**Abstract**—Glushkov’s construction has many interesting properties and they become even more evident when applied to transducers. This article strives to show the vast range of possible extensions and optimisations for this algorithm. Special flavour of regular expressions is introduced, which can be efficiently converted to ε-free functional subsequential weighted finite state transducers. Produced automata are very compact, as they contain only one state for each symbol (from input alphabet) of original expression and only one transition for each range of symbols, no matter how large. Such compactified ranges of transitions allow for efficient binary search lookup during automaton evaluation. All the methods and algorithms presented here were used to implement open-source compiler for regular expressions. Some lexicographic order defined as $\Sigma^* \times D$. The set of strings $\Sigma^*$ is the monoid of input strings and $D$ is the monoid of output strings. Lexicographic transducer is defined as tuple $(Q, I, W, D, \delta, \tau)$ where $Q$ is some finite set of states, $I$ is the set of initial states, $\delta$ is a state output (partial) function $Q \times I \times W \times \Sigma \times D \times Q$, and $\tau$ represents transitions of the form $\delta \subset Q \times W \times \Sigma \times D \times Q$.

Thanks to $\tau$, such machines are subsequential. As an example consider the simple transducer from figure 1. The states $q_0$, $q_1$, and $q_2$ have no output, which can be denoted with $\tau(q_0) = \emptyset$. The only set which does have output is $q_3$. Every time automaton finishes reading input string and reaches $q_3$, it will append $d_0$ to its output and then accept. For instance, on input $\sigma_1\sigma_2$ it will first read $\sigma_1$, produce output $d_0d_4$ and go to state $q_1$, then read $\sigma_2$ and append output $d_3$, go to state $q_3$, finally reaching end of input, appending $d_0$ and accepting. The total output would be $d_0d_4d_3d_0$. Note that the automaton is nondeterministic, as it could take alternative route passing through $q_2$ and producing $d_3d_0$. In such scenarios weights are used to disambiguate output. The first route produces weight string $w_2w_3w_1$, while the second produces $w_3w_2w_1$. According to our definition of lexicographic order we have $w_2w_3w_1 > w_3w_2w_1$ (assuming that $w_3 > w_2$). Throughout this article we will consider smaller weights to be "better". Hence the automaton should choose $d_3d_0$ as the definitive output for input $\sigma_1\sigma_2$. There might be situations in which two different routes have the exact same (equally highest) weight while also producing different outputs. In such cases, the automaton is ambiguous and produces multiple outputs for one input.

**II. EXPRESSIVE POWER**

There are some remarks to be made about lexicographic transducers. They recognize relations on languages, unlike "plain" finite state automata (FSA) which recognize languages. If $M$ is some transducer, then we denote its recognized relation with $L(M)$. Those relations are subsets of $\Sigma^* \times D$. The set of strings $\Sigma^*$ accepted by $M$ must be a regular language (indeed, if we erased output labels, we would as a result obtain FSA). The weights are erasable in the sense that, give any lexicographic transducer we can always build an equivalent automaton without weighted transitions. If we didn’t have $\tau$, the only output possible to be expressed for empty input would be an empty string as well. With $\tau$ we can express pairs like $(\epsilon, d) \in L$ where $d \neq \epsilon$.

The transducers can return at most finitely many outputs for any given input (see infinite superposition in [4]). If we allowed for $\epsilon$-transitions (transitions that have $\epsilon$ as input label) we could build $\epsilon$-cycles and produce infinitely many outputs. However, automata that do so are not very interesting from practical point of view. Therefore we shall focus only on functional transducers, that is those which produce at most one output. If automaton does not have any $\epsilon$-cycles and is functional, then it’s possible to erase all $\epsilon$-transitions (note that it would not be possible without $\tau$, because $\epsilon$-transitions allow for producing output given empty input). Therefore $\epsilon$-transitions don’t increase power of functional transducers.

We say that transducer has conflicting states if it’s possible to reach both of them simultaneously (there are two possible routes with the same inputs and weights) given some input $\sigma$ and there is some another state $q_3$ to which both of those states can transition over the same input symbol $\sigma$. Alternatively, there might be no third state $q_3$ but instead both $q_1$ and $q_2$ have non-empty $\tau$ output (so $\tau$ can in a sense be treated like $q_3$). We say that transitions $(\sigma_1, \sigma_2, w, d, q_3)$ and $(\sigma_2, \sigma_1, w', d, q_3)$ are weight-conflicting if they have equal weights $w = w'$. For instance in figure 1 the states $q_1$ and $q_2$ are indeed conflicting because they both transition to $q_3$ over $\sigma_2$ but their transitions are not weight conflicting. It can be shown that transducers without weight-conflicting transitions are functional. Moreover, if a transducer is functional but contains weight-conflicting
transitions, then the weights can be reassigned in such a way that eliminates all conflicts \[^4\]. The only requirement is that there are enough symbols in \(W\) (for instance, if \(W\) had only one symbol, then all transitions of conflicting states would always be weight-conflicting). If there are at least as many weight symbols as there are states \(|W| = |Q|\), then every functional transducer on \(|W|\) states can be built without weight-conflicting transitions. For convenience we can assume that \(W = \mathbb{N}\), but in practice all algorithms presented here will work with bounded \(W\). Hence transducers without weight-conflicting transitions are equivalent in power to functional transducers. This is important because by searching for weight-conflicting transitions we can efficiently test whether transducer is functional or not.

III. RANGED AUTOMATA

Often when implementing automata the algorithm behind \(\delta\) function needs to efficiently find the right transition for a given \(\sigma\) symbol. It’s beneficial to optimise UNIX-style ranges like \([0-9]\) or \([a-z]\) as they arise often in practical settings. Even the \(\cdot\) wildcard can be treated as one large range spanning entire \(\Sigma\). If the alphabet is large (like ASCII or UNICODE), then checking every one of them in a loop is not feasible. A significant improvement can be made by only checking two inequalities like \(\sigma_{1} \leq x \leq \sigma_{10}\), instead of large number of equalities. The current paper presents a way in which simplified model of \(\Sigma\) can be used to obtain major improvements. In particular we consider only automata that don’t have any registers apart from constant values, that is \(k = 0\). Therefore we provide a more specialized definition of “ranged automata”.

Let \(\Sigma\) be the (not necessarily finite) alphabet of automaton. Let \(\chi\) be the set of subsets of \(\Sigma\) that we will call ranges of \(\Sigma\). Let \(\overline{\chi}\) be the closure of \(\chi\) under countable union and complementation (so it forms a sigma algebra). For instance, imagine that there is total order on \(\Sigma\) and \(\chi\) is the set of all intervals in \(\Sigma\). Now we want to build an automaton whose transitions are not labelled with symbols from \(\Sigma\), but rather with ranges from \(\chi\). Union \(\chi_{0} \cup \chi_{1}\) of two elements from \(\chi\) “semantically” corresponds to putting two edges, \((q, \chi_{0}, q') \in \delta\) (for a moment forget about outputs and weights) and \((q, \chi_{1}, q') \in \delta\). There is no limitation on the size of \(\delta\). It might be countably infinite, hence it’s natural that \(\overline{\chi}\) should be closed under countable union. Therefore, \(\chi\) is the set of allowed transition labels and \(\overline{\chi}\) is the set of all possible “semantic” transitions. We could say that \(\overline{\chi}\) is discrete if it contains every subset of \(\Sigma\). An example of discrete \(\overline{\chi}\) would be finite set \(\Sigma\) with all UNIX-style ranges \([\sigma_{1}-\sigma_{9}] \) included in \(\chi\).

Another example would be set \(\Sigma = \mathbb{R}\) with \(\chi\) consisting of all ranges, whose ends are computable real numbers (real number \(x\) is computable if the predicate \(q \leq x\) is decidable for all rational numbers \(q\)). If we also restricted \(\delta\) to be a finite set, then we could build effective automata that work with real numbers of arbitrary precision.

IV. REGULAR EXPRESSIONS

Here we describe a flavour of regular expressions specifically extended to interplay with lexicographic transducers and ranged automata.

Transducers with input \(\Sigma^{*}\) and output \(\Gamma^{*}\) can be seen as FSA working with single input \(\Sigma^{*} \times \Gamma^{*}\). Therefore we can treat every pair of symbols \((\sigma, \gamma)\) as an atomic formula of regular expressions for transducers. We can use concatenation \((\sigma, \gamma_{0})(\epsilon, \gamma_{1})\) to represent \((\sigma, \gamma_{0}\gamma_{1})\). It’s possible to create ambiguous transducers with unions like \((\epsilon, \gamma_{0}) + (\epsilon, \gamma_{1})\). To make notation easier, we will treat every \(\sigma\) as \((\sigma, \epsilon)\) and every \(\gamma\) as \((\epsilon, \gamma)\). Instead of writing lengthy \((\sigma, \epsilon)(\epsilon, \gamma)\) we could introduce shortened notation \(\sigma : \gamma\). Because we would like to avoid ambiguous transducers we can put restriction that the right side of \(\cdot\): should always be a string of \(\Gamma^{*}\) and writing entire formulas (like \(\sigma : \gamma_{1} + \gamma_{2}\)) is not allowed. This restriction will later simplify Glushkov’s algorithm.

We define \(A_{\Sigma}\) to be the set of atomic characters. For instance we could choose \(A_{\Sigma} = \Sigma \cup \{\epsilon\}\) for FSA/transducers and \(A_{\overline{\Sigma}} = \chi\) for ranged automata.

We call \(RE_{\Sigma}^{\mathbb{D}}\) the set of all regular expression formulas with underlying set of atomic characters \(A_{\overline{\Sigma}}\) and allowed output strings \(D\). It’s possible that \(D\) might be a singleton monoid \(\{\epsilon\}\) but it should not be empty, because then no element would belong to \(\Sigma^{*} \times D\). By inductive definition, if \(\phi\) and \(\psi\) are \(RE_{\Sigma}^{\mathbb{D}}\) formulas and \(d \in D\), then union \(\phi + \psi\), concatenation \(\phi \cdot \psi\), Kleene closure \(\phi^{*}\) and output concatenation \(\phi : d\) are \(RE_{\Sigma}^{\mathbb{D}}\) formulas as well. Define \(V_{\Sigma}^{\mathbb{D}} : RE_{\Sigma}^{\mathbb{D}} \rightarrow \Sigma^{*} \times D\) to be the valuation function:

\[
V_{\Sigma}^{\mathbb{D}}(\phi + \psi) = V_{\Sigma}^{\mathbb{D}}(\phi) \cup V_{\Sigma}^{\mathbb{D}}(\psi)
\]

\[
V_{\Sigma}^{\mathbb{D}}(\phi \cdot \psi) = V_{\Sigma}^{\mathbb{D}}(\phi) \cdot V_{\Sigma}^{\mathbb{D}}(\psi)
\]

\[
V_{\Sigma}^{\mathbb{D}}(\phi^{*}) = (\epsilon, \phi) + V_{\Sigma}^{\mathbb{D}}(\phi) + V_{\Sigma}^{\mathbb{D}}(\phi)^{2} + ...
\]

\[
V_{\Sigma}^{\mathbb{D}}(\phi : d) = V_{\Sigma}^{\mathbb{D}}(\phi) \cdot (\epsilon, d)
\]

\[
V_{\Sigma}^{\mathbb{D}}(a) = a \text{ where } a \in A_{\overline{\Sigma}}^{\mathbb{D}}
\]

Some notable properties are:

\[
x : y_{0} + x : y_{1} = x : (y_{0} + y_{1})
\]

\[
x : x + y : x + y_{2} = x : y_{2} + x : y
\]

\[
x : y_{0}(y_{1} + y_{2} + y_{3}) = (x_{0} \cdot y_{3} + x_{0} + x_{1} : y_{1}) \cdot (x_{1} \cdot \epsilon)
\]

\[
x : y_{0} + x : y_{0} + x : y_{1} = x : (y_{0} + y_{1} + x : y_{1})
\]

Therefore we can see that expressive power with and without \(\cdot\) is the same.

It’s also possible to extend regular expressions with weights. Let \(RE_{\overline{\Sigma}}^{\mathbb{D}}\) be a superset of \(RE_{\Sigma}^{\mathbb{D}}\) and \(W\) be the set of weight symbols. If \(\phi \in RE_{\overline{\Sigma}}^{\mathbb{D}}\) and \(w_{0}, w_{1} \in W\) then \(w_{0} \phi\) and \(w_{1} \phi\) are in \(RE_{\overline{\Sigma}}^{\mathbb{D}}\). This allows for inserting weight at any place. For instance, the automaton from figure \[^1\] could be expressed using

\[
((\sigma_{1} : d_{0}d_{4})w_{2}(\sigma_{2} : d_{3})w_{3} + (\sigma_{1} : d_{4})w_{3}\sigma_{2}w_{2}) : d_{0}
\]
The definition of \( V^{Σ,D}(ϕw) \) depends largely on \( W \) but associativity \( ϕw_1w_2 = ϕ(w_1 + w_2) \) should be preserved, given that \( W \) is a multiplicative monoid. This also implies that \( w_1w_2 = w_1 + w_2 \), which is semantically equivalent to the addition \( w_1 + w_2 \).

We showed that regular expressions for transducers can be expressed using pairs of symbols \((σ,γ)\). There is an alternative approach. We can encode both input and output string by interleaving their symbols like \( σ_1γ_1σ_2γ_2 \). Such regular expressions “recognize” relations rather than “generate” them. This approach has one significant problem. We have to keep track of the order. For instance, this

\[
\sigma_1\gamma_1\sigma_2\gamma_2
\]

should notice that \((ΣΓ)^*\) is a multiplicative monoid. This also implies that every atomic character (including duplicates but excluding \( ϵ \)) in \( ϕ \) is treated as a new individual character. As a result we should obtain new rewritten formula \( ψ ∈ RE^{Ω→D}_W \) along with mapping \( α : Ω → A^Ω \). This mapping will remember the original atomic character, before it was rewritten to unique symbol in \( Ω \). For example

\[
ϕ = (ε : x_0)x_0(x_0 : x_1x_3)x_2w_0 + (x_1x_2)^*w_1
\]

will be rewritten as

\[
ψ = (ε : x_0)w_1(ω_2 : x_1x_3)ω_3w_0 + (ω_4w_5)^*w_1
\]

with \( α = \{(ω_1, x_0), (ω_2, x_0), (ω_3, x_3), (ω_4, x_1), (ω_5, x_2)\} \).

Every element \( x \) of \( A^Ω \) may also be member of several subalphabets. For simplicity we can assume that all expressions are annotated and we know exactly which subalphabet a given \( x \) belongs to. In practice, we would try to infer the annotation automatically and ask user to manually annotate symbols only when necessary.

Next step is to define function \( Λ : RE^{Ω→D}_W \rightarrow (D × W) \). It returns the output produced for empty word \( ε \) (if any) and weight associated with it. (We use symbol \( → \) to highlight the fact that \( Λ \) is a partial function and may fail for ambiguous transducers.) For instance in the previous example empty word can be matched and the returned output and weight is \( (ε, w_1) \). Because both \( D \) and \( W \) are monoids, we can treat \( D × W \) like a monoid defined as \( (yd, w_1)(y, w_2) = (y(yd, w_1), w_1 + w_2) \). We also admit \( 0 \) as multiplicative zero, which means that \((y, w_1) ⊩ 0 = \emptyset \). We denote \( W \)’s neutral element as \( 0 \). This facilitates recursive definition:

\[
Λ(ψ_1 + ψ_2) = Λ(ψ_1) ∪ Λ(ψ_2)
\]

if at least one of the sides is \( 0 \), otherwise error

\[
Λ(ψ_0ψ_1) = Λ(ψ_0) · Λ(ψ_1)
\]

\[
Λ(ψ_0 · y) = Λ(ψ_0) · (y, 0)
\]

\[
Λ(ψ_0w) = Λ(ψ_0) · (ε, w)
\]

\[
Λ(wψ_0) = Λ(ψ_0) · (ε, w)
\]

\[
Λ(ψ_0) = (ε, 0)\text{ if } (ε, w) = Λ(ψ_0)\text{ or } 0 = Λ(ψ_0), \text{ otherwise error}
\]

\[
Λ(ε) = (ε, 0)
\]

\[
Λ(ω) = ω \text{ where } ω \in Ω
\]

Next step is to define function \( B : RE^{Ω→D}_W \rightarrow (Ω → D × W) \) which for a given formula \( ψ \) returns set of \( Ω \) characters that can be found as the first in any string of \( V^{Ω→D}(ψ) \) and to each such character we associate output produced “before” reaching it. For instance, in the previous example of \( ψ \) there are two characters that can be found at the beginning: \( ω_1 \) and \( ω_4 \). Additionally, there is \( ε \) which prints output \( x_0 \) before reaching \( w_1 \). Therefore \((ω_1, (x_0, 0)) \) and \((ω_2, (ε, 0)) \) are the result of \( B(ψ) \). For better readability, we admit operation of multiplication \( · : (Ω → D × W) × (D × W) → (Ω → D × W) \) that performs monoid multiplication on all \( D × W \) elements returned by \( Ω → D × W \).

\[
B(ψ_0 + ψ_1) = B(ψ_0) ∪ B(ψ_1)
\]

\[
B(ψ_0ψ_1) = B(ψ_0) ∪ Λ(ψ_0) · B(ψ_1)
\]

\[
B(ϕw_0) = B(ϕ_0) \text{ if } (ε, w) = B(ψ_0)
\]

\[
B(ψ_0w) = B(ψ_0) \text{ if } (ε, w) = B(ψ_0)
\]

\[
B(ψ_0) = B(ψ_0)
\]

\[
B(ε) = 0
\]

\[
B(ω) = \{(ω, (ε, 0))\}
\]

It’s worth noting that \( B(ψ_0) ∪ B(ψ_1) \) always yields function (instead of relation) because every \( Ω \) character appears in \( ψ \) only once and it cannot be both in \( ψ_0 \) and \( ψ_1 \).
Next step is to define $E : RE_v^{Q \times D} \rightarrow (\Omega \rightarrow D \times W)$, which is very similar to $B$, except that $E$ collects characters found at the end of strings. In our example it would be $((\omega_3, (\epsilon, w_0)))$ and $((\omega_5, (\epsilon, w_1)))$. Recursive definition is as follows:

$E(\psi_0 + \psi_1) = E(\psi_0) \cup E(\psi_1)$

$E(\psi_0 \cdot \psi_1) = E(\psi_0) \cdot \Lambda(\psi_1) \cup B(\psi_1)$

$E(\psi_0 \cdot W) = E(\psi_0) \cdot (e, w)$

$E(w \psi_0) = E(\psi_0)$

$E(\psi_0 \cdot d) = E(\psi_0) \cdot (d, 0)$

$E(\epsilon) = \emptyset$

$E(\omega) = \{(\omega, (\epsilon, 0))\}$

Next step is to use $B$ and $E$ to determine all two-character substrings that can be encountered in $V^{Q \rightarrow D}(\psi)$. Given two functions $b, e : \Omega \rightarrow D \times W$ we define product $b \times e : \Omega \times \Omega \rightarrow D \times W$ such that for any $(\omega_0, (y_0, w_0)) \in b$ and $(\omega_1, (y_1, w_1)) \in e$ there is $((\omega_0 \omega_1), (y_0 y_1, w_0 + w_1)) \in b \times e$.

Then define $L : RE_v^{Q \times D} \rightarrow (\Omega \times \Omega \rightarrow D \times W)$ as:

$L(\psi_0 + \psi_1) = L(\psi_0) \cup L(\psi_1)$

$L(\psi_0 \cdot \psi_1) = L(\psi_0) \cup L(\psi_1) \cup E(\psi_0) \times B(\psi_1)$

$L(\psi_0 \cdot W) = L(\psi_0)$

$L(w \psi_0) = L(\psi_0)$

$L(\psi_0 \cdot d) = L(\psi_0)$

$L(\epsilon) = \emptyset$

$L(\omega) = \emptyset$

One should notice that all the partial functions produced by $B$, $E$ and $L$ have finite domains, therefore they are effective objects from computational point of view.

The last step is to use results of $L, B, E, \Lambda$ and $\alpha$ to produce automaton $(Q, q_0, W, \Sigma, D, \delta, \tau)$ with

$\delta : Q \times \Sigma \rightarrow (Q \rightarrow D \times W)$

$\tau : Q \rightarrow D \times W$

$Q = \{q_\omega : \omega \in \Omega\} \cup \{q_e\}$

$\tau = E(\psi)$

$(q_{\omega_0}, \alpha(\omega_1), q_{\omega_2}, d, w) \in \delta$ for every $(\omega_0, \omega_1, d, w) \in L(\psi)$

$(q_0, \epsilon(q_w), q_e, d, w) \in \delta$ for every $(\omega, d, w) \in B(\psi)$

This concludes the Glushkov’s construction. Now it’s possible to use specification $U$, $V_2$, $W$ and $\Sigma$ of interleaved alphabet to check if regular expression was correct. We can treat alphabets $\Sigma_1, \Sigma_2, \ldots$ as colours and then colour each state with it’s respective alphabet. If transition leads from state of colour $\Sigma_i$ to $\Sigma_j$ then we check that the pair $(\Sigma_i, \Sigma_j)$ is indeed present in $V_2$. Similarly we check colours of initial and accepting states.

**VI. Optimisations**

The above construction can detect some obvious cases of ambiguous transducers, but it doesn’t give us complete guarantee. We can check in quadratic time[12] for weight conflicting transitions to be sure. If there are none, then transducer must be functional. If we find at least one, it doesn’t immediately imply that the transducer is ambiguous. In such cases we can warn the user and demand additional weight annotations in the regular expression.

When $A^2$ consists of all possible ranges $\chi$, then the obtained $\delta$ is of the form $Q \times W \times \chi \times D \times Q$. While, theoretically equivalent to $Q \times W \times \Sigma \times D \times Q$, in practice it allows for more efficient implementations. For instance given two ranges $[1-50]$ and $[20-80]$, we do not need to check equality for all 80 numbers. The only points worth checking are 1, 50, 20, 80. Let’s arrange them in some sorted array. Then given any number $x$, we can use binary search to find which of those points is closest to $x$ and then lookup the full list of intervals that $x$ is a member of. This approach works even for real numbers. More precise algorithm can be given a follows. Let $(x_0, y_0), (x_1, y_1), \ldots (x_n, y_n)$ be closed ranges. Build an array $P$ sorted in ascending order that contains all $y_i$ and also for every $x_i$ contains the largest element of $\Sigma$ smaller than $x_i$ (or more generally the supremum). Build a second array $R$ that for every $i$th element of $R$ assign list of ranges containing $i$th element of $P$. Then in order to find ranges containing any $x$, run binary search on $P$ that returns index of the largest element smaller or equal to $x$. Then lookup the list of ranges in $R$.

Note that in Glushkov’s construction epsilons are not rewritten to $\Omega$, which means that there are also no $\epsilon$-transitions. Hence we can use dynamic programming to efficiently evaluate automaton for any input string $x \in \Sigma^*$. The algorithm is as follows. Create two dimensional array $c_{i,j}$ of size $|Q| \times (|x| + 1)$ where $i$-th column represents all nondeterministically reached states after reading first $i - 1$ symbols. Each cell should hold information about the previously used transition. This also tells us the weight, output and source state of transition. For instance cell $c_{i,j} = k$ should encode transition coming from state $k$ to state $i$, after reading $j - 1$th symbol. If state $q_i \in Q$ does not belong to $j$th superposition, then $c_{i,j} = \emptyset$. The first column is initialized with some dummy value at $c_{i,1} = -1$ for $i$ referring to initial state $q$, and set to $c_{i,1} = \emptyset$ for all other $i$. Then algorithm progresses building next column from previous one. After filling out the entire array. The last column should be checked for any accepting states according to $\tau$. There might be many of them but the one with largest weight should be chosen. If we checked that the automaton has no weight-conflicting transitions, then there should always be only one maximal weight. Finally we can backtrack, to find out which path “won”. This will determine what outputs need to be concatenated together to obtain path’s output. This algorithm is quadratic $O(|Q| \times |x|)$, but in practice each iteration itself it very efficient, especially when combined with binary search described in previous paragraph.

**VII. Conclusion**

Interleaved alphabets could find numerous applications with many possible extensions. In the setting of natural language processing, interleaved alphabets of the form \{Verb,Noun,Adj\} × \{a,b,...,z\}" could be used to represent human sentences divided into words with linguistic meta-information.

This approach cannot fully replace OpenFST, because it lacks their flexibility. The goal of OpenFST is to provide general and extensible implementation of many different transducer’s, whereas the approach presented in this paper sacrifices extensibility for highly integrated design and optimal efficiency. For instance, Glushkov’s algorithm could never support many operations such as inverses, projections or reverses.
ACKNOWLEDGMENT

The authors would like to thank Piotr Radwan for all the inspiration.

REFERENCES

[1] M. Mohri, “Weighted finite-state transducer algorithms an overview,” *AT&T Labs*, 2004.
[2] M. Droste, W. Kuich, and H. Vogler, *Handbook of Weighted Automata*, 01 2009.
[3] M. Droste and D. Kuske, “Weighted automata,” *Institut fur Informatik, Universitat Leipzig*, 2010.
[4] A. Mendoza-Drosik, “Multitape automata and finite state transducers with lexicographic weights,” *ArXiv*, vol. abs/2007.12940, 2020.
[5] V. M. Glushkov, *The abstract theory of automata*. Russian Mathematics Surveys, 1961.
[6] F. P. Mehryar Mohri and M. Riley, “Weighted finite-state transducers in speech recognition,” *AT&T Labs Research*, 2008.
[7] M. Mohri, *Weighted Finite-State Transducer Algorithms. An Overview*. Springer, 2004.
[8] C. E. Hasan Ihne Akram, Colin de la Higuera, “Actively learning probabilistic subsequential transducers,” *JMLR: Workshop and Conference Proceedings*, 2012.
[9] C. de la Higuera, *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010.
[10] K. Meer and A. Naif, “Generalized finite automata over real and complex numbers,” vol. 591, 04 2014.
[11] A. Gandhi, B. Khoussainov, and J. Liu, “Finite automata over structures,” in *Theory and Applications of Models of Computation*, M. Agrawal, S. B. Cooper, and A. Li, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 373–384.
[12] M.-P. Béal, O. Carton, C. Prieur, and J. Sakarovitch, “Squaring transducers: An efficient procedure for deciding functionality and sequentiality of transducers,” in *LATIN 2000: Theoretical Informatics*, G. H. Gonnet and A. Viola, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 397–406.