AutomataViz

https://github.com/adhamafis/AutomataViz

Technical Report

May 18, 2025

Abstract

This report presents an in-depth analysis of AutomataViz, a Java application that transforms regular expressions into minimal Deterministic Finite Automata (DFA) and visualizes them as interactive graphs. The report covers the application's architecture, key components, algorithms, and implementation details, highlighting the theoretical foundations underlying the automata conversion pipeline.

Contents

1 Introduction

AutomataViz is a tool designed for the visualization and manipulation of finite automata. It provides a complete pipeline for converting regular expressions to minimal DFAs through a series of well-established algorithms in automata theory. This report examines the project's architecture, implementation details, and the theoretical underpinnings of its core algorithms.

The project is open-source and available on GitHub at: https://github.com/adhamafis/AutomataViz

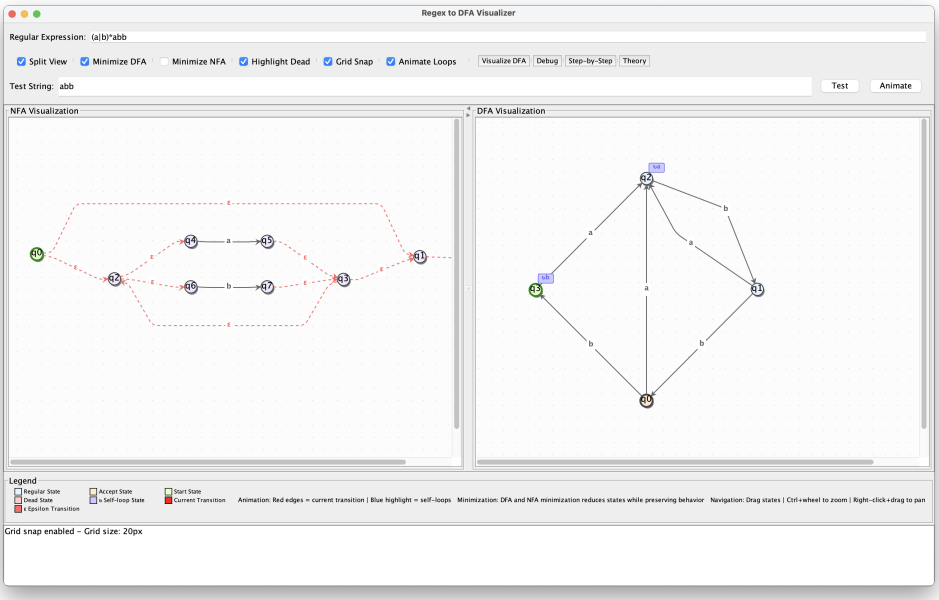

Figure 1: Example visualization of a DFA generated from a regular expression

2 System Architecture

The application follows a modular architecture, dividing functionality across several key packages:

- **algorithm**: Contains implementations of Thompson's Construction, Subset Construction, and Hopcroft's Algorithm for the conversion and minimization of automata.
- **model**: Provides data structures for NFA and DFA representation.
- **controller**: Manages the automata conversion pipeline and simulation.
- **ui**: Implements the graphical interface and visualization components.
- **util**: Contains utility classes.

3 Key Components

3.1 Regular Expression Parser

The regular expression parser converts input expressions into tokens that can be processed by Thompson's Construction. The parser supports:

- Basic symbols for literal character matching
- Alternation (|) for matching either of two expressions
- Kleene star (*) for matching zero or more occurrences
- Plus (+) for matching one or more occurrences
- Optional (?) for matching zero or one occurrence
- Grouping with parentheses
- Character classes with square brackets
- The dot operator for matching any character

3.2 Automata Models

The project provides comprehensive implementations for both NFAs and DFAs:

- **NFA (Nondeterministic Finite Automaton)**: Supports epsilon transitions and multiple possible transitions for a given input symbol.
- **DFA (Deterministic Finite Automaton)**: Provides deterministic behavior with exactly one transition for each input symbol from any state.

3.3 Visualization Components

The visualization framework renders automata as interactive graphs with:

- Color-coded states and transitions
- Zoom and pan controls
- Interactive simulation for testing input strings
- Split view to compare NFA and DFA side by side
- Export capabilities for saving visualizations as PNG files

4 Algorithms

4.1 Thompson's Construction

Thompson's Construction Algorithm

Thompson's Construction algorithm converts a regular expression into an NFA with epsilon transitions. The algorithm builds NFAs for basic components of the regular expression and then combines them according to the operations in the expression.
For each regular expression element:

- **Symbol**: Creates an NFA with two states and a transition on that symbol
- **Concatenation**: Connects two NFAs with an epsilon transition
- **Alternation**: Creates a new start state with epsilon transitions to the start states of two NFAs
- **Kleene Star**: Adds epsilon transitions to create a loop structure

4.2 Subset Construction

Subset Construction Algorithm

The Subset Construction algorithm converts an NFA to a DFA by:

- Computing epsilon closures for each set of NFA states
- Creating DFA states from sets of NFA states
- Constructing transitions based on combined behavior of NFA states
- Identifying accepting states based on whether any constituent NFA state is accepting

4.3 Hopcroft's Algorithm

Hopcroft's Minimization Algorithm

Hopcroft's Algorithm minimizes a DFA by combining equivalent states. The algorithm:

- Begins with an initial partition of states into accepting and non-accepting sets
- Refines partitions iteratively by splitting them based on transition behavior
- Continues until no further refinement is possible
- Constructs a minimal DFA using one representative state from each partition

The implementation of Hopcroft's Algorithm in AutomataViz includes optimization for special cases:

```
public DFA minimize(DFA dfa) {
    // Initial partitioning into accepting and non-accepting states
    Set<DFA.State> acceptStates = dfa.getAcceptStates();
    Set<DFA.State> nonAcceptStates = new HashSet<>(allStates);
    nonAcceptStates.removeAll(acceptStates);

    // Refinement queue and partitioning logic
    // ...

    // Create minimized DFA from partitions
    return createMinimizedDfa(dfa, partitions);
}
```

5 Implementation Details

5.1 DFA Minimization

The DFA minimization process is implemented using Hopcroft's Algorithm, which has a time complexity of O(n log n) where n is the number of states. The implementation handles edge cases such as:

- Empty DFAs or DFAs with a single state
- DFAs where all states are accepting or all are non-accepting
- DFAs with no transitions for certain input symbols

Algorithm	**Time Complexity**	**Space Complexity**
Thompson's Construction	$O(n)$	$O(n)$
Subset Construction	$O(2^n)$	$O(2^n)$
Hopcroft's Algorithm	$O(n \log n)$	$O(n)$

Table 1: Complexity of algorithms used in AutomataViz

5.2 Visualization Engine

The visualization system leverages the JGraphT library for graph representation and JGraphX for rendering. Custom styling is applied to differentiate:

- Start states (green border)
- Accept states (orange fill)
- Regular states (blue/purple)
- Self-loops (blue arrows)
- Epsilon transitions (red dashed lines)
- Normal transitions (grey arrows)

Figure 2: Visualization legend showing different state and transition types

6 Project Structure

The project follows a standard Maven structure with clear separation of concerns:

```
src/
|- main/
| |- java/com/dfavisualizer/
| | |- algorithm/ # Automata algorithms
| | |- controller/ # Application logic
| | |- model/ # Domain models (DFA, NFA)
| | |- ui/ # UI components and visualizers
| | \- util/ # Utility classes
| \- resources/ # Application resources
\- test/ # Test classes and resources
```

7 Building and Running

The project uses Maven for dependency management and build automation:

Build Commands

```
# Build
mvn clean package

# Run JAR
java -jar target/AutomataViz-1.0-SNAPSHOT-jar-with-dependencies.jar

# Or run with Maven
mvn exec:java -Dexec.mainClass="com.dfavisualizer.ui.MainApp"
```

8 References

1. Sipser, M. (2012). *Introduction to the Theory of Computation*. Cengage Learning.

2. Java Graph Library (JGraphT): https://jgrapht.org/

3. AutomataViz GitHub Repository: https://github.com/adhamafis/AutomataViz