Explanations for Guess-and-Check ASP Encodings Using an LLM

Tobias Geibinger[1 [0000−0002−0856−7162]], Tobias Kaminski[2 [0000−0001−9776−0417]], and Johannes Oetsch[3 [0000−0002−9902−7662]]

[1] Institute for Logic and Computation, TU Wien, Austria
[2] Bosch Center for AI, Renningen, Germany
[3] Department of Computing, Jönköping University, Sweden
tobias.geibinger@tuwien.ac.at, tobias.kaminski@de.bosch.com, johannes.oetsch@ju.se

Abstract. Two pages plus references.

1 Introduction

Answer-set Programming (ASP) [4] is a rule-based decision making and problem-solving formalism used in many areas; among them are industrial optimisation, knowledge management, or life sciences. Thus, it is of great interest in the context of explainability [2]. To ensure the successful application of ASP as a problem-solving paradigm in the future, it is crucial to investigate explanations for ASP solutions. Such an explanation generally tries to give an answer to the question of why something is, respectively is not, part of the decision produced or solution to the formulated problem.

An approach that addresses this problem is xclingo [1], which wraps around the well-known ASP solver clingo [3] and provides explanations for the atoms in the produced answer set. The method works by having the user annotate the rules of the original ASP program with template sentences conveying the informal meaning of each rule. Those template sentences are then instantiated with the concrete solution and strung together in a derivation-like fashion.

Another related method for computing explanations for clingo programs that addresses the problem of why there is no solution is implemented as part of the clingo-explaid tools (github.com/potassco/clingo-explaid). There, the input is an unsatisfiable clingo program, and the output is a set of constraints that needs to be relaxed to produce a solution.

Our approach also solves the problem of explaining the answer sets of an ASP program. However, in contrast to previous attempts, we utilise a large language model (LLM) [9] as interface between the user and the explanation component. In particular, the LLM is used to interpret and formalise the questions of the user regarding the provided answer set. For example, a user can ask a contrastive question like “Why is the frame type aluminium and not carbon fiber?” and will receive an answer in natural language. The formalised question is then passed on to the ASP-based explanation component based on techniques described in [6], which in turn provides a formal answer. The latter is then again passed to the LLM, which translates it into natural language.

In general, there can be several possible explanations for a given question, and by default, only one is displayed to the user in natural language. However, our approach can be extended to let users choose between different explanations. By storing and analysing these choices, the system can statistically learn which parts of the underlying ASP model the user is most interested in to provide more tailored and relevant explanations over time.

The main advantages of our method are twofold. First, we heavily utilise natural language, both for receiving the questions regarding the solution from the user and in the answers provided to them. This enables much greater flexibility in what kind of question can be asked and makes the answers more concise and understandable.

The second main advantage is the greater range of types of questions and thus answers that are supported. The tool xclingo [1] only provides explanations of why an atom is in the answer set; clingo-explaid only explains why there is no answer set. Our method handles explanations that are contrastive in the sense that it provides an answer based on why something is included instead of something else. It has been argued that this is the more appropriate way to answer a "why" question [8].

In particular, integrating the LLM enhances the flexibility of our interfaces for question input and answer output. The LLM enables the classification of different types of input questions, such as conflict explanations (why is there no solution), contrastive explanations (why not A instead of B), and counterfactual explanations (why not B). Additionally, it allows for different levels of control over the answers through three alternative explanation modules for back-translation from ASP to natural language: direct translation with or without prompt engineering, using a template language, or using controlled natural language.

2 Background

ASP is a compact relational, in essence propositional, formalism in which variables in the input language are replaced by constant symbols in a preprocessing step called *grounding*. An ASP program is a set of rules of the form

$$p_1 \mid \ldots \mid p_k \texttt{ :- } q_1, \ldots, q_m, \texttt{not } r_1, \ldots, \texttt{not } r_n\,.$$

where all p_i, q_j, and r_l are atoms. The head is all the atoms before the implication symbol `:-`, and the body is all the atoms and negated atoms afterward. The intuitive meaning of this rule is that if all atoms $q_1, \ldots, q_m$ can be derived and there is no evidence for any of the atoms $r_1, \ldots, r_n$ (i.e., the rule fires) then at least one of $p_1, \ldots, p_k$ has to be derived. An *interpretation* I is a set of atoms. It is an *answer-set* of a program, if all its rules are satisfied in a minimal and consistent way [5]; intuitively, I must be a $\subseteq$-minimal model of all rules that fire.

A rule with an empty body is called a *fact*, with `:-` usually omitted. Facts are used to express knowledge that is unconditionally true. A rule with empty head is a *constraint*. The body of a constraint cannot be satisfied by any answer set and is used to prune away unwanted solution candidates.

A common syntactic extension are choice rules of the form

$$i \,\{\, p_1, \ldots, p_k \,\}\, j \texttt{ :- } q_1, \ldots, q_m, \texttt{not } r_1, \ldots, \texttt{not } r_n\,.$$

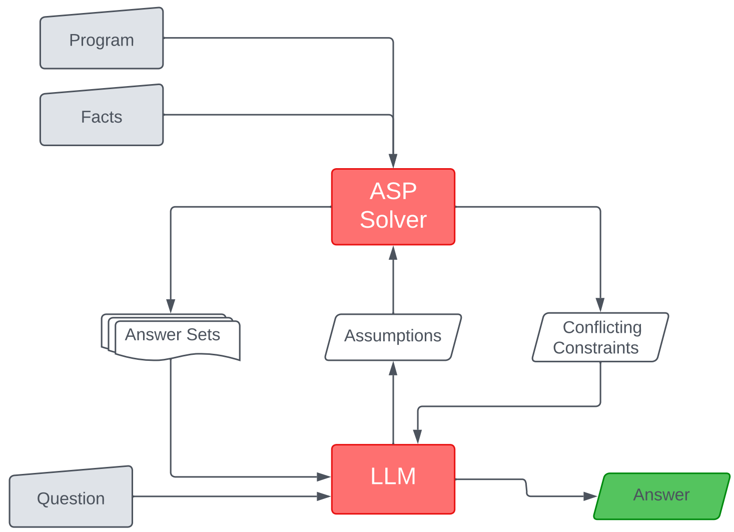

Fig. 1. Overview of our explanation method.

The meaning is that if the rule fires, then some subset S of $p_1, \dots, p_k$ with $i \leq |S| \leq j$ has to be true as well.

When used for combinatorial problem solving, ASP programs can usually be split into two components, a *guess* part enumerating potential solution candidates and a *check* part pruning the candidates which are not solutions.

Contrastive explanations [7] answer why a decision has been reached in contrast to a different one, i.e., they answer questions of the form "Why P rather than Q?". It has been argued that contrastive explanations are intuitive for humans to understand and produce, and also that standard "Why" questions contain a hidden *contrast case*, e.g., "Why P?" represents "Why P rather than not P?" [8]. Lipton [7] defines an answer to a question of the form "Why P rather than Q?" as the *difference condition*, which states that the answer should contain a cause for P that is missing in the causal history of not Q. More formally, an answer consists of a cause c of P and a cause c' of Q such that $c \neq c'$ and c did occur, whereas c' did not. One of the benefits of contrastive explanation is that the question includes partial information about the explainee. That is, the contrast case reveals which parts of the causal history are already clear to them.

3 Explaining Guess-And-Check Encodings

Figure 1 shows an overview of the method, especially how the components interact. It should be noted that the design is modular and the concrete underlying ASP solver and LLM are not fixed.

The individual steps can be described as follows:

1. The basic input is an ASP program that includes input facts. The latter generally represents the input instance data, whereas the program remains largely fixed for a particular problem.
2. The input is then passed to an ASP solver, which is tasked with providing an answer set.
3. If the problem has a solution, the corresponding answer set is printed. Otherwise, we jump directly towards finding a minimally unsatisfiable set of constraints.
4. After that, the user is asked if they have any questions regarding the answer set. Preferably those are questions that are directly contrastive or "why" questions for which the contrast can be inferred.
5. The user question is then passed on to the LLM, which implicitly classifies the type of question, using the information provided in a custom prompt (i.e., conflict explanation, contrastive explanation or counterfactual explanation).
6. Based on the custom prompt, the LLM extracts assumptions about the answer set which corresponds to the question. The custom prompt contains several in-context examples, ASP background, as well as the input, and tasks the LLM to produce ASP constraints, which force certain atoms to be true or false, respectively. Those constraints can be viewed as assumptions on the truth values of these atoms. Intuitively, if the user asks "Why P and not Q?", the assumptions will enforce a counterfactual where P is false and Q is true.
7. Subsequently, the generated assumptions are added to the original ASP program.
8. With these assumptions in place, the ASP solver is then tasked with finding a minimally unsatisfiable set of constraints for this scenario, thus giving a formal explanation as to why the counterfactual cannot hold.
9. Finally, the LLM is provided with the violated constraints and prompted to translate the constraints into natural language to summarise this information. For this translation, one of at least three alternative approaches can be applied:
   – direct back-translation using the LLM,
   – using a template language to constrain the output, or
   – using controlled natural language to have complete control over the shape of the output.

Example 1. Consider the following input program, representing the 3-colourability problem with a simple instance of three nodes:

```
% coloring.lp

% choose a color for each node
{ chosenColor(N,C) : color(C) } = 1 :- node(N).

% adjacent nodes are not allowed to have the same color
:- edge(N1,N2), chosenColor(N1,C), chosenColor(N2,C).

% input
color(red).
color(green).
color(blue).
```

```
node(1..3).
edge(1,2).
edge(1,3).
edge(2,3).
```

An example execution of the explainer will look as follows:

```
> python main.py examples/coloring.lp
Answer:
chosenColor(3,red) chosenColor(2,green) chosenColor(1,blue)
node(1) node(2) node(3) edge(1,2) edge(1,3) edge(2,3)
color(red) color(green) color(blue)
Question:
why are node 1 and node 2 not both blue?

Explanation:
Node 1 and node 2 are not both blue because there is an
edge between them, and adjacent nodes are not allowed to
have the same color.
```

In this example, the question "Why are node 1 and node 2 not both blue?" can be reformulated as "Why is node 1 blue and node 2 green in contrast to node 1 blue and node 2 blue?" from which the following assumptions are generated by the LLM:

```
:- chosenColor(1,blue), chosenColor(2,green).
:- not chosenColor(1,blue).
:- not chosenColor(2,blue).
```

The ASP program joined with these assumptions is now inconsistent. The ASP solver is then used to search for a minimal set of constraints from the original program, whose removal restores consistency. In this case, this is achieved by removing an instance of the constraint that expresses that adjacent nodes are not allowed to have the same colour. The LLM finally translates this information into natural language to produce the answer to the initial question.

In order to compute the minimally unsatisfiable set of constraints, we use the following approach. First, the program is rewritten so that new dummy atoms are added to the head of each constraint. In this way, each set of answers will show which constraints have been violated. We can then compute answer sets where the dummy atoms are minimal (either subset minimal or cardinality minimal). Those sets are generally called minimal correction sets. Lastly, we generate a hitting set of all minimal correction sets, which corresponds to a minimally unsatisfiable set of constraints. The approach we employ in this part of our method is largely based on the methods for computing minimal correction sets and minimally unsatisfiable sets described in [6].

Acknowledgments. This work was supported by funding from the Bosch Center for AI at Renningen, Germany and has benefited from the European Union's Horizon 2020 research and innovation programme under grant agreement No 101034440 (LogiCS@TUWien). Tobias Geibinger is a recipient of a DOC Fellowship of the Austrian Academy of Sciences at the Institute of Logic and Computation at the TU Wien.

References

1. Cabalar, P., Fandinno, J., Muñiz, B.: A system for explainable answer set programming. In: Technical Communications of the 36th International Conference on Logic Programming (ICLP 2020). EPTCS, vol. 325, pp. 124–136 (2020)
2. Fandinno, J., Schulz, C.: Answering the "why" in answer set programming - A survey of explanation approaches. Theory Pract. Log. Program. **19**(2), 114–203 (2019). https://doi.org/10.1017/S1471068418000534
3. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Clingo = ASP + control: Preliminary report. CoRR **abs/1405.3694** (2014), arxiv.org/abs/1405.3694
4. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Logic Programming, Proceedings of the Fifth International Conference and Symposium. pp. 1070–1080. MIT Press (1988)
5. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Gener. Comput. **9**(3/4), 365–386 (1991)
6. Herud, K., Baumeister, J., Sabuncu, O., Schaub, T.: Conflict handling in product configuration using answer set programming. In: Proceedings of the ICLP 2022 Workshops, 2022. CEUR Workshop Proceedings, vol. 3193. CEUR-WS.org (2022), https://ceur-ws.org/Vol-3193/paper2ASPOCP.pdf
7. Lipton, P.: Contrastive explanation. Royal Institute of Philosophy Supplement **27**, 247–266 (1990). https://doi.org/10.1017/S1358246100005130
8. Miller, T.: Explanation in artificial intelligence: Insights from the social sciences. Artificial Intelligence **267**, 1–38 (2019). https://doi.org/https://doi.org/10.1016/j.artint.2018.07.007
9. Raiaan, M.A.K., Mukta, M.S.H., Fatema, K., Fahad, N.M., Sakib, S., Mim, M.M.J., Ahmad, J., Ali, M.E., Azam, S.: A review on large language models: Architectures, applications, taxonomies, open issues and challenges. IEEE Access **12**, 26839–26874 (2024). https://doi.org/10.1109/ACCESS.2024.3365742, https://doi.org/10.1109/ACCESS.2024.3365742