

UNIVERSITATEA "ALEXANDRU IOAN CUZA" DIN IAȘI
FACULTATEA DE INFORMATICĂ



LUCRARE DE LICENȚĂ

Tool for building and working with automata

propusă de

Bogdan-Andrei Lionte

Sesiunea: *iulie, 2019*

Coordonator științific

Lect. Dr. Mihai Alex Moruz

UNIVERSITATEA "ALEXANDRU IOAN CUZA" DIN IAȘI
FACULTATEA DE INFORMATICĂ

Tool for building and working with automata

Bogdan-Andrei Lionte

Sesiunea: *iulie, 2019*

Coordonator științific

Lect. dr. Mihai Alex Moruz

Avizat,
Indrumator Lucrare de Licență

Titlul, Numele și prenumele _____
Data _____ Semnătura _____

DECLARAȚIE privind originalitatea conținutului lucrării de licență

Subsemnatul(a)
domiciliul in
născut(ă) la data de, identificat prin CNP,
absolvent(a) al(a) Universității „Alexandru Ioan Cuza” din Iași, Facultatea de
..... specializarea
promoția, declar pe propria răspundere, cunoscând consecințele
falsului în declarații în sensul art. 326 din Noul Cod Penal și dispozițiile Legii Educației
Naționale nr. 1/2011 art.143 al. 4 si 5 referitoare la plagiat, că lucrarea de licență cu titlul:

_____elaborată sub îndrumarea dl. / d-na
_____, pe care urmează să o
susțină în fața comisiei este originală, îmi aparține și îmi asum conținutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licență să fie verificată prin
orice modalitate legală pentru confirmarea originalității, consimțind inclusiv la introducerea
conținutului său într-o bază de date în acest scop.

Am luat la cunoștință despre faptul că este interzisă comercializarea de lucrări științifice
în vederea facilitării fasificării de către cumpărător a calității de autor al unei lucrări de licență,
de diploma sau de disertație și în acest sens, declar pe proprie

răspundere că lucrarea de față nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Data azi,

Semnătură student

DECLARAȚIE DE CONSIMȚĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul „*Tool for building and working with automata*”, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de Informatică.

De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea „Alexandru Ioan Cuza” din Iași, să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Iași, *data*

Absolvent *Bogdan-Andrei
Lionte*

(semnătura în original)

Introduction	3
Motivation	3
Objectives	3
Chapters	3
1. Similar solutions	4
1.1. Automata editor	4
1.2. AutomatonSimulator	5
1.3. Finite State Machine Designer	6
2. Application architecture	8
3. Development	10
3.1. Models	10
3.1.1. Automaton	10
3.1.2. States	10
3.1.3. StateArc	11
3.2. Persistence	11
3.2.1. Format	11
3.2.2. GUI	12
3.3. Building an automaton	12
3.3.1. Add symbol	13
3.3.2. Create transition	13
3.3.3. Delete state	14
3.3.4. Special cases	14
3.4. Drawing an automaton	14
3.4.1. States	14
3.4.2. Arcs	15
3.4.3. Symbols	17
3.4.4. Dragging	18
3.5. Parsing	19
3.5.1. Advancing	19
3.5.2. Highlighting	19
3.5.3. Info label	20
3.5.4. Success	20
3.5.5. Error	20
3.6. Minimizing	20
3.6.1. Basic notions	20

3.6.2.	Data structures	21
3.6.3.	Algorithm	21
3.6.4.	Example	22
3.7.	Settings	23
4.	Conclusions	24
	Bibliography	25

Introduction

Motivation

The purpose of the application is to help the user create, visualize and execute specific operations on automata. Automata have many uses, for example in compilers, parsers, interpreters. In the first phase of the compiling process, code is turned into tokens and then the tokens are parsed by a deterministic finite automaton. The tokens can be accepted or not. If they are not accepted by the automaton, the compiler will show an error. Automata are also useful in probabilities, for example in representing Markov Chains. The transitions in such automata are probabilistic. Regular expressions are also widely used in many software applications and are implemented with automata.

Objectives

The main objectives are:

- Create an automaton step by step by adding states and transitions
- Save the created automaton to system file
- Load an automaton from system file
- Minimize a deterministic finite automaton
- Visually and interactively parse words with the current automaton
- Have a simple, clean and intuitive interface
- Provide a customizable representation of an automaton
- Possibility to align states to a grid of customizable size

Chapters

I will continue with the description of the chapters that follow. In the first chapter, “Similar solutions” I will mention some similar applications, explain what they can do better and what they lack. The second chapter contains the application architecture and the reason why that particular architecture was chosen. The next chapter will be focused on the development process, where I explain how I implemented certain features and why I implemented them in that certain way. As a last chapter I will include some conclusions.

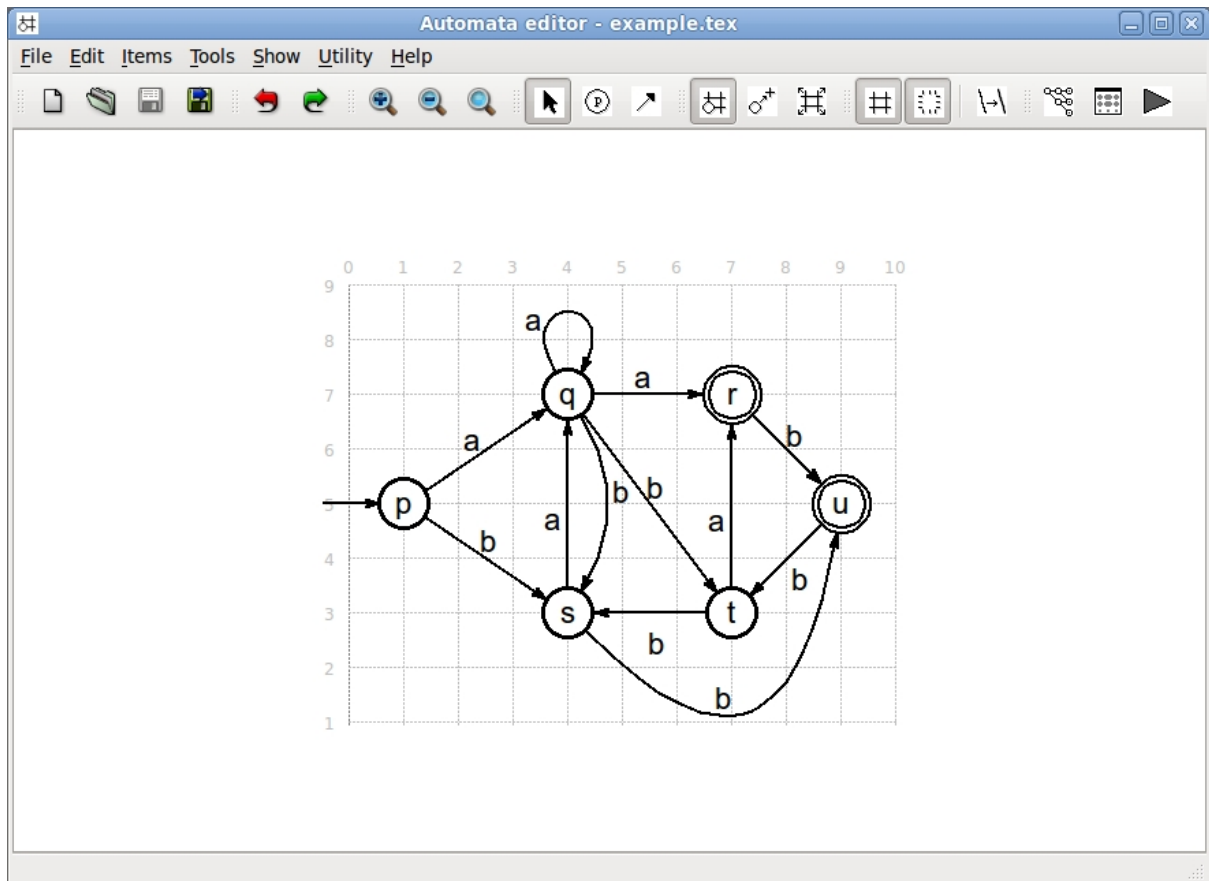
Here I will write about what objectives I managed to achieve and how I am going to improve the application in the future.

1. Similar solutions

1.1. Automata editor

A similar application is **Automata editor**, available at <http://automataeditor.sourceforge.net>. Automata editor is vector editor for drawing finite automata according to VauCanSon-G format (a LaTeX package). The editor supports export to several vector formats (e.g. EPS, SVG, VauCanSon-G, GraphML, ...) as well as bitmap formats (e.g. BMP, JPEG, ...). The program offers a basic tools for working with finite automata such as basic algorithms for finite automata, export of transition table to the LaTeX format and simulation of work of finite automata with possibility to export the steps of simulation in the VauCanSon-G format. There were designed an interface to external dynamically loaded modules, so that any algorithm can be added to extend the editor's functionality. [1]

The fact that it has many features means that it is more complex and has a steep learning curve. The interface has an old design and is not intuitive.



[1]

1.2. AutomatonSimulator

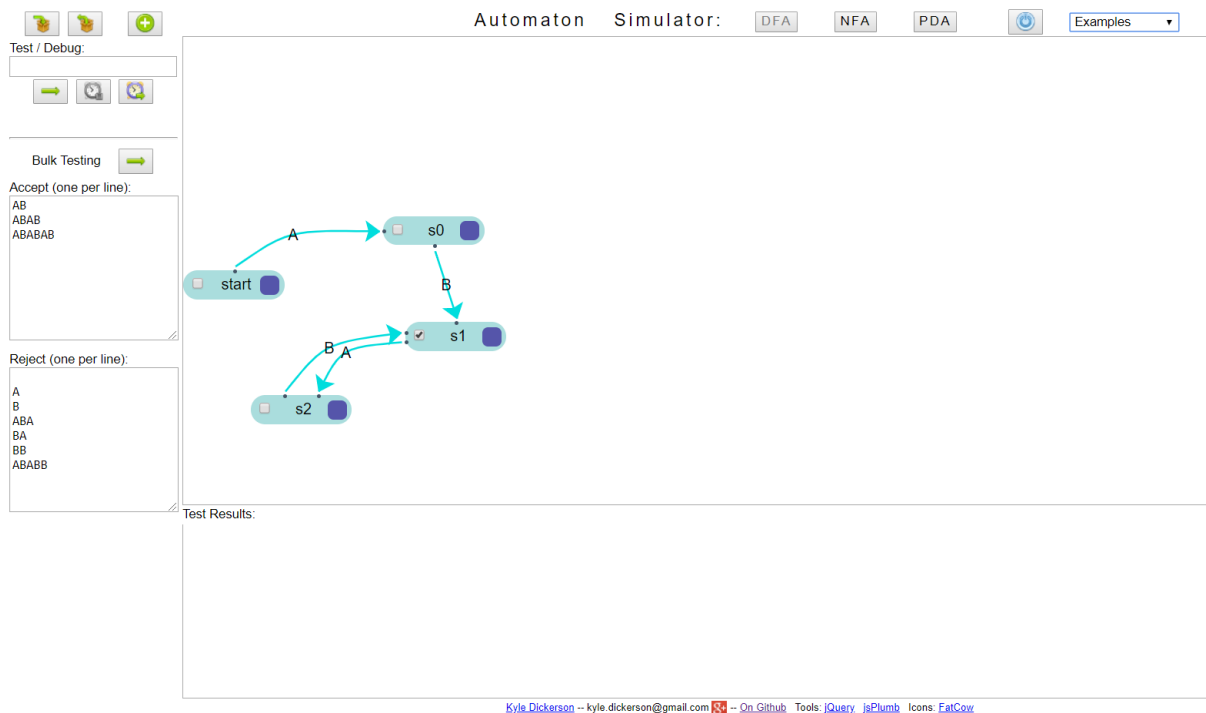
AutomatonSimulator is another similar solution exposed as a website hosted at <http://automatonsimulator.com>

A website that simulates various finite state machines: Deterministic Finite Automata (DFA), Nondeterministic Finite Automata (NFA), Pushdown Automata (PDA).

A GUI is provided to create, save/load (browser local storage), export/import (plaintext format), and debug finite state machines.

A large set of test strings can be designated to be accepted or rejected and then all tested at once.

Or, a specific string can be step-debugged to see exactly how the finite state machine would handle it. [2]



[3]

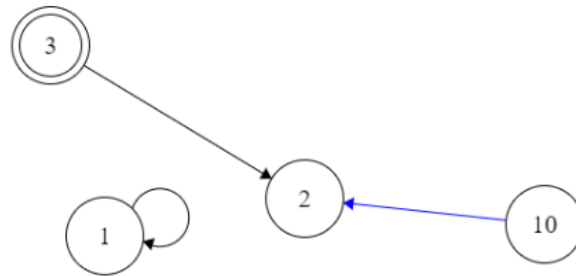
The fact that it is hosted on the web might make the user experience of someone with a slow internet connection an unpleasant one.

The lack of documentation or user guide directly on the website is also a disadvantage.

1.3. Finite State Machine Designer

Finite State Machine Designer at <http://madebyevan.com/fsm/> is a simple web application which serves for designing finite state machines.

Finite State Machine Designer



Export as: [PNG](#) | [SVG](#) | [LaTeX](#)

The big white box above is the FSM designer. Here's how to use it:

- **Add a state:** double-click on the canvas
- **Add an arrow:** shift-drag on the canvas
- **Move something:** drag it around
- **Delete something:** click it and press the delete key (not the backspace key)
- **Make accept state:** double-click on an existing state
- **Type numeric subscript:** put an underscore before the number (like "S_0")
- **Type greek letter:** put a backslash before it (like "\beta")

This was made in HTML5 and JavaScript using the canvas element.

[4]

The good thing about this application is that it is focused only on drawing automata, thus it is very simple and easy to use. The bad thing is that it doesn't support operations on automata, like minimization or parsing. Also, the user is not able to save an existing automaton in a format that allows him to load it later, but only to save it as PNG, SVG or LaTeX.

Contributions

I used an MVC architecture because it offers many benefits that would fit my application. It brings high cohesion and loose coupling which make the application easy to modify and extend.

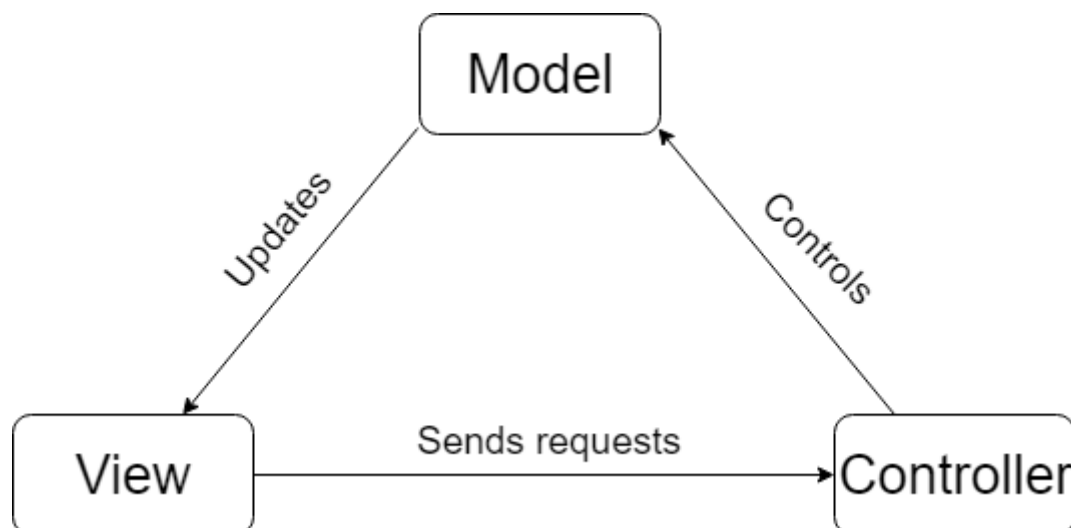
I chose proper data structures like sets, hashmaps, lists, based on the problem I had to solve in order to reduce time and space complexity.

For GUI I chose JavaFX over Swing because JavaFX works better with MVC. Another advantage is the look and feel, which is more modern. JavaFX UI development is easier and faster thanks to the integrated scene builder in IDEs.

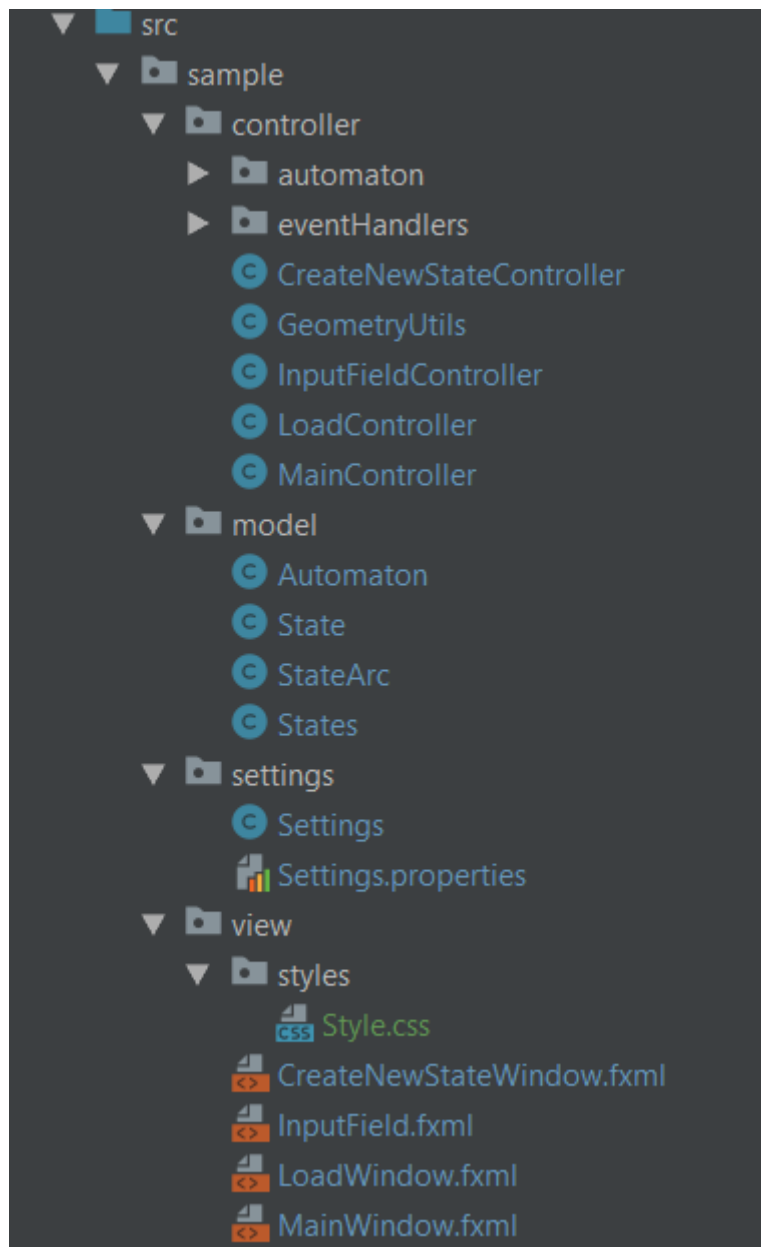
2. Application architecture

The application is written in java, using JavaFX, a client application platform for desktop.

From an architectural point of view, it follows the MVC pattern, which stands for Model View Controller.



In this case, the user interacts with the view, which is represented by the interface build with fxml files. Interface nodes have event handlers attached to them so they can be handled by controllers when a user interacts with them. When a controller receives an event, it starts manipulating the model, which is the automaton shown by the view. When the model changes, it updates the view and the user gets the feedback.

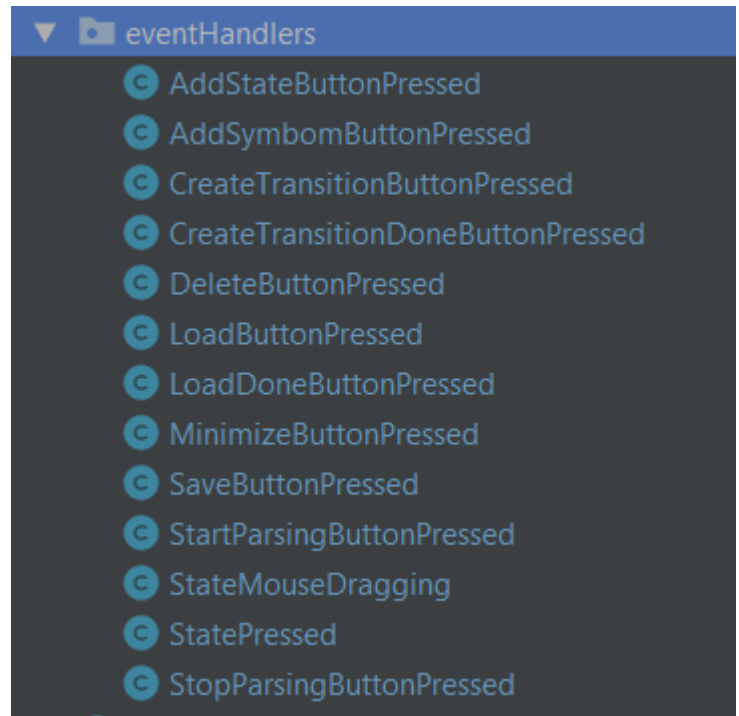
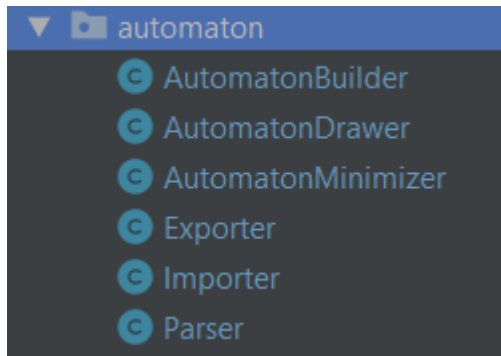


Views contain the main window, which consists of a GridPane layout with a Pane for drawing the automaton and another GridPane for grouping the buttons.

Views have css files for styling UI elements, like button skins, paddings, font styles, sizes.

The model package holds all the automata information, initial state, final states, normal states, transitions, set of symbols.

Controller package contains the main logic of the application, handled by controllers, event handlers and classes specialized in importing, exporting, building, drawing, parsing and minimizing automata.



3. Development

3.1. Models

3.1.1. Automaton

The **Automaton** class contains the states and the possible transitions.

3.1.2. States

States class is a wrapper for a list of **State** objects with methods for retrieving a state by it's name or retrieving the destination of a state with a transition symbol.

The **State** class extends from **javafx.scene.control.Label** so that it can be drawn in the interface. It contains a list of states to which it has transitions and a list of states from which transition come. It also has a **StateArc** attribute used for transitions to itself as well as a list of **StateArc** objects for transitions to other states. I chose to have a separate field for self transitions because they have to be handled in a different manner which will be explained in a future chapter.

3.1.3. StateArc

StateArc extends **javafx.scene.shape.ArcTo** which is used to build the arcs in the interface with JavaFX. It has a source state, a destination state and lists of transition symbols. Transition symbols are stored twice: once as a list of Strings and one more time as a list of JavaFX Text objects. The reason for that is so that I won't have to make the conversion between them everytime I needed one or the other. The Text object was needed in order to display the transition symbols on the screen, and Strings were used because it's easier and faster to operate with them. For self transitions a single separate Text field is used.

3.2. Persistence

Persistence is done through import and export features, which save and load automata from system files.

3.2.1. Format

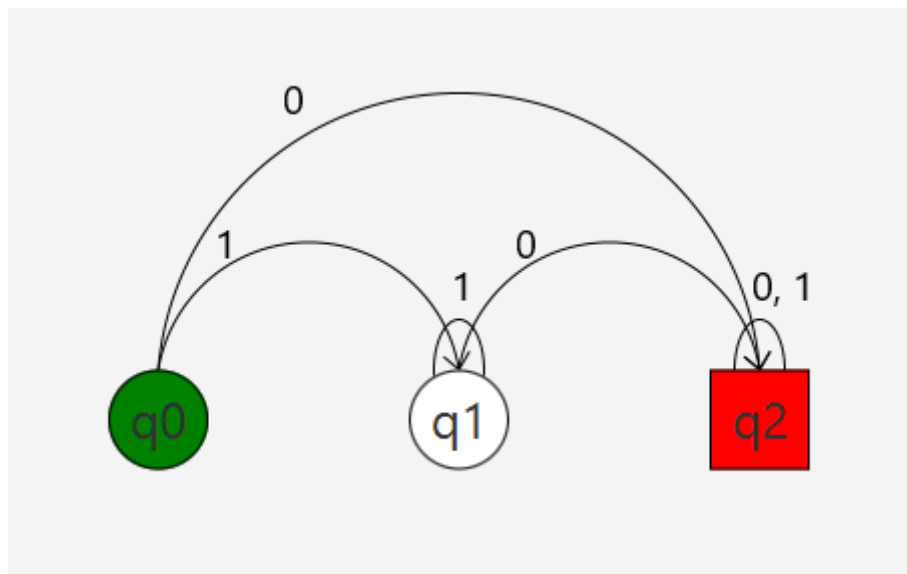
The format in which an automaton is simple and can be read and modified by any user. A persistence file can contain the following:

- **Possible_transitions**, followed by the alphabet symbols.
- **Initial_state**, followed by the initial state number. If multiple initial states are defined, only the last one will be considered an initial state.
- **Final_state** followed by final state number. Each final state must be declared separately.
- **State**, followed by state number specifies a regular state.
- **Transition** followed by two state numbers and a transition symbol. Transition symbol must be present in the alphabet, otherwise the transition won't be added.

Any line that doesn't start with one of the keywords mentioned above will be ignored.

Initial_state, **final_state** and **state** can also be followed by X and Y coordinates, which will be used if the user doesn't choose to position states automatically. X and Y coordinates are not mandatory.


```
possible_transitions 0 1
initial_state q0
state q1
final_state q2
transition q0 q1 1
transition q0 q2 0
transition q1 q1 1
transition q1 q2 0
transition q1 q1 1
//transition q0 q0 99
transition q2 q2 0
transition q2 q2 1
transition q2 q2 0
transition q2 q2 1
```



3.2.2. GUI

The save and load functionalities are triggered by two dedicated buttons which open a window with a textfield where the user inputs the path to the file. In case of loading, if the file does not exist, an error type Alert will show with the message “File does not exist”. In case of saving, the file will be created.

3.3. Building an automaton

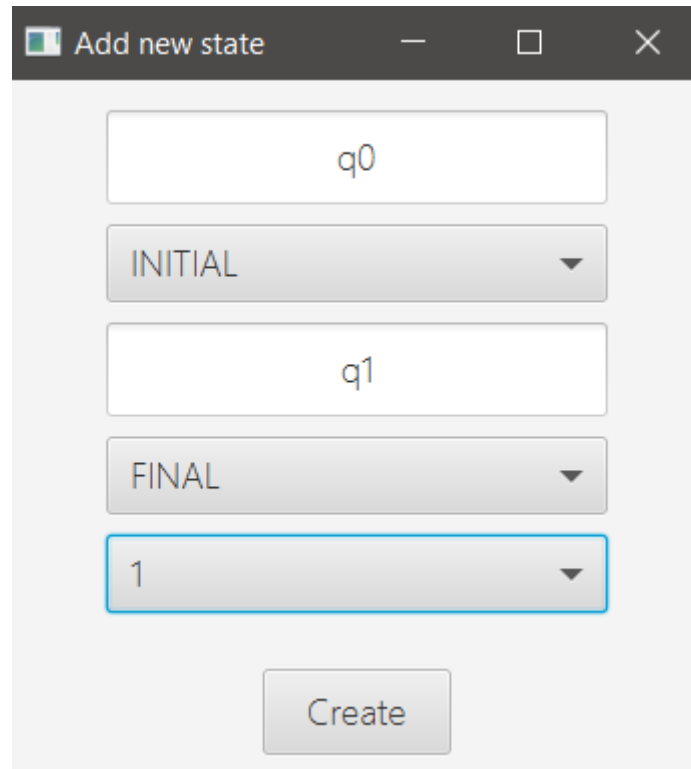
Apart from the load feature, an automaton can be built through the user interface.

3.3.1. Add symbol

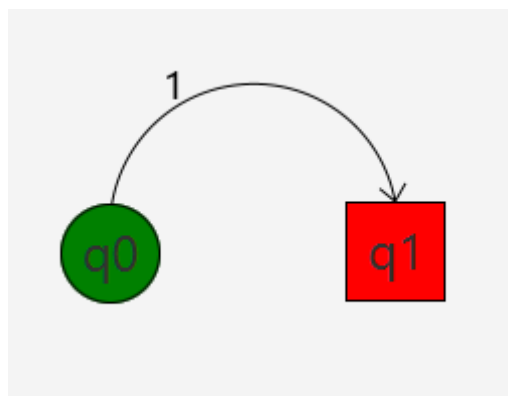
The first step is to add the alphabet symbols by using the “Add Symbol” button. After adding a symbol, it will be present in the dropdown menu when creating a transition.

3.3.2. Create transition

Then, the user presses the “Create Transition” button and a window pops up. The user has to set the names of the two states, the types and the transition symbol.



The screenshot shows a window titled "Add new state" with standard window controls (minimize, maximize, close). Inside the window, there are four input fields and one button. The first field contains "q0" and has a dropdown menu set to "INITIAL". The second field contains "q1" and has a dropdown menu set to "FINAL". The third field contains "1" and has a dropdown menu set to "1". The "1" dropdown is highlighted with a blue border. At the bottom of the window is a "Create" button.



3.3.3. Delete state

There is also the possibility to delete a state with the “Delete state” button. When this is pressed the state is removed and so are all the transitions related to that state.

3.3.4. Special cases

The user sets both states as initial states and states have different name. An error Alert with the message “Only one state can be initial state” is shown. However, if the states have the same name, it will be a self transition and the operation will be successful.

The user doesn’t set a name for one of the states. An error Alert with the message “Choose a name for both states” is shown.

The user doesn’t select a transition symbol. An error Alert with the message “Choose a transition symbol” is shown.

The state type is not chosen. In this case the **regular** state type will be chosen by default, since this is the type which will be chosen most often.

If states already exist, they won’t be created. Instead, only the transition symbol will be added.

If the transition already exists, the function will return immediately and nothing will happen.

3.4. Drawing an automaton

Elements are drawn in the GUI by adding them to the children of the Pane node.

Drawing is handled by **AutomatonDrawer**, which is called after an automaton is imported and everytime the “Create transition” and “Delete transition” buttons are pressed.

3.4.1. States

The first step is to clear the children of the Pane so that previous operations like importing an automaton or creating a transition won’t interfere with the automaton we’re trying to draw.

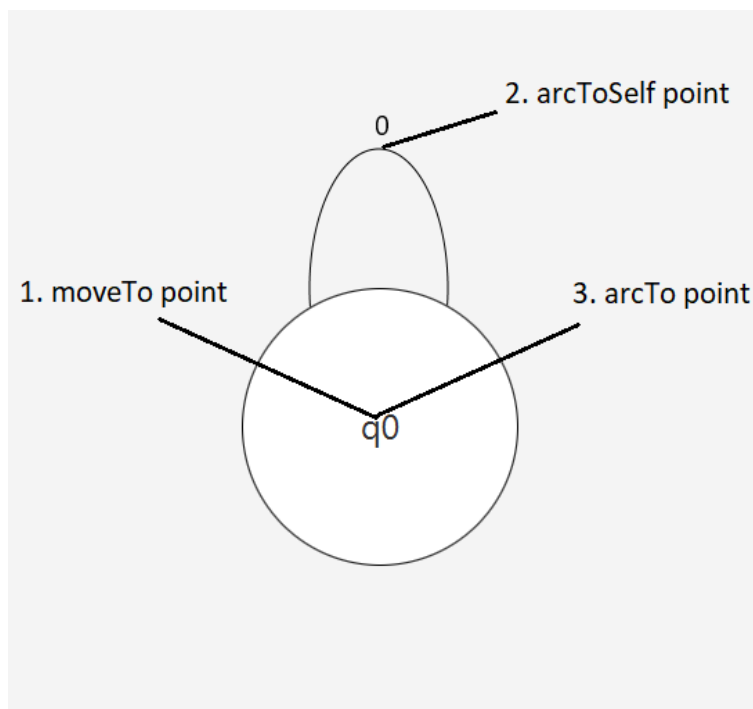
The next step is to set the color of the state, depending on it’s type: regular, initial or final. Position coordinates can be taken from the import file if they are available, or they can be set automatically so they align on a grid. The colors, grid size and space between states are customizable through a properties file.

3.4.2. Arcs

Arcs are created using **javafx.scene.shape.Path**. The starting point has to be **javafx.scene.shape.MoveTo** object. This is the reason why the State object has a MoveTo field, which contains the X and Y coordinates of the center of the state. Next elements added to the Path are points through which the path will go. I used **javafx.scene.shape.ArcTo** as points in order to make the path look like an arc. There are two cases:

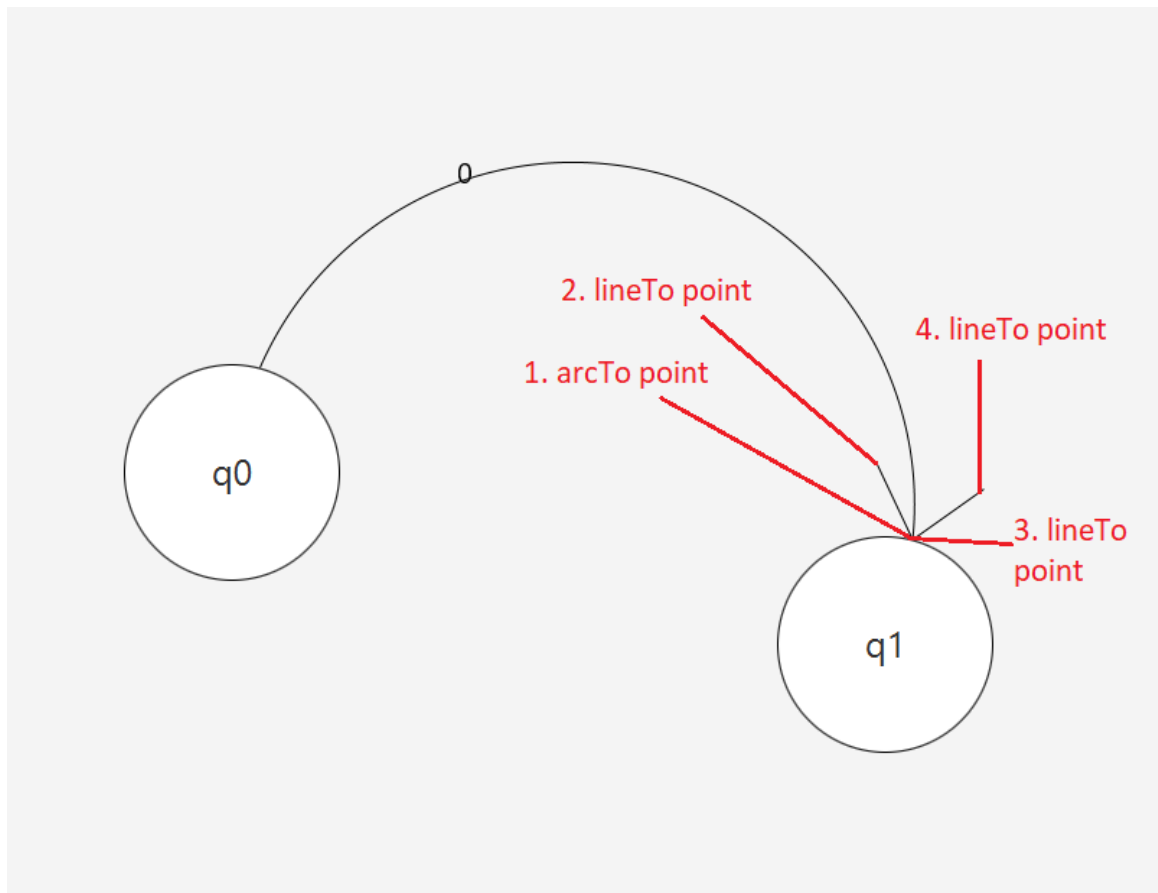
1. The arc goes to the state itself:

In this case the `arcToSelf` field is used to create a point just above the state. The path will start with the middle point of the state, the next point will be the `arcToSelf`, and the next one will be again the middle point of the state. The path is then sent to back so it won't be drawn over the state.

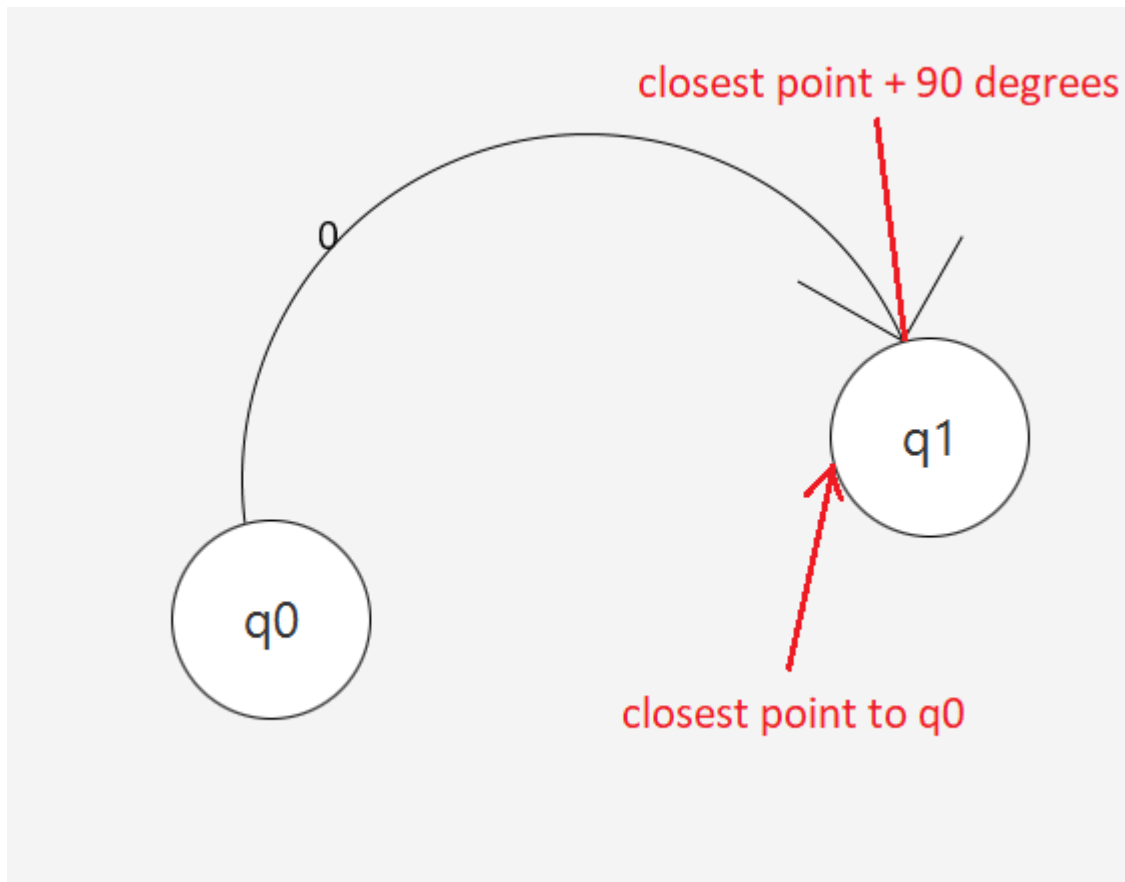


2. The arc goes into another state:

This part was a bit tricky because an arrow head had to be drawn at the end of the arc and there was no straightforward way to do it with JavaFX. What I ended up doing was to add three additional **javafx.scene.shape.LineTo** objects to the Path, so that I create an arrow head.



The tip of the arrow had to be placed exactly of the edge of the state, which means it had to be a radius distance away from the center. Finding the direction in which to move away from the center was difficult. In JavaFX points are represented as vectors, so I subtracted the vector of the middle point of one state with the vector of the middle point of the other and then I calculated the angle between the result and the horizontal unit vector with coordinates (0, 1). Using this angle I could find the point on the edge which is closest to the other state. For the X coordinate I multiplied the radius with cos of the angle and for Y i multiplied the radius with sin of the angle and I added them to the center of the state. Before that, I had to add 90 degrees to this angle so that it will be above the state.



Next, the arrowheads positions are set in a similar manner using the point we previously found. For one line of the arrow an angle is added and for the other one it is subtracted. Then I multiplied the sin and cos of the angles with the length of the arrowhead.

3.4.3. Symbols

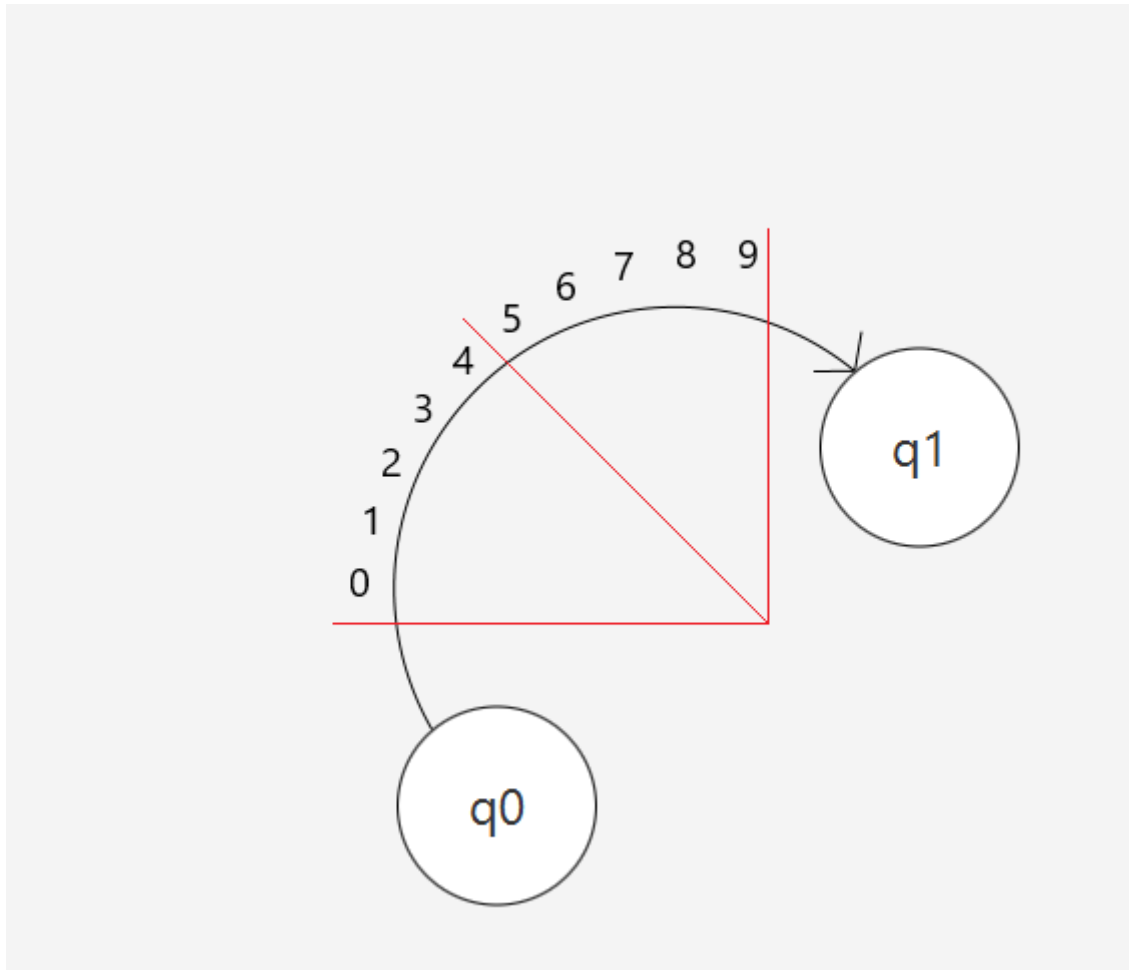
For drawing the symbols on an arc there are two cases:

1. When the arc goes into the same state

This case was simpler because I could just place the Text containing all the symbols above the state at a distance which depends on the position of the state's `arcToSelf`.

2. When the arc goes into other state

In this case I had to place the Text along the arc, so the Text had to be arched. I couldn't find a way to do this, so I split the Text which contained all the transition symbols into multiple Text object, each containing only one transition symbol. Then I had to place the Text objects along the arc. Knowing that the radius of an arc is the distance between the two states divided by 2, I could start placing the symbols in different directions, starting from the point in the middle of the two states, at a length equal to the radius.



In order to get the position of the first symbol I had to get the angle between the middle point of the two states and the horizontal axis. Then I added 90 degrees plus half of the angle along which the symbols span (this angle can be changed in the settings file), in order to place them symmetrically. I found the position of the first symbol by adding \cos of the angle multiplied by radius and \sin of the angle multiplied by radius to the coordinates of the middle point. For the rest of the symbols I did the same, while increasing the angle depending on the number of symbols and the angle along which the symbols span. This worked only if the first state had a higher Y coordinate. In order to fix this, the angle between the middle point of the two states and the horizontal axis had to be multiplied by -1 if the Y coordinate of the first state was lower.

3.4.4. Dragging

States can be dragged around the pane. When a state is dragged the arcs that go into the state and the arcs that come from that state have their positions updated. Also the symbols on the arcs have their positions updated. This method is way more efficient than drawing the whole automaton again whenever a state is dragged.

3.5. Parsing

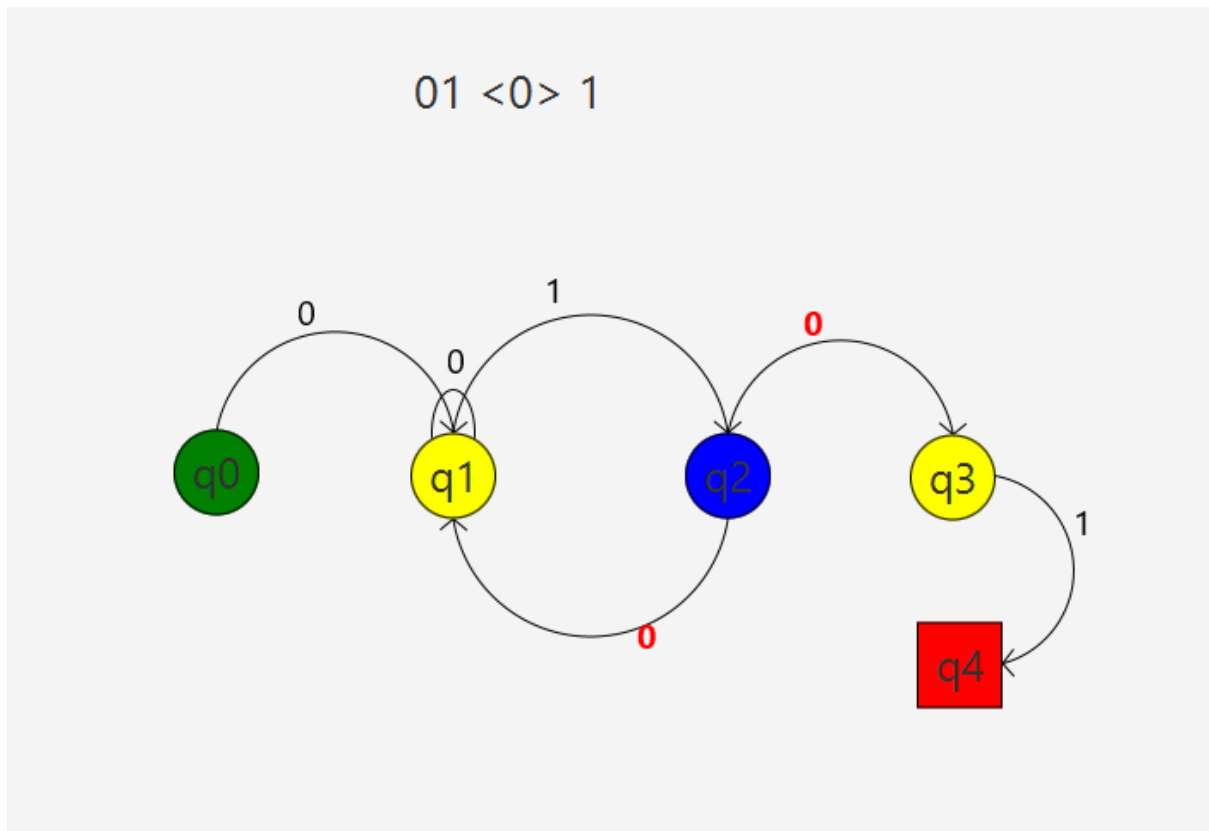
The automaton can parse a given word. Parsing starts when the user clicks “Start parsing” button. A window with a TextField appears and the user will give the input word.

3.5.1. Advancing

When the automaton is in parsing mode the states cannot be dragged, instead they can only be clicked. When the user clicks a state, the automaton advances to that state and goes to the next symbol in the word. This works only if there exists an arc with the current symbol to that state, otherwise nothing happens.

3.5.2. Highlighting

The current state will be highlighted with blue and all reachable states will be highlighted with yellow. The possible transition symbols will turn red and bold. When advancing to the next state, the current highlights will be removed.



3.5.3. Info label

In the upper part of the window there is a label which shows the word we are parsing and the current symbol.

3.5.4. Success

If the word is finished and we are in a final state, then we receive an Alert telling us that the word was successfully parsed.

3.5.5. Error

If the state is not a final state, we receive an error Alert saying that the current state is not final.

If there are no possible transitions, we will receive an error Alert specifying that there are no possible transitions from the current state with the current symbol.

When parsing ends all the highlights are removed.

3.6. Minimizing

Minimization of an automaton is the process of removing the state which don't affect the language it recognizes. This process makes sure that the automaton has the least number of states. Having a minimal automaton results in a reduced complexity, and a faster execution time.

3.6.1. Basic notions

Before diving into the algorithm, we first need to understand some concepts.

Reachable state is a state such that there exists a transition with a symbol from the alphabet to it from the current state. Two states are **separable** if there exists a sequence of transitions starting from those states, from one state we end only in final states and from the other we end only in non final states. Similarly, two states are **inseparable** if for any sequence of transitions we end up with both in final states or we end up with both in non final states. The separable relation is an equivalence relation. This means that if we have two states and we transition from both of them with the same symbol and the two new states are separable, then the previous two states are also separable.

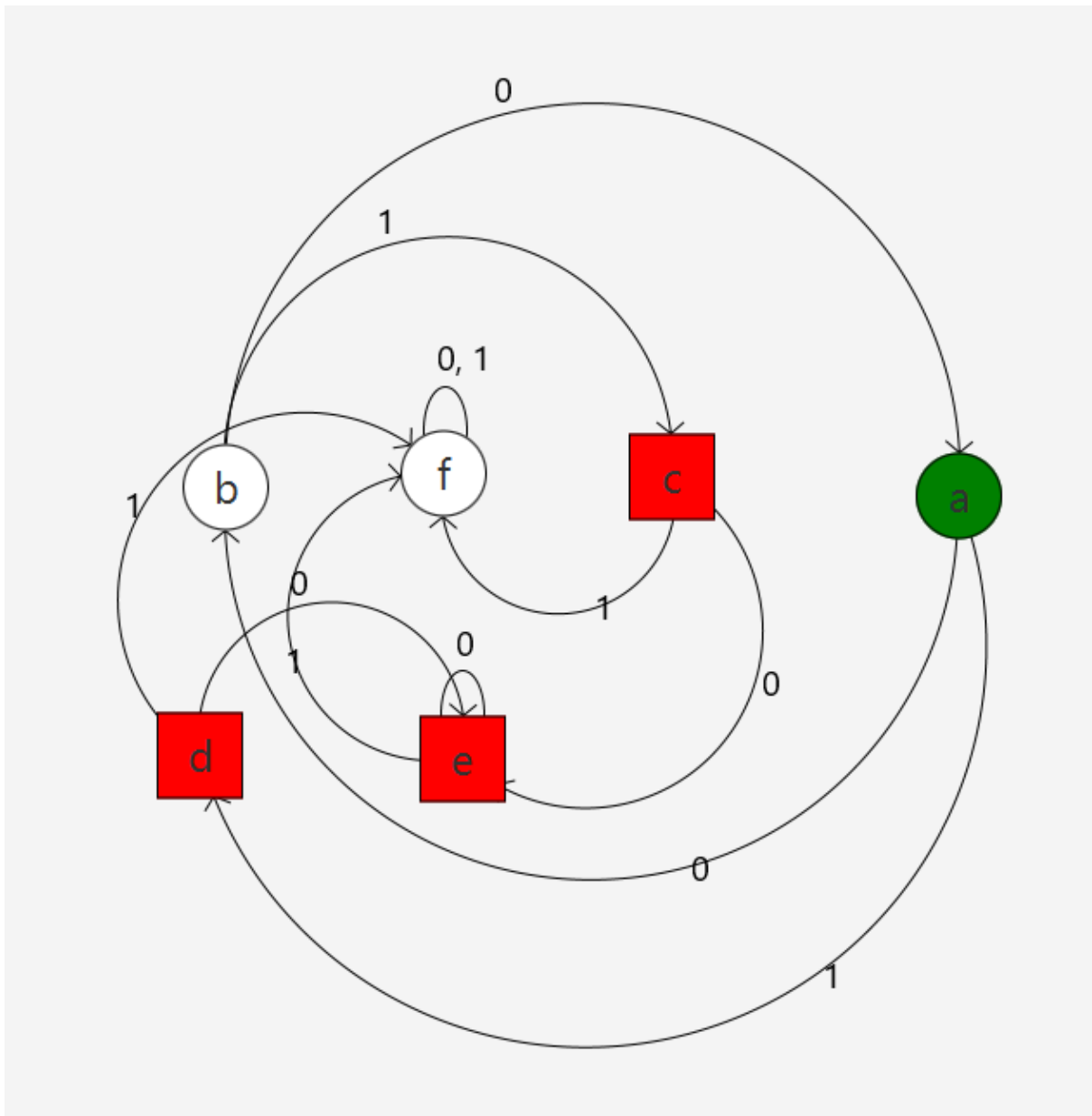
3.6.2. Data structures

In order to reduce the complexity of the implementation I assigned each state an index and I created two HashMaps for converting the state to the integer index and back. Then I created a matrix for tracking the separability of states. Since the fact that state i and state j are separable also means that state j and i are separable, the matrix will only be filled above the main diagonal. The elements on the main diagonal are 0, since a state is not separable from itself. The element at position $[i, j]$ is 0 if the states corresponding to indexes i and j are separable, 1 otherwise. I created a second matrix which has lists of inseparable states as elements. The element at position $[i, j]$ is the list of inseparable states such that there exists a transition from them with a symbol that leads to states i and j . I represented an element of this matrix as an ArrayList of Pairs of State objects.

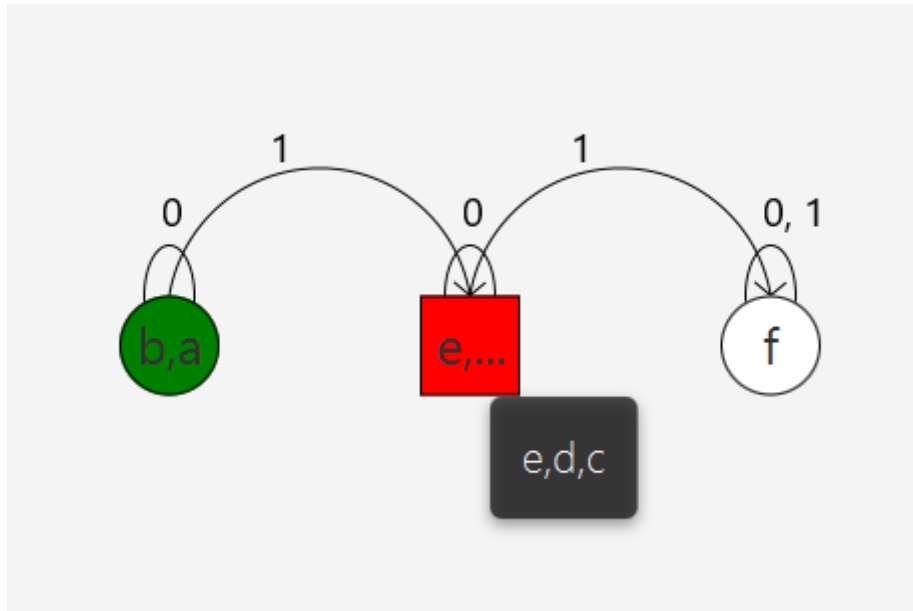
3.6.3. Algorithm

The algorithm starts by filling the separable matrix. The element at position $[i, j]$ is 0 if the state corresponding to index i is a final state and the state corresponding to index j is not final or vice versa. The element is 1 otherwise. The next step is to look for pairs of inseparable states. Then we go through every symbol of the alphabet and we see in which state we get from each of the two states. If the states we reach are separable, then we set the previous two states as separable. We also set the pairs of states which depend on those two as separable. The pairs will be present in the second matrix. In case we didn't find such symbol, we go again through every symbol in the alphabet and we find the states we reach by transitioning with the symbol. If the states are different and not both of those transitions were self transitions, then we add the initial states in the second matrix at position given by the indexes of the reached states.

3.6.4. Example



The minimal automaton is drawn on screen and a file named “minimized.txt” is automatically created and the automaton is exported there. The new names for the states are constructed from the merged states separated by comma. If too many states are merge into a single one, it’s name will be too long and won’t fit into the label. In order to solve this I added a tooltip showing the full name of the state when the user hovers the mouse over it.



3.7. Settings

The application comes with a Settings.properties file which can be used to modify multiple parameters without the need to recompile the project.

The list of customizable parameters:

- State height
- State width
- Initial state color
- Final state color
- Regular state color
- State border color
- State border radius
- State font size
- Grid row size
- Grid row padding
- Grid column padding
- Arrowhead length
- Arrowhead angle
- Current state color
- Possible state transition color
- Possible symbol transition color
- Transition font size

- Transition symbol color
- Angle along which the symbols span

4. Conclusions

I managed to create an application that is able to work with automata and I consider it to be an important and easy to use tool for learning how an automaton is working.

In the future I want to extend the application by making it available as a website so that it will be easier to access.

One thing that I would like to change is the way the arcs are drawn so that they don't overlap so much and the user won't have to find the right positioning in order to reduce overlapping. This would be difficult to accomplish given the actual implementation of the arcs.

Another thing would be the addition of functionalities, such as conversion from nondeterministic finite automaton to deterministic finite automaton, adding support for regular expressions, creating an automaton from a regular expression.

Bibliography

1. <http://automataeditor.sourceforge.net>
2. <https://github.com/kdickerson/automatonSimulator>
3. <http://automatonsimulator.com>
4. <http://madebyevan.com/fsm>