Solomonoff Library Specification and Technical Documentation

Aleksander Mendoza-Drosik

January 25, 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.

1 Mealy automata and transducers

The standard definition of automaton found in introductory courses 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.

A Mealy machine 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 $\delta{:}Q \times \Sigma \to Q \times \Gamma$.

Such a model 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. 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 "one apple" should become "1 apple". The prefix preserving property would be too limiting because it often happens that the suffix of string has decisive impact on the translation. The phrase "once again" also starts with prefix "one" but it should not be translated as "1ce again"!

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. The 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.

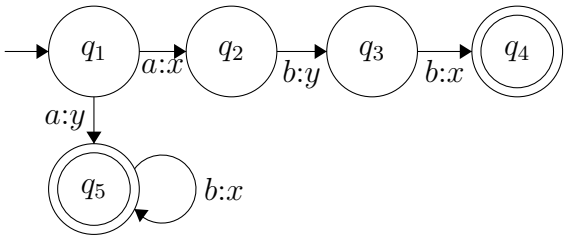
Over the course of research and development we have discovered that the power of multitape Mealy machines was still too limiting for our purposes. In particular we could define congruence classes on pairs of strings similar to those in Myhill-Nerode theorem. Then it can be easily noticed that as long as the number of output tapes is finite, the number of congruence classes must be finite as well. A very simple counter-example would the language that for every string a, aa, aaa,... respectively prints output c, bc, bbc,...and so on. It could not be expressed using only a 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

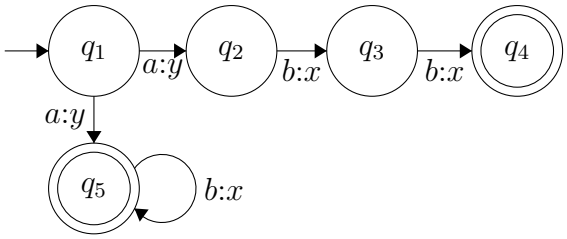
Mealy automata. Their definition allows the transition function to print output strings of arbitrary length $\delta{:}Q \times \Sigma \to Q^*$.

The nondeterministic transducers 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. Nondeterministic transducers may contain epsilon transitions, which could lead to infinite number of outputs 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 model of automata that was finally implemented in Solomonoff, are the functional nondeterministic transducers. Functionality of 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 path 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. Using advance-and-delay 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 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.

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

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

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

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

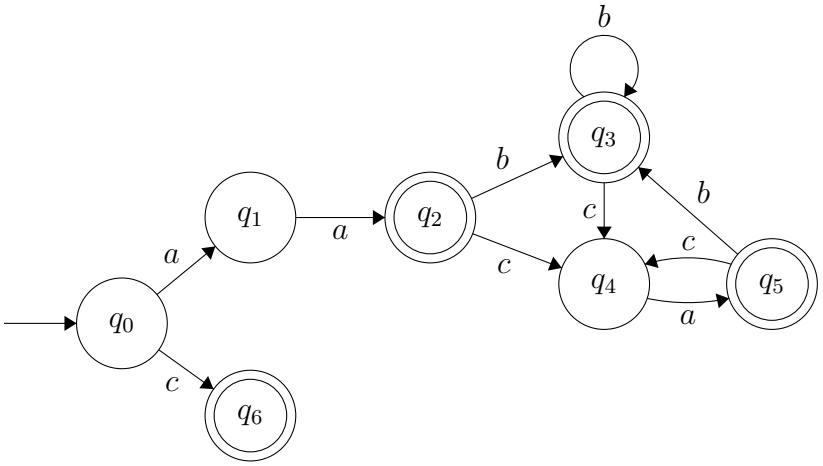
we first convert every symbol into a state and obtain the following intermediate "regular expression"

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

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

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

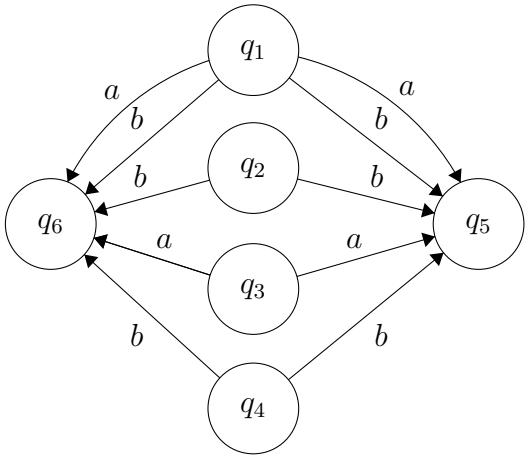
Next we analyse it and determine, which state can be reached from any other. After "reading" q_1 we can "read" q_2. After q_2 we can either read q_3 or q_4. After q_3 we can read either q_3 again or go to q_4. Analogically for the remaining states. We also check, which states can appear at the end of 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.



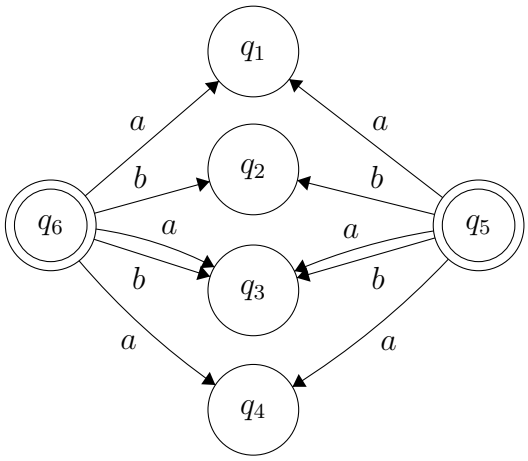
This way the produced automata are very small even without the need for minimisation. If the regular expression consists of n input symbols, then the resulting transducers has $n + 1$ states. Because there is one-to-one correspondence between regular expression symbols and automata states, we are able 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. 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

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

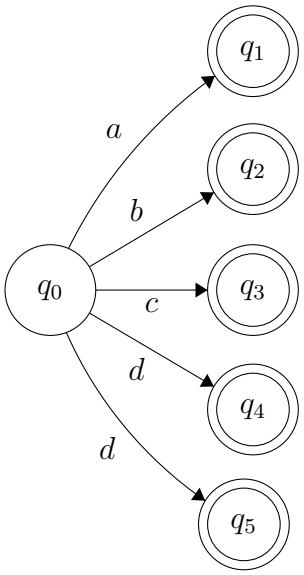
could be shortened to

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

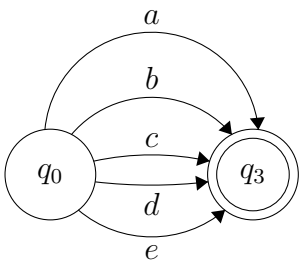
by isolating the common prefix a and suffix b. Extracting common prefixes exactly correspond to merging states with identical incoming transition, while common suffixes correspond to states with the same outgoing transitions. Sometimes there are cases that do not have any common prefix/suffix in the 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.

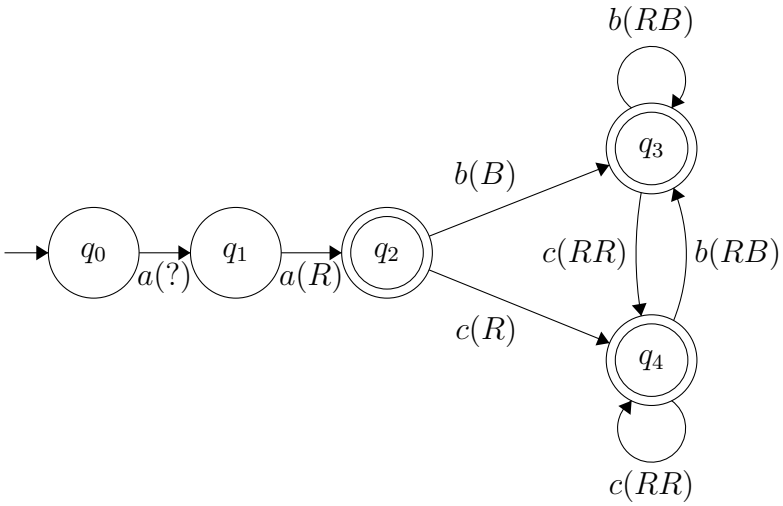
Glushkov's construction has one more advantage. Because every symbol becomes a state, it's very easy for the user to predict the exact placement of transitions between them. This way, it's easy to embed any property of the transition directly inside the regular expression. For example, suppose that we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

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

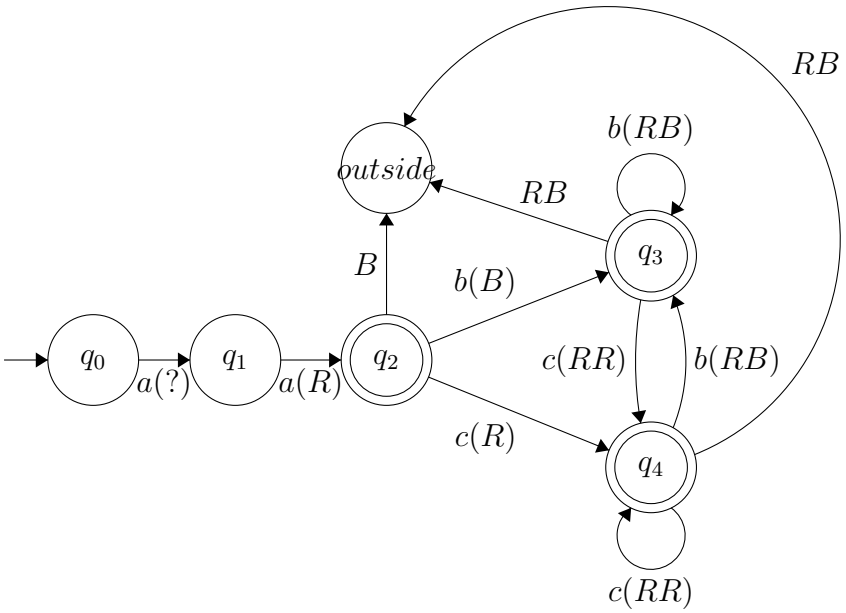
which becomes

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

We know that after q_1 we can read q_2 and along the way we have to cross the color R, because it stand 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



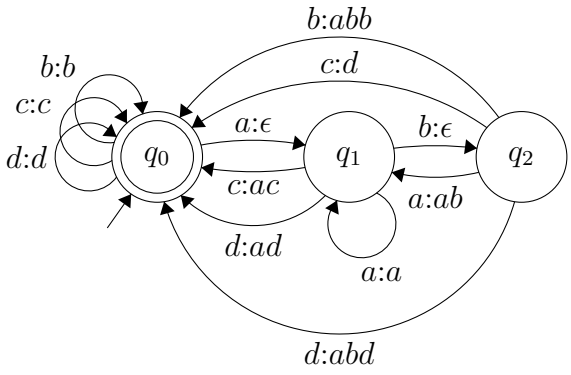
Because colours form a monoid, we can use neutral element as a default value for all unspecified edges like $a(?)$. Moreover, it is also possible to attach meta-information to final states. We could imagine that an accepting state is nothing more than a state with a special outgoing transition that goes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



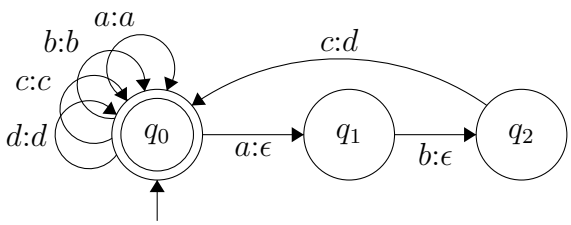
In Solomonoff, the meta-information of interest are transition outputs. We could imagine that strings composed of R and B symbols are the output of transducer. At this point the reader should appreciate how beautifully and naturally subsequential transducers (automata with output produced at accepting 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 embeded 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 operations 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. 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



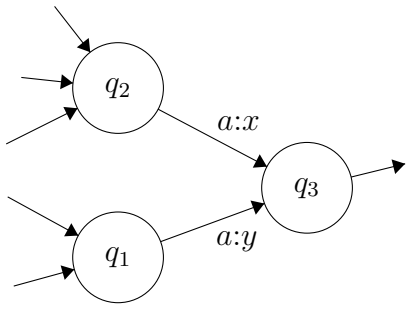
A much simpler alternative would be the nondeterministic transducer



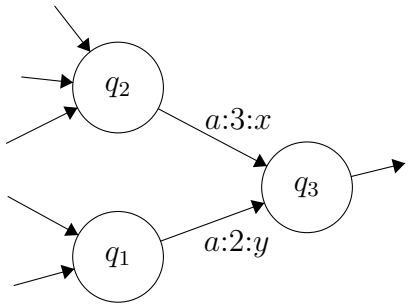
The problem with such solution is that for input strings like $aabcb$ both outputs $aabcb$ and adb are generated. If only there was a way to assign priority

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.



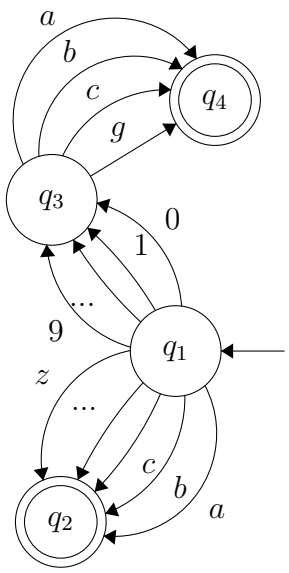
If the next input symbol is a, then the transducer will reach state q_3 and generate two outputs - one ending in x and the other in y. If the automaton later accepts it will produce at least two ambiguous outputs. Lexicographic weights allow us to assign priority to different transitions. In the following 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.



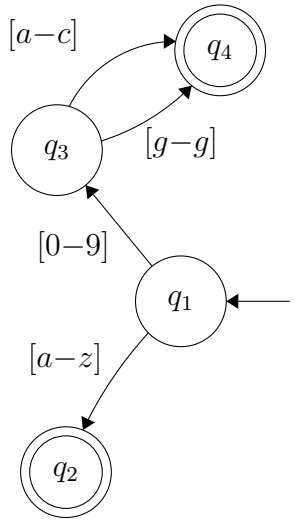
Lexicographic weights allow to make the transducers even more compact. We proved that there exist weighted automata exponentially smaller than even the smallest unweighted nondeterministic ones. In the presence of lexicographic weights, context-dependent rewrites become expressible directly in Glushkov's 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

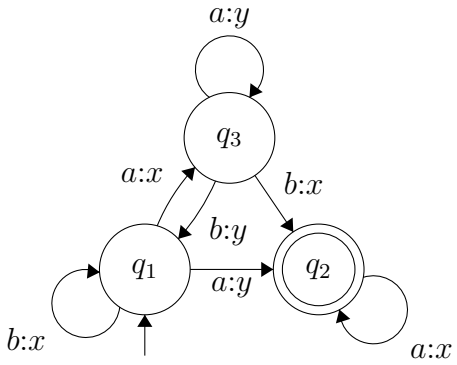
symbols, foreign words or even emojis and other UNICODE entities. The regular expressions should handle such cases gracefully, especially when using wildcards such as `.*` or `\p{Lu}`. Representation of large character classes is a challenging task for finite state automata. In order to use the dot wildcard 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 computers 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 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 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 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 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 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 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 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 strings (that is, all substrings of length 2). The early prototype versions of Solomonoff would represent sets as bit-sets, where each bit specifies whether 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 smallest memory footprint but it 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 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. 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, compiler can point user to the exact line and column of text where the conflict arises, whereas advance-and-delay might miss the origin of problem and only fail further down the line.

References