Glushkov's Construction for Functional Mealy Machines

Aleksander Mendoza-Drosik

***Abstract*—Mealy and Moore machines have been heavily researched from the perspective of sequential circuits. Transducers (their probabilistic) counterpart found numerous applications in natural language processing and grammatical inference. This article focuses on more theoretical aspects, trying to compare several variations of this computational model. Many achievements of automata theory, have been readjusted to Mealy machines. Several new properties were found and proposed in this paper. Finally the most important result discovered here, is an extension of Glushkov's algorithm that combines all the previous results together. The main driving force for this research was an attempt to build strong theoretical foundations for regular expression compiler that could compete with Thrax developed by OpenFst project.**

***Index Terms*—Mealy-Moore machines, Glushkov follow automata, regular expressions, time complexity, space complexity**

I. Introduction

THE goal of this paper is to describe an extension of Glushkov's [1] algorithm for Mealy machines - in particular non-deterministic functional Mealy machines, with possible extensions. Additionally, it also describes how such construction gives ground for efficient evaluation algorithm of described automata.

The definition of deterministic Mealy [2] automata (**DMA** for short) is similar to that of finite state automata (**FSA**), but augmented with output. More precisely, it's a tuple $(Q, q_n, \Sigma, \Gamma, \delta, F)$ with finite set of states Q, initial state q_n, input alphabet Σ, output alphabet Γ, transition function $\delta : Q \times \Sigma \to Q \times \Gamma$ and set of accepting states $F \subset Q$. Similarly to non-deterministic finite state automata (**NFA**) one can also define non-deterministic Mealy machines (**NMA**). The only difference is transition function $\delta : Q \times \Sigma \to 2^{Q\times\Gamma}$. One can go further and add ϵ-transitions using $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^{Q\times(\Gamma\cup\{\epsilon\})}$, obtaining non-deterministic Mealy machines with epsilons (ϵ**NMA**). Unlike in the case of FSA and NFA, the DMA and NMA are not equivalent in power and neither are NMA and ϵNMA. Functional Mealy machines (**FMA**) are non-deterministic Mealy machines that guarantee at most a single accepting output for every input string. Moore machines [3] (**DMM**) are defined similarly to Mealy machines as $(Q, q_n, \Sigma, \Gamma, \delta, G, F)$, with the only exception being transition function $\delta : Q \times \Sigma \to Q$ and separate output function $G : Q \to \Gamma$ that depends solely on state (instead of both state and input). DMA can be extended with variable-length output as $\delta : Q \times \Sigma \to Q \times \Gamma^*$, which gives them greater expressive power (called **vDMA** for short). Analogically **vNMA** and **vFMA** have $\delta : Q \times \Sigma \to 2^{Q\times\Gamma^*}$. Finally, the last model introduced here are non-deterministic variable-length Mealy-Moore machines (**vMMM**), which are a combination of both models and are defined as $(Q, q_n, \Sigma, \Gamma, \delta, F)$ with $\delta : Q \times \Sigma \to 2^{Q\times\Gamma^*}$ and **partial** accepting/output function $F : Q \rightharpoonup \Gamma^*$ (if F returns a string then we append it to output and accept, otherwise F returns $\emptyset$ and automaton rejects).

A. Semantics

Every automaton can be treated as a directed (multi)graph (Q, δ). A path [4] is defined to be a finite sequence of edges $(q_{k_1}, x_1, q_{k_2}), (q_{k_2}, x_2, q_{k_3}), ...(q_{k_m}, x_m, q_{k_{m+1}})$ where $q_{k_i}, q_{k_{i+1}} \in Q$, $x_i \in \Sigma$ and $q_{k_{i+1}} \in \delta(q_{k_i}, x_i)$ for every index i. (The output alphabet Γ has been implicitly omitted in δ, as it doesn't play any role). Signature of a path is the string produced by concatenating $x_1x_2...x_m \in \Sigma^*$. Cosignature of a path is obtained by concatenating respective outputs of δ as in $\delta(q_{k_1}, x_1)...\delta(q_{k_m}, x_m) \in \Gamma^*$ (this doesn't apply to Moore machines). A path is accepting if it starts in $q_{k_1} = q_n$ and ends in $q_{k_{m+1}} \in F$. An automaton accepts a string $x \in \Sigma^*$ if it is a signature of some accepting path. For ϵ-free automata $m = |x|$. For deterministic automata path is uniquely determined for each signature. For ϵNMA $x_i \in \Sigma \cup \{\epsilon\}$. An automaton generates output $y \in \Gamma^*$ if it's cosignature of some accepting path. A special exception are vMMM, for which cosignature is defined as $\delta(q_{k_0}, x_0)...\delta(q_{k_m}, x_m)F(q_{k_{m+1}})$. Notice that in Kleene algebra the empty set $\emptyset$ behaves like multiplicative zero, and since F is partial it may "return" $\emptyset$, resulting in $\emptyset$ being the signature of a path. vMMM accepts a path iff its cosignature is not $\emptyset$. vMMM accepts a string iff it's a signature of some accepting path. Paths are allowed to be empty and in such cases $q_{k_{m+1}} = q_{k_1}$.

Combination is defined to be a subset of Q. Superposition is a subset of $Q \times \Gamma^*$. We extend definition of δ to image of combination $K \subset Q$ as

$$\delta(K, x) = \{q' \in Q : \underset{q\in K}{\exists} q' \in \delta(q, x)\}$$

resulting in a new combination. Analogically given $S \subset Q \times \Gamma^*$ we define

$$\delta(S, x) = \{(q', yy') \in Q \times \Gamma^* : \underset{(q,y)\in S}{\exists} (q', y') \in \delta(q, x)\}$$

Lastly $\hat{\delta} : Q \times \Sigma^* \to Q$ is a transitive extension of δ (for deterministic machines) defined recursively as:

$$\hat{\delta}(q, \epsilon) = q$$
$$\hat{\delta}(q, cx) = \hat{\delta}(\delta(q, c), x) \text{ where } c \in \Sigma$$

Analogically, $\hat{\delta}$ can be made to work with ϵ-free non-deterministic machines by using combinations $2^Q \times \Sigma^* \to 2^Q$ and superpositions $2^{Q\times\Gamma^*} \times \Sigma^* \to 2^{Q\times\Gamma^*}$ instead of $q \in Q$.

While FSA are often associated with languages Σ^*, the same might be noted about Mealy machines and relations on languages $\Sigma^* \times \Gamma^*$. Given some Mealy automaton M, we shall use notation $M(x, y)$ to state that M accepts $x \in \Sigma^*$,

while producing $y \in \Gamma^*$. Notice that $M(x, y)$ iff there is $q \in F$ such that $(q, y) \in \hat{\delta}(S, x)$ where S is the initial superposition $S = \{(q_n, \epsilon)\}$. In cases when M is functional or deterministic we may write $M(x) = y$. We define π_0 and π_1 to be input and output language projections such that $x \in \pi_0(M) \iff \exists_y M(x, y)$ and $y \in \pi_1(M) \iff \exists_x M(x, y)$

II. Expressive power

We propose a rule of matching prefixes to be the theorem, which states that

Theorem 1. *Let M be some deterministic Mealy machine (DMA or vDMA). Then for any strings $x, x' \in \Sigma^*$ such that $xx' \in M$ and $x \in M$, there exists $y' \in \Gamma^*$ for which $M(xx') = M(x)y'$.*

Proof: It follows directly from uniqueness of path that corresponds to signature xx'. ∎

More intuitively, this theorem says that if $M(x) = y$, then for any string x', the output of $M(xx')$ must start with y as prefix.

Another essential tool is the pumping lemma for Mealy machines.

Theorem 2. *Let M be some Mealy machine. There exists number $p \geq 1$ such that if $M(x, y)$ and $|x| \geq p$ then one can decompose $x = x_0x_1x_2$ and $y = y_0y_1y_2$ for which the following holds:*

- $|x_1| \geq 1$
- $|x_0x_1| \leq p$
- $\forall_{i \in \mathbb{N}} M(x_0x_1^ix_2, y_0y_1^iy_2)$

Proof: Assume p to be the number of states of M. Every path with signature x and cosignature y must be at least as long as the string x. By pigeonhole principle of one of the states q_k on that path must appear twice, which implies that the path contains a cycle. We can repeat the cycle as many times (including 0) as we want. This proves the first and third condition. Second one follows from the fact that if path has length p then there are $p + 1$ states visited in total, which is enough for the pigeonhole principle to work. ∎

Myhill-Nerode theorem can be extended to Mealy machines as follows.

Theorem 3. *Let $M \subset \Sigma^* \to \Gamma^*$ be some function on languages and $x_0, x_1 \in \Sigma^*$. Additionally assume that M follows the rule of matching prefixes. A pair $(x', y') \in \Sigma^*$ is defined to be a distinguishing extension of x_0 and x_1 iff exactly one of $(x_0x', M(x_0)y')$ or $(x_1x', M(x_1)y')$ belongs to M. We define an equivalence relation $=_M$ on $\Sigma^* \times \Gamma^*$ such that $p_0 =_M p_1$ iff there is no distinguishing extension for p_0 and p_1. Myhill-Nerode theorem states that M is recognizable by some deterministic Mealy machine iff there are only finitely many equivalence classes induced by $=_M$.*

Proof: First assume there are finitely many equivalence classes. Then build a Mealy machine equivalent to M by treating every class as a state. Put a transition from class q to q' over $c \in \Sigma$ whenever appending c to some member of q yields a member of q'. The difference in outputs (thanks to rule of matching prefixes) shall be used as transition output. Conversely, if there is a Mealy machine for M, then there could be found a homomorphism from states of machine to equivalence classes. ∎

Myhill-Nerode cannot be extended to non-deterministic Mealy machines, because if it could, that would imply existence of unique (up to isomorphism) minimal automaton, which is impossible, because there is no unique minimal NFA [5] and it can be observed that every NFA can be treated like a vNMA/ϵNMA with ϵ on all outputs.

Theorem 4. *DMA have strictly less power than vDMA.*

Proof: DMA produce exactly one character Γ per transition, yielding output strings of the exact same length as input strings., while vDMA can produce output of arbitrary size ∎

Theorem 5. *vDMA have strictly less power than vFMA.*

Proof: Rule of matching prefixes applies to vDMA but not to FMA. ∎

Theorem 6. *FMA have strictly less power than NMA.*

Proof: In NMA there might be multiple accepting paths corresponding to the same signature and each path having distinct cosignature. In FMA by definition there is only one accepting cosignature. ∎

Theorem 7. *NMA have strictly less power than vNMA.*

Proof: Same as in case of DMA and vDMA. ∎

Theorem 8. *vNMA have strictly less power than ϵNMA.*

Proof: Let M be vNMA. There is no way of expressing mapping $M(\epsilon, y)$ such that $y \neq \epsilon$, while in ϵNMA ϵ-transitions coming out q_n could allow for it. Conversely every vNMA can be simulated with ϵNMA by creating additional states with ϵ transitions. For every edge of vNMA which produced output string of length m, ϵNMA would require m ϵ-transitions producing the said string letter by letter. ∎

Theorem 9. *ϵNMA without ϵ-cycles (cycles which have $x_1 = x_2 = ... = x_m = \epsilon$) have strictly less power than ϵNMA with ϵ-cycles.*

Proof: ϵ-cycles would allow for infinitely many cosignatures to be associated with a given signature. Without ϵ-cycles there can be only a finite number of paths corresponding to each signature and there is only one cosignature per path. ∎

It's worth noting that the pumping lemma for FSA could be applied separately to both $\pi_0(M)$ and $\pi_1(M)$. ϵ-cycles are capable of producing a relation on languages, which allows for $\forall_i M(x, y_0y_1^iy_2)$ while violating the first condition of $|x_1| \geq 1$. However, the language $\forall_i \pi_1(M)(y_0y_1^iy_2)$ can be explained with pumping lemma for FSA.

Theorem 10. *Let M be ϵNMA without ϵ-cycles and without ϵ-mapping (there is no $M(\epsilon, y)$ such that y is non-empty), then there is an equivalent ϵ-free vNMA.*

Proof: The procedure for ϵ elimination is analogical to that for ϵNFA. ∎

Theorem 11. *ϵNMA without ϵ-cycles have equal power to vMMM.*

Proof: Most of the ϵ-transitions can be eliminated, with the only exception being those that start in q_n. This case, however, can be dealt with by setting the accepting

function $F(q_n)$ of vMMM. Conversely every vMMM can be converted to ϵNMA. Every accepting state in $F(q_i) \neq \emptyset$ can be emulated by 1. making it a non-accepting state 2. creating a new state, which is marked as accepting 3. adding ϵ-transition from previously accepting state to the new accepting state 4. setting appropriate output for that transition. ■

III. Complexity and optimisations

Theorem 12. *Deciding whether NMA is functional is coNP-hard.*

Proof: Given formula ϕ in 3-conjunctive normal form over variables $x_1, x_2, ...x_n$ we can construct NMA M over alphabet $\Sigma = \Gamma = \{0, 1\}$ such that $\exists_{x \in \Sigma^*} |M(x)| > 1$ iff ϕ is not tautology [6]. In fig.1 every state q_{kl} stands for "after reading input $x_1x_2x_3...x_k \in \{0,1\}^k$, partially assigned clause $(X_{h_k} \vee X_{i_k} \vee X_{j_k})$ still has potential to be true". We shall put edge with label $1 : \epsilon$ from state q_{kl} to $q_{(k+1)l}$ iff $\neg x_{k+1}$ is in the clause. Analogically we put edge $0 : \epsilon$ iff x_{k+1} is in the clause. The final vertices $q_{n1}, ..q_{nm}$ tell us which clauses are true for given assignment. Next to them we also designate special vertices $\neg q_{n1}, ..., \neg q_{nm}$, that collect all the failed false clauses. In the end when automaton reads all n variables, there is an epsilon edge coming from every state $q_{n1}, ..q_{nm}$ that prints $\epsilon : 1$ (meaning clause is true). Similarly from vertices $\neg q_{n1}, ..., \neg q_{nm}$ automaton prints $\epsilon : 0$ if clause is false. If ϕ is a tautology, then for all assignments we will always end up in one of the $q_{n1}, ..q_{nm}$ and never in $\neg q_{n1}, ..., \neg q_{nm}$. Therefore the output associated with input sequence $x_1x_2x_3...x_n$ will be unambiguously 1. If at least one clause fails, then ϕ is not tautology and M returns both 0 and 1. There is also a special case, when ϕ is contradiction (false for all assignments). Thanks to states $q_0, ...q_n$ we should also get ambiguous output. q_n is a special state, that can be reached only in one case - when all clauses are simultaneously false. We put edge $1 : \epsilon$ transitioning from q_k to q_{k+1} iff x_k is not present in any clause. Similarly $0 : \epsilon$ iff $\neg x_k$ is not present in any clause. Such construction guarantees us that ϕ is tautology iff M is functional. ■

Theorem 13. *For any vFMA M, there is an equivalent vFMA such that for any input $x \in \Sigma^*$ the superposition $S = \hat{\delta}(\{(q_n, \epsilon)\}, x)$ is a partial function $S \subset Q \to \Gamma^*$.*

Proof: Suppose to the contrary that there is x and q_i such that $|S(q_i)| > 1$. Then there are two possibilities: either there is a path that starts in q_i and ends in F or there is not. If the first case is true, then M is not FMA, because we might follow that path and accept with multiple outputs. If the second case applies, then the state q_i is redundant and we are free to delete it. ■

The exact same argument applies to ϵFMA. The theorems 12 and 13 imply that there might be no way to efficiently find FMA, but instead we can use the following heuristic algorithm to find cases posing a risk of being non-deterministic. The idea is to explore the network of reachable combinations (and ignore superpositions). If the algorithm at any point reaches a combination K, such that there exists $c \in \Sigma$ and two states $q_0, q_1 \in K$ such that $\delta(\{q_0\}, c) \cap \delta(\{q_1\}, c) \neq \emptyset$, then we say that q_0 and q_1 are in conflict. If there is any pair of conflicting states, then the automaton might not be functional. It's possible to make this algorithm very efficient by introducing ranged transitions.

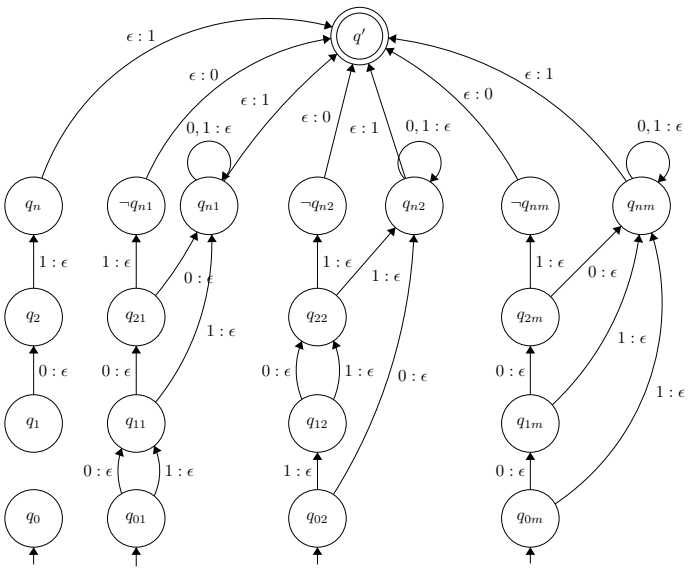

Fig. 1. Construction that uses ϵNMA/vNMA to represent CNF formula. Each column evaluates one clause and ends in state q_{nj} if clause is true or $\neg q_{nj}$ when it's false. State q' will accept producing output 0 if any clause was false and 1 of any clause was true. Leftmost column is a "bias" that prints 1 if all clauses are false. This graph uses ϵ outputs, but by treating ϵ as an ordinary letter of alphabet, it could be adapted to NMA easily.

Let $\leq$ be some total order on Σ. A range $R \subset \Sigma$ is defined to be some closed interval in Σ. A pair (r_0, r_1) is a presentation of range $\{c \in \Sigma : r_0 \leq c \leq r_1\}$. A total order on ranges is defined as $(r_0, r_1) \leq (r_2, r_3)$ iff $r_0 < r_2$ or $r_0 = r_2 \wedge r_1 \leq r_3$. A ranged transition is tuple (q, r_0, r_1, y) where $q \in Q$ is the destination state of transition, (r_0, r_1) is presentation of some range and $y \in \Gamma^*$ is the generated output of transition. A ranged transition array is an increasing sequence of ranged transitions where the order is induced by ranges in those transitions.

Ranged transition array is a data structure that can be used for encoding transitions of automata (as an alternative to transition matrix or directed graphs). Given $x \in \Sigma$ one can use binary search to find the last ranged transition containing x and then iterate in the decreasing order until the first such transition is found. This way one can optimally query all transitions containing x. Given some combination K, one may efficiently find all reachable combinations $\delta(K, \Sigma)$ by performing $\delta(K, x)$ only for a tiny subset of Σ. That's because $\delta(K, x)$ yields the same results for most of the values of x. The only x worth checking are $min(\Sigma)$, r_0 of every ranged transition of every state in K (because this is where "change" begins) and every $r_1 + 1$ (this is where "change" ends). As an analogy, one might see that many rows in transition matrix are the same and "in practice" (due to ranges) there is a strong regularity in the way those rows repeat.

Another opportunity for optimisation is augmenting automata with weights. Let W be some set with total ordering $\leq$. We define weighted ranged transition to be a tuple (q, w, r_0, r_1, y) where $w \in W$. Weighted vFMA (vWMA) is defined with $\delta : Q \times \Sigma \to 2^{Q \times \Gamma^* \to W}$. Let S be a

superposition, then $\delta(S,x)$ is

$$\delta(S,x) = \{(q', yy') \in Q \times \Gamma^* : \exists_{(q,y)\in S} (q', y', w_{max}) \in \delta(q,x)\}$$

What exactly max means can be explained as follows. Suppose there exists a pair of conflicting states (q_1, y_1) and (q_2, y_2) in S. Then there must exist state $(q_3, y_3) \in \delta(S,x)$ with transitions $(q_3, y_1, w_1) \in \delta(q_1, x)$ and $(q_3, y_2, w_2) \in \delta(q_2, x)$. For sake of simplicity assume that q_1 and q_2 are the only states in S connected to q_3. If $w_1 < w_2$ then $y_3 = y_2$. If $w_1 > w_2$ then $y_3 = y_1$. If $w_1 = w_2$ then q_1 and q_2 are said to be weight-conflicting (notice that states cannot be weight-conflicting if they are not conflicting in the first place).

It's very important not to confuse vWMA with probabilistic automata [7]. Here weights are not probabilities and are not accumulated in any way as the computation progresses.

Theorem 14. *vWMA are equivalent in power to vFMA.*

Proof: Conversion can be carried out using (extended) powerset construction. Suppose Q are the states of vWMA, then $Q' = 2^Q \times Q$ are states of equivalent vFMA. For any state $(K, q) \in Q'$, the K represents particular combination of Q and q is some representative in K (we can discard all those states of Q' where $q \notin K$). We need the representative because theorem 13 states that every superposition S' in vFMA is a function. Therefore we set $S'((K,q)) = S(q)$, where S is the corresponding superposition in vWMA (and K is the combination underlying S). We put transition from (K_0, q_0) to $(K_1, q_1) \in Q'$ over $x \in \Sigma$ with output $y \in \Gamma^*$ iff $\delta(K_0, x) = K_1$ and $(q_1, y, w) \in \delta(q_0, x)$. The only exception is when there are multiple states in Q' that transition to (K_1, q_1) over x. In such cases we pick only the one with highest weight w. Finally a state of $(K,q) \in Q'$ is accepting iff $q \in K$ is accepting. If vWMA has no weight-conflicting states, then the resulting vFMA doesn't have any conflicting states. ∎

Theorem 15. *There exists a family of vWMA such that their corresponding minimal vFMA are exponentially larger.*

Proof: We define family of vWMA in such a way that for every $i \geq 3$ there is vWMA on i states. The number of states of minimal equivalent vFMA is $O(2^i)$. Figure 2 presents awy to build such automata. state q_0 is initial. Using strings from $0, 1^*$ one can obtain any combination of states q_1 to q_n. Let's associate each combination with a string $z \in \{0,1\}^n$ (for instance $z = 011$ would be a combination $\{q_2, q_3\}$). That gives 2^n possible strings. State q_{n+1} is accepting and all the states $q_1 ... q_n$ are connected to it. Essentially the relation described by this automaton is a subset of $0, 1^+ 2 \times \{y_1, ..., y_n\}$. Without loss of generality suppose that weights $w_1 ... w_n$ are in ascending order. Then the automaton maps every z determined by $x \in 0, 1^+$ to some $(x2, y_i)$ such that i indicates the least significant bit in z. In the spirit similar to Myhill-Nerode theorem, it can be seen that no two x strings that map two different z are equivalent. Therefore when trying to built vFMA, there must be at least 2^n states (including one state for $z = 0^n$). ∎

One can easily notice that every vFMA can be treated like a vWMA with all weights equal, therefore the opposite of theorem 15 doesn't hold (there is no family of vFMA such that vWMA would be larger).

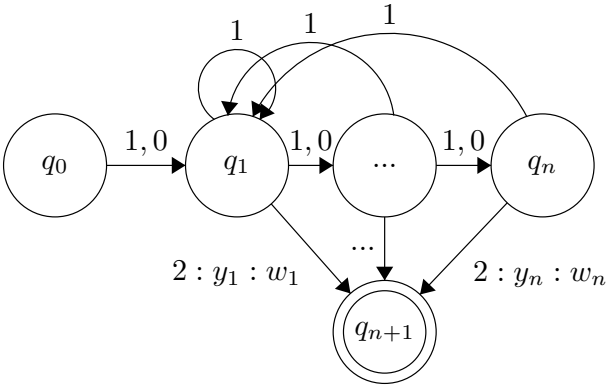

Fig. 2.

IV. Regular expressions

Here we describe a flavour of regular expressions specifically extended for better optimised opportunities. Every alphabet is treated like a type.

We define $\mathcal{A}^\Sigma \subset \Sigma^*$ to be some set of atomic characters and $\mathcal{A}^{\Sigma:\Gamma} \subset \Sigma^* \times \Gamma^*$ to be set of atomic relations. In case of FSA $\mathcal{A}^\Sigma = \Sigma$. In case of DMA and NMA $\mathcal{A}^{\Sigma:\Gamma} = \Sigma \times \Gamma$. For ϵNMA $\mathcal{A}^{\Sigma:\Gamma} = \Sigma \cup \{\epsilon\} \times \Gamma \cup \{\epsilon\}$. A great optimisation strategy for vDMA and vNMA is to use $\mathcal{A}^{\Sigma:\Gamma} = \Sigma \times \Gamma^*$. For ranged automata one might use $\mathcal{A}^{\Sigma:\Gamma} = R \times \Gamma$ where R is set of range presentations (R is subset of $\Sigma \times \Sigma$ such that the first element is no greater than the second).

We call RE^Σ and $RE^{\Sigma:\Gamma}$ the set of formulas of regular expression with underlying atomic sets $\mathcal{A}^\Sigma$ and $\mathcal{A}^{\Sigma:\Gamma}$ respectively. If $\phi, \psi \in RE^{\Sigma:\Gamma}$ then $\phi + \psi \in RE^{\Sigma:\Gamma}$ (union), $\phi\psi \in RE^{\Sigma:\Gamma}$ (concatenation), $\phi^* \in RE^{\Sigma:\Gamma}$ (Kleene closure). Analogically for RE^Σ. Additionally if $\phi \in RE^\Sigma$ and $\psi \in RE^\Gamma$ then $\phi : \psi \in \mathcal{A}^{\Sigma:\Gamma}$. Let $V^{\Sigma:\Gamma} : RE^{\Sigma:\Gamma} \to \Sigma^* \times \Gamma^*$ be a valuation function defined as:
$V^{\Sigma:\Gamma}(\phi + \psi) = V^{\Sigma:\Gamma}(\phi) \cup V^{\Sigma:\Gamma}(\psi)$
$V^{\Sigma:\Gamma}(\phi\psi) = V^{\Sigma:\Gamma}(\phi) V^{\Sigma:\Gamma}(\psi)$
$V^{\Sigma:\Gamma}(\phi^*) = (\epsilon, \epsilon) + V^{\Sigma:\Gamma}(\phi) + V^{\Sigma:\Gamma}(\phi)^2 + ...$
$V^{\Sigma:\Gamma}(\phi : \psi) = V^\Sigma(\phi) V^\Gamma(\psi)$
$V^{\Sigma:\Gamma}(a) = a$ where $a \in \mathcal{A}^{\Sigma:\Gamma}$
Analogically for V^Σ, except without $V^\Sigma(\phi : \psi)$. It should be noted that every RE^Σ is isomorphic to $RE^{\{\epsilon\}:\Sigma}$ and $RE^{\Sigma:\{\epsilon\}}$. Very often one can use atomic characters to build atomic relations as follows $x : y = (x, y) \in \mathcal{A}^{\Sigma:\Gamma}$ where $x \in \mathcal{A}^\Sigma$ and $y \in \mathcal{A}^\Gamma$. For this reason, the distinction between atomic characters and relations can sometimes be blurred. Some notable properties are:
$x : y_0 + x : y_1 = x : (y_0 + y_1)$
$x : \epsilon + x : y + x : y^2 ... = x : y^*$
$(x : y_0)(\epsilon : y_1) = x : (y_0 y_1)$
$x_0 : (y' y_0 y'') + x_1 : (y' y_1 y'') = (\epsilon : y')(x_0 : y_0 + x_1 : y1)(\epsilon : y'')$

One can easily see that any $\phi \in RE^{\Sigma:\Gamma}$, which contains $x : (y_0 + y_1)$ or $x : y^*$ cannot produce functional machine, as there are multiple outputs associated with x. Therefore we define $RE^{\Sigma \to \Gamma}$ as a subset of $RE^{\Sigma:\Gamma}$, which doesn't allow such structures (although $x : y_0 + x : y_1$ is still allowed, so there is no guarantee that $V^{\Sigma\to\Gamma} : RE^{\Sigma\to\Gamma} \to \Sigma^* \times \Gamma^*$ produces $\Sigma^* \to \Gamma^*$). $RE^{\Sigma\to\Gamma}$ allows for substantial simplification, especially when combined with $\mathcal{A}^{\Sigma:\Gamma} = \Sigma \times \Gamma^*$. Every $\phi : \psi$ is guaranteed to be of the form $\phi : y$ where

$y \in \Gamma^*$. This will allow for creating efficient Glushkov's algorithm for Mealy machines.

It's also possible to extend regular expressions with weights to express vWMA. Let $RE_W^{\Sigma\to\Gamma}$ be a superset of $RE^{\Sigma\to\Gamma}$ and W be the set of weights. Every $\phi \in RE^{\Sigma\to\Gamma}$ is also in $RE_W^{\Sigma\to\Gamma}$. Additionally if $\phi \in RE_W^{\Sigma\to\Gamma}$ and $w_0, w_1 \in W$ then $w_0\phi$, ϕw_1 and $w_0\phi w_1$ are in $RE_W^{\Sigma\to\Gamma}$. This allows for inserting weight at any place. As an example where it matters consider $\phi w_0 + \psi w_1$. If it's possible to match both w_0 and w_1 simultaneously, then the alternative with greater weight is taken. This will be shown in more detail in Glushkov's algorithm.

Lastly, we introduce product alphabets. Given some sets Σ_1, Σ_2 one can build a new set $\Sigma_1 \times \Sigma_2$. Every string in $(\Sigma_1 \times \Sigma_2)^*$ can be viewed as a string in $\Sigma_1 \cup \Sigma_2$ of the form $(\Sigma_1\Sigma_2)^*$. Automata working on such sets as alphabets exhibit special property - and the graph of the automaton is bipartite. The set of states Q can be partitioned into subsets $Q_1 \cap Q_2 = \emptyset$, such that all transitions starting in Q_1, end in Q_2 and vice-versa. This can be generalised for multiple sets $\Sigma_1 \times ... \times \Sigma_i$. Even further generalisation would allow for using Kleene closure as in $\Sigma_1^* \times \Sigma_2$, $\Sigma_1^* \times \Sigma_2^*$ or $\Sigma_1 \times \Sigma_2^*$, except that such infinite sets are no longer viable alphabets. Every element of these products can still be encoded as a single string over $\Sigma_1 \cup \Sigma_2$. If Σ_1 and Σ_2 are disjoint, the states of automaton can be partitioned into Q_1 and Q_2, such that all transitions incoming to Q_1 use alphabet Σ_1, and those incoming to Q_2 use Σ_2. If Σ_1 and Σ_2 are not disjoint, then a special separator character $\#$ is required, that would mark ends of strings in Σ_1^* and Σ_2^*. For example $x_0x_1x_2x_1\#x_1x_2x_2\# \in \{x_0, x_1, x_2, \#\}^*$ encodes $(x_0, x_1x_2x_1)(x_1, x_2x_2) \in \{x_0, x_1\} \times \{x_1, x_2\}^*$. Given any product $\Pi = X_1 \times X_2 ... \times X_i$ where X_k is either Σ_k or Σ_k^*, it's possible to build simple FSA that checks whether a given string over $\Sigma_1 \cup ... \cup \Sigma_i \cup \{\#\}$ is a valid encoding of string over product alphabet Π ($\#$ doesn't belong to any Σ_k, and we don't assume that any of alphabets are pairwise disjoint). Similarly, if we annotate every atomic letter $a \in \mathcal{A}^{\Sigma_k}$ of regular expression with a^{Σ_k}, we can use it to efficiently check if any given $\phi \in RE^{\Sigma_1\cup...\cup\Sigma_i\cup\{\#\}}$ is a valid expression $\phi \in RE^{\Pi}$.

V. Extended Glushkov's construction

The core result of this paper is Glushkov's algorithm capable of producing very compact, ϵ-free, weighted, ranged, functional, variable-length Mealy-Moore machines and additionally allows for using product alphabets (and validates if any typed regular expression recognizes/produces valid encodings of product alphabets).

Let $\phi \in RE_W^{\Pi\to\Pi'}$ such that $\Pi = X_0 \times ... \times X_{k-1}$, $\Pi' = X_k \times ... \times X_{l-1}$ where X_i is either Σ_i^* or Σ_i. W is the set of weights with ordering $\leq$ and for the purposes of this algorithm there must additionally be operation $+$ that forms a group $(W, +)$. Let $\Sigma = \Sigma_0 \cup ... \cup \Sigma_{k-1} \cup \{\#\}$ and $\Gamma = \Sigma_k \cup ... \cup \Sigma_{l-1} \cup \{\#\}$.

First step of the algorithm is to create a new alphabet Ω in which every atomic character (including duplicates) from Π in ϕ is treated as a new individual character. As a result we should obtain new rewritten formula $\psi \in RE_W^{\Omega\to\Pi'}$ along with mapping $\alpha : \Omega \to \mathcal{A}^{\Sigma}$ and $\beta : \Omega \to \{0, ..., k-1\}$. For example assume $\Pi = \Sigma_0 \times \Sigma_1$ and $\Pi' = \Sigma_2 \times \Sigma_3^*$, with $\Sigma_0 = \Sigma_1 = \Sigma_2 = \Sigma_3 = \{x_0, x_1, x_2\}$ then

$$\phi = (\epsilon : x_0^{\Sigma_2})x_0^{\Sigma_0}x_0^{\Sigma_1} : x_1^{\Sigma_3}\#^{\Sigma_3}x_2^{\Sigma_2}x_1^{\Sigma_3}\#^{\Sigma_3}w_0 + (x_1^{\Sigma_0}x_2^{\Sigma_1})^*w_1$$

will be rewritten as

$$\psi = (\epsilon : x_0^{\Sigma_2})\omega_1\omega_2 : x_1^{\Sigma_3}\#^{\Sigma_3}x_2^{\Sigma_2}x_1^{\Sigma_3}\#^{\Sigma_3}w_0 + (\omega_3^{\Sigma_0}\omega_4^{\Sigma_1})^*w_1$$

with $\alpha = \{(\omega_1, x_0), (\omega_2, x_0), (\omega_3, x_1), (\omega_4, x_2)\}$ and $\beta = \{(\omega_1, 0), (\omega_2, 1), (\omega_3, 0), (\omega_4, 1)\}$.

From now on we shall treat $\psi \in RE_W^{\Omega\to\Pi'}$ merely as $\psi \in RE_W^{\Omega\to\Gamma}$. Next step is to define $\Lambda : RE_W^{\Omega\to\Gamma} \rightharpoonup (\Gamma^* \times W)$ which returns what output is produced for empty word ϵ (if any) and weight associated with it. Because Γ^* is a monoid and W is a group, we can treat $\Gamma^* \times W$ 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. This facilitates recursive definition Λ:
$\Lambda(\psi_0 + \psi_1) = \Lambda(\psi_0) \cup \Lambda(\psi_1)$ if at least one of the sides is $\emptyset$, otherwise error,
$\Lambda(\psi_0\psi_1) = \Lambda(\psi_0) \cdot \Lambda(\psi_1)$,
$\Lambda(\psi_0 : y) = \Lambda(\psi_0) \cdot (y, 0)$
$\Lambda(\psi_0 w) = \Lambda(\psi_0) \cdot (\epsilon, w)$
$\Lambda(w\psi_0) = \Lambda(\psi_0) \cdot (\epsilon, w)$,
$\Lambda(\psi_0^*) = (\epsilon, 0)$ if $(\epsilon, w) = \Lambda(\psi_0)$ or $\emptyset = \Lambda(\psi_0)$, otherwise error,
$\Lambda(\epsilon) = (\epsilon, 0)$
$\Lambda(\omega) = \epsilon$ where $\omega \in \Omega$

Next step is to define $B : RE_W^{\Omega\to\Gamma} \to (\Omega \rightharpoonup \Gamma^* \times W)$ which for a given formula ψ returns set of Ω characters that can be found as the first in any string of $V^{\Omega\to\Gamma}(\psi)$ and to each such character there must come associated output produced "before" reaching it. For instance, in the example above there are two characters that can be found at the beginning: ω_1 and ω_3. Additionally, there is ϵ which prints output x_0 before reaching ω_1. Therefore $B(\psi)(\omega_1) = x_0$ and $B(\psi)(\omega_3) = \epsilon$. The symbol $\rightharpoonup$ stands for partial function, so $B(\psi)(\omega_2) = B(\psi)(\omega_4) = \emptyset$. For better readability, we admit operation of multiplication $\cdot : (\Omega \rightharpoonup \Gamma^* \times W) \times (\Gamma^* \times W) \to (\Omega \rightharpoonup \Gamma^* \times W)$ that performs monoid multiplication on all $\Gamma^* \times W$ elements returned by $\Omega \rightharpoonup \Gamma^* \times W$. B is defined recursively as:
$B(\psi_0 + \psi_1) = B(\psi_0) \cup B(\psi_1)$
$B(\psi_0\psi_1) = B(\psi_0) \cup \Lambda(\psi_0) \cdot B(\psi_1)$
$B(\psi_0 w) = B(\psi_0)$
$B(w\psi_0) = (\epsilon, w) \cdot B(\psi_0)$
$B(\psi_0^*) = B(\psi_0)$
$B(\psi_0 : y) = B(\psi_0)$
$B(\epsilon) = \emptyset$
$B(\omega) = \{(\omega, (\epsilon, 0))\}$
It's worth noting that $B(\psi_0) \cup B(\psi_1)$ always yields function (instead of 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 $E : RE_W^{\Omega\to\Gamma} \to (\Omega \rightharpoonup \Gamma^* \times W)$, which is very similar to B, except that E collects characters found at the end of strings.
$E(\psi_0 + \psi_1) = B(\psi_0) \cup B(\psi_1)$
$E(\psi_0\psi_1) = B(\psi_0) \cdot \Lambda(\psi_1) \cup B(\psi_1)$
$E(\psi_0 w) = B(\psi_0) \cdot (\epsilon, w)$
$E(w\psi_0) = B(\psi_0)$
$E(\psi_0^*) = B(\psi_0)$
$E(\psi_0 : y) = B(\psi_0) \cdot (y, 0)$

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

Next step is to use B and E to determine all two-character substrings that can be encountered in $V^{\Omega\to\Gamma}(\psi)$. Given two functions $b, e : \Omega \rightharpoonup \Gamma^* \times W$ we define product $b \times e : \Omega \times \Omega \rightharpoonup \Gamma^* \times W$ such that for any $(\omega_0, y_0, w_0) \in b$ and $(\omega_1, y_1, w_1) \in c$ there is $(\omega_0, \omega_1, y_0 y_1, w_0 + w_1) \in b \times e$. Then define $L : RE_W^{\Omega\to\Gamma} \to (\Omega \times \Omega \rightharpoonup \Gamma^* \times W)$ as:
$L(\psi_0 + \psi_1) = L(\psi_0) \cup L(\psi_1)$
$L(\psi_0\psi_1) = L(\psi_0) \cup L(\psi_1) \cup E(\psi_0) \times B(\psi_1)$
$L(\psi_0 w) = L(\psi_0)$
$L(w\psi_0) = L(\psi_0)$
$L(\psi_0^*) = L(\psi_0) \cup E(\psi_0) \times B(\psi_0)$
$L(\psi_0 : y) = L(\psi_0)$
$L(\epsilon) = \emptyset$
$L(\omega) = \emptyset$ One should notice that all the partial function 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, Λ and α to produce ranged transition array with weights. The recommended set of atomic relations is $\mathcal{A} = R \times \Gamma^*$. This algorithm is generic and can easily work with range presentations as atomic characters. For instance, in the example showed at the beginning, one might substitute x_0 for (x_0, x_0) , x_1 for (x_1, x_1) and x_2 for (x_2, x_2). Each range is later translated into a single character of Ω and treated in atomic fashion. Care should be taken, not to use ranges in the output strings, as that would yield non-functional automata.

As shown in the theorem 11, vMMM work as an alternative to ϵNMA, but don't require ϵ transitions (which is exactly what L produces). Later we show that vMMM allows for simple and efficient evaluation algorithm. The way to build functional weighted vMMM $(Q, q_n, \Sigma, \Gamma, \delta, F)$ with $\delta : Q \times \Sigma \to (Q \rightharpoonup \Gamma^* \times W)$ and $F : Q \rightharpoonup \Gamma^* \times W$ is as follows:
$Q = \Omega \cup \{q_n\}$ where $n = |\Omega| + 1$,
$\Sigma_r \subset \Sigma \times \Sigma$ which is set of ranges on Σ,
$\delta_r : Q \times \Sigma_r \to (Q \rightharpoonup \Gamma^* \times W)$,
$(\omega_0, \alpha(\omega_1), \omega_1, y, w) \in \delta_r$ for every $(\omega_0, \omega_1, y, w) \in L(\psi)$
$(q_n, \alpha(\omega), \omega, y, w) \in \delta_r$ for every $(\omega, y, w) \in B(\psi)$
δ becomes range transition array built from δ_r,
$F = E(\psi)$
Because here ranges are considered atomic, type of α becomes $\Omega \to \Sigma_r$ rather than $\Omega \to \Sigma$.

Additionally at this point result of L could be used together with β to validate product language $\Pi = X_0 \times X_1 ... \times X_{k-1}$. For any element $(\omega_0, \omega_1, y, w) \in L(\psi)$ let $i = \beta(\omega_0)$ and $j = \beta(\omega_1)$.
If $X_i = \Sigma_i$ then assert $i + 1 \equiv j \mod k$.
If $X_i = \Sigma_i^*$ then assert $\alpha(\omega_i) = \#$ and $i + 1 \equiv j \mod k$; or $\alpha(\omega_i) \neq \#$ and $i = j$.
In both cases it doesn't matter whether $X_j = \Sigma_j$ or $X_j = \Sigma_j^*$, because either way the next character after ω_0 should be from Σ_j (and it might be $\#^{\Sigma_j}$). Finally, after checking L, the last thing left to check is whether every element of $B(\psi)$ belongs to Σ_0 and every element of $E(\psi)$ belongs to Σ_{k-1}.

Another approach works for checking output $\Pi' = X_k \times ... \times X_{l-1}$. Instead of β, define a new function $\gamma : \Omega \to \{k, ..., l-1\}$, which the algorithm will gradually "discover". Initially it "discovers" that for every ω in $B(\psi)$, the function should return $\gamma(\omega) = k$. Next for every "discovered" ω_0 find all elements $(\omega_0, \omega_1, y, w) \in L(\psi)$. The algorithm should check (similarly as above) if the string y is valid in Π'. Then look up the alphabet of last character in y (characters of y should be annotated just as they were in ϕ). If that last character is $\#^{\Sigma_i}$ (and $X_i = \Sigma_i^*$) or it's not $\#$ but $X_i = \Sigma_i$, then the algorithm "discovers" value of $\gamma(\omega_1) = k + (i - k + 1 \mod l - k)$. If $y = \epsilon$ then $\gamma(\omega_1) = \gamma(\omega_0)$. If y ends with $c^{\Sigma_i} \neq \#^{\Sigma_i}$ and $X_i = \Sigma_i^*$, then the algorithm "discovers" $\gamma(\omega_1) = i$. Push ω_1 onto stack of elements to visit (if it's not on the stack yet). Repeat the process until all elements of Ω are "discovered". If at any point algorithm discovers contradicting values for any ω, an error should be returned. Finally check that all elements of $E(\psi)$ have been "discovered" to be in Σ_k.

One last very important detail that requires mentioning is that $L(\psi) = \emptyset$ when $\psi = w_0\epsilon : yw_1$. This special case must be handled separately as there is no way to build ranged transitions array from $\emptyset$.

A. *Efficient evaluation algorithm*

Thanks to lack of ϵ transitions in vMMM, one can use dynamic programming to efficiently evaluate automaton for any input string $x \in \Sigma^*$. Create 2D array of size $|Q| \times (|x| + 1)$ where i-th column represents superposition after reading first $i - 1$ characters. Each cell should hold information about the transition used (which includes weight and source state of transition). For instance cell $c_{i,j} = (k, w)$ should encode transition with weight w coming from state k to state i, after reading $j - 1^{th}$ character. If state $q_i \in Q$ does not belong to j^{th} superposition, then $c_{i,j} = \emptyset$. The first column is initialized with dummy value $c_{n,1} = (0, 0)$ and $c_{i,1} = \emptyset$ everywhere else. Then algorithm progresses building next superposition from previous one. After filling out the entire array, the last column should be checked for any accepting states. There might be many of them but the one with largest weight should be chosen. (Since produced vMMM is functional, there will be only one largest value). Finally backtracking should be used to find out which path "won". This will determine what outputs need to be concatenated together to obtain path's cosignature. This algorithm is quadratic $O(|Q|, |x|)$.

VI. Further research

Product alphabets could find numerous applications with many possible extensions. One particular idea is to use temporal logics to express more complex constraints. In the setting of natural language processing, product languages of the form $\{Verb, Noun, Adj\} \times \{a, b, ...z\}^*$ could be used to represent human sentences divided into words with linguistic meta-information.

Furthermore, it's worth exploring grammatical inference algorithm with weights like in vWMA, as they would promise more compact automata.

Lastly it's worth pointing out that there is no obvious minimization algorithm for vMMM, due to its non-deterministic nature. Theorem 15 gives a clue that in fact, that minimization might not be necessary. It should be mentioned that the author has successfully found some approaches for reducing number of states on vMMM, but they are too long to describe in one article.

ACKNOWLEDGMENT

The authors would like to thank... TODO

REFERENCES

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

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

[3] E. F. Moore, "Gedanken-experiments on sequential machines," *Princeton University*, 1956.

[4] J.-E. ric Pin, *Mathematical Foundations of Automata Theory*. IRIF, 2019.

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

[6] S. Arora and B. Barak, *Computational Complexity: A Modern Approach*. Princeton University, 2007.

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