Department of Computer Science
Faculty of Science
Palacký University Olomouc

# MASTER THESIS

Two-Way Finite Automata

2023

Bc. Kristína Poláková

Supervisor:
doc. RNDr. Tomáš Masopust, Ph.D., DSc.

Study program: Computer Science,
Specialization: Artificial Intelligence

**Bibliografické údaje**

| | |
|---|---|
| Autor: | Bc. Kristína Poláková |
| Název práce: | Dvoucestné konečné automaty |
| Typ práce: | diplomová práce |
| Pracoviště: | Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci |
| Rok obhajoby: | 2023 |
| Studijní program: | Informatika, Specializace: Umělá inteligence |
| Vedoucí práce: | doc. RNDr. Tomáš Masopust, Ph.D., DSc. |
| Počet stran: | 69 |
| Přílohy: | elektronická data v úložišti katedry informatiky |
| Jazyk práce: | anglický |

**Bibliographic info**

| | |
|---|---|
| Author: | Bc. Kristína Poláková |
| Title: | Two-Way Finite Automata |
| Thesis type: | master thesis |
| Department: | Department of Computer Science, Faculty of Science, Palacký University Olomouc |
| Year of defense: | 2023 |
| Study program: | Computer Science, Specialization: Artificial Intelligence |
| Supervisor: | doc. RNDr. Tomáš Masopust, Ph.D., DSc. |
| Page count: | 69 |
| Supplements: | electronic data in the storage of department of computer science |
| Thesis language: | English |

## Anotace

*Hlavním cílem práce je prostudovat dvoucestné konečné automaty, jejich základní vlastnosti a operace, jako například sjednocení, průnik a další. V praktické části je vytvořena knihovna pro práci s těmito automaty, která je pak využita v ukázkové aplikaci vytvořené pro potřeby této práce.*

## Synopsis

*The main goal of the thesis is to study two-way finite automata, their properties and operations, such as union, intersection, etc. The practical part consists of a library to work with such automata and of a sample application created for the purpose of this work.*

**Klíčová slova:** dvoucestné konečné automaty; konečné automaty; regulární jazyk; operace

**Keywords:** two-way finite automata; finite automata; regular languages; operations

I would like to express my thanks to doc. RNDr. Tomáš Masopust, Ph.D., DSc. for supervision and feedback on this work.

# Contents

# List of Figures

# 1 Introduction

Finite automata (FA) is a well-known and commonly used term in computer science. A finite automaton is a computational model providing a simple and effective way to recognize whether a word belongs to a specific language or not.

Usually, when the term automata is mentioned it refers to a one-way finite automata (1FA), more precisely to a deterministic one-way finite automaton (1DFA), or a non-deterministic one-way finite automaton (1NFA). Both of these automata types can decide if some given word belongs to a regular language, and therefore we say that they recognize regular languages. As they both accept the same type of languages, it is clear that they are equivalent to each other, which can be proven by converting 1NFA to 1DFA and 1DFA to 1NFA. An exponential-time algorithm is known for converting 1NFA to 1DFA. No algorithm is needed to convert 1DFA to 1NFA as 1DFA is just a special case of 1NFA.

Nevertheless, another type of automata that accept regular languages is two-way finite automata (2FA), whose concept is not that widespread in computer science. They can also be deterministic (2DFA) or non-deterministic (2NFA). The difference is that 2FA can read a word in both directions while 1FA can only read a word from left to right.

All of the four types mentioned above are equivalent, interchangeable, and have the same computational power. One type can be converted to any of the other types, although no polynomial-time algorithm exists for the conversion of 2NFA to 2DFA and 1NFA to 2DFA.

This thesis deals with the concept of 2FA. The differences between 1FA and 2FA are shown and a proof of their equivalence is provided.

The main part of the thesis covers basic operations on 2FAs - union, intersection, concatenation, square, complementation, Kleene star - which are then implemented in a library using C# and the .NET platform.

Last, but not least, to demonstrate the computational process of 2FA and the usage of the library, a sample application is created.

A reader is required to have some knowledge about 1FA, though some pieces of information about them are provided.

# 2 Finite Automata

This section covers the finite automata theory concerning regular languages, regular grammars, and automata themselves. To be able to compare 2FA and 1FA, some knowledge about them is a necessity. The theory of regular languages, regular grammars, and 1FA is briefer than that of 2FA, as we assume that the reader already has some basic knowledge of them.

The similarities and differences between those types are discussed and the equivalence of them is shown by providing an algorithm for conversion. For each type, an example automaton is attached for better understanding. The example automata are visualized by oriented graphs as we suppose the reader is familiar with this construction.

## 2.1 Regular Languages

All types of automata discussed in this thesis, FA and 2FA, accept only regular languages. This section explains the basics of regular languages and their grammars.

Any regular language is generated by some grammar of type 3 (also called regular or linear) when referred to Chomsky's Hierarchy of Languages [10]. As we can see in Figure 1, type 3 languages are the most restricted ones. In the terminology of grammars, it means that the production rules are very strict.

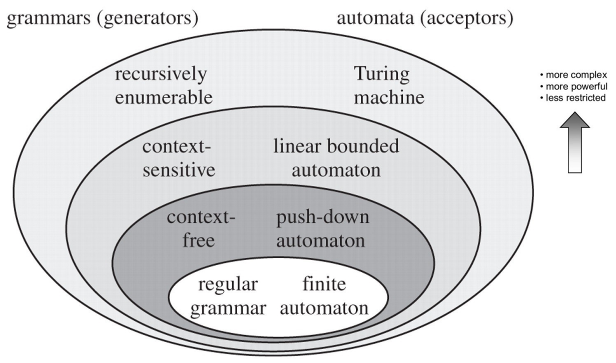


Figure 1: Chomsky Hierarchy [10]

To remind the reader what a grammar is in the concept of formal languages the formal definition follows [12].

**Definition 1**

A grammar is a structure $G = (\Sigma, N, S, P)$, where

- $\Sigma$ is a set of terminal symbols, $\Sigma \cap N = \emptyset$ (also called alphabet)
- $N$ is a set of non-terminal symbols (also called non-terminals or variables)

- $S$ is an initial non-terminal, $S \in N$,
- $P$ is a set of production rules in any of the following forms
  1. $A \rightarrow xB$, where $A, B \in N$ and $x \in \Sigma^*$
  2. $A \rightarrow x$, where $A \in N$ and $x \in \Sigma^*$
  3. $A \rightarrow \epsilon$, where $A \in N$ and $\epsilon$ represents nothing (the empty word of length 0)

The restriction of the rules lies in the fact that at each step exactly one non-terminal symbol is changed and at most one non-terminal symbol can be produced. In other types of grammars, it is possible to manipulate more than one non-terminal at one step.

We can differ between two types of type-3 grammars based on the direction of the production

- Right Linear Grammars - uses rules only of the forms
  1. $A \rightarrow xB$, where $A, B \in N$ and $x \in \Sigma^*$
  2. $A \rightarrow x$, where $A \in N$ and $x \in \Sigma^*$
- Left Linear Grammars - uses rulers only of the forms
  1. $A \rightarrow Bx$, where $A, B \in N$ and $x \in \Sigma^*$
  2. $A \rightarrow x$, where $A \in N$ and $x \in \Sigma^*$

The production process of some regular language $L$ (as of Right Linear Grammar) starts with the initial non-terminal $S$.
One step of the production: $\alpha A \dashv \alpha y B$, where $A, B \in N$ and $\alpha, y \in \Sigma^*$ and $\dashv$ tells us that there exists a production rule $A \rightarrow yB$.
The language that is produced by the grammar consists of all words $w$ that can be reached by the production process from the initial non-terminal: $S \dashv^* w$, where $S \in N$ (the initial non-terminal), $w \in \Sigma^*$ and $\dashv^*$ is a reflexive and transitive closure of $\dashv$ meaning there is a sequence of the production rules by which we can achieve $w$ in a finite number of steps.

Therefore $L(G) = \{w \in \Sigma^* \mid S \dashv^* w\}$.

From the process of production described above we can easily imagine the production process of the Left Linear Grammars. As we will not work with this type of grammars and this section should be only a brief summary of regular languages, we will not cover this topic. The topic that is also not covered in this thesis is regular expressions which also represent regular languages.

Example 2

Let $G = (N, \Sigma, P, S)$ be the grammar of the type 3 where

- $\Sigma = \{0, 1\}$
- $N = \{S, A\}$
- $S \in N$,
- $P$ is a set of production rules
  1. $S \rightarrow 0A$
  2. $A \rightarrow 1A \mid \epsilon$

The language of the grammar: $L(G) = \{0, 01, 011, 0111, 01111, ...\}$ or $01^*$ written as a regular expression.

## 2.2 Basics of Finite Automata

All types of FA have the same purpose which is to determine whether some given word belongs to a certain regular language. Therefore, a finite automaton can be defined as a computational model or mechanism that takes a word as an input and outputs true (accepts the word) or false (rejects the word) based on whether it belongs to the language that is recognized by the automaton.

Each automaton has an alphabet $\Sigma$, a set of states $Q$, a starting state $q_0$, and a transition function $\delta$.

An automaton performs its task by moving through its states according to its transition function and accepts if it reads the whole word and reaches some accepting state. The input word is written to an automaton's memory, the so-called tape. The tape is read-only after the input is written to it, and we cannot reach outside of the memory where the word is written (bounded memory).

To define a language of an automaton or its computational process we have to define a term *configuration*.

**Definition 3**

A configuration $c$ is a sequence $c = w_1 q w_2$, where $w_1, w_2 \in \Sigma^*$ are words, $w = w_1 w_2$ is an input word and $q \in Q$ is some state, then the position of $q$ defines what part of the word was already read ($w_1$) and what part is to be read ($w_2$). At each step of the computational process, the configuration provides the overall state of the automaton.

The position of $q$ defines that the next character that will be read is the first character of $w_2$ as we read at most one character at each step. We can also say that the *reading head* or simply *the head* points to the character to be read.

For example, if an automaton $A$ is in a configuration $0001q_20111$ for the input alphabet $\Sigma = \{0, 1\}$ then the automaton is in a state $q_2$ considering it belongs to a set of its possible states $Q$; the tape contains characters 00010111; the head points to the place on a tape where the 0 after $q_2$ is present (0001**0**111).

Configurations are used to trace steps that an automaton makes by performing transitions defined by $\delta$.

Let us have two configurations $c_1$ and $c_2$, then one step (one transition is performed) from configuration $c_1$ to $c_2$ is denoted as $c_1 \vdash c_2$. If we can get from $c_1$ to $c_2$ in more than one step (performing more transitions), it is denoted as $c_1 \vdash^* c_2$.

Finally, we can define the language of an automaton: $L(A) = \{w\Sigma^* \mid q_0 w \vdash^* wq_f$ where $q_f$ is a final (accepting) state $\}$.

The main difference between 1FA and 2FA lies in the transition function, which is a reason why it is not defined in detail here but separately for each type.

The multi-direction transitions allow for 2FA to have fewer states than 1FA as you can see from the examples in the next sections.

## 2.3 One-Way Finite Automata

In this section, we will look at the basics of one-way finite automata which can be also referred to as just finite automata in comparison to two-way finite automata.

One-way finite automata (1FA) is a computational model that decides whether a given input word belongs to a certain language. As already mentioned, the language accepted by the automaton is a regular language covered in Section 2.1. One-way finite automata can be divided into deterministic (1DFA) and non-deterministic (1NFA). Both of those types have the same computational power. Although 1NFA can be constructed with much less space, 1DFA is much easier to work with.

### 2.3.1 Deterministic One-Way Finite Automata

Determinism in automata theory means that for every input word of a 1DFA $A$, the automaton outputs the same value. As for the transition function, only one transition at a time is allowed for every alphabet symbol and state.

**Definition 4**

Let $A$ be a 1DFA such that $A = (\Sigma, Q, q_0, F, \delta)$ where

- $\Sigma$ is an input alphabet, a set of symbols of which the input words can consist of
- $Q$ is a set of states

- $q_0 \in Q$ is an initial state
- $F \subset Q$ is a set of final states
- $\delta : Q \times \Sigma \to Q$ is a transition function

Input word $w \in \Sigma^*$ is a word given to the automaton to determine whether it belongs to the language $L(A)$. It is written to the automaton memory.

The accepting computation of 1DFA denoted by configurations is $q_0 w \vdash^* w_1 q_f w_2$ where $q_0$ is the initial state and $q_f \in F$ is a final state of the automaton. To learn more about configurations see Section 2.2.

Therefore, the definition of the language of A is
$L(A) = \{w \in \Sigma^* \mid q_0 w \vdash^* w q_f, q_f \in F\}$.

In Figure 2 we can see an example of a 1DFA recognizing the language $L = \{0, 01, 011, 0111, 01111, ...\}$ or $01^*$ written as a regular expression.

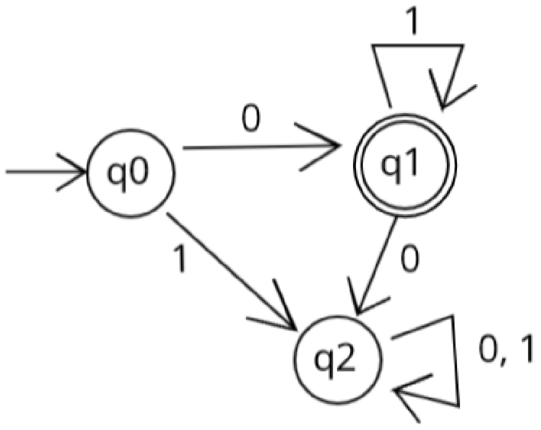


Figure 2: 1DFA recognizing the language $01^*$

### 2.3.2 Nondeterministic One-Way Finite Automata

Nondeterminism in automata theory means that for every input word of some 1NFA $A$, the automaton might not output the same value. As for the transition function, more than one transition at a time is allowed for every symbol and state.

**Definition 5**
Let $A$ be a 1NFA such that $A = (\Sigma, Q, q_0, F, \delta)$ where

- $\Sigma$ is an input alphabet
- $Q$ is a set of states
- $q_0 \in Q$, is an initial state
- $F \subset Q$ is a set of final states
- $\delta : Q \times \Gamma \to 2^Q$ is a transition function, where $2^Q$ denotes the power set of $Q$

From the definition above we can see that the only change to 1DFA is in the transition function. We can perform multiple transitions under any symbol and any state.

According to the change in $\delta$ the other change has to be in the acceptance condition. At every step of a computation, the automaton has to choose a transition to use. In theory, we suspect that it always chooses correctly but in the real implementation it is not possible to simply give a computer the command "choose the correct transition". The automaton has to test all possible transitions. Therefore, we can say that the computational process creates a new branch of computation for each possible transition.

The computational process 1NFA can be perceived as a tree with nodes that represent the configurations of the computation on some input word $w$:

- $q_0w$ is the root of the tree
- each leaf node is either
  - $wq_f$, where $q_f \in F$ - accepting configuration
  - $wq$, where $q \notin F$ - non-accepting configuration
- each non-leaf node is the configuration $(w_1qw_2)$ where $q \in Q$ and $w_1w_2 = w$

The edges of the computational tree are only one-way, and therefore each node has only one in-going edge but can have multiple out-going edges.

If, for some input word $w \in \Sigma^*$, the current node of computation is $w_1 q c w_2$, where $q \in Q, c \in \Sigma$ and $w_1 c w_2 = w$, then for each transition there exists an edge to a corresponding configuration.

The automaton accepts if there is a path to at least one leaf node with an accepting configuration from the root. Denoted by configurations: $q_0 w \dashv^* w q_f$ where $q_f \in F$.

The automaton rejects if all reachable leaf nodes contain rejecting configurations. Denoted by configurations: $q_0 w \vdash^* wq$ where $q \notin F$.

The language of $A$ $L(A) = \{w \in \Sigma^* \mid q_0 w \dashv^* w q_f\}$, is defined the same way as for 1DFA.

It can be easily seen that 1DFA is just a special case of 1NFA where the computational tree is linear.

In Figure 3 we can see an example of 1NFA recognizing the language $L$ defined by a regular expression (0|1)*01*.

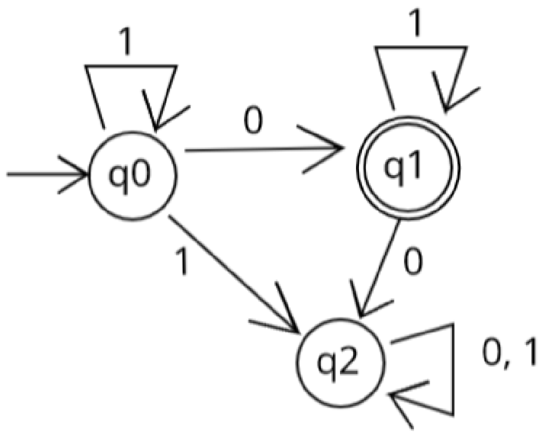


Figure 3: 1NFA recognizing language 01*

## 2.4 Two-Way Finite Automata

A two-way finite automaton (2FA) is a computational model that decides whether a given input word belongs to a certain language. As we already mentioned, the type of languages accepted by these automata are regular languages (more in Section 2.1) just like that accepted by 1FA.

2FA can be viewed as a special type of 1FA where the tape head can move in both directions and can therefore read the characters of an input word more times. Similarly, 2FA can be defined as a Turing Machine that has restricted memory (referred to as the tape) from both sides.

In both cases, the tape is read-only and only an input word can be written on the tape. Afterwards its length or content does not change and only the part of the tape where the word is present can be reached. To define the start and the end of the accessible tape, the symbols called left and right endmarkers are used.

We can also say that the reverse is true: 1FA is a special type of 2FA where the input word is read in only the right direction (no left transitions or stationary can be performed).

There are two possible formal definitions of 2FA:

1. Kozen's [5] [6] - uses left and right tape endmarkers and allows the tape head to move left or right, no stationary transitions are allowed

2. Shepherdson's [5] - more similar to FA definition, no tape endmarkers are used, and stationary transitions are allowed

2FAs can also be deterministic (2DFA) or nondeterministic (2NFA) while each of the definitions above can be used for both types.

### 2.4.1 Deterministic Two-Way Finite Automata

Determinism for two-way deterministic finite automata (2DFA) has the same meaning as for 1DFA. Only one transition per symbol is possible in each state therefore we get the same output for every input. For 2DFA this also means that the head moves only in one direction at a time.

#### 2.4.1.1 Kozen's 2DFA

Kozen's definition of 2DFA [5] [6] uses the left and right endmarkers in transitions which might provide the reader with a good inside into how the automaton works and how its head moves. This definition differs from that of the 1FA as it has a special accepting and rejecting state, and the initial state cannot be final.

**Definition 6**

Let $A = (\Sigma, Q, \triangleright, \triangleleft, q_0, q_a, q_r, \delta)$ be a deterministic two-way finite automaton where

- $\Sigma$ is an input alphabet

- $\triangleright$ is the symbol marking the beginning of the accessible tape, also called left endmarker

- $\lhd$, the symbol marking the end of the accessible tape, also called right endmarker

- $Q$ is a set of states

- $q_0 \in Q$ is an initial state

- $q_a \in Q$ is an accepting state

- $q_r \in Q$ is a rejecting state

- $\delta : Q \times \Sigma \cup \{\rhd, \lhd\} \to Q \times \{L, R\}$ is a transition function where $\{L, R\}$ is a set of directions for the head movement ($L$ for left, $R$ for right), therefore, the head moves one character to the left or one character to the right after each transition

For each state $q \in Q$ there should exist transitions

- $\delta(q, \rhd) = (p, R)$ where $p \in Q$ - ensures that the head does not reach outside of the accessible tape on the left

- $\delta(q, \lhd) = (p, L)$ where $p \in Q$ - ensures that the head does not reach outside of the accessible tape on the right

For each symbol $c \in \Sigma$ there should exist transitions

- $\delta(q_r, c) = (q_r, R)$ - ensures that we stay in the rejecting state $q_r$

- $\delta(q_a, c) = (q_a, R)$ - ensures that we stay in the accepting state $q_a$

Input word $w \in \Sigma^*$ is given to the automaton to determine whether it belongs to the language $L(A)$.

Let $c$ be a configuration such that $c = \rhd w_1 q w_2 \lhd$ where $w_1, w_2 \in \Sigma^*$ and $q \in Q$. To learn more about configuration see Section 2.2 as the configuration theory is the same for all types of automata.

A step from one configuration to the next configuration if a right transition is applied: $\rhd w_1 x q y w_2 \lhd \vdash \rhd w_1 x y p w_2 \lhd$ where $q, p \in Q, w = w_1 x y w_2, w_1, w_2 \in \Sigma^*$ and $x, y \in \Sigma$. The transition: $\delta(q, y) = (p, R)$.

A step from one configuration to the next configuration if a left transition is applied: $\rhd w_1 x q y w_2 \lhd \vdash \rhd w_1 p x y w_2 \lhd$ where $q, p \in Q, w = w_1 x y w_2, w_1, w_2 \in \Sigma^*$ and $x, y \in \Sigma$. The transition: $\delta(q, y) = (p, L)$.

The complete accepting computation of the automaton denoted by configurations: $\rhd q_0 w \lhd \vdash^* \rhd w_1 q_a w_2 \lhd$, where $q_0$ is the initial state and $q_a$ is the accepting state of the automaton.

The complete rejecting computation of the automaton in the configuration: $\triangleright q_0 w \triangleleft \vdash^* \triangleright w_1 q_r w_2 \triangleleft$. The automaton also rejects if it enters an infinite loop which is possible as it can move in both directions.

Therefore, the definition of the automaton language: $L(A) = \{w \in \Sigma^* \mid \triangleright q_0 w \triangleleft \vdash^* \triangleright w_1 q_a w_2 \triangleleft\}$.

It is clear from the accepting configuration $w_1 q_a w_2$ that there is no need to read the word to the end. The reason is that the accepting state cannot be left after a transition brings us to it. The same goes for the rejecting state. In the formal definition, we added the transitions to read the word to the end in the accepting and rejecting state to keep consistency with 1FA.

In Figure 4 we can see an example of 2DFA recognizing the language $L = \{0, 01, 011, 0111, 01111, ...\}$ or $01^*$ written as a regular expression. Please note that the right endmarker is denoted as $<$.

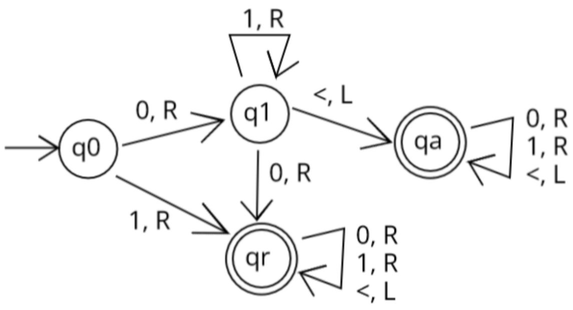


Figure 4: Kozen's two-way finite automaton accepting language $01^*$

#### 2.4.1.2 Shepherdson's 2DFA

Shepherdson's definition of 2FA [5] is more similar to that of 1FA as it does not use endmarkers or any special rejecting and accepting states.

**Definition 7**

Let $A = (\Sigma, Q, q_0, F, \delta)$ be a deterministic two-way finite automaton (2DFA) where

- $\Sigma$ is an input alphabet
- $Q$ is a set of states
- $q_0 \in Q$ is an initial state

- $F \subset Q$ is a set of final states
- $\delta : Q \times \Sigma \to Q \times \{L, S, R\}$ is a transition function where $\{L, S, R\}$ is a set of directions for the head movement ($L$ for left, $S$ for stay, $R$ for right) therefore, the head moves one character to the left, stay at the same position, or moves one character to the right after each transition

Input word $w \in \Sigma^*$ is given to the automaton to determine whether it belongs to the language $L(A)$.

Configuration $c = w_1 q w_2$ where $w_1, w_2 \in \Sigma^*$ and $q \in Q$. To learn more about configuration see Section 2.2 as the configuration is the same for all types of automata.

A step from one configuration to the next configuration if a right transition is applied: $w_1 x q y w_2 \vdash w_1 x y p w_2$ where $q, p \in Q, w = w_1 x y w_2, w_1, w_2 \in \Sigma^*$ and $x, y \in \Sigma$. The transition: $\delta(q, y) = (p, R)$.

A step from one configuration to the next configuration if a stationary transition is applied: $w_1 x q y w_2 \vdash w_1 x p y w_2$ where $q, p \in Q, w = w_1 x y w_2, w_1, w_2 \in \Sigma^*$ and $x, y \in \Sigma$. The transition: $\delta(q, y) = (p, S)$. If $q = p$ this transition results in a loop therefore, the automaton rejects the word.

A step from one configuration to the next configuration if a left transition is applied: $w_1 x q y w_2 \vdash w_1 p x y w_2$ where $q, p \in Q, w = w_1 x y w_2, w_1, w_2 \in \Sigma^*$ and $x, y \in \Sigma$. The transition: $\delta(q, y) = (p, L)$.

The complete accepting computation of the automaton denoted by configurations: $q_0 w \vdash^* w q_f$, where $q_0$ is the initial state and $q_f \in F$.

The complete rejecting computation of the automaton in the configuration: $q_0 w \dashv^* w q_r$ where $q_0$ is the initial state and $q_r \notin F$. The automaton is also rejecting if it enters an infinite loop which is possible as it can move in both directions or perform stationary transitions. It also rejects if it reaches outside of the tape as we do not have the endmarkers in transitions to avoid such behavior.

Therefore, the definition of the automaton language: $L(A) = \{w \in \Sigma^* \mid q_0 w \dashv^* w q_f\}$.

In Figure 5 we can see the example of 2DFA recognizing the language $L = \{0, 01, 011, 0111, 01111, ...\}$ or $01^*$ written as a regular expression.

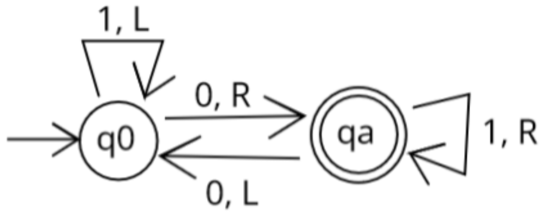


Figure 5: Shepherdson's two-way finite automaton accepting language 01*

### 2.4.2 Nondeterministic Two-Way Finite Automata

Non-determinism for two-way finite automata (2NFA) has the same meaning as for 1NFA. For any state and any symbol, there can be more transitions in any direction.

#### 2.4.2.1 Kozen's 2NFA

Kozen's definition of 2NFA [5] [6] also uses the left and right endmarkers in transitions like its 2DFA version.

**Definition 8**

Let $A = (\Sigma, Q, \triangleright, \triangleleft, q_0, q_a, q_r, \delta)$ be a 2NFA where

- $\Sigma$ is an input alphabet
- $\triangleright$ is the left endmarker
- $\triangleleft$ is the right endmarker
- $Q$ is a set of states
- $q_0 \in Q$ is an initial state
- $q_a \in Q$ is an accepting state
- $q_r \in Q$ is a rejecting state
- $\delta : Q \times \Sigma \cup \{\triangleright, \triangleleft\} \to 2^{Q \times \{L,R\}}$ is a transition function, where $\{L, R\}$ is a set of directions for the head movement ($L$ for left, $R$ for right) and $2^{Q \times \{L,R\}}$ is the power set of all possible states and directions, therefore, the head can move in multiple directions and to multiple states in any state under any symbol

The following transitions are the same as for 2DFA.

For each state $q \in Q$ there should exist transitions

- $\delta(q, \triangleright) = (p, R)$ where $p \in Q$ - ensures that the head does not reach outside of the accessible tape on the left
- $\delta(q, \triangleleft) = (p, L)$ where $p \in Q$ - ensures that the head does not reach outside of the accessible tape on the right

For each symbol $c \in \Sigma$ there should exist transitions

- $\delta(q_r, c) = (q_r, R)$ - ensures that we stay in the rejecting state $q_r$
- $\delta(q_a, c) = (q_a, R)$ - ensures that we stay in the accepting state $q_a$

From the definition above we can see that the only change to 2DFA is in the transition function $\delta$. We can reach multiple configurations under any symbol and any state.

According to the change in $\delta$ there also has to be a change in the acceptance criterion.

The computational process of 2NFA can be perceived as a graph with nodes that are represented as configurations and the edges represent transitions to all other possible configurations. The difference between 1NFA and 2NFA is that there can be a loop in the computational graph caused by the two-way movement of the head. Alternatively, there can be a computational branch that never halts.

The computational graph for 2NFA:

- $\triangleright q_0 w \triangleleft$ is the root
- each leaf node is either
  - $\triangleright w_1 q_a w_2 \triangleleft$ where $q_a$ is the accepting state and $w_1 w_2 = w$
  - $\triangleright w_1 q_r w_2 \triangleleft$ where $q_r$ is the rejecting state and $w_1 w_2 = w$
- each non-leaf node is a configuration $\triangleright w_1 q w_2 \triangleleft$ where $q \in Q - \{q_0, q_a, q_r\}$ and $w_1 w_2 = w$

Each node can have multiple in-going and out-going edges. If it was a 2DFA, the computational tree would have nodes that can have multiple in-going edges but only one out-going edge.

If for some input word $w$, the current node of the computation is $\triangleright w_1 q c w_2 \triangleleft$ where $q \in Q - \{q_0, q_a, q_r\}$ and $w_1 c w_2 = w$, then for each transition in any direction there exists an edge to the corresponding configuration.

Nodes according to the type of transitions are

- right transition $\delta(q, c) = (p, R)$ there is an edge to the configuration $w_1 c p w_2$
- zero transition - $\delta(q, c) = (p, S)$ there is an edge to the configuration $w_1 p c w_2$

- left transition - $\delta(q, c) = (p, L)$ there is an edge to the configuration $w_1 pcw_2$

The automaton accepts if there is at least one leaf node with an accepting configuration.

The automaton rejects if all branches contain either a leaf node with rejecting configurations or a loop.

Therefore, the definition of the automaton language: $L(A) = \{w \in \Sigma^* \mid \triangleright q_0 w \triangleleft \dashv^* \triangleright w_1 q_a w_2 \triangleleft\}$.

#### 2.4.2.2 Shepherdson's 2NFA

Shepherdson's definition of 2NFA [5] is similar to Kozen's 2NFA in terms of non-determinism but it still does not use endmarkers or any special rejecting or accepting states.

**Definition 9**

Let $A = (\Sigma, Q, q_0, F, \delta)$ be a 2NFA where

- $\Sigma$ is an input alphabet
- $Q$ is a set of states
- $q_0 \in Q$ is an initial state
- $F \subset Q$ is a set of final states
- $\delta : Q \times \Sigma \to 2^{Q\times\{L,S,R\}}$ is a transition function where $\{L, R\}$ is a set of directions for the head movement ($L$ for left, $S$ for stay, $R$ for right), therefore the head moves one character to the left, stays at the same position, or moves one character to the right after each transition, and $2^{Q\times\{L,S,R\}}$ is the power set of all states and directions

From the definition above we can see that again the only change to 2DFA is in the transition function $\delta$. We can reach multiple configurations under any symbol and any state.

The computational process of 2NFA can be perceived as a graph with nodes that are represented as configurations and the edges represent transitions to all other possible configurations. The difference from Kozen's 2NFA is that apart of the loop there can also be a step outside of the tape.

For the purpose of the graph, let's consider the configuration with endmarkers. This does not change anything we have already learned about this version of 2DFA as the tape is still bounded.

The computational tree for 2NFA:

- $\triangleright q_0 w \triangleleft$ is the root
- each leaf node is either
  - $\triangleright w q_f \triangleleft$ where $q_f \in F$ and $w_1 w_2 = w$ - the accepting configuration
  - $\triangleright w q \triangleleft$ where $q \notin F$ and $w_1 w_2 = w$ - the rejecting configuration
  - $q \triangleright w \triangleleft$ where $q \notin F$, $w_1 w_2 = w$ - the rejecting configuration where the head stepped outside the tape on the left side
  - $\triangleright w \triangleleft q$ where $q \notin F$, $w_1 w_2 = w$ - the rejecting configuration where the head stepped outside the tape on the left side
- each non-leaf node $= \triangleright w_1 q w_2 \triangleleft$ where $q \in Q$ and $w_1 w_2 = w$

The edges of the computational can also form a loop. The rest of the tree construction is the same as for Kozen's in Definition 8.

The automaton accepts if there is at least one leaf node with an accepting configuration.

The automaton rejects if all branches contain either a leaf node with rejecting configurations or a loop.

Therefore, the definition of the automaton language: $L(A) = \{w \in \Sigma^* \mid q_0 w \dashv^* w q_f\}$.

# 3 Equivalence of One-Way and Two-Way Finite Automata

As already mentioned, 2FA and 1FA both accept regular languages, therefore, they are equivalent. The equivalence can be shown by converting 1FA to 2FA and 2FA to 1FA such that they still accept the same language.

## 3.1 1DFA to 2DFA

The conversion of 1DFA to 2DFA is quite simple. 1DFA has only transitions that move the head to the right and therefore can be considered 2DFA that never moves to the left.

**Theorem 10**
*Let $A = (\Sigma, Q, q_0, F, \delta)$ be a 1DFA. There is a 2DFA $A' = (\Sigma, Q, q_0, F, \delta')$ that accepts the same language ($L(A') = L(A)$).*

*Proof*
$\delta'$ is a transition function created from the transition function $\delta$ such that if there is a transition $\delta(q, c) = p$ then the equivalent transition in $\delta'$ would be $\delta'(q, c) = (p, R)$.

□

This conversion is trivial. As 1DFA can only move its head to the right, it is enough to take all possible transitions of 1DFA and make them the right transitions of 2DFA.
When we look at the complexity of this operation, we can see that 1DFA with $n$-states can be converted to 2DFA with $n$-states, therefore it takes $n$ steps to compute the 2DFA automaton.

## 3.2 2DFA to 1DFA

Although converting 1DFA to 2DFA is easy, the conversion of 2DFA to 1DFA is not that simple because we have to deal with the left and stationary transitions that are not in 1DFA.

The idea of the conversion is finding a finite set of states and the right transitions between them that are enough to represent the 2DFA. This is done by finding classes of equivalent words [4].

First, let us look at the Myhill-Nerode equivalence relation $=_L$ [4] which is defined as follows:

**Definition 11**

Myhyll-Nerode equivalence relation is denoted $=_L$, where $L$ is some language. We take some $x, y \in L$, the following stands

- $x =_L y$
- $xz \in L \iff yz$ where $z \in \Sigma^*$
- $(xz \in L \iff yz) \implies (yz \in L \iff xz)$ where $z \in \Sigma^*$
- $(xz \in L \iff yz) \wedge (yz \in L \iff xz) \implies xz \in L \iff yz \in L$ where $z \in \Sigma^*$

**Theorem 12**

*(Myhill-Nerode) A language $L$ over an alphabet $\Sigma$ can be accepted by a 1DFA, therefore is regular, if and only if $=_L$ has finitely many equivalence classes.*

The idea of the proof is to show that $L$ is a regular language. We do so by defining a 1DFA $M$ such that $L(M) = L$.

Let $M = (\Sigma, Q, q_0, F, \delta)$ be a 1DFA where

- $\Sigma$ is the input alphabet, the same as the alphabet of $L$
- $Q = \{[w]_{=_L} \mid w \in \Sigma^*\}$ where $[w]_{=_L}$ is an equivalence class
- $q_0 = [\epsilon]_{=_L}$ where $[\epsilon]_{=_L}$ the equivalence class of the empty word
- $F = \{[w]_{=_L} \mid w \in L\}$, where $[w]_{=_L}$ is an equivalence class of $w \in L$
- $\delta$ is a transition function defined by $\delta([w]_{=_L}, x) = [wx]_{=_L}$ where $w \in \Sigma^*, x \in \Sigma$

The proof that the automaton is indeed a valid 1DFA can be found in [4] in section 4. For our purpose, the definition of the automaton is enough. Now we need to show a mechanism to construct such an automaton from 2DFA.

**Definition 13**

We define a function $T_w$

- $T_w \colon Q \cup \{\bullet\} \to Q \cup \{\perp\}$, where the symbols $\bullet$ and $\perp$ are new and their meaning will be shown in a moment.

Let $w = xy$ be such that $w \in \Sigma^*, x \in \Sigma^*, y \in \Sigma$, then

$$T_{xy}(q \in Q) = \begin{cases} p & \text{if } xqy \vdash^* xyp \\ \perp & \text{otherwise} \end{cases}$$

$$T_{xy}(\bullet) = \begin{cases} p & \text{if } q_0xy \vdash^* xyp \\ \perp & \text{otherwise} \end{cases}$$

From the definition above we can see that function $T_w$ for the word $w = xy$ and some state $q \in Q$ outputs state $p$ if there exist such transitions that the automaton reads the character $y$ that follows $x$ on the tape and moves to the right and therefore, gets from the configuration $xqy$ to $xyp$ ($xqy \vdash^* xyp$). If no such sequence exists, the function outputs the new symbol $\perp$.

When we pass $\bullet$ to $T_{xy}$, the output tells us if we ever read the whole word to the right starting in the initial configuration. If there exists such a sequence of configurations from $q_0xy$ to $xyp$ ($q_0xy \vdash^* xyp$) it outputs the state $p$. If we do not ever fully read the word it outputs the symbol $\perp$.

What we achieve here is getting rid of the left and zero transitions. Even if there occurs any of such transitions in the sequence, we ignore them and only care about whether the word is ever read fully to the right. Now we will use this function to determine the new states of the 1DFA from 2DFA.

As applying this function to every word in the automaton language would be impossible because the language might be infinite, we stop after we cannot find any more equivalence classes ($T_w$).

Description of the process of finding the states of 1DFA from 2DFA:

1. initial word $w = \epsilon$, the empty word
2. we create a new starting state $T_\epsilon$, which will be the starting state of the 1DFA
3. for every symbol $c \in \Sigma$ we add it to the current $w$ such that $w = wc$
4. for every $q \in Q \cup \{\bullet\}$ the $T_{wc}(q)$ is computed and state $T_{wc}$ is created
    - if there already exists state $T_x$ such that for each $q \in Q \cup \{\bullet\}$, $T_x(q) = T_{wc}(q)$, we denote them as equivalent (this information will be needed when finding transitions), keep only one of the states and terminate the computation on this branch
    - if there is no equivalent $T_x$, then the state is marked as final state, if $T_{wc}(\bullet) = q_f, q_f \in F$, then we add the state $T_{wc}$ to the list of states and continue with step 3

5. when all branches of computation were terminated we have the list of states of the 1DFA (classes of equivalence)

**Definition 14**
Let $A = (\Sigma, Q, q_0, F, \delta)$ be a deterministic two-way finite automaton. We construct a deterministic one-way finite automaton $A' = (\Sigma, Q', T_\epsilon, F', \delta')$ that accepts the same language ($L(A) = L(A')$) where

- $\Sigma$ is an input alphabet
- $Q'$ is a set of states, $Q' = \{T_w \mid w \in \Sigma^*\}$, the states we found in the algorithm described above
- $q_0 \in Q$, $q_0 = T_\epsilon$ is an initial state
- $F' \subset Q$ is a set of final states such that $F' = \{T_w \mid q_0 w \dashv^* w q_a\}$
- $\delta'$ is a transition function where any transition is $\delta(T_w, c) = T_{wc}$

When defining transitions $\delta'(T_w, c) = T_{wc}$ beware that there might be some equivalent states that we found in the process described above (applying function $T_w$).

The algorithm has a downside - it can end up creating a 1DFA with an exponential increase in the number of states.

## 3.3 2NFA to 1DFA

Conversion of 2NFA to 2DFA is an open problem of the automata theory mentioned also in Section 5. To show the equivalence of 1DFA and 2DFA, the conversion of 2NFA to 1DFA is enough.

The process of conversion is very similar to that in 3.2 when converting 2DFA to 1DFA. We will find classes of equivalennt words like in the Section 3.2.

First, the reader should be familiar with the term configuration. It is explained in Section 2.2.

**Definition 15**
We define a function $T_w$ where $w \in \Sigma^*$ that is adjusted to 2NFA

- $T_w : Q \cup \{\bullet\} \to 2^{Q \cup \{\perp\}}$ where the symbols $\bullet$ and $\perp$ are the same new symbols as for the 2DFA to 1DFA conversion.

Let $w = xy$, such that $w \in \Sigma^*, x \in \Sigma^*, y \in \Sigma$.

$$T_{xy}(q \in Q) = \begin{cases} \{p \in Q \mid xqy \vdash^* xyp\} & \text{for all } p \in Q : xqy \vdash^* xyp \\ \bot & \text{if } \nexists\, p \in Q : xqy \dashv^* xyp \end{cases}$$

$$T_{xy}(\bullet) = \begin{cases} \{p \in Q \mid q_0xy \vdash^* xyp\} & \text{for all } p \in Q : q_0xy \vdash^* xyp \\ \bot & \text{if } \nexists\, p \in Q : q_0xy \dashv^* xyp \end{cases}$$

We can see that function $T_w$ for the word $w = xy$ and some state $q \in Q$ outputs a set of states to which we can get by performing some sequence of transitions. If no such sequence exists, the function outputs the new symbol $\bot$.

When we pass $\bullet$ to $T_{xy}$ the output tells us if we ever read the whole word to the right while starting in the initial configuration. If there exists such a sequence of configurations, it outputs a set of states to which we can get by performing transitions starting in the initial configuration $q_0w$. If we do not ever read the word fully, it outputs the symbol $\bot$.

What we achieve here is getting rid of the left and stationary transitions just like in the conversion of 2DFA to 1DFA. The function $T_w$ will be used to find states of 2NFA from 1DFA. The process is similar except that we will find all possible states we can get to instead of just one.

**Definition 16**
Let $A = (\Sigma, Q, q_0, F, \delta)$ be a 2NFA. We construct a 1DFA $A' = (\Sigma, Q', T_\epsilon, F', \delta')$ that accepts the same language ($L(A) = L(A')$) where

- $\Sigma$ is an input alphabet
- $Q'$ is a set of states, $Q' = \{T_w \mid w \in \Sigma\}$ that are the states we found by the algorithm described above
- $q_0 \in Q$, $q_0 = T_\epsilon$ is an initial state
- $F' \subset Q$ is a set of final states such that $F' = \{T_w \mid \exists q \in F : q \in T_w(\bullet)\}$
- $\delta'$ is a transition function where any transition is $\delta'(T_w, c) = T_{wc}$

When defining transitions $\delta'(T_w, c) = T_{wc}$ beware that there might be some equivalent states.

# 4 Operations

As on 1FA we can perform multiple operations on 2FA. Below are described steps to follow to perform some basic operations [7].

Shepherdson's definition of 2FA is used throughout this and all the later sections meaning the endmarkers cannot be used in the transitions of the user-defined automata.

Although the endmarkers cannot be used to create the user-defined automaton, they are used in the transitions of automata that are created by performing some operations. Those endmarkers do not cause problems with performing other operations on them, though.

## 4.1 Union

Union is a binary operation creating one 2FA from two 2FAs. The final automaton would accept words that belong to the language of either automaton while we consider that both automata share the same alphabet $\Sigma$.

### 4.1.1 Union of 2DFAs

Consider two 2DFAs $A_1$ and $A_2$:

- $A_1 = (\Sigma, Q_1, q_{0,1}, F_1, \delta_1)$
- $A_2 = (\Sigma, Q_2, q_{0,2}, F_2, \delta_2)$

We want to perform union [7] [8] on these automata: *Union(*$A_1$*,* $A_2$*)* $= A$ where $A$ is an automaton such that $A = (\Sigma, Q, q_{0,1}, F, \delta)$ and $L(A) = L(A_1) \cup L(A_2)$. This automaton will be 2NFA.

To perform the union we need to create a new starting state. The starting state will be accepting if either of the final starting states $q_{0,1}$ or $q_{0,2}$ is accepting to keep the acceptance of the empty word.

Then we need to add a stationary transition from the new starting state to each of the starting states which makes the automaton 2NFA.

Now we can define the union of two 2DFAs.

**Definition 17**

Let $A_1 = (\Sigma, Q_1, q_{0,1}, F_1, \delta_1)$ and $A_1 = (\Sigma, Q_2, q_{0,2}, F_2, \delta_2)$ be a 2DFAs. Now let $A = (\Sigma, Q, q_{uninonStart}, F, \delta)$ be a 2NFA created as a union of these automata where

- $\Sigma$ is an input alphabet, the same for both given automata
- $Q$ is a set of states created by uniting $Q_1$ and $Q_2$ (the states with the same name need to be renamed) and adding the new state:

    - $q_{uninonStart}$ is the new starting state

- $q_{uninonStart}$ is a starting state, the one added to $Q$ above
- $F \subset Q$ is a set of final states such that $F = F_1 \cup F_2$
- $\delta$ is a transition function that contains all the transitions defined by $\delta_1$ and $\delta_2$ along with the new transitions
    - $\delta(q_{uninonStart}, c) = (q_{0,1}, S)$ for all $c \in \Sigma$
    - $\delta(q_{uninonStart}, c) = (q_{0,2}, S)$ for all $c \in \Sigma$

If $n_1$ is a number of states of automaton $A_1$ ($n_1 = |Q_1|$) and $n2$ is a number of states of automaton $A_2$ ($n_2 = |Q_2|$), then the automaton $A$ has $n_1 + n_2 + 1$ states, where the extra state is the new starting state.

To summarize the computational process of the automaton $A$, created by the union, on input word $w$:

1. We start in the initial configuration $q_{uninonStart}w$
2. Here, the process branches by performing the stationary transition to $q_{0,1}$ and $q_{0,2}$
3. The first branch after the transition $\delta(q_{uninonStart}, c) = (q_{0,1}, S)$ where $c \in \Sigma$
    - If some accepting configuration of the first automaton $A_1$ is reached $wq_1$ where $q_1 \in F$, the automaton $A$ also accepts
4. The second branch after the transition $\delta(q_{uninonStart}, c) = (q_{0,2}, S)$ where $c \in \Sigma$
    - If some accepting configuration of the first automaton $A_2$ is reached $wq_2$ where $q_2 \in F$, the automaton $A$ also accepts
5. If any rejecting configuration is reached in both branches, the automaton $A$ also rejects the word

### Alternative algorithm

There exists an alternative way to perform the union of 2DFAs that outputs the 2DFA [7]. It uses the conversion of one of the original 2DFAs to a halting 2DFA that has no zero transitions, has only one accepting state, and halts on every input. The conversion of some automaton with $n$ states results in the automaton with $4n + 3$ states. When the operation of a union is performed on them the resulting automaton would have a lot of states and would be quite complex and hard to understand so this algorithm was not used for the purpose of this thesis

[9].

Even though it was not used in the operation of the union, the part of the algorithm, specifically the conversion of 2DFA to a 2DFA with no zero transitions and only one accepting state defined in Definition 18, was used in the operation of complementation.

The algorithm may not be used but the idea of how it works is provided here.

To perform union we first need to convert 2DFA to 2DFA which has no stationary transitions, and only one accepting state [9].

**Definition 18**

Let $A = (\Sigma, Q, q0, F, \delta)$ be a 2DFA with $n$ states. We construct $A' = (\Sigma, Q', q0, F', \delta')$ that has no stationary transitions and has only one final state ($|F| = 1$). Such an automaton will have $n + 1$ states. The one extra state is the new final state.

If you look at source [9] you will find that they create such automata with the same number of states using one of the original states as the one finite state. In our case, this is not possible as our automaton does not stay in the finite states (there can be a transition to the non-final state).

First, we create a new state $q_f$ such that $q_f \notin Q$ will be the special accepting state where the automaton always ends when accepting. For every $q \in F$ the transition $\delta(q, \lhd) = (q_f, L)$ is added where $\lhd$ is the right endmarker.

To read the word all the way to the right in the accepting state, for each $c \in \Sigma \cup \{\rhd\}$ where $\rhd$ is the left endmarker the transition $\delta(q_f, c) = (q_f, R)$ is added.

Second, the stationary transitions are removed. A stationary transition $\delta(q, c) = (q', S)$ where $q, q' \in Q$ and $c \in \Sigma$ two possible outcomes are possible:

1. There is a sequence of stationary transitions after which the head moves to the left or to the right, therefore, there is a transition $\delta(p, c) = (p', L)$ or $\delta(p, c) = (p', R)$ where $p, p' \in Q$ that is eventually performed. To remove the sequence of stationary transitions we define $\delta(q, c) = (p', L)$ or $\delta(q, c) = (p', R)$ according to the last stationary transition in the sequence.

2. There is no sequence of transitions after which the head moves anywhere which means the loop is entered. In this case, we simply remove the transition so the automaton would have no transition to make and will reject the word.

As Shepherdson's definition is used we do not need to worry about left or right endmarkers in stationary transitions in user-defined automata which would have caused a problem because if the transition would be $\delta(q, \rhd) = (q, S)$ it could be accepting even if there would be no sequence of transitions after which the head moves to the left or to the right.

Now we can define the union of two 2DFAs.

**Definition 19**

Let $A'_1 = (\Sigma, Q_1, q_{0,1}, F_1, \delta_1)$ be a 2DFA created as described in Definition 18 and $A_2 = (\Sigma, Q_2, q_{0,2}, F_2, \delta_2)$ be a 2DFA. The $A'_1$ has to be converted to automaton $A''_1$ that halts on every inputs as described here [9].

Now let $A = (\Sigma, Q, q_0, F, \delta)$ be a union of these automata where

- $\Sigma$ is an input alphabet, the same for both given automata
- $Q$ is a set of states created by uniting $Q''_1$ and $Q_2$ (the states with the same name need to be renamed and adding the new state)
  - $q_{return}$ state used to return to the start of the tape
- $q_0 \in Q, q_0 = q''_{0,1}$ is an initial state which was also the initial state of $A''_1$
- $F \subset Q$ is a set of final states such that $F = F''_1 \cup F_2$
- $\delta$ is a transition function that contains all the transitions defined by $\delta''_1$ and $\delta_2$ along with the new transitions which have to ensure that when the first automaton $A''_1$ rejects, the head returns to the beginning of the tape using the transitions with the new state $q_{return}$ and then computing on the second automaton $A_2$

If $4n + 3$ is a number of states of automaton $A''_1$ and $n_2$ is a number of states of automaton $A_2$, then the automaton $A$ has $4n + n_2 + 4$ states.

To summarize the computational process of the automaton $A$ on an input word $w$:

1. The initial configuration $q_{0,1}w$
2. If an accepting configuration of the first automaton $A_1$ is reached $wq_{f1}$ where $q_1 \in F$, the automaton $A$ also accepts
3. If a rejecting configuration of the first automaton $A_1$ is reached $wq_1$ where $q \notin F$, we move the head to the start of the tape using transitions with state $q_{return}$ and then move to the starting state $q_{0,2}$ of the second automaton $A_2$
4. If an accepting configuration of the second automaton $A_2$ is reached $wq_{f2}$ where $q_2 \in F$, the automaton $A$ also accepts
5. If a rejecting configuration of the second automaton $A_2$ is reached $wq_2$ where $q_2 \notin F$ , the automaton $A$ also rejects

### 4.1.2 Union of 2NFAs

Consider two 2NFAs $A_1$ and $A_2$:

- $A_1 = (\Sigma, Q_1, q_{0,1}, F_1, \delta_1)$
- $A_2 = (\Sigma, Q_2, q_{0,2}, F_2, \delta_2)$

We want to perform union [7] [8] on these automata: *Union(*$A_1$*,* $A_2$*)* $= A$, where $A$ is the final automaton created by the union, such that $A = (\Sigma, Q, q_{0,1}, F, \delta)$ and $L(A) = L(A_1) \cup L(A_2)$. This automaton will be a 2NFA.

The algorithm for the union of 2NFAs is the same as for the union of 2DFAs described in Section 4.1.1. The alternative way of union also described in that section is not possible for 2NFA as no polynomial-time conversion of 2NFA to 2DFA is possible.

## 4.2 Intersection

Intersection is a binary operation creating one 2FA from two 2FAs. The final automaton would accept words that belong to the language of both automata while we consider that both automata share the same alphabet $\Sigma$.

### 4.2.1 Intersection of 2DFAs

Consider two 2DFAs $A_1$ and $A_2$:

- $A_1 = (\Sigma, Q_1, q_{0,1}, F_1, \delta_1)$
- $A_2 = (\Sigma, Q_2, q_{0,2}, F_2, \delta_2)$

We want to perform intersection [7] [8] on these automata: $Intersect(A_1, A_2) = A$, where $A = (\Sigma, Q, q_{0,1}, F, \delta)$ is 2DFA created by the intersection that accepts the language $L(A) = L(A_1) \cap L(A_2)$.

**Definition 20**

Let $A_1 = (\Sigma, Q_1, q_{0,1}, F_1, \delta_1)$ and $A_1 = (\Sigma, Q_2, q_{0,2}, F_2, \delta_2)$ be a 2DFAs. Now let $A = (\Sigma, Q, q_0, F, \delta)$ be a 2DFA created by performing an intersection on these automata where

- $\Sigma$ is an input alphabet, the same for both given automata
- $Q$ is a set of states created by uniting $Q_1$ and $Q_2$ (the states with the same name need to be renamed) and adding the new state
  - $q_{return}$ state used to return to the start of the tape
- $q_0 \in Q, q_0 = q_{0,1}$ is a starting state which was also the starting state of $A_1$

- $F \subset Q$ is a set of final states such that $F = F_1 \cup F_2$
- $\delta$ is a transition function that contains all the transitions defined by $\delta_1$ and $\delta_2$ along with the new transitions
  - $\delta(q_f, \triangleleft) = (q_{return}, L)$ for all $q_f \in F$ while $\triangleleft$ is the right endmarker
  - $\delta(q_{return}, c) = (q_{return}, L)$ for all $c \in \Sigma$
  - $\delta(q_{return}, \triangleright) = (q_{0,2}, R)$ where $q \in Q$ and $\triangleright$ is the left endmarker

To summarize the computational process of the automaton $A$ created by the union on input word $w$:

1. The initial configuration $q_{0,1}w$
2. If a rejecting configuration of the first automaton $A_1$ is reached $wq_1$ where $q_1 \notin F$, the automaton $A$ also rejects
3. If an accepting configuration of the first automaton $A_1$ is reached $wq_{f1}$ where $q_{f1} \in F$ the head is moved to the start of the tape using transitions with state $q_{return}$ and then moved to the starting state $q_{0,2}$ of the second automaton $A_2$
4. If an accepting configuration of the second automaton $A_2$ is reached $wq_{f2}$ where $q_{f2} \in F$, the automaton $A$ also accepts
5. If a rejecting configuration of the second automaton $A_2$ is reached $wq_2$ where $q_2 \notin F$, the automaton $A$ also rejects

### 4.2.2 Intersection of 2NFAs

Consider two 2NFAs $A_1$ and $A_2$:

- $A_1 = (\Sigma, Q_1, q_{0,1}, F_1, \delta_1)$
- $A_2 = (\Sigma, Q_2, q_{0,2}, F_2, \delta_2)$

We want to perform intersection [7] [8] on these automata: *Intersect(*$A_1$*,* $A_2$*)* $= A$, where $A = (\Sigma, Q, q_0, F, \delta)$ is 2NFA created by the intersection that accepts the language $L(A) = L(A_1) \cap L(A_2)$.

The algorithm for the intersection of 2NFAs is the same as for the union of 2DFAs described in Section 4.2.1.

## 4.3 Concatenation

Concatenation is a binary operation creating one 1DFA from two 2FAs. The final automaton accepts the concatenation of their languages, while we consider that both automata share the same alphabet $\Sigma$.

The final automaton is not 2FA but 1FA because there is no known algorithm to perform such an operation on 2FAs. The problem with this algorithm are the multi-directional transitions of 2FAs. 2FAs first need to be converted to 1DFAs and then the concatenation is performed just like on the classic 1DFAs.

### 4.3.1 Concatenation of 2DFAs

Consider two 2DFAs $A_1$ and $A_2$:

- $A_1 = (\Sigma, Q_1, q_{0,1}, F_1, \delta_1)$
- $A_2 = (\Sigma, Q_2, q_{0,2}, F_2, \delta_2)$

We want to perform concatenation [7] on these automata: *Concatenate(*$A_1$*,* $A_2$*)* $= A$ where $A = (\Sigma, Q, q_0, F, \delta)$ is 1DFA created by the concatenation that accepts the language $L(A) = L(A_1)L(A_2)$.

The steps to achieve concatenation of two 2DFAs:

1. Perform the conversion to 1DFA for both automata by using the algorithm described in 3.2. We get 1DFA $A_{dfa1}$ from $A_1$ and 1DFA $A_{dfa2}$ from $A_2$.
2. Create an $\epsilon$-1NFA by adding the transition under the empty word $\epsilon$ from the states in $F_{dfa1}$ to the starting state of the second automaton $q_{0,dfa2}$
3. Convert 1NFA to 1DFA

The process of conversion from 1NFA to 1DFA is omitted in this work but details can be found here [11].

**Definition 21**

Let $A_1 = (\Sigma, Q_1, q_{0,1}, F_1, \delta_1)$ and $A_1 = (\Sigma, Q_2, q_{0,2}, F_2, \delta_2)$ be 2DFAs, then they are converted to 1DFAs $A_{dfa1}$ and $A_{dfa2}$, and then we create an $\epsilon$-1NFA as mentioned in the process above. Now let $A = (\Sigma, Q, q_{0,1}, F, \delta)$ be a concatenation of these automata created from the $\epsilon$-1NFA where

- $\Sigma$ is an input alphabet, the same for both given automata
- $Q$ is a set of states created by converting $\epsilon$-1NFA to 1DFA (epsilon closures)
- $q_0 \in Q$ is the state representing the epsilon closure of $q_{0,1}$
- $F \subset Q$ is a set of finite states created in conversion from 1NFA to 1DFA by finding closures that contains the finite states

- $\delta$ is a transition function

The process of computation of the concatenated automata $A$ on an input word $w$ is the same as for a classic 1DFA.

### 4.3.2 Concatenation of 2NFAs

Consider two 2NFAs $A_1$ and $A_2$:

- $A_1 = (\Sigma, Q_1, q_{0,1}, F_1, \delta_1)$
- $A_2 = (\Sigma, Q_2, q_{0,2}, F_2, \delta_2)$

We want to perform concatenation [7] on these automata: *Concatenate($A_1$, $A_2$)* $= A$, where $A = (\Sigma, Q, q_0, F, \delta)$ is a 1DFA created by the intersection that accepts the language $L(A) = L(A_1)L(A_2)$.

The steps to achieve concatenation of two 2NFAs:

1. Perform the conversion to 1DFA for both automata by using the algorithm described in Section 3.3. We get a 1DFA $A_{dfa1}$ from $A_1$ and a 1DFA $A_{dfa2}$ from $A_2$.
2. Create an $\epsilon$-1NFA by adding the transition under the empty word $\epsilon$ from the states in $F_{dfa1}$ to the starting state of the second automaton $q_{0,dfa2}$
3. Convert 1NFA to 1DFA

The rest of the algorithm is the same as for 2DFAs described in the section above.

## 4.4 Square

Square is a unary operation creating one 1DFA from two 2FAs. For this operation, the number of squares $n$ has to be specified. This operation creates an automaton that accepts the concatenation of $n$ words that belong to the original language.

The final automaton is not 2FA but 1FA because there is no known algorithm for performing such an operation on 2FA. 2FA first needs to be converted to 1DFA and then the square operation is performed just like on a classic 1DFA.

The definition of $n$-square

- if $n = 0$ then 0-square of a language $L$ is a language accepting only the empty word $\epsilon$
- if $n > 0$ then $n$-square of a language $L$ is the language $L^n = \{w_1.....w_n | i \in \{1, 2, ..., n\}, w_i \in L_i\}$

As we can see from the definition above, the $n$-square of a language can be easily done using concatenation.

### 4.4.1 Sqaure of 2DFA

Consider a 2DFA $A$:

- $A = (\Sigma, Q, q0, F, \delta)$

We want to perform square [7] of n where $n \geq 0$ on the automaton: $Square(A, n) = A'$, where $A' = (\Sigma, Q', q0', F', \delta')$ is a 1DFA created by the square that accepts the language $L(A') = L(A)^n$.

The steps to achieve the $n$-square of the 2DFA:

1. If $n = 0$, then create an automaton accepting only the empty word
2. If $n > 0$, then perform concatenation in a cycle $n - 1$ times; the language $L(A') = L(A)^n$ will be created inductively such that $L(A)^1 = L(A)$, and for some $m > 1, m \leq n$ then $L(A)^m = L(A)L(A)^{m-1}$

**Definition 22**
Let $A = (\Sigma, Q, q_0, F, \delta)$ be a 2DFA and $n \geq 0$ then use concatenation in a cycle to create the automaton of $n$-square. Let $A = (\Sigma, Q', q_0', F', \delta')$ be the $n$-square of the automaton created where

- $\Sigma$ is an input alphabet
- $Q'$ is a set of states created by concatenation
- $q_0' \in Q'$ is an initial state created by concatenation
- $F' \subset Q'$ is a set of final states
- $\delta'$ is a transition function (the description is not mentioned as it should be clear from the process of concatenation)

The process of computation of the $n$-square is the same as for concatenation.

### 4.4.2 Square of 2NFA

Consider a 2NFA $A$:

- $A = (\Sigma, Q, q0, F, \delta)$

We want to perform square [7] of $n$ where $n \geq 0$ on the automaton: $Square(A, n) = A'$, where $A' = (\Sigma, Q', q_0', F', \delta')$ is a 1DFA created by the square that accepts the language $L(A') = L(A)^n$.

The rest of the algorithm is the same as for 2DFA.

## 4.5 Complement

Complement is a unary operation creating a complementary 2FA. The final automaton accepts all words except the words that belong to the language of the original automaton.

The operation of complement is simple when performed on a 1FA as the head can move only in one direction, just changing the accepting type to non-accepting and vice versa is enough.

When it comes to 2FA, the head can move in both directions or perform stationary transitions, therefore, there is a possibility of a loop. The complement of a 2FA should accept any word that loops on the original 2FA. The problem is that the loops stay in 2FA even if we interchange the accepting states with non-accepting so some loop detection has to be done.

### 4.5.1 Complement of 2DFA

Consider a 2DFA

- $A = (\Sigma, Q, F, \delta)$

We want to create a complement [7] of this automaton: $Complement(A) = A'$ where $A' = (\Sigma, Q', q0, F', \delta')$ created by complementing the original automaton that accepts the language $L(A') = L(A)$.

To perform the complementation the automaton should be converted to a 2DFA that halts on every input as mentioned in Section 4.1.1. When we get an automaton that halts on every input, it is then easy to determine whether the automaton halted in an accepting or a rejecting state. In this case, the modification of transitions is quite simple to create the complement of a 2DFA.

Although, as already mentioned in Section 4.1.1, the conversion to a halting automaton outputs an automaton that has four times the number of states of the original 2FA, and is quite hard to comprehend. That is the reason why this conversion is not used.

The other reason for not using the conversion is because the library described in Section 6 that is implemented in this work provides a way to detect loops in the computation of 2FA. If the automaton reaches a configuration that it has already been to, it entered a loop. Therefore, we only use this part of the original algorithm from Definition 18 that converts a 2DFA to a 2DFA that has no zero transitions and only one accepting state, and the transitions are modified such that every transition that ends in the accepting state is removed and for every missing transition the new one is added to the accepting state. The rest of the algorithm complementing is done by the loop detection.

Now we can define the complement of the 2DFA:

**Definition 23**

Let $A = (\Sigma, Q, q0, F, \delta)$ be a 2DFA that has no stationary transitions and only one accepting state created using Definition 18. Now let $A' = (\Sigma, Q, q_{0,1}, F, \delta')$ be a 2DFA created by complementing this automaton where

- $\Sigma$ is an input alphabet
- $Q$ is a set of states
- $q_0 \in Q$ is the initial state
- $F \subset Q$ is a set containing one final state
- $\delta'$ is a transition function with modified transitions of $A$
  - for each $c \in \Sigma \cup \{\triangleleft, \triangleright\}$ and for each $q \in Q - F$
    1. if there is a transition $\delta'(q, c) = (q_f, direction)$ where $q_f \in F$ the transition is removed
    2. if there is no transition such that $(q, c) = \perp$ a new transition $\delta'(q, c) = (q_f, direction)$ is added, and $direction = R$ if $c = \triangleright$ else it is $L$

To summarize the process of computation of the complement automaton $A$ on an input word $w$:

1. The initial configuration $q_0 w$
2. If the configuration $wq_f$ is reached where $q_f \in F$ or the automaton loops, the word is accepted as it would be rejected by $A$
3. If the configuration $wq$ is reached where $q \notin F$ or no transition exists, the word is rejected as it would be accepted by $A$

### 4.5.2 Complement of 2NFAs

Consider a 2NFA $A = (\Sigma, Q, F, \delta)$.

We want to create a complement [7] of this automaton: $Complement(A) = \bar{A}$ where $\bar{A} = (\Sigma, Q', q0, F', \delta')$ is a 1DFA created by converting 2NFA to 1DFA and the complementing it and it accepts a complement of the original language $L(\bar{A}) = \overline{L(A)}$.

As we can see the complementing of 2NFA is not possible to perform like on 2DFA as we cannot convert 2NFA to 2DFA that has no stationary transitions and only one halting state due to non-determinism and branching in computation.

The computation can be done by performing conversion to 1DFA as described in Section 3.3 and then performing complementation on 1DFA by switching the accepting and non-accepting states.

Now we can define the complement of 2NFA:

**Definition 24**

Let $A = (\Sigma, Q, q_0, F, \delta)$ be a 2NFA we convert it to a 1DFA $A' = (\Sigma, Q', q_0', F', \delta')$ as described in Section 3.3 and having a full transition table (every state has a transition defined under any symbol from the alphabet). Now let $\bar{A} = (\Sigma, Q', q_0', \bar{F}', \delta')$ be a 1DFA created by complementing the automaton $A'$ where

- $\Sigma$ is an input alphabet
- $Q'$ is a set of states
- $q_0' \in Q'$ is the initial state
- $\bar{F}' \subset Q'$ is a set of final states, $\bar{F}' = Q - F'$
- $\delta'$ is a transition function, the same as for the automaton $A'$

## 4.6 Kleene Star

Kleene star is a unary operation creating 1FA from 2FA. The final automaton accepts a language created by a concatenation of the original one where the concatenation can be performed n times where $n \geq 0$ where $L(A)^0 = \{\epsilon\}$ and the rest is done by concatenation with language $L$.

There is no known algorithm for performing a star on 2FA and creating a 2FA. Therefore, the 2FA is first converted to a 1DFA and then the star is performed like on a classic 1DFA.

### 4.6.1 Kleene Star of 2DFA

Consider a 2DFA

- $A = (\Sigma, Q, q_0, F, \delta)$

We want to perform star [7] on this automaton: $Star(A) = A'$, where $A' = (\Sigma, Q', q_0', F, \delta')$ is the final automaton created by applying the star to a 1DFA that we create from $A$ as described in Section 3.2. The language of $A'$ is $L(A') = L(A)^*$, where $L(A)^* = \bigcup_{n \geq 0} L(A)^n$ and $n$ is a number of concatenations performed on $L(A)$ and it starts with $L(A)^0 = \{\epsilon\}$ and $L(A)^n = L(A)^{n-1}, n \geq 1$. The resulting automaton accepts any number of concatenated words of the language of the original automaton.

The star operation on 1DFA requires to convert 1DFA to 1NFA:

1. Let $A$ be a 1DFA such that $A = (\Sigma, Q, q_0, F, \delta)$
2. Create a 1NFA $A' = (\Sigma, Q', q_0', F', \delta')$ such that
    - $\Sigma$ is the alphabet
    - $Q'$ is a set of states containg $Q$ and new states

    - $q'_0$, a new starting state
    - $q'_f$, a new accepting state
- $q'_0 \in Q'$ is the initial state
- $F' \subset Q'$ is a set of final states, $F' = F \cup \{q_f\}$
- $\delta'$ is a transition function contain new transitions
    - $\delta'(q'_0, \epsilon) = q_0$, transition under empty word to the original initial state
    - $\delta'(q, \epsilon) = q'_f$ for each $q \in F$, transitions under empty word to a final state
    - $\delta'(q, \epsilon) = q_0$ for each $q \in F$, transitions under empty word from every original finite state to original starting state
    - $\delta'(q'_0, \epsilon) = q_f$ a transition under empty word from every the new starting state to the finite state

As we can see, the final automaton $A'$ is 1NFA with epsilon transitions. Epsilon transition ($\epsilon$-transition) is a special transition where the automaton can move to the next state without reading any symbol from the tape. By converting the automaton from 1NFA to 1DFA we get the wanted automaton created by performing the Kleene Star operation [11].

**Definition 25**

Let $A = (\Sigma, Q, q_0, F, \delta)$ and let $A_{dfa}$ be a 1DFA that $A$ was converted to by Definition 3.2, then we create some $\epsilon$-1NFA as mentioned in the process above. Now let $A' = (\Sigma, Q', q'_0, F', \delta)$ be a star of that automaton created from the epsilon-1NFA where

- $\Sigma$ is an input alphabet
- $Q'$ is a set of states created by converting epsilon-1NFA to 1DFA (epsilon closures)
- $q'_0 \in Q'$ is the state representing the epsilon closure of $q_0$
- $F' \subset Q'$ is a set of final states created in conversion from 1NFA to 1DFA by finding closures that contains the final states
- $\delta'$ is a transition function, it is not described here as it should be clear from the process of concatenation

### 4.6.2 Kleene Star of 2NFA

Consider a 2NFA $A = (\Sigma, Q, pq_0, F, \delta)$

We want to perform star [7] on this automaton: $Star(A) = A'$, where $A' = (\Sigma, Q', q'_0, F, \delta')$ is the final automaton created by applying the star to a 1DFA that we create from $A$ as described in Section 3.3.

The rest of the algorithm is the same as for 2DFA.

## 5 Open problems of 2FA

In the theory of two-way final automata lies many problems that have not been solved yet. In this section, we briefly mention some of them.

The problem with solving the open problems is mainly caused by the head moving in more directions in 2DFAs. As we have seen in Section 4 many algorithms for operations rely on converting 2DFA to 1DFA as no algorithms for 2FA have been discovered yet.

### 5.1 Size Complexity

The problem: Can a 2NFA with n states be converted to a 2DFA with at most $p(n)$ states, where $p$ is a polynomial function?

This is one of the most popular problems that is spread throughout automata theory. The problem can be defined for all types of automata.

The most general case is NP vs P problem (Turing machines). It is well-known that this problem has not been solved yet but it is assumed that P is not equal to NP.

Now let's look at the problem for 2DFAs. The interpretation of the problem is whether the conversion of 2NFA to 2DFA is bounded above polynomially.

Sakoda and Sipser came up with a theory that has its origin in the theory of NP-completeness [3].

This theory starts with defining two classes of regular problems

- $2D$ - the class of problems for which a 2DFA exists such that it has a maximum of polynomial number states
- $2N$ - the class of problems for which a 2NFA exists such that it has a maximum of polynomial number of states

If we have some $L \in 2N$, the problem asks whether $L \in 2D$. If this would stand for all problems in $2D$, then $2D = 2N$.

They also introduced so-called homomorphic reductions between the problem families. It was proven that 2D is closed under them.

This problem shows the power of non-determinism and if it was proven that $2D = 2N$, it would mean that every regular problem might be solved in polynomial time.

There is a next branch to this problem considering a conversion of 1NFA to 2DFA: Can 1FA with $n$ states be converted to 2DFA with at most $p(n)$ states, where $p$ is a polynomial function?

This as well is thought to not be true and was researched along the 2NFA to 2DFA problem above.

## 5.2 Open Problem of Operations of 2FA

As mentioned in the section about operations (in Section 4) some of them cannot be done without conversion to 1DFA. The reason is that algorithms considering 2FAs either have large time-complexity or have not yet been discovered.

The main problem of 2DFAs when it comes to operations is the possibility to perform transitions in more directions. Although, this gives 2DFAs the power to have fewer states in many cases.

The research on the complexity of operations was done by Jirásková and Okhotin [7] that were used as the main source for implementing operations in Section 4. They used the knowledge about operations on 1DFA and compared them to 2DFA.

It is clear that even the basic operations have a trend of increasing the number of states in 2DFA. It is not clear if better algorithms can be discovered.

# 6 Library

A library TwoWayFiniteAutomata for 2FA was created as a .NET class library using the object-oriented programming language C#. It can work with deterministic and also non-deterministic automata.

It allows users to create 2FA with a custom alphabet and transitions, load 2FA from an XML file, save 2FA to an XML file, and perform basic operations on 2FA.

The most basic operation the library can perform is the computation on an input word that determines whether the word belongs to the language of an automaton or not. The process of computation can be stored as a list of configurations if a user wishes to see how the automaton works.

As 2FA is just an upgraded version of 1FA the library can also work with 1FA. As we will see, this is important as the outputs of some operations on 2FA are 1FA.

Before we look at some example usages of the library the following sections give a deeper knowledge of the structure of the library.

## 6.1 Classes

### Automaton

Automaton is the main class of the library representing 2FA. Its fields are:

- **Id** - an integer representing the identifier of the automaton
- **Description** - a description of the automaton and its language
- **Tape** - an array of strings representing the tape of the automaton
- **Type** - a type of automaton (enum AutomataType - 1DFA, 2DFA, 1NFA, 2NFA)
- **Operation** - an operation by which the automaton was created, the default is ORIGINAL (enum OperationType)
- **States** - a list of states (class State) that the automaton can reach, it is obtained from the Transitions
- **StartState** - the starting state, also belongs to the list of States
- **WordAlphabet** - an input alphabet, a list of strings
- **Transitions** - a list of transitions that the automaton can make (class Transition)

- **ProcessingConfigurations** - list of strings, each string represents a configuration and therefore, the list of them represents some computation on an input word
- **NonendingManually** - a non-ending state used in operations as a universal non-ending state
- **InitialManually** - a starting state used in operations as a universal starting state
- **Count** - an integer used to produce the Id field, it is used as a static variable for all instances of the class

The fields described above are used in the following methods:

- (constructor) **Automaton** - creates an instance of the class and initializes the fields to default values as it has no arguments, this constructor is needed for XML loading and saving
- (constructor) **Automaton** - creates an instance of the class, its arguments are tape alphabet, transitions, type, and operation, some arguments are optional and have a default value
- **GetStatesFromTransition** - the method to get a list of states from the transitions
- **UpdateStates** - updates the States field when a transition is added
- **AddState** - adds a state to the States filed while checking its validity
- **AddTransition** - adds a transition while checking its validity
- **CheckTransitionValidity** - checks the validity of a Transition
- **AddAlphabet** - adds alphabet while checking its validity
- **ComputeType** - computes type of the automaton (1DFA, 1NFA, 2DFA, 2NFA) based on transitions
- **LoadFromXmlFile** - loads all automata from an XML file which path is given as an argument
- **SaveToXmlFile** - saves the automata to an XML file which is given as an argument
- **Input** - gets a word from the method Compute as an argument, saves it to the Tape, and prepares automaton for computation

- **Compute** - performs a computation with the input word that it gets as an argument, it determines whether the word belongs to the language of the automaton, it saves the configurations of the computation
- **ComputeInput** - helper function called by compute
- **Configuration** - creates the configuration string based on the head position and current state
- **Union** - performs a union of two automata (the one that the method is called from and one is given as an input argument), it outputs the new automaton
- **Intersect** - performs intersection of two automata (the one that the method is called from (this) and one is given as an input argument)
- **Square** - creates square of an automaton based on a given integer parameter
- **Complement** - creates an automaton that accepts the complement of the language
- **Concatenate** - concatenate two automata
- **Kleene** - creates a Kleene Star from an automaton
- **T** - helper function used by ConvertToDfa
- **EpsilonClosure** - a helper function that creates an epsilon closure of some state
- **ConvertToNoZeroTransitionsOneFiniteState** - converts automaton to an automaton with no zero transitions and one final state
- **ConvertToDfa** - converts 2DFA to 1DFA
- **ConvertNfaToDfa** - converts 2NFA to 2DFA

**State**

The State class represents a state of an automaton. Its fields are:

- **Name** - a name of a state
- **InitType** - a type of a state (enum StateType), can only be STARTING or NONENDING, determines whether the state is initial or not
- **FiniteType** - a type of state (enum StateType), can have any value from the enum, determines whether the state is accepting or not

### Transition

The Transition class represents a transition of an automaton. Its fields are:

- **Start** - the starting state of the transition (class State)
- **Finish** - the finishing state of the automata (class State)
- **Character** - the transition character (string)
- **Direction** - the direction in which the head moves over tape after the transition (enum Direction)

Based on the description above, it should be clear how the transitions work. When there is a symbol (stored in the Character field) on the tape and we are in the Start state, then we can perform the transition to the Finish state and move the head in the given Direction.

## 6.2 Enums

### StateType

The enum *StateType* represents the type of automaton state. Its values are:

- **STARTING** - the type of the starting state of an automaton ($S$ in the definition of the automaton)
- **NONENDING** - the type of a non-ending state of an automaton (the state that is not of type STARTING or ACCEPTING)
- **ACCEPTING** - the type of an accepting state of the automaton

### AutomatonType

The enum *AutomatonType* represents the type of automaton based on its transitions. Its values are:

- **DFA2** - deterministic two-way finite automaton
- **DFA** - deterministic one-way finite automaton
- **NFA2** - non-deterministic two-way finite automaton
- **NFA** - non-deterministic one-way finite automaton

**OperationType**

The enum *OperationType* represents the operation by which the automaton was created. Its values are:

- **ORIGINAL** - the default value
- **UNION**
- **INTERSECTION**
- **STAR**
- **COMPLEMENT**
- **CONCATENATION**
- **SQUARE**

## 6.3 Automata in XML File

As the creation of the automata is quite a time-consuming operation, the library provides users with functions to work with automata stored in an XML file. Also, the automata created by the library can be saved to an XML file.

The automaton represented in XML format must have the following structure as presented in the source code 1 to be a valid automaton. The source code presents a file that contains one 2DFA automaton.

The root element of the XML file is *ArrayOfAutomaton*. It represents a list of automata, and therefore more than one automaton can be stored in one XML file.

An automaton inside the *ArrayOfAutomaton* starts and ends with the *Automaton* element. It has no attributes.
Inside the *Automaton* element, the following elements can be used to define an automaton:

- **Description** (optional) - the description of the automaton, if not given, the description of the automaton will be an empty string
- **Type** (optional) - the type of the automaton, can have values from enum AutomataType; please note that this field is not necessary while creating the automaton as the Type will be recomputed once the library loads the file to not cause any inconsistency, so this field has more usage when saving the automaton as the user can be sure it is valid

- **Operation** (optional) - the type of operation by which the automaton was created; when creating the automaton manually, always set it to ORIGINAL or omit it completely, other types of operations from enum OperationType can be used while saving the automaton; please note that you should never change the type of operation in the saved file as this may lead to an unwanted behavior on the next load
- **StartState** (required) - the initial state of the automaton, it has no body, but its parameters are
  - **Name** (required) - the name of the state
  - **InitType** (optional) - the inital type of the state, has to be the STARTING from enum StateType, if not given, this will be set after loading
  - **FiniteType** (optional) - the finite type of the state, can have any value from enum StateType except STARTING; if not given, it will be set to NONENDING after loading
- **WordAlphabet** (optional) - an element conating elements of strings representing the automaton alphabet
- **Transitions** (required) - the list of Transitions, in the body we use the tag Transition
  - **Transition** (at least one required) - represents a transition of the automaton
    - **Start** (required) - the start state of the transition, the arguments are the same as for the StartState
    - **Finish** (required) - the finish state of the transition
    - **Character** (required) - the symbol from the alphabet under which the transition should happen; the character cannot be an endmarker symbol if it is a user defined automaton, the endmarker symbol can be present only if such a transition is added by an operation
    - **Direction** (required) - the direction in which the head should move while performing the transition; it can have values from enum Direction; if not given, its default value will be STAY (stationary transition)

To make the XML file simple and consistent while loading automata from the XML file, all other fields of automata are generated from the transitions. Those are WordAlphabet and States. The id of an automaton is generated after each load.

The WordAlphabet is not always generated. As you can see the alphabet can be passed as an element but if you do not specify it, it will be generated from the transitions.

Also, if you specify an alphabet and then use a symbol that is not in it in a transition it will be added to the alphabet to keep the consistency.

Please note that if you do not use the StartState in any transition, you will get an error as such an automaton is useless as it can never start the computation. Also, when you do not use any of the required elements a parse error should be thrown.

Source code 1: automata.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<ArrayOfAutomaton>
    <Automaton>
        <Description>01*</Description>
        <Type>DFA2</Type>
        <Operation>ORIGINAL</Operation>
        <StartState Name="q0" InitType="STARTING" FiniteType="NONENDING"/>
        <WordAlphabet>
                <string>0</string>
                <string>1</string>
        </WordAlphabet>
        <Transitions>
            <Transition>
                <Start Name="q0" InitType="STARTING" FiniteType="NONENDING"/>
                <Finish Name="q0" InitType="STARTING" FiniteType="NONENDING"/>
                <Character>1</Character>
                <Direction>LEFT</Direction>
            </Transition>
            <Transition>
                <Start Name="q0" InitType="STARTING" FiniteType="NONENDING"/>
                <Finish Name="q1" InitType="NONENDING" FiniteType="ACCEPTING"/>
                <Character>0</Character>
                <Direction>RIGHT</Direction>
            </Transition>
            <Transition>
                <Start Name="q1" InitType="NONENDING" FiniteType="ACCEPTING"/>
                <Finish Name="q1" InitType="NONENDING" FiniteType="ACCEPTING"/>
                <Character>1</Character>
                <Direction>RIGHT</Direction>
            </Transition>
            <Transition>
                <Start Name="q1" InitType="NONENDING" FiniteType="ACCEPTING"/>
                <Finish Name="q0" InitType="STARTING" FiniteType="NONENDING"/>
                <Character>0</Character>
                <Direction>LEFT</Direction>
            </Transition>
        </Transitions>
    </Automaton>
</ArrayOfAutomaton>
```

**Restriction on Automata Creation**

1. required starting state
2. the starting state has to be used in at least one transition
3. input alphabet cannot contain "<" and ">" as those are used internally as tape endmarkers
4. at least one transition has to be present

## 6.4 Automata Creation

The library also allows users to create an automaton in the code.

You can create an empty automaton by calling the empty constructor Automaton. The fields will be initialized to the default values. This constructor is mainly used for XML parsing.

The automata can be created using the custom fields with the constructor Automaton whose parameters are:

- alphabetWords - a list of strings representing the input alphabet of the automaton
- transitions - a list of transitions
- operationType - an operation type of the automaton (enum OperationType), the default value is ORIGINAL

This constructor is used in the operations of the library.

As already mentioned, the library can work with all types of FA, one-way, two-way, deterministic, and non-deterministic. The 1FAs are stored as 2FA with only right transitions.

## 6.5 Computation

The operation of computation is the most basic one. The method Compute is called with an input word as a parameter. The input word is validated by checking if all the symbols of it are in the alphabet. As the alphabet symbols can have multiple characters, the word is given as a list of strings, not a simple string.
If the word is validated correctly, it is written to the tape, and then the transitions are performed.

The computation process saves itself as a list of configurations so the user can check it. In the case of NFA, it saves all branches of the computation.

If there is such a sequence of transitions that the automaton can read the word all the way to the right endmarker and no transitions under the right endmarker are possible (can be present in an automaton created by some operations like union etc.), it accepts the word by outputting the boolean value *true.*

If the computation process gets stuck in a loop, no more transitions are possible, or it falls out of the tape, it rejects the word by outputting the boolean value *false.*

The loop detection is done by checking the configurations. If the automaton gets into a configuration it has already been to, it starts to loop.

The FA computes on a word given as an input by a user and outputs whether the input word belongs to the language of the automaton or not. The input word is given as a list of strings as it is possible to have multi-character symbols in the alphabet.

The process of computation works in the following way:

1. If the word is the empty word ($\epsilon$) and the starting state is accepting, the automaton accepts
2. If the current state is accepting, the head reached the right endmarker, the automaton accepts
3. If the automaton enters a configuration that it has already been to, it means it entered a loop and should reject, if the automaton is created by the operation of complement, then it should accept
4. All possible transitions are obtained from the current state and symbol, if there are none, the automaton rejects
5. If there are some transitions possible, the computational process will branch as many times as is the count of the transitions
6. If any of the possible transitions lead to an accepting configuration, the automaton accepts
7. If no accepting configuration can be reached, it rejects the word

The computational process allows the user to see the process by remembering all the configurations. If the automaton is non-deterministic it stops the process when it reaches the first accepting configuration.

## 6.6 Operations

The operation on automata are implemented based on the algorithms and definitions in Section 4.

# 7 Sample application

A sample console application was created to test the library. It was written in C#, using the platform .NET 7.0. The app uses our library described in Section 6 to demonstrate

- loading from an XML file
- saving to an XML file
- computation on a word
- union of automata
- intersection of automata
- concatenation of automata
- square of an automaton
- complement of an automaton
- star of an automaton
- conversion of 2DFA to 1DFA
- conversion of 2NFA to 1DFA

As you can see, the functionality of the sample app is simple and dedicated to showing all aspects of the library.

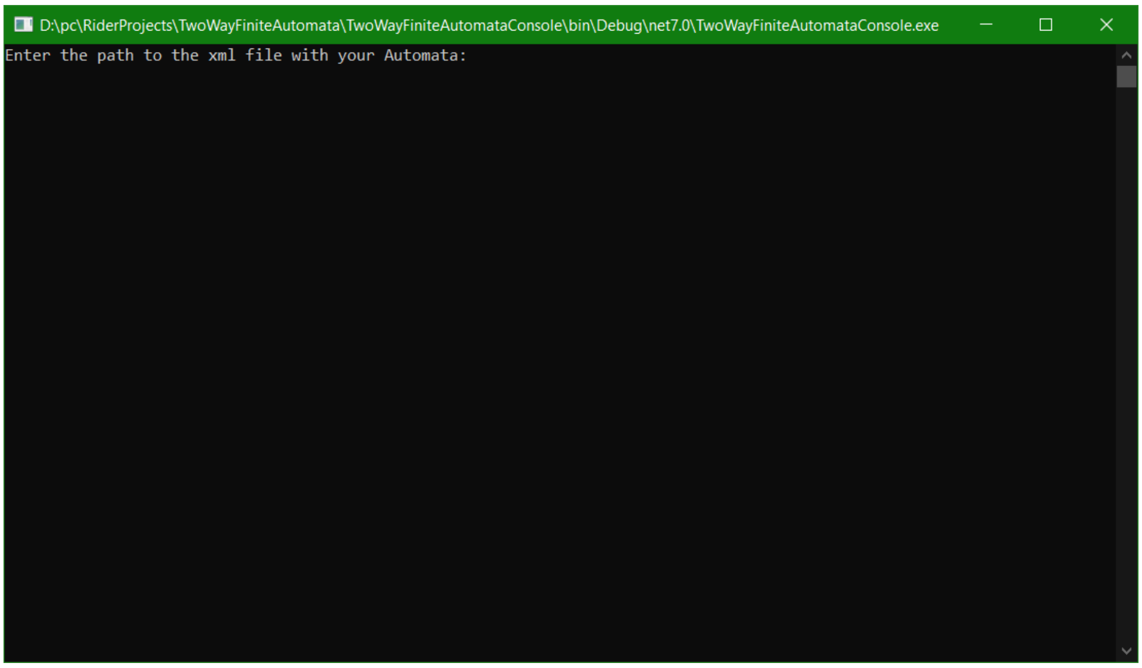
## 7.1 Walkthrough

First, prepare an XML file with your automata as shown in Section 6.3. There are several example files in an automata folder in the src folder of the electronic data included to this work that you can use.

### Loading automata

After starting the console app, a prompt to enter a path to an XML file with your automata pops on the screen (Figure 6). The path should be a full one, not a relative one.

For example: C:\Users\user\Desktop\automata.xml



```
D:\pc\RiderProjects\TwoWayFiniteAutomata\TwoWayFiniteAutomataConsole\bin\Debug\net7.0\TwoWayFiniteAutomataConsole.exe
Enter the path to the xml file with your Automata:
```

Figure 6: Loading automata from XML file

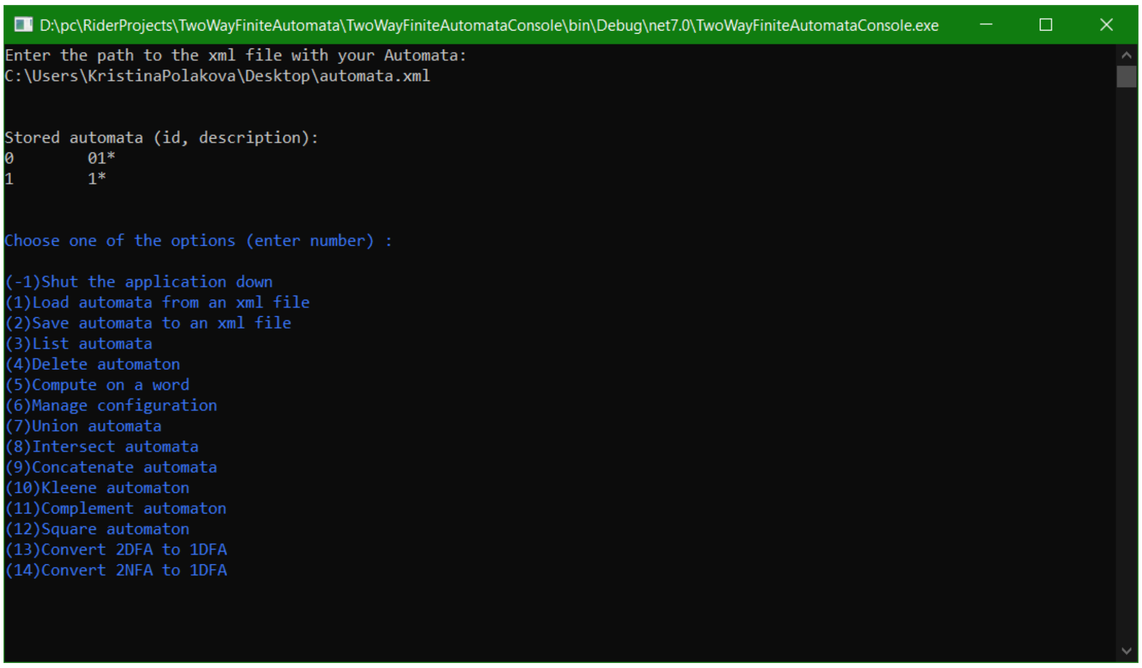
If the path is not correct, the file does not exist, or it contains an error, the automata cannot be parsed and an exception will be thrown and you will be prompted again to enter a path.

### Main menu

After entering the correct path the loaded automata are listed along with the main menu (Figure 7). The menu contains all possible actions by which a user can interact with the automata that were parsed from the user's XML file.

To choose one of the options, enter the number in the brackets written next to the option.

To return to the main menu, it is usually enough to enter "n" and press enter. It is prompted on all the places that it can be used.


D:\pc\RiderProjects\TwoWayFiniteAutomata\TwoWayFiniteAutomataConsole\bin\Debug\net7.0\TwoWayFiniteAutomataConsole.exe

```
Enter the path to the xml file with your Automata:
C:\Users\KristinaPolakova\Desktop\automata.xml


Stored automata (id, description):
0        01*
1        1*


Choose one of the options (enter number) :

(-1)Shut the application down
(1)Load automata from an xml file
(2)Save automata to an xml file
(3)List automata
(4)Delete automaton
(5)Compute on a word
(6)Manage configuration
(7)Union automata
(8)Intersect automata
(9)Concatenate automata
(10)Kleene automaton
(11)Complement automaton
(12)Square automaton
(13)Convert 2DFA to 1DFA
(14)Convert 2NFA to 1DFA
```

Figure 7: Main Menu

### Reloading automata

Option number 1 allows you to load automata from a file. The default file is the file you entered at the start of the application, or you can enter a path to a new file. If the file does not exist, it will be automatically created. If you decide to the same file you loaded the automata from, the file will be rewritten.

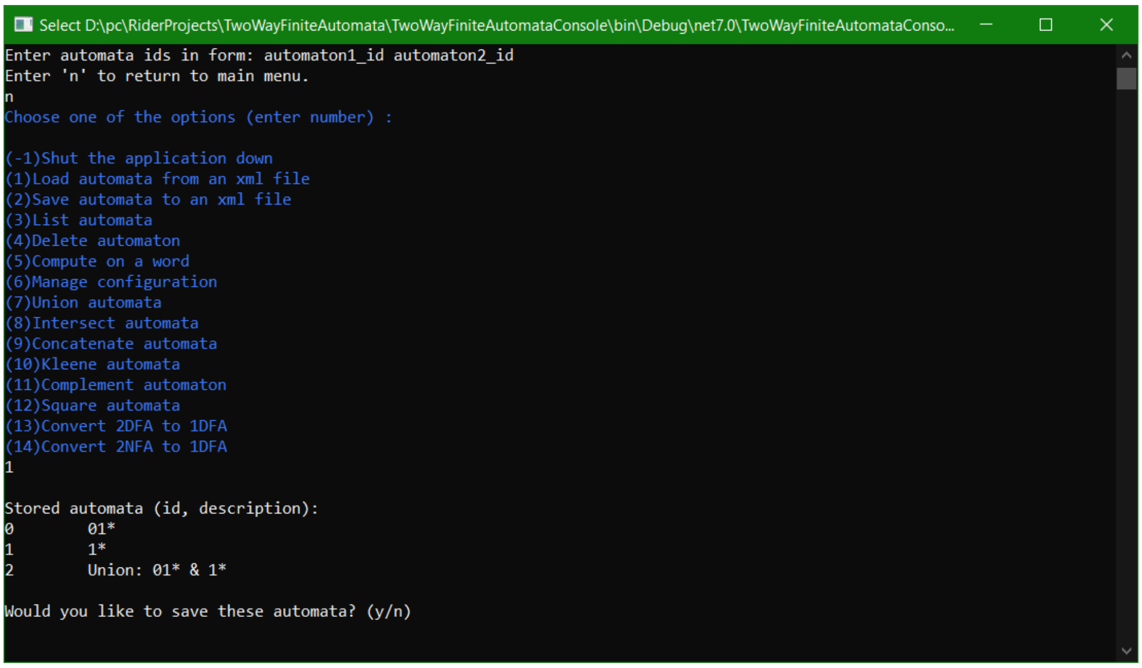
The app allows you to save automata to only one file.

After entering the option number, you get asked to enter a path to a file, where you want to save the automaton (Figure 8). If you do not enter anything, the automata will be saved to the original file.

Now let's imagine that we write "n" to the prompt, therefore no automata are saved.

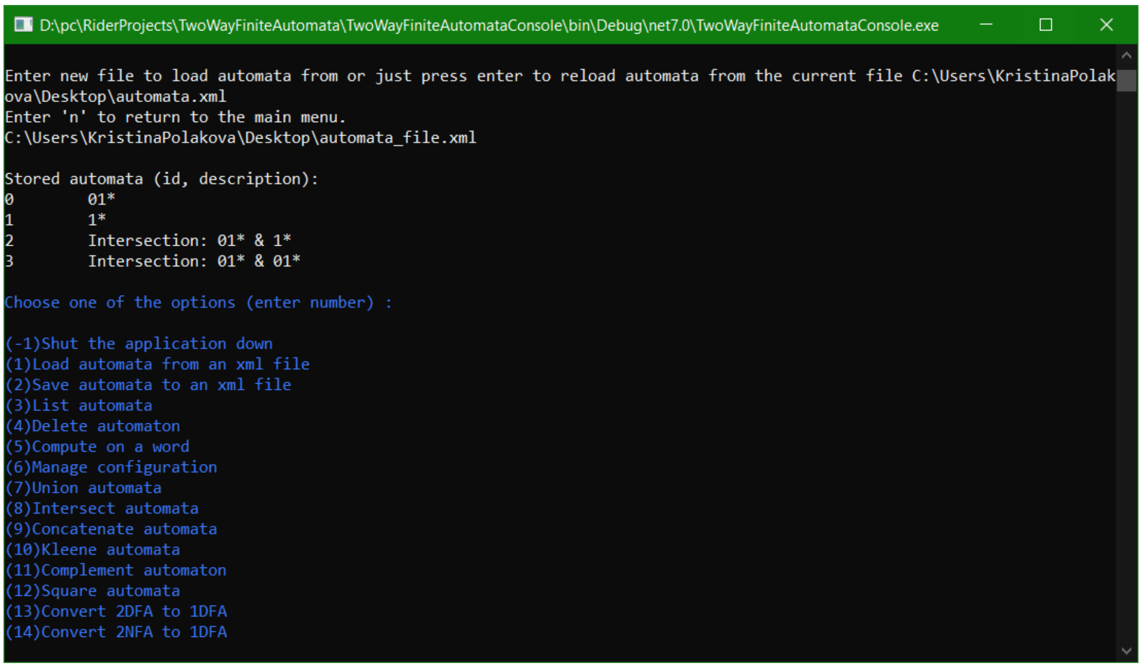
The process proceeds to loading and asking what file you want to load from (Figure 10). If you just press enter, the automata will be reloaded from the file you entered at the start of the application.

If you enter a path to a new file, automata will be loaded from there.


Select D:\pc\RiderProjects\TwoWayFiniteAutomata\TwoWayFiniteAutomataConsole\bin\Debug\net7.0\TwoWayFiniteAutomataConso...

```
Enter automata ids in form: automaton1_id automaton2_id
Enter 'n' to return to main menu.
n
Choose one of the options (enter number) :

(-1)Shut the application down
(1)Load automata from an xml file
(2)Save automata to an xml file
(3)List automata
(4)Delete automaton
(5)Compute on a word
(6)Manage configuration
(7)Union automata
(8)Intersect automata
(9)Concatenate automata
(10)Kleene automata
(11)Complement automaton
(12)Square automata
(13)Convert 2DFA to 1DFA
(14)Convert 2NFA to 1DFA
1

Stored automata (id, description):
0        01*
1        1*
2        Union: 01* & 1*

Would you like to save these automata? (y/n)
```

Figure 8: Reload automata from XML file


D:\pc\RiderProjects\TwoWayFiniteAutomata\TwoWayFiniteAutomataConsole\bin\Debug\net7.0\TwoWayFiniteAutomataConsole.exe

```
Enter new file to load automata from or just press enter to reload automata from the current file C:\Users\KristinaPolak
ova\Desktop\automata.xml
Enter 'n' to return to the main menu.
C:\Users\KristinaPolakova\Desktop\automata_file.xml

Stored automata (id, description):
0        01*
1        1*
2        Intersection: 01* & 1*
3        Intersection: 01* & 01*

Choose one of the options (enter number) :

(-1)Shut the application down
(1)Load automata from an xml file
(2)Save automata to an xml file
(3)List automata
(4)Delete automaton
(5)Compute on a word
(6)Manage configuration
(7)Union automata
(8)Intersect automata
(9)Concatenate automata
(10)Kleene automata
(11)Complement automaton
(12)Square automata
(13)Convert 2DFA to 1DFA
(14)Convert 2NFA to 1DFA
```

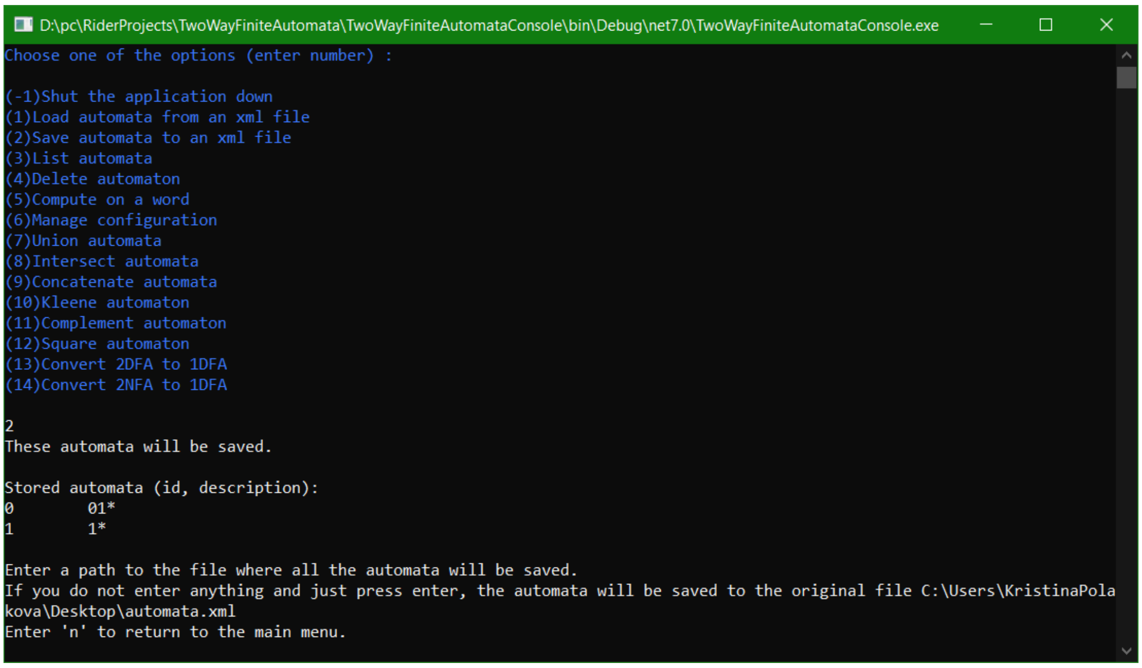
Figure 9: Reloaded automata from XML file

### Saving automata

Option number 2 allows the user to save the automata to a file. The default file is the file you entered at the start of the application or you can enter a path to a new file. If the file does not exist, it will be automatically created. If you decide to save to the same file you loaded the automata from, the file will be rewritten.

The app allows you to save automata to only one file.

After entering the option number, you get asked to enter a path to a file, where you want to save the automaton (Figure 8). If you do not enter anything, the automata will be saved to the original file.



```
D:\pc\RiderProjects\TwoWayFiniteAutomata\TwoWayFiniteAutomataConsole\bin\Debug\net7.0\TwoWayFiniteAutomataConsole.exe
Choose one of the options (enter number) :

(-1)Shut the application down
(1)Load automata from an xml file
(2)Save automata to an xml file
(3)List automata
(4)Delete automaton
(5)Compute on a word
(6)Manage configuration
(7)Union automata
(8)Intersect automata
(9)Concatenate automata
(10)Kleene automaton
(11)Complement automaton
(12)Square automaton
(13)Convert 2DFA to 1DFA
(14)Convert 2NFA to 1DFA

2
These automata will be saved.

Stored automata (id, description):
0        01*
1        1*

Enter a path to the file where all the automata will be saved.
If you do not enter anything and just press enter, the automata will be saved to the original file C:\Users\KristinaPola
kova\Desktop\automata.xml
Enter 'n' to return to the main menu.
```

Figure 10: Save automata to an XML file

When you enter a valid path, the confirmation of saving will be written on the screen (Figure 11).

### Listing Automata

Option number 3 allows the user to list all the loaded automata and show the details about them.

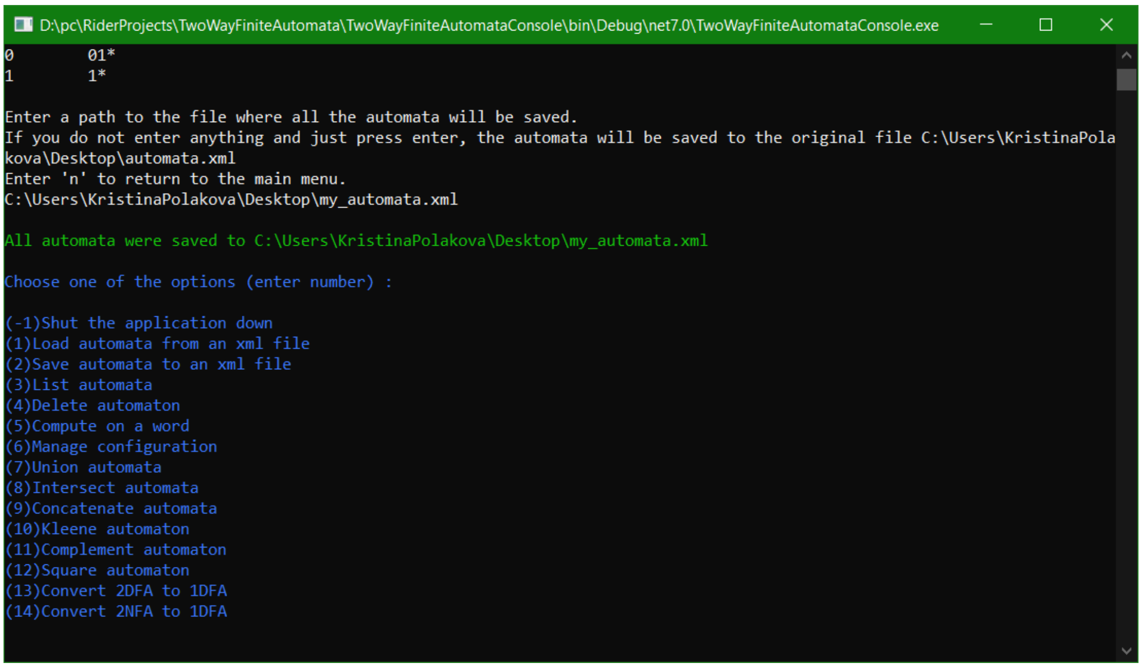
After entering the option number, the list of all the automata will appear including their id and description (Figure 12). If the description is too long, only some of it will be visible.

If you enter an id of a listed automaton, all the details will be shown (Figure 13).

### Deleting Automata

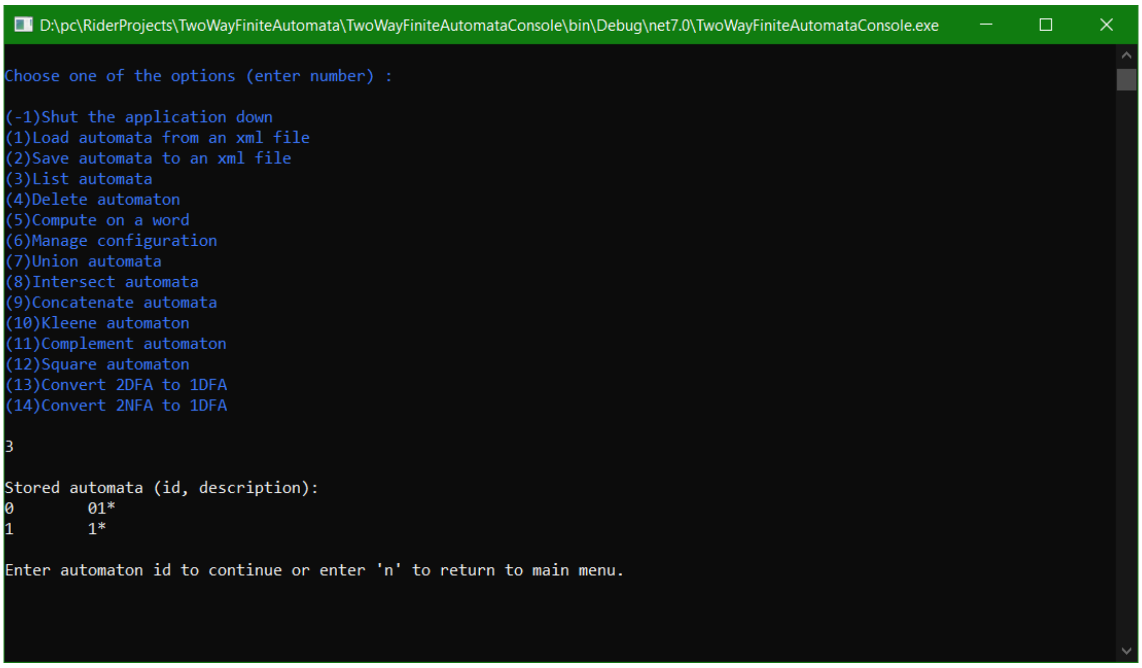
Option number 4 allows the user to delete an automaton from the loaded automata.

After entering the option number, the list of all the automata will appear (Figure 14).


D:\pc\RiderProjects\TwoWayFiniteAutomata\TwoWayFiniteAutomataConsole\bin\Debug\net7.0\TwoWayFiniteAutomataConsole.exe

```
0        01*
1        1*

Enter a path to the file where all the automata will be saved.
If you do not enter anything and just press enter, the automata will be saved to the original file C:\Users\KristinaPola
kova\Desktop\automata.xml
Enter 'n' to return to the main menu.
C:\Users\KristinaPolakova\Desktop\my_automata.xml

All automata were saved to C:\Users\KristinaPolakova\Desktop\my_automata.xml

Choose one of the options (enter number) :

(-1)Shut the application down
(1)Load automata from an xml file
(2)Save automata to an xml file
(3)List automata
(4)Delete automaton
(5)Compute on a word
(6)Manage configuration
(7)Union automata
(8)Intersect automata
(9)Concatenate automata
(10)Kleene automaton
(11)Complement automaton
(12)Square automaton
(13)Convert 2DFA to 1DFA
(14)Convert 2NFA to 1DFA
```

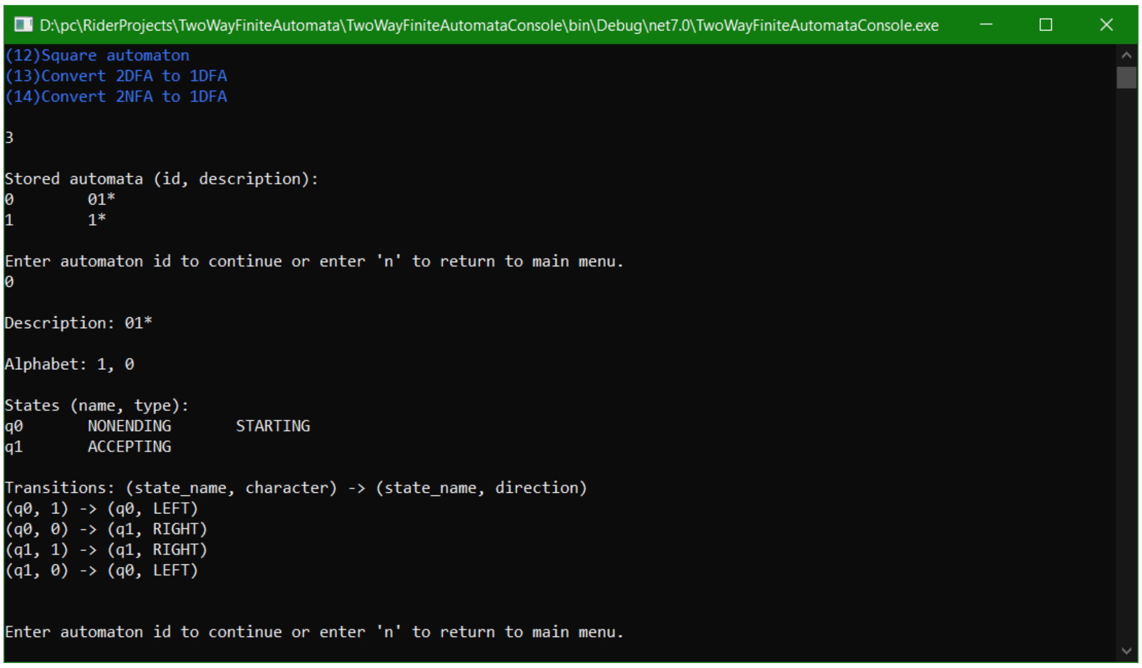
Figure 11: Saved automata to an XML file


D:\pc\RiderProjects\TwoWayFiniteAutomata\TwoWayFiniteAutomataConsole\bin\Debug\net7.0\TwoWayFiniteAutomataConsole.exe

```
Choose one of the options (enter number) :

(-1)Shut the application down
(1)Load automata from an xml file
(2)Save automata to an xml file
(3)List automata
(4)Delete automaton
(5)Compute on a word
(6)Manage configuration
(7)Union automata
(8)Intersect automata
(9)Concatenate automata
(10)Kleene automaton
(11)Complement automaton
(12)Square automaton
(13)Convert 2DFA to 1DFA
(14)Convert 2NFA to 1DFA

3

Stored automata (id, description):
0        01*
1        1*

Enter automaton id to continue or enter 'n' to return to main menu.
```

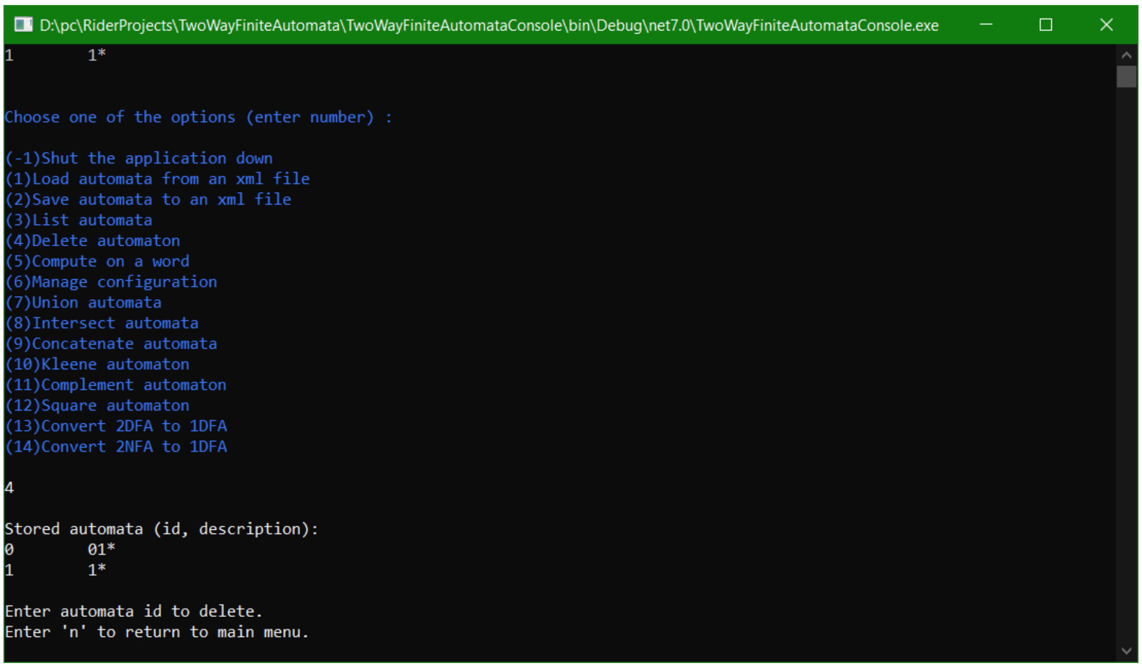
Figure 12: List all automata

If you enter an id of a listed automaton, this automaton will be deleted (Figure 15).



```
(12)Square automaton
(13)Convert 2DFA to 1DFA
(14)Convert 2NFA to 1DFA

3

Stored automata (id, description):
0        01*
1        1*

Enter automaton id to continue or enter 'n' to return to main menu.
0

Description: 01*

Alphabet: 1, 0

States (name, type):
q0       NONENDING       STARTING
q1       ACCEPTING

Transitions: (state_name, character) -> (state_name, direction)
(q0, 1) -> (q0, LEFT)
(q0, 0) -> (q1, RIGHT)
(q1, 1) -> (q1, RIGHT)
(q1, 0) -> (q0, LEFT)


Enter automaton id to continue or enter 'n' to return to main menu.
```

Figure 13: Detail of an automaton



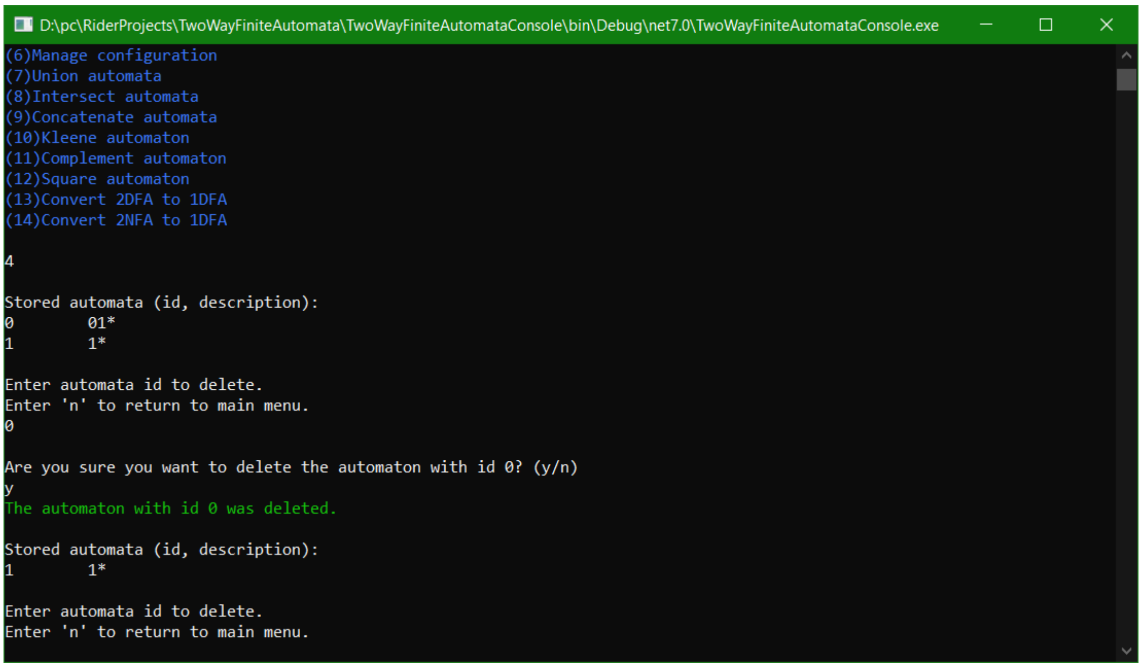
```
1        1*


Choose one of the options (enter number) :

(-1)Shut the application down
(1)Load automata from an xml file
(2)Save automata to an xml file
(3)List automata
(4)Delete automaton
(5)Compute on a word
(6)Manage configuration
(7)Union automata
(8)Intersect automata
(9)Concatenate automata
(10)Kleene automaton
(11)Complement automaton
(12)Square automaton
(13)Convert 2DFA to 1DFA
(14)Convert 2NFA to 1DFA

4

Stored automata (id, description):
0        01*
1        1*

Enter automata id to delete.
Enter 'n' to return to main menu.
```

Figure 14: List all automata to delete from

## Compute on an Input Word

Option number 5 allows you to enter an input word to an automata and find out whether it belongs to its language or not.

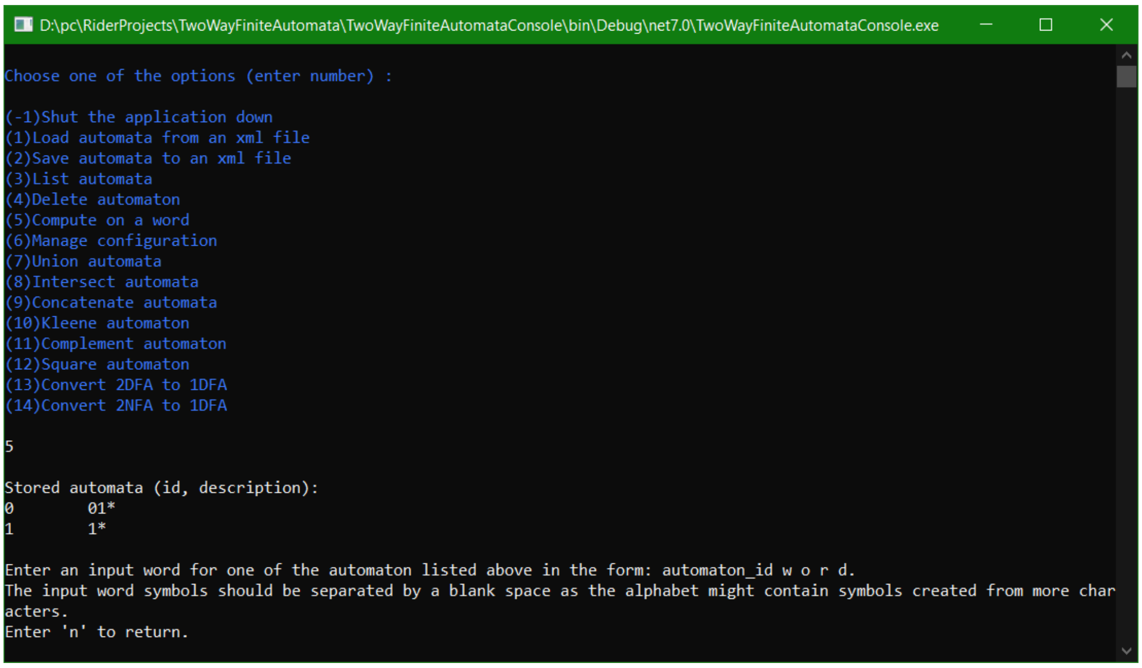
After entering the option number, the list of all automata will appear (Figure


D:\pc\RiderProjects\TwoWayFiniteAutomata\TwoWayFiniteAutomataConsole\bin\Debug\net7.0\TwoWayFiniteAutomataConsole.exe

```
(6)Manage configuration
(7)Union automata
(8)Intersect automata
(9)Concatenate automata
(10)Kleene automaton
(11)Complement automaton
(12)Square automaton
(13)Convert 2DFA to 1DFA
(14)Convert 2NFA to 1DFA

4

Stored automata (id, description):
0        01*
1        1*

Enter automata id to delete.
Enter 'n' to return to main menu.
0

Are you sure you want to delete the automaton with id 0? (y/n)
y
The automaton with id 0 was deleted.

Stored automata (id, description):
1        1*

Enter automata id to delete.
Enter 'n' to return to main menu.
```

Figure 15: Automaton deleted

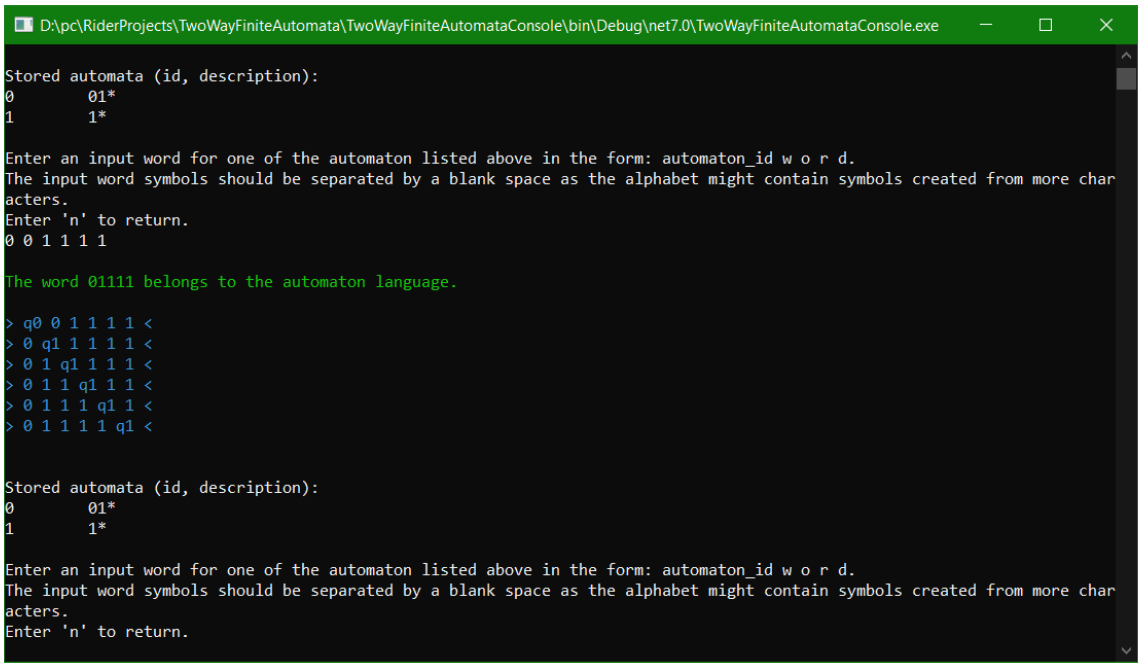
16).


D:\pc\RiderProjects\TwoWayFiniteAutomata\TwoWayFiniteAutomataConsole\bin\Debug\net7.0\TwoWayFiniteAutomataConsole.exe

```

Choose one of the options (enter number) :

(-1)Shut the application down
(1)Load automata from an xml file
(2)Save automata to an xml file
(3)List automata
(4)Delete automaton
(5)Compute on a word
(6)Manage configuration
(7)Union automata
(8)Intersect automata
(9)Concatenate automata
(10)Kleene automaton
(11)Complement automaton
(12)Square automaton
(13)Convert 2DFA to 1DFA
(14)Convert 2NFA to 1DFA

5

Stored automata (id, description):
0        01*
1        1*

Enter an input word for one of the automaton listed above in the form: automaton_id w o r d.
The input word symbols should be separated by a blank space as the alphabet might contain symbols created from more char
acters.
Enter 'n' to return.
```

Figure 16: List of automata to compute on

If you enter an automaton id and a word, it will output whether it belongs to the language or not (Figure 17). It might also print a list of configurations of the computation process. You can turn the configuration printing on and off from the main menu in option number 6. By default, configurations get printed.



D:\pc\RiderProjects\TwoWayFiniteAutomata\TwoWayFiniteAutomataConsole\bin\Debug\net7.0\TwoWayFiniteAutomataConsole.exe

```
Stored automata (id, description):
0        01*
1        1*

Enter an input word for one of the automaton listed above in the form: automaton_id w o r d.
The input word symbols should be separated by a blank space as the alphabet might contain symbols created from more char
acters.
Enter 'n' to return.
0 0 1 1 1 1

The word 01111 belongs to the automaton language.

> q0 0 1 1 1 1 <
> 0 q1 1 1 1 1 <
> 0 1 q1 1 1 1 <
> 0 1 1 q1 1 1 <
> 0 1 1 1 q1 1 <
> 0 1 1 1 1 q1 <


Stored automata (id, description):
0        01*
1        1*

Enter an input word for one of the automaton listed above in the form: automaton_id w o r d.
The input word symbols should be separated by a blank space as the alphabet might contain symbols created from more char
acters.
Enter 'n' to return.
```

Figure 17: Computation on a Word

### Manage Configuration

Option number 6 allows you to either turn the configuration printing on or off as mentioned in Section 7.1. By default, the configuration is on.

After entering the option number, information about configuration settings is shown on the screen (Figure 18).

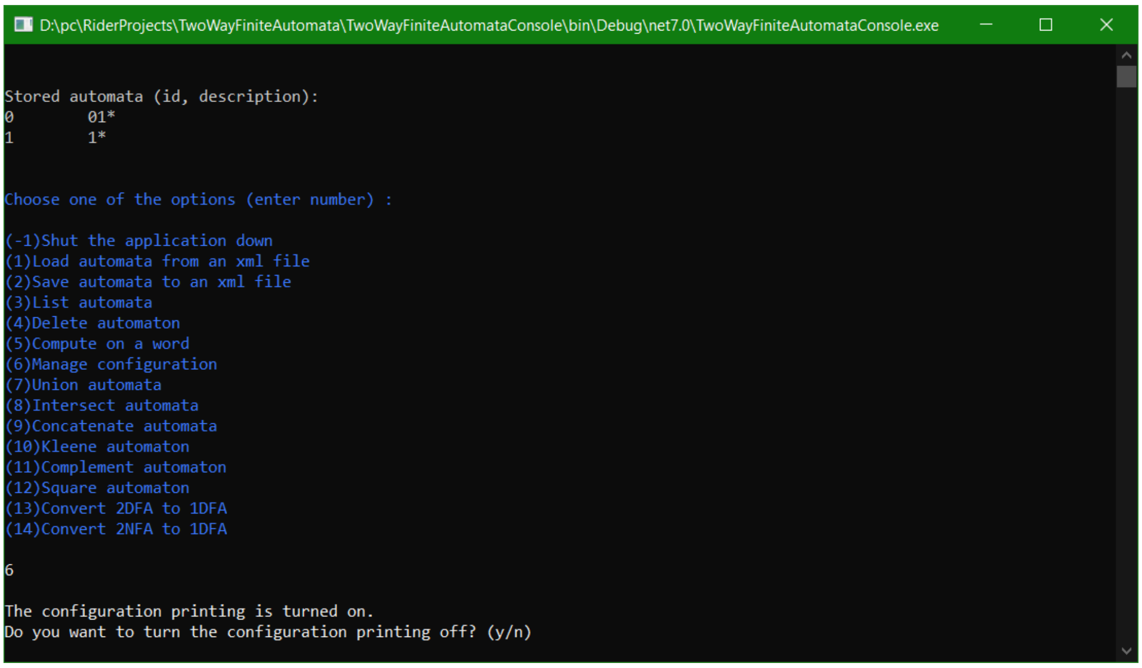
If it is turned on, you are asked if you want to turn it off. If it is off, you are asked if you want to turn it on. In both cases you simply print "y" if you want to perform the action, if anything else is entered, the settings will stay the same (Figure 19).

### Union

Option number 7 allows you to union two of your automata.

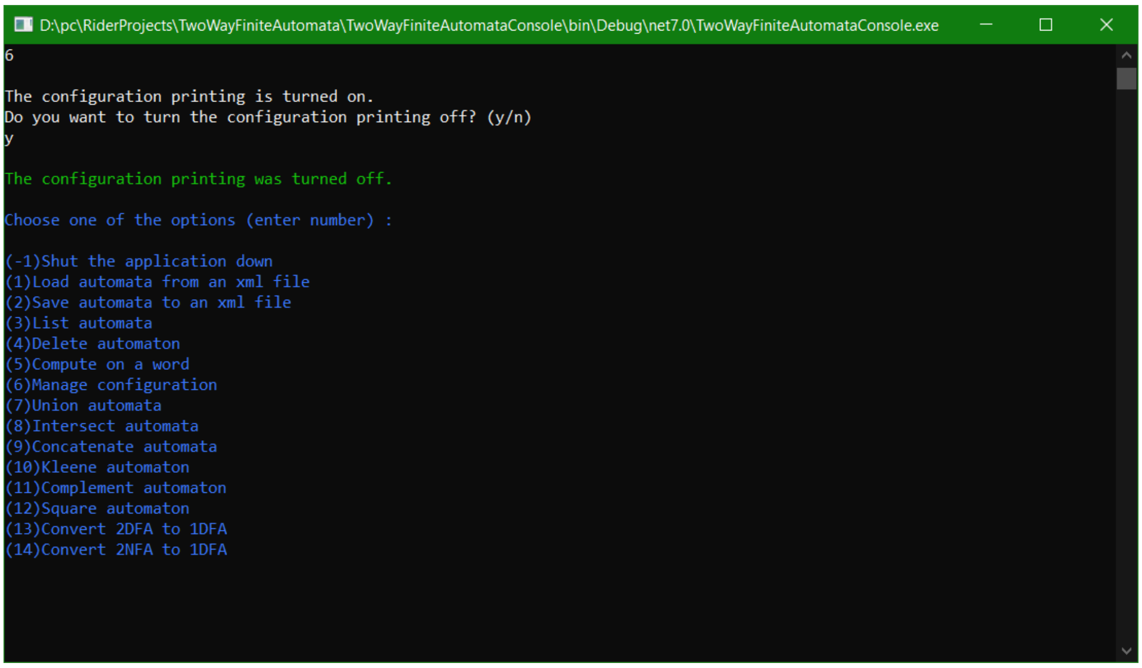
After entering the option number, the list of all automata will appear (Figure 20).

If you enter the ids of two automata from the list, they will be united and the created automaton will be added to the list and you can interact with it in other operations too (Figure 21).


D:\pc\RiderProjects\TwoWayFiniteAutomata\TwoWayFiniteAutomataConsole\bin\Debug\net7.0\TwoWayFiniteAutomataConsole.exe

```
Stored automata (id, description):
0        01*
1        1*


Choose one of the options (enter number) :

(-1)Shut the application down
(1)Load automata from an xml file
(2)Save automata to an xml file
(3)List automata
(4)Delete automaton
(5)Compute on a word
(6)Manage configuration
(7)Union automata
(8)Intersect automata
(9)Concatenate automata
(10)Kleene automaton
(11)Complement automaton
(12)Square automaton
(13)Convert 2DFA to 1DFA
(14)Convert 2NFA to 1DFA

6

The configuration printing is turned on.
Do you want to turn the configuration printing off? (y/n)
```

Figure 18: List of automata to compute on

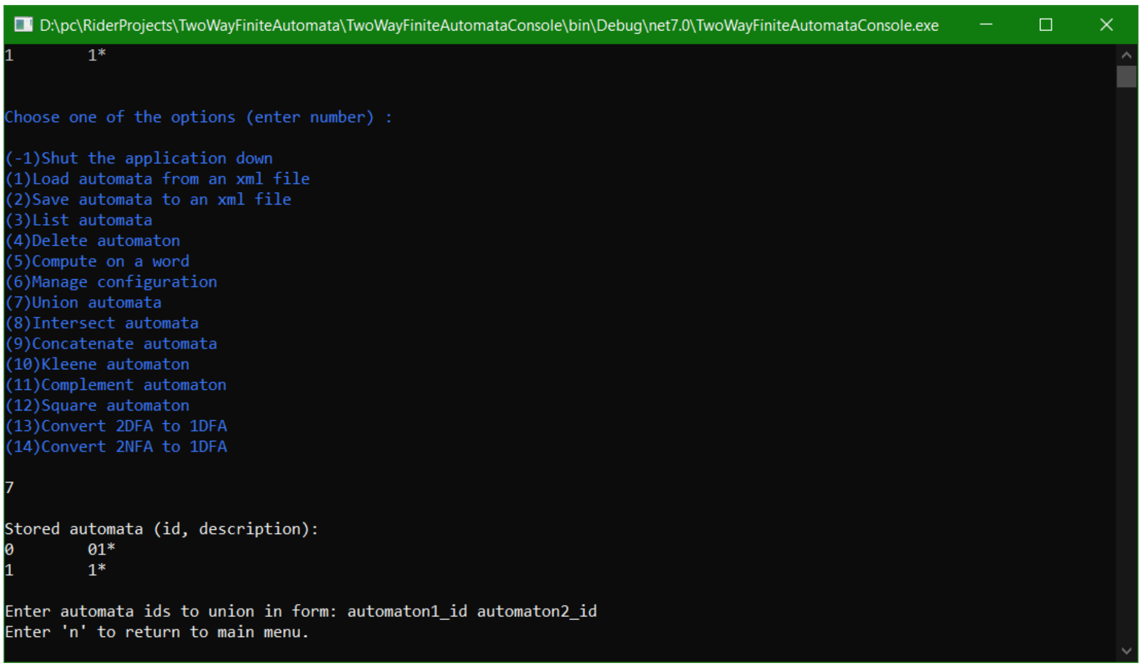

D:\pc\RiderProjects\TwoWayFiniteAutomata\TwoWayFiniteAutomataConsole\bin\Debug\net7.0\TwoWayFiniteAutomataConsole.exe

```
6

The configuration printing is turned on.
Do you want to turn the configuration printing off? (y/n)
y

The configuration printing was turned off.

Choose one of the options (enter number) :

(-1)Shut the application down
(1)Load automata from an xml file
(2)Save automata to an xml file
(3)List automata
(4)Delete automaton
(5)Compute on a word
(6)Manage configuration
(7)Union automata
(8)Intersect automata
(9)Concatenate automata
(10)Kleene automaton
(11)Complement automaton
(12)Square automaton
(13)Convert 2DFA to 1DFA
(14)Convert 2NFA to 1DFA
```

Figure 19: Configuration printing turned off

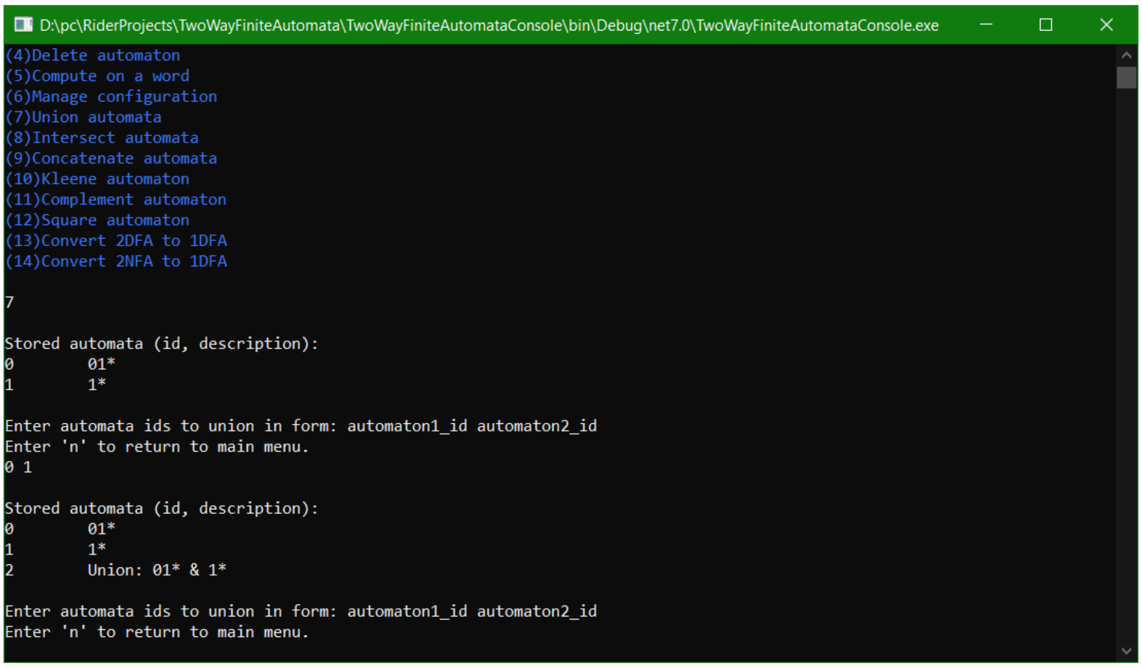
### Intersection

Option number 8 allows you to perform an intersection of two of your automata.
After entering the option number, the list of all automata will appear (Figure 22).


D:\pc\RiderProjects\TwoWayFiniteAutomata\TwoWayFiniteAutomataConsole\bin\Debug\net7.0\TwoWayFiniteAutomataConsole.exe

```
1        1*


Choose one of the options (enter number) :

(-1)Shut the application down
(1)Load automata from an xml file
(2)Save automata to an xml file
(3)List automata
(4)Delete automaton
(5)Compute on a word
(6)Manage configuration
(7)Union automata
(8)Intersect automata
(9)Concatenate automata
(10)Kleene automaton
(11)Complement automaton
(12)Square automaton
(13)Convert 2DFA to 1DFA
(14)Convert 2NFA to 1DFA

7

Stored automata (id, description):
0        01*
1        1*

Enter automata ids to union in form: automaton1_id automaton2_id
Enter 'n' to return to main menu.
```

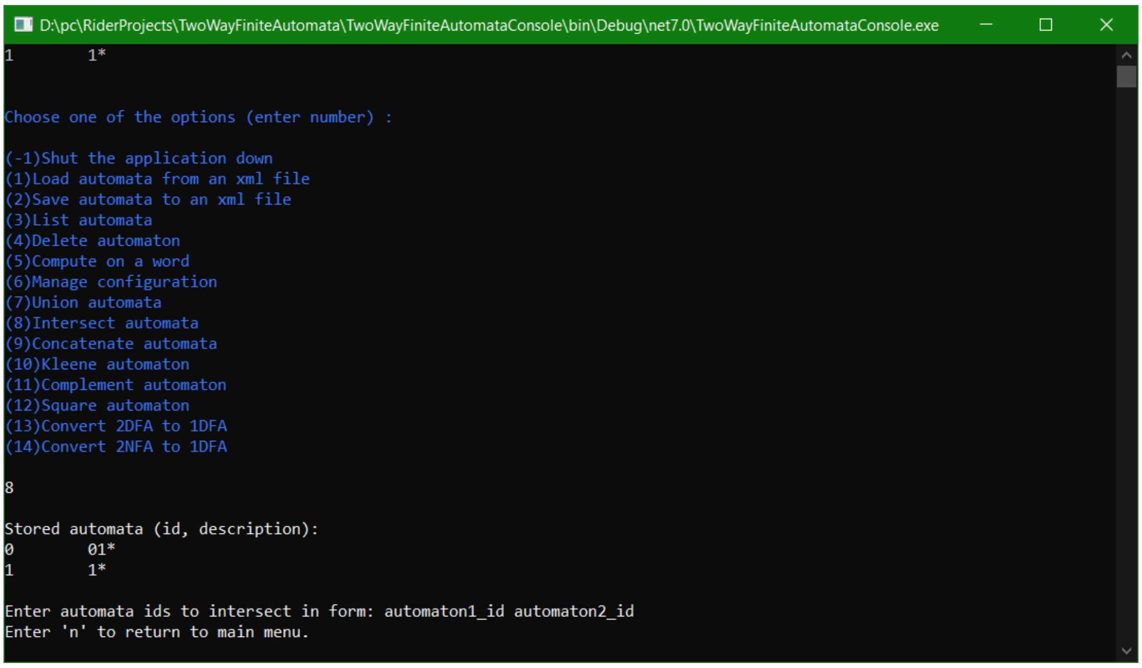
Figure 20: List of automata to union


D:\pc\RiderProjects\TwoWayFiniteAutomata\TwoWayFiniteAutomataConsole\bin\Debug\net7.0\TwoWayFiniteAutomataConsole.exe

```
(4)Delete automaton
(5)Compute on a word
(6)Manage configuration
(7)Union automata
(8)Intersect automata
(9)Concatenate automata
(10)Kleene automaton
(11)Complement automaton
(12)Square automaton
(13)Convert 2DFA to 1DFA
(14)Convert 2NFA to 1DFA

7

Stored automata (id, description):
0        01*
1        1*

Enter automata ids to union in form: automaton1_id automaton2_id
Enter 'n' to return to main menu.
0 1

Stored automata (id, description):
0        01*
1        1*
2        Union: 01* & 1*

Enter automata ids to union in form: automaton1_id automaton2_id
Enter 'n' to return to main menu.
```

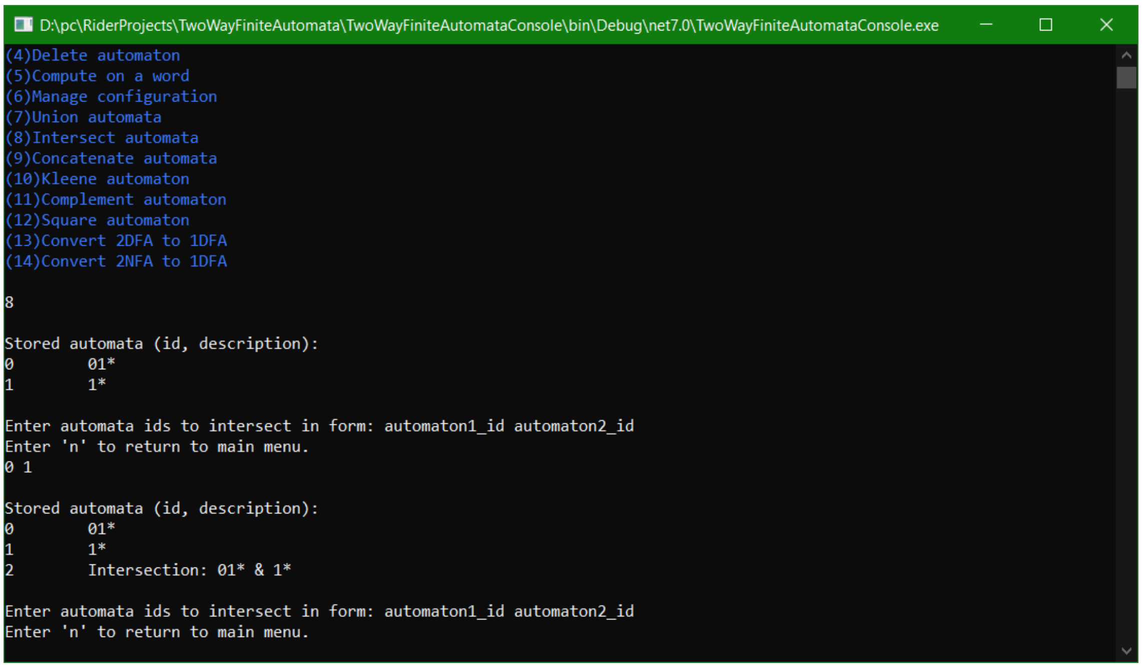
Figure 21: Automata united

If you enter the ids of two automata from the list, they will be intersected and the created automaton will be added to the list 23.



```
D:\pc\RiderProjects\TwoWayFiniteAutomata\TwoWayFiniteAutomataConsole\bin\Debug\net7.0\TwoWayFiniteAutomataConsole.exe
1        1*


Choose one of the options (enter number) :

(-1)Shut the application down
(1)Load automata from an xml file
(2)Save automata to an xml file
(3)List automata
(4)Delete automaton
(5)Compute on a word
(6)Manage configuration
(7)Union automata
(8)Intersect automata
(9)Concatenate automata
(10)Kleene automaton
(11)Complement automaton
(12)Square automaton
(13)Convert 2DFA to 1DFA
(14)Convert 2NFA to 1DFA

8

Stored automata (id, description):
0        01*
1        1*

Enter automata ids to intersect in form: automaton1_id automaton2_id
Enter 'n' to return to main menu.
```

Figure 22: List of automata to interesect



```
D:\pc\RiderProjects\TwoWayFiniteAutomata\TwoWayFiniteAutomataConsole\bin\Debug\net7.0\TwoWayFiniteAutomataConsole.exe
(4)Delete automaton
(5)Compute on a word
(6)Manage configuration
(7)Union automata
(8)Intersect automata
(9)Concatenate automata
(10)Kleene automaton
(11)Complement automaton
(12)Square automaton
(13)Convert 2DFA to 1DFA
(14)Convert 2NFA to 1DFA

8

Stored automata (id, description):
0        01*
1        1*

Enter automata ids to intersect in form: automaton1_id automaton2_id
Enter 'n' to return to main menu.
0 1

Stored automata (id, description):
0        01*
1        1*
2        Intersection: 01* & 1*

Enter automata ids to intersect in form: automaton1_id automaton2_id
Enter 'n' to return to main menu.
```

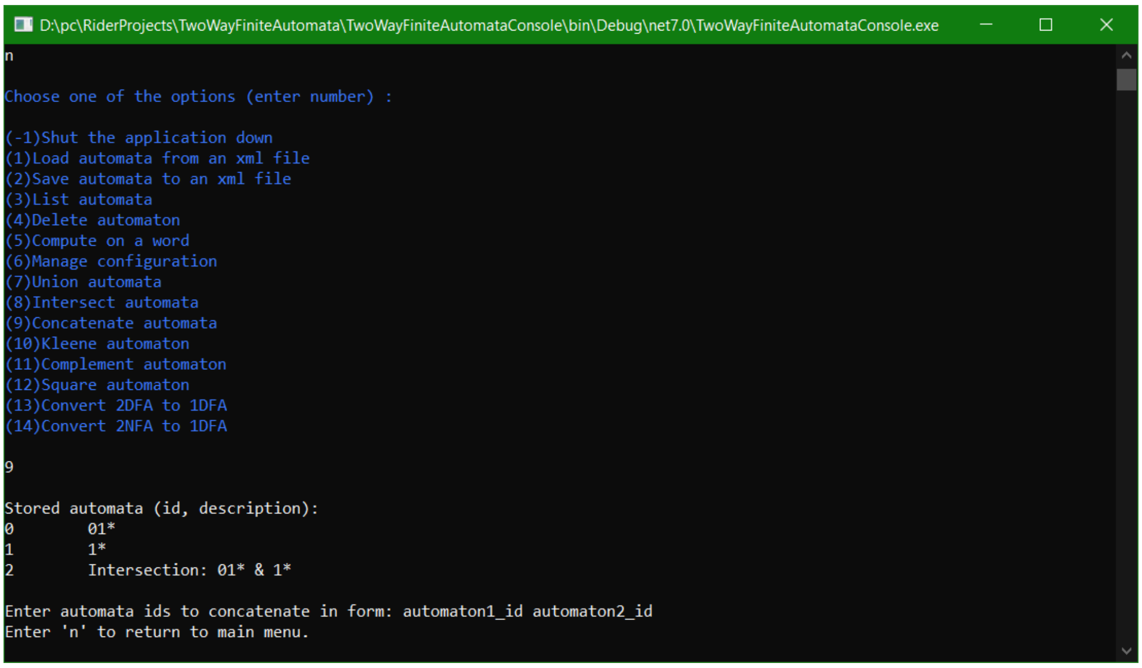
Figure 23: Intersected automata

### Concatenate

Option number 9 allows you to perform the concatenation of two of your automata.

After entering the option number, the list of all automata will appear (Figure

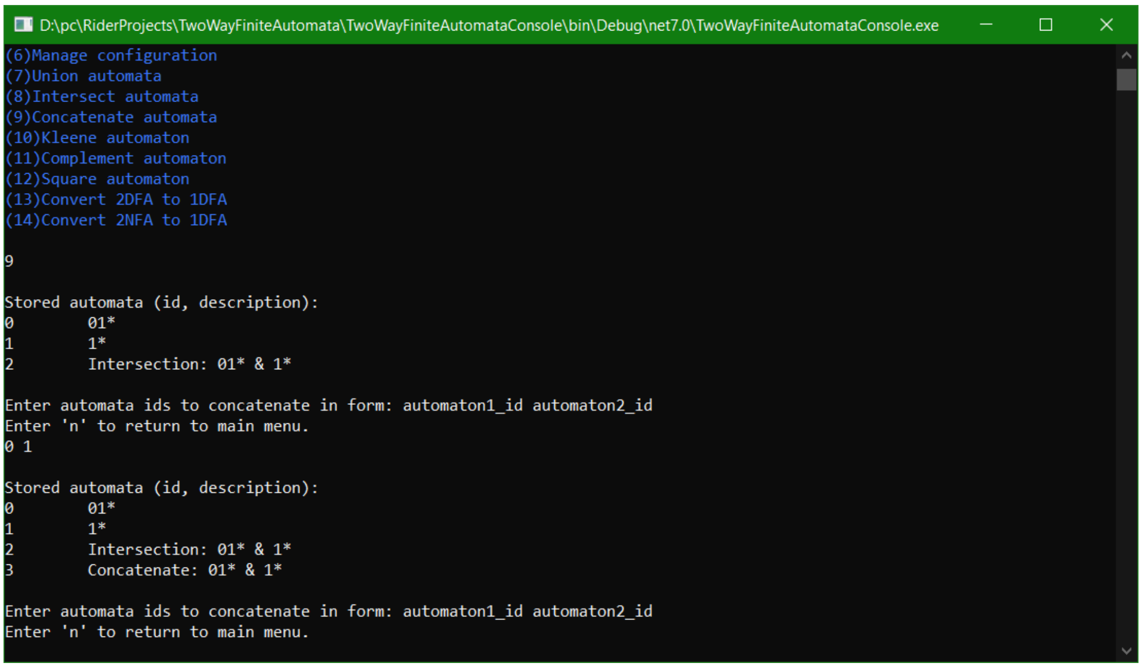
24).


D:\pc\RiderProjects\TwoWayFiniteAutomata\TwoWayFiniteAutomataConsole\bin\Debug\net7.0\TwoWayFiniteAutomataConsole.exe

```
n

Choose one of the options (enter number) :

(-1)Shut the application down
(1)Load automata from an xml file
(2)Save automata to an xml file
(3)List automata
(4)Delete automaton
(5)Compute on a word
(6)Manage configuration
(7)Union automata
(8)Intersect automata
(9)Concatenate automata
(10)Kleene automaton
(11)Complement automaton
(12)Square automaton
(13)Convert 2DFA to 1DFA
(14)Convert 2NFA to 1DFA

9

Stored automata (id, description):
0        01*
1        1*
2        Intersection: 01* & 1*

Enter automata ids to concatenate in form: automaton1_id automaton2_id
Enter 'n' to return to main menu.
```

Figure 24: List of automata to concatenate

If you enter the ids of two automata from the list, they will be concatenated and the created automaton will be added to the list (Figure 25).


D:\pc\RiderProjects\TwoWayFiniteAutomata\TwoWayFiniteAutomataConsole\bin\Debug\net7.0\TwoWayFiniteAutomataConsole.exe

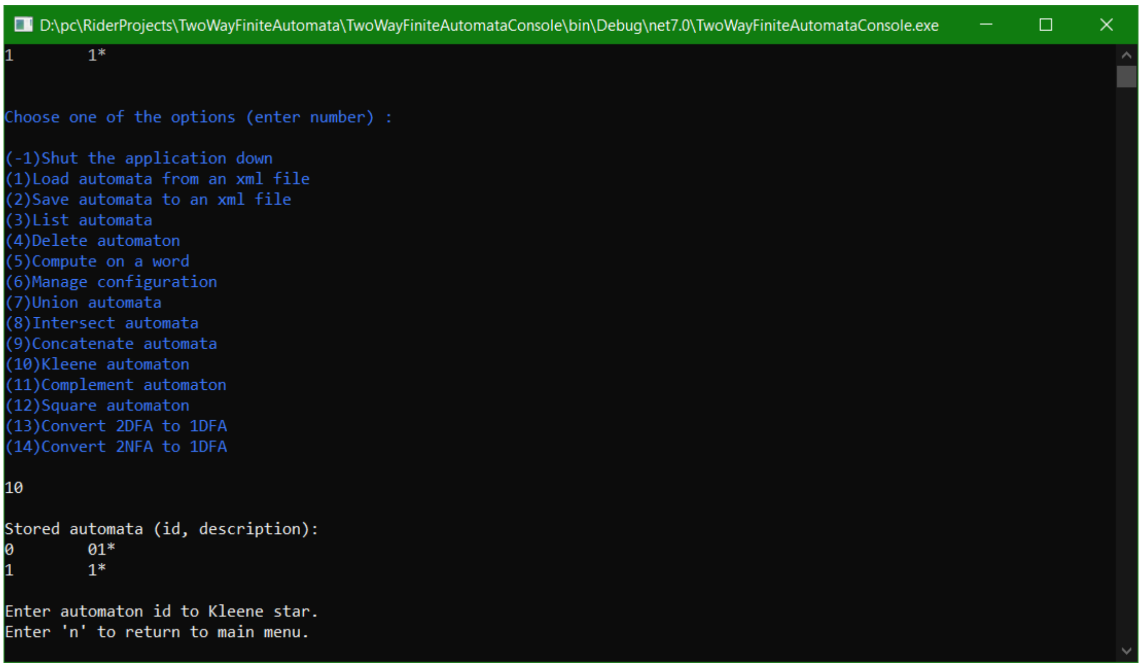
```
(6)Manage configuration
(7)Union automata
(8)Intersect automata
(9)Concatenate automata
(10)Kleene automaton
(11)Complement automaton
(12)Square automaton
(13)Convert 2DFA to 1DFA
(14)Convert 2NFA to 1DFA

9

Stored automata (id, description):
0        01*
1        1*
2        Intersection: 01* & 1*

Enter automata ids to concatenate in form: automaton1_id automaton2_id
Enter 'n' to return to main menu.
0 1

Stored automata (id, description):
0        01*
1        1*
2        Intersection: 01* & 1*
3        Concatenate: 01* & 1*

Enter automata ids to concatenate in form: automaton1_id automaton2_id
Enter 'n' to return to main menu.
```

Figure 25: Concatenated automata

### Kleene Star

Option number 10 allows you to perform the star of any of your automata.

After entering the option number, the list of all automata will appear (Figure 26).



D:\pc\RiderProjects\TwoWayFiniteAutomata\TwoWayFiniteAutomataConsole\bin\Debug\net7.0\TwoWayFiniteAutomataConsole.exe

```
1       1*


Choose one of the options (enter number) :

(-1)Shut the application down
(1)Load automata from an xml file
(2)Save automata to an xml file
(3)List automata
(4)Delete automaton
(5)Compute on a word
(6)Manage configuration
(7)Union automata
(8)Intersect automata
(9)Concatenate automata
(10)Kleene automaton
(11)Complement automaton
(12)Square automaton
(13)Convert 2DFA to 1DFA
(14)Convert 2NFA to 1DFA

10

Stored automata (id, description):
0       01*
1       1*

Enter automaton id to Kleene star.
Enter 'n' to return to main menu.
```

Figure 26: List of automata to star

If you enter an id of an automaton from the list, it will be Kleene stared and the created automaton will be added to the list (Figure 27).

### Complement

Option number 11 allows you to create a complement of any of your automata.

After entering the option number, the list of all automata will appear (Figure 28).

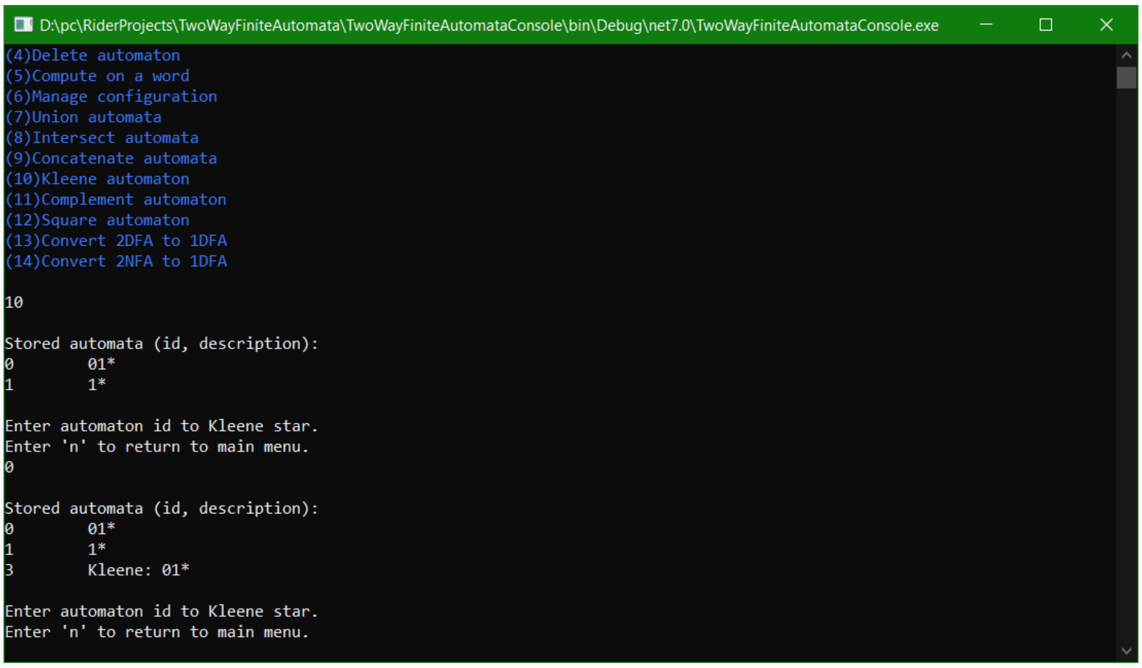
If you enter an id of one of your automaton from the list, it will be complemented and the created automaton will be added to the list (Figure 29).

### Square

Option number 12 allows you to square any of your automata.

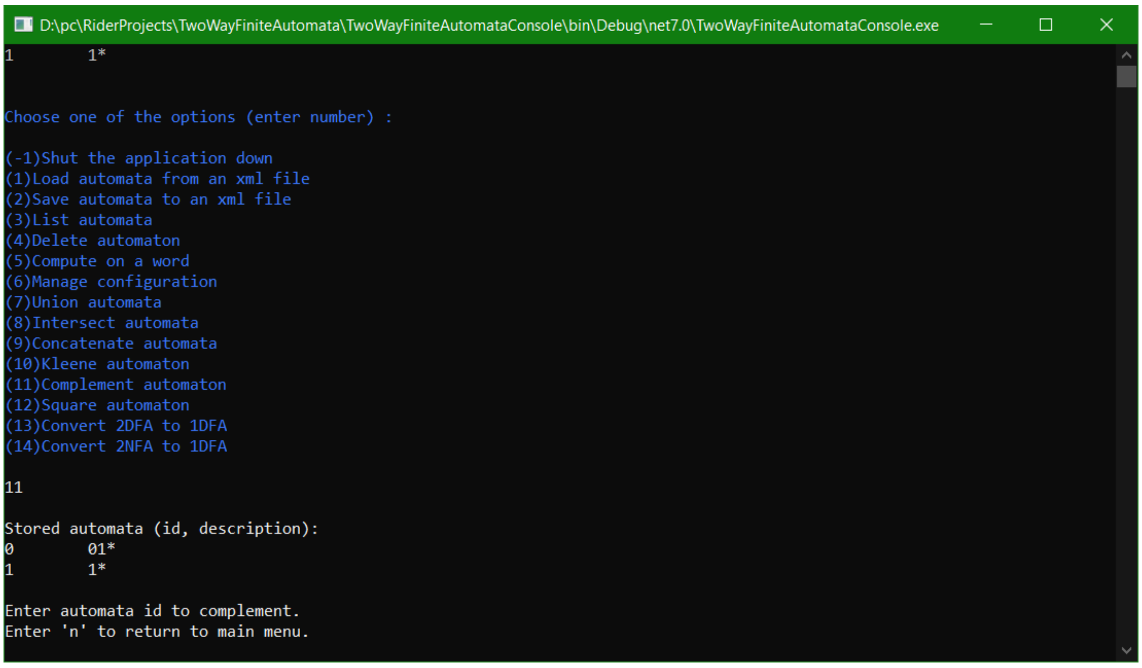
After entering the option number, the list of all automata will appear (Figure 30).

If you enter an id of one of your automaton from the list and a number of squares to create, it will be squared and the created automaton will be added to the list (Figure 31).



```
D:\pc\RiderProjects\TwoWayFiniteAutomata\TwoWayFiniteAutomataConsole\bin\Debug\net7.0\TwoWayFiniteAutomataConsole.exe
(4)Delete automaton
(5)Compute on a word
(6)Manage configuration
(7)Union automata
(8)Intersect automata
(9)Concatenate automata
(10)Kleene automaton
(11)Complement automaton
(12)Square automaton
(13)Convert 2DFA to 1DFA
(14)Convert 2NFA to 1DFA

10

Stored automata (id, description):
0        01*
1        1*

Enter automaton id to Kleene star.
Enter 'n' to return to main menu.
0

Stored automata (id, description):
0        01*
1        1*
3        Kleene: 01*

Enter automaton id to Kleene star.
Enter 'n' to return to main menu.
```

Figure 27: Concatenated automata



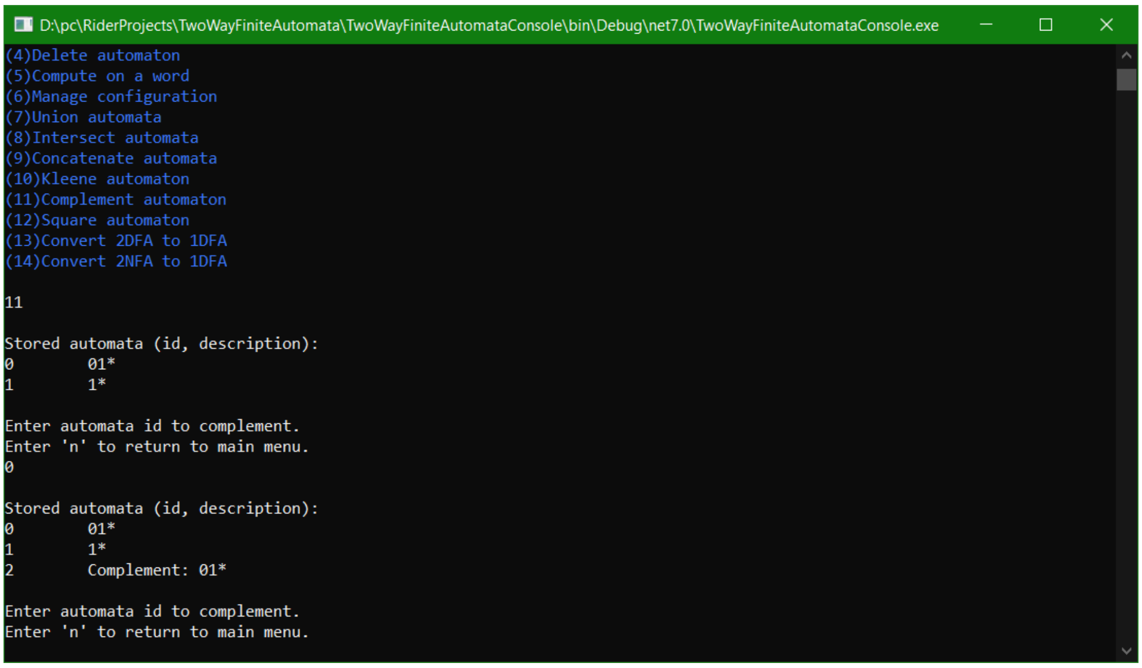
```
D:\pc\RiderProjects\TwoWayFiniteAutomata\TwoWayFiniteAutomataConsole\bin\Debug\net7.0\TwoWayFiniteAutomataConsole.exe
1        1*


Choose one of the options (enter number) :

(-1)Shut the application down
(1)Load automata from an xml file
(2)Save automata to an xml file
(3)List automata
(4)Delete automaton
(5)Compute on a word
(6)Manage configuration
(7)Union automata
(8)Intersect automata
(9)Concatenate automata
(10)Kleene automaton
(11)Complement automaton
(12)Square automaton
(13)Convert 2DFA to 1DFA
(14)Convert 2NFA to 1DFA

11

Stored automata (id, description):
0        01*
1        1*

Enter automata id to complement.
Enter 'n' to return to main menu.
```

Figure 28: List of automata to complement

## Convert 2DFA to 1DFA

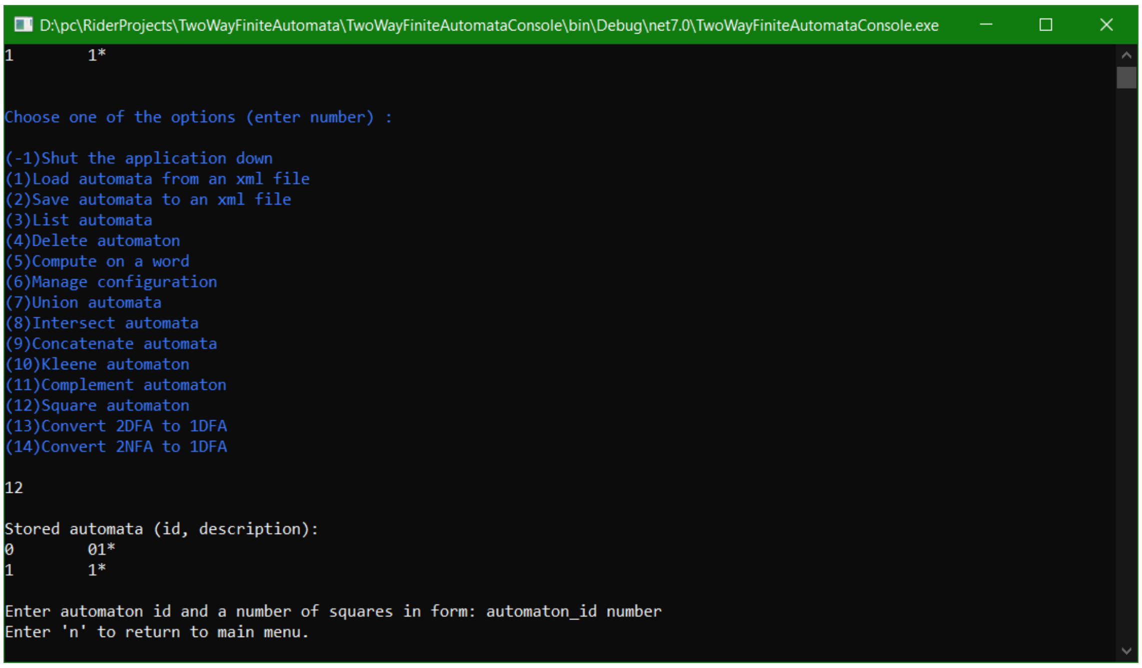
Option number 13 allows you to perform a conversion of any of your automata that are 2DFA.

After entering the option number, the list of all 2DFAs will appear (Figure



```
(4)Delete automaton
(5)Compute on a word
(6)Manage configuration
(7)Union automata
(8)Intersect automata
(9)Concatenate automata
(10)Kleene automaton
(11)Complement automaton
(12)Square automaton
(13)Convert 2DFA to 1DFA
(14)Convert 2NFA to 1DFA

11

Stored automata (id, description):
0        01*
1        1*

Enter automata id to complement.
Enter 'n' to return to main menu.
0

Stored automata (id, description):
0        01*
1        1*
2        Complement: 01*

Enter automata id to complement.
Enter 'n' to return to main menu.
```

Figure 29: Complemented automaton

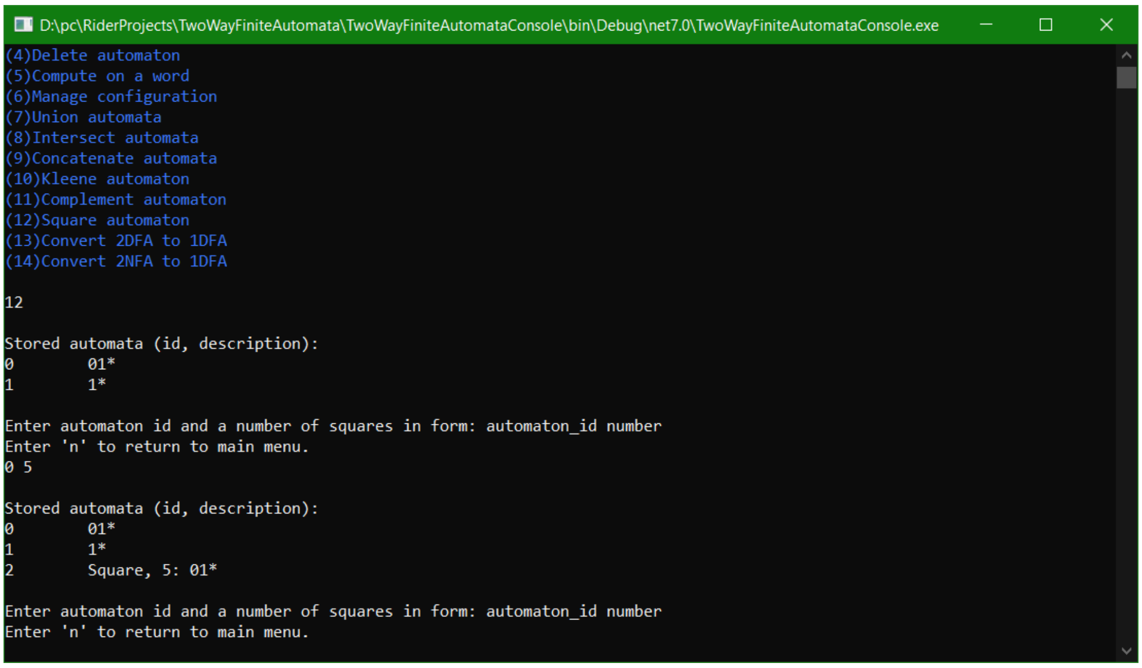


```
1        1*


Choose one of the options (enter number) :

(-1)Shut the application down
(1)Load automata from an xml file
(2)Save automata to an xml file
(3)List automata
(4)Delete automaton
(5)Compute on a word
(6)Manage configuration
(7)Union automata
(8)Intersect automata
(9)Concatenate automata
(10)Kleene automaton
(11)Complement automaton
(12)Square automaton
(13)Convert 2DFA to 1DFA
(14)Convert 2NFA to 1DFA

12

Stored automata (id, description):
0        01*
1        1*

Enter automaton id and a number of squares in form: automaton_id number
Enter 'n' to return to main menu.
```

Figure 30: List of automata to square

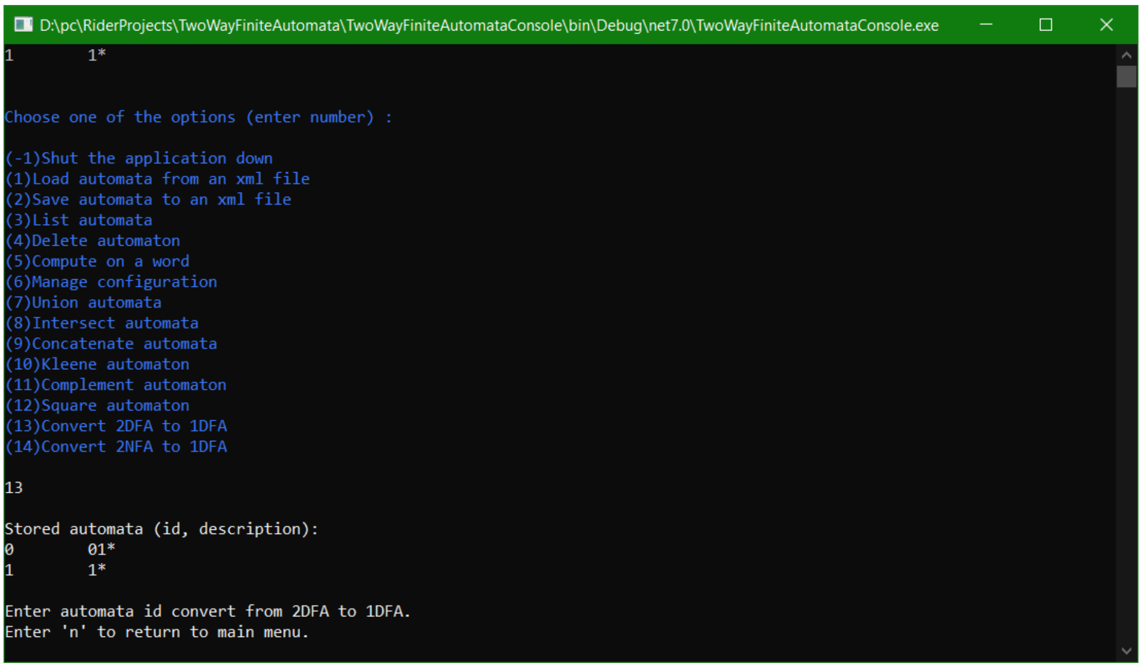
32).

If you enter an id of an automaton from the list, it will be converted to 1DFA. The created automaton will not be added to the list as the automata listed are just 2DFAs, but a confirmation message will appear (Figure 33).



```
D:\pc\RiderProjects\TwoWayFiniteAutomata\TwoWayFiniteAutomataConsole\bin\Debug\net7.0\TwoWayFiniteAutomataConsole.exe
(4)Delete automaton
(5)Compute on a word
(6)Manage configuration
(7)Union automata
(8)Intersect automata
(9)Concatenate automata
(10)Kleene automaton
(11)Complement automaton
(12)Square automaton
(13)Convert 2DFA to 1DFA
(14)Convert 2NFA to 1DFA

12

Stored automata (id, description):
0        01*
1        1*

Enter automaton id and a number of squares in form: automaton_id number
Enter 'n' to return to main menu.
0 5

Stored automata (id, description):
0        01*
1        1*
2        Square, 5: 01*

Enter automaton id and a number of squares in form: automaton_id number
Enter 'n' to return to main menu.
```

Figure 31: Squared automaton



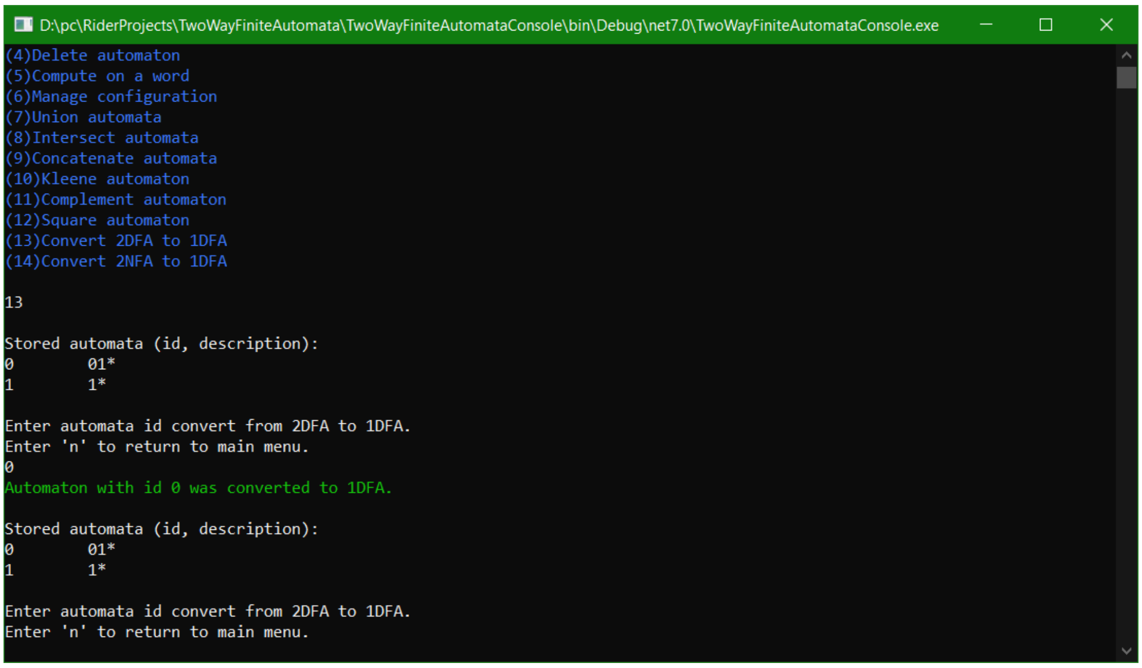
```
D:\pc\RiderProjects\TwoWayFiniteAutomata\TwoWayFiniteAutomataConsole\bin\Debug\net7.0\TwoWayFiniteAutomataConsole.exe
1        1*


Choose one of the options (enter number) :

(-1)Shut the application down
(1)Load automata from an xml file
(2)Save automata to an xml file
(3)List automata
(4)Delete automaton
(5)Compute on a word
(6)Manage configuration
(7)Union automata
(8)Intersect automata
(9)Concatenate automata
(10)Kleene automaton
(11)Complement automaton
(12)Square automaton
(13)Convert 2DFA to 1DFA
(14)Convert 2NFA to 1DFA

13

Stored automata (id, description):
0        01*
1        1*

Enter automata id convert from 2DFA to 1DFA.
Enter 'n' to return to main menu.
```

Figure 32: List of 2DFAs to convert to 1DFA

### Convert 2NFA to 1DFA

Option number 14 allows you to perform a conversion of any of your automata that are 2NFAs.

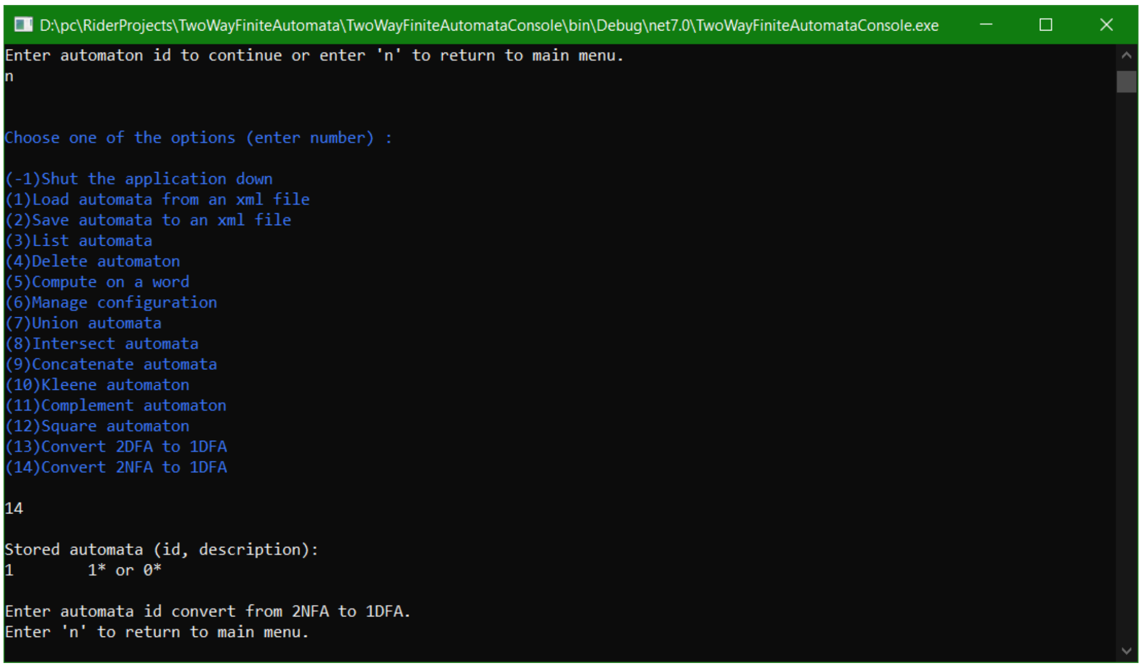
After entering the option number, the list of all 2NFAs will appear (Figure


D:\pc\RiderProjects\TwoWayFiniteAutomata\TwoWayFiniteAutomataConsole\bin\Debug\net7.0\TwoWayFiniteAutomataConsole.exe

```
(4)Delete automaton
(5)Compute on a word
(6)Manage configuration
(7)Union automata
(8)Intersect automata
(9)Concatenate automata
(10)Kleene automaton
(11)Complement automaton
(12)Square automaton
(13)Convert 2DFA to 1DFA
(14)Convert 2NFA to 1DFA

13

Stored automata (id, description):
0        01*
1        1*

Enter automata id convert from 2DFA to 1DFA.
Enter 'n' to return to main menu.
0
Automaton with id 0 was converted to 1DFA.

Stored automata (id, description):
0        01*
1        1*

Enter automata id convert from 2DFA to 1DFA.
Enter 'n' to return to main menu.
```

Figure 33: Converted automata

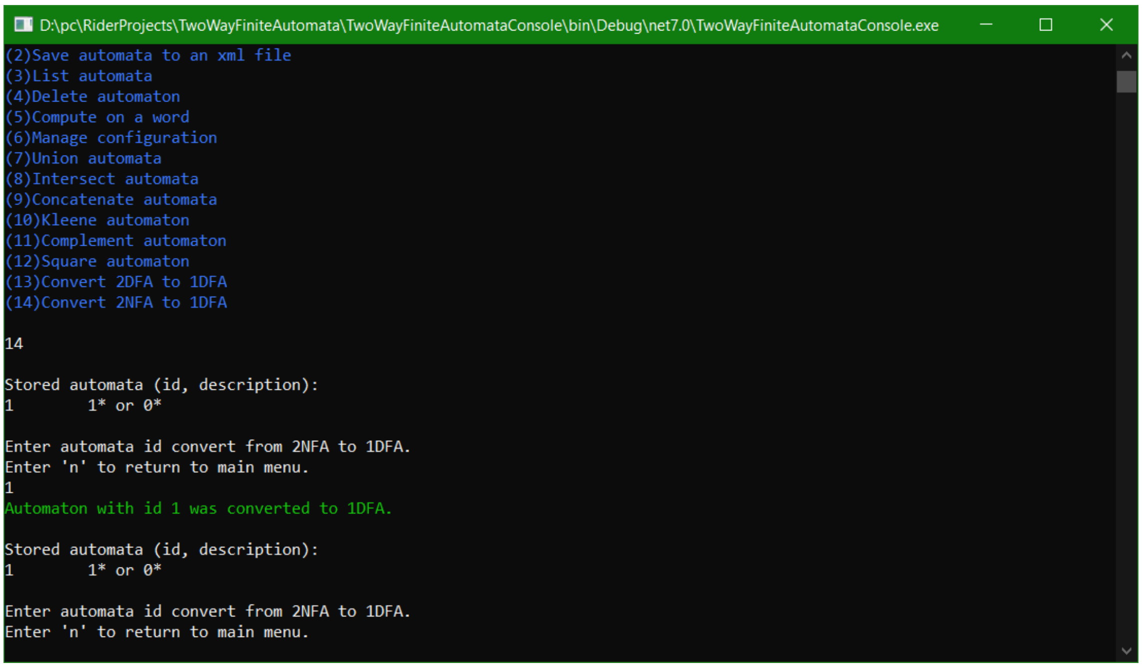
34).


D:\pc\RiderProjects\TwoWayFiniteAutomata\TwoWayFiniteAutomataConsole\bin\Debug\net7.0\TwoWayFiniteAutomataConsole.exe

```
Enter automaton id to continue or enter 'n' to return to main menu.
n


Choose one of the options (enter number) :

(-1)Shut the application down
(1)Load automata from an xml file
(2)Save automata to an xml file
(3)List automata
(4)Delete automaton
(5)Compute on a word
(6)Manage configuration
(7)Union automata
(8)Intersect automata
(9)Concatenate automata
(10)Kleene automaton
(11)Complement automaton
(12)Square automaton
(13)Convert 2DFA to 1DFA
(14)Convert 2NFA to 1DFA

14

Stored automata (id, description):
1        1* or 0*

Enter automata id convert from 2NFA to 1DFA.
Enter 'n' to return to main menu.
```

Figure 34: List of 2NFAs to convert to 1DFA

If you enter an id of an automaton from the list, it will be converted to 1DFA. The created automaton will not be added to the list as the automata listed are just 2NFAs, but a confirmation message will appear (Figure 35).



```
(2)Save automata to an xml file
(3)List automata
(4)Delete automaton
(5)Compute on a word
(6)Manage configuration
(7)Union automata
(8)Intersect automata
(9)Concatenate automata
(10)Kleene automaton
(11)Complement automaton
(12)Square automaton
(13)Convert 2DFA to 1DFA
(14)Convert 2NFA to 1DFA

14

Stored automata (id, description):
1        1* or 0*

Enter automata id convert from 2NFA to 1DFA.
Enter 'n' to return to main menu.
1
Automaton with id 1 was converted to 1DFA.

Stored automata (id, description):
1        1* or 0*

Enter automata id convert from 2NFA to 1DFA.
Enter 'n' to return to main menu.
```

D:\pc\RiderProjects\TwoWayFiniteAutomata\TwoWayFiniteAutomataConsole\bin\Debug\net7.0\TwoWayFiniteAutomataConsole.exe

Figure 35: Converted automata

### Output

In the folder automata in the src folder of the electronic data attached to this work, there are example outputs for each operation.

# Conclusions

This thesis covered the basics of two-way finite automata theory, provided how basic operations are performed on them, and provided the equivalence of 2FA and 1FA. In the practical part, the operations were implemented in a library and used in a sample application.

2FAs have the same computational power as 1FAs but because of the possibility to move the head in both directions, the operations are sometimes hard to perform and it is a necessity to convert them to 1FA for which the algorithms for the operations are known. This usually ends up in large automata with many states.

In conclusion, more research is needed for 2FAs to be comparable to 1FAs especially if it comes to operations. Therefore, even when they have the same computational power as 1FAs, their usage is now limited.

# Závěr

Tato práce pokryla základy teorie obousměrných konečných automatů, uvedla, jak se na nich provádějí základní operace, a ukázala ekvivalenci 2FA a 1FA. V praktické části byly operace implementovány v knihovně a použity ve ukázkové aplikaci.

2FA mají stejný výpočetní výkon jako 1FA, ale kvůli možnosti pohybovat hlavou ve obou směrech jsou operace někdy obtížně proveditelné a je nutné 2FA převést na 1FA, pro které jsou algoritmy pro operace známy. Výstupem jsou obvykle velké automaty s mnoha stavy.

Závěrem lze říci, že je zapotřebí více výzkumu, aby 2FA byly srovnatelné s 1FA, zejména pokud jde o operace. Proto i když mají stejný výpočetní výkon jako 1FA, jejich použití je nyní omezené.

# A Contents of Electronic Data

**`bin/`**

The executable program TwoWayAutomataConsole. All third-side libraries needed to run the program are included.

**`text/`**

The thesis created with style for thesis by KI Přf UP in Olomouc, including all appendices, images, source codes of the text and all other files needed to generate PDF document (in ZIP archive).

The generated PDF document.

**`src/`**

Source codes of the library and the example application, and automata to use in the application and its output automata.

**`readme.txt`**

Instructions and requirements for successful installation and execution of the program.

# Bibliography

[1] PIGHIZZINI, Giovanni; KUTRIB, Martin. *Recent Trends in Descriptional Complexity of Formal Languages.*
Bulletin of EATCS, 2013.

[2] BOWEN HUNT III, Harry. *On the Time and Tape Complexity of Languages.*
Department of Computer Science, Cornell University, Ithaca, New York 14850, 1973.

[3] SAKODA, William J.; SIPSER, Michael. *Nondeterminism and the Size of Two Way Finite Automata.*
Computer Science Division, University of California, Berkeley, California 94720, 1978.

[4] VAN DER HULST, Alex. *Exploring the difference between 2DFA and DFA for G-automata.*
Bachelor Thesis, Computer Science, Radboud University, 2022.
Source: Exploring the difference between 2DFA and DFA for G-automata

[5] RIETBERGEN, Serena. *2-Way Finite Automata.*
Bachelor Thesis, Radboud University, Nijmegen, 2017-2018.
Source: 2-Way Finite Automata

[6] KOZEN, Dexter C. *Automata and Computability.*
Department of Computer Science, Cornell University, Ithaca, NY 14853-7501.
Springer, 1997.
ISBN: 0-387394907-0

[7] JIRÁSKOVÁ, Galina; OKHOTIN, Alexander. *On the state complexity of operations on two-way finite automata.*
Information and Computation, Volume 253, Part 1, 2017, Pages 36-63, ISSN 0890-5401.
Source: https://www.sciencedirect.com/science/article/pii/S0890540116301316#en0160

[8] KUNC, Michal; OKHOTIN, Alexander. *State Complexity of Union and Intersection for Two-way Nondeterministic Finite Automata.*
Fundamenta Informaticae, Volume 110, 2011.
Source: https://content.iospress.com/articles/fundamenta-informaticae/fi110-1-4-18

[9] GEFFERT, Viliam; MEREGHETTI, Carlo; PIGHIZZINI, Giovanni. *Complementing two-way finite automata.*
Information and Computation, Volume 205, 2007.
Source: https://core.ac.uk/download/pdf/82144434.pdf

[10] *Chomsky Hierarchy.*
Source: https://devopedia.org/chomsky-hierarchy

[11] *Conversion from NFA with $\epsilon$ to DFA.*
Source: https://www.javatpoint.com/automata-conversion-from-nfa-with-null-to-dfa

[12] *Regular Grammars.*
Source: https://www.geeksforgeeks.org/right-and-left-linear-regular-grammars/