancements, or even new features that leverage the base functionality in unforeseen ways. The ultimate aim is to help with the understanding and development of automata algorithms for a broad spectrum of users, ranging from seasoned developers to students of computer science.

While many of these extensions would be valuable additions to the system, their development fell outside of the scope of this thesis. However, these ideas demonstrate the flexibility and potential of the system to evolve and adapt to new requirements and challenges. Some of these extensions may be relatively easy to implement, offering a promising avenue for future contributors to the project.

In this chapter, we discuss several possible future extensions that could further enhance the capabilities of the automata visualization system. These include both technical improvements to the underlying system and more user-centric enhancements, aimed at improving the overall experience of using the system for automata visualization.

7.1 Generic Jupyter Notebook Integration

While the current Jupyter notebook integration, `matavizbook`, is specifically designed for `mataviz`, our reference implementation of the `mata` library integration for Python, there is potential for a more generalized approach. This would involve creating a generic Python module capable of working with any Python library integration for automata visualization.

The advantage of this more generic approach is that it would significantly broaden the applicability of the Jupyter notebook integration, making it useful for a wider array of library integrations. Given the prevalence of Jupyter notebooks in data analysis, research, and teaching, this could substantially enhance the reach of this automata visualization as it would simplify implementation of Python library integrations for other automata libraries.

This generic Python module would sit between the Jupyter notebook and any Python library integration, serving as a middle-man of sorts. This module would need to handle the

task of taking the output of the library integration and ensuring that it can be displayed by the Jupyter notebook, regardless of the specific library integration being used. To accomplish this, a standardized baseline for outputting visualization data would need to be defined. This would allow any Python library integration to use the same interface for outputting visualization data, allowing the Jupyter Notebook integration to capture this data and show the visualization withing the notebook.

7.2 Rendering Improvements

The primary focus of this thesis was the system's design, providing an extendable platform for automata visualization. Therefore, not much emphasis was given to the aesthetic appeal of the output generated and substantial scope for refining the visual aspect of the system was left for future.

The visual representation of automata, when executed well, greatly aids in understanding the complex concepts of automata algorithms. A well-rendered image is not only more appealing but can also clarify the underlying principles and operation more efficiently. Hence, refining the aesthetics of the visualization output is an important step in future work.

In initial implementation, there are several aspects in need of improvement. For instance, the current handling of node sizes in GraphViz is not optimal. Nodes are sometimes represented too large in relation to the font size used for labels, resulting in a visualization that seems disproportional.

Moreover, the size of nodes increases if the labels get longer, but the edges do not scale accordingly. This discrepancy leads to edges appearing too thin in comparison with larger nodes, potentially causing a visual imbalance.

It is worth mentioning that these are not inherent limitations of the system. In fact, all of the issues above can be addressed within the scope of the existing graph rendering system. To improve the aesthetics, GraphViz visualizer needs enhancements, which can be considered for future extensions.

Possible solutions could include:

- Utilizing line wrapping or truncation for overly long names to maintain a consistent and readable node size.
- Incorporating interactive features that allow users to hover over an element to view its full name or additional details.
- Exploring different fonts, colors, and shapes for nodes and edges to create a better visual output.

These improvements are certainly within reach given the existing capabilities of the system. However, they have not been prioritized in this initial implementation due to the focus on system design. Further research and development can help make these enhancements a reality, resulting in a more aesthetically pleasing visualization tool for automata algorithms.

7.3 Integrations for Other Automata Libraries

The system designed in this thesis was purposefully made adaptable and extensible to support the implementations of integration for other automata libraries. The primary

objective of such integrations would be to equip a wide range of automata libraries with the capability to generate visualizations, thereby contributing to the understanding and analysis of algorithms using these libraries.

The only requirement for any implementation of library integration is the ability to output JSON objects (messages), encapsulating all the necessary information to visualize the state of the automata as the algorithm manipulates it. Beyond this requirement, the specifics of the library integrations are largely at the discretion of the developer.

One recommended approach for implementing the library integration is to implement a `print`-like function, which would basically „print“ out the data for visualization. This function would essentially serve as the bridge between the automata library and the visualization tool, translating the automata state information into a format that can be consumed and visualized.

However, it is important to note that this approach is not strictly required. There is variability among different programming languages. Each language has unique features and constructs, so the design and implementation of the library integration may differ depending on the language in question.

Another important advantage related to the design of this system is that it limits how much pre-processing the library integrations must do. Most of the data processing is handled by the visualization sub-system. This means that the library integration primarily focuses on generating the raw data for visualization, simplifying its role and allowing for simpler implementation.

The implementation of integration for other automata libraries would significantly enhance the versatility of the visualization tool. Different libraries may offer unique features, or they may be preferred for specific types of problems or domains. By integrating with a wide array of libraries, it will allow for automata algorithms written using these libraries can be visualized.

7.4 Visualizer as Single Page Web Application

The concept of utilizing a Single Page Web Application (SPA) as an automata visualization tool introduces an user-friendly approach to presenting and analyzing automata algorithms. Using the capabilities of modern web technologies, this extension could help user experience by providing an intuitive interface for automata visualization.

One way to design this extension could follow a server-client architecture. The server side would handle the conversion of JSON objects, received via the standardizable interface, into DOT code that graphically represents the automata to be visualized. This DOT code would then be transmitted to the client-side web application through a WebSocket connection.

The web application, operating within the user's browser, would be responsible for receiving the DOT code and rendering the corresponding graphical representation. By leveraging the inherent interactivity of web interfaces, users could interact with the visualization in real time, providing an better understanding and ability to analyse the algorithm. This may prove highly beneficial in understanding complex automata algorithms.

This approach may prove to be particularly user-friendly, especially for those accustomed to interacting with applications within a web browser. However, it is noteworthy that a similar user experience could be achieved locally using Python and GUI frameworks like Tkinter [42] or Qt [34] for rendering the visualizations.

Moreover, there is also possibility of developing the entire visualizer in JavaScript, including the processing of data received via the standardizable interface. Such approach would eliminate the inherent requirement for a server-side component, and data for visualization could be delivered through various means, including file uploads to the web page.

7.5 Improvements for Real-time Visualization

The current design of the visualizer allows for the potential of real-time visualization of algorithm behavior, such as during the debugging process. While this approach can be advantageous in providing immediate feedback, it also presents several areas of potential improvement.

Enhanced Consistency of State and Edge Positioning

Currently, the consistency of state and edge positioning does not attempt to maintain the positions of states when a new step is introduced. The visualizer regenerates previous visualizations to match the layout when a new step is introduced. However, this approach does not guarantee that the layout will remain unchanged when adding a new step.

To address this, it would be possible to implement a method to maintain the relative positions of states while moving them to accommodate the inclusion of new states. The current renderer lacks this capability, which could be beneficial for real-time visualization, where consistency and coherence across steps are crucial for effective understanding.

User Interface for Real-time Visualization

A user interface for real-time viewing of the visualization updates could enhance the current debugging process. Currently, due to the necessity of regenerating visualizations when adding new steps, traversing through the generated visualizations can be cumbersome without a proper user interface.

Such a user interface could help with browsing through the visualized steps more conveniently, allowing users to better understand the process and changes between steps. The interface could be implemented as a standalone application or as a web application, both of which would provide an interactive platform for the user to manipulate and view the generated visualizations. This could not only benefit developers but also provide an engaging educational tool for students studying automata theory and algorithms.

7.6 Visualizers for Different Types of Automata

As it stands, the reference implementation of the visualizer and Mata library integration currently supports deterministic and non-deterministic finite automata. Although this serves the needs of many users, there are numerous types of automata which are not currently supported, and these present opportunities for future extension.

An immediate avenue of exploration is supporting other closely related types of automata. With some careful design and implementation, we can create the similar `print` functions to handle various automata types, extending the versatility of the current implementation.

For instance, automata such as pushdown automata (PDA), used in parsing context-free languages, and linear bounded automata (LBA), used in parsing context-sensitive languages, could potentially be incorporated. Support for these types of automata would increase the system's overall functionality.

Another intriguing direction is the support for Turing machines. However, the nature of Turing machines, with their unbounded memory and read-write head, imposes a greater challenge in terms of visualization. The standardizable interface would need to be extended to handle the building blocks of Turing machines, including the infinite tape and the read-write head. Designing an effective and intuitive visualization for Turing machines would require careful consideration of how to represent these elements and their dynamic interactions.

Beyond automata, the visualization system could potentially extend to other computational structures, such as logic trees. Extending the system to visualize logic trees could help with understanding and debugging algorithms which work with logic trees.

Further, in a broader sense, any algorithm working with graph-based data structure could potentially benefit from the visualization capabilities offered by the system. However, this would entail assessing the standardizable interface's capability to meet all the necessary functionality and possibly tailoring the interface and visualizer to better suit these structures.

7.7 Debugger Integration into IDE

In the age of modern Integrated Development Environments (IDEs), developers often rely on integrated debugging tools. The Debugger Integration into IDE is one of the possible future enhancements of the automata visualization system that could be considered. The main idea of this extension would be to enable developers to visualize the state of the automata algorithm directly from the debugger of their IDE.

When debugging a program with the automata visualization system integrated, an additional window or tab in the IDE could display the visualization of the last logged step of the algorithm, updating with each logged step in the debugging process. This functionality would be akin to a live automata viewer, showing the changing state of the automata as the developer steps through the code.

In addition to this, the IDE could provide further interactivity features to enhance the debugging process. For instance, when a variable holding a state is selected in the debugger, the corresponding node in the visualization could be highlighted, providing an immediate visual link between the code and the automata. The same functionality could be applied for edges. By clicking on a variable representing an edge in the debugger, the corresponding edge in the visualization would be highlighted. This would allow developers to have a much clearer understanding of the relationship between the code and the automata.

Unfortunately integrating this visualization system into various IDEs would require a considerable amount of effort as each IDE may have different requirements and support different extension mechanisms. Furthermore, to maintain such an extension, it would be necessary to keep up with changes in the IDE's API and overall functionality. Nonetheless, if implemented, this feature could significantly enhance the debugging process of automata algorithms, and thus holds great potential for future development.

7.8 Tool for Generating Animations

Another prospective extension of the proposed system includes the development of a tool to facilitate the generating animations from the visualization of automata algorithm execution. This functionality could be particularly beneficial for educators and researchers who wish to demonstrate the step-by-step execution of automata algorithms.

Tool would generate animations from a series of steps logged by the algorithm execution. Each frame of the animation would represent a particular step in the execution, thereby showcasing the evolution of the automata state over time.

Chapter 8

Examples of the Automata Algorithm Visualization

During the development of the Automata Visualization System, a variety of common automata algorithms and an advanced algorithm, namely the Inclusion Algorithm using Antichains, were tested. This chapter presents some examples of the visualizations produced for these algorithms.

8.1 Intersection Algorithm

The Intersection Algorithm combines two automata to form a new automata that recognizes the intersection of the languages recognized by the original automata. This can be visualized by showing each step of the algorithm's execution, with each frame showing the actions or changes performed on the automata.

Figure 8.1 depict an example of this visualization. Highlighting is used to indicate corresponding nodes and edges in the source automata and in the resulting automata. This helps to clarify the correlation between elements in the source automata and their counterparts in the output.

8.2 Union Algorithm

The Union Algorithm forms an automata that accepts the union of the languages recognized by two input automata. Union operation for non-deterministic finite automata (NFAs) is essentially about creating a copy of the two automata into one.

Visualizing this process allows us to monitor the algorithm's progress, particularly how nodes and edges from the original automata are copied and adapted in the resulting automata. Highlighting and label mapping are used to show the correspondence of new automata states to the original states. For visual example, see Figure 8.2.

8.3 Negation Algorithm

The Negation Algorithm creates an automata that recognizes the complement of the language recognized by the input automata. For complete deterministic finite automata (DFA), negation is primarily about copying the automata, followed by inverting the final states in the result.

As this is a straightforward process, the visualization mainly focuses on highlighting changes as the automata is being copied and highlighting the new final states. For visual example, see Figure 8.3.

8.4 Inclusion Algorithm using Antichains

The Inclusion Algorithm checks if the language of one automata is a subset of the language of another automata. This implementation applies a technique known as „antichains“ to reduce the amount of necessary checks.

The visualization of this algorithm is more intricate, as it involves creating a virtual automata for visualization, where nodes represent visited state groups. Removed subsumed states are shown as custom nodes, and custom edges indicate which subsum removed them. This technique offers a more explicit depiction of the algorithm's operations as can be seen in Figure 8.4.

8.5 Finding Reachable Automata States

When visualizing the algorithm for finding reachable states in an automata, highlighting can be used to differentiate between nodes that are currently being processed and nodes that are in the stack, awaiting processing. This helps to clarify the current state of the algorithm and the nodes that are being processed. For visual example, see Figure 8.5.

8.6 Detecting Emptiness of Automata

The algorithm for detecting emptiness of an automata can be visualized in a similar manner to the algorithm for finding reachable states. Highlighting can be used to indicate nodes that are currently being processed and nodes that are in the stack, awaiting processing. When the algorithm finishes, the path to the final state can be visualized by highlighting the nodes that were used to reach the final state, as can be seen shown in Figure 8.6.

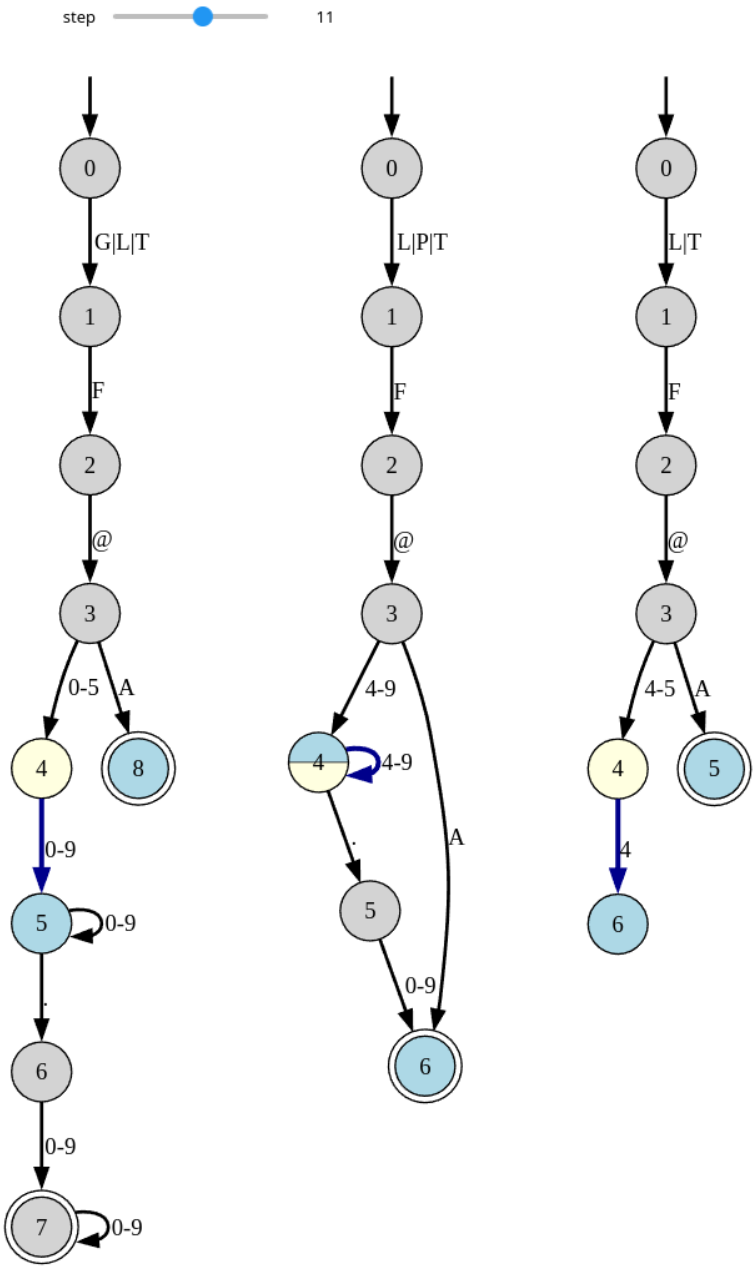

Figure 8.1: Intersection algorithm visualization; Yellow nodes represent currently processed states, blue nodes represent states in the stack, and blue edges represent currently processed transition

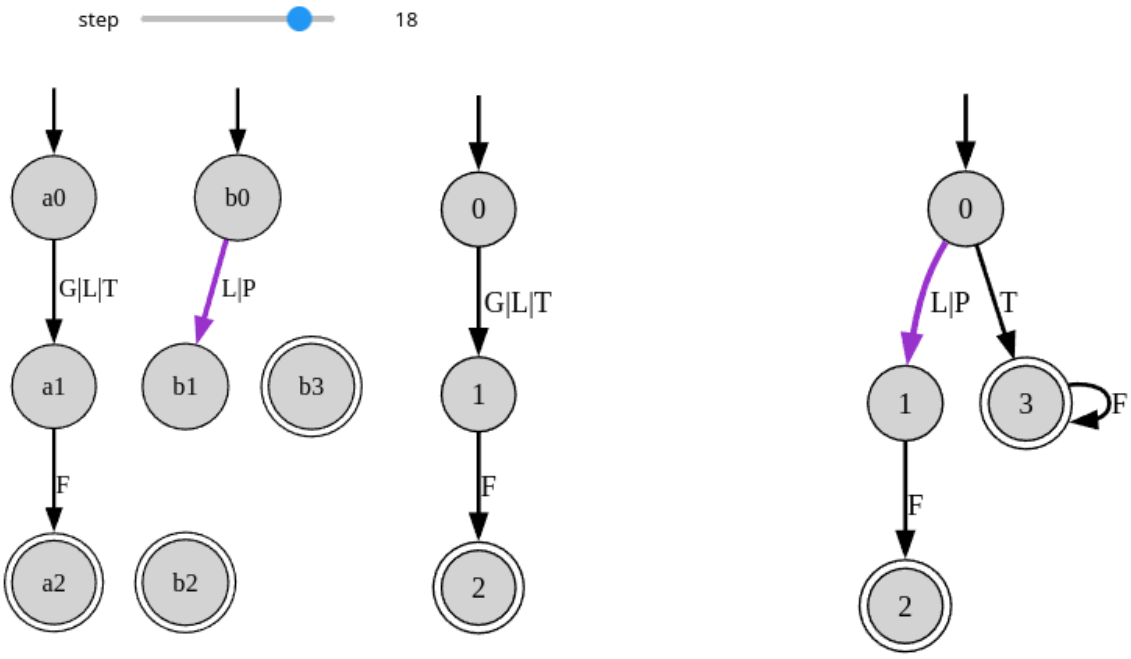

Figure 8.2: Union algorithm visualization; Purple edge represent currently processed transition

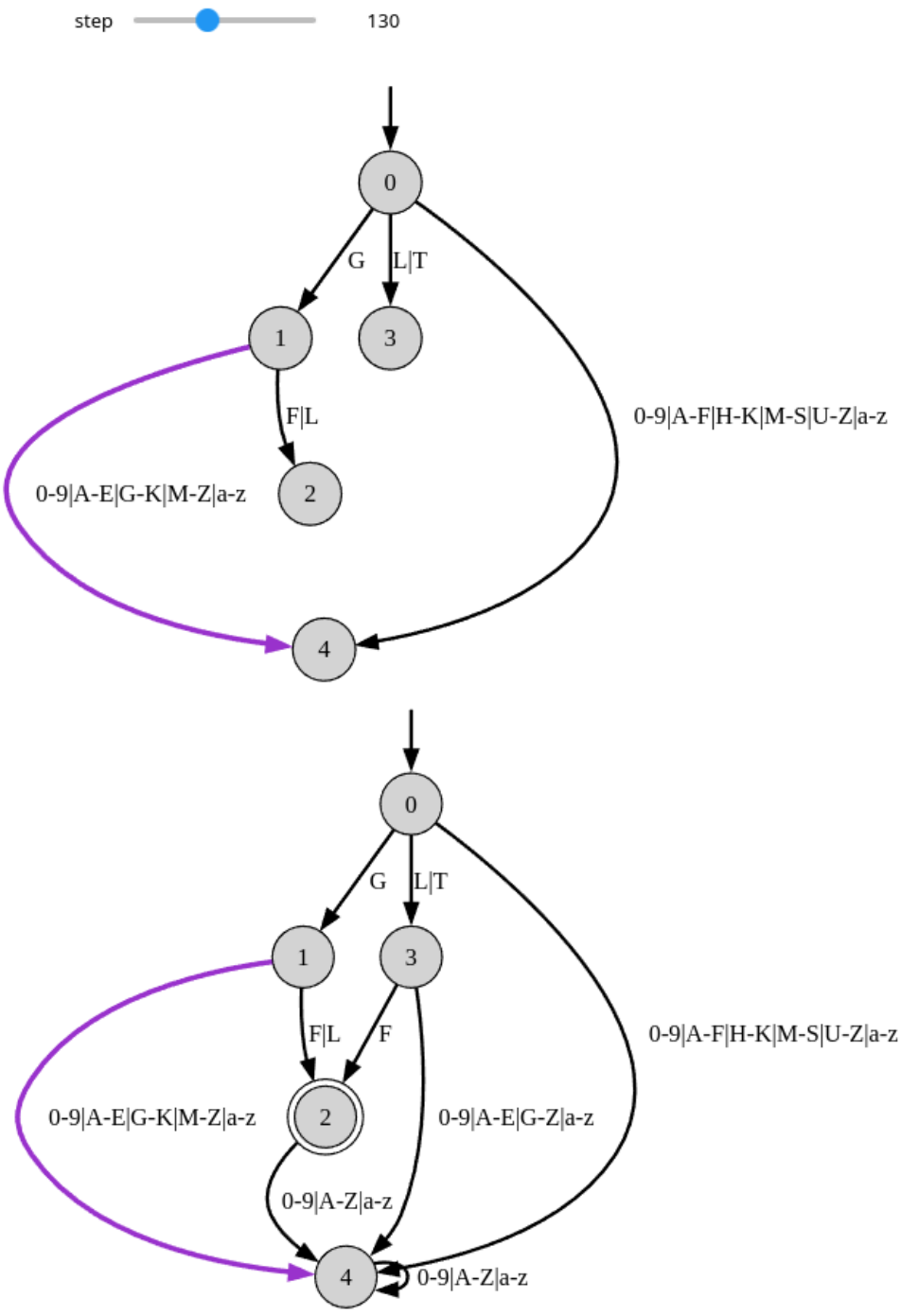

Figure 8.3: Negation algorithm visualization; Purple edge represent currently processed transition

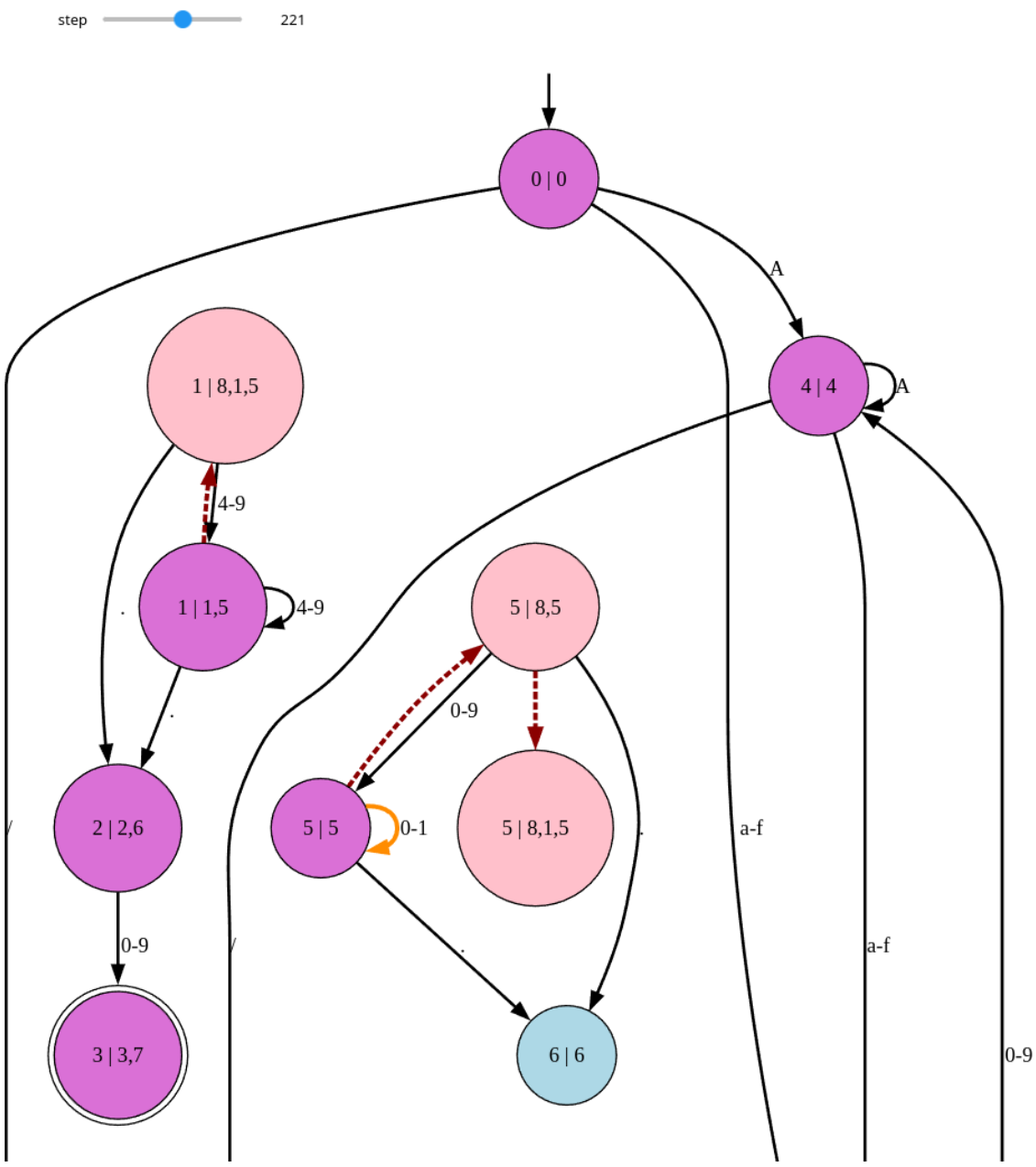

Figure 8.4: Inclusion algorithm using antichains visualization; Purple nodes represent already processed state groups, blue nodes represent state groups in the stack, red nodes represent subsumed groups, and orange edges represent currently processed transition

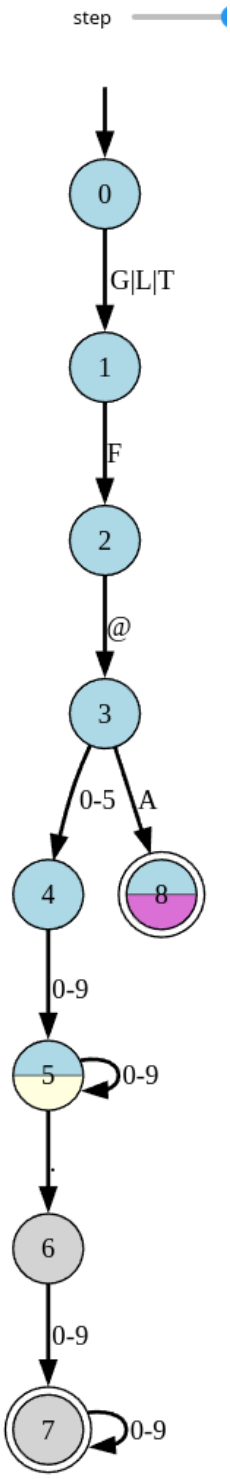

Figure 8.5: Reachable states algorithm visualization; Purple nodes represent currently processed states, yellow nodes represent states in the stack, and blue nodes represent nodes which the algorithm knows are reachable

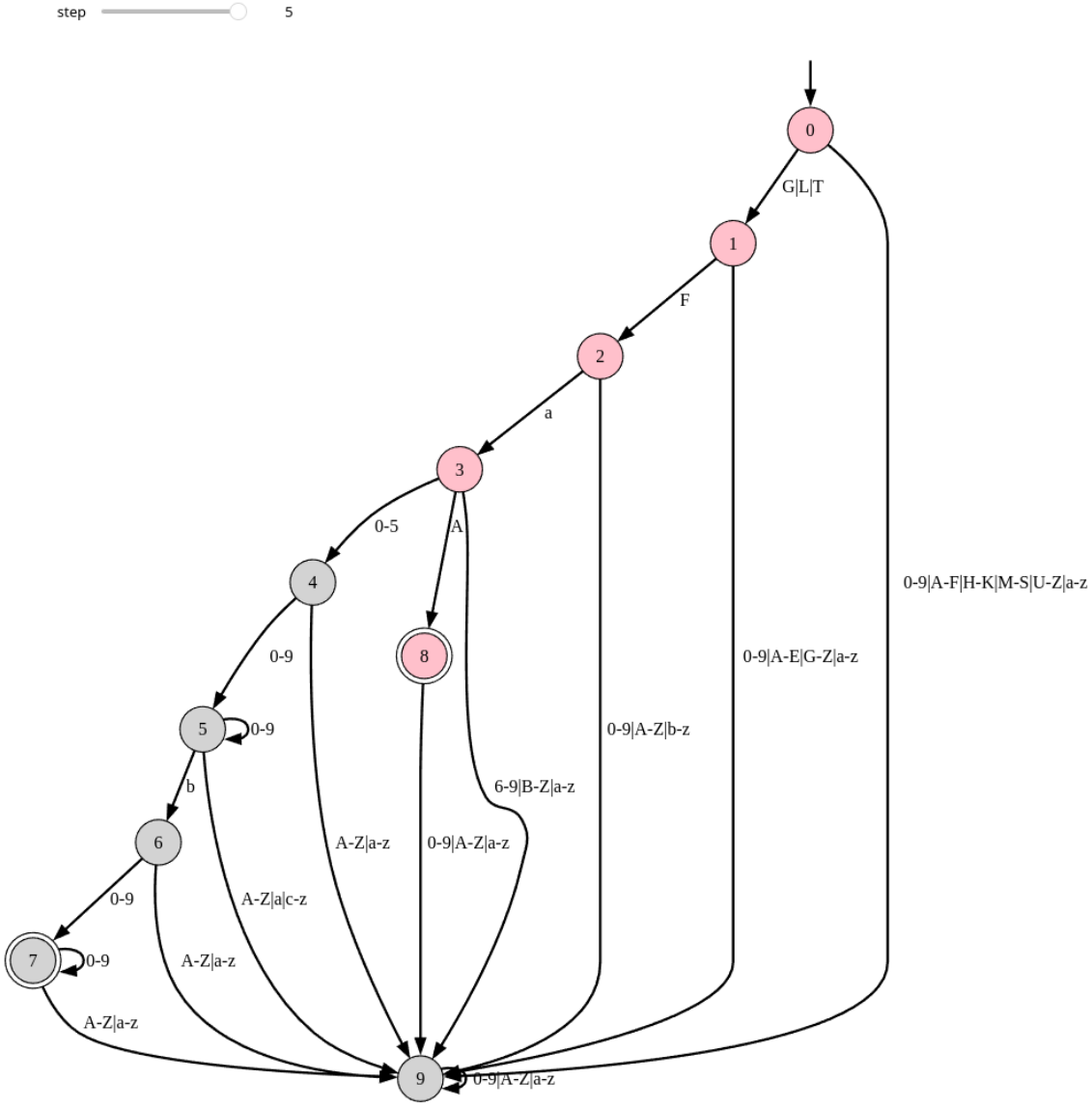

Figure 8.6: Emptiness algorithm visualization; Path to the final state is highlighted in red.

Chapter 9

Conclusion

This thesis has presented an approach for visualizing the state of automata algorithms during their execution, with a focus on improving comprehension and debugging. The proposed system includes a standardizable interface for logging the state of automata algorithms and a reference implementation of the `GraphViz`-based visualizer integrated with the `mata` library.

The visualization system has been designed to be extensible and adaptable, enabling users to create custom library integrations or develop their own visualizers. This flexibility fosters innovation in the field of automata visualization and encourages the development of new techniques and tools. The thesis has also explored the potential benefits of automata algorithm visualization for researchers, developers, teachers, and educators, highlighting its potential to improve understanding, communication, and collaboration in the study and development of automata algorithms.

An essential aspect of this thesis is the test of the proposed automata visualization system on a variety of common and interesting automata algorithms. In Chapter 8, the system is tested on a series of algorithms, including Intersection, Union, Negation, Inclusion using Antichains, Finding Reachable Automata States, and Detecting Emptiness of Automata. By applying the visualization system to these diverse algorithms, it has been demonstrated that the proposed system can effectively facilitate understanding and debugging for a wide range of automata-related tasks. The successful visualization of these algorithms showcases the system's versatility and adaptability, as well as its potential to provide valuable insights into the inner workings of complex automata algorithms.

In conclusion, the proposed visualization system represents a significant advancement in the field of automata algorithm visualization, offering a more intuitive and effective means of understanding and analyzing these complex computational processes. By promoting better comprehension, debugging, optimization, and collaboration, this system has the potential to significantly improve the study and development of automata algorithms and contribute to the education of future generations of computer scientists. Future work in this area could involve the refinement of the proposed system, the development of additional library integrations and visualizers, and the application of these tools to novel automata algorithms and problems.

Bibliography

[1] Aadah. *AADAH/Thompson: Library for working with deterministic and non-deterministic finite state automata.* [cit. 2023-05-08]. Available at: https://github.com/aadah/thompson.

[2] Abate, A., Almulla, Y., Fox, J., Hyland, D. and Wooldridge, M. *Learning Task Automata for Reinforcement Learning using Hidden Markov Models.* 2022.

[3] Abdulla, P. A., Chen, Y.-F., Holík, L., Mayr, R. and Vojnar, T. When Simulation Meets Antichains. In: Esparza, J. and Majumdar, R., ed. *Tools and Algorithms for the Construction and Analysis of Systems.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. ISBN 978-3-642-12002-2.

[4] Aho, A. V. and Corasick, M. J. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM.* New York, NY, USA: Association for Computing Machinery. jun 1975, vol. 18, no. 6. DOI: 10.1145/360825.360855. ISSN 0001-0782. Available at: https://doi.org/10.1145/360825.360855.

[5] Andrewthad. *Andrewthad/Automata: DFA and NFA for Haskell.* [cit. 2023-05-08]. Available at: https://github.com/andrewthad/automata.

[6] Andrewthad. *Automata for Haskell.* [cit. 2023-05-08]. Available at: https://hackage.haskell.org/package/automata.

[7] Antvis. *Antvis/G6: Readme.* [cit. 2023-05-08]. Available at: https://github.com/antvis/G6/blob/master/README.en-US.md.

[8] Aunsiels. *Aunsiels/pyformlang: A python library to manipulate formal languages.* [cit. 2023-05-08]. Available at: https://github.com/Aunsiels/pyformlang.

[9] Bebui. *Bebui/automaton: A java automaton library to create and manipulate deterministic and non-deterministic finite automaton with integer used as symbols.* [cit. 2023-05-08]. Available at: https://github.com/bebui/Automaton.

[10] Bniemczyk. *BNIEMCZYK/Automata: Finite Automata for python.* [cit. 2023-05-08]. Available at: https://github.com/bniemczyk/automata.

[11] *Package dk.brics.automaton for Java.* [cit. 2023-05-08]. Available at: https://www.brics.dk/automaton/.

[12] caleb531. *Caleb531/Automata: A python library for simulating finite automata, pushdown automata, and Turing machines.* [cit. 2023-05-08]. Available at: https://github.com/caleb531/automata.

[13] Campeanu, C. *Grail: Hosted at CSIT at UPEI.* Computers Science Department at the University of Western Ontario, London, Ontario, Canada [cit. 2023-05-08]. Available at: http://www.csit.upei.ca/theory/.

[14] Campeanu, C. *Grail how to.* [cit. 2023-05-08]. Available at: http://www.csit.upei.ca/~ccampeanu/Grail/.

[15] Chomsky, N. Three models for the description of language. *IEEE Transactions on Information Theory.* 1956, vol. 2, no. 3. DOI: 10.1109/tit.1956.1056813.

[16] Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C. *Introduction to algorithms.* MIT Press, 2001.

[17] d3. *D3/D3: Bring data to life with SVG, canvas and HTML.* [cit. 2023-05-08]. Available at: https://github.com/d3/d3.

[18] ESPARZA, J. *Automata theory: An algorithmic approach.* 1st ed. MIT PRESS, 2017. ISBN 9780262048637.

[19] *Graphviz.* [cit. 2023-05-08]. Available at: https://graphviz.org/.

[20] HeroicKatora. *Heroickatora/Automata: A Rust Library to model several different Automata Algorithms.* [cit. 2023-05-08]. Available at: https://github.com/HeroicKatora/automata.

[21] *Ipywidgets.* [cit. 2023-05-08]. Available at: https://pypi.org/project/ipywidgets/.

[22] Jensen, P. G., Srba, J., Ulrik, N. J. and Virenfeldt, S. M. Automata-Driven Partial Order Reduction and Guided Search for LTL Model Checking. In: Finkbeiner, B. and Wies, T., ed. *Verification, Model Checking, and Abstract Interpretation.* Cham: Springer International Publishing, 2022.

[23] *Jupyter Project Documentation.* [cit. 2023-05-08]. Available at: https://docs.jupyter.org/en/latest/.

[24] Kaecoutinho. *Kaecoutinho/DFA: Deterministic finite automaton C++ Library.* [cit. 2023-05-08]. Available at: https://github.com/kaecoutinho/DFA.

[25] Kappa Dev. *Kappa-dev/regraph: Tool for building graph-based hierarchical knowledge representation systems.* Available at: https://github.com/Kappa-Dev/ReGraph.

[26] Kukimoto, Y. *Introduction to formal verification.* [cit. 2023-05-08]. Available at: https://ptolemy.berkeley.edu/projects/embedded/research/vis/doc/VisUser/vis_user/node4.html.

[27] LearnLib. *LearnLib/automatalib: A free, open-source java library for modeling automata, graphs, and Transition Systems.* Dortmund University of Technology, Germany [cit. 2023-05-08]. Available at: https://github.com/LearnLib/automatalib.

[28] Navarro, G. and Raffinot, M. *Flexible pattern matching in strings: Practical on-line search algorithms for texts and biological sequences.* Cambridge University Press, 2007.

[29] *NetworkX documentation.* [cit. 2023-05-08]. Available at: https://networkx.org/.

[30] Nystrom, R. *Crafting interpreters*. Genever Benning, 2021. ISBN 978-0990582939.

[31] Oneiroe. *Oneiroe/PySimpleAutomata: Academic python library to manage DFA, NFA and AFW automata*. [cit. 2023-05-08]. Available at: https://github.com/Oneiroe/PySimpleAutomata.

[32] *OpenFst: Open-source library for weighted finite-state transducers*. [cit. 2023-05-08]. Available at: https://www.openfst.org/twiki/bin/view/FST/WebHome.

[33] Qntm. *QNTM/Greenery: Regular expression manipulation library*. [cit. 2023-05-08]. Available at: https://github.com/qntm/greenery.

[34] *Qt: Tools for each stage of software development lifecycle*. [cit. 2023-05-08]. Available at: https://www.qt.io/.

[35] Reity. *Reity/NFA: Pure-Python Library for building and working with nondeterministic finite automata (nfas)*. [cit. 2023-05-08]. Available at: https://github.com/reity/nfa.

[36] rogeriореis. *Fado: A library of tools to manipulate formal languages*. [cit. 2023-05-08]. Available at: https://pypi.org/project/fado/.

[37] rogeriореis. *Fado: Home page*. [cit. 2023-05-08]. Available at: https://fado.dcc.fc.up.pt/.

[38] Romero, J. Pyformlang: An Educational Library for Formal Language Manipulation. In: *SIGCSE*. 2021. DOI: https://doi.org/10.1145/3408877.3432464.

[39] *Spot: A platform for LTL and omega-automata manipulation*. [cit. 2023-05-08]. Available at: https://spot.lre.epita.fr/.

[40] *Spot: A platform for LTL and omega-automata manipulation (Logo)*. [cit. 2023-05-08]. Available at: https://spot.lre.epita.fr/spot2-2023042011.svg.

[41] Thompson, K. Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM*. New York, NY, USA: Association for Computing Machinery. jun 1968, vol. 11, no. 6. DOI: 10.1145/363347.363387. ISSN 0001-0782. Available at: https://doi.org/10.1145/363347.363387.

[42] *Tkinter: Python interface to TCL/TK*. [cit. 2023-05-08]. Available at: https://docs.python.org/3/library/tkinter.html.

[43] VeriFIT. *VeriFIT/Mata: A new Automata Library*. Brno University of Technology [cit. 2023-05-08]. Available at: https://github.com/VeriFIT/mata.

[44] Welch. A Technique for High-Performance Data Compression. *Computer*. 1984, vol. 17, no. 6. DOI: 10.1109/MC.1984.1659158.

[45] Češka, M., Vojnar, T., Smrčka, A. and Rogalewicz, A. *Teoretická informatika*. Aug 2020.

Appendix A

Contents of the Included Storage Media

- **automataviz**

  Source code of the AutomataViz GraphViz visualizer tool, which is used for visualization of automata.

- **examples**

  Examples automata algorithms in Jupyter notebooks. These examples use this Automata Visualization System to visualize their behavior.

- **interface**

  Documentation for the Standardizable Interface used in communication between library integration and visualizer.

- **matacviz**

  Source code of the `Mata` library integration for the C++ programming language.

- **mataviz**

  Source code of the `Mata` library integration for the Python programming language.

- **matavizbook**

  Source code of the Jupyter notebook integration for the `mataviz` library.

- **xkuchy02-text**

  Source code of this document in LaTeX.

- **README.md**

  Short manual for the Automata Visualization System.

- **xkuchy02-print.pdf**

  Text of the master thesis (print version).

- **xkuchy02-wis.pdf**

  Text of the master thesis (online version).