Symbolic Reinforcement Learning Framework with Incremental Learning of Rule-based Policy

Kinjal Basu[1,2], Elmer Salazar[1], Huaduo Wang[1], Joaquín Arias[3], Parth Padalkar[1] and Gopal Gupta[1]

[1]University of Texas at Dallas, Richardson, USA
[2]IBM Research, NY, USA
[2]CETINIA, Universidad Rey Juan Carlos, Spain

Abstract

In AI research, Relational Reinforcement Learning (RRL) is a vastly discussed domain that combines reinforcement learning with relational learning or inductive learning. One of the key challenges of inductive learning through rewards and action is to learn the relations incrementally. In other words, how an agent can closely mimic the human learning process. Where we, humans, start with a very naive belief about a concept and gradually update it over time to a more concrete hypothesis. In this paper, we address this challenge and show that an automatic theory revision component can be developed efficiently that can update the existing hypothesis based on the rewards the agent collects by applying it. We present a symbolic reinforcement learning framework with the automatic theory revision component for incremental learning. This theory revision component would not be possible to build without the help of a goal-directed execution engine of answer set programming (ASP) - s(CASP). The current work has demonstrated a proof of concept about the RL framework and we are still working on it.

Keywords

Logic Programming, Reinforcement Learning, Incremental Learning, Goal Directed Execution (GDE)

1. Introduction

One of the key goals of AI is to teach machines how to mimic human learning techniques. Learning through examples is one of those techniques that human employs in their day-to-day life. We often generate beliefs by seeing a very small amount of examples and, gradually, when we encounter more examples, we try to reason using our existing beliefs. If we succeed, our beliefs get stronger, however, if we fail, we update our beliefs to accommodate the new example. For instance, let's say a child, who lives in a tropical area, holds a glass of hot water. Holding a glass of hot water in hot weather increases the child's discomfort. With this experience, the child quickly learns to not hold a glass of hot water in hot weather. However, when the child experiences very cold weather, then the same hot water glass may feel comforting to hold. He/she then may update the prior belief to "do not hold a glass of hot water unless the surrounding weather is cold". This action reward-based learning technique of humans

2nd Workshop on Goal-directed Execution of Answer Set Programs (GDE'22), August 1, 2022
Kinjal.Basu@ibm.com (K. Basu); ees101020@utdallas.edu (E. Salazar); Huaduo.Wang@utdallas.edu (H. Wang); joaquin.arias@urjc.es (J. Arias); Parth.Padalkar@utdallas.edu (P. Padalkar); gupta@utdallas.edu (G. Gupta)
0000-0003-4148-311X (J. Arias)

closely relates to Reinforcement Learning (RL) in the realm of Machine Learning. In RL, while exploring an environment, an agent gets positive/negative rewards based on its actions. From these rewards, the agent may learn a policy to take better actions in its operating environment. Symbolic policy is one of the types of policies that are learned as a set of logic rules that can be applied next time by the agent if the same situation is encountered. A way of learning these policies is Inductive Logic Programming (ILP) which uses the environment state and agent's action along with the rewards received from the environment to learn the symbolic rules inductively. Then, with the help of a reasoner/solver, the agent applies the learned policies to another state in the environment and collects rewards. Again, after receiving multiple rewards, the agent employs ILP to learn a new set of policies. In this way, the cycle of learning policies and applying them goes on, and the agent's understanding about the environment improves with more experience. Recent works [1] using this idea show really good performance on text-based games. A key issue here is about the ILP algorithms. Most of the algorithms can learn new rules but are not capable of updating the existing rules. Like a human, we expect an RL agent to learn the policies incrementally by updating the existing beliefs and learning new policies.

In this paper, we introduce a symbolic reinforcement learning framework that can perform inductive learning using ILP to learn new rules as well as use incremental learning to update the existing rules (policy). We believe our framework closely mimics the human learning mechanism where we can learn new concepts in parallel with updating the existing beliefs. Note also that humans perform non-monotonic reasoning with the help of default rules and exceptions. In other words, we quickly come to a conclusion by seeing only a few sets of examples and later correct our beliefs after encountering an exceptional scenario. As we can easily model these defaults and exceptions using Answer Set Programming (ASP), a logic programming paradigm, we heavily rely on ASP for symbolic policy representation. By representing the policy in ASP, an agent can also reason using an ASP solver to get a possible action given a state. The novelty of our work is the incremental learning by revising the existing hypothesis and that is only possible if we use a goal-directed implementation of ASP solver—s(CASP). With the s(CASP) system, we can get a proof tree of any reasoning task it performs. This tree can be analyzed and used to update the rules, in a manner very similar to humans. We discuss more details about our approach in the later sections.

2. Background

2.1. Answer Set Programming

An answer set program is a collection of rules of the form -

$$l_0 \leftarrow l_1,\ \ldots,\ l_m,\ not\, l_{m+1},\ \ldots,\ not\, l_n.$$

Classical logic denotes each l_i is a literal [2]. In an ASP rule, the left hand side is called the *head* and the right-hand side is the *body*. Constraints are ASP rules without *head*, whereas facts are without *body*. The variables start with an uppercase letter, while the predicates and the constants begin with a lowercase. We will follow this convention throughout the paper.

The semantics of ASP is based on the stable model semantics of logic programming [3]. ASP supports *negation as failure* [2], allowing it to elegantly model common sense reasoning, default rules with exceptions, etc.

2.2. s(CASP)

s(CASP) [4] is a query-driven, goal-directed implementation of ASP that includes constraint solving over reals. Goal-directed execution of s(CASP) is indispensable for automating common-sense reasoning, as traditional grounding and SAT-solver based implementations of ASP may not be scalable. There are three major advantages of using the s(CASP) system: (i) s(CASP) does not ground the program, which makes our framework scalable, (ii) it only explores the parts of the knowledge base that are needed to answer a query, and (iii) it provides natural language justification (proof tree) for an answer [5]. The key component of the paper - automatic theory revision has been developed by exploiting the justification from s(CASP). So, the s(CASP) is indispensable for our work.

2.3. ILP: FOLD Family of Algorithms

Inductive Logic Programming (ILP) [**?**] is a sub-field of machine learning that learns models in the form of logic programming clauses comprehensible to humans. This problem is formally defined as:
Given

1. A background theory B, in the form of an extended logic program, i.e., clauses of the form $h \leftarrow l_1, ..., l_m,\ not\ l_{m+1}, ...,\ not\ l_n$, where $h, l_1, ..., l_n$ are positive literals and *not* denotes *negation-as-failure* (NAF) as described in [**?**]. For reasons of efficiency, we restrict B to be stratified [**?**].
2. Two disjoint sets of ground target predicates E^+, E^- known as positive and negative examples, respectively.
3. A hypothesis language of function free predicates L, and a refinement operator ρ under θ-subsumption [**?**] (for more details see [**?**]). The hypothesis language L is also assumed to be stratified.

Find a set of clauses H such that:

1. $\forall e \in\ E^+,\ B \cup H \models e$.
2. $\forall e \in\ E^-,\ B \cup H \not\models e$.
3. $B \wedge H$ is consistent.

The target predicate is the predicate whose definition we want to learn as a stratified normal logic program. The positive and negative examples are grounded target predicates, i.e., suppose we want to learn the concept of which creatures can *fly*, then we will give positive examples $E^+ = \{$`fly(tweety)`, `fly(sam)`, `...` $\}$ and negative examples $E^- = \{$`fly(kitty)`, `fly(polly)`, `...` $\}$, where `tweety`, `sam`, ..., are names of creatures that can fly, and `kitty`, `polly`, ..., are names of creatures that cannot fly.

The FOIL algorithm by [6] is a popular top-down inductive logic programming algorithm for classification. The FOLD algorithm by [7] is a novel top-down algorithm inspired by FOIL that

learns default rules along with exceptions that closely model human thinking. The FOLD-R++ algorithm by Wang and Gupta is a new scalable ILP algorithm that builds upon the FOLD algorithm to deal with the efficiency and scalability issues of the FOIL and FOLD algorithms. It can deal with mixed type (numerical and categorical) data and generate much simpler rule sets compared to its predecessors. The FOLD-R++ algorithm by [8] is also competitive in performance with the widely-used XGBoost and Multi-Layer Perceptron(MLP) algorithm. The FOLD-RM algorithm by [9] is built upon FOLD-R++ to deal with multi-class classification tasks while keeping all the features.

3. RL Framework

In this section we present our Symbolic Reinforcement Learning framework that can learn the hypothesis incrementally. Figure 1 illustrates our framework showing how an agent learns the rules by interacting with the environment. In the diagram, *State* (S_t) represents an instance of the environment given a time stamp 't', *Action* (A_t) means the agent's act at time 't', and *Reward* (R_t) shows the award the agent achieves from the environment at time 't' by taking the action A_t. With iterative interaction between the environment and agent, the agent learns the world rules and performs better next time. In the given framework, the rule learning is divided into two categories - (i) learning new rules, and (ii) updating the existing rules. The *Action Generator* component utilize the power of s(CASP) engine to generate the actions based on a action strategy. The agent's actions can be of two types - (i) actions that helps the agent to explore the environment, and (ii) exploit the learned rules to perform better in the environment. In the framework, the exploration actions are symbolized as A_t^R, where 'R' stands for *random*. Using random action generator we can built a very naive explorer, however, we can train a neural agent to do the exploration very efficiently. A_t^S are the actions that exploits the learned hypothesis (here 'S' stands for *symbolic*). Based on these two types of actions we have two different rule learning strategies. Using the <S_t, R_t, A_t^R>, the agent learns new policy by employing the FOLD family of ILP algorithms. Additionally, when we apply the learned rules to generate an action (A_t^S) and not get the reward that was expected, we update the existing rules by revising them and that is done inside the *Theory Revision* component. Incorporating this automatic *Theory Revision* component (discussed in the next Section) is the main novelty of the paper.

4. Finding Defeaters

As discussed earlier, we used s(CASP), an ASP solver, to build our *'Action Generator'* module. The s(CASP) system is a top-down, goal-directed system. This means that for each successful query, it finds a proof. If we expected this query to fail (for example, because the reward for the previously chosen action using the existing hypothesis is negative), then we can try to figure out how to change the rules such that the query will fail instead of succeeding. This change essentially *defeats* the *argument* that led to the query's success and is called a *defeater*. Now, we give an example to elaborate the problem statement followed by the solution that can perform automatic theory revision [10].

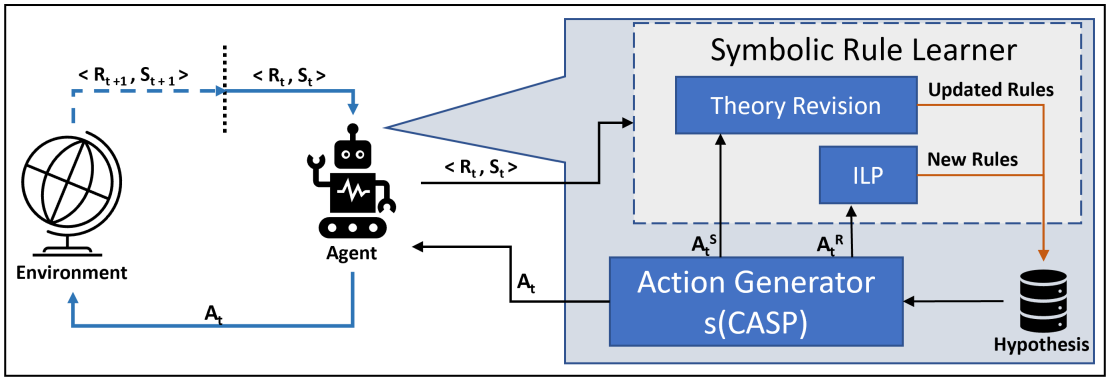

Figure 1: Symbolic RL framework

4.1. An Example

Consider a house that has sensors installed to protect it from fires and floods. To protect from fire, a fire sensor is installed that will automatically turn on water sprinklers (also installed in the house) if fire is detected. Likewise, a water leak detection sensor in the house will automatically turn off water supply, if no one is present in the house and water is sensed on the floor/carpet. The following answer set programs models these rules:

```
fireDetected :- fire.
turnSprinklerOn :- fireDetected.
sprinklerOn :- turnSprinklerOn.
water :- sprinklerOn.

sprinklerOff :- waterSupplyOff.
waterSupplyOff :- turnWaterSupplyOff.
turnWaterSupplyOff :- houseEmpty, waterLeakDetected.
waterLeakDetected :- water.

houseFloods :- water, not waterSupplyOff.
houseBurns :- fireDetected, SprinklerOff.
houseSafe :- not houseFloods, not houseBurns.
```

The program is self-explanatory. It models fluents (`sprinklerOn`, `sprinklerOff`, `waterLeakDetected`, `fireDetected`, `fire`, `water`, `houseEmpty`) and actuators (`turnWaterSupplyOff`, `turnSprinklerOn`). For simplicity, time is not considered, though it must be taken into account if we want to model the system faithfully. This is because there will always be a time lag as actuators are activated and fluents change in response to them. Note that 'fire' means fire broke out in the house and 'water' means that a water leak occurred in the house.

Given the theory above, if we add the fact `fire.` to it, we will find that the property `houseBurns` defined above will succeed. This is because occurrence of fire eventually leads to

sprinklers being turned on, which causes water to spill on the floor, which, in turn, causes the flood protection system to turn on and turn off the water supply. We want `houseBurns` to fail. To ensure that it fails, we have to recognize that water supply should indeed be turned off due to a water leak in an empty house *unless* fire is present:

```
turnWaterSupplyOff :- houseEmpty, waterLeakDetected,
                      not fireDetected.
```

So, a simple patch to the theory shown above will ensure that houseBurns fails in all situations. By adding *not fireDetected* we are subtracting knowledge preventing `houseBurns` from succeeding. Likewise, note that the house can be flooded if water leaks ("water." is added as a fact) and people are present in the house. Analysis of the proof tells us the offending conjuct is (`water, not houseEmpty`). Since neither `water` nor `not houseEmpty` can be forced to be false, we will have to detect such a situation and raise an alarm. So here the idea will be to sound an alarm that will alert people present in the house. Thus, in situations where a successful proof cannot be falsified by altering a theory, we may want to identify critical components of the proof (via Craig interpolation perhaps) and identify their conjunction as a distinct condition.

If we want our house to be safe against theft as well, then we need to *augment* the theory with rules that will automatically lock down the house if the sensor indicates that no one is present in the house. Note that to solve the theory revision problem we should be able to reason with success as well as failure of proofs. Answer set programming provides a good mechanism for doing so, as it allows actions to be taken if a proof fails within the theory itself.

We would like to detect revisions to a theory automatically. We outline methods to do so below.

4.2. Automatic Theory Revision

The "system" we want to check needs to first be encoded as a s(CASP) program. Currently, only propositional programs are supported. Once the "system" is encoded, we must identify state knowledge that the "system" does not control. This knowledge will become abducibles. An abducible is a proposition for which we have the choice of making it true or false, as needed. Finally, any requirements or constraints we wish to enforce need to be encoded.

Figure 2 shows the ASP program of the example discussed above. We can run the program with s(CASP), querying what we want to fail. In this example, that would be "`?- burndown.`". We will use s(CASP)'s "--tree" option to get the proof tree, and look for propositions with rules that we can change. In this example those propositions are "`turn_sprinkler_on`" and "`turn_water_off`".

Once the subtrees we can use are identified, we look for a proposition (or its negation) that is false in the model associated with the tree, provided by s(CASP). If the subtree or any of it's ancestors are not dependent on that proposition, it becomes a suggested defeater. So, for our example (figure 3 presents the s(CASP) proof tree of the program) we have the following possible defeaters:

- For rule: `turn_water_off :- leak, not home.`
  - `not fire`
- For rule: `turn_sprinkler_on :- fire.`

- `not water_off` OR
- `not leak` OR
- `home`

Each suggestion, when used to modify the program, will ensure that the proof cannot succeed. This makes no guarantee the query will not still succeed, nor that the changes will make sense according to our interpretation of the program. To combat the former case, we run the above algorithm for every tree the query causes. Each of these trees represent a different way the query can succeed. The generated suggestions are grouped together with their proof tree and model and used to generate a knowledgebase to be used with s(CASP). This knowledgebase can then be combined with some common sense and domain specific knowledge to reason about the "best" defeater. Once we have the "best" defeaters, we can modify the program and try again. After all, the change itself may introduce a new way for the query to succeed.

The second case, of suggestions that do not make sense, can be easily encountered. In the example above, we can ensure this proof fails by ensuring `turn_sprinkler_on` fails. This is counterproductive. We know that by not turning on the sprinklers when there is a fire, the house will always burn down – regardless of the rest of the state. To make a more intelligent choice, the possible defeaters, along with the associated model and the original program, are combined, as data, with another s(CASP) program. The purpose of this program is to filter out the irrelevant defeaters. For the above example, we may add a rule that does not keep the second set of defeaters. There are three parts to this stage. First, is the defeaters, models, and program as stated above. These are presented as data in the program to be analysed and processes.

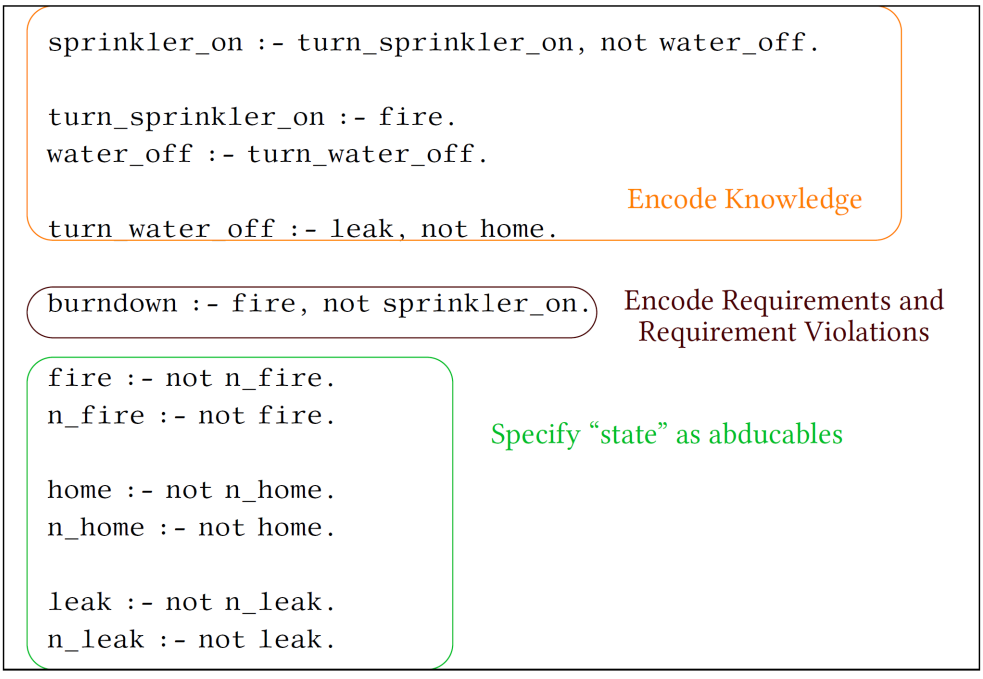

Figure 2: Example: ASP program

Secondly, a driver program that contains non-domain specific knowledge that contains the following query: `?- suggest(A).` As of this writing, the driver program provides a default implementation for suggest that simply provides each defeater, as is. The third part is the domain specific logic. This program is defined for the specific system and defines `suggest/1`. For the above example, we can provide the following implementation for `suggest/1`:

```
suggest(A) :- defeaters(_,Defs), find_suggestions(Defs,A).

find_suggestions([H|T], H) :- H=defeater(proof(neg,_,_),_).
find_suggestions([H|T], H) :- H=defeater(proof(pos,Pred,_),_),
                              Pred\=turn_sprinkler_on.
find_suggestions([_|T], S) :- find_suggestions(T,S).
```

The first rule defines `suggest/1`. The call to `defeaters/2`, provided when generating the knowledgebase, gets the list of defeaters for a proof tree. The `find_suggestions/2` call loops through these defeaters binding `A` to a valid defeater. Each binding of `A` is a possible way of making the proof fail. The other three rules define what we think are valid suggestions.

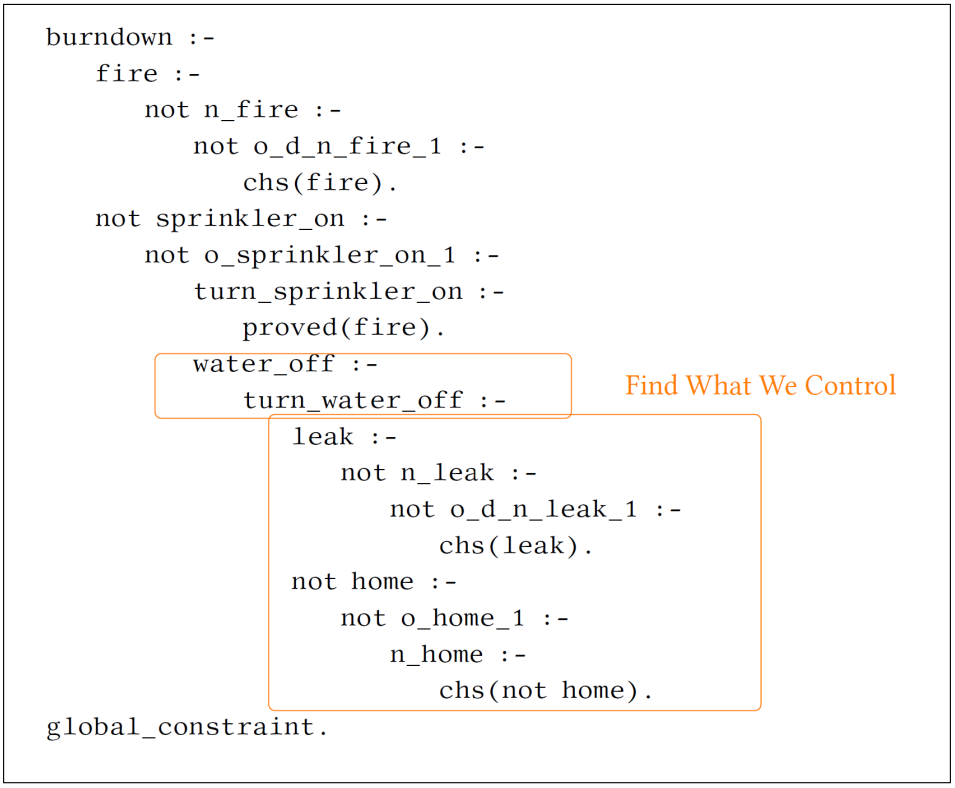

Figure 3: s(CASP) generated proof tree

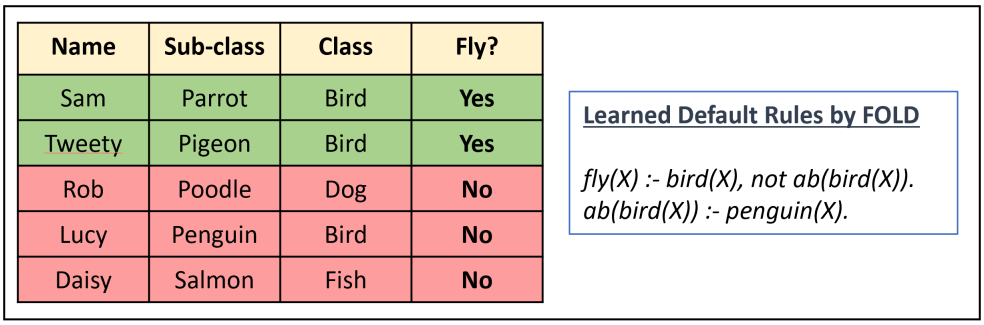

Figure 4: Learning default rules using FOLD algorithm for *"which animal can fly?"*

The third rule is a simple recursive case that allows us to traverse the list of defeaters. The decision is being made by the first two rules. The first rule for `find_suggestions/2` allows suggestions for duals (the negation of a proposition). The second rule considers positive literals (propositions without negation). However, It only accepts such defeaters if the proposition is not `turn_sprinkler_on`. Since we know that `turn_sprinkler_on` can only be true if there is a fire, by disallowing it from being made false we are enforcing the constraint "The sprinkler must turn on when there is a fire". Using this definition of `suggest/1`, the second set of defeaters (from the output given above) will not be given. This gives us only the suggestion:

- For rule: `turn_water_off :- leak, not home.`
  - `not fire`

This answer makes the most sense according to our interpretation of the code.

The above example illustrates how the system behaves when a proposition needs to be made false. However, it is possible that instead of a falsifying a proposition, we will want to falsify its dual.

5. An Example

Next, we provide an example to elaborate the process of rule learning by an agent and then revising them through more experience. Let us assume, an RL agent is trying to learn the

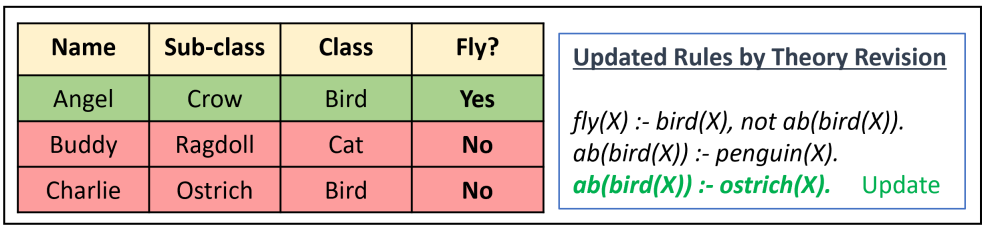

Figure 5: Update existing rules using Theory Revision for *'ostrich'*

concept of "*which animal can fly?*". Figure 4 shows a set of examples about different animals including feature details and the rules learned by the FOLD algorithm.

After learning a set of rules, the agent applies them in the environment and discovers that Charlie the ostrich is a type of bird that cannot fly and this is not covered by the rules. Now, our *theory revision* module will correct the existing hypothesis by learning a new exception (shown in figure 5).

While this is a simple example, it illustrates our logic-based symbolic reinforcement learning framework. The goal here is to emulate humans who can learn quite effectively from a small amount of data by forming an initial hypothesis and then correcting it over time as more instances are encountered.

6. Future Work and Conclusion

With the current proof-of-concept, we believe, we are one step closer to building a fully explainable symbolic reinforcement learning framework that can generate the relations (that the agent learns by interacting with the environment) in a human understandable way in first-order logic. As discussed above, the current implementation of our theory revision effort is in propositional logic, and our next task is to generalize it to first-order logic. This should allow us to perform end-to-end testing on different state-of-the-art reinforcement learning datasets. We believe, our work will not only perform well in terms of accuracy but will also be explainable.

References

[1] K. Basu, K. Murugesan, M. Atzeni, P. Kapanipathi, K. Talamadupula, T. Klinger, M. Campbell, M. Sachan, G. Gupta, A hybrid neuro-symbolic approach for text-based games using inductive logic programming, in: Combining Learning and Reasoning: Programming Languages, Formalisms, and Representations, 2021.

[2] M. Gelfond, Y. Kahl, Knowledge representation, reasoning, and the design of intelligent agents: The answer-set programming approach, Cambridge University Press, 2014.

[3] M. Gelfond, V. Lifschitz, The stable model semantics for logic programming., in: ICLP/SLP, volume 88, 1988, pp. 1070–1080.

[4] J. Arias, M. Carro, E. Salazar, K. Marple, G. Gupta, Constraint answer set programming without grounding, TPLP 18 (2018) 337–354. doi:10.1017/S1471068418000285.

[5] J. Arias, M. Carro, Z. Chen, G. Gupta, Justifications for goal-directed constraint answer set programming, arXiv preprint arXiv:2009.10238 (2020).

[6] J. R. Quinlan, Learning logical definitions from relations, Machine Learning 5 (1990) 239–266.

[7] F. Shakerin, E. Salazar, G. Gupta, A new algorithm to automate inductive learning of default theories, TPLP 17 (2017) 1010–1026.

[8] H. Wang, G. Gupta, FOLD-R++: A scalable toolset for automated inductive learning of default theories from mixed data, in: Functional and Logic Programming, 2022, pp. 224–242.

[9] H. WANG, F. SHAKERIN, G. GUPTA, FOLD-RM: A scalable, efficient, and explainable inductive learning algorithm for multi-category classification of mixed data, Theory and Practice of Logic Programming (2022) 1–20. doi:10.1017/S1471068422000205.
[10] E. Salazar, Theory revision with goal-directed asp., in: ICLP Workshops, 2021.