Discovering Editing Rules by Deep Reinforcement Learning

Yinan Mei, Shaoxu Song, Chenguang Fang
BNRist, Tsinghua University
{myn18@mails., sxsong@, fcg19@mails.}tsinghua.edu.cn

Ziheng Wei, Jingyun Fang, Jiang Long
Huawei Cloud Computing Technologies Co., Ltd.
{ziheng.wei, fangjingyun, longjiang4}@huawei.com

***Abstract*—Editing rules specify the conditions of applying high quality master data to repair low quality input data. Discovering editing rules, however, is challenging, since it considers not only the well curated master data but also the large-scale input data, an extremely large search space. A natural baseline, namely EnuMiner, costly enumerates the rules with possible conditions from both master and input data. Although several pruning strategies are enabled, the algorithm still takes a long time when the enumeration space is large. To avoid enumerating all candidate rules during mining, we argue to model the rule discovery process as a Markov Decision Process. Specifically, we discover editing rules by growing a rule tree where each node corresponds to a rule. The algorithm generates a new rule from the current node as a child node. We propose a reinforcement learning-based editing rule discovery algorithm, RLMiner, which trains an agent to wisely make decisions on branches when traversing the tree. Following the idea of evaluating rules, we design a reward function that is more in line with rule discovery scenarios and makes our algorithm perform effectively and efficiently. The experimental results show that our proposed RLMiner can mine high-utility editing rules like EnuMiner and scale well on the datasets with many attributes and large domains.**

***Index Terms*—Editing Rule, Rule Discovery, Master Data, Reinforcement Learning, Data Repairing**

I. Introduction

Due to various errors introduced by humans and machines, data quality issues including inconsistency, conflict and violation are widely observed [11]. Low-quality data not only require more time to maintain [15] but also prevent us from discovering the potential value in the data. There are many studies in data cleaning, constraint-based [12], [29], rule-based [42], or learning-based [44]. Compared to ML-based approaches, constraint and rule-based approaches are easier to interpret and thus helpful for users to understand the data. Most data cleansing methods fix errors based solely on input data. However, in many cases, it is insufficient to fix errors with input data alone. For example, assuming that we have a functional dependency (FD) ZIP $\rightarrow$ AC, we still do not know how to fix t_3 in Figure 1(a). With the external data as shown in Figure 1(b), we can find the fix for $t_3[AC]$, i.e., "571".

For this reason, editing rules (eR) [18] are designed to make use of relational master data for cleaning, where master data (a.k.a. reference data) is a repository of high-quality data. With the development of master data management (MDM) [25], [47], many enterprises such as IBM, SAP, Microsoft, and Oracle maintain master data [18]. KATARA [13] and detective rules [20] design graph-based rules to leverage the knowledge base as external data. Note that a knowledge base can also be extracted as relational master data [30]. For simplicity and universality, we choose to study editing rules in this work.

	Name	City	ZIP	AC	Phone	Sex	Case	Date	Overseas
t_1	Kevin	HZ	NULL	NULL	325-8455	Male	NULL	2021-12	No
t_2	Kyrie	BJ	10021	010	358-1553	NULL	contact with imports	2021-11	No
t_3	Robin	HZ	31200	NULL	325-7538	Male	Others	2021-12	Yes

(a) Input data (Registration Information)

	FN	LN	City	Zip	AC	Phone	Sex	Infection	Date
s_1	Kevin	Lee	SZ	51800	755	625-0418	Male	contact with imports	2021-10
s_2	Kyrie	Wang	BJ	10021	010	358-1563	Female	contact with imports	2021-11
s_3	Kevin	Sun	HZ	31200	571	325-8465	Male	contact with patient	2021-12
s_4	Susan	Lu	HZ	31200	571	325-8931	Female	contact with patient	2021-12

(b) Master relation (National COVID-19 Records)

Fig. 1: Example of (a) input data D and (b) master data D_m

Example 1 (editing rules)**.** *Consider two tables in Figure 1.*[1] *To prevent the spread of COVID-19, the local tourism department asks passengers who have been infected with COVID-19 to register for health monitoring. Since there are some typos and omissions when inputting data, self-reported registration information has some errors and missing values (marked in red) as shown in Figure 1(a). To improve the information of passengers for accurate health monitoring, we use the national COVID-19 records as master data (Figure 1(b)) to fix the registration information. It is worth explaining here why we do not directly use the clean national records and chose to repair the registration information. The reason is that the master data may not be comprehensive. For instance, the national data do not include information of patients infected overseas, e.g., $t_3[Overseas] = Yes$. As supplementary, it is still worthwhile to curate the self-reported data.*

Consider an editing rule (eR),

$$\varphi_0 = ((City, City), (Date, Date)) \rightarrow (Case, Infection),$$
$$t_p[City, Date, Overseas] = (HZ, 2021\text{-}12, No))$$

It means that if the tourist, living in "HZ" ($t_p[City] = HZ$), was infected in "2021-12" ($t_p[Date] = 2012\text{-}12$) and not

Shaoxu Song (https://sxsong.github.io/) is the corresponding author.

[1]For privacy reasons, examples are revamped by the customer of Huawei.

oversea (t_p[Overseas] = No), then the infection case is "contact with patient" according to the master data D_m. Since master data does not record patients infected overseas, the condition t_p[Overseas] = No in the pattern avoids incorrect repair of t_3.

A. Challenge

The existing work [18] studies certain fixes with editing rules but does not devise the corresponding discovery algorithm. Discovering editing rules is challenging, since it has to consider the conditions in both the master data and the input data, such as t_p[Date] = 2021-12 and t_p[Overseas] = No.

Although the discovery of matching rules also considers two data sources [40], it treats both sources equally and identifies their matching relationships. The matching rule (MD) discovery algorithms [32], [37] are not applicable to discover editing rules (eR) for the following reasons. (1) The labelling is different. MD discovery requires to label the pairs from two sources denoting the same entity in the real world. The labelling for eR discovery, however, is optional. We alternatively utilize the labelled truths of errors only in the input data source, to calculate the Quality measure. It evaluates the accuracy of the possible repairs in the input table by eR, rather than the matching of data from two tables by MD. (2) The evaluation is different. The confidence of MD [39] denotes the maximum proportion of pairs from two sources that can satisfy the matching rule. In contrast, the editing rules target on certain fixes of the input data referring to the master data [18]. That is, the Certainty measure is evaluated on the tuples in the input table, telling whether the tuples can be fixed certainly. (3) The syntax is different. MD considers similarity matching relationship of values, but does not have any condition to discover. On the contrary, eR considers equality relationship of values and needs to discover the conditions in patterns.

To discover the conditions, one may mine conditional functional dependencies (CFD) [16] from the clean master data and use them as editing rules. However, such a discovery strategy ignores the possible conditions from the input data such as t_p[Overseas] = No. Moreover, since the master data may not be extensive, the rules discovered from master data could be underfitting. Please see Section V-B2 for a comparison.

B. Intuition

A natural idea is thus to enumerate the conditions from both the input and master data for possible editing rules, namely EnuMiner. Although several pruning strategies could be enabled as introduced in Section II-D, it is not surprising that EnuMiner is still costly given the huge space of enumeration. Again, please see Section V for an evaluation.

A more practical study is how to find editing rules that are still desired, without enumerating all the conditions. To this end, we (1) formalize several metrics for evaluating desired editing rules, and (2) consider reinforcement learning (RL).

The rationality of using reinforcement learning for discovering editing rules is as follows. In essence, both playing Go [34]–[36] and discovering eRs need to make the decision according to the current state. For playing Go, we need to decide the next step based on the chess pieces on the board. To discover editing rules, we also need to decide which attribute pair or attribute value pair should be added. The decisions made will determine whether we win a game of Go or succeed in discovering a set of correct editing rules.

It is different from frequent pattern mining [9], [19]. (i) Instead of finding all frequent patterns, we only need to discover a subset of editing rules that can have certain fixes for the input data. (ii) While efficient algorithms are available for frequent pattern mining [9], [19], effective pruning or heuristic for discovering editing rules is absent. As presented below, the heuristic is still costly to search the huge space of remaining possible rules. In this sense, reinforcement learning can guide the discovery process to wisely explore the search space.

Our major contributions in this work are as follows:

(1) We first formalize the eR discovery problem and analyze the challenge. A set of metrics is proposed to measure the utility of editing rules.

(2) We devise a novel RL-based editing rule mining method, named RLMiner. It trains an agent to wisely discover rules without enumerating values as conditions.

(3) We conduct extensive experiments on real datasets. The results demonstrate that RLMiner can mine the desired editing rules that repair data as accurately as EnuMiner. Moreover, RLMiner is much more efficient than EnuMiner and scales well on the datasets with many attributes and large domains.

The remainder of this paper is organized as follows. First, we introduce the preliminary in Section II. Section III presents RLMiner. The implementation details of RLMiner are in Section IV. Our extensive experimental evaluation is reported in Section V. We introduce the related work in Section VI. Finally, Section VII concludes this paper.

II. PRELIMINARY

A. Editing Rules

Editing Rules [18] are proposed to utilize the high-quality master data D_m to fix errors in the low-quality input data. We use R_m and R to denote the schema of D_m and D.

Definition 1 (Editing Rules)**.** *An editing rule (eR) φ defined on (R, R_m) is a pair $((X, X_m) \to (Y, Y_m), t_p)$, where*

- *$X \subset R$ and $X_m \subset R_m$ are two lists of distinct attributes, with the same length, i.e., $|X| = |X_m|$;*
- *Y is an attribute such that $Y \in R \setminus X$, and $Y_m \in R_m$;*
- *t_p is a pattern tuple over $X_p \subset R$ such that for each $A \in R, t_p[A] = a$. Here a is a constant drawn from the domain of A, i.e., $dom(A)$.*

Note that the pattern t_p is defined on schema R rather than R_m. We can say that a tuple t matches the pattern tuple t_p, denoted by $t[X_p] = t_p[X_p]$, if for each $A \in X_p$, we have $t[A] = t_p[A] = a$. We use $LHS(\varphi) = (X, X_m)$. In this study, for simplicity, we do not consider $\bar{a}$ in t_p that denotes $t_p[A] \neq a$ as stated in [18]. Obviously, $\bar{a}$ in t_p is equivalent to enumerating the rest of the values in $dom(A)$. Thus, omitting $\bar{a}$ does not affect the correctness of the editing rule.

Editing rules are used to update low-quality input data according to high-quality master data. Given an editing rule φ, a master tuple t_m and an input tuple t, t_m can update t by assigning $t_m[Y_m]$ to $t[Y]$, if $t[X_p] = t_p[X_p] \wedge t[X] = t_m[X_m]$.

B. Utility Measure

Since our target is cleaning data with rules, we hope that rules can cover as many tuples as possible and return a minimal set of candidate fixes, preferably with set size as one for a tuple, namely certain fixes [18]. Following [38], we also design utility measure w.r.t. support, certainty, and quality to evaluate the rule comprehensively.

1) Support: Unlike rules defined on one data source, editing rules require not only the data to be fixed but also the master data to satisfy predicates. Given a rule φ, its support $S(\varphi)$ denotes how many tuples in D can be applied by φ and D_m.

$$S(\varphi) = \sum_{t \in D} f_s(\varphi, t) \tag{1}$$

$f_s(\varphi, t)$ is a signature function that outputs 1, if a tuple $t_m \in D_m$ can update t according to φ. Otherwise, $f_s(\varphi, t) = 0$.

$$f_s(\varphi, t) = \begin{cases} 1, & \exists t_m \in D_m, t[X_p] = t_p[X_p] \wedge t[X] = t_m[X_m] \\ 0, & \text{otherwise} \end{cases}$$

2) Certainty: The certainty $C(\varphi)$ denotes how many candidate fixes are returned according to φ and master data D_m.

$$f_c(\varphi, t_i) = \begin{cases} \frac{\max_v \text{count}(v,\varphi)}{\sum_v \text{count}(v,\varphi)}, & \text{if Cand}(t_i, \varphi) \neq \emptyset \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

$\text{Cand}(t_i, \varphi) = \{t_m[Y_m] | t[X_p] = t_p[X_p] \wedge t[X] = t_m[X_m], t_m \in D_m\}$ are the candidate fixes returned by D_m according to φ. And v denotes a candidate fix of $\text{Cand}(t_i, \varphi)$, i.e., $v \in \text{Cand}(t_i, \varphi)$. $\text{count}(v, \varphi)$ is the count of the value v in $\text{Cand}(t_i, \varphi)$. We have $f_c(\varphi, t_i) \in [0, 1]$. If $f_c(\varphi, t_i) = 1$, it means that φ returns only one possible fix from D_m.

Based on $f_c(\varphi, t_i)$, we can define the certainty $C(\varphi)$ of the rule φ as follows:

$$C(\varphi) = \frac{1}{\sum_{t_i \in D} f_s(\varphi, t_i)} \sum_{t_i \in D} f_c(\varphi, t_i) * f_s(\varphi, t_i) \tag{3}$$

where D is the input relation.

$C(\varphi)$ describes the average certainty of repair results returned by the rule φ over its covered tuples. $C(\varphi)$ ranges from 0 to 1, and the closer it gets to 1, the more certain the rule is about the result returned, which is exactly what we want.

3) Quality: Given a labelled instance D_l with a relation schema R, same as D, we can further consider whether a discovered rule φ can repair a tuple $t_i \in D_l$ correctly.

$$f_q(\varphi, t_i) = f_s(\varphi, t_i) * \kappa(\varphi, t_i) \tag{4}$$

$\kappa(\varphi, t_i)$ identifies whether the most frequent candidate fix is the truth.

$$\kappa(\varphi, t_i) = \begin{cases} 1, & \text{if } \arg\max_v \text{count}(v, \varphi) = \widehat{t_i}[Y] \\ -1, & \text{otherwise} \end{cases}$$

where $\widehat{t_i}[Y]$ denotes the ground truth of $t_i \in D_l$ on the dependent attribute Y and $\text{count}(v, \varphi)$ returns the count of the value $v \in \text{Cand}(t_i, \varphi)$.

Then, we can define the quality of the rule φ as below:

$$Q(\varphi) = \frac{1}{\sum_{t_i \in D_l} f_s(\varphi, t_i)} \sum_{t_i \in D_l} f_q(\varphi, D_l) \tag{5}$$

Labelled data is optional. When there is no labelled data available, we can omit the Quality measure. If most of the input data are clean, we can also take the input data as labelled data to obtain an approximate Quality measure.

4) Utility: Given an eR φ, we define utility $\mathcal{U}(\varphi)$ as a comprehensive metric w.r.t. support, certainty and quality.

$$\mathcal{U}(\varphi) = (\log S(\varphi))^2 \times (C(\varphi) + Q(\varphi))$$

Figure 2 illustrates the relations of Utility $\mathcal{U}(\varphi)$ with Support $S(\varphi)$ and Certainty $C(\varphi)$. Quality $Q(\varphi)$ has similar effect as Certainty and thus is omitted. Intuitively, rules leading to certain fixes in high quality are preferred. Thereby, Utility increases linearly as Certainty (and Quality), in Figure 2(a). For Support, however, a rule with relatively large support, not necessarily covering most tuples, is already useful (if the corresponding quality is positive leading to correct fixes). To avoid the high marginal benefit [31] of support for dominating the utility measure, we use $\log^2$ to scale the support. Consequently, as illustrated in Figure 2(b), Utility is high already with a relatively large Support, while further increase of Support leads to marginal Utility increment.

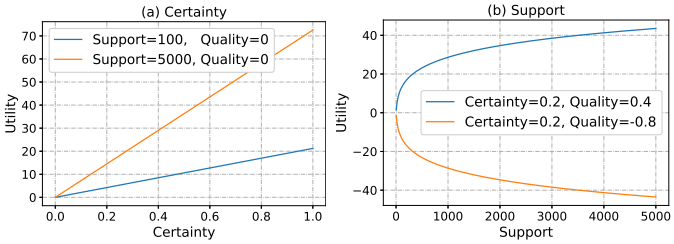

Fig. 2: Utility function

C. Editing Rule Discovery Problem

Due to the distinct schemes R_m and R, we need to first apply schema matching [28], [33] to find a match $M = \{A : \{A_m\} \mid A \in X \wedge A_m \in X_m\}$. $M(A)$ denotes the matched attribute A_m of A. If there does not exist any matched attribute of A, then $M(A) = \emptyset$. A schema matching algorithm can be written as $\mathcal{M}(A, A_m) \in \{0, 1\}$, where $\mathcal{M}(A, A_m) = 1$ indicates that the attributes A matches A_m, otherwise 0. In this work, we assume that R_m and R are already matched, i.e., given the match M, and focus on discovering editing rules.

Given an input relation D of a schema R, a master relation D_m of a schema R_m, a match M between schema R and R_m (and an optional D_l), the editing rule (eR) discovery algorithm aims to find eRs that can lead to certain fixes for D according to D_m. Obviously, we do not hope that the discovered rule set Σ includes any redundant editing rules. Thus, our target is to

mine a non-redundant set of eRs with high utility. Intuitively, we say there exists redundancy in a pair of rules φ_1 and φ_2, if the LHS in φ_1 is a subset of LHS in φ_2 and the constant pattern in φ_1 is also a subset of the pattern in φ_2. Of course, the opposite is also true. To avoid mining redundant rules, we first introduce the domination relationship of patterns.

Definition 2 (Pattern Domination)**.** *Given two patterns t_{p1} and t_{p2}, we say t_{p1} dominates t_{p2}, denoted by $t_{p1} \preccurlyeq t_{p2}$, if $X_{p1} \subsetneq X_{p2} \wedge t_{p1}[X_{p1}] = t_{p2}[X_{p1}]$, where X_{p1} and X_{p2} are sets of attributes specified in t_{p1} and t_{p2}, respectively.*

Based on the definition of pattern domination, we can define the domination relationship of editing rules as follows.

Definition 3 (Editing Rule Domination)**.** *Given two editing rules $\varphi_1 = ((X_1, X_{m1}) \rightarrow (Y, Y_m), t_{p1})$ and $\varphi_2 = ((X_2, X_{m2}) \rightarrow (Y, Y_m), t_{p2})$, we say φ_1 dominates φ_2, denoted by $\varphi_1 \preccurlyeq \varphi_2$, if $X_1 \subset X_2 \wedge X_{m1} \subset X_{m2} \wedge t_{p1} \preccurlyeq t_{p2}$.*

Based on the editing rule dominate definition, the following lemma describes the relationships of supports between two editing rules when one is dominated by the other.

Lemma 1. *Given two editing rules $\varphi_1 = ((X_1, X_{m1}) \rightarrow (Y, Y_m), t_{p1})$ and $\varphi_2 = ((X_2, X_{m2}) \rightarrow (Y, Y_m), t_{p2})$, if $\varphi_1 \preccurlyeq \varphi_2$, then $S(\varphi_1) \geq S(\varphi_2)$.*

Given a set of editing rules Σ, we say it is non-redundant if any eR $\varphi_1 \in \Sigma$ is not dominated by another eR $\varphi_2 \in \Sigma$.

Definition 4 (Non-redundant Editing Rules Set)**.** *Given a set of editing rules Σ, if $\nexists \varphi_1, \varphi_2 \in \Sigma, \varphi_1 \preccurlyeq \varphi_2$, then Σ is a non-redundant set of editing rules.*

In practice, we find that even if only non-redundant eRs are returned, the number of rules is still large. An overly large rule set not only makes it difficult for users to focus on the valuable rules (i.e., with high utility), but also makes it more time-consuming to apply. Thus, we propose to only discover top-k editing rules with the highest utility. We give a formal definition of the Editing Rule Discovery problem as follows.

Problem 1 (Editing Rule Discovery)**.** *Given input data D, master data D_m, (optional D_l), a match M between schema (R, R_m), a target attribute pair (Y, Y_m), rule utility function $\mathcal{U}$ and a constant K, the Editing Rule Discovery problem is to find a non-redundant set of K editing rules Σ whose utility measures are maximized.*

D. Enumeration-based Editing Rules Discovery

Following the classical rule mining approaches such as CTANE [17], we propose EnuMiner that also starts from singleton attribute sets (X, X_m) and proceeds to add more attributes pairs to $LHS(\varphi)$ or value conditions to pattern t_p. Similarly, EnuMiner also adopts a variety of pruning strategies, such as pruning according to support, to speed up the mining process. Since the enumeration space size $N_{\text{enum}} = 2^{|M|} * \prod_{A \in R \setminus Y} (|dom(A)| + 1)$ is exponentially large in the number of attributes, traversing the entire space is too expensive when the data set is large, even with the help of pruning strategies. Thereby, EnuMiner fails to mine data with lots of attributes and large domains efficiently (see Section V). Implementation details of EnuMiner can be found in [4].

III. Reinforcement Learning for Rule Discovery

In this section, we first justify our choice of reinforcement learning (RL) to avoid enumeration in editing rule discovery. An overview of RLMiner as an RL system is first presented, with implementation details in the following Section IV.

A. Why Reinforcement Learning?

In many cases, an exact set of non-redundant editing rules that consists of top-k eRs sorted by utility measure is unnecessary. The users may prefer an algorithm that can discover an approximate set of editing rules that still achieves similar cleaning performance as the rule set returned by EnuMiner, but only costs 10% time of EnuMiner. To achieve this, we need smarter search and pruning strategies.

In recent years, reinforcement learning has achieved superhuman performance over many complex tasks such as Go [34]–[36] and Database Tuning and Optimization [22], [23], [43], [46]. This demonstrates the potential of reinforcement learning to solve problems with large domains and decision spaces.

If we see the rule mining process as a decision process, reinforcement learning can also be applied to train an agent to intelligently discover and prune rules, thus avoiding a large number of enumerations and greatly reducing the time cost. Following the idea, we propose to model the rule mining task as a Markov Decision Process, that is, each time we specialize an editing rule φ by adding a new pair of attributes (A, A_m) to $LHS(\varphi)$ or an attribute-value pair (A, v) to t_p.

Definition 5 (Editing Rule Discovery Markov Decision Process)**.** *Editing Rule Discovery Markov Decision Process is defined by a four-tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$.*

$\mathcal{S}$ is the state space, where each $s \in \mathcal{S}$ corresponds to an editing rule φ.

$\mathcal{A}$ is a set of actions and each $a \in A$ denotes adding a pair of attributes (A, A_m) to $LHS(\varphi)$ or adding a constant v on attribute A to pattern t_p in φ.

$\mathcal{T} : \mathcal{S} \times \mathcal{A} \mapsto \mathcal{S}$ is a transition function linking state-action pairs to new states, i.e., generating a new editing rule φ' from φ by adding (A, A_m) to $LHS(\varphi)$ or (A, v) to pattern t_p of φ.

$\mathcal{R} : \mathcal{S} \mapsto \mathbb{R}$ is a reward function mapping states to a reward value that is calculated according to rule measures.

B. Rule Discovery by Building a Rule Tree

Following the rule discovery of EnuMiner and Definition 5, we propose to model the process of rule discovery as tree growth as illustrated in Figure 3, where each node in the tree is a rule. A child node is generated by refining its parent node. As shown in Figure 3, given a parent node with a rule $\varphi = ((X, X_m) \rightarrow (Y, Y_m), t_p)$, there are two operations for growing a child from it.

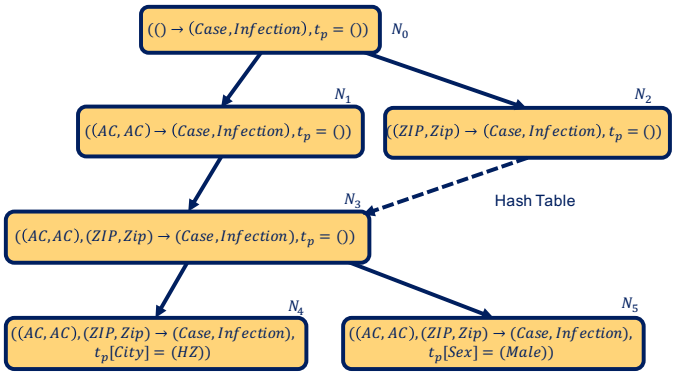

Fig. 3: Example of a rule tree in discovery

(1) Adding a pair of attributes $(A, A_m), A_m \in M(A)$ to the determinate (LHS) attributes, i.e., $\varphi' = ((X \cup \{A\}, X_m \cup \{A_m\}) \rightarrow (Y, Y_m), (t_p))$.

(2) Adding a new condition in pattern t_p by specifying a value $v \in dom(A)$ on attribute $A \in R \setminus (X_p \cup Y)$, i.e., $\varphi' = ((X, X_m) \rightarrow (Y, Y_m), t_p \cup \{(A, v)\})$.

To avoid generating the same rules multiple times, we maintain a hash table to record the considered rules. Before generating a new rule, we first check the possible operation spaces according to the current rule and whether the generated rule has already been considered.

Example 2 (tree growth for rule discovery)**.** *For the node N_1 in Figure 3, its rule is $\varphi_1 = ((AC,AC) \rightarrow (Case, Infection), t_p = ())$ where the determinate attribute set $LHS(\varphi_1)$ is $\{(AC,AC)\}$, the dependent attribute pair is (Case, Infection), and the pattern is empty. If we add an attribute pair (ZIP,Zip) into the rule φ_1, then we will obtain a child node N_3 of N_1. The rule in N_3 is $\varphi_3 = ((ZIP,Zip),(AC,AC) \rightarrow (Case, Infection), t_p = ())$. The maintained hash table avoids adding (AC,AC) into φ_2 of N_2, which will generate a considered rule φ_3. Then, we can further refine φ_3. For example, if we add (City, "HZ") or (Sex, "Male") to φ_3, another two editing rules φ_4 and φ_5 are generated and the corresponding nodes are grown from N_3.*

C. An Overview of RL for RLMiner

Following Definition 5, we propose a reinforcement learning based rule mining algorithm, namely RLMiner. RLMiner learns to build a rule tree to discover editing rules. Figure 4 illustrates the overview of RLMiner. Next, we briefly introduce each module in RLMiner.

1) Environment: It consists of the input data D, the master data D_m, (the optional D_l) and the growing rule tree.

2) Agent: Agent is a rule miner system. It first receives the reward of the last action and the state from the environment, and then updates the policy to generate the next rule according to the current state.

3) State: Before we generate a new rule, we should observe the environment first, including what rules are generated thus far and how many tuples are covered by those rules. Intuitively, the state in RLMiner should represent the whole rule tree. However,encoding a growing tree into a fixed-length vector is challenging. If we assign a unique label to each rule and represent the rule tree as a multi-hot vector, the vector length would be too large. Although recent works of graph representation [27], [45] can embed a graph into a fixed-length vector, they are not suitable to encode a growing tree whose length increases at each time step. Recall that our rule discovery strategy generates new rules by refining a rule contained in a node of the tree so that we can focus only on one node when generating a new rule. Thus, we propose to encode a rule (node) rather than the whole rule tree. The state encoding should identify $LHS(\varphi)$ and t_p of the corresponding rule. We will introduce the details of the state encoding in Section IV-A.

4) Action: Action a_t denotes the behavior of the agent at time t. As stated in Section III-B, we can generate a new rule by adding a pair of matched attributes to $LHS(\varphi)$ or specifying an attribute-value pair (A, v) in the pattern t_p. Intuitively, the attribute pair selection action is discrete. For text attributes, the attribute value selection action is also discrete. For numerical attributes, we can treat them as discrete attributes or discrete them into ranges. Then, the action space is modeled as a discrete space. More details and discussions can be found in Section IV-B.

5) Policy: The policy is a map that models agent's action selection according to the state s. Since our state and action are discrete, it is tractable to estimate the "value" for each selection, where "value" here evaluates how good a state is [41]. For this reason, compared to policy-based RL algorithms designed for continuous action space, Q-learning methods are more suitable for our scenario. Q-learning methods first estimate the action-value $Q^\pi(s, a)$. And the policy greedily chooses the action a with the maximum $Q^\pi(s, a)$ as the next action for s. Specifically, a deep neural network is used to estimate $Q^\pi(s, a)$ in DQN [26] and its variants. The input of the value network is the encoding vector of the current rule and the output is the estimated values for each action, i.e., the attribute pair or attribute value pair with higher value is going to be added into the current rule.

6) Reward: Reward is a scalar r_t that represents the difference between the performance at time t and time $(t-1)$. The design of reward is the soul of RL which directly affects the performance of the model. Benefiting from domain knowledge and rule measure defined in Section II-B, we design a reward function that is in line with rule mining and enables RLMiner to achieve quick convergence and good performance.

IV. Implementation Details

A. State Representation

In Section III-C, the state is the encoding of a rule (node). We represent the state by a one-hot vector s, consisting of two

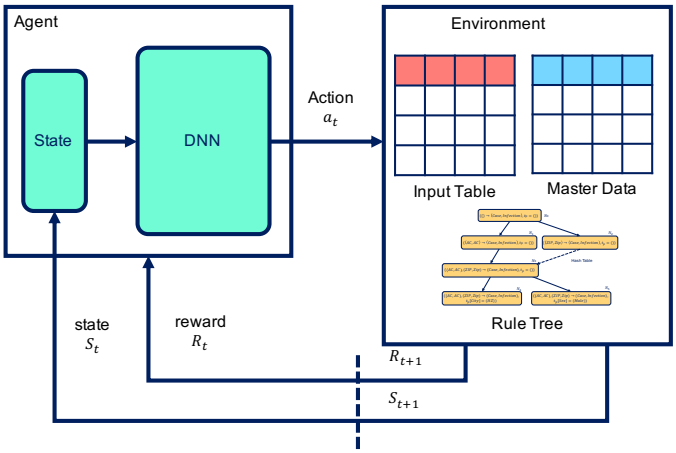

Fig. 4: RLMiner as an RL system

parts: LHS attributes state s_l and pattern specification state s_p.

$$s = [s_l; s_p] \tag{6}$$

$$dim(s_l) = \sum_{A \in R \setminus Y} |M(A)| \tag{7}$$

$$dim(s_p) = \sum_{x_i \in X_C} N_{split} + \sum_{x_i \in X_D} |dom(x_i)| \tag{8}$$

where X_D is the set of discrete attributes and X_C is the set of continuous attributes. The LHS attributes state s_l represents what attribute pairs in LHS. Each dimension of s_l denotes a pair of matched attributes (X, X_m). The pattern specification state s_p specifies what attributes and values are specified in the pattern t_p of φ. Each dimension of s_p corresponds to a specific value from the domain of $A \in R \setminus Y$.

In many existing rule discovery algorithms [17], continuous attributes are also seen as discrete attributes. If so, the domain of continuous attributes would be too large to encode. Compared to discrete values, continuous values are comparable, since continuous attributes contain ordering relationships. Moreover, it is unnecessary to specify each continuous value in the pattern, which results in low support for the generated rule. In most cases, a range of continuous values satisfy the same rule. Thus, we propose to split continuous attributes into multiple ranges, where N_{split} is the number of ranges.

To encode discrete attributes in φ, each specific value in t_p can be encoded by a $|dom(x_i)|$-dimensional one-hot vector. However, this encoding approach will cause the encoding dimension to be very large, if the domain size is large. When we learn a DNN-based value network, a too large dimension of state representation leads to a huge number of parameters in the value network, making the network difficult to converge. A simple and straightforward idea is to reduce the domain size by using common prefix. In this way, we can reduce the encoding dimension from $dom(x_i)$ to $K \ll dom(x_i)$.

B. Action Space

As stated in Section III-B, we add a new attribute pair (A, A_m) to $LHS(\varphi)$ or attribute value (A, v) to the pattern t_p of φ. Note that we cannot change the existed attribute pairs in $LHS(\varphi)$ or attribute value pairs in t_p. Therefore, the corresponding actions of existed $LHS(\varphi)$ and t_p should not be selected. Consider a node with rule $\varphi = ((X, X_m) \rightarrow (Y, Y_m), t_p)$, we can only consider the attribute pairs in $\{(A, M(A))|A \in R \setminus X\}$ as the candidate determinate attribute pair for $LHS(\varphi)$, and the attribute value pair $\{(A, v)|A \in R \setminus X_p, v \in dom(A)\}$ as the candidate condition for pattern t_p.

To avoid enumerating all combinations, we add a stop operation for pruning. If the policy returns the "stop" action, it indicates that there does not exist any new attribute pair (A, A_m) or attribute value pair (A, v) which can be added into the current rule to obtain a large reward. Then, RLMiner will stop refining the rule from the current node and move to the next node. In this way, RLMiner prunes the non-explored search space of the current node. In each step, we will select one operation for the current node with rule $\varphi = ((X, X_m) \rightarrow (Y, Y_m)), t_p)$ from the followings:

(1) Adding an attribute pair from $\{(A, M(A))|A \in R \setminus X\}$ to $LHS(\varphi)$;

(2) Specifying one more condition from $\{(A, v)|A \in R \setminus X_p, v \in dom(A)\}$ in t_p;

(3) Stopping refinement and moving to the next node.

Following the state encoding, we define the action space with a vector a consisting of three parts a_l, a_p and a_{stop}

$$a = [a_l; a_p; a_{stop}] \tag{9}$$

$$dim(a_l) = \sum_{A \in R \setminus Y} |M(A)| \tag{10}$$

$$dim(a_p) = \sum_{x_i \in X_C} N_{split} + \sum_{x_i \in X_D} |dom(x_i)| \tag{11}$$

$$dim(a_{stop}) = 1 \tag{12}$$

a_l denotes which attribute pair should be added into $LHS(\varphi)$. a_p represents which attribute value pair is specified as a new condition in t_p. a_{stop} is the "stop" action that is encoded as a one-dimensional vector. We can find that a_l and a_p are actually corresponding to state encoding s_l and s_p.

C. Value Network

Recall that our action is to refine a given rule via the operations (1) and (2), and never deletes or replaces the existed attributes pairs in $LHS(\varphi)$ or attribute value pairs in pattern t_p. In addition, following EnuMiner, we should also not consider any action that will generate an existing rule. To achieve it, we propose a rule mask mechanism to restrict the action space according to the current state and the rule tree.

The mask is initialized as a vector filled with $m = \vec{1}$. Each dimension corresponds to an action. By setting the corresponding m to 0, the policy will not consider those actions as the next action. As illustrated in Algorithm 1, we can divide the mask into local mask (lines 3-11) and global mask (lines 12-17). The local mask is designed to generate the mask according to the state s of the current rule $\varphi = ((X, X_m) \rightarrow (Y, Y_m), t_p)$. For each attribute pair $(A, A_m) \in LHS(\varphi)$, the actions corresponding to (A, A_m) are masked (lines 6-8). For

Algorithm 1 mask

Input: State Encoding s; Discovered Rule Set Σ;
Output: Mask vector m;

1: $m = \vec{1}, |m| = dim(s)+1$ // Never mask the last dimension corresponding to stop action
2: candIndex $= \{s[ix] = 1 | ix \in [0, |m|)\}$
3: // Local mask according to s
4: **for** $ix \in$ candIndex **do**
5: res = invertedIndex$[ix]$
6: **if** res is (A, A_m) **then** // Mask attribute action space
7: **for** $(A, A'_m), A'_m \in M(A) \setminus \{A_m\}$ **do**
8: $m[\text{indexOf}(A, A'_m)] = 0$
9: **if** res is (A, v) **then** // Mask pattern action space
10: **for** $v' \in dom(A) \setminus \{v\}$ **do**
11: $m[\text{indexOf}(A, v')] = 0$
12: // Global mask: avoid repeated enumerations
13: **for** $\varphi \in \Sigma$ **do**
14: $s_0 = \text{decode}(\varphi)$
15: diff $= s_0 \oplus s$
16: **if** $sum(\text{diff}) = 1$ and $sum(s_0) - sum(s) = 1$ **then**
17: $m[i] = 0$, where diff$[i] = 1$
18: **return** m

each attribute value pair $(A, v) \in t_p$, the actions corresponding to (A, v') for any $v' \in dom(A)$ are also masked (lines 9-11). The global mask is proposed to avoid discovering rules repeatedly. We choose to mask any action that will transform $s \mapsto s_0$ and s_0 already exists (lines 12-17).

The rule mask mechanism is applied to the original logits outputted by the value network. The logits corresponding to the actions not allowed will be assigned a small negative value so that the greedy policy will not select them later.

The masked value network $\pi(s_t)$ is shown in Figure 5. The input is the state encoding s_t in Section IV-A. Then, a deep neural network extracts features from s_t and obtains the corresponding feature vector $z_t \in R^h$, where h is the dimension of the feature vector. A linear layer $W \in R^{h \times d}$ calculates the logits $q_t \in R^d$ (Q-values) according to z_t, where $q_t[i]$ denotes the q-value of the action $a_t[i]$. Then a rule mask layer is applied to restrict the action space.

$$q' = q \odot m - inf * (1 - m) \tag{13}$$

Example 3 (masked value network)**.** *Figure 5 shows input data with schema $R = (A_1, A_2, Y)$, master data with schema $R_m = (A_{m1}, A_{m2}, Y_m)$ and a match $M = \{A_1 : \{A_{m_1}\}, A_2 : \{A_{m_2}\}\}$. According to the state encoding in Section IV-A, the input state $s_t = [1, 0, 1, 0, 0, 0, 0, 0]$ denotes the rule $\varphi = ((A_1, A_{m2}) \rightarrow (Y, Y_m), t_p[A_1] = (v_1))$. Then, the DNN transforms s_t into z_t and the MLP calculates its logits q_t.*

First, we consider the local mask. Since the attribute pair (A_1, A_{m1}) already exists in $LHS(\varphi)$, we should not consider (A_1, A_{m1}) as our next action. Thus, the action logits $q_t[0]$ are masked. For the pattern $t_p[A_1] = (v_1)$, A_1 is included in X_p so that all actions corresponding to values in $dom(A_1)$ are masked, i.e., $q_t[2:5]$ are masked.

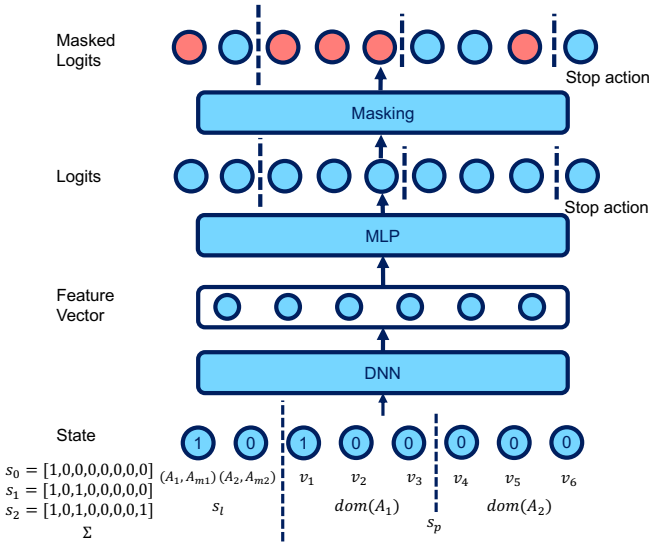

Fig. 5: Masked value network

Then, we consider the global mask. According to Algorithm 1, the action corresponding to (A_2, v_6) should be masked, otherwise the generated rule will be the same as s_2 that already exists in the discovered rule set Σ. Then, we can obtain the masked logits where allowed action is in blue. It means that RLMiner can add (A_2, A_{m2}) into $LHS(\varphi)$ or add (A_2, v_4) or (A_2, v_5) into the pattern or stop refining the current rule and move to the next node in the rule tree.

D. Reward Function

At time t, the agent will generate a new rule by refining the current rule. In order to help the agent learn a good policy, we need to design a reward function to evaluate how good the discovered rule is. As discussed in Section II-B, utility is a comprehensive measure for rule. We propose to calculate the reward based on utility as shown in Algorithm 2.

For "stop" action, we assign it a constant θ reward that should be a small positive value such as 0.01 used in our study (lines 1-2). The reason we suggest θ as a positive value is to encourage the agent to prune automatically and move to the next rule when it cannot generate a good new rule. But if we assign θ a big value, the agent may be drunk on the "easy money", i.e., getting a large sum of rewards by always choosing the "stop" action, and fail to discover valuable rules.

Note that the reward here is based on utility $\mathcal{U}$ and the utility is calculated by querying input data D and master data D_m with rule φ. Therefore, we maintain a hash map $\mathcal{R}_\Sigma$ to store the reward for each discovered rule. In this way, we avoid executing the same queries when discovering the same editing rules in different episodes during RLMiner training, and thus greatly reducing the time cost.

Compared to the nodes in the deeper level, the nodes in the lower level contain more general rules that may lead to

Algorithm 2 env.CalReward

Input: Current State s_t; Current Node N; Action a_t; Rule Reward HashMap R_Σ;
Output: Reward r_t

1: **if** a_t is "stop" **then**
2: $r_t = \theta$ // $\theta = 0.01$ by default.
3: **else**
4: // Calculate the rule reward
5: $s_{t+1} = \text{transform}(s, a)$
6: **if** $s_{t+1} \in \mathcal{R}_\Sigma$.keys **then**
7: $r_t = \mathcal{R}_\Sigma.\text{get}(s_{t+1})$ // Reuse reward
8: **else**
9: $\varphi = \text{decode}(s_{t+1})$
10: **if** $S(\varphi) \geq \eta$ **then**
11: $r_t = \mathcal{U}(\varphi)$
12: **else**
13: $r_t = -0.01$ // By default
14: $R_\Sigma.\text{put}(s_{t+1}, r_t)$ // Store the reward for each rule
15: **if** N.children $= \emptyset$ and $S(\varphi) \geq \eta_s$ **then**
16: $r_t = r_t + (r_t - R_\Sigma.get(s_t))$
17: **return** r_t

many incorrect fixes and thus be of low utility. For example, a single "First Name" attribute cannot accurately infer a person's information, but if the "Last Name" is added, the utility of the rule can be greatly improved. It causes the utility of the rule in the lower-level node may exceed the rule in the upper-level node. To encourage RLMiner to explore nodes with low reward but whose child contains rules with high utility, when we generate a valid new rule from a node with no children, the reward for that action will be added with the utility difference between the current and the new rule (lines 15-16).

On the other hand, if the node can already return correct fixes and achieve high utility, the reward function will penalize RLMiner for choosing to grow new nodes from the current node. Specifically, as shown in Lines 15-16 in Algorithm 2, when we generate a valid new rule with $S(\varphi) \geq \eta_s$ from a node with no children, the reward for the action is the sum of two parts: (a) the original reward r_t of the new rule and (b) the reward difference between the new and current rule, i.e., $r_t - R_\Sigma.get(s_t)$. On one hand, if $r_t > R_\Sigma.get(s_t)$, it encourages RLMiner to generate child rule with high utility from the current rule with low utility, by assigning it extra rewards. On the other hand, for $r_t < R_\Sigma.get(s_t)$, it penalizes RLMiner for continuing to generate over specific rules from highly-utility rules. In addition, if the generated rule cannot satisfy the support threshold, i.e., $S(\varphi) < \eta_s$, the reward function will return a constant value -0.01 (Lines 10-13), also preventing RLMiner from generating too specific rules.

E. Overall Algorithm

The overall algorithm of RLMiner is as shown in Algorithm 3. RLMiner starts with the root node of the rule tree, s^*. The goal of RLMiner is to learn an optimized value function

Algorithm 3 RLMiner

Input: Input Data D; Master Data D_m; To-Repair Attribute Y; Maximum number of transitions N; Rule Number K; Labelled Data D_l (optional);
Output: Value Network π

1: Init environment env=BuildEnv($\bigcup dom(A), D_m, D, D_l$)
2: Init rule reward hashmap R_Σ = HashMap()
3: Init Replay Memory replay = [] and Value Network π
4: **while** $n < N$ **do**
5: Reset State $s = \text{Reset}(s*)$
6: Generate the mask $m = \text{mask}(s, env)$
7: done=False
8: **while** not done **do**
9: Calculate Q values $q = \pi(s)$
10: Generate the mask $m = \text{mask}(s, env)$
11: Mask Q values $q' = q \odot m - inf * (1 - m)$
12: Select action $a = \arg\max_{a \in A} q'[a]$
13: Get next state $s = \text{env.GrowTree}(s, a)$ //Alg. 4
14: **if** s is None or |env.tree.leaves| $\geq K$ **then**
15: done = True
16: Calculate reward r = env.CalReward(s, a, R_Σ) //Alg. 2
17: Collect transition replay.add$((s_t, a_t, r_t, s_{t+1}))$
18: Optimize Value Network via RL algorithm by sampling from replay memory
19: $n = n + 1$
20: **return** π

π that can precisely estimate the value of each action for the state. With π, the greedy policy will refine the current rule by selecting the action with maximum value and discover rules by growing a rule tree in this way.

The details of tree growing are illustrated in Algorithm 4. Specifically, if the selected action corresponds to "stop", RLMiner will stop growing from the current node and move to the next node in lever-order, i.e., bread-first-search (lines 1-3). Otherwise, we will refine the current rule according to the action and grow a new node from the current node (lines 5-17).

As EnuMiner, RLMiner also employs several optimizations for acceleration except reusing utility in Algorithm 2. First, RLMiner also sets a support threshold η_s and only rules satisfying $S(\varphi) \geq \eta_s$ will be added as leaf nodes and refined later (lines 14-17). And if the rule only returns one certain fix, we do not need to further refine it (line 14). In addition, RLMiner stores the index of cover(φ) and conducts subspace search on it like EnuMiner (lines 9-10). After training, the rules in leaf nodes are returned as the discovered editing rules. Then we will reset the state and the rule tree.

V. Experimental Study

A. Experimental Settings

The experiments run on a machine with 2.1GHz CPU, Nvidia 1080ti GPU and 128GB memory. The source code and data have been made available online [3].

Algorithm 4 env.GrowTree

Input: Current State s_t; Action a_t;
Output: Next State s_{t+1}

if a_t is "stop" **then**
  node = env.getNextNode() // Level-order Walk
  $s_{t+1} =$ node.state
else
  $s_{t+1} = \text{copy}(s_t)$
  $s_{t+1}[a_t] = 1$
  $\varphi = \text{decode}(s_{t+1}) = ((X, X_m) \rightarrow (Y, Y_m), t_p)$
  $\text{node}_c =$ env.tree.currentNode
  $D_c = \text{node}_c.\text{getCover}(), D_c \subset D_m$ //
  $D_p = \{t_i | t_i[X] = t_p[X], t_i \in D_c\}$ // subspace search
  $\text{node} = \text{Node}(s_{t+1}, D_p, \text{parent} = \text{node}_c)$
  $\text{node.cover} = D_p$
  $\text{node}_c.\text{children.add}(\text{node})$
  if $C(\varphi) < 1$ and $S(\varphi) \geq \eta_s$ **then**
    env.queue.add(node)
  else
    continue // stop refinement
return s_{t+1}

1) Dataset: We employ several public and real datasets for evaluation. Table I provides statistics on input/master schema size (the number of attributes), input/master data size (the number of tuples). In the experiments, we do not use any labelled data D_l but take input data D to obtain an approximate Quality measure as discussed in Section II-B3.

For the datasets without labelled errors, Nursery, Adult and Covid, we assume the original data clean. Thereby, the input data and master data are sampled separately from the original dataset. To simulate dirty input, following the same line of error generation [10], we inject errors into the input data.

(1) Adult [1] is a publicly available dataset in size 48842. We randomly sample 40000 tuples as input data and 5000 tuples as master data. "Income" is taken as the Y-attribute. The default support threshold η_s of dataset Adult is 1000.

(2) Covid-19 [7] is a public dataset that records COVID-19 information in South Korea. We keep the tuples whose values in the attribute "state" are "released" as master data and randomly sample 2500 tuples as input data. "infection_case" is taken as the Y-attribute. The default support threshold η_s of dataset Covid-19 is 100.

(3) Nursery [2] is a public dataset whose size is 12960. Similarly, we randomly sample 10000 tuples as input data and 2980 tuples as master data. "finance" is taken as the Y-attribute. The default support threshold η_s of Nursery is 1000.

(4) Location [6] contains location information of 2,559 coffee shops, with nine attributes. we manually label the errors and the corresponding truths. It has 14.7% missing values in the Postcode attribute, i.e., already dirty. We download the clean postcode data from the government website [8] as master data, including the postcode information of 3,430 counties. There are five attributes in the master data, among which "City", "County", "AreaCode" and "Postcode" are matched with the Location dataset. The ground truths of the missing values are then manually labelled referring to the master data. In addition to missing values, we also find 19.6% real-world errors existing in the raw data.

TABLE I: Dataset summary

Dataset	# A	# A_m	# Input	# Master
Adult	10	9	40000	5000
Covid-19	7	8	2500	1824
Nursery	9	9	10000	2980
Location	9	5	2559	3430

2) Criteria: We evaluate the correctness of all cell predictions returned by applying the discovered editing rules. As illustrated in Table I, all the to-repair attributes contain multiple classes. Thus, the weighted precision/recall/f-measure are considered as the evaluation metric.

$$\text{Precision}_w = \frac{1}{\sum_{l \in L} |\hat{y}_l|} \sum_{l \in L} |\hat{y}_l| \text{Precision}(y_l, \hat{y}_l)$$

$$\text{Recall}_w = \frac{1}{\sum_{l \in L} |\hat{y}_l|} \sum_{l \in L} |\hat{y}_l| \text{Recall}(y_l, \hat{y}_l)$$

$$\text{F-Measure}_w = \frac{1}{\sum_{l \in L} |\hat{y}_l|} \sum_{l \in L} |\hat{y}_l| \text{F-Measure}(y_l, \hat{y}_l)$$

In addition to the proposed EnuMiner and RLMiner, we adapt the CFD discovery algorithm for comparison. An intuitive approach is to mine CFDs on master data and convert them to eRs. The CFDs whose attributes in LHS and the pattern are matched with the attributes in input data can be transformed. The CFD discovery algorithm CTANE [5] is implemented in C++. Both EnuMiner and RLMiner are implemented in python and the number of rules K is set to 50. Since CTANE and our proposed methods are implemented with different programming languages and CTANE only discovers CFDs on master data of smaller size, it is unfair to compare the time cost. We thus focus on time cost comparison between EnuMiner and RLMiner. Note that for RLMiner, the time cost is composed of both training time and rule discovery time. All the experiments repeat five times, and the mean and standard deviation of them are reported as the final results.

B. Discovery Results

1) Rule Statistics: We report the mean, standard deviation (std), max and min on the number of left hand side (LHS) attributes and patterns in the discovered rules. For instance, the eR $\varphi_0 = ((A_1, A_{m1}), (A_2, A_{m2}) \rightarrow (Y, Y_m), t_p[A_3] = v)$ has LHS length 2 and pattern length 1. As shown, RLMiner does not return too specific or too general rules, with lengths similar to those discovered by RLMiner. It is worth noting that in the Nursery dataset, the average length of EnuMiner reaches 5.62, i.e., too specific. The reason is that the domain size of Nursery is small. Thereby, more attributes are employed to determine the fixes. Nevertheless, the small domain size leads to higher support, i.e., avoiding the negative reward -0.01 in

TABLE II: Statistics on rule length

Dataset	Method	# LHS (mean+std)	# LHS (max/min)	# Pattern (mean+std)	# Pattern (max/min)
Nursery	CTANE	1.55 ± 0.73	4 / 1	3.37 ± 0.76	4 / 1
	EnuMiner	5.62 ± 0.57	7 / 5	0.00 ± 0.00	0 / 0
	RLMiner	1.18 ± 0.38	2 / 1	0.72 ± 0.45	1 / 0
Adult	CTANE	1.76 ± 0.65	3 / 1	1.33 ± 0.47	2 / 1
	EnuMiner	1.13 ± 0.35	3 / 1	0.88 ± 0.33	1 / 0
	RLMiner	1.24 ± 0.49	3 / 1	1.17 ± 0.55	2 / 0
Covid	CTANE	1.38 ± 0.51	3 / 1	1.54 ± 0.55	3 / 1
	EnuMiner	1.54 ± 0.78	4 / 1	0.81 ± 0.60	3 / 0
	RLMiner	2.04 ± 0.81	4 / 1	0.90 ± 0.71	2 / 0
Location	CTANE	1.00 ± 0.00	1 / 1	1.00 ± 0.00	1 / 1
	EnuMiner	1.06 ± 0.24	2 / 1	0.94 ± 0.24	1 / 0
	RLMiner	1.03 ± 0.16	2 / 1	1.11 ± 0.40	2 / 0

TABLE III: Repair results of RLMiner compared to baselines

Dataset	Method	Precision	Recall	F1
Nursery	CTANE	0.51 ± 0.01	0.23 ± 0.00	0.32 ± 0.01
	EnuMiner	0.52 ± 0.00	0.52 ± 0.00	0.52 ± 0.00
	RLMiner	0.52 ± 0.01	0.52 ± 0.00	0.51 ± 0.01
Adult	CTANE	0.77 ± 0.00	0.19 ± 0.00	0.31 ± 0.00
	EnuMiner	0.78 ± 0.01	0.77 ± 0.00	0.67 ± 0.00
	RLMiner	0.75 ± 0.01	0.76 ± 0.01	0.70 ± 0.03
Covid	CTANE	0.58 ± 0.02	0.17 ± 0.00	0.26 ± 0.01
	EnuMiner	0.62 ± 0.01	0.63 ± 0.01	0.61 ± 0.01
	RLMiner	0.73 ± 0.10	0.59 ± 0.06	0.63 ± 0.03
Location	CTANE	0.70 ± 0.00	0.53 ± 0.00	0.59 ± 0.00
	EnuMiner	0.86 ± 0.00	0.86 ± 0.00	0.85 ± 0.00
	RLMiner	0.90 ± 0.03	0.77 ± 0.07	0.81 ± 0.03

RLMiner. Consequently, it avoids overfitting, and leads to a smaller LHS length.

Moreover, we illustrate some examples of the discovered rules in the datasets Covid-19 and Location.

$$\varphi_1 = ((\text{city, city}), (\text{confirmed\_date, confirmed\_date}) \rightarrow (\text{infection\_case, infection\_case}), t_p[\text{state}] = \text{release})$$

$$\varphi_2 = ((\text{area\_code, area\_code}), (\text{County, County}) \rightarrow (\text{Postcode, Postcode}), ())$$

Similar to the eR in Example 1, φ_1 states that we can infer the "infection_case" for the person, whose state is "release", according to "City" and "confirmed_date" from the master data. Referring to φ_2, "Postcode" can be inferred by "County" and "area_code" in master data. The lengths of the left hand side attributes in both rules are 2, whereas φ_1 has an extra pattern specified on one attribute.

2) Rule Effectiveness: To evaluate the effectiveness of the mined rules, we utilize the mined rules to infer the values of the Y-attribute and compare the inferred values with the original value. Given an editing rule $\varphi \in \Sigma$ and a tuple t, we can obtain the certainty score $\sigma_{v,\varphi} = \frac{\text{count}(v,\varphi)}{\sum_{v'} \text{count}(v',\varphi)}$ for each candidate fix v returned by applying φ. The candidate fix with the maximum sum of certainty scores is considered as the fix $\arg\max_v \sum_\varphi \sigma_{v,\varphi}$.

Table III demonstrates the evaluation results over the four datasets. EnuMiner and RLMiner achieve similar results on all three datasets, which proves that RLMiner can mine high-quality rules without iterative the entire enumeration space. The CFD discovery algorithm CTANE [17] has low recall over the four datasets. The results are not surprising, since CFDs discovered from master data could only specify conditions on master data attributes and values. More eRs with conditions on input data attributes and values are ignored. Indeed, the distributions of input and master data may differ. A rule that has low support in master data (and thus is pruned) may have a great value on input data. On the other hand, since the master data may not be extensive, the rules discovered from the master data could be underfitting.

The precision is about 0.7 or even lower in the datasets Nursery, Adult and Covid. The reason is that for these three datasets, we do not have the labels of errors embedded in the raw data. Thereby, we manually inject additional errors in the data and evaluate the repairs by comparing with the original data. It is worth noting that the original data may also be dirty with unknown errors and truths. To illustrate this, we also report the results of noise rate 0, i.e., no additional errors injected, in Figure 6. As shown, even without injecting additional errors, there are still some data repaired with comparable F-Measure.

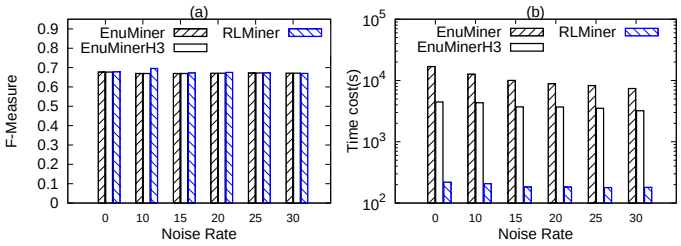

Fig. 6: Varying noise rate over Adult

Nevertheless, to conduct a more reliable evaluation, we consider another dataset, Location, by manually labelling the errors embedded in the raw data and the corresponding truths, as introduced in Section V-A1. Table III shows that the relationships of different methods' results on Location with real errors are generally consistent with the randomly sampled datasets. Moreover, the accuracy on Location is higher than the other datasets with manually added errors. The reason is that these raw datasets may originally contain errors but not labelled, and thus leading to lower precision and recall.

C. Varying Data Features

1) Varying Noise Rate: We test the algorithms at different noise rates to evaluate the robustness of algorithms against noise. As shown in Figure 6(a), both EnuMiner and RLMiner are stable under various noise rates.

RLMiner shows order-of-magnitude improvement in time cost but is more sensitive to noise. The reason is that RLMiner does not traverse the entire search space for efficiency. It is not surprising that EnuMiner leads to more stable results by costly enumerating all the possible rules. Nevertheless, the difference in accuracy is not significant, especially over the large dataset Adult, as illustrated in Figure 6, i.e., a worthwhile trade-off.

2) Varying Duplicate Rate: The duplicate rate $d\%$ means how many input data correspond to the same entity in D_m.

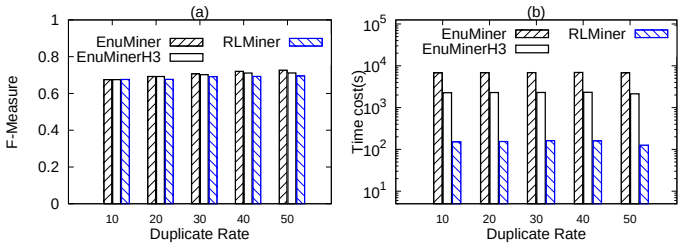

Fig. 7: Varying duplicate rate over Adult

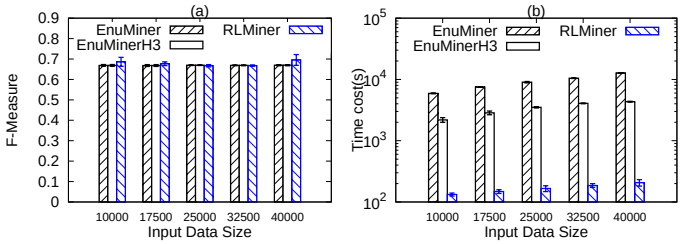

Fig. 8: Varying input data size over Adult

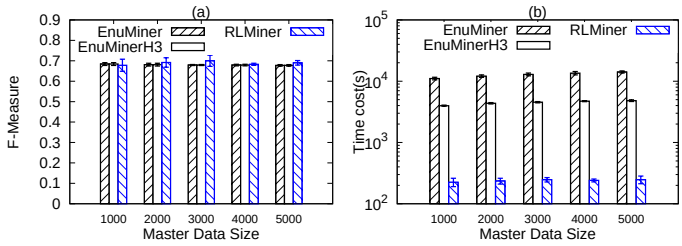

Fig. 9: Varying master data size over Adult

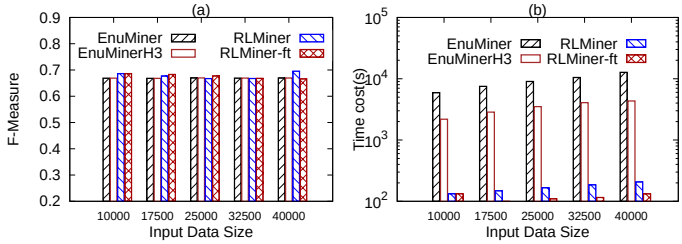

Fig. 10: Incremental input data over Adult

We fix master data size as 5000 and input data size as 10000. Then we first sample $d\%$ tuples from the master data and the others from other data. As shown in Figure 7, we can find F-Measures of EnuMiner and RLMiner increase as the duplicate rate $d\%$ increases. This is accord with our intuition, because the more input data exists in the master data, the likely it is to discover eRs that can return fixes from the master data.

D. Varying Data Size

To evaluate the scalability, we test algorithms by varying input data sizes and master data sizes.

1) Varying Input Data Size: To show the scalability of the proposed discovery algorithms, we vary input data sizes of Adult dataset from 10k to 40k, and the experimental results are shown in Figure 8. Obviously, when the input data size increases, the domain size increases. After we get the schema matching result as introduced in Section II-C, EnuMiner needs to enumerate combinations of attributes and values for LHS and patterns of the rules. That is, its enumeration space size is positively associated with the input domain size. As the input data size increases, the time costs of EnuMiner increase, while that of RLMiner does not increase significantly, demonstrating superior scalability of RLMiner.

2) Varying Master Data Size: To see how master data size affects the performance of algorithms, we fix input data size as 40k and vary master data sizes from 1k to 5k over Adult dataset. The F-measure and time cost results are illustrated in Figure 9. Compared to the results of input data size in Figure 8(b), the master data size has less effect on the time consumption of EnuMiner. By contrast, RLMiner is still more efficient on master data of all sizes.

In the experiments, we randomly sample multiple times to get different master and dirty datasets. Figures 8 and 9 report the mean (bar) and variance (interval) of the results in multiple times, to illustrate the sensitivity w.r.t. the choice of the master data sample. As shown, the variances of the results are small, illustrating the robustness of our proposals. It is not surprising that the variance of RLMiner guided by the learned model is a bit larger than that of EnuMiner enumerating all the possible rules.

It is notable that in most cases, the left hand side (LHS) attribute set of editing rules is small, as illustrated in Table II. In this sense, limiting the size of LHS X to 3 is a practical heuristic strategy. We implement a method EnuMinerH3 by limiting the length of LHS attributes and patterns to 3. The results in Figures 8 and 9 show that EnuMinerH3 achieves nearly the same precision and recall as EnuMiner, but takes much less time. Nevertheless, the time cost of EnuMinerH3 is still much higher than that of RLMiner. Such a significantly higher time cost prevents the heuristic handling the incrementally enriched input and master data, as introduced in Section V-D3 below.

3) Incremental Discovery: As shown in Figure 6 and Figure 7, EnuMiner costs about 4.7 hours, while the heuristic EnuMinerH3, also takes about 1.2 hours. It might be acceptable to take hours if conducted only once. In practice, however, both the input data and the master data are enriched gradually. That is, the discovery is performed repeatedly. As illustrated in Figures 10 and 11, an efficient discovery algorithm is highly desired. It is also the reason why we introduce RLMiner-ft with fine-tuning to further reduce the time cost of incremental discovery, w.r.t. the enriched input and master data.

As illustrated in Figures 10 and 11, the input and master data are enriched incrementally. Rather than re-training and discovering over the enriched data each time, RLMiner-ft only fine-tunes the model. As shown, the F-Measure of RLMiner-ft is very close to those by EnuMiner or RLMiner starting from scratch. The results exhibit the good generalization of

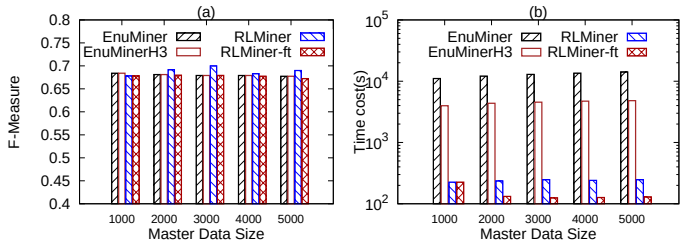

Fig. 11: Incremental master data over Adult

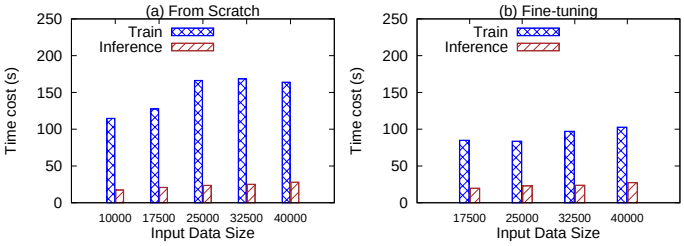

Fig. 12: Training and inference time of RLMiner

our approach. The corresponding time cost of RLMiner-ft is of course much lower.

4) Training and Inference Time Costs: As shown in Figure 4, a step means the agent selects an action a_t according to the current state s_t and the environment updates the state to s_{t+1}. An episode in RLMiner represents the entire process of generating a rule tree from scratch. Following [24], we train RLMiner with a fixed number of steps (5000) instead of a fixed number of episodes.

We show the training costs of RLMiner (from scratch) in Figure 12(a), and RLMiner-ft (fine-tuning) in Figure 12(b). Since fine-tuning uses less steps, the corresponding time cost is lower. We also show the inference time of RLMiner in Figure 12. RLMiner generally mines top-K rules in about 150 steps in the inference phase, taking less time than the training phase.

VI. Related Work

A. Rules with External Data

Rules-based approaches play an important role in data cleansing due to their good interpretability (e.g., [21], [42]). In recent years, more attention has been paid to data cleansing leveraging high-quality external data. For example, KATARA [13] is crowd powered data cleaning system that utilizes rules about knowledge base for repair. Detective rules [20] is a kind of graph-based rule and defined on knowledge base for data cleaning. Editing rules [18] are designed for leveraging relational master data to repair data. The previous work [18] focuses on the editing rule definition and determining certain fixes with editing rules. There does not exist any efficient and effective editing rule discovery algorithm. Thus, our work is aimed to discover high-utility editing rules efficiently. When the data size is large, our proposed RLMiner can efficiently discover a set of editing rules from data, because RLMiner avoids enumerating all candidate rules.

B. Rule Discovery

Unfortunately, our solution designed for editing rule (eR) discovery cannot be applied to discover other types of rules, owing to the unique scenario of repairing input data referring to the master data by eR. (1) Different from the discovery of most other rules from one data source, e.g., CFD discovery [16], the editing rule discovery considers another data source, i.e., master data. In particular, CFD discovery concerns whether the input data satisfy the rule, whereas eR discovery focuses on how certain the fix of input data is returned from the master data by eR. (2) Although the discovery of matching rules (MD) also considers two data sources [40], it treats both sources equally and identifies their matching relationships. In contrast, eR utilizes the clean master data source to repair the dirty input data source. The algorithm for discovering eRs with certain fixes in the input data does not apply to find MD for matching the data from two sources. (3) Consequently, the measure for evaluating the certainty of fix by eR is different from the confidence measure for CFD [14] or MD [39]. The latter denotes the maximum number of data that satisfy the rules.

VII. Conclusion

Editing rules utilize high-quality master data to repair low-quality input data. To automatically discover editing rules, we first introduce EnuMiner that costly enumerates all attribute and value combinations. For this reason, EnuMiner does not scale well. Then, we argue to mine editing rules by growing a rule tree, which is indeed a Markov Decision Process. Specifically, each node in the tree denotes a rule. The algorithm will generate a new rule as the child node from the current node. We propose an RL-based editing rule discovery algorithm (RLMiner) that trains an agent to wisely discover rules without enumerating the value combination space. Remarkably, when the master and input data are gradually enriched, very likely in practice, the learned agent could be incrementally fine-tuned, rather than re-discovering the rules from scratch. Following the idea of evaluating rules, we design a reward function that is in line with the rule discovery scenario and makes our algorithm perform effectively and efficiently. The experimental results show that our proposed RLMiner is able to mine high-utility editing rules and is more scalable on the datasets with lots of attributes and large domains.

Acknowledgment: This work is supported in part by the National Natural Science Foundation of China (62072265, 62232005, 62021002), the National Key Research and Development Plan (2021YFB3300500, 2019YFB1705301, 2019YFB1707001), Beijing National Research Center for Information Science and Technology (BNR2022RC01011), and Alibaba Group through Alibaba Innovative Research (AIR) Program. Shaoxu Song (https://sxsong.github.io/) is the corresponding author.

REFERENCES

[1] https://archive.ics.uci.edu/ml/datasets/Adult.

[2] https://archive.ics.uci.edu/ml/datasets/nursery.

[3] https://github.com/EliasMei/ERMiner.

[4] https://github.com/EliasMei/ERMiner/blob/main/doc/er.pdf.

[5] https://github.com/j-r77/cfddiscover.

[6] https://www.kaggle.com/datasets/starbucks/store-location.

[7] https://www.kaggle.com/kimjihoo/coronavirusdataset.

[8] https://www.spb.gov.cn.

[9] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. Fast discovery of association rules. In *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI/MIT Press, 1996.

[10] P. C. Arocena, B. Glavic, G. Mecca, R. J. Miller, P. Papotti, and D. Santoro. Messing up with BART: error generation for evaluating data-cleaning algorithms. *Proc. VLDB Endow.*, 9(2):36–47, 2015.

[11] C. Batini and M. Scannapieco. *Data Quality: Concepts, Methodologies and Techniques*. Data-Centric Systems and Applications. Springer, 2006.

[12] X. Chu, I. F. Ilyas, and P. Papotti. Holistic data cleaning: Putting violations into context. In *ICDE*, pages 458–469. IEEE Computer Society, 2013.

[13] X. Chu, J. Morcos, I. F. Ilyas, M. Ouzzani, P. Papotti, N. Tang, and Y. Ye. KATARA: A data cleaning system powered by knowledge bases and crowdsourcing. In *SIGMOD Conference*, pages 1247–1261. ACM, 2015.

[14] G. Cormode, L. Golab, F. Korn, A. McGregor, D. Srivastava, and X. Zhang. Estimating the confidence of conditional functional dependencies. In U. Çetintemel, S. B. Zdonik, D. Kossmann, and N. Tatbul, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 469–482. ACM, 2009.

[15] W. W. Eckerson. Data quality and the bottom line. *TDWI Report, The Data Warehouse Institute*, pages 1–32, 2002.

[16] W. Fan, F. Geerts, L. V. S. Lakshmanan, and M. Xiong. Discovering conditional functional dependencies. In Y. E. Ioannidis, D. L. Lee, and R. T. Ng, editors, *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, pages 1231–1234. IEEE Computer Society, 2009.

[17] W. Fan, F. Geerts, J. Li, and M. Xiong. Discovering conditional functional dependencies. *IEEE Trans. Knowl. Data Eng.*, 23(5):683–698, 2011.

[18] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Towards certain fixes with editing rules and master data. *VLDB J.*, 21(2):213–238, 2012.

[19] J. Han, H. Cheng, D. Xin, and X. Yan. Frequent pattern mining: current status and future directions. *Data Min. Knowl. Discov.*, 15(1):55–86, 2007.

[20] S. Hao, N. Tang, G. Li, and J. Li. Cleaning relations using knowledge bases. In *ICDE*, pages 933–944. IEEE Computer Society, 2017.

[21] M. Interlandi and N. Tang. Proof positive and negative in data cleaning. In *ICDE*, pages 18–29. IEEE Computer Society, 2015.

[22] H. Lan, Z. Bao, and Y. Peng. A survey on advancing the DBMS query optimizer: Cardinality estimation, cost model, and plan enumeration. *Data Sci. Eng.*, 6(1):86–101, 2021.

[23] G. Li, X. Zhou, S. Li, and B. Gao. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proc. VLDB Endow.*, 12(12):2118–2130, 2019.

[24] E. Liang, H. Zhu, X. Jin, and I. Stoica. Neural packet classification. In *SIGCOMM*, pages 256–269. ACM, 2019.

[25] D. Loshin. *Master data management*. Morgan Kaufmann, 2010.

[26] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[27] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal. graph2vec: Learning distributed representations of graphs. *CoRR*, abs/1707.05005, 2017.

[28] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10(4):334–350, 2001.

[29] T. Rekatsinas, X. Chu, I. F. Ilyas, and C. Ré. Holoclean: Holistic data repairs with probabilistic inference. *Proc. VLDB Endow.*, 10(11):1190–1201, 2017.

[30] D. Ritze, O. Lehmberg, and C. Bizer. Matching HTML tables to dbpedia. In *WIMS*, pages 10:1–10:6. ACM, 2015.

[31] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. *Inf. Process. Manag.*, 24(5):513–523, 1988.

[32] P. Schirmer, T. Papenbrock, I. K. Koumarelas, and F. Naumann. Efficient discovery of matching dependencies. *ACM Trans. Database Syst.*, 45(3):13:1–13:33, 2020.

[33] R. Shraga, A. Gal, and H. Roitman. Adnev: Cross-domain schema matching using deep similarity matrix adjustment and evaluation. *Proc. VLDB Endow.*, 13(9):1401–1415, 2020.

[34] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. P. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of go with deep neural networks and tree search. *Nat.*, 529(7587):484–489, 2016.

[35] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[36] 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, and D. Hassabis. Mastering the game of go without human knowledge. *Nat.*, 550(7676):354–359, 2017.

[37] S. Song and L. Chen. Discovering matching dependencies. In *CIKM*, pages 1421–1424. ACM, 2009.

[38] S. Song, L. Chen, and H. Cheng. Parameter-free determination of distance thresholds for metric distance constraints. In *ICDE*, pages 846–857. IEEE Computer Society, 2012.

[39] S. Song, L. Chen, and H. Cheng. On concise set of relative candidate keys. *Proc. VLDB Endow.*, 7(12):1179–1190, 2014.

[40] S. Song, L. Chen, and P. S. Yu. On data dependencies in dataspaces. In S. Abiteboul, K. Böhm, C. Koch, and K. Tan, editors, *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 470–481. IEEE Computer Society, 2011.

[41] R. S. Sutton and A. G. Barto. *Reinforcement learning - an introduction*. Adaptive computation and machine learning. MIT Press, 1998.

[42] J. Wang and N. Tang. Towards dependable data repairing with fixing rules. In *SIGMOD Conference*, pages 457–468. ACM, 2014.

[43] J. Wang, I. Trummer, and D. Basu. UDO: universal database optimization using reinforcement learning. *Proc. VLDB Endow.*, 14(13):3402–3414, 2021.

[44] M. Yakout, L. Berti-Équille, and A. K. Elmagarmid. Don't be scared: use scalable automatic repairing with maximal likelihood and bounded changes. In *SIGMOD Conference*, pages 553–564. ACM, 2013.

[45] Y. You, T. Chen, Y. Sui, T. Chen, Z. Wang, and Y. Shen. Graph contrastive learning with augmentations. In *NeurIPS*, 2020.

[46] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu, M. Ran, and Z. Li. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *SIGMOD Conference*, pages 415–432. ACM, 2019.

[47] C. Zhao, L. Ren, Z. Zhang, and Z. Meng. Master data management for manufacturing big data: a method of evaluation for data network. *World Wide Web*, 23(2):1407–1421, 2020.