For submission to the Journal of Automata, Languages and Combinatorics
Created on September 12, 2025

REINFORCEMENT LEARNING OF FINITE-STATE STRING TRANSDUCTIONS

Johanna Björklund Jingwen Cai Anna Jonsson

Dept. Computing Science, Umeå University
90187 Umeå, Sweden
{johanna,jingwenc,aj}@cs.umu.se

ABSTRACT

Finite state transducers (FSTs) are a valuable tool in data processing systems, where they are used to realise string-to-string transductions. We consider the problem of inferring transductions representable by FSTs through reinforcement learning. In this machine-learning paradigm, a learning algorithm repeatedly interacts with an environment by performing one out of a fixed set of candidate actions. Each action taken yields a reward, the size of which depends on the environment's current state, and causes the environment to change into a new state. The algorithm's objective is to maximise the accumulated reward. In the setting explored here, the environment consists of the next symbol in the input string to be rewritten and a transducer state. An action consists in choosing the symbol to output next, and the transducer state to shift into, thus causing a change in the environment. We propose a learning algorithm that starts out from a singleton set of states, and every time the learning rate stagnates, splits a state into two. For the split, it chooses a state that has been visited often, but despite this provides little information about how to maximise the reward. We evaluate the algorithm through empirical experiments, and the results suggest that it is robust enough to handle situations where the target transduction changes during the learning process.

Keywords: string transducers, reinforcement learning, Q-learning

1. Introduction

We study reinforcement learning of (deterministic and total) string-to-string transductions, represented by finite-state string transducers (FST). This is a computational device for expressing functions on strings, which despite its simplicity has a range of applications. For example, FSTs have been applied to transliteration, inflection generation, cognate project and phoneme-to-grapheme conversion in machine translation [11, 17], and offer an efficient analysis tool in speech processing [24]. Prior

The authors are given in alphabetical order. The project is supported by the Swedish Research Council under Grant Number 2020-03852, by the Wallenberg AI, Autonomous Systems and Software Program (WASP) and WASP-HS.

Johanna Björklund: 0000-0003-0596-627X, Jingwen Cai: 0009-0004-0580-6270, Anna Jonsson: 0000-0002-9873-4170

research has emphasised three areas: (i) grammatical inference within the context of active learning [4, 1], (ii) distillation of finite automata from trained neural networks [28, 8], and (iii) supervised learning of transition functions represented by SVM classifiers [18, 19] or LSTMs [30, 9, 33]. Our work complements these endeavours by exploring the application of reinforcement learning to finite-state transducers [25, 14]. Finite-state transducers (FSTs) [6, 3] are essentially finite state automata [22], where the transitions in addition to carrying input symbols also carry output symbols. This feature enables these automata to generate an output string in response to an input string, thus realising string-to-string transductions.

Reinforcement learning (RL) shares similarities with active learning in that the learning algorithm can influence what training data it will receive [12]. The learning framework is typically modelled as a Markov decision process [2], described by (i) a potentially infinite set of environment states, (ii) a set of actions that the learner may take, (iii) a probability distribution describing the impact of the learner's actions on the environment states, and (iv) a reward function that maps each combination of state and action to a weight. The learning process unfolds through discrete time steps, divided into finite episodes. At each time step, the learner selects an action based on the current state of the environment, resulting in a reward and a change in the environment's state. The learner's objective is to optimise its accumulated reward, meaning it must weigh the value of discovering new information that may increase future rewards against that of levering existing knowledge for more immediate profit.

In our setting, the learner acquires a target transduction $\mathcal{T}$ through a sequence of *episodes*. In each episode, it is given an input string u and asked to produce $\mathcal{T}(u)$ by iterating over the symbols of u – one at a time – and for each symbol generating a suitable output. We consider the transduction $\mathcal{T}$ to have been successfully inferred when the learner has improved to the point where it can make an uninterrupted series of N perfect translations from a given input string to the correct output string (where N is an integer-valued hyperparameter of the learning algorithm). We do not require that the resulting FST is in some sense canonical, though in our experiments, the learner always found an FST of minimal size.

To guide the transduction, the agent has access to a set of transducer states P, which will eventually come to make up the states of the inferred FST. It is common in reinforcement learning to view all input on which the agent acts as part of the environment [29]. For example, in an RL model of a robot that learns to return to its charging station every time its battery runs low, the battery status would be treated as part of the environment, even though it is inside the robot. The advantage of attributing all statefulness to the environment and treating the agent as a purely reactive entity, is that it isolates the problem of choosing the optimal strategy (which is the agent's responsibility) from modelling the rules that govern cause and effect (which is what the environment is used for). In the upcoming pages, we keep with this convention as it makes it easier to apply standard RL algorithms.

The reader should thus be prepared that to our proposed FST learner, the set of states P of the FST that it is constructing is considered to be part of the environment, even though it is an integral part of the inferred device and something that the learner can manipulate directly. We consider both the case where P is fixed from start, and

when it is gradually built up during learning. At each step of the learning process, the environment state consists of the symbol α in u to process next and a transducer state q' in P. We pause here to underline the distinction between the *transducer states* P, and the *environment states*, which are tuples consisting of an input symbol and a transducer state in P. The motivation for having two types of states is that the former is needed to infer an FST, but the latter is needed to represent also the input symbols as part of the environment. The choice of naming is not ideal, but it is respects the convention in RL to refer to each successive configuration of the environment as a state.

For the learner, an action consists in choosing a string v to output and a transducer state q in P to move to. The reward is based on the Levenshtein distance [15] between the output generated by the learner and the correct output according to $\mathcal{T}$. In other words, it is (inversely) proportionate to the minimal number of one-symbol insertions, deletions, and replacements required to turn one string into the other. This reward function is only meaningful because the target transduction is assumed to be deterministic and total, that is, there is exactly one correct output for each input. In Section 5 we outline ideas for how the reward function can be varied to accommodate probabilistic transductions.

Also when it comes to the timing and distribution of the reward there are different design options: We may either give the learner a positive reward at each time step, or give it neutral rewards until the very end of an episode, but then provide it with feedback that reflects the entire output string. As we shall see, the former method often accelerates the learning process, but can cause the learner to make mistakes that are time-consuming to correct, thereby delaying or even preventing convergence.

As previously mentioned, the agent is provided with the input strings to translate. This means that it has limited freedom to collect training data, but it still has some influence as it can choose how to react to each input symbol and observe the effects of its actions. The setting has similarities to the applications of RL to the game of Go: the agent cannot control its opponents moves, but it can decide how to respond, and at the end of the game it receives a strong feedback signal on its performance [26].

We base our learning algorithm on Q-learning [31] which is a standard RL approach. Q-learning leverages the observation that the value $val(s, a)$ of transitioning from an environment state s by taking an action a can be approximated as the direct reward incurred by a, plus the maximum value that can be obtained by then continuing from the resulting environment state s'. The result is a recursive function, where the current estimate of $val(s, a)$ is updated based on estimates of the form $val(s', b)$ where b is some applicable action at environment state s'. This refinement process continues until value function val stabilises, at which point it can be discretised to obtain a transition function for an FST.

A drawback of the Q-learning approach is that it supposes that we know the appropriate number of transducer states in advance. If we give it too few transducer states to work with, then the learner will be unable to deduce the target transduction. Conversely, if we give it too many, then the learning process will take a prohibitively long time to converge. To address this challenge, we employ a novel state-splitting method, which operates as follows: Initially, the learner is allowed to explore the envi-

ronment for a limited period of time. Afterwards, the learning progress is periodically evaluated to ensure that some improvement has occurred since the last evaluation. If no progress is evident, we select a state that the learner has frequently used but which still possesses little predictive value for the eventual reward. This state is then split into two. The rationale behind this approach is that if there is a sufficient number of states, then the learner should eventually make progress. If this is not the case, then additional states are required. Furthermore, if the learner frequently passes through a state q but lacks an effective strategy for maximising the reward from that point, it is likely because there are input strings u and u' that both reach this state but belong to different congruence classes with respect to $\mathcal{T}$, and hence need to be treated differently by the learner (and eventually also the inferred transducer). By splitting the state q into two, the learner gains the opportunity to adapt its behaviour to these distinct cases.

A strength of reinforcement learning is that it makes it possible to model dynamic environments, where the state behaviour changes over time. This can for instance happen when we are trying to predict the prices of auctioned advertising space in marketing, which changes based on season and market demand. We may therefore ask how a Q-based learner is affected when the target transduction $\mathcal{T}$ is modified during the learning process. For instance, $\mathcal{T}$ might initially replace every second a in its input strings with b, but after a certain number of episodes, it may change this pattern to only replace a every third time. In theory, the combination of a small amount of random exploration and the above-mentioned state-splitting approach should be able to accommodate such a moving target, and in Section 4.7 we conduct a simple experiment to test the idea.

In summary, we study reinforcement learning for string-to-string transductions. Our primary research question concerns the adaption of Q-learning to FST inference, while the secondary question revolves around the effectiveness of this approach in the face of a dynamically changing target transduction. The main contributions are the formalisation of FST inference within the context of reinforcement learning, and the provision of an experimental framework for assessing its effectiveness.

1.1. Related work

There is a rich literature on grammatical inference and machine learning of finite-state devices in general, and to some extent also of transducers. A pioneer in the field is Eisner [7] who presented an expectation-maximisation (EM) algorithm to train weighted finite-state transducers from data. The algorithm estimates the transition distributions based on observed input and output strings through the expectation step, and then updates the distribution parameters by maximising the expected log-likelihood through the maximisation step. Local convergence is achieved through repeated iterations of the expectation and maximisation steps.

The EM approach bears a similarity to Q-learning in the iterative execution-update process used to approximate the training goal, but differs in how information about the target transduction is presented to the learner: In Q-learning, the learner receives a potentially infinite sequence of input strings together with feedback on its attempts

at generating matching output strings, but it is never allowed to inspect the correct output strings directly. In the expectation-maximisation approach on the other hand, the learning algorithm is provided with a finite set of input-output pairs, but no additional feedback is given.

A step closer to reinforcement learning, we have algorithms for active learning of transducers. An early example is the work by Carme et al. [4] that aims to infer *node-selecting transducers*, which are an extension of *step-wise tree automata* [16]. Carme et al. show that node-selecting transducers can be learnt by querying an oracle capable of (i) correcting node selections made with respect to individual input trees, and (ii) giving counterexamples to the hypothesis that a given node-selecting transducer represents the target transduction.

Another work in this vein is that by Akram et al. [1] which applies active learning to probabilistic finite-state transducers. They propose an algorithm which learns the target transduction from two data sources: a finite corpus of sample input-output pairs, and an oracle with a fixed set of capabilities. To ensure the algorithm's effectiveness, the corpus must constitute a *characteristic sample* for the target transduction $\mathcal{T}$. In essence, this implies that every state and edge in a minimal transducer for $\mathcal{T}$ should be represented in the sample, and there must be sufficient contextual information for the learner to distinguish between strings that are not equivalent with respect to $\mathcal{T}$. In addition to inspecting the corpus, the learner can query the oracle for the total probability distribution assigned to strings with a given prefix w through an *extended prefix language* query. We return in Section 5 to discuss how an approximate version of this algorithm could be made to work in the current RL setting.

1.2. Outline

This paper is organised as follows. Section 2 recalls relevant concepts from formal language theory and statistics. The problem of learning string transductions as an instance of reinforcement learning is formulated in Section 3, and in relation to that, a basic learning algorithm is proposed. In Section 4, we evaluate and further develop this algorithm through practical experiments. Finally, Section 5 concludes the article by outlining directions for future work.

2. Preliminaries

The set of *natural numbers* (excluding zero) is denoted by $\mathbb{N}$ and the set of *non-negative reals* by $\mathbb{R}_{\geq 0}$. The *size* of a set S is denoted by $|S|$, and the *empty set* is written $\emptyset$. An *alphabet* Σ is a finite nonempty set. A *string* is an ordered sequence of zero or more symbols from Σ. The set of all strings over Σ is denoted by Σ^*, and the *empty string* is written ε. The length of a string $u \in \Sigma^*$ is written $|u|$, and we denote by $u[i]$, where $i \in \mathbb{N}$ and $1 \leq i \leq |u|$, the ith symbol of u.

Let A be a finite set. A *probability distribution* is a mapping $p : A \to \mathbb{R}_{\geq 0}$ such that for all $a \in A$, $0 \leq p(a) \leq 1$, and $\sum_{a \in A} p(a) = 1$. If A is a field and ϕ is a function over A, we may write $E_p[\phi]$ for the *expected value* of ϕ when its arguments are sampled from A with respect to p.

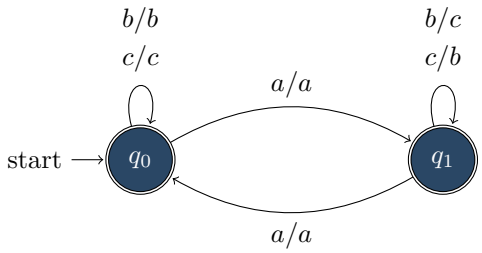


Figure 1: An FST over the alphabets $\Sigma = \Delta = \{a, b, c\}$. The state set is $P = \{q_0, q_1\}$, where q_0 is the initial state, and both states are final states (the FST is all-accepting). There are six transition rules whose effect can be described as follows: Every time the transducer encounters an a in its input string, it shifts between either faithfully replicating its input, or replacing b:s with c:s and vice versa.

To ensure that the reward is always well-defined, we restrict ourselves to string transductions that are total functions on their domains. In other words, for every input string there is exactly one correct output string. Moreover, we only allow transductions with bounded size increase to make it easier for the learner to align input and output strings. This leads us to the following definition:

Definition 1 Finite-State Transducer. An (all-accepting deterministic) finite-state string transducer (FST) is a tuple $M = (\Sigma, \Delta, P, q_0, \delta)$ where Σ is an alphabet of *input symbols*, Δ is an alphabet of *output symbols*, P is a finite set of *states*, $q_0 \in P$ is an *initial state*, and $\delta\colon (P \times \Sigma) \mapsto ((\Delta \cup \{\varepsilon\}) \times P)$ is a *transition function*.

We henceforth refer to all-accepting deterministic finite-state string transducers simply as 'transducers' when there is no risk for confusion.

Let M be an FST and let $\rho = ((q', w), (u, q)) \in \delta$ (remember that we can view every relation, and hence every function, as a set of tuples of input-output pairs). The *source* of the transition ρ is $src(\rho) = q'$, and the *target* is $tar(\rho) = q$. The *input string* of ρ is $inp(\rho) = w$ and the output string is $outp(\rho) = u$.

A *run* π of M is a sequence of transitions $\pi = \rho_1 \dots \rho_k$ in δ^* such that $src(\rho_1) = q_0$ and $tar(\rho_i) = src(\rho_{i+1})$ for all $i \in \mathbb{N}$ such that $1 \leq i \leq k-1$. The *input* and *output strings* of π are

$$inp(\pi) = inp(\rho_1) \cdots inp(\rho_k) \text{ and } outp(\pi) = outp(\rho_1) \cdots outp(\rho_k) \ ,$$

respectively. We denote by $runs_M(w)$ the set of all runs π of M such that $inp(\pi) = w$. The *transduction* computed by M is the mapping $\Sigma^* \to 2^{\Delta^*}$ given by

$$M(w) = \bigcup_{\pi \in runs_M(w)} outp(\pi) \ . \tag{1}$$

The transducer is a *relabelling* if $|w| = |u|$ for every transition $((q', w), (u, q)) \in \delta$. It should be clear that a relabelling transducer translates every input string to an output string of equal length. See Figure 1 for an example of a relabelling transducer.

3. Reinforcement learning

Reinforcement learning is one of the core paradigms in machine learning, and addresses the problem of optimising decision-making under dynamically changing conditions. At the centre is a learning algorithm which continually interacts with its environment in pursuit of some target objective. Initially, its understanding of the environment is limited. However, through a systematic process of observing and learning from the consequences of its actions, it gradually refines its ability to determine the most effective strategy for achieving its objective, and to revise this when the environment changes. This methodology, often described as a "trial and reward" approach, thus allows the algorithm to leverage real-time information, and choose those actions that yield the most rewarding outcomes.

More formally, the learning algorithm (henceforth, the learner), operates in an environment represented by a state s. Learning takes place over one or more *episodes*, each consisting of a finite number of discrete time steps T. At each time step $t \in \{1, ..., T\}$, the learner interacts with the environment, where the current environment's state information is represented as s_t. Based on s_t, the learner chooses an action from all available actions $a_t \in A$, where A is a finite discrete action set within our specific context. This action triggers a reward r_{t+1} from the environment and causes it to change to state s_{t+1}. The learner's objective is to maximise the cumulative return, which requires it to recognise and exploit properties of the unknown reward function.

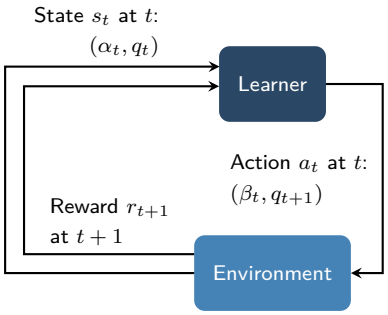


Figure 2: The learner's interaction with its environment is modelled as a Markov decision process. Observing a state s_t of the environment at time step t, the learner chooses an action a_t. The action yields a reward r_{t+1} and moves the environment into state s_{t+1}. The reward provides a feedback signal to the learner that guides subsequent decisions.

In our baseline application of RL to the inference of an FST M for a target transduction $\mathcal{T}$, we first choose a set of transducer states P with a designated initial state $q_0 \in P$. Then, for every natural number $j = 1, 2, \ldots$, we generate a string $u_j \in \Sigma^*$ and let $w_j = \mathcal{T}(u_j)$ and $v_j = \varepsilon$. At time $t \in [1, |u_j|]$, the learner observes the environment state $s_t = (u_j[t], q_t)$, where $q_t \in P$.[1] Based on the environment state, it chooses as next action a pair $(\beta_t, q_{t+1}) \in \Sigma^* \times P$ and we update v_j to $v_j\beta_t$. In return, the learner receives the reward r_{t+1}. If $t < |u_j|$ then this reward is 0, but if $t = |u_j|$ then it is computed based on the *Levenshtein* distance between the correct output w_j and the output v_j produced by the learner. This distance is the minimal number of single-symbol edits (insertion, deletion, or substitution) required to translate one of the strings into the other. Given a string $w = aw' \in (\Sigma^* \setminus \{\varepsilon\})$, we denote the symbol

[1]Recall that the environment states are distinct from the transducer states. To underline this difference, we denote the environment states by s_i, $i \in \mathbb{N}$, and the transducer states by p_i, $i \in \mathbb{N}$.

a by $head(w)$ and the string w' by $tail(w)$. More formally, the Levenshtein distance between w and v is then:

$$lev(w,v) = \begin{cases} |v| & \text{if } |w| = 0 \\ |w| & \text{if } |v| = 0 \\ lev(tail(v), tail(w)) & \text{if } head(v) = head(w) \\ 1 + \min \begin{cases} lev(tail(v), w) \\ lev(v, tail(w)) \\ lev(tail(v), tail(w)) \end{cases} & \text{otherwise} \end{cases}$$

Finally, the reward is defined as the negative logarithm of $(lev(w_j, v_j)+1)$ to encourage the learner to emulate the behaviour of $\mathcal{T}$. After the reward has been received, either we have $t = |u_j|$ and the episode ends, or there are still input symbols left to consume and the environment changes to the state $s_{t+1} = (u[t+1], q_{t+1})$. (We note here that the state q_{t+1} is the state chosen by the learner as part of its last action, and is not guaranteed to reflect the correct transition behaviour of some canonical FST for the target transduction.)

In Section 4.4 we also investigate the scenario where the main reward is distributed over the individual time steps, and not postponed until the end of the episode.

3.1. Q-learning

As mentioned in the introduction, we design our learning algorithm along the Q-learning approach. This means that the learner estimates the potential value of a state-action pair (s, a) as the maximal obtainable accumulated rewards that can be achieved by reaching the state s and then taking the action a to the end of its trajectory. The learner systematically records the values of all encountered (s, a) pairs in a lookup table, where these pairs serve as keys. By consistently selecting the most valuable action associated with a specific state, the learner can implement a decision-making policy. During its engagements with the environment, the learner not only relies on this table to guide its actions but also actively updates it based on the rewards resulting from those actions. The values thus aid the learner in its decision-making and evolve according to the feedback received. This interconnect process progressively brings the stored values closer to the actual values, and in doing so refines the learner's understanding of the environment.

Let us now consider how to best represent the environment. If computational complexity was not an issue, then the learner could in principle index the Q-value table with all possible combinations of a prefix on an input string (corresponding to an environment state) and any possible output string (corresponding to an action). However, since there is no upper bound on the length of these strings, this is not a feasible way forward. Instead, we consider more compact means of encoding the input and output strings. The perhaps most obvious choice, given our ambition to learn finite-state transducers, is to provide a set of states P which the learner can use to remember key properties of the input string consumed so far, together with the next symbol in the input string to be consumed. (In Section 4.4, we discuss other

alternatives.) Given that an action is a pair consisting of the symbol to output, and the next state to move to, this results in a Q-learning table *tab* that is a subset of the set $(\Sigma \times P) \times ((\Sigma \cup \{\varepsilon\}) \times P)$.

Now, let A be the set of all actions, and S be the set of all environment states. The value of taking an action $a_t \in A$ in state $s_t \in S$ (which in Q-learning is called the *Q-value*) is the expected value of the sum of all future rewards received from time step $t+1$ and onward. When formulating this value, it is standard to introduce a discount factor $\gamma \in [0 \cdots 1]$ to express that rewards in the near future are valued higher than those in the distant future. Moreover, a *policy* is a family of probability distributions $p = (p_s)_{s \in S}$ where $p_s : A \to \mathbb{R}_{\geq 0}$ is the probability distribution for a particular state s. Intuitively, the policy tells us how likely it is for the learner to take an action a in a state s. We can now express the value of a state-action (s_t, a_t) pair as follows:

$$val_p(s_t, a_t) = E_p[r_{t+1} + \gamma \cdot r_{t+2} + \gamma^2 \cdot r_{t+3} + \ldots + \gamma^{T-t-1} \cdot r_T \mid s_t, a_t] \,. \tag{2}$$

Recognising that Equation 2 contains the Q-value equation for (s_{t+1}, a_{t+1}) as a subterm, we can rewrite it as:

$$val_p(s_t, a_t) = E_p[r_{t+1} + \gamma \cdot val_p(s_{t+1}, a_{t+1}) \mid s_t, a_t] \,. \tag{3}$$

Since the value of $val_p(s_{t+1}, a_{t+1})$ is unknown to the learner, the learner uses its current estimation $tab(s_{t+1}, a_{t+1})$ as replacement when computing Equation 3. As the learning process continues, the learner must seek to minimise the error err_t, which is given by the difference between the new updated target and the previous estimation, that is,

$$err_t = r_{t+1} + \gamma \cdot tab(s_{t+1}, a_{t+1}) - tab(s_t, a_t) \,. \tag{4}$$

A policy $\hat{p}$ is optimal if, for every state, it always assigns the greatest likelihood to the action that leads to the greatest possible future reward. The set of optimal policies for the state share the same optimal action-value function:

$$val_{\hat{p}}(s_t, a_t) = E_{\hat{p}}[r_{t+1} + \gamma \cdot \max_{a_{t+1}} val_{\hat{p}}(s_{t+1}, a_{t+1}) \mid s_t, a_t] \,. \tag{5}$$

Based on the above equations, we can derive a recursive update formula, using the approximation of the optimal action-value function as learning objective:

$$tab(s_t, a_t) := tab(s_t, a_t) + \alpha[r_{t+1} + \gamma \cdot \max_{a_{t+1}} tab(s_{t+1}, a_{t+1}) - tab(s_t, a_t)] \,, \tag{6}$$

where $\alpha \in (0, 1]$, is the *learning rate*, that is, a constant which determines to what extent the learner adjusts its previous estimation based on *err*.

Through iterative refinement of the action value function, the learner progressively brings its estimates closer to the true values. This suggests a greedy strategy, where the learner consistently selects the highest-valued action in each state. However, for more effective adaptation to changes in the environment, the learner can be set to follow an *epsilon-greedy* strategy. In this approach, with a small probability $\epsilon \in \mathbb{R}_{\geq 0}$

where $0 \leq \epsilon \leq 1$, the learner deviates from the greedy strategy and instead chooses its next action randomly. The epsilon-greedy strategy encourages the learner to explore also high-risk actions, potentially acquiring valuable information which can lead to greater future rewards. Additionally, it allows the learner to reassess current strategies and discover new opportunities which may arise due to changes in the environment. Consequently, the learner's action selection policy is represented as:

$$a_t = \begin{cases} \text{argmax}_a \ tab(s_t,\ a) & \text{with probability } \ 1-\epsilon, \\ \text{random } a & \text{with probability } \epsilon \ . \end{cases}$$

4. Experiments

To study the ideas presented in Section 3, we implement a reinforcement learning algorithm in Python and apply it to infer a series of string transductions. First, we present the transductions involved and describe how the training data was sampled. We then continue to detail a simple experiment to give the reader an intuition about how the various design choices made in the algorithm influence its performance. Finally, we discuss variations of the reward function and the state space, and also what happens when the target transduction shifts during the inference process.

4.1. Data sets

To study different properties of the learning algorithm, we define the transductions presented in Table 1. For each transduction, we create a finite-state transducer and sample up to 10 000 input-output pairs. Only REVERSE needs to be treated differently since it cannot be represented by an FST. For this transduction, we therefore wrote a simple script to generate a suitable corpus of input-output pairs where the output is the reverse of the input string. During an experiment on a transduction $\mathcal{T}$, we repeatedly pick a pair (u, w) from the corpus belonging to $\mathcal{T}$, feed the input string u to the learner, and then provide it with feedback on its performance by relating its output string v to the correct output w. Henceforth we refer to w as the *target string*.

The FST-based sampling was done using the `pyfoma` library [2] for weighted FSTs. Since our FSTs are not weighted (or rather, they are weighted over the Boolean semiring), we set the weights of every transition to 1 so that we can extract a list of increasingly large input-output pairs. The extracted corpus is sorted according to the number of rule applications in ascending order, but when training, we randomly sample pairs.

4.2. Experimental setup

To describe the experimental setup, we take the inference of the transduction C-TAIL as an example (see Table 1 for its definition). In each experiment, we train the learner until it converges to the correct transduction, or an upper limit of N episodes has

[2] `https://github.com/mhulden/pyfoma`

Table 1: The transductions used in the experiments. With the exception of REVERSE, all can be represented by an FST, and in these cases we construct a minimal all-accepting FST for the transduction. The right-most columns indicate whether a transduction is a relabelling, and the minimal number of states needed.

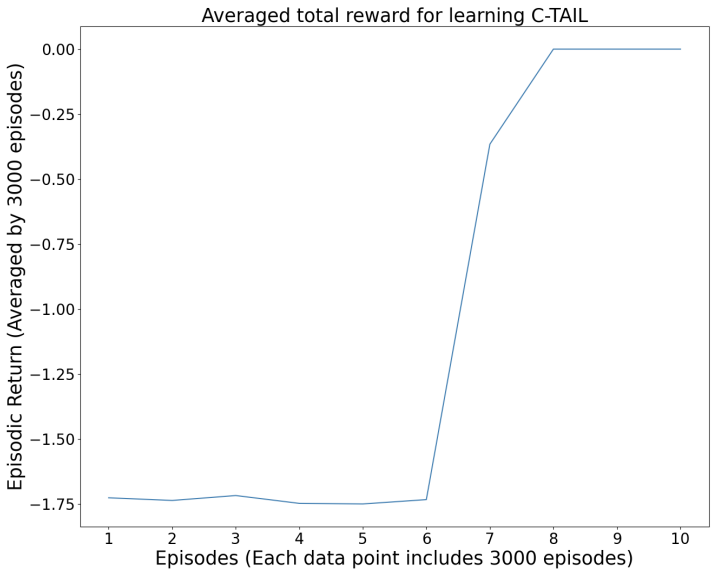
Name	**Description**	**Relab.**	**States**
IDENTITY	The output string is equal to the input string.	✓	1
	$A = (\{q_0\}, \{a, b, c\}, \{a, b, c\}, q_0, \delta, \{q_0\})$ $\delta = \{(q_0, a, a, q_0), (q_0, b, b, q_0), (q_0, c, c, q_0)\}$		
B-TO-A	Every b in the input string is replaced by an a in the output string.	✓	1
	$A = (\{q_0\}, \{a, b, c\}, \{a, b, c\}, q_0, \delta, \{q_0\})$ $\delta = \{(q_0, a, a, q_0), (q_0, b, a, q_0), (q_0, c, c, q_0)\}$		
REMOVE-B	Every b in the input string is deleted.	✗	1
	$A = (\{q_0\}, \{a, b, c\}, \{a, b, c\}, \delta, q_0, \{q_0\})$ $\delta = \{(q_0, a, a, q_0), (q_0, b, \varepsilon, q_0), (q_0, c, c, q_0)\}$		
C-TAIL	The output string is equal to the input string, until the first c is encountered. From that point on, the remainder of the input string is rewritten to c:s.	✓	2
	$A = (\{q_0, q_1\}, \{a, b, c\}, \{a, b, c\}, q_0, \delta, \{q_0, q_1\})$ $\delta = \{(q_0, a, a, q_0), (q_0, b, b, q_0), (q_0, c, c, q_1), (q_1, a, c, q_1), (q_1, b, c, q_1), (q_1, c, c, q_1)\}$		
ABC	The output is equal to the input, except when the substring 'abc' is encountered. In these cases, the c is changed to a b.	✓	3
	$A = (\{q_0, q_1, q_2\}, \{a, b, c\}, \{a, b, c\}, q_0, \delta, \{q_0, q_1, q_2\})$ $\delta = \{(q_0, a, a, q_1), (q_1, b, b, q_2), (q_2, c, b, q_0), (q_0, b, b, q_0), (q_0, c, c, q_0), (q_1, a, a, q_0), (q_1, c, c, q_0), (q_2, b, b, q_0), (q_2, a, a, q_0)\}$		
n-COUNT	Every nth input symbol is replaced by a c in the output string. The transducer below specifies 3-COUNT.	✓	$n \in \mathbb{N}$
	$A = (\{q_0, q_1, q_2\}, \{a, b\}, \{a, b, c\}, q_0, \delta, \{q_0, q_1, q_2\})$ $\delta = \{(q_0, a, a, q_1), (q_1, a, a, q_2), (q_2, a, c, q_0), (q_0, b, b, q_1), (q_1, b, b, q_2), (q_2, b, c, q_0)\}$		
REVERSE	Output the input string in reverse. This transduction cannot be represented by an FST.	✓	N/A

Algorithm 1 Learn a string transduction through reinforcement learning.

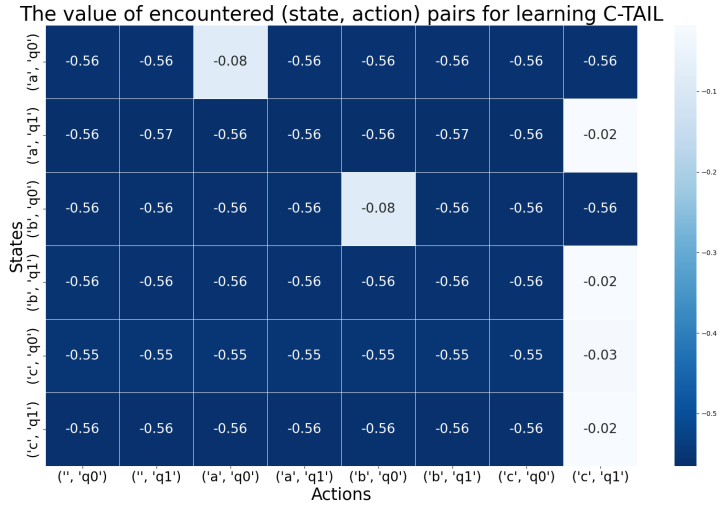
procedure LEARN($\mathcal{T}$ is the target transduction over the alphabets Σ and Δ, P is a set of predefined transducer states, $q_0 \in P$ is a fixed initial state, *threshold* is the number of times the learner needs to be correct before training ends)
  Initialise Q-learning table *tab* ▷ Set all values to zero.
  Initialise an empty set *return* ▷ To store the episodic rewards.
  repeat
    Draw (u, w) from $\mathcal{T}$ ▷ Generate the next training example.
    $v \leftarrow \varepsilon$ ▷ At first, the learner's output is empty.
    $q_t \leftarrow q_0$ ▷ Start from the initial state.
    $r \leftarrow 0$ ▷ The total episodic reward r is initially zero.
    for $t \leftarrow 1$ to $|u|$ **do** ▷ Until the end of the input:
      $(v_t, q_{t+1}) \leftarrow \text{argmax}_{(v'_t, q'_{t+1})} tab((u[t], q_t), (v'_t, q'_{t+1}))$ ▷ Choose action greedily, and
      $v \leftarrow vv_t$ ▷ extend the output string.
      if $t < |u|$ **then** ▷ If the end of the input has not been reached:
        $r_{t+1} \leftarrow 0$ ▷ withhold feedback,
      else ▷ otherwise
        $r_{t+1} \leftarrow -\log(lev(v, w) + 1)$ ▷ report distance from correct output.
      end if
      Update *tab* with r_{t+1} using Equation 6 ▷ Update Q-value table.
      $r \leftarrow r + r_{t+1}$ ▷ Update total reward.
    end for
    return.add(r)
  until the learner has been correct for *threshold* episodes ▷ Termination criteria
  Extract FST M from *tab* ▷ Read transition table from Q-value table.
  return (M, r) ▷ Return inferred FST and total reward.
end procedure

been reached. The learning algorithm is outlined in Algorithm 1. The user-defined parameters are – in addition to a corpus of samples of the target transduction $\mathcal{T}$ – a state set P, an initial state $q_0 \in P$, and a natural number *threshold* indicating how many times in a row the learner needs to be correct before it can be considered to have learnt the transduction. At the start of each episode, we randomly sample an input-output string pair (u, w) from the target corpus. The symbols of the input string are provided one at a time to the learner, which starts in the transducer state q_0. At each time step, it makes a greedy choice of what symbol to output and what transition to move to based on the current input symbol and transducer state; it does this by consulting the Q-value table. If an epsilon-greedy approach is used, the learner sometimes chooses an action at random instead. However, in this first example we let $\epsilon = 0$, so there is no non-greedy exploration. Once the entire input string has been consumed, it is rewarded in relation to the edit distance between its generated output and the target output. The reward is used to update the estimations in the Q-value table, whereupon the training enters the next episode.

In the case of C-TAIL, we trained the learner for 30 000 episodes without exploration (i.e., with $\epsilon = 0$), to teach it that when the symbol c is encountered, the rest of the input should be rewritten to c:s. The learner was limited to choosing a string in $\{a, b, c, \varepsilon\}$, and the available transducer states were $P = \{q_0, q_1\}$. Figure 3(a) shows the averaged total (episodic) return per 3 000 episodes acquired by the learner during training. Specifically, an episodic return closer to 0 means that the learner's



(a) C-TAIL episodic return



(b) C-TAIL heat map

Figure 3: Learning outcomes for C-TAIL. (a) The episodic returns obtained during 30 000 episodes while training on the C-TAIL transduction. The returns are averaged by 3 000 episodes. (b) The heat map of the Q-value table after training on C-TAIL for 30 000 episodes. The rows are indexed by environment states, and the columns by available actions. The lighter the colour, the greater the learner's estimation of taking an action in a particular environment state.

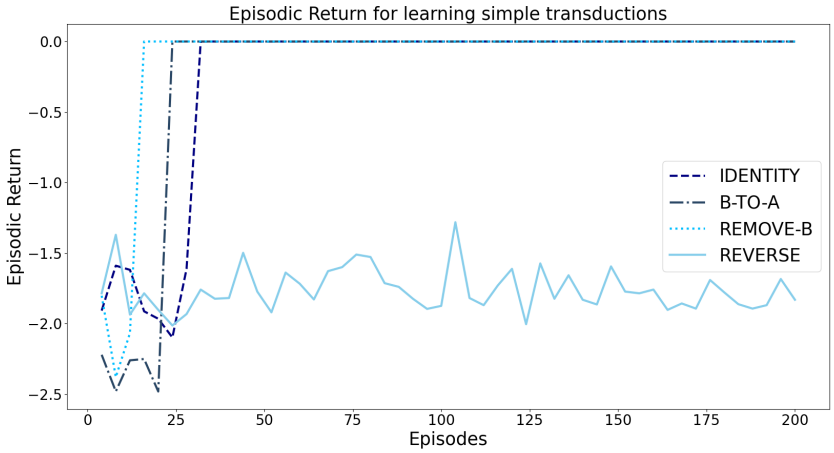


Figure 4: The episodic returns obtained by the learner when training on the transductions IDENTITY, B-TO-A, REMOVE-B, and REVERSE. For all transductions but REVERSE, the learner quickly identifies the target transduction.

output is close to the target string. Figure 3(b) shows a heat map of the Q-value table after training. It contains the learner's estimated values of taking different actions in different states. If the target transduction M' has been correctly inferred, then the heat map corresponds to a transition table for an FST M representing the transduction (under the semantics of Equation 1). In other words, $M(w) = M'(w)$ for every string w over the input alphabet. In the table of Figure 3(b), we can for example see that when the FST is in state q_0 and receives an a, it outputs an a and stays in the same state (as this cell has the greatest value, indicated with the lightest colour). However, if it is in state q_0 and sees a c, it instead transitions to the state q_1 (and outputs a c). Finally, we note that once it is in state q_1, it always stays in q_1 and outputs c:s, no matter what the input is. In other words, it has learnt C-TAIL.

4.3. Learning string transductions

Let us now see how the learner behaves on a more varied set of inference tasks. To this end, we set it to learn the following transductions: IDENTITY, B-TO-A, REMOVE-B, and REVERSE. Each learning problem involved 200 training episodes. Excepting the smaller number of training episodes N, the remaining parameters are as reported in Section 4.2. Figure 4 shows the learner's performances during the training. As we can see, the learner quickly completes the first three learning tasks, converging to acquire the maximum reward of 0.

It is not surprising that the learner struggles to understand the REVERSE transducer: The rewards are intended to reveal the relation between an action and the observed state, providing a scalar assessment that helps the learner to make optimal choices. However, in the REVERSE task, determining the correct action at a specific

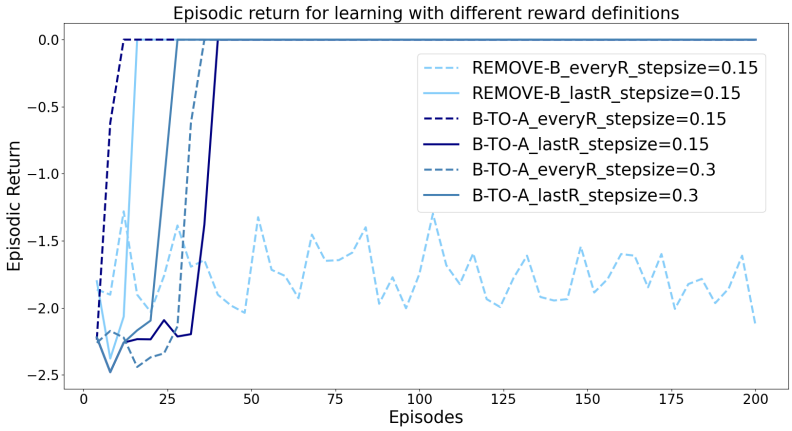


Figure 5: The episodic returns obtained when training on the transductions B-TO-A and REMOVE-B using different reward definitions. The labels "lastR" and "everyR" indicate that a reward is computed at the last time step and at every time step, respectively. The step size refers to the learning rate of the algorithm.

time step becomes impossible, because there is not enough information in the observed states to predict and hence leverage the rewards. To accurately learn REVERSE, one may try a learning algorithm that in addition to outputting symbols and shifting states can also take the actions of pushing or popping symbols on a stack, but this falls outside the scope of the current investigation.

4.4. Reward definition

The choice of reward function is central to reinforcement learning, and key to a successful algorithmic framework. In this section, we report on experiments to assess the learner's performance using two natural ways of providing the reward. In the first scenario, the learner receives 0 at each time step but gets the entire reward at the end of an episode, at which point it is computed as the edit difference between the complete learner-generated and target strings. In the second scenario, a partial reward is given at each time step, computed as the edit distance between the output string generated for the prefix u' of the input string consumed so far, and the output string produced by an FST for the target transduction $\mathcal{T}$ on u'. With both definitions, the learner's action influences the accumulated reward, but the learning happens at different paces and with varying degrees of stability. When the reward is only computed at the end, the progress is slower but also more stable. Conversely, when a reward is computed at each time step, the learner constantly "reflects" on its previous actions and trajectory. Here, the feedback gained from each action has a direct impact on the subsequent time steps and offers valuable insights into the structure of the target transduction. This generally increases the pace of the learning process, but it is questionable how realistic such a reward protocol would be in practice.

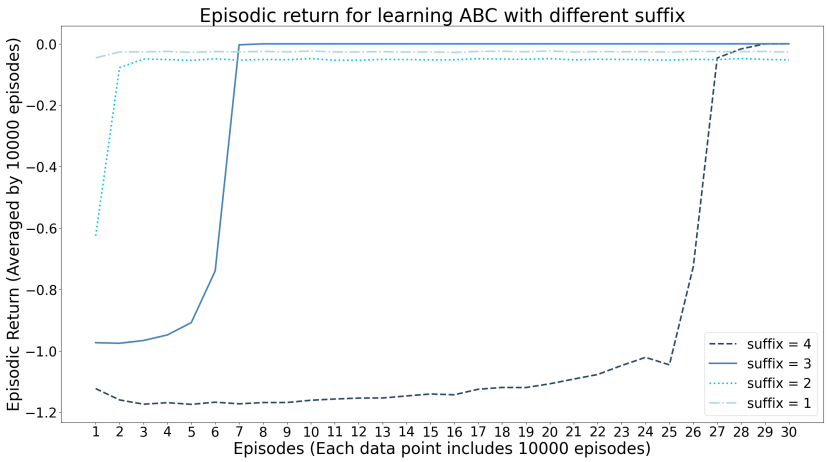


Figure 6: The episodic returns obtained by the learner when training on the transduction ABC using suffixes of varying lengths.

We compare the learner's performances using the two reward definitions outlined above. As our target transductions, we take B-TO-A and REMOVE-B. Figure 5 shows the performances under the alternative rewards in 200 training episodes. We start with an already high learning rate of 0.15 to underline the differences between the two reward definitions. Here, it is clear that the learner infers the B-TO-A transduction faster if it is rewarded at every time step. However, for the REMOVE-B task, the more frequent rewards actually prevent the learner from optimising its policy within the allotted 200 episodes. This we believe is because the target strings do not explicitly represent the "missing *b*:s", which makes them more difficult targets and can mislead the learner. Although the learner still has access to the complete target string at the end of the episode, the misconceptions it has picked up along the way are time-consuming to correct. Similarly, when doubling the learning rate, the speed of learning the B-TO-A transduction with rewards at every time step becomes slower than when rewards are computed only at the last time step. Also this, we believe, is explained by the extra work needed to correct erroneous updates of the Q-value table, made based on incomplete information in the early stages of an episode.

In short, while frequent rewards encourage a continuous strategy revision which may speed up learning, it may also have the opposite effect, because it makes the learning process more volatile. As we have seen, in some cases too frequent rewards can even prevent the learner from converging to the correct solution.

4.5. Environment states

Another important decision is what information to include in the environment states. In the experiments that we have looked at until now, we use a fixed set of transducer states provided by the user. However, this approach has its drawbacks: It either

Algorithm 2 Find and split transducer states with low predictive value.

procedure SPLIT(P is a set of states, *tab* is a Q-value table, A is the action space, *visits* is a table storing the number of visits for each table cell in *tab*)
  $q \leftarrow \text{argmin}_{q' \in P}$ENTROPY($q'$, *visits*) ▷ Split the least predictive transducer state
  return SPLITSTATE(q, *tab*, A, *visits*)
end procedure
procedure ENTROPY(q, *visits*) ▷ Compute the predictive value of each transducer state
  $val \leftarrow 0$
  $vis \leftarrow 0$
  for every cell $((u, w, q'), (\alpha, q))$ in *tab* **do**
    $val \leftarrow val + tab((u, w, q'), (\alpha, q)) \cdot visits((u, w, q'), (\alpha, q))$
    $vis \leftarrow vis + visits((u, w, q'), (\alpha, q))$
  end for
  return val/vis
end procedure
procedure SPLITSTATE(q, *tab*, A, *visits*) ▷ Split a transducer state into two
  Create a new state q_{new}
  for every cell $((u, w, q'), (\alpha, q))$ in *tab* **do** ▷ Update the Q-value table accordingly
    $val \leftarrow tab((u, w, q'), (\alpha, q))$
    $vis \leftarrow visits((u, w, q'), (\alpha, q))//2$
    $visits(((u, w, q'), (\alpha, q))) \leftarrow vis$
    tab.add$(((u, w, q'), (\alpha, q_{new})), val)$
    $visits$.add$(((u, w, q'), (\alpha, q_{new})), vis)$
    A.add$((\alpha, q_{new}))$
  end for
  return *tab*, A
end procedure

assumes the availability of a key property of the unknown transduction before the learning process commences, or it requires us to make an educated guess regarding the suitable number of states – posing negative consequences if our estimation is imprecise in either direction. An alternative approach is to use a fixed-size embedding of the input strings, so that the learner does not need to depend on transducer states for memory. For this to work, the embedding needs to capture every aspect of the string that is needed for the learner to choose the correct action. In computer vision and natural language processing, this approach has proved extremely successful. This may be attributed to the fact that humans often possess an abstract, symbolic understanding of images and language. Consequently, much of the data can be considered noise and omitted from the representation without adversely affecting the performance of the machine learning model. However, the situation is different in formal language theory, where any property of a string, in principle, is as likely as another to be the target for grammatical inference.

For the above-mentioned reasons, embeddings are only meaningful in the present setting if we take the target class of transductions into consideration. A possible embedding for transductions whose domain is a locally testable language [32] is to represent the portion of the input string currently being rewritten by a finite-length suffix. In other words, while the learner is iterating over the input, it can remember the last n symbols seen, but no more. To illustrate the idea, we applied the learner to the transduction ABC (see Table 1 for details), using suffixes of length $n \in \{1, \ldots, 4\}$ as environment states, and output symbols as actions. On the one hand, this robbed

Algorithm 3 Learn a string transduction through RL with state-splitting.

procedure Optimisation(ϵ is the fraction of the times the learner should explore, that is, choose an action at random, *number_of_episodes* is the number of episodes to run, r is the total episodic reward.)
  Initialise non-explored returns set *CleanReturn*
  $split \leftarrow False$
  $explore \leftarrow False$
  for $i \leftarrow 0$ to *number_of_episodes* **do**
    $explore \leftarrow (\text{Random}(0,1) < \epsilon)$
    if not *explore* **then**
      CleanReturn.add(r)
    end if
    if *split* **then**
      $tab, A \leftarrow \text{Split}(P, tab, A)$
      $split \leftarrow False$
    end if
    if $i =$ *CheckRules* **then**:
      $SumCleanReturn \leftarrow 0$
      for $j \leftarrow 0$ to 100 **do**
        $SumCleanReturn \leftarrow SumCleanReturn + CleanReturn_{|CleanReturn|-j}$
      end for
      if $SumCleanReturn > -0.01$ **then**
        break
      end if
      $var1 \leftarrow$ the variance of the values in $CleanReturn[-2000:]$
      $var2 \leftarrow$ the variance of the values in $CleanReturn[-4000:-2000]$
      $norma\_delta\_var \leftarrow (|var2 - var1|)/(var2 + var1)$
      $split \leftarrow (norm\_delta\_var < 0.001)$
    end if
  end for
end procedure

the learner of the ability to choose the next environment state, but on the other, it ensured that the learner always had complete information about to n most recently encountered symbols of the input string.

We ran 300 000 episodes for each experiment, with a learning rate of 0.001. Since this particular transduction makes it easy to compute helpful immediate rewards (one simply uses the same length of the suffixes from the generated string and the target string), we reward the learner after each time step, as discussed in Section 4.4. This speeds up the learning process and underlines differences in performance. Figure 6 shows the performance of the learner learning the ABC transduction with varying-length suffixes. As we can see, the learner is unable to infer the target transduction when it is given a suffix of length two or shorter. It still manages to obtain a fairly high average reward by simply replicating the input string to the output. However, with a suffix of length three it learns to faithfully represent ABC and obtain the full reward. The learner also achieves this with a suffix of length four, but convergence to the correct transduction takes longer.

Although string embeddings are helpful when we know what properties of the input strings are worth preserving, this cannot be the case in general. Moreover, even when we as in the case of locally testable languages know that it is enough to look at a suffix of the input, the number of environment states will be exponential in the length of

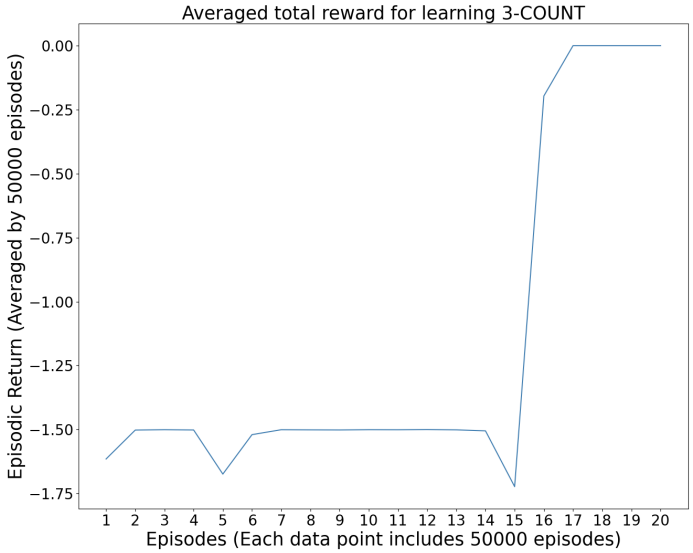


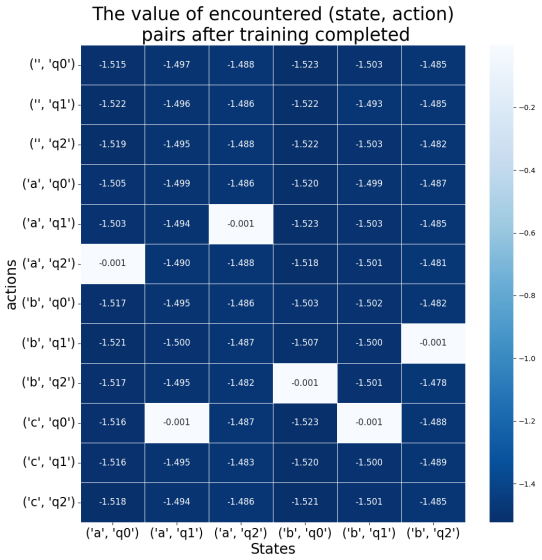
Figure 7: The average (non-exploring) episodic returns obtained by the learner with state splitting when trained on the transduction 3-COUNT.

these suffixes. For these reasons, we now turn to a more dynamic means of providing the learning algorithm with memory.

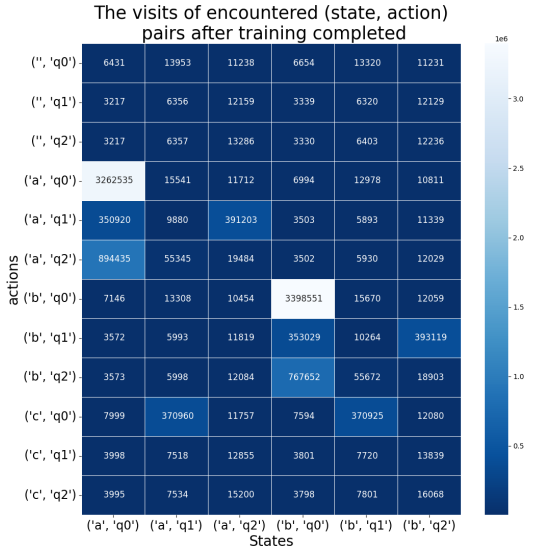
4.6. Adding transducer states

Let us recall that if the learner has too few transducer states to work with, then it will fail to infer the target transduction and hence receive a non-optimal reward. In this section, we present a technique that allows the learner to detect when the number of transducer state hinders progress. In such cases, the learner selects a transducer state for which there seems to be no suitable action and divides it into two. The underlying idea is that if there is not a good choice of action, then it is likely because the state is reached by strings that represent different syntactic classes with respect to the target transduction, so there is not one appropriate way forward. By splitting the state, it becomes possible for the learner to better separate between these classes. This use of state-splitting is influenced by Petrov et al. who extended an algorithm for grammatical inference of context-free grammars [13] to alternately split and merge nonterminals to maximise the likelihood of the target corpus [23]. These ideas were later formalised by Dietze [5] into an algorithmic framework for supervised learning of weighted tree automata, i.e., a generalisation of finite-state string automata.

The algorithm is outlined in Algorithm 2 and Algorithm 3. The first of these defines a procedure SPLIT which in turn calls a procedure ENTROPY. The purpose of ENTROPY is to find a transducer state that has been frequently visited by the



(a) Final Q-value heat map for 3-COUNT



(b) Final visit heat map for 3-COUNT

Figure 8: The heat map for learning the 3-COUNT transduction. The rows are indexed by environment states, and the columns by possible actions. (a) Q-value heat map: the lighter the colour, the greater the learner's estimated value of taking an action in a particular environment state. (b) Visit heat map: the lighter the colour, the more frequently the learner has taken an action in a particular environment state.

algorithm, but for which the values of all actions according to the Q-value table are still low. Once such a state has been picked, the learner calls the procedure SplitState to divide the state into two. The cells in the Q-value table stemming from the split state inherit the values of the split cells, and are considered to have been visited half as often as their parent cells. If we had set the visits to zero, then the new states would be essentially immune to future splitting, but if we had kept the full number of visits, then they might be split immediately again.

Let us now proceed to the main algorithm, that is, to Optimisation (see Algorithm 3). This algorithm is similar to Algorithm 1, with a number of small changes. The first is that the algorithm keeps track of the variance of the reward received, and if this stagnates, then it splits a state into two. The idea is that when the variance ceases to change, then this is either because the correct transduction has been inferred (in which case the algorithm halts), or because the state space has been fully explored and there still has been no break-through. At the outset, the algorithm is given a single transducer state to work with, which is sufficient for the first three transductions of Table 1, but not the rest. The second change is that the algorithm now uses exploration, where an ϵ fraction of the time chooses an action at random. For this reason, the algorithm also uses a set of variables labelled "clean" to be able to track the reward obtained in those cases where it acts purely greedy, and bases its decision on whether the target transduction has been acquired on these.

To evaluate this approach, we let the learner to infer the transduction 3-count. The exploration rate is set to $\epsilon = 0.0001$ and the learning rate to $\alpha = 0.001$. We ran 1 000 000 episodes: Figures 7 and 8 show the averaged clean episodic return during the training and the final Q table. The learner split states at episode 23 000 and 662 000, and terminated at 984 000 episodes. Looking at the heat map in Figure 8, we can see that the learner has discovered a valid transition table for an FST representing 3-count: If the FST it reads an input symbol in state q_0, then it outputs the same symbol and moves to q_2. The case is similar for q_2, but the move is now to q_1. If the FST is in q_1 and reads a symbol, then it moves back to q_0 and outputs a c. This completes the loop and yields a correct representation of 3-count. It is worth remarking that the order in which the states appear in the loop (q_0, q_2, q_1) does not match the order in which they were created (q_0, q_1, q_2), and no such correspondence can be expected in general.

Figure 8(b) shows the number of times the learner has visited each state-action combination. Due to the exploration, there is some noise in the table, but for each environment state, the actions that are rated the highest are generally also those that are visited the most often. However, we also see an influence of when the transducer states were created: Initially, the learner only had the transducer state q_0 to work with, so this created a looping behaviour that is still visible in the final table. That is, when in state q_0, the learner has frequently returned to the same state and simply copied its input symbol to the output. This was clearly a good strategy while the learner had too few states to capture the full cyclic behaviour of 3-count, as it meant that 2 out of 3 symbols in the output string were correct. However, as more states were added, exploration allowed the learner to shift over to the correct cycle length and settle on the target transduction.

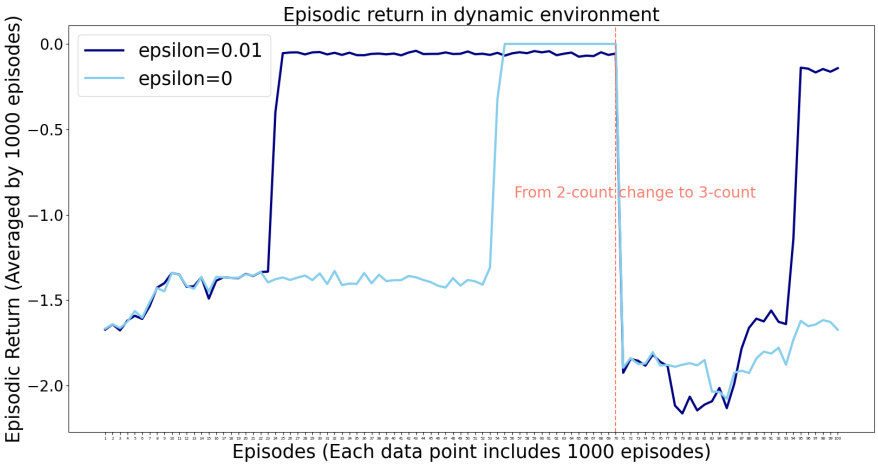
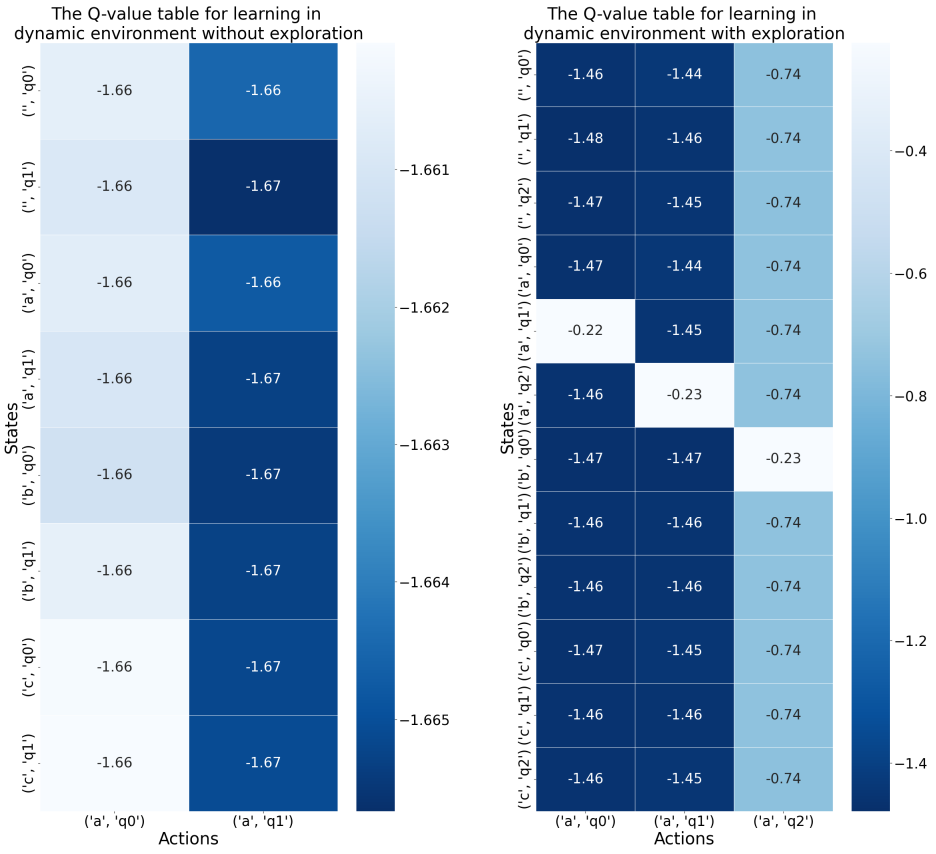


Figure 9: The episodic returns obtained by the learner in the dynamic environment. The target transduction is initially 2-COUNT, but at episode 70 000, it is replaced by the transduction 3-COUNT.

4.7. Performance in a dynamic environment

To conclude the experimental section, we examine the behaviour of the state-splitting learner in a dynamic environment. For this purpose, we train it on the n-COUNT transduction for 100 000 episodes, where the value of n changes during training. To speed up convergence, we use a unary input alphabet which means that the Q-table will remain fairly small. We choose a learning rate of $\alpha = 0.001$ and combine this with two different exploration rates, namely $\epsilon = 0.01$ and $\epsilon = 0$. Figure 9 illustrates the comparison of episodic returns with and without exploration. Let us first consider the case where it uses exploration, that is, when ϵ is non-zero. Initially, the value of n is two. As the learner begins with a single state, it decides around episode 22 000 that it lacks sufficient states and chooses to split one. Since it now has sufficiently many states, it quickly converges to the correct solution. It does not reach 0 in the plot because for more precise comparisons, we average every 1 000 episodic returns including the explored ones. The learner then continues at the theoretical optimal reward level, until the value of n changes to 3 at episode 70 000. The learner begins to struggle again and decides around episode 86 000 that another state is necessary. After this, the learner regains its footing and uses its three transducer states to accurately represent 3-COUNT, after which it terminates.

In the case where the learner does not explore, it fails to learn the target transduction. This is because it takes much longer to understand how to act in a particular transducer state, and its incapability to detect the environment's new changes thus makes the learning even slower. This is visible in the heat maps of Figures 10(a) and Figure 10(b) which represent the heat maps for the final Q-value table without and with exploration, respectively. The former only makes full use of the initial state q_0, while the latter in addition makes use of states q_1 and q_2.



(a) n-COUNT without exploration.

(b) n-COUNT with exploration.

Figure 10: The heat map of the Q-value table for learning dynamic transductions. The rows are indexed by environment states, and the columns by possible actions. The lighter the colour, the higher the value. (a) Q-value table for learning dynamic transductions without exploration. (b) Q-value table for learning dynamic transductions with exploration.

The reader may have noticed that our approach incorporates state splitting but lacks state merging. Consequently, if the learner initially sets out to learn a complex transduction that is later replaced by a simpler one, it may end up with an excess of states. While this might slow down continued learning, it does not hinder the learner from adapting to the new transduction, since it can choose not to use some

of its transducer states. For future research, it would be interesting to explore the possibility of identifying under-utilised transducer states on the fly, and if they exist, merging them. This way, the learner could maintain sufficiently many states to make progress without being held back by a needlessly complex state space.

5. Conclusion and future work

We have presented an RL algorithm based on Q-learning for learning string-to-string transductions. We then explored different methods of representing information about the input strings during inference: using a fixed set of states, working with suffixes, and allowing the learner to add new states when the learning process stagnates. Among these approaches, the last one seems most reasonable for real-world settings, as it requires no *a priori* information about the target transduction and can, in principle, accommodate any transduction that can be represented by an FST. We have also demonstrated the effectiveness of this approach in dynamic environments where the target transduction changes during inference, and shown that the learner can make progress even when having to wait until the end of each episode to receive non-trivial feedback on its actions.

Despite these efforts, we have only scratched the surface of the intersection of transduction inference and RL. As noted in Section 2, we have restricted ourselves to transducers with bounded size increase, that is, we do not allow the transducer to output symbols without also consuming an input symbol. This makes it easier for the learner to align the input and output strings, but it also means that a large and interesting class of transductions is not supported. Future work should therefore look at ways of removing this restriction without sacrificing too much computational efficiency. A potential solution is to adopt curriculum learning [27], where the learner is provided with increasingly more complex input strings to rewrite.

Another natural generalisation is to explore RL of probabilistic FSTs, where a normalisation step is added to turn the columns of the Q-value table into probability distributions over the available actions. A challenge here is that for every input string, there is no longer a single output that is correct, but rather a distribution over outputs. It is not clear how a reward based on the Levenshtein distance could be generalised to such a setting. However, one possible way forward is to allow the learner to compute, for every input string, a compact representation of a distribution over matching output strings, and then use Kullback–Leibler divergence [10] to provide a reward that is inversely proportional the difference between the learner's distribution and the target distribution. Also this is however not straightforward, as even in the case for relabellings this would mean summing over finite but potentially exponentially large sets (in relation to the size of the input), so some kind of approximation technique would be necessary.

It also remains to restrict the framework to transductions that compute the identity function on a regular subset of their input domain, in other words, to finite-state automata, since this would allow for benchmarking against other statistical and symbolic inference methods. This would however require that the datasets used for evaluation are sufficiently large, so that RL can be simulated through repeated sampling.

Let us now return to the active-learning algorithm by Akram et al. [1] discussed in the introduction. Recall that that algorithm realises exact learning by collecting a characteristic set for (an FST for) the target transduction. Provided that some assumptions are made about the distribution of input strings, the learning setting proposed here could accommodate a statistical version of this algorithm, because the likelihood of the learner being exposed to a characteristic sample would increase over time. Furthermore, the EXP queries could be approximated by counting the frequencies of relevant prefixes in the data. We leave an empirical comparison between such an approximated version of Akram et al.'s algorithm and the Q-learning approach as a topic for future work. However, to truly support exact learning and guarantee that the learner converges to a canonical representation of the target transduction, one must alter the learning setting itself. Again, one could build on the work by Akram et al. but allow the learner to choose what input strings it is given, and then devise a strategy that allows it to collect a characteristic set for the target transduction.

Finally, in the realm of reinforcement learning, there is previous work on multi-agent systems, where the learning algorithms are represented by finite state automata and learn individually but are penalised collectively after each action is taken [20]. It is found that when the reward function permits so-called Nash equilibria [21], then the set of learners will converge to one of these. This suggests an interesting line of future research, namely learning complex transduction obtained through compositions of simpler functions.

Acknowledgements

We are grateful to the reviewers for their time and effort. Their feedback was immensely valuable for improving the manuscript. Additionally, we would like to express our gratitude to Martin Berglund for his literature recommendations on string embeddings, and to Yikun Hou for providing suggestions on optimisation.

Finally, we would like to express our gratitude to Henning Fernau for his invaluable contributions to the research community, as well as for the kindness and encouragement he has extended to Björklund throughout her research career. Fernau's combination of a sharp intellect with a warm heart and generous nature makes him a role model to look up to.

References

[1] H. I. Akram, C. La Higuera, C. Eckert, Actively learning probabilistic subsequential transducers. In: J. Heinz, C. Higuera, T. Oates (eds.), *Proceedings of the Eleventh International Conference on Grammatical Inference*. Proceedings of Machine Learning Research 21, PMLR, University of Maryland, College Park, MD, USA, 2012, 19–33.

[2] R. Bellman, A Markovian decision process. *Indiana University Mathematics Journal* **6** (1957), 679–684.

[3] J. Berstel, *Transductions and context-free languages*. Teubner Studienbücher: Informatik 38, Teubner, 1979.

[4] J. Carme, R. Gilleron, A. Lemay, J. Niehren, Interactive learning of node selecting tree transducer. *Machine Learning* **66** (2007), 33–67.

[5] T. Dietze, *A Formal View on Training of Weighted Tree Automata by Likelihood-driven State Splitting and Merging*. Ph.D. thesis, Technische Universität Dresden, 2004.

[6] S. Eilenberg, *Automata, Languages, and Machines*. Number pt. 2 in Automata, Languages, and Machines, Academic Press, 1974.

[7] J. Eisner, Expectation semirings: Flexible EM for learning finite-state transducers. In: *Proceedings of the ESSLLI Workshop on Finite-State Methods in NLP Helsinki*. 2001, 1–5.

[8] R. Eyraud, S. Ayache, Distillation of weighted automata from recurrent neural networks using a spectral approach. *Machine Learning* (2021), 1–34.

[9] W. Foland, J. H. Martin, Abstract meaning representation parsing using LSTM recurrent neural networks. In: R. Barzilay, M.-Y. Kan (eds.), *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. ACL, 2017, 463–472.

[10] H. Jeffreys, *The Theory of Probability*. Third edition, Oxford University Press, 1961.

[11] S. Jiampojamarn, C. Cherry, G. Kondrak, Integrating joint n-gram features into a discriminative training framework. In: R. Kaplan, J. Burstein, M. Harper, G. Penn (eds.), *North American Chapter of the Association for Computational Linguistics*. ACL, 2010, 697–700.

[12] L. P. Kaelbling, M. L. Littman, A. W. Moore, Reinforcement learning: A survey. *Journal of Artificial Intelligence Research* **4** (1996) 1, 237–285.

[13] D. Klein, C. D. Manning, Accurate unlexicalized parsing. In: *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*. ACL, Sapporo, Japan, 2003, 423–430.

[14] W. Kuich, A. Salomaa (eds.), *Semirings, Automata, Languages*. Springer-Verlag, Berlin, Heidelberg, 1985.

[15] V. I. Levenshtein, Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics. Doklady* **10** (1965), 707–710.

[16] W. Martens, J. Niehren, On the minimization of XML schemas and tree automata for unranked trees. *Journal of Computer and System Sciences* **73** (2007) 4, 550–583. Special Issue: Database Theory 2005.

[17] G. Nicolai, S. Najafi, G. Kondrak, String transduction with target language models and insertion handling. In: S. Kuebler, G. Nicolai (eds.), *Proceedings of the 15th Workshop on Computational Research in Phonetics, Phonology, and Morphology*. Association for Computational Linguistics, Brussels, Belgium, 2018, 43–53.

[18] J. Nivre, Incremental non-projective dependency parsing. In: C. Sidner, T. Schultz, M. Stone, C. Zhai (eds.), *Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics; Proceedings of the Main Conference*. ACL, 2007, 396–403.

[19] J. Nivre, Algorithms for deterministic incremental dependency parsing. *Computational Linguistics* **34** (2008) 4, 513–553.

[20] A. Nowé, K. Verbeeck, M. Peeters, Learning automata as a basis for multi agent reinforcement learning. In: K. Tuyls, P. J. Hoen, K. Verbeeck, S. Sen (eds.),

Learning and Adaption in Multi-Agent Systems. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, 71–85.

[21] M. J. Osborne, A. Rubinstein, *A course in game theory*. The MIT Press, Cambridge, USA, 1994. Electronic edition.

[22] D. Perrin, Finite automata. In: J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Elsevier and MIT Press, 1990, 1–57.

[23] S. Petrov, L. Barrett, R. Thibaux, D. Klein, Learning accurate, compact, and interpretable tree annotation. In: N. Calzolari, C. Cardie, P. Isabelle (eds.), *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*. ACL, 2006, 433–440.

[24] M. Riley, C. Allauzen, M. Jansche, OpenFST: An open-source, weighted finite-state transducer library and its applications to speech and language. In: M. Ostendorf, M. Collins, D. W. O. Shri Narayanan, L. Vanderwende (eds.), *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics, Companion Volume: Tutorial Abstracts*. ACL, Boulder, Colorado, 2009, 9–10.

[25] A. Salomaa, M. Soittola, *Automata-theoretic Aspects of Formal Power Series*. Texts and monographs in computer science, Springer-Verlag, 1978.

[26] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. P. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, D. Hassabis, Mastering the game of go without human knowledge. *Nature* **550** (2017) 7676, 354–359.

[27] P. Soviany, R. T. Ionescu, P. Rota, N. Sebe, Curriculum learning: A survey. *International Journal of Computer Vision* **130** (2022) 6, 1526–1565.

[28] A. T. Suresh, B. Roark, M. Riley, V. Schogol, Distilling weighted finite automata from arbitrary probabilistic models. In: H. Vogler, A. Maletti (eds.), *Proceedings of the 14th International Conference on Finite-State Methods and Natural Language Processing*. 2019, 87–97.

[29] R. S. Sutton, A. G. Barto, *Reinforcement Learning: An Introduction*. Second edition, The MIT Press, 2018.

[30] C. Wang, N. Xue, Getting the most out of AMR parsing. In: M. Palmer, R. Hwa, S. Riedel (eds.), *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Copenhagen, Denmark, 2017, 1257–1268.

[31] C. J. C. H. Watkins, P. Dayan, Q-learning. *Machine learning* **8** (1992), 279–292.

[32] Y. Zalcstein, Locally testable languages. *Journal of Computer and System Sciences* **6** (1972) 2, 151–167.

[33] S. Zhang, X. Ma, K. Duh, B. Van Durme, AMR parsing as sequence-to-graph transduction. In: A. Korhonen, D. Traum, L. Màrquez (eds.), *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. ACL, 2019, 80–94.