Development history and justification

Aleksander Mendoza-Drosik

March 4, 2021

The theory behind implementation and design of Solomonoff has been studied and developed over the course of 2 years. The early versions and our initial ideas looked very different to the final results we've achieved.

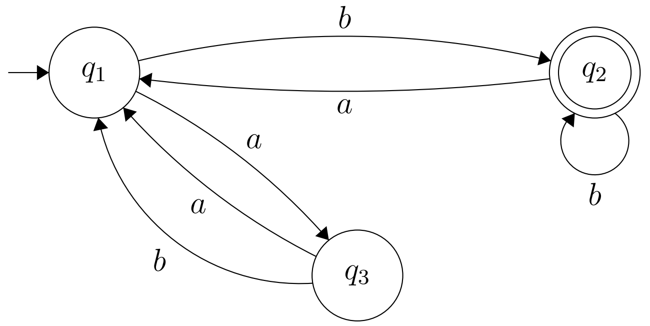
At the beginning it was meant to be a simple tools that focused only on deterministic Mealy automata. Such a model was very limited and it quickly became apparent that certain extensions would need to be made. In the end we implemented a compiler that supports nondeterministic functional weighted symbolic transducers.

The theory behind implementation and design of Solomonoff has been studied and developed over the course of 2 years. The early versions and our initial ideas looked very different to the final results we've achieved.

At the beginning it was meant to be a simple tool that focused only on deterministic Mealy automata. Such a model was very limited and it quickly became apparent that certain extensions would be necessary. In the end we implemented a compiler that supports nondeterministic functional weighted symbolic transducers.

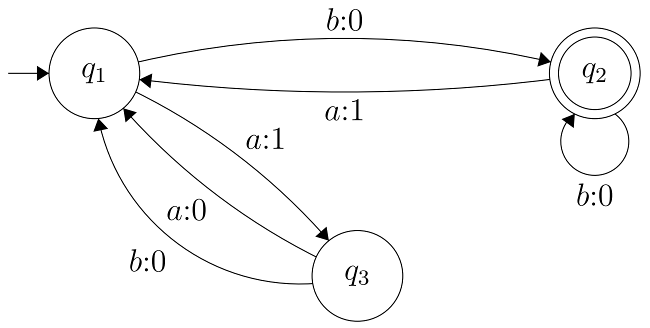
1 Mealy automata and transducers

The standard definition of automaton found in introductory courses [1, 2, 3] states that finite state automaton is a tuple $(Q, I, \delta, F, \Sigma)$ where Q is the set of states, $I \subset Q$ are the initial states, $F \subset Q$ are the final (or accepting) states, $\delta{:}Q \times \Sigma \to Q$ is the transition function and Σ is some alphabet. The automaton can either accept or reject a certain input string. Below is an example in the form of a state graph.



The accepting state $q_2 \in F$ is marked with a double border. Such automaton accepts strings like bb,bbb,bab,aab but rejects ϵ,a,ba,aa and so on.

A Mealy machine [4] extends the above definition with output $(Q, I, \delta, F, \Sigma, \Gamma)$ where Γ is some output alphabet and transition function has the form of $\delta{:}Q \times \Sigma \to Q \times \Gamma$. For example the below machine



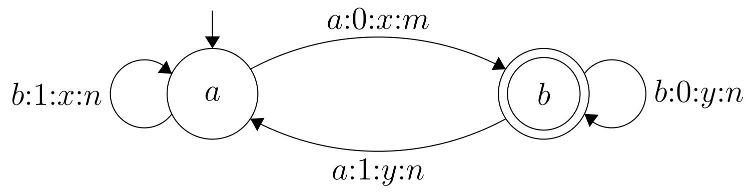
produces output 1000 for input $aabb$. Some articles do not include the set F in definition of Mealy machines and instead assume that all states are accepting. Automata with all states final are called prefix-closed. Here we do not assume Mealy machines to be prefix-closed.

The ability to produce output along each transition allows for modelling reactive systems. Any program or phenomenon that reads input from its environment and "reacts" to it by producing output in one form or another is a reactive system. It is frequently used in the field of formal methods. Many complex state-based systems can be modelled and simplified using Mealy machines []. As an example consider a black-box computer program whose logs can be observed []. The current snapshot of the program's memory determines its state. Depending on subsequent user input, we might observe different log traces. There are many existing machine learning and inference algorithms that can build an equivalent model of Mealy machine only by interacting with such a black-box system and reading its logs.

It can be proved that the expressive power of deterministic automata with output is strictly less than that of their nondeterministic counterparts [5]. It is known as the prefix-preserving property []. If a deterministic automaton reads input string $\sigma_1\sigma_2\sigma_1$ and prints $\gamma_1\gamma_1\gamma_2$, then at the next step it is only allowed to append one more symbol to the previously generated output. For instance we could not suddenly change the output to $\gamma_2\gamma_2\gamma_2\gamma_1$ after reading one more input symbol $\sigma_1\sigma_2\sigma_1\sigma_2$. The prefix $\gamma_1\gamma_1\gamma_2$ must be preserved.

The problems that we wanted to tackle with Solomonoff revolved around building sequence-to-sequence models. For instance we might want to translate from numbers written as English words into digits. A sentence like "zero bugs" should become "0 bugs". The prefix preserving property would be too limiting because it often happens that the suffix of string has a decisive impact on the translation. The phrase "zeroed bit" also starts with the prefix "zero" but it should not be translated as "0ed bit"!

We intended to find the right balance between expressive power of nondeterministic machines and strong formal guarantees of Mealy automata. To achieve this, we initially decided to use multitape automata [6]. For example the automaton below has 4 tapes



First tape is designated as the input state reading strings over the alphabet $\{a, b\}$. The remaining three tapes are the output tapes, printing strings in the respective alphabets $\{0, 1\}$, $\{x, y\}$, $\{m, n\}$. In theory there could be any (finite) number of input and output tapes.

Our idea was to write all possible continuations of given output and store each in a separate output tape. Then upon reaching the end of string, the state would decide which tape to use as output.

Over the course of research and development we have discovered that the power of multitape Mealy machines was still too limiting for our purposes. In particular we could define congruence classes on pairs of strings similar to those in Myhill-Nerode theorem [1]. Then it can be easily noticed that as long as the number of output tapes is finite, the number of congruence classes must be finite as well. A very simple and intuitive counter-example would be the language that for every string a, aa, aaa,... respectively prints output *c*, *bc*, *bbc*,... and so on. It could not be expressed using any finite number of output tapes, because there are infinitely many ways to continue the output and none of them is a prefix of the other.

Yet another limitation of Mealy machines is that their δ function enforces outputs of the exact same length as inputs. In the field of natural language processing such an assumption is too strict. For instance, we might want to build a machine that translates sentences from one language to another. A word "fish" in English might have 4 letters but Spanish "pescado" is much longer. Our first idea was to use sequential transducers instead of the plain Mealy automata. Their definition allows the transition function to print output strings of arbitrary length $\delta{:}Q \times \Sigma \to Q^*$.

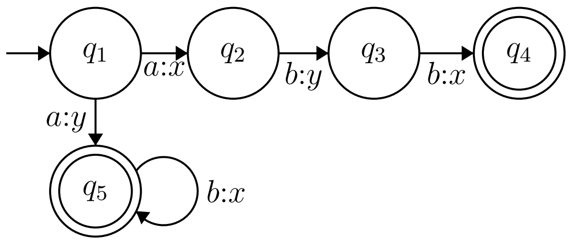
The nondeterministic transducers [7] [8] [9] don't suffer from any of the above problems. Their expressive power exactly corresponds to that of regular transductions. It's a very strong and expressive class. The only way to obtain a stronger model would be by introducing context-free grammars and pushdown automata. The reason why we were hesitant to use this approach was because of the possible ambiguity of output [10]. Nondeterministic transducers may contain epsilon transitions, which could lead to infinite number of outputs [5] for any given input. Without epsilons, the number of outputs is finite but it's still possible to return more than one ambiguous output.

The hierarchy of automata presented so far could be illustrated as follows.

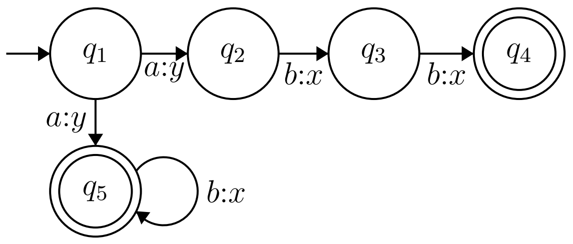
$$\begin{tikzcd}
& \mathit{Mealy\ Machines} \arrow[dl] \arrow[dr] & \\
\begin{matrix}\mathit{Sequential}\\ \mathit{transducers}\end{matrix} \arrow[dr] & & \begin{matrix}\mathit{Multitape}\\ \mathit{Mealy\ Machines}\end{matrix} \arrow[dl] \\
& \begin{matrix}\mathit{Nondeterministic}\\ \mathit{transducers}\end{matrix} &
\end{tikzcd}$$

Mealy machines are the simplest and most basic model. Both sequential transducers and multitape automata extend the expressive power of Mealy machines in different ways but neither is more powerful than the other. Finally, nondeterministic transducers are on top of the hierarchy. This is a simplified view. The full family of different models capable of producing output is much larger. The transduction hierarchy is more complex than Chomsky hierarchy, even within the layer of regular transductions (not to mention context-free and context-sensitive transductions).

The model of automata that was finally implemented in Solomonoff, is the functional nondeterministic transducer [6]. Functionality of a transducer means that for any input, there may be at most one output. For instance the transducer below is not functional, because for input string *abb* it prints both outputs *xyx* and *yxx*.



This should not be confused with unambiguous automata. For example the transducer below is functional but ambiguous. The input *abb* prints only a single output *yxx*, albeit there are two ambiguous accepting paths that produce it.



Functional nondeterministic transducers proved to provide the perfect balance of power with many strong formal guarantees. While epsilon transitions strictly increase power of transducers, when restricted only to functional automata, the erasure of epsilons becomes possible (with the exception of epsilon transitions coming from initial state). Using advance-and-delay [10] algorithm one can test functionality of any automaton in quadratic time. There exists a special version of powerset construction that can take any functional transducers and produce an equivalent unambiguous automaton.

One can take advantage of unambiguity to build an inference algorithm for learning functional automata from sample data. The automata are closed under union, concatenation, Kleene closure and composition. Unlike non-functional transducers, they are not closed under inversion but we developed a special algebra that uniquely defines an invertible bijective transduction for any automaton. Glushkov's construction [11] can be augmented to produce functional transducers. Lexicographic semiring of weights can be used to make functional automata more compact.

Once we decided to use the power of functional transducers, the next problem we had to solve was their optimisation. One of the most important operations in natural language processing is the context-dependent rewrite. The standard way of implementing it is by building a large transducer that handles all possible cases. In Solomonoff we've developed our own approach that produces much smaller automata. It is done with lexicographic weights [5].

Another important optimisation is the state minimisation. Many existing libraries implement separate functions for all regular operations and additional one for minimisation. A regular expression like $A(B+C)*$ would be then translated to a series of function calls like

$$\texttt{minimise}(\texttt{concatenate}(A, \texttt{kleene\_closure}(\texttt{union}(B, C))))$$

This was our initial idea too but later we stumbled upon a better approach. In Solomonoff we don't have separate implementations for those operations. Instead everything is integrated in the form of one monolithic procedure that implements Glushkov's construction. For example, in order to compile the expression

$$aa(b+ca)*+c$$

we first convert every symbol into a state and obtain the following intermediate "regular expression"

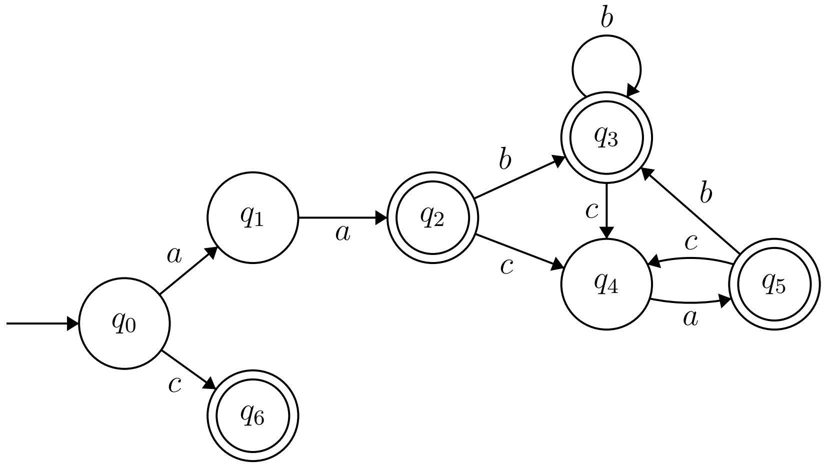
$$q_1q_2(q_3+q_4q_5)*+q_6$$

Then we add one more state at the beginning that will serve as initial

$$q_0(q_1q_2(q_3+q_4q_5)*+q_6)$$

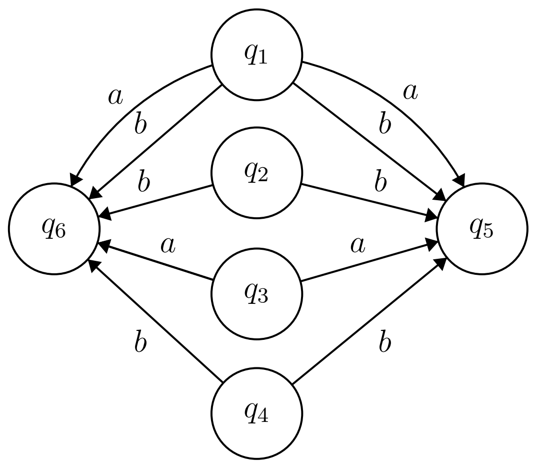
Next we analyse it and determine, which state can be reached from any other. After "reading" q_1 we can "read" q_2. After q_2 we can either read q_3

or q_4. After q_3 we can read either q_3 again or go to q_4. Analogically for the remaining states. We also check, which states can appear at the end of the regular expression. In this example q_2, q_3, q_5 and q_6 are the final states because reading may end after reaching them. By putting all the above information together, we are able to produce the following automaton.

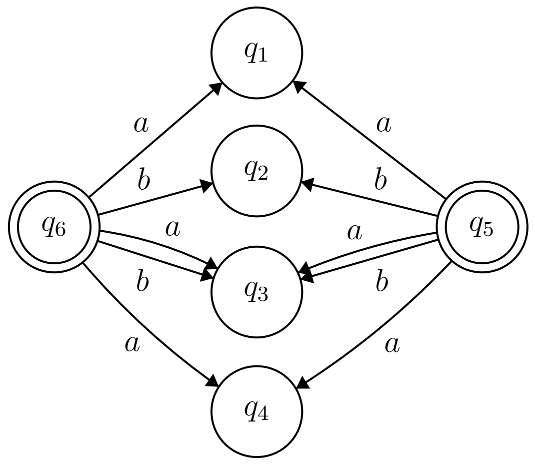


This way the produced automata are very small even without the need for minimisation. If the regular expression consists of n input symbols, then the resulting transducers has $n + 1$ states. Because there is one-to-one correspondence between regular expression symbols and automata states, we are able to retain plenty of useful meta information. In particular each state can tell us exactly, which source file it comes from and the precise position in that file. This way, whenever compilation fails, user can see meaningful error messages.

The standard minimisation procedure used by most libraries works by finding the unique smallest deterministic automaton. Nondeterministic automata do not admit unique smallest representative and might be even exponentially smaller than their equivalent minimal deterministic counterparts. Finding the smallest possible nondeterministic automaton is a hard problem [12]. For this reason Solomonoff implements a heuristic pseudo-minimisation algorithm that attempts to compress nondeterministic transducers and does not attempt to determinise them. Glushkov's construction already produces very small automata, hence any attempt at minimising them by determinisation would result in larger automata than the initial ones. Solomonoff's minimisation is inspired by Brzozowski's algorithm [] and is based on the duality of reachable and unobservable states. In simple terms, if two states have the exact same sets of incoming (or outgoing) transitions then they have no reachable (or observable) distinguishing sequence. For example in the fragment of automaton below, the states q_5 and q_6 are indistinguishable because they have the exact same incoming transitions. As a result, reaching one state always implies also reaching the other.



Analogically in the example below the states also are indistinguishable but this time the outgoing transitions are the same. Hence the effects of reaching one state are equivalent to the other.



The states q_5 and q_6 can be merged without affecting the language of automaton. Such a pseudo-minimisation procedure has been chosen, because it works especially well when combined with Glushkov's construction. The process of merging indistinguishable states is analogical to the process of isolating common parts of regular expression. For example the following

$$ab + aab + ac(b + bb)$$

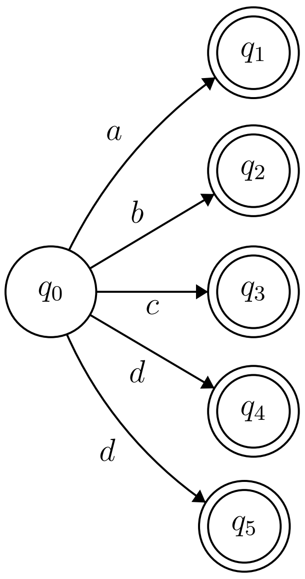
could be shortened to

$$a(\epsilon + a + c(\epsilon + b))b$$

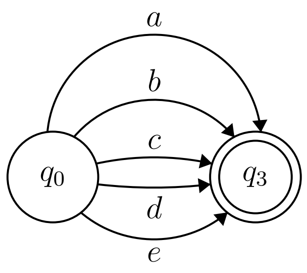
by isolating the common prefix a and suffix b. Extracting common prefixes exactly correspond to merging states with identical incoming transition, while common suffixes correspond to states with the same outgoing transitions. Sometimes there are cases that do not have any common prefix/suffix in the regular expression itself but still produce indistinguishable states in the automaton. For instance consider

$$a + b + c + d + e$$

which yields automaton



that could be minimised down to only two states



Interestingly, the regular expression could not be minimised any further. Hence we observe that our procedure does more than a simple syntactic manipulation could achieve.

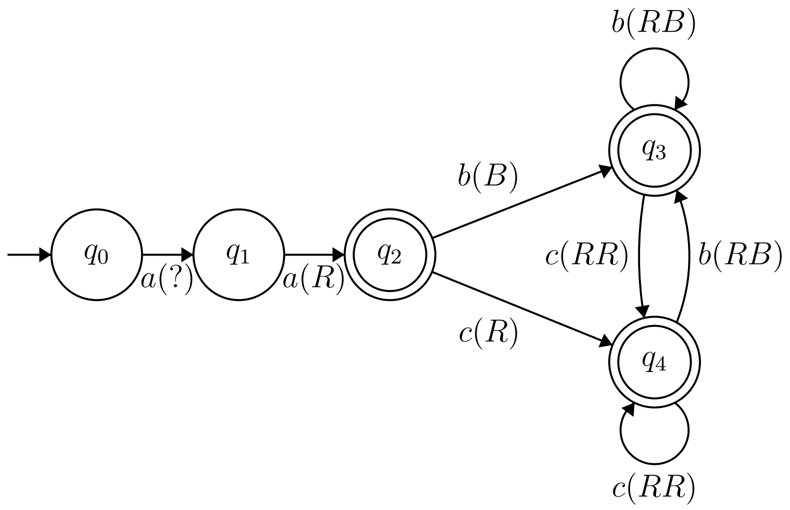
Glushkov's construction has one more advantage. Because every symbol becomes a state, it's very easy for the user to predict the exact placement of transitions between them. This way, it's possible to embed any property of the transition directly inside the regular expression. For example, suppose that we want to assign come colours to all transitions. Let B stand for blue and R for red. Then we can embed these colours in an expression like below

$$aRa(Bb + Rc)R*$$

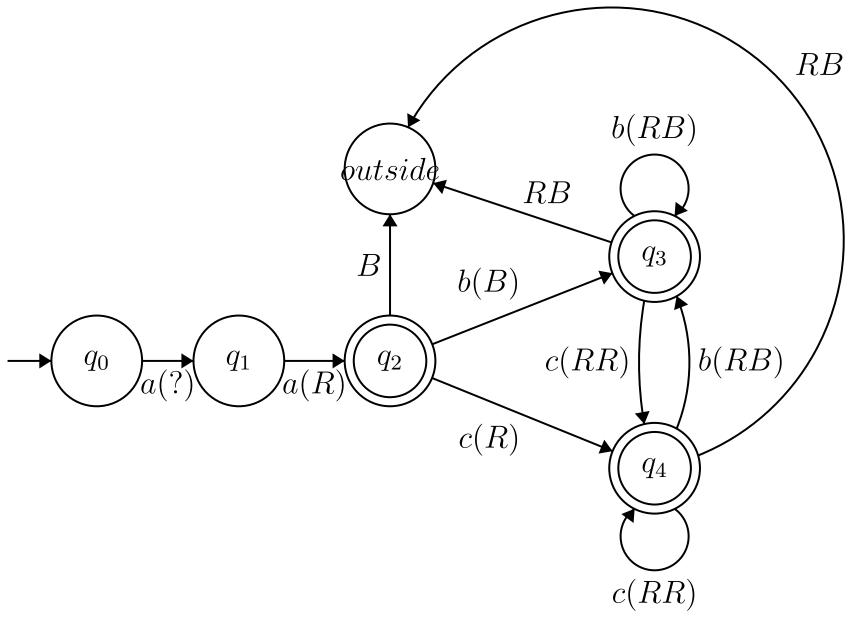
which becomes

$$q_1Rq_2(Bq_3 + Rq_4)R * B$$

We know that after q_1 we can read q_2 and along the way we have to cross the color R, because it stands in between q_1 and q_2. Hence it determines that the transition from q_1 to q_2 should be red. Similarly transition from q_2 and q_3 must be blue and from q_2 and q_4 is red. The transition from q_3 to q_3 will have colour RB, because R is under the Kleen closure and B is right before q_3. At this point we notice that there must be defined some way of mixing colours. In other words, any meta-information that we wish to embed in our regular expressions must at least form a monoid. The graph resulting from our example looks as follows



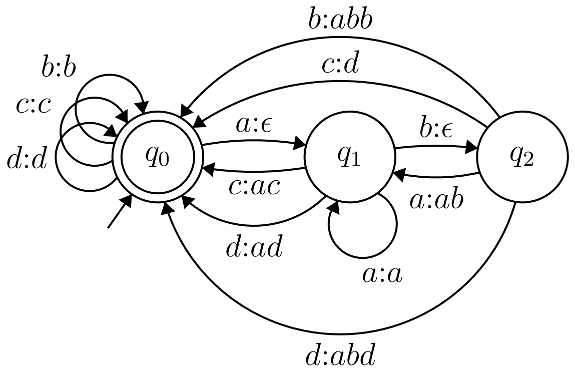
Because colours form a monoid, we can use neutral element as a default value for all unspecified edges like $a(?)$. Moreover, it is also possible to attach meta-information to final states. We could imagine that an accepting state is nothing more than a state with a special outgoing transition that goes "outside" of automaton and has no target state. In our example the colours of such "final" transitions are as follows



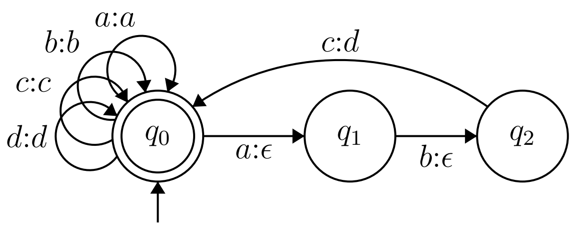
In Solomonoff, the meta-information of interest are transition outputs. We could imagine that strings composed of R and B symbols are the output of a transducer. At this point the reader should appreciate how beautifully and naturally subsequential transducers (automata with output produced at accepting states) are derived from Glushkov's construction. As soon as we enrich regular expressions with meta-information, the state outputs emerge as if they were always there, merely hiding from the view.

Glushkov's construction together with minimisation procedure and embedded meta-information allowed for efficient implementation of union, concatenation and Kleene closure together with output strings. In order to make the compiler applicable to real-life linguistic problems it also must support context dependent rewrites. This is a heavyweight operation that often constitutes a major performance bottleneck. Our goal was to make it as efficient as possible. In order to solve this issue we developed a special lexicographic semiring [5]. This is an innovative solution never seen before.

Suppose we want to replace every occurrence of abc with d. Even for such a simple scenario, the corresponding transducer will look rather complex

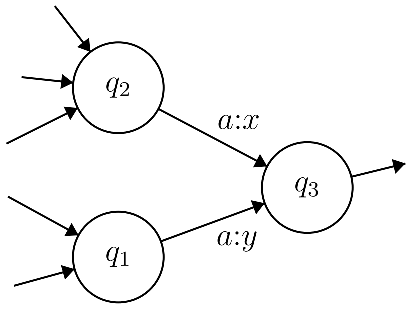


A much simpler alternative would be the nondeterministic transducer

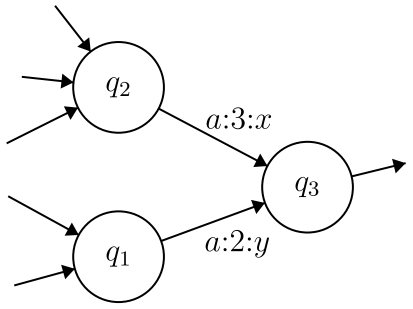


The problem with such a solution is that for input strings like $aabcb$ both outputs $aabcb$ and adb are generated. If only there was a way to assign priority to some outputs in order to disambiguate them, then context dependent rewrites could be expressed by much simpler automata. This is precisely what lexicographic weights are for.

Consider the following fragment of automaton and suppose that it's possible to simultaneously nondeterministically reach both q_2 and q_1.

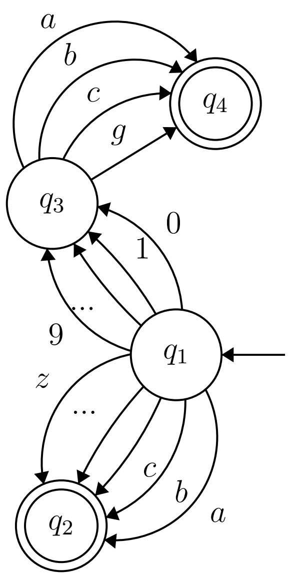


If the next input symbol is a, then the transducer will reach state q_3 and generate two outputs — one ending in x and the other in y. If the automaton later accepts it will produce at least two ambiguous outputs. Lexicographic weights allow us to assign priority to different transitions. In the following example, only the output ending in x will reach state q_3 and the other ending in y will be discarded because it came to q_3 over transition with lower weight.

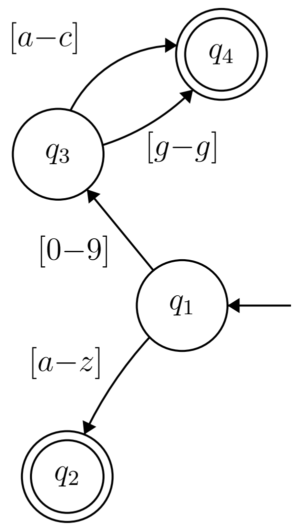


Lexicographic weights allow to make the transducers even more compact. We proved that there exist weighted automata exponentially smaller than even the smallest unweighted nondeterministic ones. In the presence of lexicographic weights, context-dependent rewrites become expressible directly in Glushkov's construction, without the need for calling any "external operations".

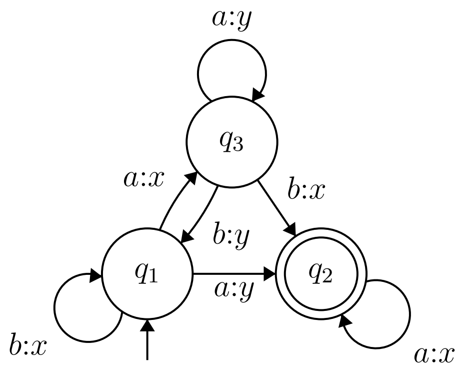
In the tasks of natural language processing it's common to work with large alphabets. User input might contain unexpected sequences like math symbols, foreign words or even emojis and other UNICODE entities. The regular expressions should handle such cases gracefully, especially when using wildcards such as `.*` or `\p{Lu}`. Representation of large character classes is a challenging task for finite state automata. In order to use the dot wildcard in UNICODE, the automaton would require millions of transitions, one for each individual symbol. In order to optimise this, Solomonoff employs symbolic transitions []. While classic automata have edges labelled with individual symbols, our transducers have edges that span entire ranges. A range is easy to encode in computer memory. It's enough to store the first and last symbol. Below is an example of classical finite state automaton



and an equivalent symbolic automaton that uses closed ranges of symbols as transition predicates



Compilation of transducers is the core part of our library but in order to make the automata useful there must be a way to execute them. Deterministic transducers and Mealy machines could be evaluated in linear time. Nondeterministic automata are more expensive. The computation might branch and the automaton could be in multiple states simultaneously. Using dynamic programming it's possible to build a table with rows representing consecutive symbols of input string and each column keeping track of one state. At each step of execution, one input symbol is read and one row in the table is filled based on the contents of the previous row. Below is an example of nondeterministic finite state automaton and its corresponding evaluation table after reading input string *aba*.



By the end of evaluating such a table, it's enough to scan the last row to find a column of some accepting state. In the above example, such a state is q_2 and we can observe that it's active in the last row.

symbol	q_1	q_2	q_3
initial configuration	1	0	0
a	0	1	1
b	1	1	0
a	0	1	1

In case of transducers, we not only want to know, whether the string was accepted or not but also the output generated along the way. In order to do this the table can be backtracked from the accepting state backwards. For the backtracking step to be possible, we need to store information about the source state of each taken transition.

symbol	q_1	q_2	q_3
initial configuration	q_1	$\emptyset$	$\emptyset$
a	$\emptyset$	q_1	q_1
b	q_3	q_3	$\emptyset$
a	$\emptyset$	q_1	q_1

Assuming that every transition is uniquely determined by its symbol, source state and target state (in other words $\delta{:}Q \times \Sigma \times Q \to \Gamma^*$), it becomes possible to use such an evaluation table to find the exact accepting path in automaton and collect all the outputs printed along the way.

This algorithm has been made even more efficient by using techniques from graph theory. Every automaton is a directed graph that could be represented as an adjacency matrix [**?**]. An example is shown below.

adjacency	q_1	q_2	q_3
q_1	b:x	a:y	a:x
q_2	$\emptyset$	a:x	$\emptyset$
q_3	b:y	b:x	a:y

Sparse graphs can be optimised and instead of using matrix, it's possible to only store the list of adjacent vertices.

$$(q_1, b{:}x, q_1), (q_1, a{:}y, q_2), (q_1, a{:}x, q_3), (q_2, a{:}x, q_2), (q_3, b{:}y, q_3), ...$$

Nondeterministic automata very often are the perfect example of sparse graphs. Hence instead of using an evaluation table, it's better to use a list of states nondeterministically reached at any given step of evaluation.

symbol	list
initial configuration	q_1
a	q_2, q_3
b	q_1, q_2
a	q_2, q_3

The backtracking can be made efficient by storing pointer to the source state of any taken transition.

symbol	list
initial configuration	$q_1($ from $q_1)$
a	$q_2($ from $q_1), q_3($ from $q_1)$
b	$q_1($ from $q_3), q_2($ from $q_3)$
a	$q_2($ from $q_1), q_3($ from $q_1)$

Glushkov's construction relies on building three sets: the set of initial symbols, final symbols and 2-factor [11] strings (that is, all substrings of length 2). The early prototype versions of Solomonoff would represent sets as bitsets, where each bit specifies whether an element belongs to the set or not. This representation simplified implementation of many algorithms but was highly inefficient. Later implementation used hash sets instead. While hash maps have constant insertions and deletions, they ensure it at the cost of larger memory consumption. During benchmarks on real-life datasets, Solomonoff would often run out of memory and crash. The final version uses signly linkged graphs backed by arrays. This provides the highest efficiency with the smallest memory footprint but it is non-trivial to implement. The optimisation relies on representing sparse sets as arrays of elements. This approach proved to be the best choice, because Glushkov's construction inherently ensures uniqueness of all inserted elements (hence using hashes to search for duplicates before insertion was not necessary) and it does not use deletions. As a result any array was guaranteed to behave like a set.

The standard definition of Glushkov's construction produces a set of 2-factors as its output. Then a separate procedure would be necessary to collect all such strings and convert them into a graph of automaton. We found a way to make it more efficient and build the graph directly. The step of building 2-factors was entirely bypassed. This provided even further optimisation.

One of the core features of Solomonoff is the algorithm for detecting ambiguous nondeterminism. Advance-and-delay procedure can check functionality of automaton in quadratic time. While the compiler does provide implementation of this procedure, it is not used for checking functionality. Instead we use a simpler algorithm for checking strong functionality of lexicographic weights [5]. While the performance difference between the two is negligable, the advantage of our approach comes from better error messages in case of nondeterminism. If some lexicographic weights are in conflict with each other, the compiler can point the user to the exact line and column of text where the conflict arises, whereas advance-and-delay might miss the origin of the problem and only fail further down the line.

2 Gluszkov's construction

In this section we provide formal and mathematically rigorous presentation of Glushkov's construction.

The theory of automata is primarily founded on the theory of semigroups and monoids. A set X together with operation $\oplus$:$X \times X \to X$ forms a semigroup if it satisfies associativity $(x_1 \oplus x_2) \oplus x_3 = x_1 \oplus (x_2 \oplus x_3)$. If the semigroup contains a neutral element 0_X such that $0_X \oplus x = x \oplus 0_X = 0_X$ then it forms a monoid. For example the set of positive (zero excluded) integers together with addition operation forms a semigroup, while a set of non-negative integers (zero included) forms a monoid. Because of associativity, the placement of brackets does not matter and can be omitted altogether. As a result instead of writing

$$((x_1 \oplus (x_2 \oplus x_3)) \oplus x_4) \oplus (x_5 \oplus x_6)$$

we can just write

$$x_1 x_2 x_3 x_4 x_5 x_6$$

Every element of X can be presented in such a way. String of elements of X connected together with the operation $\oplus$ is called a word. There might be more than one word denoting the same element. For example, the number 6 could be framed as $0+1+2+3$ or $0+6+0$ or just 6. All of those are different words denoting the same number. A free semigroup is one in which no two distinct words denote the same element. The canonical example of free semigroup is the set of all non-empty strings under concatenation. No two strings are equal, unless their notations are syntactically the same. Definition of free monoid is similar to free semigroup, with the exception that neutral elements do not change denotation of a word. For example the following words must all be equal $0+3+4=3+0+4=3+4+0+0$ but all of $4+3 \neq 3+4 \neq 1+1+1+4 \neq 1+5+1$ must be different from each other. The standard example of free monoid is the set of all strings with concatenation operation. No two strings are equal, except for the ones that are concatenated with the empty string ϵ.

Given two monoids X and Y with operations $\oplus$ and $\odot$ respectively, it's possible to build a new one by performing their direct product $X \times Y$. Their joint operation is defined as $(x_1, y_1) \cdot (x_2, y_2) = (x_1 \oplus x_2, y_1 \odot y2)$.

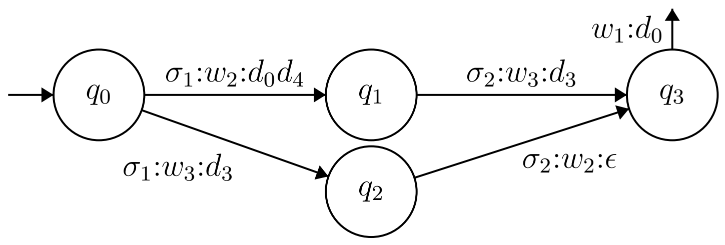
Let Σ be the (not necessarily finite) alphabet of automaton. Let χ be the set of subsets of Σ that we will call ranges of Σ. Let $\overline{\chi}$ be the closure of χ under countable union and complementation (so it forms a sigma algebra). For instance, imagine that there is a total order on Σ and χ is the set of all intervals in Σ. Now we want to build an automaton whose transitions are not labelled with symbols from Σ, but rather with ranges from χ. Union $\chi_0 \cup \chi_1$ of two elements from χ "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 δ. It might be countably infinite, hence it's natural that $\overline{\chi}$ should be closed under countable union. Therefore, χ 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 Σ. An example of discrete $\overline{\chi}$ would be a finite set Σ with all UNIX-style ranges `[`σ`-`σ'`]` included in χ.

Transducers with input Σ^* and output Γ^* can be seen as a finite state automaton working with single input $\Sigma^* \times \Gamma^*$. Therefore we can treat every pair of symbols (σ, γ) 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 σ as (σ, ϵ) and every γ as (ϵ, γ). Then 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 : should always be a string of Γ^* and writing entire formulas (like $\sigma{:}\gamma_1 + \gamma_2^*$) is not allowed. This restriction will later simplify Glushkov's algorithm.

We define $\mathcal{A}^\Sigma$ to be the set of atomic characters. For instance we could choose $\mathcal{A}^\Sigma = \Sigma \cup \{\epsilon\}$ for FSA/transducers and $\mathcal{A}^\Sigma = \chi$ for ranged automata.

We call $RE^{\Sigma:D}$ the set of all regular expression formulas with the underlying set of atomic characters $\mathcal{A}^\Sigma$ 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 ϕ and ψ are $RE^{\Sigma:D}$ formulas and $d \in D$, then union $\phi + \psi$, concatenation $\phi \cdot \psi$, Kleene closure ϕ^* and output concatenation $\phi{:}d$ are $RE^{\Sigma:D}$ formulas as well. Define $V^{\Sigma:D}{:}RE^{\Sigma:D} \to \Sigma^* \times D$ to be the valuation function:
$V^{\Sigma:D}(\phi + \psi) = V^{\Sigma:D}(\phi) \cup V^{\Sigma:D}(\psi)$
$V^{\Sigma:D}(\phi \cdot \psi) = V^{\Sigma:D}(\phi) \cdot V^{\Sigma:D}(\psi)$
$V^{\Sigma:D}(\phi^*) = (\epsilon, \epsilon) + V^{\Sigma:D}(\phi) + V^{\Sigma:D}(\phi)^2 + ...$
$V^{\Sigma:D}(\phi{:}d) = V^{\Sigma:D}(\phi) \cdot (\epsilon, d)$
$V^{\Sigma:D}(a) = a$ where $a \in \mathcal{A}^{\Sigma:D}$
Some notable properties are:
$x{:}y_0 + x{:}y_1 = x{:}(y_0 + y_1)$
$x{:}\epsilon + x{:}y + x{:}y^2 ... = x{:}y^*$
$(x{:}y_0)(\epsilon{:}y_1) = x{:}(y_0 y_1)$
$x_0{:}(y_0 y') + x_1{:}(y_1 y') = (x_0{:}y_0 + x_1{:}y1) \cdot (\epsilon{:}y')$
$x_0{:}(y' y_0) + x_1{:}(y' y_1) = (\epsilon{:}y') \cdot (x_0{:}y_0 + x_1{:}y1)$
Therefore we can see that expressive power with and without : is the same.

It's also possible to extend regular expressions with weights. Let $RE_W^{\Sigma:D}$ be a superset of $RE^{\Sigma:D}$ and W be the set of weight symbols. If $\phi \in RE_W^{\Sigma \to D}$ and $w_0, w_1 \in W$ then $w_0\phi$ and ϕw_1 are in $RE_W^{\Sigma \to D}$. This allows for inserting weight at any place. For instance, the automaton below



could be expressed using

$$((\sigma_1{:}d_0d_4)w_2(\sigma_2{:}d_3)w_3 + (\sigma_1{:}d_3)w_3\sigma_2 w_2){:}d_0$$

The definition of $V^{\Sigma:D}(\phi w)$ depends largely on W but associativity $(\phi w_1)w_2 = \phi(w_1 + w_2)$ should be preserved, given that W is an additive monoid. This also implies that $w_1\epsilon w_2 = w_1w_2$, which is semantically equivalent to the addition $w_1 + w_2$.

First step of Glushkov's algorithm is to create a new alphabet Ω in which 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 $\psi \in RE^{\Omega\to D}_W$ along with mapping $\alpha{:}\Omega \to \mathcal{A}^\Sigma$. This mapping will remember the original atomic character, before it was rewritten to a unique symbol in Ω. For example

$$\phi = (\epsilon{:}x_0)x_0(x_0{:}x_1x_3)x_3w_0 + (x_1x_2)^*w_1$$

will be rewritten as

$$\psi = (\epsilon{:}x_0)\omega_1(\omega_2{:}x_1x_3)\omega_3w_0 + (\omega_4\omega_5)^*w_1$$

with $\alpha = \{(\omega_1, x_0), (\omega_2, x_0), (\omega_3, x_3), (\omega_4, x_1), (\omega_5, x_2)\}$.

Next step is to define the function $\Lambda{:}RE^{\Omega\to D}_W \rightharpoonup (D \times W)$. It returns the output produced for empty word ϵ (if any) and weight associated with it. (We use the symbol $\rightharpoonup$ to highlight the fact that Λ is a partial function) For instance in the previous example the 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 $D{\times}W$ like a monoid defined as $(y_0, w_0){\cdot}(y_1, w_1) = (y_0y_1, w_0{+}w_1)$. We also admit $\emptyset$ as multiplicative zero, which means that $(y_0, w_0) \cdot \emptyset = \emptyset$. We denote W's neutral element as 0. This facilitates recursive definition:
$\Lambda(\psi_0 + \psi_1) = \Lambda(\psi_0) \cup \Lambda(\psi_1)$ if at least one of the sides is $\emptyset$, otherwise error
$\Lambda(\psi_0\psi_1) = \Lambda(\psi_0) \cdot \Lambda(\psi_1)$
$\Lambda(\psi_0{:}y) = \Lambda(\psi_0) \cdot (y, 0)$
$\Lambda(\psi_0w) = \Lambda(\psi_0) \cdot (\epsilon, w)$
$\Lambda(w\psi_0) = \Lambda(\psi_0) \cdot (\epsilon, w)$
$\Lambda(\psi_0^*) = (\epsilon, 0)$ if $(\epsilon, w) = \Lambda(\psi_0)$ or $\emptyset = \Lambda(\psi_0)$, otherwise error
$\Lambda(\epsilon) = (\epsilon, 0)$
$\Lambda(\omega) = \emptyset$ where $\omega \in \Omega$

Next step is to define $B{:}RE^{\Omega\to D}_W \to (\Omega \rightharpoonup D \times W)$ which for a given formula ψ returns set of Ω characters that can be found as the first in any string of $V^{\Omega\to D}(\psi)$ 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 ω_1. Therefore $(\omega_1,(x_0,0))$ and $(\omega_3,(\epsilon,0))$ are the result of $B(\psi)$. For better readability, we admit operation of multiplication $\cdot{:}(\Omega \rightharpoonup D\times W)\times(D\times W)\to(\Omega\rightharpoonup D\times W)$ that performs monoid multiplication on all $D\times W$ elements returned by $\Omega\rightharpoonup D\times W$.
$B(\psi_0+\psi_1)=B(\psi_0)\cup B(\psi_1)$
$B(\psi_0\psi_1)=B(\psi_0)\cup\Lambda(\psi_0)\cdot B(\psi_1)$
$B(\psi_0 w)=B(\psi_0)$
$B(w\psi_0)=(\epsilon,w)\cdot B(\psi_0)$
$B(\psi_0^*)=B(\psi_0)$
$B(\psi_0{:}d)=B(\psi_0)$
$B(\epsilon)=\emptyset$
$B(\omega)=\{(\omega,(\epsilon,0))\}$
It's worth noting that $B(\psi_0)\cup B(\psi_1)$ always yields function (instead of a 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 $E{:}RE_W^{\Omega\to D}\to(\Omega\rightharpoonup D\times W)$, 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:
$E(\psi_0+\psi_1)=E(\psi_0)\cup E(\psi_1)$
$E(\psi_0\psi_1)=E(\psi_0)\cdot\Lambda(\psi_1)\cup B(\psi_1)$
$E(\psi_0 w)=E(\psi_0)\cdot(\epsilon,w)$
$E(w\psi_0)=E(\psi_0)$
$E(\psi_0^*)=E(\psi_0)$
$E(\psi_0{:}d)=E(\psi_0)\cdot(d,0)$
$E(\epsilon)=\emptyset$
$E(\omega)=\{(\omega,(\epsilon,0))\}$

Next step is to use B and E to determine all two-character substrings that can be encountered in $V^{\Omega\to D}(\psi)$. Given two functions $b,e{:}\Omega\rightharpoonup D\times W$ we define product $b\times e{:}\Omega\times\Omega\rightharpoonup D\times W$ such that for any $(\omega_0,(y_0,w_0))\in b$ and $(\omega_1,(y_1,w_1))\in c$ there is $((\omega_0,\omega_1),(y_0y_1,w_0+w_1))\in b\times e$. Then define $L{:}RE_W^{\Omega\to D}\to(\Omega\times\Omega\rightharpoonup D\times W)$ as:
$L(\psi_0+\psi_1)=L(\psi_0)\cup L(\psi_1)$
$L(\psi_0\psi_1)=L(\psi_0)\cup L(\psi_1)\cup E(\psi_0)\times B(\psi_1)$
$L(\psi_0 w)=L(\psi_0)$
$L(w\psi_0)=L(\psi_0)$
$L(\psi_0^*)=L(\psi_0)\cup E(\psi_0)\times B(\psi_0)$
$L(\psi_0{:}d)=L(\psi_0)$

$L(\epsilon) = \emptyset$
$L(\omega) = \emptyset$
One should notice that all the partial functions produced by B, E and L have finite domains, therefore they are effective objects from a computational point of view.

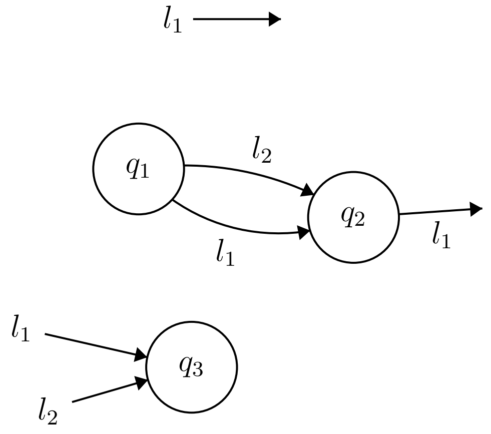
The last step is to use results of L, B, E, Λ and α to produce automaton $(Q, q_\epsilon, W, \Sigma, D, \delta, \tau)$ with
$\delta{:}Q \times \Sigma \to (Q \rightharpoonup D \times W)$
$\tau{:}Q \rightharpoonup D \times W$
$Q = \{q_\omega{:}\omega \in \Omega\} \cup \{q_\epsilon\}$
$\tau = E(\psi)$
$(q_{\omega_0}, \alpha(\omega_1), q_{\omega_1}, d, w) \in \delta$ for every $(\omega_0, \omega_1, d, w) \in L(\psi)$
$(q_\epsilon, \alpha(\omega), q_\omega, d, w) \in \delta$ for every $(\omega, d, w) \in B(\psi)$

This concludes our extended version of Glushkov's construction. This formal definition does not include the plentiful optimisation techniques that we implemented. In particular, usually this algorithm treats $\Omega \times \Omega$ as 2-factor strings $\Omega\Omega$. It's possible to instead interpret this in a graph-theoretic manner, where every element of Ω is a state and the product $\Omega \times \Omega$ defines an adjacency matrix. The function L produces an assignment $\Omega \rightharpoonup D \times W$ which to every edge associates its label. Graph theory has countless tools for optimising graph-based algorithms and data structures. While adjacency matrix works best for dense graphs, different data representations are usually used for sparse graphs. Finite state automata and transducers are the canonical example of sparse graphs, because the number of possible edges is bounded by the size of the alphabet, whereas the number of states is expected to be much larger. This graph-theoretic approach can be made more formally rigorous.

The formal definition of a graph is a tuple (Q, δ) where Q is the set of vertices and $\delta \subset Q \times Q$ is the set of edges. Most books by convention use letters V and E to refer to the set of vertices and edges respectively. Here the naming Q and δ has been used purposely to highlight the connection between graphs and automata. The formal definition of a directed graph (digraph for short) is the same as the previous one, with the additional assumption that δ is a set of ordered pairs, while undirected graphs assume the pairs to be unordered.

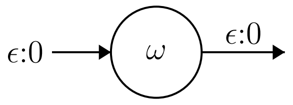
We define a labelled graph as (Q, δ) where the set of edges has the form $\delta \subset Q \times \mathbb{L} \times Q$ for some set of labels $\mathbb{L}$. Every multitape automaton can be viewed as a labelled graph. For example finite state automaton with single input tape is usually defined in terms of $\delta{:}Q \times \Sigma \to Q$, which could be rewritten as $\delta \subset Q \times \Sigma \times Q$. Mealy automata could alternatively be defined using $\delta \subset Q \times \Sigma \times \Gamma \times Q$. Sequential transducers are of the form $\delta \subset Q \times \Sigma \times \Gamma^* \times Q$. Nonsequential transducers could further be extended as $\delta \subset Q \times \Sigma^* \times \Gamma^* \times Q$. Interestingly, there is no difference between input tapes and output tapes. A multitape automaton with one input Σ and two outputs Γ, Δ can be formalized as $\delta \subset Q \times \Sigma \times \Gamma \times \Delta \times Q$ but the exact same formalization would be achieved if all Σ, Γ and Δ were input tapes. Hence the concept of "input" and "output" is an algorithmic distinction that appears in implementation of automaton but from the algebraic and graph-theoretic point of view the distinction is artificial.

We define partial graph as a labelled graph allowing partial edges. A full edge $(q_1, l, q_2) \in \delta$ has source vertex q_1, label l and target vertex q_2. A partial edge might be lacking source $(\emptyset, l, q_2)$, target $(q_1, l, \emptyset)$ or both $(\emptyset, l, \emptyset)$ but it must have a label. An edge that neither has any target nor source is called an epsilon edge. Those with no source are called incoming and those with no target are outgoing. Hence partial graphs could be formalized as (Q, δ) with edges in the form of $\delta \subset (Q \cup \{\emptyset\}) \times \mathbb{L} \times (Q \cup \{\emptyset\})$. Below is an example of a partial graph state diagram.

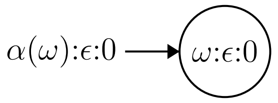


The partial graphs have a lot in common with Glushkov's construction. Let $D \times W$ be the set of labels. Then Λ function returns a single label $\Lambda(\psi) = (y, w)$, which looks very much like an instance of the epsilon edge $(\emptyset, y, w, \emptyset)$. The B function is responsible for collecting initial atomic symbols $\omega \in \Omega$ and their labels $(y, w) \in D \times W$, which resembles partial edges with no source $(\emptyset, y, w, \omega)$. Similarly for $(\omega, y, w) \in E(\psi) \subset \Omega \rightharpoonup D \times W$, which we could interpret as partial edge with no target $(\omega, y, w, \emptyset)$. The set of 2-factors $L(\psi) \subset (\Omega \times \Omega \rightharpoonup D \times W)$ could be interpreted as a set of full labelled edges $\Omega \times D \times W \times \Omega$. As a result, instead of converting Λ, B, E and L to a subsequential transducer, we can directly obtain definition of a partial graph with no "post processing" required. This could be better seen if we combine all of the Λ, B, E and L functions into one large procedure G that returns the entire partial graph $G{:}RE_W^{\Omega \rightharpoonup D} \to (\Omega, \delta)$. We have several cases to consider.

Let's recall how each function behaves on the atomic symbol ω. We can see that G should return a partial graph with no full edges
$L(\epsilon) = \emptyset$
one outgoing edge
$E(\omega) = \{(\omega, (\epsilon, 0))\}$
one incoming edge
$B(\omega) = \{(\omega, (\epsilon, 0))\}$
and no epsilon edge
$\Lambda(\omega) = \emptyset$
Hence it could be presented as the following very simple single-state graph:



In order to use such a graph as an automaton it's enough to treat all the outgoing edges state's subsequential output and the input label of every full or incoming edge is directly determined by the target state. Hence this graph becomes the following automaton:

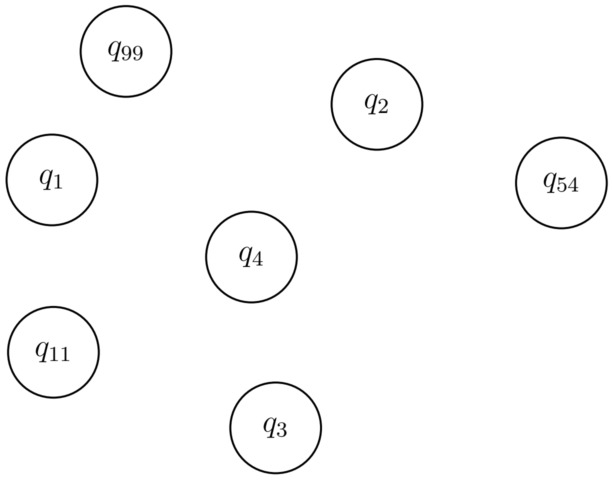


It is worth pointing out that there will always be at most only one outgoing edge per state and at most one global epsilon edge. This directly follows from the definition of E as function $E(\psi){:}\Omega \rightharpoonup D \times W$ rather than relation $E(\psi){:}\Omega \times D \times W$ and because Λ returns only a single element.

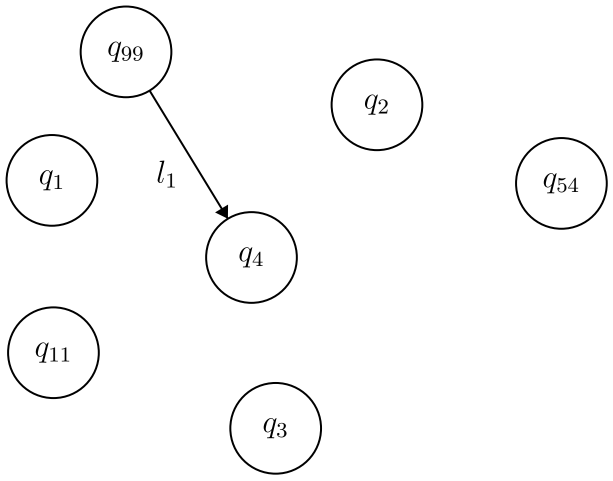
The set Ω is directly treated as the set of states Q. An input string $\sigma_1\sigma_2...\sigma_n \in \Sigma^*$ is accepted by the partial graph if there exists a path
$(\emptyset, y_0, w_0, \omega_1) \to (\omega_1, y_1, w_1, \omega_2) \to ... \to (\omega_n, y_n, w_n, \emptyset)$ starting in incoming edge $(\emptyset, y_0, w_0, \omega_1)$, ending in outgoing edge $(\omega_n, y_n, w_n, \emptyset)$ and such that for

every i the input symbol σ_i equals to the symbol of target state $\alpha(\omega_i)$. The produced output is determined by $y_0 y_1 ... y_n$.

It's possible to omit the use of α mapping entirely during the implementation. There is no need to linearise ψ or build Ω. Instead we could use a special data structure of singly-linked graphs. Every vertex itself is an array holding its own outgoing (full) edges. Such representation of graphs has certain benefits. Unlike in most other graph data structures (adjacency matrices and lists) the singly-linked graphs don't have a well defined set of vertices. There could be millions of vertices allocated in computer memory and none of them connected to any other.



Each of those individual vertices could be thought of as a separate graph in and of itself but the distinction is blurry. As soon as we add some connection



the two graphs suddenly merge into one (or perhaps, one is a subgraph of the other, since it's possible to reach q_4 from $q_9 9$ but not the other way around). We might think of one "infinite supergraph" containing all vertices that have ever been created or could potentially be created in the future. Hence it makes more sense to speak of connected components rather than the "supergraph" itself. Given some set of incoming partial transitions $B \subset \{\emptyset\} \times \mathbb{L} \times Q$, the connected component induced by B is the set $\overline{B}$ defined as subset of Q containing all the reachable vertices, starting from some initial edge in B.

Every time we encounter some symbol σ, we allocate a new vertex q and initialize it as an empty array $q = []$. Next we create a set of incoming edges B containing only a single edge $B = \{(\emptyset, \epsilon, 0, q_1)\}$. Similarly we also need to keep track of outgoing edges $E = \{(q_1, \epsilon, 0, \emptyset)\}$. The $G(\sigma)$ function will return a partial graph $(\overline{B}, B \cup E)$ but for technical details (presented in upcoming paragraphs) the sets B and E should be kept in their own separate lists. There is no need to store the full edges in their own list, since they are stored in each corresponding vertex. The set of vertices $\overline{B}$ is implied by B and also doesn't need to be stored in computer memory explicitly. Hence $(\overline{B}, B \cup E)$ is merely the formal presentation of a graph but in the actual implementation a data structure like (B, E) is used instead.

We have a full formal and algorithmic description of $G(\sigma)$. The next case to consider is the union $G(\psi_0 + \psi_1)$. We need to combine the full edges from both subgraphs
$L(\psi_0 + \psi_1) = L(\psi_0) \cup L(\psi_1)$
as well as the respective outgoing
$E(\psi_0 + \psi_1) = E(\psi_0) \cup E(\psi_1)$
and incoming edges
$B(\psi_0 + \psi_1) = B(\psi_0) \cup B(\psi_1)$
The epsilon edges cannot be combined in any way hence union might result in errors when both subgraphs simultaneously contain their own epsilon edges
$\Lambda(\psi_0 + \psi_1) = \Lambda(\psi_0) \cup \Lambda(\psi_1)$
We can combine all of those operations into a single operation G on partial graphs. We first perform $G(\psi_0)$ and then $G(\psi_1)$. The result of $G(\psi_0 + \psi_1)$ does not require to perform any additional operations. The graph $G(\psi_0 + \psi_1)$ is merely one "supergraph" consisting of two disconnected components $G(\psi_0)$ and $G(\psi_1)$. Algorithmic implementation can be achieved by concatenating the list of incoming edges of the first subgraph with the other (and analogically for outgoing edges). Because we use singly-linked graphs as the backing

data structure, the set of reachable vertices is automatically implied and does not require to be updated. As a result union operation is $O(1)$ with respect to size of the graph and its memory footprint is kept low.

The next case to consider is the concatenation $G(\psi_0\psi_1)$. First we need to compute $G(\psi_0)$ to obtain the sets B_{ψ_0}, E_{ψ_0}, L_{ψ_0} and Λ_{ψ_0} of its incoming, outgoing, full and epsilon edges respectively. Analogically for $G(\psi_1)$. The L function performs product
$\Lambda(\psi_0\psi_1) = \Lambda(\psi_0) \cdot \Lambda(\psi_1)$
This operation can be redefined in terms of partial edges. If the set of labels $\mathbb{L}$ is a monoid, then multiplication of edges becomes possible. Outgoing edge $(q_1, l_1, \emptyset)$ multiplied together with incoming edge $(\emptyset, l_2, q_2)$ will give us a full edge $(q_1, l_1 l_2, q_2)$. Similarly, multiplication of epsilon edge $(\emptyset, l_3, \emptyset)$ with either outgoing edge $(q_1, l_1, \emptyset)$ or incoming edge $(\emptyset, l_2, q_2)$ yields $(q_1, l_3 l_1, \emptyset)$ or $(\emptyset, l_3 l_2, q_2)$ respectively. If we extend those operations to work on entire sets of edges
$X \cdot Y = \{x \cdot y{:}x \in X \text{ and } y \in Y\}$
then we can introduce multiplication (concatenation) of partial graphs.
$G(\psi_0) \cdot G(\psi_1) = (\overline{B_{\psi_0}}, L_{\psi_0} \cup L_{\psi_1} \cup E_{\psi_0} \cdot B_{\psi_1} \cup \Lambda_{\psi_0} \cdot B_{\psi_1} \cup E_{\psi_0} \cdot \Lambda_{\psi_1} \cup \Lambda_{\psi_0} \cdot \Lambda_{\psi_1})$
In the context of singly-linked graphs, the operation $E_{\psi_0} \cdot B_{\psi_1}$
for every $(q_0, y_0, w_0, \emptyset) \in E_{\psi_0}$ and $(\emptyset, y_1, w_1, q_1) \in B_{\psi_1}$ computes a full edge $(q_0, y_0 y_1, w_0 w_1, q_2)$ and inserts it to the list q_0. Analogically partial edges are computed and inserted to the respective lists as well.

The case of Kleene closure $G(\psi_0^*)$ is similar to concatenation. When working with singly linked graphs, Kleen closure behaves like concatenation of graph $G(\psi_0) \cdot G(\psi_0)$ with itself, except that epsilon edges require special handling.

The remaining cases of $G(\psi{:}d)$, $G(\psi w)$ and $G(w\psi)$ are achieved by multiplying outgoing edges with $(d, 0)$, (ϵ, w) with incoming edges and outgoing edges with (ϵ, w) respectively. It should be noted that labels $\mathbb{L}$ can act on partial graphs. Left action $l \cdot G$ multiplies $l \cdot b$ with all incoming edges of G and the epsilon edge $l \cdot \lambda$. Similarly right action $G \cdot l$ multiplies $e \cdot l$ with all outgoing edges of G and the epsilon edge $\lambda \cdot l$.

We conclude this construction with a few notes about performance and possible extensions. As it can be noticed, by eliminating the need for computing linearised alphabet Ω, the algorithm became fully parallelizable. Any subexpression of the original regular expression can be compiled independently to the rest. Singly-linked graphs guarantee that no reallocations of states are necessary. The weights and outputs only act on partial edges,

hence once a full edge is computed it is never mutated. As a result, the algorithm never has the need to revisit already compiled parts of the automaton. In such sense the construction is fully linear and all individual operations are of $O(1)$ complexity. There are exactly as many states as there are input symbols, hence memory consumption is also linear. The produced automata are epsilon-free, except for the only one global partial epsilon edge. The compiled singly-linked graph forms an automaton in and of itself, hence no conversion from 2-factors to automata is necessary, like it was in the case of "standard" Glushkov's construction. Moreover, it's straightforward to extend the compilation with custom "external" functions
$G(F(\psi)) = F_{custom\_implementation}(G(\psi))$
This allows for adding non-standard operations like subtraction, composition, inversion, powerset and many others.

References

[1] M. Sipser, *Introduction to the Theory of Computation 3rd Edition.*

[2] S. Eilenberg, *Automata, Languages and Machines Vol. A.* Academic Press, 1974.

[3] ——, *Automata, Languages and Machines Vol. B.* Academic Press, 1976.

[4] G. H. Mealy, "A method for synthesizing sequential circuits," *Bell System Technical Journal*, 1955.

[5] A. Mendoza-Drosik, "Multitape automata and finite state transducers with lexicographic weights," *ArXiv*, vol. abs/2007.12940, 2020.

[6] K. U. S. Stoyan Mihov, *Finite-State Techniques: Automata, Transducers and Bimachines*, 2019.

[7] F. P. Mehryar Mohri and M. Riley, "Weighted finite-state transducers in speech recognition," *AT&T Labs – Research*, 2008.

[8] M. Mohri, *Weighted Finite-State Transducer Algorithms. An Overview.* Springer, 2004.

[9] ——, “Weighted finite-state transducer algorithms an overview,” *AT&T Labs*, 2004.

[10] 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.

[11] V. M. Glushkov, *The abstract theory of automata.* Russian Mathematics Surveys, 1961.

[12] P. W. Tsunehiko Kameda, “On the state minimization of nondeterministic finite automata,” *IEEE Transactions on Computers*, 1970.