Design and Implementation of a Deterministic and Nondeterministic Finite Automaton Simulator

Senior Project

Student: Camron Christopher Dennler

Advisor: Dr. Hasmik Gharibyan

Computer Science and Software Engineering Department
California Polytechnic State University
Spring 2020

Table of Contents

Abstract

The purpose of this project is to assist students in visualizing and understanding the structure and operation of deterministic and nondeterministic finite automata. This software achieves this purpose by providing students with the ability to build, modify, and test automata in an intuitive environment. This enables a simple and efficient avenue for experimentation, which upholds the Cal Poly ideal of Learning by Doing.

Readers of this report should be familiar with basic concepts in the theory of finite state machines; a general understanding of object-oriented programming is also necessary.

1 Problem Description

This project's aim is to provide a piece of software with a graphical user interface in which students in the Theory of Computation course (CSC445) can gain a deeper understanding of the functionality of Deterministic Finite Automata (DFAs) and Nondeterministic Finite Automata (NFAs) by designing and running such machines using an interactive state diagram. This software must be capable of creating functional DFAs and NFAs with a variable number of states and variable alphabet sizes. It also must be capable of displaying an understandable animation to highlight the state-by-state testing process for an input string.

1.1 Motivation

Throughout my time at Cal Poly, there have been few classes that intrigued and entertained me to the same extent as Theory of Computation with Dr. Vakalis. Most of the class was rooted in simplifications and generalizations of forms of computation, but my peers and I still struggled to process some of the conceptual material. However, I found myself thoroughly enamored, as all of the concepts seemed like a perfect blend of computer science theory and logic puzzles.

When working to develop a project topic, I decided that I wanted to reduce some of the barriers to understanding that future students may face when taking this class. This led me to explore the idea of building a test-animating simulator for DFAs and NFAs. I reached out to ask Dr. Gharibyan if she would advise this project, as she also teaches Theory of Computation and I hoped to work on this project with the goal of it being adopted by a professor to use in their class. I believe that viewing an interactive and dynamic visual representation of these automata would not only help students grasp the concepts, but also prompt students to view the automata as a form of simple theoretical computing rather than just a logical graph.

1.2 Design Overview

Before working on this project, I had plenty of experience using Java, which provides an object-oriented structure that would be valuable for designing the internal functions of a DFA or NFA. However, I had minimal experience developing graphical user interface applications in any language. I found that Java had plenty of graphical packages that can be used to create robust user applications, so I chose an open source framework called JavaFX along with one of its community-developed extension frameworks. Learning JavaFX was an overarching task for most of the project, and I intended this learning to serve as a challenge for myself.

The first part of the project consisted of designing and implementing the backend for the software. The first backend step was creating the object-oriented structure for DFAs and NFAs (from this point forward referred to together as "finite automata"). This included defining objects representing a finite automaton as a whole, objects representing states that exist within a finite automaton, and objects representing transitions between states in a finite automaton. The second backend step was developing input string tests, which would determine whether a string over the automaton's alphabet would be accepted or rejected by the automaton. A final backend step was to develop a command-line interface to run the object-oriented structure and input string testing

functionality. This allowed me to begin constructing my own finite automata to test the backend structure, without first developing a full visual representation.

The second part of the project consisted of developing the frontend graphical user interface to represent the finite automata. The first interface step was sketching ideas of how the interface should appear to the user, considering all actions they should be able to take. It was clear to me that I needed a sidebar for adding/removing states, adding/removing transitions between states, and toggling the initial and final status of the states. I also needed a bottom bar for the user to provide an input test string for the finite automaton. Finally, and most importantly, I needed a large section of the interface to be reserved for the visual graphed state diagram. The second interface step was developing the skeleton to fit my sketches. The last interface step was linking and altering the backend code to fit and interact with the manipulation sidebar and testing bottom bar. This step was also done before developing a full visual representation, because I continued depending on the intermediate step to ensure that all sidebar and bottom bar components were interacting with the backend correctly.

The third and final part of the project consisted of constructing the visual representation of a state diagram in the interface. The first visual representation step involved further altering the backend code to interact with the graphing area of the interface. This allowed individual finite automaton states to appear in the graphing area upon addition and disappear upon removal, and it allowed changes in the manipulation sidebar to cause transitions and initial/final state markers to appear in the graphing area as well. The second visual representation step involved animating the work of an automaton on a given input string, which was done by manipulating the state diagram colors as input strings are processed to clearly display its path through the automaton to the user.

Dr. Gharibyan was involved throughout the design and implementation stages. Her opinions and feedback were extremely valuable, as she steered me towards effective ways of ensuring that the interface was clearly illustrating the concepts of finite automata and their work process. She also helped clarify small differences in the teachings of these concepts between professors at Cal Poly, and we worked together to ensure that this software would be compatible with her coursework.

1.3 Lessons

This project taught me the importance of being able to properly handle open source code. The JavaFX framework is incredibly well-developed, and most of its documentation is filled with examples and explanation. It has community-developed extension frameworks that proved to be very useful in my project. However, two issues with open source frameworks are the potential existence of uncorrected bugs, and an occasional lack of clear documentation. JavaFX was not the culprit, but rather one of the extensions. There was a scenario in which the removal of a state from an NFA would cause the list of transitions specified in the manipulation sidebar to shift and potentially include null values. I spent a lot of time searching for the bug in my code and scouring documentation for answers, but eventually realized that the bug existed in the extension framework. I was forced to develop a slightly ugly workaround to continue using the element from the framework that contained the bug.

Another important lesson I learned from this project was to be patient with myself, as I found myself consistently getting frustrated by the portions of the project I expected to be simple. In hindsight, there were periods in which I overconfidently felt that graphical user development (with a framework I had never used before) could come easily to me. I can with confidence say that it did not! This project helped me to understand that I'm entering a field of work in which very few things are truly simple, and hard work is required to ensure that I never stop learning.

Finally, I realized that I tend to be a little bit too focused on perfecting a single piece of a project at a time, rather than procedurally developing piece by piece and improving incrementally. I occasionally spend too much mental energy on small pieces of a project, even if I find that I am stuck. It's important to move to another task when I get stuck and accept temporary imperfection on individual parts of a project, especially if they're cosmetic. On top of that, if there's anything studying computer science at Cal Poly has taught me, it's that you sometimes need to step away from a problem and approach it again later to keep your perspective fresh.

1.4 Future Work

This project has plenty of room for improvement. When Dr. Gharibyan and I were initially planning the schedule for this project, we included functionality to remove nondeterminism from the NFAs, or in other words, transforming an NFA into an equivalent DFA. We eventually considered it out-of-scope for this project, but it is absolutely an improvement that can be made in time.

Another improvement is preserving finite automata built using this software by giving the user the capability to save to and load from a file. This would allow students to easily keep digital track of their experiments, as well as providing a convenient avenue for students to share their creations with a professor or a peer.

Finally, there are plenty of cosmetic improvements that can be made. When developing this software, functionality was the main focus.

2 Requirements and Implementation

2.1 Usability

The purpose of this software is to assist Cal Poly Theory of Computation (CSC445) students in understanding finite automata concepts. The software is thus intuitive and easy-to-use. Particularly:

- users are intuitively able to understand how to construct a finite automaton's state diagram using clear graphical user interface components

- the software is able to produce a clear and accurate visual representation of the state diagram of a finite automaton

- the software is able to correctly determine whether or not an input string is accepted by a finite automaton while animating the testing process in an understandable way

2.2 Functionality

The software provides the following functionality:

- Setup
    - Choose creation of DFA or NFA
    - Choose alphabet for automaton
    - Clear the graphing area to set up a new automaton

- Automaton Construction and Manipulation
    - Add a state
    - Delete a state and all connected transitions
    - Relocate a state in the graphing area
    - Set and unset state as initial
    - Set and unset state as final
    - Add and remove a transition between two states

- Testing
    - Validate design to ensure it follows finite automaton rules
    - Test input string with animation
    - Test input string without animation

2.3 Technologies

The software utilizes the following technologies:

- Java JDK 14
  Software developed using Java. Will run in Java Runtime Environment 14.

- JavaFX and ControlsFX
  Software's graphical user interface developed using the JavaFX Version 14 Framework, as well as the ControlsFX Version 11.0.1 Framework.

- Maven
  Handles framework dependencies and build path for the software.

- Eclipse
  All code for software written using the open source Eclipse SDK.

2.4 Description of Classes

2.4.1 Application Classes

`App`
Class for the graphical user interface of the software. Constructs the skeleton of the interface and connects all interactive components to the object-oriented structure of a finite automaton. Handles error checking. Makes use of the `Automaton` interface to store, change, and call functions from either a `DFA` object or an `NFA` object. Contains a button to reset the simulator, validate a finite automaton, or test input strings on the automaton.

2.4.2 Automaton Structure Interfaces

`Automaton`
Interface implemented by `DFA` and `NFA` classes.

`AutomatonState`
Interface implemented by `DFAState` and `NFAState` classes.

2.4.3 Automaton Structure Classes

`DFA`
Class that represents a deterministic finite automaton. Contains a list of `DFAState` objects, the alphabet to use, the fields to monitor in the manipulation sidebar, and the graphing area of the interface. Provides functions for adding and removing states, as well as testing input strings.

`NFA`
Class that represents a nondeterministic finite automaton. Contains a list of `NFAState` objects, the alphabet to use, the fields to monitor in the manipulation sidebar, and the graphing area of the interface. Provides functions for adding and removing states, as well as testing input strings.

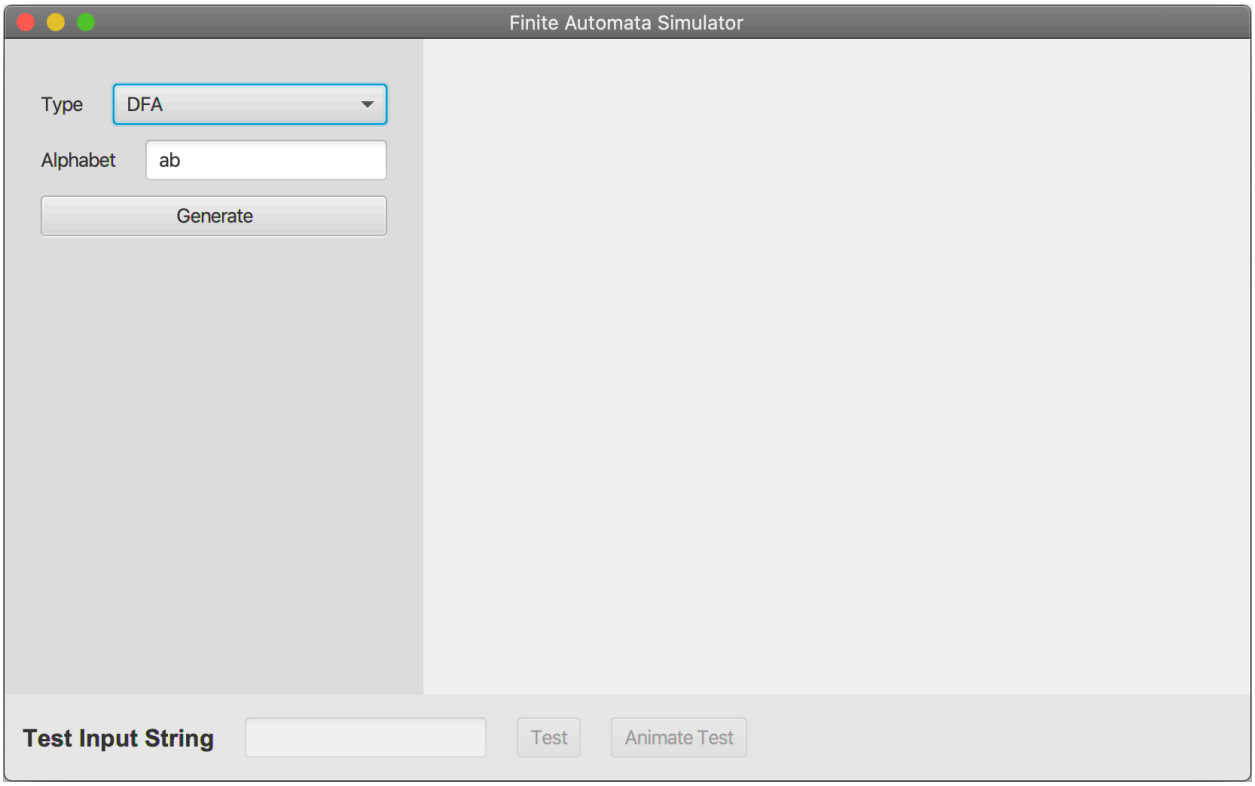
DFAState
Class that represents a state in a DFA. Contains a state number, a dictionary of `Transition` objects organized by the destination state, an initial state marker, and a final state marker. Provides a set of property fields in the manipulation sidebar to update initial status, final status, and transitions. Provides a circular state icon to the parent `DFA` object for graphing as well as functions to relocate the icon on a click and drag. Provides functions to animate input string testing as it gets processed.

NFAState
Class that represents a state in an NFA. Contains a state number, a dictionary of `Transition` objects organized by the destination state, an initial state marker, and a final state marker. Provides a set of property fields in the manipulation sidebar to update initial status, final status, and transitions. Provides a circular state icon to the parent `NFA` object for graphing as well as functions to relocate the icon on a click and drag. Provides functions to animate input string testing as it gets processed.

Transition
Class that represents a transition from one state to another. Makes use of the `AutomatonState` interface to connect either two `DFAState` objects or two `NFAState` objects, clarified as source and destination. Tracks alphabet symbols used to label transition. Provides an arrow graphic to the parent `Automaton` object for graphing, as well as functions to relocate and adjust the arrows when source or destination states are moved. Provides functions to animate input string testing process when the transition is used.

3 User's Manual

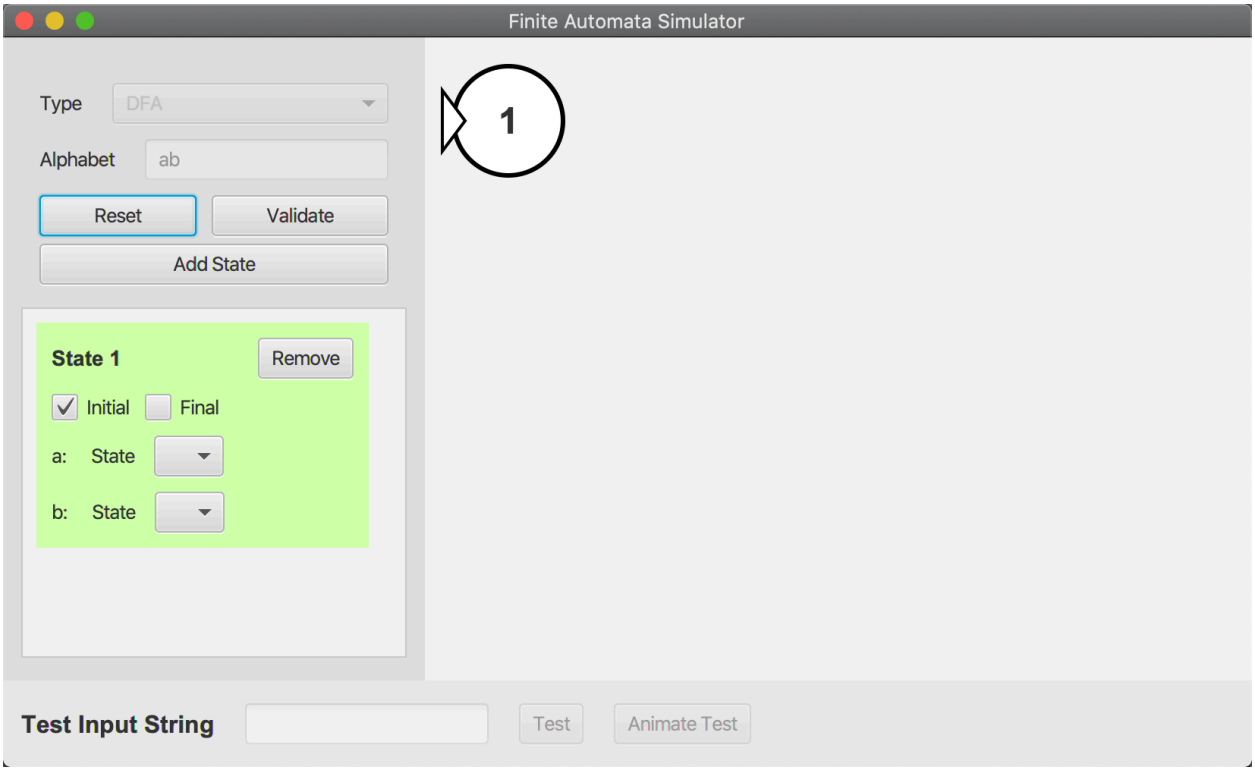
3.1 Creating an Automaton



The simulator opens to the entry screen.

- The "Type" drop-down box allows the user to select construction of a DFA or an NFA.
- The "Alphabet" field allows the user to input a set of symbols to use as the automaton's alphabet.
- Alphabet input errors:
    - Alphabet cannot contain repeated symbols.
    - Alphabet must contain at least one symbol.
- The "Generate" button will set up the manipulation sidebar for the specified type of automaton with the specified alphabet. The simulator will then be ready for the user to add and delete states and transitions.

3.2 Constructing an Automaton

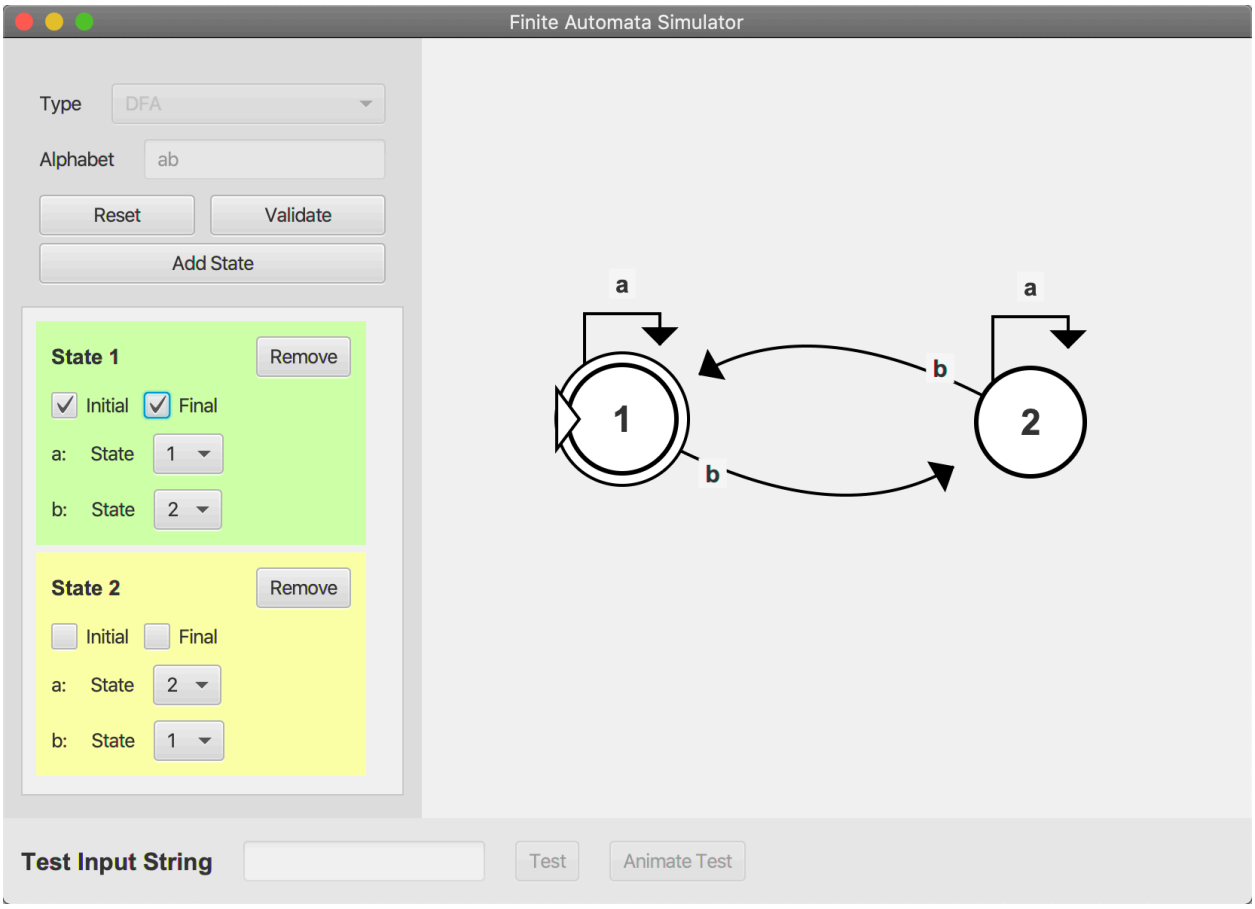


When the "Generate" button is clicked, the simulator will show the automaton construction screen. The left portion of the screen is the manipulation sidebar, where the user can choose from multiple options to construct an automaton. The right side of the screen is the graphing area.

- The "Add State" button will add a new state to the automaton, and the circle icon for the new state will appear in the graphing area. It will also add a set of state property fields to the manipulation sidebar.
- The property pane in the manipulation sidebar contains sets of state property fields, with each set colored either green or yellow. Each set of property fields corresponds to a state, and interacting with these fields will manipulate the state and its transitions in the automaton.

Note: When the "Generate" button is clicked and the automaton construction screen appears, one state and its set of property fields will be added automatically.

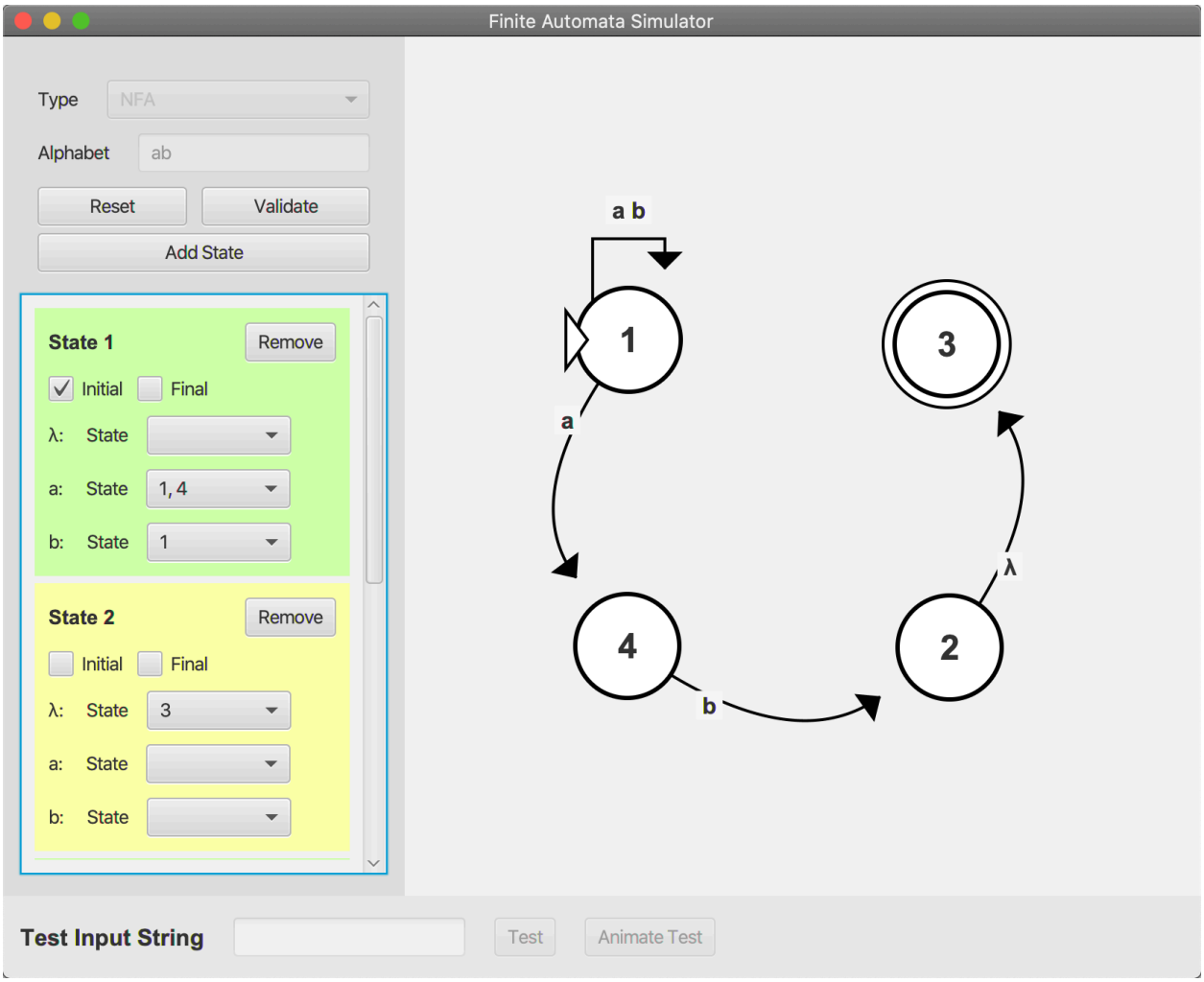
3.2.1 Property Fields in a DFA



The property fields in the manipulation sidebar allow the user to change the DFA's initial states, final states, and transitions. It also allows the user to remove states.

- The "Initial" checkbox will toggle the corresponding state as initial, represented by a triangle on the left side of the icon.
- The "Final" checkbox will toggle the corresponding state as final, represented by a double-outlined circle.
- The "a: State" drop-down box will determine to which state the corresponding state will transition to on input symbol "a". The property field has a State drop-down box for each symbol in the DFA's alphabet.
- The "Remove" button will remove the corresponding state from the DFA, its property fields from the manipulation sidebar, all connected transitions, and all corresponding graphical components.

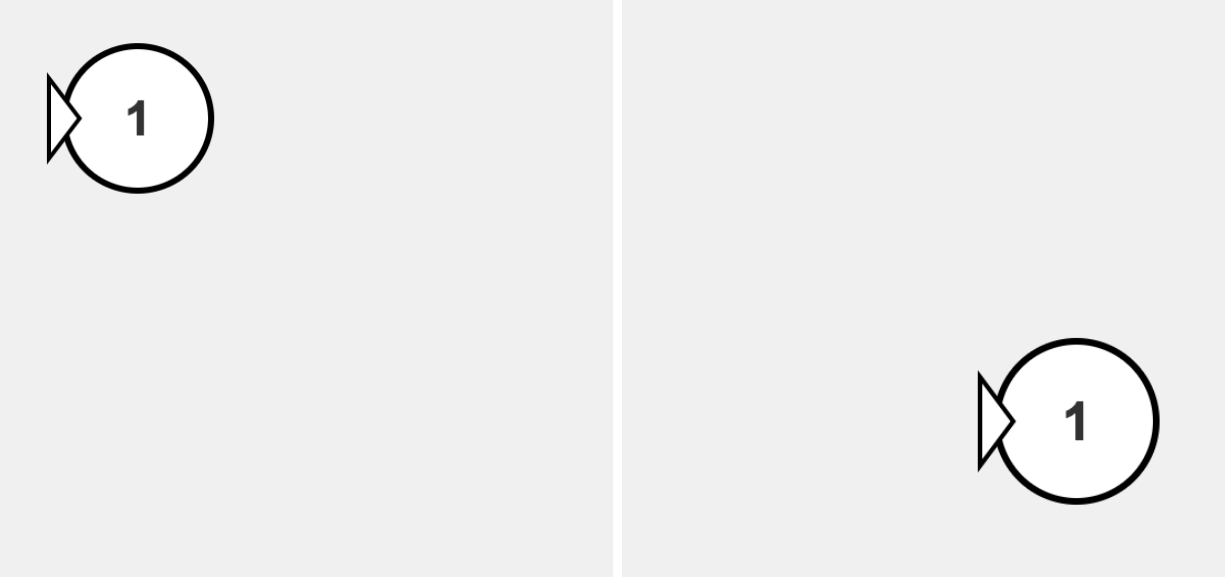
3.2.2 Property Fields in an NFA



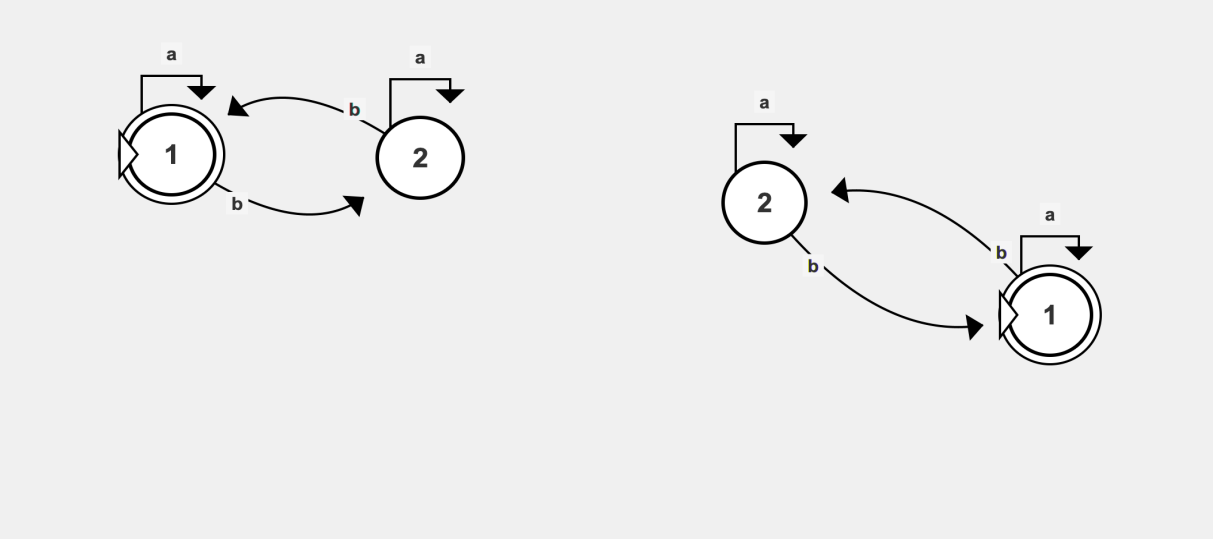
The property fields in the manipulation sidebar allow the user to change the NFA's initial states, final states, and transitions. It also allows the user to remove states.

- The "Initial" checkbox, the "Final" checkbox, and the "Remove" button all serve the same purpose as they do in the DFA property fields.
- The "a: State" drop-down multi-choice box determines to which states the corresponding state will transition on input symbol "a". The property field has a State drop-down multi-choice box for each symbol in the NFA's alphabet.
- The "λ: State" drop-down multi-choice box will determine to which states the corresponding state will transition on a lambda (an instantaneous transition).

3.2.3 Moving States in the Graphing Area

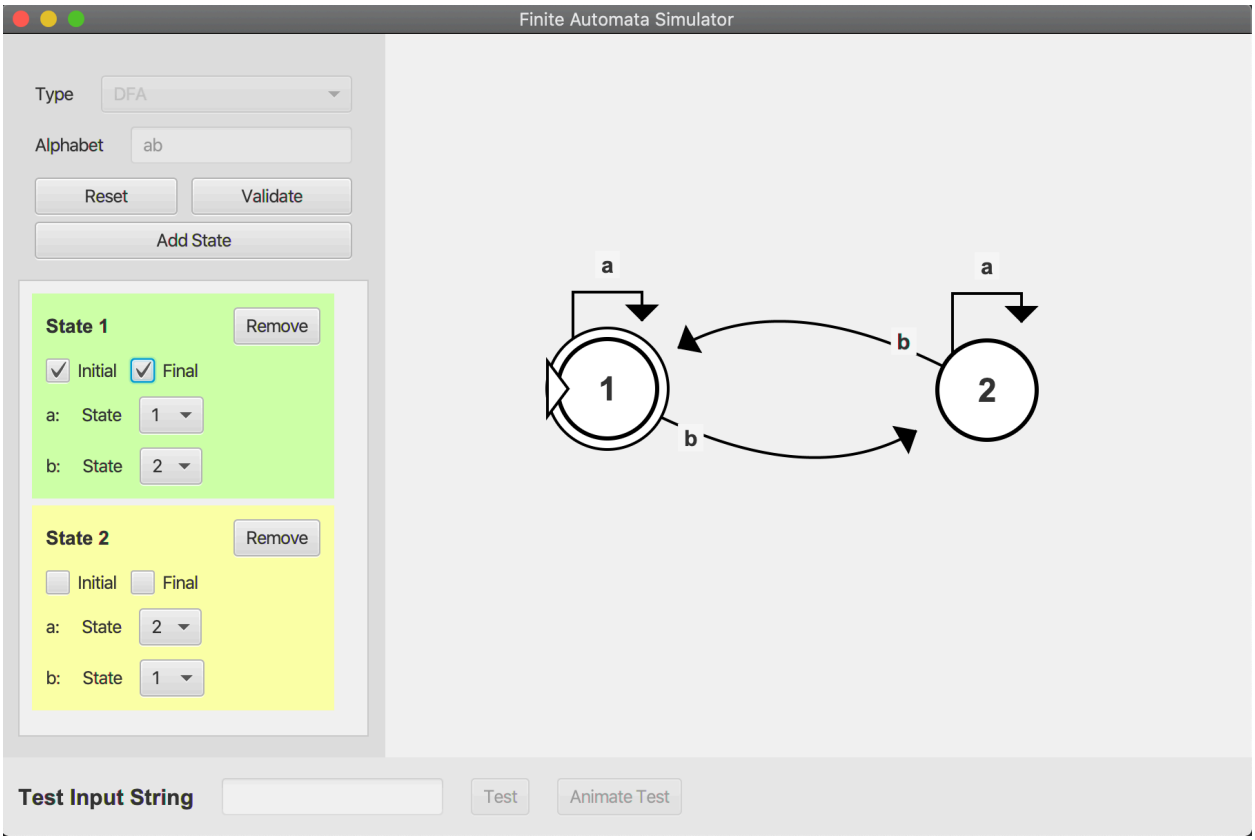


States can be reorganized in the graphing area by clicking, dragging, and releasing its circle icon.



Moving a state in the graphing area will maintain the integrity of the diagram and its full functionality.

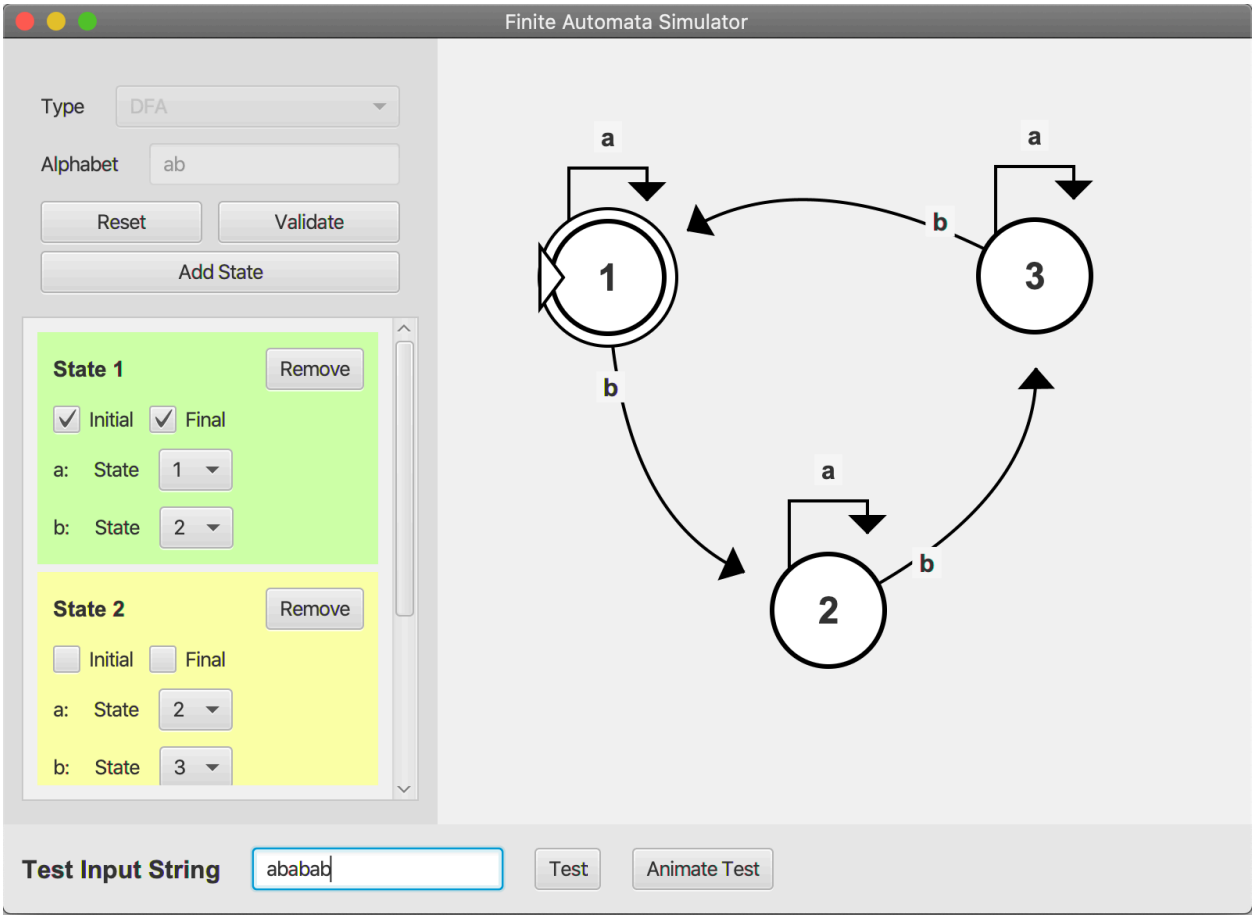
3.2.4 Resetting or Validating an Automaton



The simulator now shows a visual representation of the completed state diagram.

- The "Reset" button will return the simulator to the entry screen and delete the current state diagram.
  Note: The "Reset" button can be used at any point during construction to return to the entry screen and start over. The automaton does not need to be complete.
- The "Validate" button will check if the created graph is a correct representation of a complete finite automaton. Namely, it should satisfy the following requirements:
  - DFA Rules
    - Every state must have exactly one transition for each alphabet symbol.
    - DFA must contain exactly one initial state.
  - NFA Rules
    - States may have 0, 1, or multiple transitions for each alphabet symbol, as well as 0, 1, or multiple lambda transitions.
    - NFA must contain exactly one initial state.
  - If the automaton is valid, input string testing will be enabled.

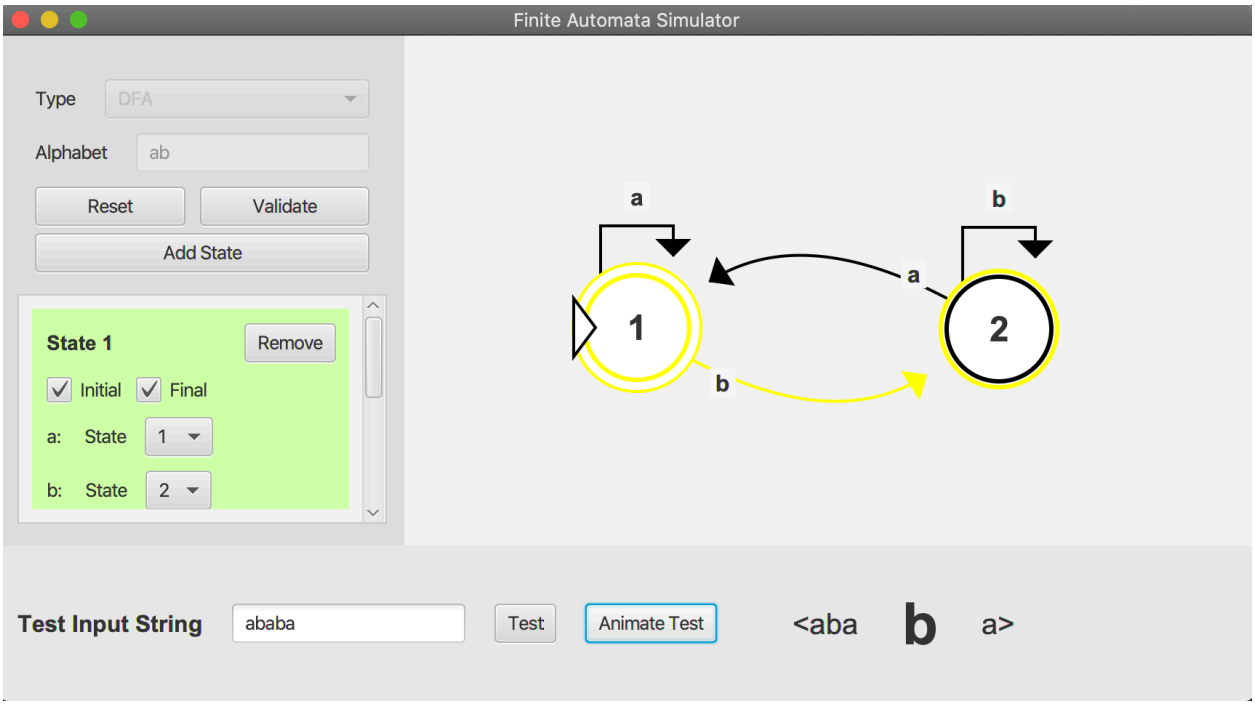
3.3 Testing Input Strings in an Automaton



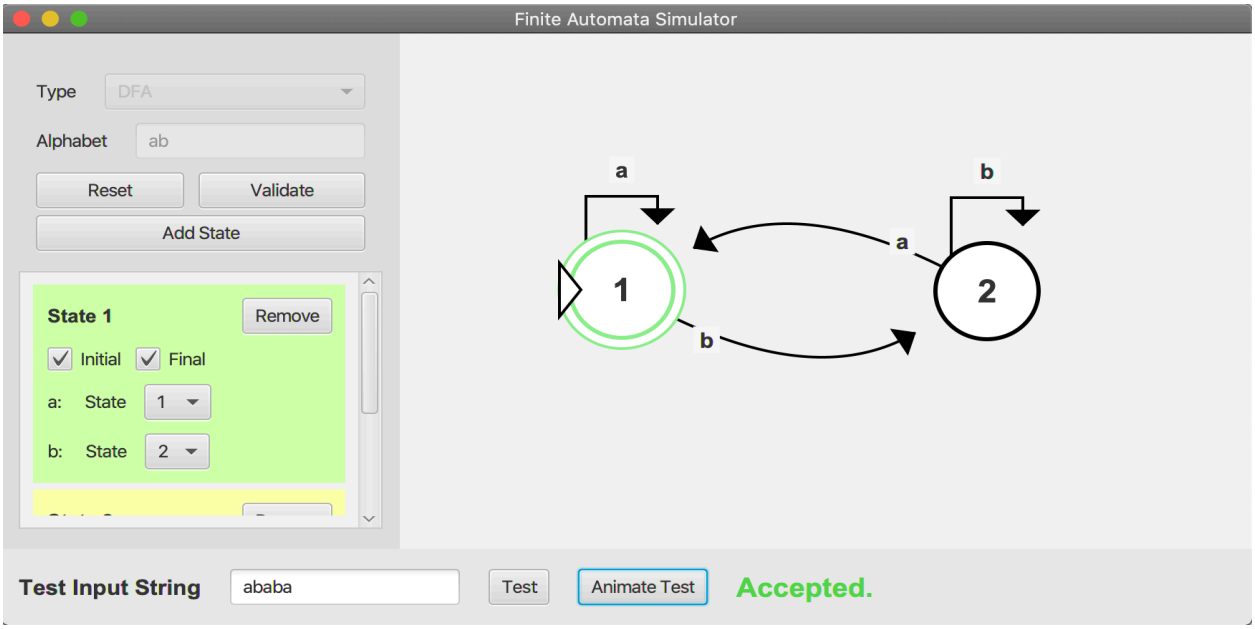
Once an automaton is validated, input string testing is enabled.

- The "Test Input String" field allows the user to input a string over the alphabet to be tested in the automaton.
    - Input string error: testing will be canceled if the input string contains non-alphabet symbols.
- Clicking the "Test" button will display a result of either "Accepted" or "Rejected" instantly.
- Clicking the "Animate Test" button will display the step-by-step animation of the input string's processing, starting from the initial state and moving through its path(s) in the automaton before displaying a result of "Accepted" or "Rejected".
- If changes are made to a validated automaton, input string testing will be disabled and the user must validate it again.

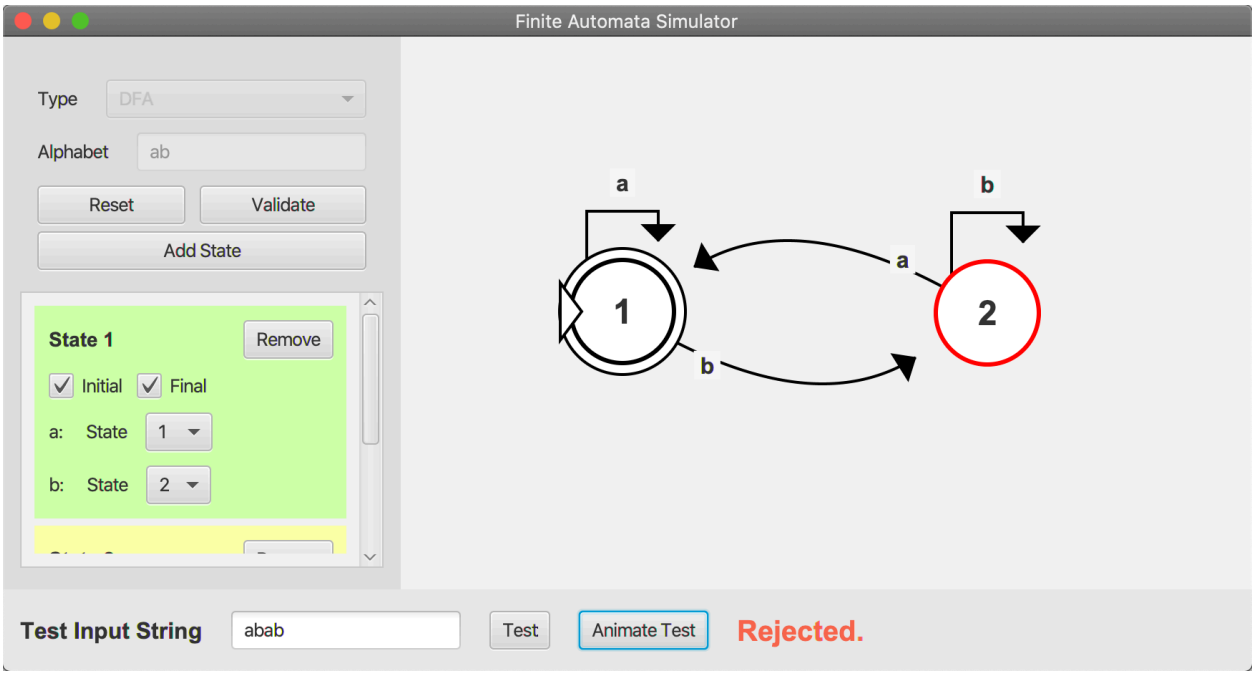
3.3.1 Animation of Testing in a DFA



When an input string's processing is animated in a DFA, the string's deterministic path through the diagram is displayed by continuously highlighting the current state, the followed transition, and the next state. The above example shows the DFA moving from State 1 to State 2 on input "b".



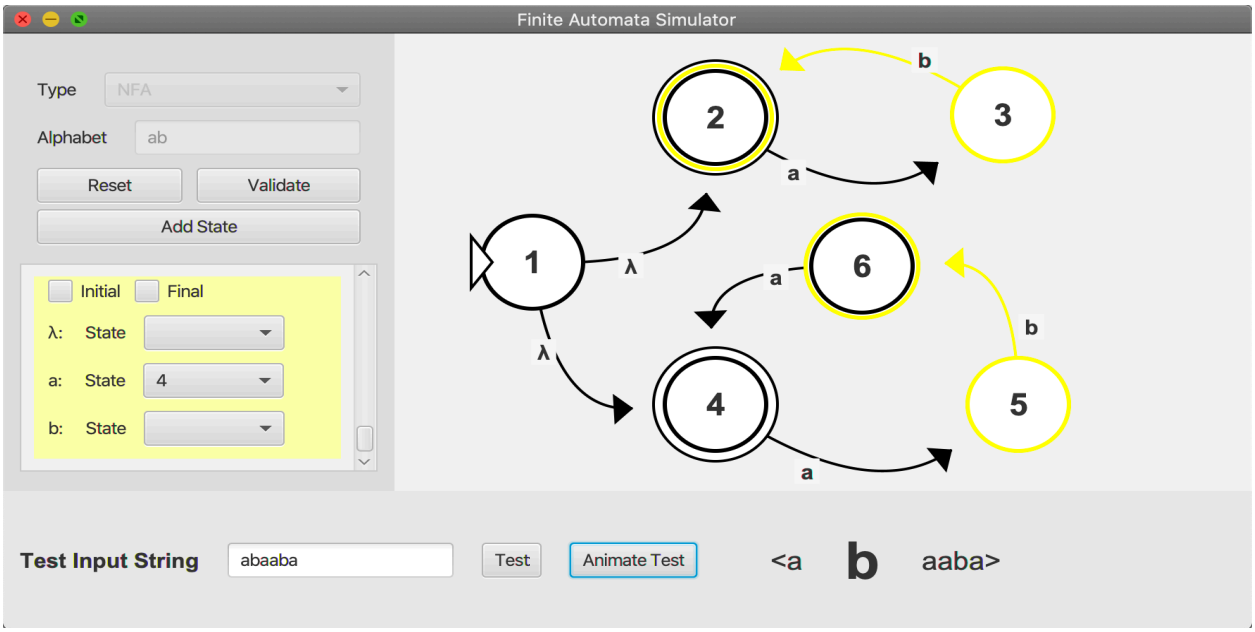
If the processing of the input string ends in a final state, that last state will be highlighted green and a result of "Accepted" will be displayed.



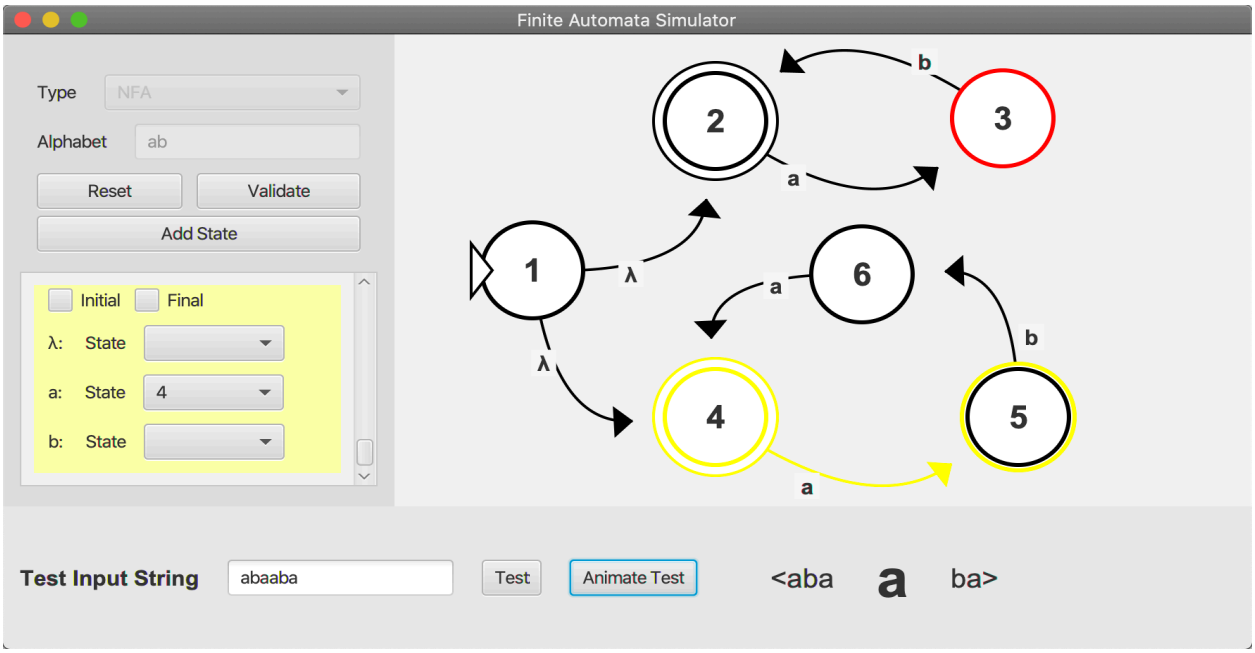
If the processing of the input string ends in a nonfinal state, that last state will be highlighted red and a result of "Rejected" will be displayed.

After the input string has finished processing and the result of the testing is displayed, the highlighted state will remain in color until the user makes a change to the DFA or tests a new string.

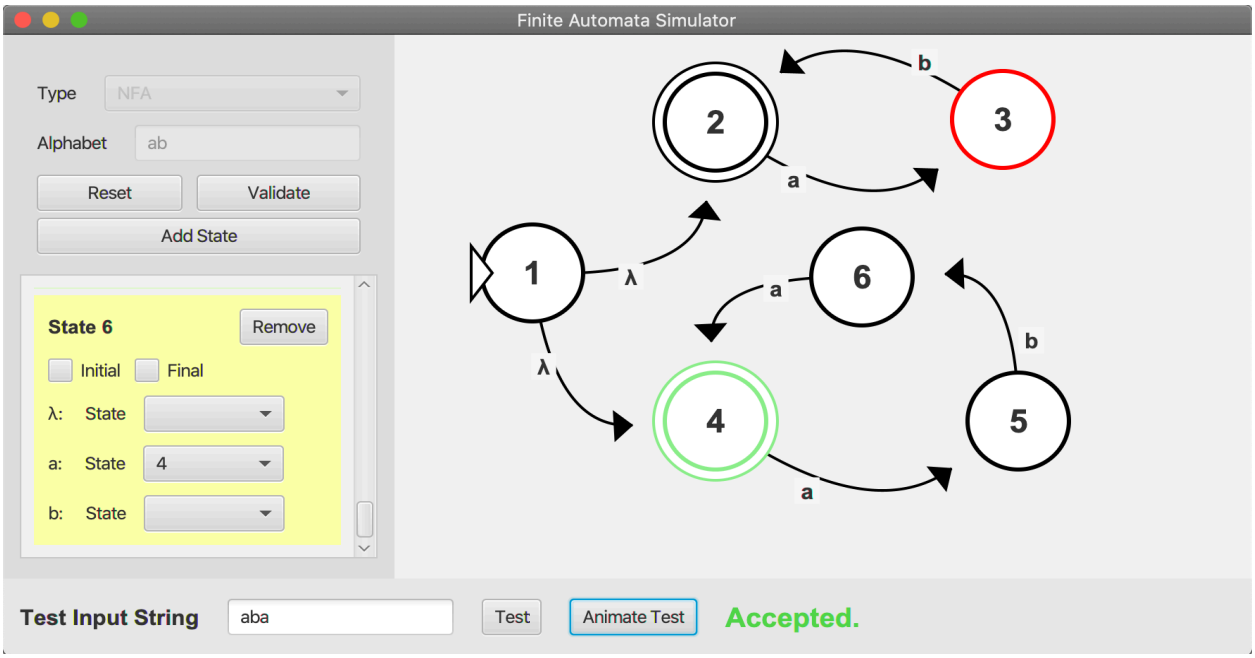
3.3.2 Animation of Testing in an NFA

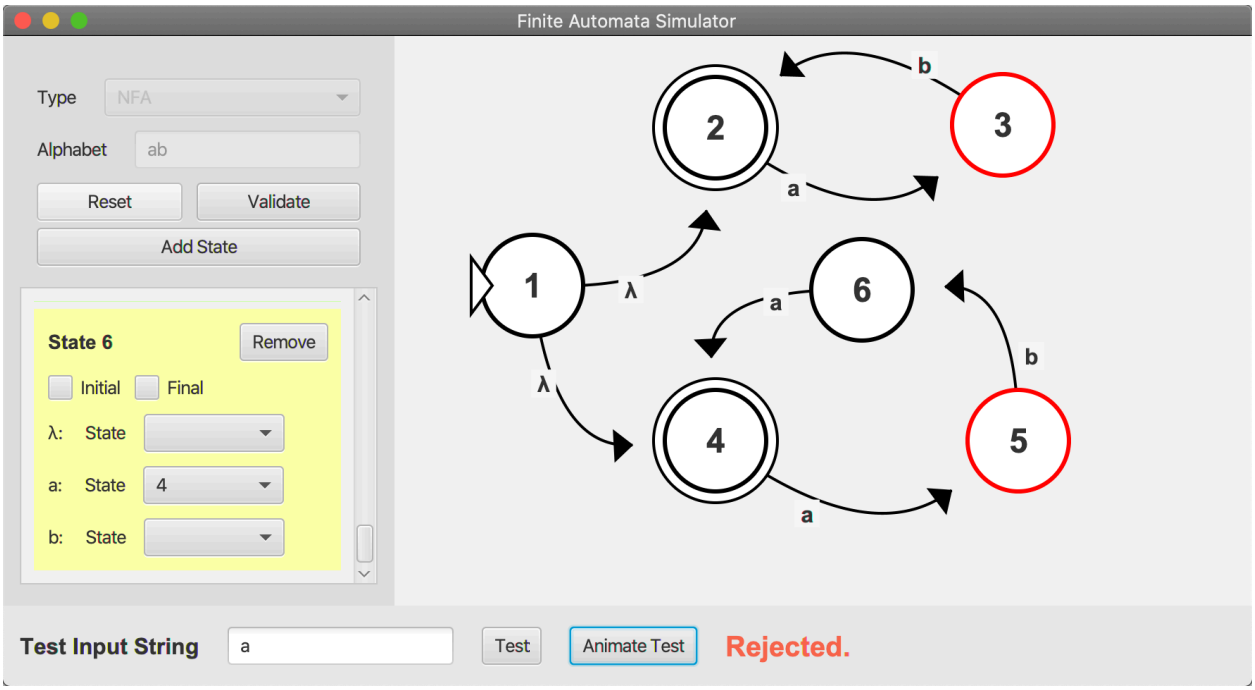


When an input string's processing is animated in an NFA, each of the string's nondeterministic paths through the diagram are displayed by continuously highlighting the current states, the followed transitions, and the next states. The above example shows the NFA moving from State 3 and State 5 to State 2 and State 6 on input "b".

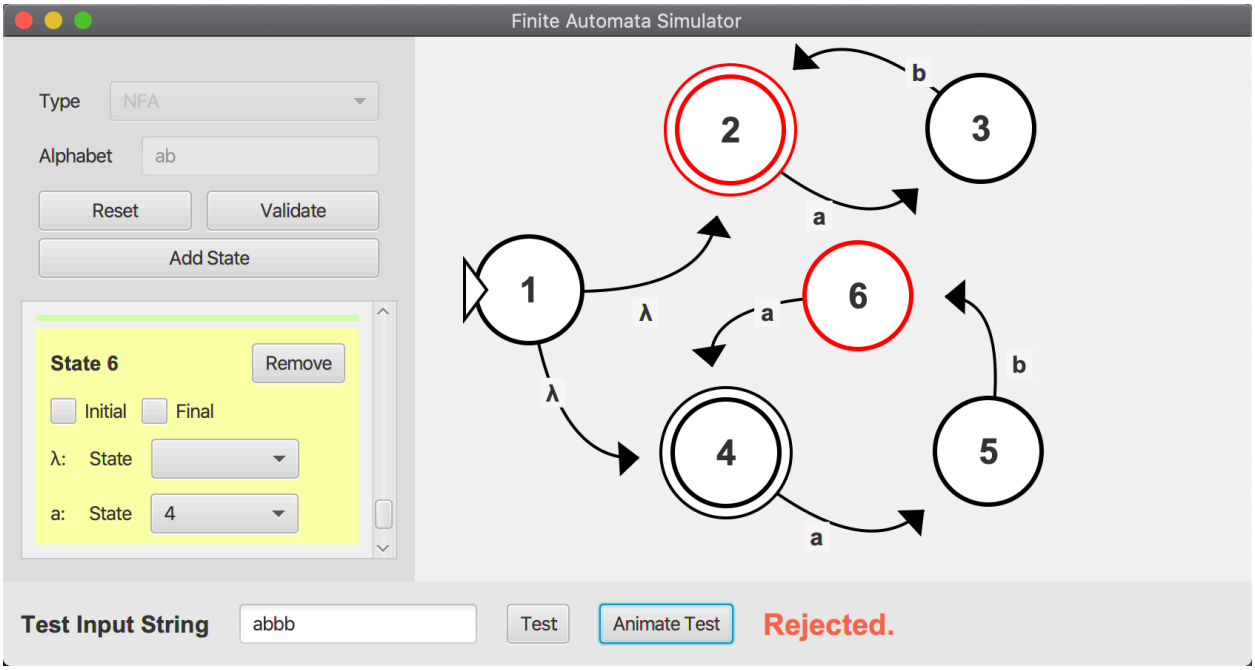


If one of the current states has no outgoing transition for the current alphabet symbol, that state will be highlighted red, processing through that path will be discontinued, and testing will continue with the other current states.





The processing of a string in an NFA may end in multiple states. When processing is complete, each of these states will be highlighted green if it's a final state, or highlighted red if it's a non-final state. If at least one final state is highlighted green, a result of "Accepted" will be displayed. Otherwise, a result of "Rejected" will be displayed.



If the input string cannot be processed completely, i.e. there is an alphabet symbol that cannot be scanned/read from any of the states the NFA is in at the moment, the execution will stop and a result of "Rejected" will be displayed. The above example shows the NFA stopping execution at State 2 and State 6 on the second "b" symbol of the input string, leaving the "bb" postfix unable to be processed.

After the result of testing is displayed, the highlighted states will remain in color until the user makes a change to the NFA or tests a new string.