BIG DATA MINING AND ANALYTICS
ISSN 2096-0654 03/14 pp31–44
DOI: 10.26599/BDMA.2024.9020070
Volume 8, Number 1, February 2025

A Reinforcement Learning Approach for Graph Rule Learning

Zhenzhen Mai, Wenjun Wang, Xueli Liu*, Xiaoyang Feng, Jun Wang, and Wenzhi Fu

Abstract: We study the problem of learning rules for graphs. Traditional methods often suffer from large search spaces due to the enumeration of all candidate rules. Although some recent neural logic methods are more efficient in learning rules, they are generally restricted to learning chain-like rules with limited expressiveness. Taking the advantage of Reinforcement Learning (RL) in reducing search space, we implement a policy network based RL method for learning graph rules, denoted as GraphRulRL. In our research, we convert graph rules into sequences of edges, transforming the task of graph rule learning into a process of sequentially adding edges that can be solved by RL. Specifically, GraphRulRL follows a two-stage framework. In the first stage, we train a policy network for graph rule learning, which evaluates graph rules using support with anti-monotonicity as rewards during training. In the second stage, we integrate the well-trained policy network with beam search for iterative searching to generate graph rules. Experimental results prove the effectiveness of the proposed method.

Key words: graph rules; Reinforcement Learning (RL); rule learning

1 Introduction

A variety of rules has been studied for graphs, and implemented in various domains, including error detection[1, 2], entity resolution[3], and user recommendation[4]. Rules on graphs usually have more complex structures than relational rules. For instance, Graph Pattern Association Rules (GPAR)[4] are defined as $Q(x, y) \Rightarrow q(x, y)$, where $Q(x, y)$ is a graph pattern in which x and y represent two designated nodes, and $q(x, y)$ is an edge labeled q from x to y, on which the search conditions imposed on x and y are the same as in Q. $Q(x, y)$ can be regarded as the rule body, and $q(x, y)$ is the rule head that is derived from the conditions in rule body.

To make practical use of graph rules, many researches have made significant efforts in discovering rules from real-life data. Traditional methods[4–6] follow the level-wise search paradigm to mine graph rules, requiring enumeration and verification of candidate rules in an exponential space. Although they typically employ different pruning strategies to reduce search space, the subgraph matching that is used to evaluate candidates (i.e., check whether it satisfies the support threshold) still leads to exponential computational costs. With the significant breakthroughs in deep learning, some researches[7–10] have applied neural networks for rule learning tasks. For instance, path-based methods[7] enumerate all relational paths on a graph as candidate rules, and then learn a weight for each rule to evaluate its quality. There are also neural logic programming methods[10]

• Zhenzhen Mai, Xueli Liu, Xiaoyang Feng, and Jun Wang are with College of Intelligence and Computing, Tianjin University, Tianjin 300072, China. E-mail: maizz2020@tju.edu.cn; xueli@tju.edu.cn; iexyfeng@tju.edu.cn; jun.wang@tju.edu.cn.
• Wenjun Wang is with College of Intelligence and Computing, Tianjin University, Tianjin 300072, and also with Yazhou Bay Innovation Institute, Hainan Tropical Ocean University, Sanya 572022, China. E-mail: wjwang@tju.edu.cn.
• Wenzhi Fu is with School of Informatics, University of Edinburgh, Edinburgh EH89YL, UK. E-mail: wenzhi.fu@ed.ac.uk.
* To whom correspondence should be addressed.
Manuscript received: 2024-05-14 ; revised: 2024-09-24; accepted: 2024-10-05

that can learn both rules and weights simultaneously. However, these methods also need to learn rules in a large search space and struggle to identify high-quality rules. Some approaches[11, 12] adopt Reinforcement Learning (RL) to search for rules, which can greatly reduce the complexity of search space by avoiding bad decisions in early stage of rule generation. But they mainly focus on learning chain-like rules, which have limited expressive ability.

However, mining graph rules from real-life data is more challenging due to their complex structures. As shown in Fig. 1, the edges in the rule body of ρ_a (ρ_a is a chain-like rule) naturally has a sequential order, while the structure of the rule body for graph rule ρ_b is more complex and unordered. Considering the advantages of RL in reducing search spaces and handling sequence decision-making problems, the key issue is how to use RL method for graph rules learning.

Considering the unordered nature of edges in the rule body of graph rules, using RL for graph rule learning needs to consider several issues. Firstly, how to convert graph rules into sequence rules so that the task of graph rule learning can be transformed into a sequential decision-making process that can be solved by RL? Secondly, how to evaluate the learned graph rules? Finally, how to avoid generating redundant rules during the whole learning process? Here, redundant rules refer to those that have the same consequence (i.e., the same rule head) and isomorphic patterns (i.e., rule bodies).

For the first issue, we consider adopting the Depth-First Search (DFS) code defined in gSpan[13] to convert graph rules into ordered edge sequences. To generate new rules, the process typically involves starting with a given rule head and gradually adding edges to the rule body. We take each graph rule as DFS codes, and the new graph rule is generated by extending the current DFS codes with an edge, that is extracted from the data graph by calculating the matches of the current DFS codes, and then extending the matches with one-edge growth from nodes on the rightmost path. Here the right most path denotes the path between the nodes with the smallest and largest ids of the current DFS codes (i.e., graph rules).

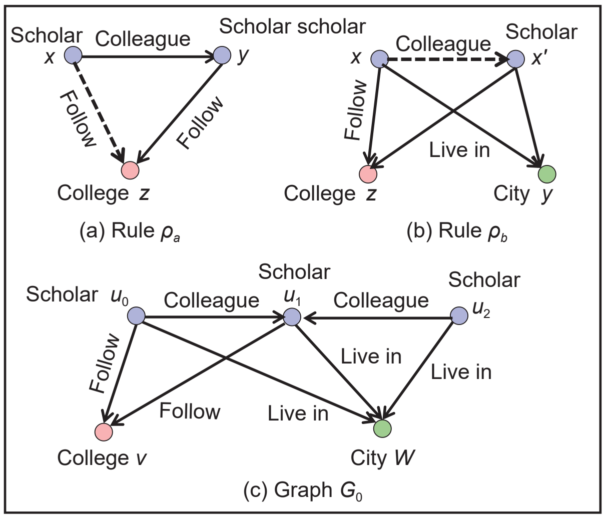

Fig. 1 Examples of rules and graphs. ρ_a is a chain-like rule, and ρ_b is a graph rule. The solid lines in a rule construct the rule body, and the dashed line represents the rule head.

For the second issue, we use the statistical metric support[14, 15] to evaluate graph rules. Support usually refers to the frequency of the rule applied on the graph. Such a definition of support often lacks anti-monotonicity, which means that as more conditions (i.e., edges) are added to a rule, the frequency or significance of the rule being satisfied remains the same or decreases. Since new rules are generated by adding new edges to the rule body, the new rules need to satisfy monotonicity. Therefore, we take support with anti-monotonicity as the reward of RL to evaluate graph rules.

For the third issue, we intend to identify and prune redundant rules, which are generated due to the different orders of adding edges, by comparing the minimum DFS codes of rule bodies for graph rules. Therefore, we can prune redundant rules by avoiding expanding new rules upon them.

With these strategies mentioned above, we propose a method that can learn graph rules with RL, denoted as GraphRulRL. Our main contributions include:

(1) By transforming graph rules into ordered sequences of edges, we formalize the process of learning graph rules into a sequential decision-making problem that can be solved by RL.

(2) We train a policy network of RL for graph rule learning, in which it designs a reward function that takes support with anti-monotonicity as evaluation. Considering the anti-monotonicity of rules helps reduce search space.

(3) We embed the trained policy network into the beam search process, aiming to generate as many high-quality graph rules as possible.

(4) Validation of the obtained rules are conducted to demonstrate the effectiveness of the proposed method.

The rest of the paper is structured as follows. In Section 2, we introduce the related works. Some preliminaries and notations are presented in Section 3. Section 4 elaborates on the overall framework of GraphRulRL in detail, including the training process of the policy network and the generation of graph rules. Section 5 reports the experimental results, and conclusions are drawn in Section 6.

2 Related Work

Rule learning. There have been several methods for discovering rules from graphs. Traditional approaches follow a level-wise search paradigm for rule mining, which usually requires enumerating and validating rules in an exponential space. For example, GFDs[5], GPARs[4], and GARs[6] can mine graph rules by adopting different pruning strategies to improve efficiency. Some recent researches apply neural networks for rule learning, in which they usually learn the weights of logic rules efficiently for quality evaluation. PRA[7] samples paths by using random walk with restart strategies from the data graph to infer chain-like rules. DeepPath[11] formulates the path-finding problem as a sequential decision-making problem with RL. The path-based methods aim at learning chain-like rules. Some graph-based methods, like GraIL[16] and CoGraph[8], extend the path-based methods and allow the interpretation to be structured as graphs, thereby providing richer expressiveness. The matrix-based methods[10, 17, 18] use matrix operations to describe the logical relationships between entities. TensorLog[10] and NeuralLP[17] focus on learning chain-like rules. NLIL[18] constructs a more compact framework for rule-learning by stacking three transformer networks, and can tackle the non-chain-like rules, which suffers the complexity of models.

In this paper, we formulate the learning of graph rules into a sequential decision-making problem that can be solved by RL. It is a heuristic method that combines beam search with a policy network to generate as many high-quality graph rules as possible.

Reinforcement learning. Some researchers apply reinforcement learning for rule learning, benefiting from its ability to explore and make optimal decisions in large search spaces. These methods[11, 19] typically model path-based approaches as sequential decision-making processes to learn chain-like rules. However, the reward signal is very sparse, making it difficult to train an effective path-finding agent. Some studies[20] attempt to obtain better rewards by using embedding-based methods for reward shaping. In our research, we design a new reward mechanism to train the network for graph rule learning, aiming to achieve better performance.

3 Preliminary

In this section, we review some basic notations. Assume a countably infinite set Θ denotes the node and edge label in graphs.

3.1 Graph rules

Graphs. We consider directed graphs $G = (V, E, L)$, where (1) V is a finite set of nodes; (2) $E \subseteq V \times V$ is a set of edges, in which (v, v') denotes an edge from node v to v'; and (3) each node $v \in V$ is labeled $L(v) \in \Theta$, indicating its label or content, e.g., scholar and college, and each edge e is labeled $L(e) \in \Theta$, indicating its label or content, e.g., "follow" and "live in", as found in property graphs, social networks, and knowledge bases. G_0 in Fig. 1c shows a simple social network.

Patterns. A graph pattern is defined as a graph $Q[\bar{x}] = (V_Q, E_Q, L_Q, \mu)$, where (1) V_Q is a finite set of pattern nodes, and E_Q is a finite set of pattern edges; (2) L_Q is a function with range in Θ that assigns a node label $L_Q(u)$ for each $u \in V_Q$, and assigns a edge label $L_Q(e)$ for each $e \in E_Q$; and (3) $\bar{x}$ is a list of distinct variables, each denoting an entity in V_Q; (4) μ is a bijective mapping from $\bar{x}$ to V_Q, i.e., it assigns a distinct variable to each node v in V_Q. For $x \in \bar{x}$, we use $\mu(x)$ and x interchangeably when it is clear in the context.

Example 1: In the rule body of ρ_a in Fig. 1, it contains three pattern nodes, i.e., x, y, and z, and each node has its own label. According to the definition of pattern, we can take the rule body of ρ_a as a pattern Q. Similarly, for the rule body of graph rule ρ_b.

Pattern matching. We adopt the isomorphism semantics of pattern matching. A match of pattern $Q[\bar{x}]$ in graph G is a bijective function h from nodes of Q to the nodes of a subgraph G' of G, such that for each node $u \in V_Q$, $L_Q(u) = L(h(u))$; and (u, u') is an edge in Q if and only if $(h(u), h(u')$ is an edge in G', and $f(u, u') = L(h(u), h(u')$. We say that G' matches Q.

We denote the match G' as a vector $h(\bar{x})$, consisting of $h(x)$ for all $x \in \bar{x}$ in the same order as $\bar{x}$. Intuitively, $\bar{x}$ is a list of entities to be identified, and $h(\bar{x})$ is an instantiation of $\bar{x}$ in G, one node for each entity.

We denote by $Q(G)$ the set of all matches of Q in G. For each pattern node u, we use $Q(u, G)$ to denote the set of all matches of u in $Q(G)$, i.e., $Q(u, G)$ consists of nodes v in G, such that there exists a function h under which a subgraph $G' \in Q(G)$ is isomorphic to Q, $v \in G'$ and $h(u) = v$.

Example 2: For convenience, we denote the rule bodies of ρ_a and ρ_b in Fig. 1 as patterns Q_a and Q_b, respectively. For Q_b and G_0, a match in $Q_b(G_0)$ is $x \mapsto u_0$, $x' \mapsto u_1$, $y \mapsto w$, and $z \mapsto v$. Here $Q_b(x, G_0)$ includes u_0.

Graph rules. A graph rule φ is defined as $Q[\bar{x}] \to q(x, y)$, where (1) $Q[\bar{x}]$ is a pattern of φ; and (2) $q(x, y)$ is an edge labeled q from x to y, where x and y are two designated nodes in $\bar{x}$. We refer Q and q as the rule body and rule head of φ, respectively. In such a definition, each pattern edge of Q can be considered as a predicate, and all the edges in the pattern Q form the rule body. Similarly, q is a predicate, serving as the consequence of φ.

Example 3: For ρ_a in Fig. 1, it can be expressed as a graph rule φ_a: $Q_a(\bar{x}) \to \text{Follow}(x, z)$, where it indicates that x works for z when x and y are colleagues, and y works for college z.

For ρ_b, it can be expressed as a graph rule φ_b : $Q_b(\bar{x}) \to \text{Colleague}(x, x')$. It says that if x and x' both work for college z, and they live in the same city y, then x and x' are colleagues.

Compared to chain-like rules, such logic rule φ exhibits a higher level of expressive capability since pattern Q possesses more complex structures. By employing pattern matching algorithm to obtain match $h(\bar{x})$ of Q, we can deduce $q(h(x), h(y))$. Match $h(\bar{x})$ is an instantiation of pattern Q on the graph.

Support. The support of a graph rule φ: $Q[\bar{x}] \to q(x, y)$ in graph G should represent the frequency that φ can be applied to G. We adopt the definition in Ref. [4], where it defines the support of φ: $Q[\bar{x}] \to q(x, y)$ in graph G as

$$\text{support}(\varphi, G) = \|P_\varphi(x, G)\| \tag{1}$$

where φ is treated as pattern $P_\varphi(x, y)$ with designated nodes x and y. It quantifies support of a rule φ in terms of the number of distinct matches of two designated nodes x and y in $P_\varphi(G)$, and $P_\varphi(G)$ is the set of all matches of P_φ in G.

One can verify that this support measure has anti-monotonicity. Specifically, if two rules $\varphi_t = Q[\bar{x}] \to q(x, y)$ and $\varphi_{t+1} = Q[\bar{x}'] \to q(x, y)$ with the same consequence $q(x, y)$, we say that φ_t has a lower order than φ_{t+1}, denoted by $\varphi_t \preceq \varphi_{t+1}$ if $Q[\bar{x}] \sqsubseteq Q[\bar{x}']$. That is, φ_t is less restrictive than φ_{t+1}, and $\text{support}(\varphi_t, G) \geqslant \text{support}(\varphi_{t+1}, G)$. The anti-monotonicity requires that when more conditions are added to a rule, the frequency of the rule being satisfied does not increase but remains the same or decreases.

3.2 DFS codes of graph

DFS codes in gSpan[13] describes a graph in an ordered sequence of edges by performing a depth-first search on it. This sequence is composed of a list of five-tuples, i.e., $(u, v, (l_u, l_e, l_v))$. Each five-tuple represents an edge, in which u is the source node id, and v is the target node id, $l_u \in \Theta$ and $l_v \in \Theta$ denote the vertex label of u and v, respectively, and $l_e \in \Theta$ is the edge label between u and v. We also denote (l_u, l_e, l_v) as a label triplet that many edge instances of the data graph have been conformed to.

Example 4: We take ρ_a in Fig. 1 as an example. For clarity, we only consider the rule body of ρ_a, i.e., the solid lines. If the searching starts from (x, Colleague, y), the DFS codes of ρ_a contain (0, 1, (Scholar, Colleague, Scholar)) and (1, 2, (Scholar, Follow, College)). If it starts from (y, Follow, z), the DFS codes of ρ_a contain (0, 1, (Scholar, Follow, College)) and (2, 0, (Scholar, Colleague, Scholar)).

Different traversal orders of nodes and edges can produce multiple DFS codes for a graph G. To address this issue, the author defined a DFS Lexicographic Order to compare each pair of DFS codes. Suppose there is linear order ($\prec_e$) for the lexicographic combinations of node ids, node labels and edge labels (please refer gSpan[13] for detailed definition). DFS Lexicographic Order is a linear order defined as follows. If $A = (a_0, a_1, \ldots, a_m)$ and $B = (b_0, b_1, \ldots, b_n)$ are two states, then $A \leqslant B$ if either of the following is true:

(1) $\exists t, 0 \leqslant t \leqslant \min(m, n), \forall k < t, a_k = b_k, a_t \prec_e b_t$,

(2) $\forall 0 \leqslant k \leqslant m, a_k = b_k$, and $n \geqslant m$.

Among all possible DFS codes of graph G, the minimum code is the one with the minimum lexicographical order with the same elements. If two graphs G and G' have the same minimum lexicographical order, then G is isomorphic to G'. By comparing the minimum lexicographical order, it can effectively identify graphs or patterns that are isomorphic.

3.3 Markov decision process for rule learning

In RL, the Markov Decision Process (MDP) is an important framework used to describe the problem of an agent learning the optimal behavior policy by interacting with an environment.

In general, MDP is modeled as $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$, where $\mathcal{S}$ is a continuous state space, $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$ is a set of all possible actions, $\mathcal{P}(S_{t+1} = s' \mid S_t = s, A_t = a)$ is a state transition function that deterministically maps the state s and action a to the next state s'. $\mathcal{R}(s, a)$ is the reward function for each (s, a) pair.

Some studies[11, 19] adopt MDP to learn chain-like rules. Typically, it starts with a given predicate as the rule head, progressively adding predicates to construct the rule body, where predicates are edge labels. Take ρ_a in Fig. 1 for instance, given the predicate Follow (x, z) as rule head, it first adds the predicate Colleague (x, y) to the rule body, deriving a partial rule Colleague (x, y) $\to$ Follow (x, z); then it adds another predicate Follow (y, z) to form a complete rule Colleague (x, y) $\wedge$ Follow $(y, z) \to$ Follow (x, z). Therefore, the whole process can be modeled as a MDP, with a reward to measure the quality of each step.

Chain-like rules can be seen as closed paths, and learning chain-like rules using RL methods is equivalent to the process of finding closed paths in a graph. Additionally, the edges (i.e., predicates) within the body of a chain-like rule have a sequential order, meaning each newly added edge must follow directly after the previous one.

However, for graph rules with complex structures, the aforementioned process cannot be applied directly. How to utilize MDP to describe the modeling process of graph rules is a key focus of our research, which will be elaborated in Section 4.

4 Methodology

Our goal is to generate graph rules by sequentially adding edges to the rule body, starting with an edge label as the rule head. A graph rule can be converted to DFS codes, and then adding all possible edges to the current DFS codes for expanding new rules, in which these edges can be obtained by enumerating all the matches of the current DFS codes and then expanding the matches with one-edge growth from nodes on the rightmost path of matches. More specifically, when given a label triplet (l_u, l_e, l_v), we encode it into $(0, 1, (l_u, l_e, l_v))$ as the rule head, upon which new edges added to it are all part of the rule body. Then we enumerate the matches of $(0, 1, (l_u, l_e, l_v))$ in the data graph and find the possible edges (i.e., actions) that can be added to it. With these available edges (i.e., actions), we adopt the policy network of RL to evaluate the quality of these edges and add a high-value edge to the rule body to form a new rule. When a new rule is evaluated as effective, it continues expanding upon the new rule by exploring the data graph for possible edges and using the RL agent to evaluate these edges. Figure 2 illustrates the framework of our method.

Taking the graph rule in Fig. 1 for instance, we set the pattern edge (Scholar, Colleague, Scholar) as the rule head, denoted as e_0 = (0, 1, (Scholar, Colleague, Scholar). By exploring the matches of e_0 in G_0, we get four possible expanding edges, including (0, 2, (Scholar, Follow, College)), (0, 2, (Scholar, LiveIn, City)), (1, 2, (Scholar, Follow, College)), and (1, 2, (Scholar, LiveIn, City)). Then, adding a new edge e_1 = (1, 2, (Scholar, Follow, College)) to the rule body if this edge has a high value by RL agent, it forms a rule $e_1 \to e_0$, and the edge sequence of the rule is $[e_0, e_1]$. If this rule is proven to be effective using support, we can expand new edges upon it. After exploring the data graph, we find that another new edge e_2 = (0, 2, (Scholar, LiveIn, City)) can be added to the rule body, forming a new rule $e_1 \wedge e_2 \to e_0$, with an edge sequence $[e_0, e_1, e_2]$. We can generate new rules by adding new edges to the rule body of this rule, as long as this rule is validated as effective.

It is obvious that each graph rule is in the form of DFS codes, which is constituted by a sequence of multiple five-tuples. We take graph rules as states, and the edges that can be added to the state are actions. Consequently, we formulate the problem of graph rule learning as a MDP that can be solved by RL and propose an approach GraphRulRL for graph rule learning. GraphRulRL consists of two parts: (1) training of the policy network of RL, and (2) the rule generation guided by the trained policy network. In the rest of this section, we will describe the components of the MDP in detail first. Then we outline the training process of the policy network, and present an algorithm that can generate graph rules with the trained policy network subsequently. The symbols used in GraphRulRL are shown in Table 1.

4.1 MDP components

Here, we will give the definitions of actions, states, and rewards for the MDP components in graph rule learning, respectively.

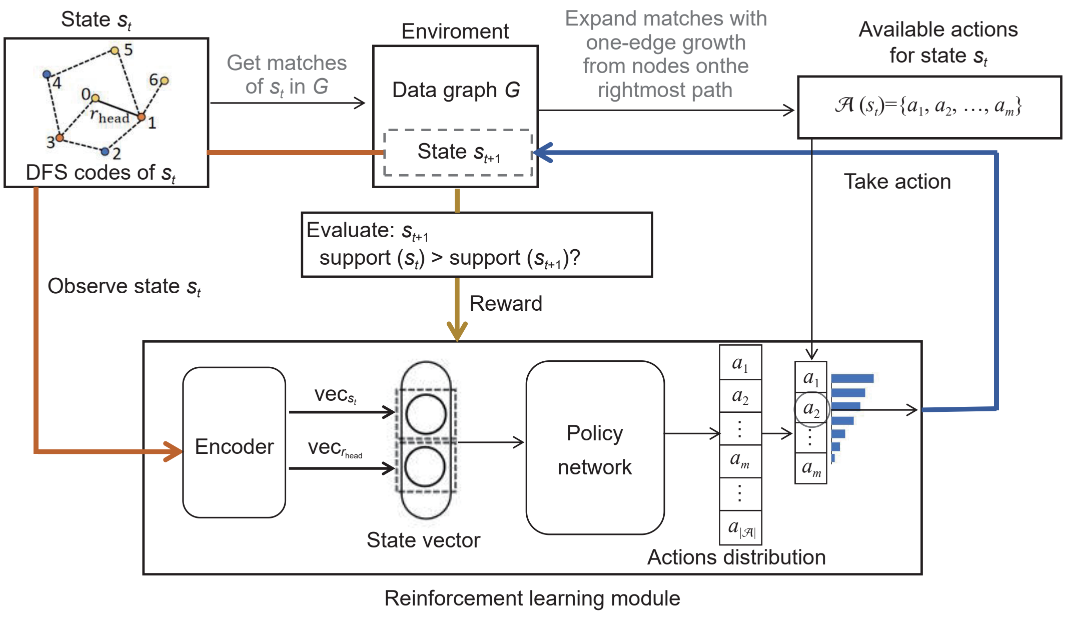
State s_t
DFS codes of s_t
r_{head}
Get matches of s_t in G
Enviroment
Data graph G
State s_{t+1}
Expand matches with one-edge growth from nodes onthe rightmost path
Available actions for state s_t
$\mathcal{A}(s_t)=\{a_1, a_2, \ldots, a_m\}$
Take action
Evaluate: s_{t+1} support (s_t) > support (s_{t+1})?
Observe state s_t
Reward
Encoder
vec_{s_t}
$\text{vec}_{r_{\text{head}}}$
State vector
Policy network
Actions distribution
Reinforcement learning module

Fig. 2 Overview of our method, which is based on the framework of RL. It converts the state s_t (i.e., a graph rule) learned from the environment (i.e., data graph G) into a sequence of edges encoded by DFS codes, and obtains an available actions set $\mathcal{A}(s_t)$ for s_t from the environment by getting matches of s_t in G and expanding matches with one-edge growth on nodes in the rightmost path. Meanwhile, it transforms state s_t into a vector using the encoder, and sets state vector sv_t as input of policy network by concatenating vec_{s_t} and $\text{vec}_{r_{\text{head}}}$. The policy network outputs distributions of all actions, and then the values of available actions $\mathcal{A}(s_t)$ can be obtained. During the training phase, an action is randomly chosen from $\mathcal{A}(s_t)$. In the generation phase of graph rules, top-k actions are selected from the sorted probabilities of $\mathcal{A}(s_t)$.

Table 1 Specifications for symbols used in GraphRulRL.

Notation	Type	Description
a_t	tuple	A five tuple $(u, v, (l_v, l_e, l_v))$ that represents an action (i.e., edge)
s_t	list	A list of five tuples that represents a state (i.e., a graph rule)
r_{head}	tuple	The first tuple s_t [0] of s_t represents the rule head
$\mathcal{R}(s_t, a_t)$	float	A return value evaluated by the environment (i.e., data graph) when an action a_t is taken at state s_t at step t
support	float	A statistical measure that counts the frequency of a graph rule applied to graph G
vec_{s_t}	vector	Vector representation of state s_t
$\text{vec}_{r_{\text{head}}}$	vector	Vector representation for the rule head r_{head} (i.e., s_t [0])
E	set	Set of all frequent label triplets in the data graph G
$\mathcal{M}_{\text{pos}}$ and $\mathcal{M}_{\text{neg}}$	set	Sets of positive samples and negative samples, respectively
MinDFSset	set	Set of minimum DFS codes of patterns for the graph rules
max_step	float	A number that limits the size of graph rules, i.e., the number of edges that can be added to the rule body
availActionlist	set	A set of all possible edges explored from data graph G that can be added to current state s_t
beam_width	float	A number that controls the size of beams in each iteration.

4.1.1 States and actions

Action. As the mentioned expansion process, each five-tuple represents an action. A five-tuple specifies not only the type of the edge but also its position in the rule, such as (0, 2, (Scholar, LiveIn, City)). Assuming there are m label triplets (l_u, l_e, l_v) in the data graph, the size $|\mathcal{A}|$ of actions space is $(A_N^2-1)\times m$ if the maximum number of nodes of a graph rule is N. Due to the large size of the actions space, we obtain available edges by interacting with the data graph directly when expanding for new rules upon existing rules. To accelerate the efficiency of searching possible edges, we follow the expansion principle defined in gSpan[13], which is only to expand the matches of current graph rule in the data graph G with one-edge growth from nodes in the rightmost path. The rightmost path is the path between the nodes with the smallest and largest ids of the current graph rule. This approach helps to reduce search space, facilitating the construction of graph rules quickly and efficiently.

State. Each state s_t is essentially a graph rule, as well as DFS codes constituted by a list of edges. As mentioned above, the edge sequence $[e_0, e_1, e_2]$ is a state, in which the first element in the edge sequence represents the rule head and the rest are the rule body. Since each element within an edge sequence is discrete, the graph rule cannot be directly used as input for the policy network. To address this issue, we employ representation learning techniques for modeling graph rules. Given that a graph rule is a sequence of edges, it can be modeled by Long Short-Term Memory (LSTM)[21] network. Simultaneously, as the sequence of edges also can be regarded as a pattern, Relational Graph Convolutional Networks (RGCN)[22] can be considered for modeling graph rules as well. Therefore, we will describe how these two models encode s_t respectively.

(1) Encoding with LSTM

Since state s_t is composed of a sequence of five-tuples, it is necessary to convert each five-tuple into a vector. We follow the sequence encoding method in Ref. [23]. Assuming the maximum values for node ids, node labels, and edge labels are V, X, and Y in the dataset, respectively, it converts each node id, node label, and edge label into B-nary digits, where B is the base, and each digit $d \in \{0, 1, \dots, B-1\}$. Then, it represents each digit as a one-hot vector, allowing each id or label to be vectorized as a multi-hot vector, i.e., the concatenation of these one-hot vectors. Additionally, these five multi-hot vectors can be concatenated to represent a five-tuple. The length d_e of the multi-hot vector for a five-tuple is $(B \cdot (2 \cdot \lceil \log_B(\text{Max}(|V|)) \rceil + 2 \cdot \lceil \log_B(\text{Max}(|X|)) \rceil + \lceil \log_B(\text{Max}(|Y|)) \rceil))$. After obtaining the multi-hot vector for each five-tuple in the state s_t, the state can be encoded into a multi-hot matrix, in which the size of the matrix is $(\text{length}(s_t) \times d_e)$, and $\text{length}(s_t)$ is the number of five-tuples in the state.

Based on the above encoding method, we can simply convert state s_t into a multi-hot matrix. Then, the multi-hot matrix is fed into an LSTM, and the final hidden state is taken as the representation of the entire rule, denoted as vec_{s_t}.

(2) Encoding with RGCN

Since state s_t can be taken as a graph, we also use RGCN to encode s_t. In a graph model, each vertex is associated with a feature vector, and each edge is utilized to transmit information from its source vertex to its target vertex. Following the encoding method of the five-tuples mentioned above, we vectorize vertex labels into multi-hot vectors, serving as vertex features. RGCN designs a dedicated transformation matrix for each relation, enabling the network to learn and reason about different types of relationships. After processing s_t with RGCN to obtain aggregated representations for each node, an average pooling is applied to generate the final representation vec_{s_t} for s_t.

4.1.2 Rewards

In general, rewards refer to the feedback or return that an agent receives after performing a certain action a_t at state s_t with RL, denoted as $\mathcal{R}(s_t, a_t)$. During the process of expanding graph rules, an action a_t is executed at the current state s_t to generate a new state s_{t+1}. Therefore, both s_t and s_{t+1} are graph rules and share the same rule head, and s_{t+1} is extended by adding a new edge to s_t. However, it is necessary to assess the effectiveness of s_{t+1}, that is, whether a positive reward can be obtained when choosing action a_t at state s_t.

Suppose that s_t and s_{t+1} are equivalent to φ_t and φ_{t+1}, respectively. Let $\varphi_t = Q[\bar{x}] \to q(x, y)$ and $\varphi_{t+1} = Q[\bar{x}'] \to q(x, y)$. Consequently, we can find that $\varphi_t \leq \varphi_{t+1}$ since they have the same consequence $q(x, y)$ and $Q_t[\bar{x}] \sqsubseteq Q[\bar{x}']$. According to the definition of support in Section 3.1, it has $\text{support}(\varphi_t, G) \geqslant \text{support}(\varphi_{t+1}, G)$.

In this case, we evaluate s_{t+1} with support. It returns reward depending on the evaluation results of s_{t+1}. Therefore, we set the following reward function:

$$\mathcal{R}(s_t, a_t) = \begin{cases} -1, \text{otherwise}; \\ 1, \text{if support}(s_{t+1}, G) \text{ is valid} \end{cases} \tag{2}$$

For each successful expansion step t, a positive reward is given.

4.2 Agent training

We train the RL agent directly through trial-and-error results with rewards. This section will elaborate on the training details of the RL agent, including the model of policy network, and the procedure of training policy with rewards.

4.2.1 Policy network

Since state s_t is a sequence of edges, we adopt the method mentioned in Section 4.1.1 to encode s_t.

After obtaining the representation vec_{s_t} of s_t, the state vector at step t is given as follows:

$$\mathbf{sv}_t = (\mathrm{vec}_{s_t},\ \mathrm{vec}_{r_{\mathrm{head}}}) \tag{3}$$

where vec_{s_t} is the representation of s_t encodes by LSTM or RGCN, and $\mathrm{vec}_{r_{\mathrm{head}}}$ is the representation of the rule head. In the initial state, $\mathrm{vec}_{s_t} = \mathrm{vec}_{r_{\mathrm{head}}}$. It is important to note that the state s_t does not include specific entities from the data graph; it only involves the labels of entities and edges. Representing the state vector $\mathbf{sv}_t$ by concatenating the representation of s_t and rule head r_{head} is beneficial for capturing semantic information between rules.

Then we employ a fully connected neural network to parameterize the policy function $\pi(a|s;\theta)$, which maps the state vector vec_{s_t} to the probability distribution over all possible actions. The neural network comprises two hidden layers, each followed by a Rectified Linear Unit (ReLU) layer. The output layer utilizes the softmax function for normalization.

4.2.2 Training with rewards

Our goal is to learn a policy $\pi\,(a|s;\,\theta)$ that enables us to generate high-quality graph rules under its guidance, where θ is the learned parameters. By maximizing the expected cumulative rewards from any given time step t, the target function of θ is

$$J\,(\theta) = E_{a\sim\pi\,(a|s;\,\theta)}\Big(\sum_t \mathcal{R}\,(s_t,\,a_t)\Big) = \sum_t \sum_{a\in\mathcal{A}} \pi\,(a|s;\,\theta)\mathcal{R}\,(s_t,\,a_t) \tag{4}$$

Using the Monte-Carlo Policy Gradient[24] to optimize $\pi\,(a|s;\,\theta)$,

$$\nabla_\theta J\,(\theta) = \nabla_\theta \sum_t \log\pi\,(a = a_t|s_t;\,\theta)R_{\mathrm{total}} \tag{5}$$

where R_{total} is the cumulative reward at time step t.

To update the parameters θ, we train the policy network with Algorithm 1. All the parameters are listed in Table 1. The procedure G.Action extracts possible actions that can be added to the state s_t from the data graph by getting matches of state s_t in G and expanding the matches with one-edge growth on nodes in the rightmost path. And the procedure MinDFScode calculates the minimum DFS codes of the bodies for the state s_{t+1}.

Specifically, taking a randomly given label triplet $(l_u,\,l_e,\,l_v)$ as the rule head r_{head}, new rules are generated through the sequential expansion of actions (i.e., edges) (Lines 7−18), where each iteration of the expanding process is regarded as an episode. At each step t, it builds a new rule s_{t+1} upon the previously obtained rule s_t with action a_t, until reaching the max_step. The action a_t is randomly selected from availActionslist (Line 8) which extracts possible actions from the data graph with the rightmost expansion principle in Ref. [13]. To ensure the validity of new rules, it needs to evaluate the rules with support (Line 13). We also maintain a set MinDFSset to collect the minimum DFS codes of the patterns for the effective rules. Here minimum DFS codes is used to prune redundant rules during training. Then a positive sample $(s_t,\,a_t)$ is added into $\mathcal{M}_{\mathrm{pos}}$ and a positive reward is given if the conditions are satisfied (Lines 13−15); otherwise, $\mathcal{M}_{\mathrm{neg}}$ gets a negative sample $(s_t,\,a_t)$ (Line 17), and the graph rule remains in its state before expansion. Expansion stops if the current expansion step reaches max_step. At the end of each episode, we update the policy network using Eq. (5) with positive and negative samples obtained, and R_{total} are defined as follows:

Algorithm 1 Training with reward functions

```
Input: Data graph $G$, max_step and $\varepsilon$
Output: Learned policy parameters $\pi_\theta$
1: $\theta \leftarrow$ init_parameters ( ); MinDFSset = ( );
2: $E \leftarrow$Collect all frequent label triplets in $G$;
3: for episode $\leftarrow$ 1 to $N$ do
4:     $\mathcal{M}_{\mathrm{pos}} \leftarrow \varnothing$; $\mathcal{M}_{\mathrm{neg}} \leftarrow \varnothing$;
5:     $r_{\mathrm{head}} \leftarrow E$.random ( );
       supp $\leftarrow$support ($r_{\mathrm{head}}$);
6:     $s_t \leftarrow r_{\mathrm{head}}$;
       num_step $\leftarrow$ 0;
7:     while num_step < max_step do
8:       availActionslist = $G$.Actions ($s_t$);
9:       $a \leftarrow$ random.choice (availActionslist);
10:      $s_{t+1} \leftarrow s_t + [a]$;
         mincode$\leftarrow$ MinDFScode ($s_{t+1}$[1:]);
11:      if ($r_{\mathrm{head}}$, mincode) in MinDFSset then
12:          Continue;
13:      els if ($r_{\mathrm{head}}$, mincode) not in MinDFSset and $\varepsilon \leqslant$
          support ($s_{t+1}$) then
14:          supp $\leftarrow$ support ($s_{t+1}$);
             $\mathcal{M}_{\mathrm{pos}}$.Add ($s_t$, $a$);
15:          $s_t \leftarrow s_{t+1}$;
             MinDFSset.Add (($r_{\mathrm{head}}$, mincode));
16:      else
17:          $\mathcal{M}_{\mathrm{neg}}$.Add ($s_t$, $a$); // The step fails
18:      Increment num_step;
19:    Update $\theta$ with $\mathcal{M}_{\mathrm{neg}}$; // penalize failed steps
20:    Update $\theta$ with $\mathcal{M}_{\mathrm{pos}}$;
21: return $\pi_\theta$
```

$$R_{\text{total}} = \begin{cases} -1, \text{if } \mathcal{M}_{\text{neg}} \neq \varnothing; \\ \text{length}\,(\mathcal{M}_{\text{pos}}), \text{if } \mathcal{M}_{\text{pos}} \neq \varnothing \end{cases} \tag{6}$$

where R_{total} is the total reward obtained in an episode. In practice, θ is updated using the Adam optimizer, with L_2 regularization applied.

4.3 Generating graph rules

Since the goal for the rule learning tasks is to discover as many high-quality rules as possible, it is inevitable to explore the entire rule search space. We employ the well-trained policy network $\pi\,(a|s;\,\theta)$ to guide the rule search procedure to find high-quality graph rules efficiently by avoiding bad decisions at an early state of graph rule generation. The policy network $\pi\,(a|s;\,\theta)$ can generate a probability distribution over all possible actions when given a state, enabling the guidance of the search process by selecting a few certain actions with higher probabilities. Algorithm 2 describes the process for high-quality rules by combining $\pi\,(a|s;\,\theta)$ with beam search.

The algorithm takes the environment (i.e., data graph G) and the well-trained policy network $\pi\,(a|s;\,\theta)$ as input, and outputs a set Σ of graph rules obtained through the iterative process. We first identify all the frequent label triplets in the graph (Line 1), and find corresponding rules that take each label triplet as rule head with beam search by iterative deepening (Lines 2−18). During each expanding process, we obtain available actions for the current state s_t through interacting with the data graoh G (Line 8), then use the policy network $\pi\,(a|s;\,\theta)$ to evaluate these actions, and select the top-k actions with the highest probabilities for expansion (Line 9). Consequently, it generates a sequence of graph rules that have the same rule head and number of edges, but different patterns. Then, we filter out rules that have the same minimum DFS codes in patterns and add high-quality rules into Σ by evaluating them with support (Lines 13−16). In this case, the list currentRules contains all the high-quality graph rules at the current expanding step. Then, it sorts rules in currentRules by their support values and selects beam_width rules for the next expansion step. The process is repeated until it reaches max_step. All the graph rules are included in Σ.

Algorithm 2 Rule generating with policy network

```
Input: Data graph $G$, $\pi_\theta$, $\varepsilon$, max_step, beam_width, and top-$k$
Output: Set $\Sigma$ of graph rules
1: $\Sigma \leftarrow \varnothing$;
   $E$ ←collect all frequent label triplets in $G$;
2: for each label triplet $e \in E$ do
3:     $r_{\text{head}} \leftarrow$ set $e$ as a rule head;
4:     supp = support ($r_{\text{head}}$); beam = [($r_{\text{head}}$, supp)];
5:     for $i$ in range(max_step) do
6:         currentRules=list ( );
           MinDFSset = set ( );
7:         for ($s_t$, supp) in beam do
8:             availActionsList= $G$.Actions ($s_t$);
9:             $\mathcal{A} \leftarrow$ Rank actions in availActionsList with $\pi\,(a|s_t)$
               and select top-$k$ actions;
10:            for each action $a$ in $\mathcal{A}$ do
11:                $s_{t+1} \leftarrow s_t$+[$a$];
12:                mincode ← MinDFScode ($s_{t+1}$ [1:]);
13:                if ($r_{\text{head}}$, mincode) not in MinDFSset and $\varepsilon \leqslant$
                   support($s_{t+1}$) then
14:                    currentRules.Add (($s_{t+1}$, support ($s_{t+1}$)));
15:                    MinDFSset.Add (($r_{\text{head}}$, mincode));
16:                    $\Sigma \leftarrow \Sigma \cup \{s_{t+1}\}$;
17:        Sort currentRules with support of each rule;
18:        beam = currentRules[: beam_width];
19: return $\Sigma$
```

With Algorithm 2, we can generate as many high-quality graph rules as possible. The use of the policy network effectively reduces the search space, and redundant rules can be pruned by checking the minimum DFS codes of graph rules.

5 Experimental Study

In this section, we evaluate the effectiveness of the proposed method.

5.1 Experimental settings

Datasets. We conduct evaluations on three datasets, including Kinship[25], FB15k-237[26], and YAGO3-10[27]. Table 2 provides the statistical information of the datasets.

Compared algorithms. In our experiments, we compare five neural-based rule learning methods, including NeuralLP[17], Drum[28], RNNLogic[29], NCRL[30], and Rlogic[31]. These methods focus on learning chain-like rules, and can be used for reasoning

Table 2 Dataset information.

Dataset	Number of entities	Number of relations	Number of triples
Kinship	1044	25	5960
FB15k-237	14 541	237	310 116
YAGO3-10	123 182	37	1 089 040

on knowledge graphs. In addition, we adopt two RL-based methods, MINERVA[32] and DacKGR[33], for comparison. MINERVA and DacKGR are multi-hop reasoning models on knowledge graphs.

We also implement five embedding-based methods, including TransE[34], DistMult[35], RotatE[36], ConvE[37], and ComplEx[38]. These embedding methods are all relatively classic approaches for knowledge graph reasoning.

Evaluation metrics. To evaluate the graph rules discovered by our RL agent, we analyze the performance on the task of link prediction (predicting the target entity) in the knowledge graph. We adopt Mean Reciprocal Rank (MRR) and Hit@k as evaluation metrics, to maintain consistency with previous works. MRR is the average of the reciprocal ranks of all the correct triples. Hit@k refers to the proportion of correct triples ranked in the top n. The probability of each rule is 1 since our method learns deterministic rules.

Experimental setup of our method. Our algorithm GraphRulRL involves the use of a representation learning model and policy network. Specifically, the representation learning model is primarily used to obtain the representation of graph rules, which are then utilized as inputs for the policy network.

Therefore, we implement two different representation models: (1) LSTM is a single-layer LSTM model[21], and the final hidden state is set as the representation for the graph rule; (2) RGCN is a two-layer RGCN model[22], and the outputs after being processed by RGCN and an average pooling layer are set as the representation of the graph rule. The policy network $\pi(a|s;\theta)$ contains two fully-connected hidden layers, each followed by a rectifier non-linear layer. The output layer is normalized with a softmax function.

Consequently, GraphRulRL is divided into two versions in terms of two representation models, GraphRulRL$_{\text{lstm}}$ and GraphRulRL$_{\text{gcn}}$. To ensure consistency, the output dimensions for RGCN and LSTM are both set to 256. For the RL model, we set the parameters as follows: the learning rate is 0.0001, and the dimension of the embedding vector is 512. During the rule mining procedure, we set the parameters as follows: the beam_width is 100, top-k is 20, and the threshold of support is 25 over all datasets.

GraphRulRL is deployed in PyTorch framework, and trained on a Linux server powered by Intel Xeon 2.2 GHz and 64 GB memory. All parameters are tuned with the Adam optimizer. The experiments are repeated 5 times, and the reported values here are the averages.

5.2 Results

Experiment-1: Comparison against existing methods. The results are shown in Table 3. Our method demonstrates excellent performance on FB15k-237, achieving high scores on three metrics. Compared to other rule-based methods, it gets a higher hit@1 on YAGO3-10 and hit@10 on Kinship. However the overall effectiveness is not as good as those of the embedding-based models on YAGO3-10 and Kinship.

Table 3 Link prediction results on three datasets. Hit@k is in %. "OOM" denotes out of memory for short.

Method	Model	Kinship			FB15k-237			YAGO3-10		
		MRR	Hit@1	Hit@10	MRR	Hit@1	Hit@10	MRR	Hit@1	Hit@10
Embedding	TransE	0.32	9.0	74.3	0.32	23	51.0	0.36	25.1	58.0
	ConvE	0.83	77.0	96.7	0.32	23.7	50.1	0.44	35.4	61.6
	RotatE	0.65	50.4	93.2	0.34	24.1	53.3	0.49	40.2	67.0
	DistMult	0.35	18.9	75.5	0.24	15.5	41.9	0.34	24.3	53.3
	Complex	0.42	24.2	81.2	0.25	15.8	42.8	0.34	24.8	54.9
Rule	NeuralLP	0.62	47.2	91.1	0.24	16.0	39.9	OOM	OOM	OOM
	DRUM	0.58	42.3	90.1	0.25	18.7	37.4	OOM	OOM	OOM
	RNNLogic	0.6	42.9	94.6	0.35	25.8	53.3	OOM	OOM	OOM
	NCRL	0.6	46.3	90.5	0.30	20.9	47.3	0.38	27.4	53.6
	Rlogic	0.58	48.6	91.4	0.31	20.3	50.1	0.36	25.2	50.4
RL-based	MINERVA	0.64	48.5	94.1	0.27	19.6	43.5	0.15	10.6	23.7
	DacKGR	0.54	43.5	72.8	0.35	29.4	47.5	OOM	OOM	OOM
Ours	GraphRulRL$_{\text{lstm}}$	0.63	46.4	95.6	0.42	32.8	60.1	0.42	35.8	52.0
	GraphRulRL$_{\text{gcn}}$	0.63	46.9	95.1	0.43	33.3	60.9	0.40	33.8	50.1

Through analyzing the characteristics of these datasets, we can find that FB15k-237 contains more kinds of edges, allowing the representation models to extract more effective information from graph rules.

Additionally, $\text{GraphRulRL}_{\text{gcn}}$ shows a certain advantage over $\text{GraphRulRL}_{\text{lstm}}$ on the FB15k-237. However, $\text{GraphRulRL}_{\text{lstm}}$ performs slightly better on the other two datasets. This indicates that the encoding effectiveness is influenced by the characteristics of the data graphs, for that FB15k-237 has more types of edge relations than the other two, which demonstrates more diverse structures in the data graph.

Experiment-2: Performance with varying sizes of the bodies of graph rules. We analyze the changes in the number of graph rules under the different sizes of the rule body, i.e., the number of edges in the rule body. As shown in Fig. 3, it is evident that our method can obtain as many high-quality graph rules as possible on all three datasets when the support is set to 25. As the size of the rule body increases, the number of rules first increases and then decreases. There is a notably higher quantity of graph rules when the rule body size is 5, which suggests that such structures of graph rules are more common in graphs.

Additionally, we find that $\text{GraphRulRL}_{\text{lstm}}$ can mine more graph rules on FB15k-237, while there is not much difference in the number of rules mined by the two methods on the other two datasets. This illustrates that this method is significantly influenced by the datasets.

Experiment-3: Performance with varying the numbers of support. We compared the MRR metrics on three datasets with different support values varying from 10−90. As shown in Figs. 4a and 4b, the results of MRR on FB15k-237 and Kinship show a decreasing trend within the increasing of support. This indicates that support, as an evaluation metric of graph rules, has a significant impact on rule mining. However, there is no noticeable change YAGO3-10. This is because YAGO3-10 is large in scale, and the graph rules of these structures are very common. However, when we set support to 120 or above on YAGO3-10, we find that all three metrics decreased to 0.36, 32.7%, and 41.6% for MRR, Hit@1, and Hit@10, respectively, with $\text{GraphRulRL}_{\text{lstm}}$ (not shown). This indicates that with the increase of the threshold of support, the number of matches of graph rules that meet the criteria decreases. Therefore, setting an appropriate threshold of support is an issue that needs careful consideration.

Experiment-4: Analysis for representation models. From the results in Table 3, $\text{GraphRulRL}_{\text{lstm}}$ demonstrates relatively superior in performance than $\text{GraphRulRL}_{\text{gcn}}$ on Kinship and YAGO3-10, while $\text{GraphRulRL}_{\text{gcn}}$ exhibits better performance on FB15k-

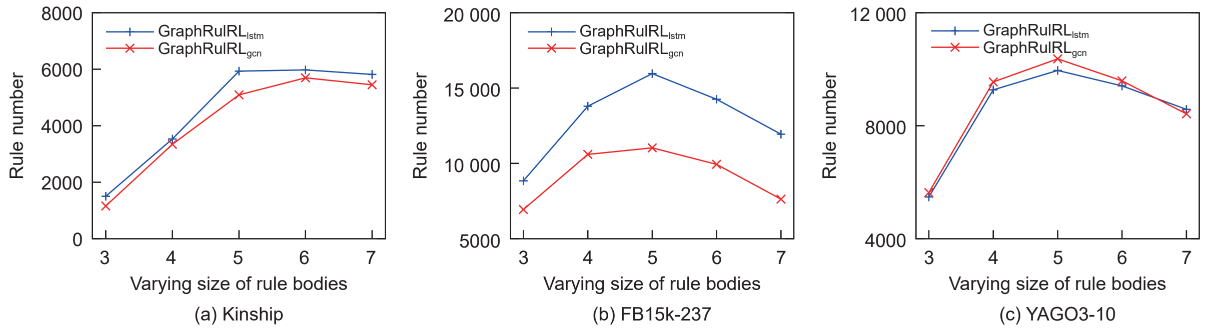

Fig. 3 Distribution of graph rules with different sizes of rule bodies.

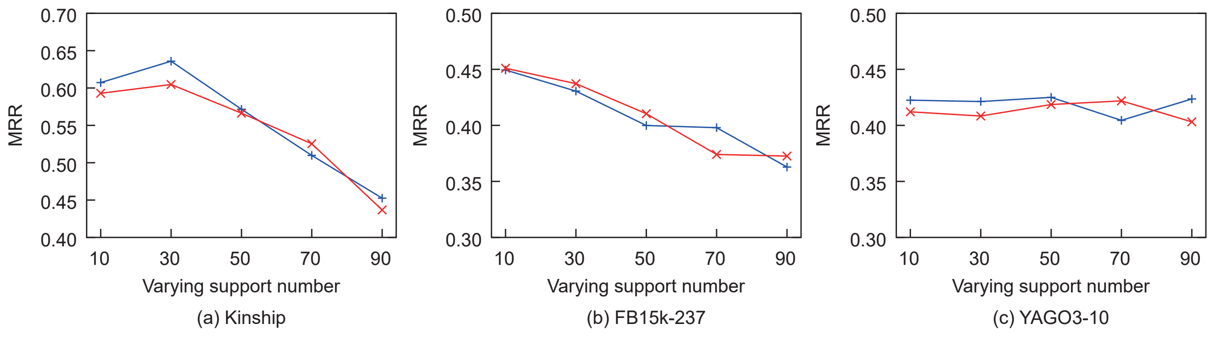

Fig. 4 Performance with varying support numbers.

237. By scrutinizing the characteristics of these datasets, We find that FB15k-237 has more kinds of edge relations, whereas Kinship and YAGO3-10 have significantly fewer edge relations. Therefore, RGCN has better performance on graphs with more kinds of edges. In addition, Fig. 3b shows that GraphRulRL$_{\text{gcn}}$ gets fewer graph rules than GraphRulRL$_{\text{lstm}}$ on FB15k-237, but the results in Table 3 show the superior performance of GraphRulRL$_{\text{gcn}}$, which also demonstrates the effectiveness of RGCN on graphs with more diverse structures.

Figures 3a and 3c show that there is little difference in the number of generating graph rules between GraphRulRL$_{\text{lstm}}$ and GraphRulRL$_{\text{gcn}}$. And the results in Table 3 shows that GraphRulRL$_{\text{lstm}}$ is relatively superior in performance than GraphRulRL$_{\text{gcn}}$ on kinship and YAGO3-10. This indicates that LSTM works well on graphs with less types of edge relations. Therefore, we suggest to choose GraphRulRL$_{\text{gcn}}$ when there are many types of edges, and choose GraphRulRL$_{\text{lstm}}$ when there is less types of edges. In fact, both GraphRulRL$_{\text{lstm}}$ and GraphRulRL$_{\text{gcn}}$ exhibit good performance in generating graph rules.

Experiment-5: Case study of generated graph rules. We present three graph rules mined from FB15k-237 in Fig. 5. The first rule ρ_c indicates that if y directs film x and wins award z, then x may be nominated for award z. The second rule ρ_d indicates location y has characteristics of being a venue for wedding x when location y becomes the choice of marrying travelers z during the period y'. The third rule ρ_e indicates that the educational institution y is contained in place x, while student z lives in x and receives education from y, and y uses z' as a currency unit for the operating income.

6 Conclusion

This paper introduces a new method for learning graph rules using a policy network based RL approach, denoted as GraphRulRL. GraphRulRL adopts DFS codes to convert graph rules into sequences of ordered edges. Consequently, it formulates the problem of graph rule learning into a sequential decision-making problem that can be solved by RL. GraphRulRL first trains the policy network of RL for graph rule learning, and the reward function for the policy network considers support with anti-monotonicity as evaluating metrics of graph rules. Then it develops a rule-generating algorithm that combines the well-trained policy network with beam search for iterative searching to generate as many high-quality graph rules as possible. The experimental results demonstrate that GraphRulRL has excellent ability in generating high-quality graph rules, and also exhibits excellent performance in terms of effectiveness.

However, this study also has some limitations. For instance, it uses support as the evaluation metric, which is less sensitive to certain infrequent rules. Exploring how to effectively combine statistical metrics with machine learning methods is also a meaningful direction for further research.

Acknowledgment

This work was supported by the Hainan Tropical Ocean Institute Yazhou Bay Innovation Research Institute Major Science (No. 2023GKF-122), the Technology Plan Project (No. 2023CXYZD001), and the Key Research and Development Projects in Hainan Province (No. ZDYF2024SHFZ051).

References

[1] W. Fan, X. Liu, P. Lu, and C. Tian, Catching numeric inconsistencies in graphs, in *Proc. SIGMOD Conf. 2018*, Houston, TX, USA, 2018, pp. 381–393.

[2] W. Fan, Y. Wu, and J. Xu, Functional dependencies for graphs, in *Proc. SIGMOD Conf. 2016*, San Francisco, CA, USA, 2016, pp. 1843–1857.

[3] W. Fan and P. Lu, Dependencies for graphs, *ACM Trans. Database Syst. TODS*, vol. 44, pp. 1–40, 2019.

[4] W. Fan, X. Wang, Y. Wu, and J. Xu, Association rules with graph patterns, *Proc. VLDB Endow.*, vol. 8, no. 12,

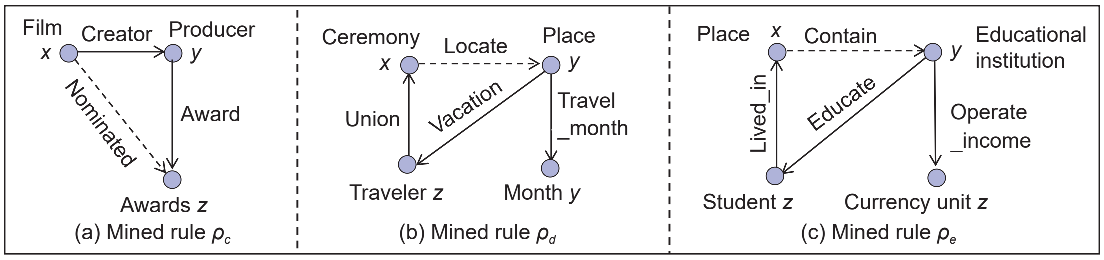

Fig. 5 Graph rules minded from FB15k-237.

pp. 1502–1513, 2015.

[5] W. Fan, C. Hu, X. Liu, and P. Lu, Discovering graph functional dependencies, in *Proc. SIGMOD Conf. 2018*, Houston, TX, USA, 2018, pp. 427–439.

[6] W. Fan, W. Fu, R. Jin, P. Lu, and C. Tian, Discovering association rules from big graphs, *Proc. VLDB Endow.*, vol. 15, no. 7, pp. 1479–1492, 2022.

[7] N. Lao, J. Zhu, X. Liu, Y. Liu, and W. W. Cohen, Efficient relational learning with hidden variable detection, in *Proc. Conf. NIPS 2010*, Vancouver, Canada, 2010, pp. 1234–1242.

[8] Z. Du, C. Zhou, M. Ding, H. Yang, and J. Tang, Cognitive knowledge graph reasoning for one–shot relational learning, arXiv preprint arXiv: 1906.05489, 2019.

[9] Zh. Zhu, Z. Zhang, L.-P. A. C. Xhonneux, and J. Tang, Neural Bellman-Ford networks: A general graph neural network framework for link prediction, in *Proc. Thirty-Fifth Conference on Neural Information Processing Systems*, Virtual Event, 2021, doi:10.48550/arXiv.2106. 06935.

[10] W. W. Cohen, TensorLog: A differentiable deductive database, arXiv preprint arXiv:1605.06523, 2016.

[11] W. Xiong, T. Hoang, and W. Y. Wang, DeepPath: A reinforcement learning method for knowledge graph reasoning, in *Proc. 2017 Conf. Empirical Methods in Natural Language Processing*, Copenhagen, Denmark, 2017, pp. 564–573.

[12] C. Meilicke, M. W. Chekol, M. Fink, and H. Stuckenschmidt, Reinforced anytime bottom up rule learning for knowledge graph completion, arXiv preprint arXiv:2004.04412, 2020.

[13] X. Yan and J. Han, gSpan: Graph-based substructure pattern mining, in *Proc. IEEE Int. Conf. Data Mining,* Maebashi City, Japan, pp. 721–724.

[14] M. Kuramochi and G. Karypis, Finding frequent patterns in a large sparse graph, in *Proc. 2004 SIAM Int. Conf. Data Mining,* Philadelphia, AR, USA, 2004, pp. 345–356.

[15] L. Galárraga, C. Teflioudi, K. Hose, and F. M. Suchanek, Fast rule mining in ontological knowledge bases with AMIE+, *VLDB J.*, vol. 24, no. 6, pp. 707–730, 2015.

[16] K. K. Teru, E. G. Denis, and W. L. Hamilton, Inductive relation prediction by subgraph reasoning, in *Proc. ICML 2020*, Virtual Event, 2020, pp. 9448–9457.

[17] F. Yang, Z. Yang, and W. W. Cohen, Differentiable learning of logical rules for knowledge base reasoning, arXiv preprint arXiv:1702.08367, 2017.

[18] Y. Yang and L. Song, Learn to explain efficiently via neural logic inductive learning, arXiv preprint arXiv: 1910.02481, 2020.

[19] Q. Wang and C. Tang, Deep reinforcement learning for transportation network combinatorial optimization: A survey, *Knowl. Based Syst.*, vol. 233, p. 107526, 2021.

[20] X. V. Lin, R. Socher, and C. Xiong, Multi-hop knowledge graph reasoning with reward shaping, in *Proc. 2018 Conf. Empirical Methods in Natural Language Processing.* Brussels, Belgium, 2018, pp. 3243–3253.

[21] S. Hochreiter and J. Schmidhuber, Long short-term memory, *Neural Comput.*, vol. 9, pp. 1735–1780, 1997.

[22] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. van den Berg, I. Titov, and M. Welling, Modeling relational data with graph convolutional networks, arXiv preprint arXiv: 1703.06103, 2017.

[23] X. Liu, H. Pan, M. He, Y. Song, X. Jiang, and L. Shang, Neural subgraph isomorphism counting, in *Proc. 26th ACM SIGKDD Int. Conf. Knowledge Discovery* & *Data Mining*, Virtual Event, 2020, pp. 1959 – 1969.

[24] R. J. Williams, Simple statistical gradient-following algorithms for connectionist reinforcement learning, *Mach. Learn.*, vol. 8, no. 3, pp. 229–256, 1992.

[25] S. Kok and P. Domingos, Statistical predicate invention, in *Proc. 24th Int. Conf. Machine learning,* Corvalis, OR, USA, 2007, pp. 433–440.

[26] K. Toutanova and D. Chen, Observed versus latent features for knowledge base and text inference, in *Proc. 3rd Workshop on Continuous Vector Space Models and their Compositionality*, Beijing, China, 2015, pp. 57–66.

[27] F. M. Suchanek, G. Kasneci, and G. Weikum, Yago: A core of semantic knowledge, in *Proc. 16th Int. Conf. World Wide Web*, Banff, Canada, 2007, pp. 697–706.

[28] A. Sadeghian, M. Armandpour, P. Ding, and D. Z. Wang, DRUM: End-to-end differentiable rule mining on knowledge graphs, in *Proc. NIPS 2019*, Vancouver, Canada, 2019, pp. 15321–15331.

[29] M. Qu, J. Chen, L. A. C. Xhonneux, Yoshua Bengio, Jian Tang, RNNlogic: Learning logic rules for reasoning on knowledge graphs, arXiv preprint arXiv:2010.04029, 2021.

[30] K. Cheng, N. K. Ahmed, Y. Sun, Neural compositional rule learning for knowledge graph reasoning, in *Proc. ICLR2023*, Kigali, Rwanda, arXiv preprint arXiv: 2303.03581, 2023.

[31] K. Cheng, J. Liu, W. Wang, and Y. Sun, RLogic: Recursive logical rule learning from knowledge graphs, in *Proc. 28th ACM SIGKDD Conf. Knowledge Discovery and Data Mining*, Washington, DC, USA, 2022, pp. 179–189.

[32] R. Das, S. Dhuliawala, M. Zaheer, L. Vilnis, I. Durugkar, A. Krishnamurthy, A. Smola, and A. McCallum, Go for a walk and arrive at the answer: Reasoning over paths in knowledge bases using reinforcement learning, arXiv preprint arXiv: 1711.05851, 2017.

[33] X. Lv, X. Han, L. Hou, J. Li, Z. Liu, W. Zhang, Y. Zhang, H. Kong, and S. Wu, Dynamic anticipation and completion for multi-hop reasoning over sparse knowledge graph, in *Proc. 2020 Conf. Empirical Methods in Natural Language Processing* (*EMNLP*), Virtual Event, 2020, pp. 5694–5703.

[34] A. Bordes, N. Usunier, A. García–Durán, J. Weston, and O. Yakhnenko, Translating embeddings for modeling multi-relational data, in *Proc. NIPS 2013*, Lake Tahoe, Nevada, USA, 2013, pp. 2787–2795.

[35] B. Yang, W. T. Yih, X. He, J. Gao, and L. Deng, Embedding entities and relations for learning and inference in knowledge bases, arXiv preprint arXiv: 1412.6575, 2015.

[36] Z. Sun, Z. Deng, J. Nie, and J. Tang, Rotate: Knowledge graph embedding by relational rotation in complex space, arXiv preprint arXiv:1902.10197, 2019.

[37] T. Dettmers, P. Minervini, P. Stenetorp, and S. Riedel, Convolutional 2D knowledge graph embeddings, in *Proc. AAAI Conf. Artif. Intell.*, New Orleans, LA, USA, pp. 1811–1818.

[38] T. Trouillon, J. Welbl, S. Riedel, É. Gaussier, and G. Bouchard, Complex embeddings for simple link prediction, arXiv preprint arXiv: 1606.06357, 2016.

Wenjun Wang received the PhD degree from Peking University, China in 2006. He is currently a professor at College of Intelligence and Computing, Tianjin University, China, the chief expert of major projects of the National Social Science Foundation (China), and the director of the Tianjin Engineering Research Center of Big Data on Public Security. His research interests include computational social science, large-scale data mining, intelligence analysis, and multilayer complex network modeling.

Xueli Liu received the PhD degree from Harbin Institute of Technology, China in 2018. She is currently an associate professor at College of Intelligence and Computing, Tianjin University, China. Her research interests include big data computing, graph association analysis, and deep learning interpretation.

Jun Wang received the MS degree in fundamental mathematics from Beijing Institute of Technology, China in 2007. From 2007 to 2016, he worked at Nankai University as an experimenter. Currently, he is engaged at Technical Research & Development Department, Tianjin University, China, and a PhD candidate at College of Intelligence and Computing, Tianjin University, China. His research interests include pattern recognition, face recognition, machine learning, and parallel programming.

Zhenzhen Mai received the MEng degree in electronics and communications engineering from Shandong University, China in 2020. She is currently a PhD candidate at College of Intelligence and Computing, Tianjin University, China. Her research interests mainly focus on big data computing, data mining, as well as rule learning and its applications in big data.

Xiaoyang Feng received the BEng degree in software engineering from Zhengzhou University, China in 2022. He is currently a master student at College of Intelligence and Computing, Tianjin University, China. His research interests mainly focus on rule learning, knowledge graph reasoning, as well as data mining and its applications in big data.

Wenzhi Fu received the MEng degree in computer science and technology from Beihang University, China in 2019. He is currently a PhD candidate at School of Informatics, University of Edinburgh, UK. His research focuses on big data processing, data mining, data modelling, and heterogeneous data management.