Grenze International Journal of Engineering and Technology, Jan Issue

Knowledge Extraction from Answer Set Programming based Encoding Selection

Lalit Kumar[1] and Praveen Ailawalia[2]
[1]Department of Computer Science and Engineering, Chandigarh University, Gharuan-Mohali, Punjab, India
Email: lalit.cse@gmail.com
[2]Department of Mathematics, University Institute of Sciences, Chandigarh University, Gharuan-Mohali, Punjab, India
Email: praveen_2117@rediffmail.com

***Abstract*— In the present article the authors introduce procedures to generate multiple alternative encodings, identical to the original encoding, to enhance encoding diversity. The procedures to generate multiple alternative encodings, identical to the original encoding, to enhance encoding diversity. System capable of taking a set of problem encodings, problem instances, and an ASP grounding solving system as inputs, automatically generating equivalent encodings. As new instances arise, the system selects the encoding expected to perform best for each instance, enabling more efficient ASP solving. The system supports both planned and interleaved execution of encodings, complementing machine learning methods. This approach can be extended to identify challenging instances for other combinatorial problems, further enhancing the utility of our framework.**

***Index Terms*— Answer Set Programming, Pythagorean Triplet, Encoding, AI, Machine Learning, NP Hard Problem.**

I. INTRODUCTION

Modeling and resolving complex research problems using Answer Set Programming (ASP) is a well-established practice. ASP represents these problems as encodings, which consist of sets of rules that define the problem's logic without providing solutions. Brewka et al. (2011) introduces the foundational principles of Answer Set Programming (ASP), shedding light on its role as a promising declarative problem-solving approach. A concise and informative overview of ASP's key concepts and motivations.

Buddenhagen M. and Lierler Y., (2015) focuses on the application of performance analysis and tuning techniques in the domain of answer set programming, which is a significant constraint programming paradigm. The study explores the roles of both human tuning and automatic configuration tools in this context and presents a case study involving a real-world answer set programming application comprising a substantial amount of code.

Cat De et al. (2018) discusses the application of Answer Set Programming (ASP) for the purpose of discovering sequential patterns. ASP is a declarative logic programming paradigm known for its effectiveness in encoding combinatorial and optimization problems, as well as for facilitating knowledge representation and reasoning. Consequently, ASP stands as a promising choice for implementing pattern mining, particularly when it involves incorporating background knowledge. This integration of background knowledge into pattern mining has been a long-standing concern within the field of data mining.

Calimeri F. et al. (2019) discussed encodings exist in Answer Set Programming (ASP) for various search and

Grenze ID: 01.GIJET.10.1.517_2

optimization tasks. These encodings do not exhibit consistent performance superiority when evaluated across a large number of problem instances. The authors suggest that leveraging machine learning techniques to identify encodings likely to perform well on specific instances can potentially enhance ASP's problem-solving capabilities. Their research on the Hamiltonian cycle problem provides empirical support for this assertion.
Bichler et.al. (2020) introduces decomposing large logic programming rules in ASP, aimed at improving solver performance. The paper includes an algorithm, theoretical foundations, and experimental results supporting its efficacy. A valuable contribution to ASP optimization.
Eiter et al. (2022) has gained popularity as a declarative method for addressing problems, it is not without its limitations when it comes to optimisation issues. High yet unrealized potential exists in the use of the optimization metaheuristic known as Large-Neighbourhood Search (LNS) for resolving ASPs.
Geibinger and Tobias (2023) discussed growing importance of AI explainability stems from its widespread integration into our lives and increasingly complex systems. This project aims to enhance explanation capabilities in Answer-Set Programming (ASP), addressing gaps in existing approaches and introducing novel formalisms like contrastive explanations for ASP solutions.
Aguado et al. (2023) introduces Temporal Logic Programming and its application in Knowledge Representation and Problem Solving. It introduces Temporal Equilibrium Logic (TEL) and extends stable model semantics for temporal formulas, offering valuable insights into this complex field.
It is common in ASP to have multiple equivalent encodings for the same problem. Recent studies have shown that the performance of different ASP encodings varies significantly when processed by modern ASP grounding and solving systems. It is rare for a single encoding to outperform all others consistently. The choice of an appropriate encoding plays a crucial role in the effectiveness of ASP solutions. In this paper, we provide a framework for LNS optimization in solving answer sets, where neighbourhoods may be defined either declaratively as part of the ASP encoding or created programmatically. We concentrate on multi-shot solutions, which helps us to prevent program re-grounding and successfully probe a variety of nearby neighbourhoods.

II. FINDINGS

Answer Set Programming (ASP) has found application in various theoretical and practical domains due to its versatile high-level modelling capabilities and the ongoing enhancements in solver performance. However, despite its simplicity in modelling, ASP poses challenges in implementation. The grounding and solution procedures encounter substantial hurdles. Particularly, when dealing with complex rule systems, solving can become time-consuming.
The computational intensity of variable instantiation diminishes the throughput of ASP systems. In situations where there are a thousand instances of each variable, the size of the grounding set can reach an enormous scale of 10^{12} rules for a simple rule with only four variables. This substantial rule expansion creates difficulties in generating the fundamental program structure, even when utilizing advanced grounding techniques. To expedite the grounding process for inefficient encodings, one effective strategy is to reduce the number of variables within the rules. These principles find application in various scenarios, including the enforcement of reachability requirements in problems like the Hamiltonian cycle, as implemented by computer systems.
The auxiliary predicate reach is defined in this set of rules by use of the three variables X, Y, and Z. Approximately 11 million lines are produced in the grounded output when the program is run on a random graph instance with 1000 nodes and 10000 edges. Assuming an additional 31.30 seconds for grounding, the total solution time is 318.63 seconds.
Encoding rewriting techniques may be used to use projection to minimize the number of variables in rules, thereby solving the "grounding" issue. Another strategy is to discover a model for the restriction that results in a rule encoding with fewer variables. To represent the reachability constraint, for instance, it is necessary to first choose a node (either statically or according to a set of constraints). Take *s* to be a chosen node. Clearly, candidates are collections of disjoint cycles spanning all vertices and cycles constitute really a single cycle due to the reachability condition that demands that each vertex be accessible from *s* by a route of chosen edges. This limitation may be phrased as such (recall that we retain the same name for the reachability predicate and that we have fixed the "start" node to be 1), so that the program will only function for graphs in which 1 is a vertex.
It is possible to rewrite the code as

```
Reach(X):- hcedge(1,X).
Reach (Y):- reach(X), hcedge(X,Y).
:- not reach (X), node(X)
```

The second difficulty is finding a solution. The intrinsic difficulty of the issue it addresses causes the solution

process to bog down. The issue of determining whether a given propositional program has an answer set is NP-complete (and even P-complete for certain implementations of the language). It is commonly believed that there is no way to solve such issues in a way that would ensure optimal performance. Therefore, when evaluated on a large set of examples, even the most cutting-edge solvers will only be able to solve a subset of the issues. Interestingly, a large body of experimental data accumulated over the years demonstrates that excellent solvers thrive in a certain domain or class of examples but inevitably underperform on many other sorts of instances. The strong points of many solvers tend not to overlap. As a result, the emergence of a single solver that is superior than the others is quite improbable. That gives chance to enhance solution with ASP solvers by choosing the best solver for each individual case.

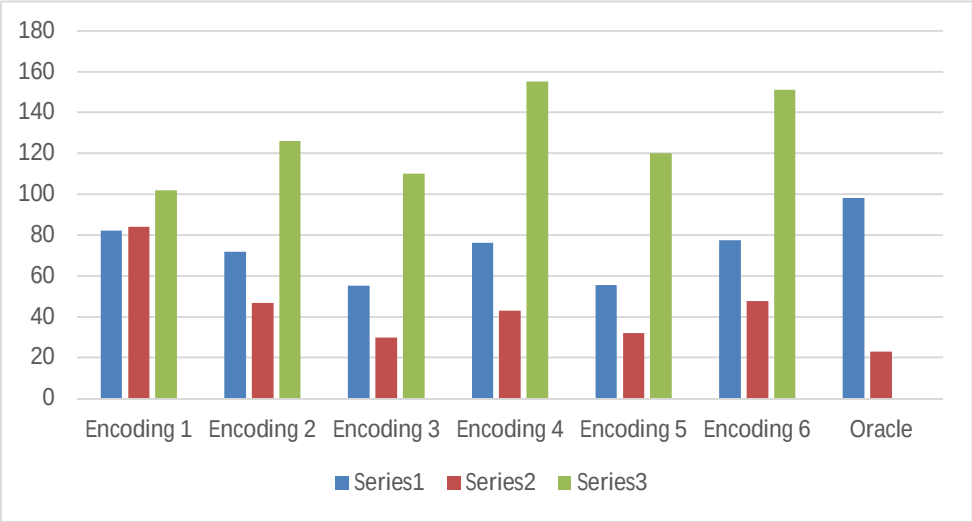

Figure1: Performance of individual encodings and the oracle

Current state-of-the-art solvers are built with tens or even hundreds of control parameters (settings), which impact the choices made throughout the searching process; this presents both a difficulty and an opportunity for the solution phase. The number of conflicts that will force a restart, the counting strategy for those conflicts (local vs. global), the number of learned causes that will be stored, the number of learned causes that will be deleted after a restart, and the number of restarts that will occur before rearranging internal data are all examples. The performance of solution may be improved by many orders of magnitude with a well-chosen parameter configuration, as shown in previous research. Modern solvers often provide a default parameter configuration that is effective in many situations; but, for some sorts of instances, the optimized settings perform better. Another option is to stop looking for the optimal parameter values, it is possible to build a library of solvers from a single one by picking several parameter settings and then executing many of them in parallel or in succession. Portfolio solution is the term for this kind of analysis.

As may be expected, the topics of solver selection, portfolio solving, and automated parameter con- figuration have all been the subject of substantial research in ASP. Researchers advocated doing solver selection on a per-instance basis, i.e., using machine learning approaches to develop performance models for particular solvers, to take advantage of the variable performance on various problems, analysis encoding performance as shown (Figure 1). These models, when presented with a new instance, predict how well the relevant choices (solvers, parameter configurations) will perform on that instance. Based on these forecasts, a suitable solution or set of parameters for the case in question may be chosen.

There is much more that can be done to boost ASP's efficiency. It is also generally known that search problems often accept several comparable encodings, as we pointed out before. While theoretically equal, the performance of these encodings tends to vary among instances. Over the previous two decades of experimentation, data has gathered suggesting that multiple AS encodings for a particular task may exhibit notably differing performance. In particular, it is unusual for the same encoding to be optimal (with a chosen grounder-solver tool) for all issue data instances.

III. Pythagorean Triplet

The three numbers a, b, and c must satisfy the equation $a^2 + b^2 = c^2$. 1 to be considered a Pythagorean Triplet.

To solve the Pythagorean Triplet issue in ASP, one must determine the biggest n such that no partition of the integers from 1 to n has the three positive integers a, b, and c such that $a^2 + b^2 = c^2$. When n is larger than 7825, there is no model for this issue. Constraints are a natural fit for modeling the Pythagorean Triplet issue. To ensure that no component includes the square sum relation, we may create a constraint and iteratively tweak n until we reach the ultimate answer, the first n that makes the issue unsatisfiable. The following is a paradigm for an encoding of the Pythagorean Triplet.

First, we use the interval operation.. to provide the range of integers (1, 2, 3,...) that will be used for the partition. According to the second guideline, the totals must be divided in half. Using the choice rule, the third rule specifies that each number belongs to exactly one of the subparts. Any given section must not have the three values X, Y, and Z such that $a^2 + b^2 = c^2$. This final criterion is a constraint.

If the above encoding is satisfiable, then the numbers may be correctly divided into two halves, one of which does not include a Pythagorean Triplet. If it is not satisfiable, then the biggest number n that can be divided is determined.

Using the aforementioned encoding to solve Pythagorean Triplet problems, the authors observed that the runtime (grounding + solution) climbs exponentially with the amount of input. After digging further, grounding step accounted for the vast majority of the total time. Figure 1 shows that the overall runtime is very close to the grounding time, indicating that once an issue has been identified and isolated, it may be readily resolved. The question of how to maximize productivity without sacrificing safety by increasing the amount of time spent grounding emerges. We need information on the grounding step to help us solve the issue. In the grounding step, the grounder executes a square sum operation, which involves instantiating X, Y, and Z with all conceivable values. In addition, it preserves just the instantiations that meet the square sum relation $X^2 + Y^2 = Z^2$, while discarding the others. When n = 10, for instance, the grounding output is from constraint import Problem, All Different Constraint

```
# Create a constraint problem
problem = Problem()
# Define the variables and their domains
partitions = list(range(1, 11))
parts = [1, 2]
# Define the partitions for each part
partitions_by_part = {}
for part in parts:
partitions_by_part[part] = [problem.addVariable(f'partition_{p}_{part}', partitions) for p in partitions]
#Define the constraints
for part in parts: problem.addConstraint(AllDifferentConstraint(), partitions_by_part[part])
# Define the equality constraints
equality_constraints = [(1, 1, 1), (2, 2, 1), (3, 1, 2), (4, 1, 2), (5, 1, 1), (5, 2, 1), (6, 1, 2), (6, 2, 2), (7, 1, 1), (7, 2, 1), (8, 1, 2), (8, 2, 2), (9, 1, 1), (9, 2, 1), (10, 1, 2), (10, 2, 2)]
for constraint in equality_constraints:
problem.addConstraint(lambda a, b, value=constraint[0]: a + b == value, (f'partition_{constraint[1]}_{constraint[2]}', f'partition_{constraint[0]}_{constraint[2]}'))
# Find the solutions
solutions = problem.getSolutions()
# Print the solutions
for solution in solutions: print(solution)
```

The grounded program is shown to only save instances when $X^2 + Y^2 = Z^2$. Therefore, in the grounding phase, once square calculations and equation evaluations are performed, the majority of the created instantiations are discarded.

One approach is to perform the necessary computations outside of the ASP program (for example, all Pythagorean Triplets over integers in 1, 2,..., *n*) and compile them into a list of facts over a new predicate; then, by using these facts, the program can be expanded and the arithmetic atom in appropriate rules can be replaced. That manner, we avoid creating any unnecessary instantiations during grounding. The results of the same sample code given in Figure 2 is shown in Figure 3. Where the list of Triplets (*a, b, c*) outside of the grounder where $c^2 = a^2 + b^2$, and then integrate that result using a suitably adjusted AS program representing the issue. In the aforementioned example, we are able to predict the expansion of the predicate *sqsum*(*a, b, c*), which includes any integer Triplets (*a, b, c*) matching the formula $a^2 + b^2 = c^2$. The original occurrence of the arithmetic equality atom is then

replaced by the precomputed predicate *sqsum*, and the facts *sqsum(a, b, c)* are added to the program.

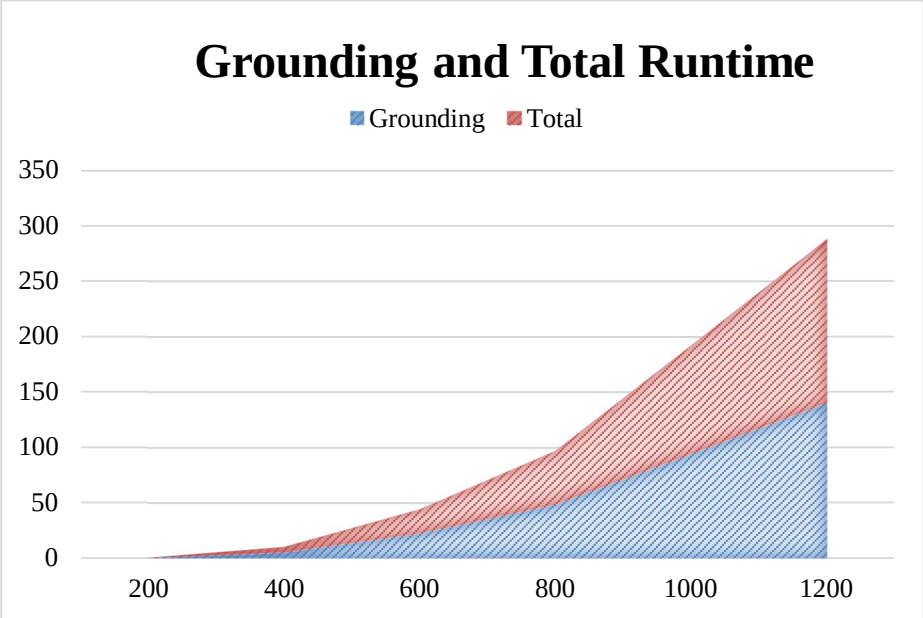

FIGURE 2: Runtime Original Pythagorean for Encoding

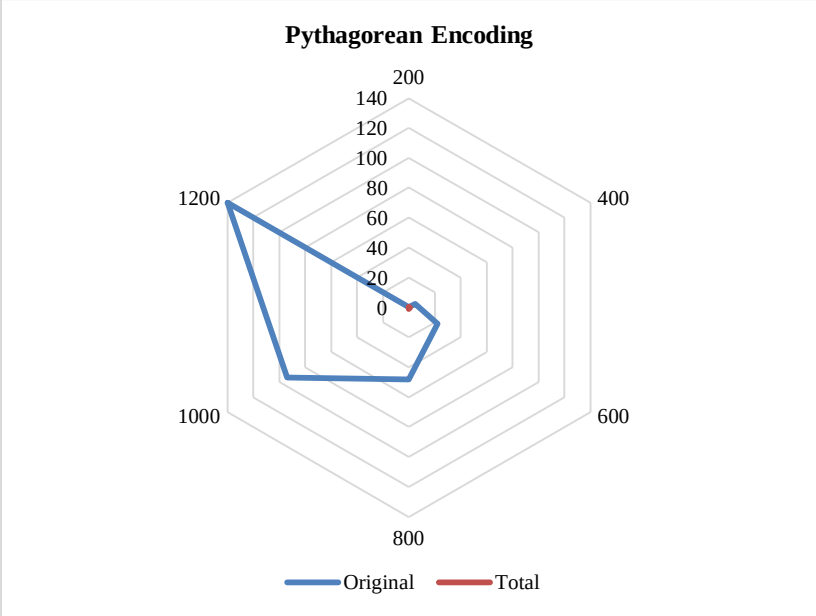

Figure 3: Pythagorean Encodings

IV. Conclusion

- Proposed Research introduces rewriting techniques for improving Answer-Set Programming (ASP) performance in grounding and solution stages.
- Manual removal of mathematical particles during grounding phase and replacement of arithmetic atoms using procedural programming to accelerate initialization.
- Testing on Pythagorean Triplet and Schur number issues demonstrates significant reduction in initialization time.
- In the solution phase, encoding rewriting and selection strategies to enhance ASP functionality.

- Development of an automatic encoding rewriting tool, *AAgg*, enabling conversion of aggregate rules to regular rules and broadening aggregate usage.
- Proposals include rewriting ASP encodings through rule replication using alternative atoms.
- Strategies for generating sets of complementary encodings for issue cases and employing machine learning for instance-specific encoding selection.

REFERENCES

[1] Geibinger T., (2023) "Explainable Answer-set Programming." Proceedings 39th International Conference on Logic Programming (ICLP 2023), Imperial College London, UK, July 2023 - 15th July 2023, Electronic Proceedings in Theoretical Computer Science 385, 423–429.

[2] Aguado, F., Cabalar, P., Diéguez, M., Pérez, G., Schaub, T., Schuhmann, A., & Vidal, C. (2023). "Linear-time temporal answer set programming." Theory and Practice of Logic Programming, 23(1), 2-56.

[3] Gebser, M., Kaminski, R., Kaufmann, B. and Schaub T. (2022) "Answer set solving in practice." Springer Nature.

[4] Liu, L. (2022) "The Performance Optimization of ASP Solving Based on Encoding Rewriting and Encoding Selection." PhD diss., University of Kentucky Libraries.

[5] Bichler, M., Morak, M., & Woltran, S. (2020) "lpopt: A rule optimization tool for answer set programming." Fundamenta Informaticae, 177(3-4), 275-296.

[6] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, M. Maratea, F. Ricca, and T. Schaub (2019) "Asp-core-2 input language format."

[7] De Cat, B., Bogaerts, B., Bruynooghe, M., Janssens, G., & Denecker, M. (2018) "Predicate logic as a modeling language: the IDP system." In Declarative Logic Programming: Theory, Systems, and Applications, pp. 279-323.

[8] Morak, M,, Bichler, M. and Woltran, S. (2017) "lpopt: A Rule Optimization Tool for Answer Set Programming." In Logic-Based Program Synthesis and Transformation, pp. 114-130.

[9] Buddenhagen, M. and Lierler Y. (2015) "Proceedings of the 13th International Conference on Logic Programming and Nonmonotonic Reasoning," LPNMR-2015, volume 9345 of Lecture Notes in Computer Science, Springer, 2015, pp. 186–198.

[10] Erdem, E., & Oztok, U. (2015) "Generating explanations for biomedical queries." Theory and Practice of Logic Programming, 15(1), pp. 35-78.

[11] Erdem, E., Aker, E., & Patoglu, V. (2012) "Answer set programming for collaborative housekeeping robotics: representation, reasoning, and execution." Intelligent Service Robotics, 5, 275-291.

[12] Brewka, G, Eiter, T. and Truszczyński M. (2011). "Answer set programming at a glance." Communications of the ACM 54, no. 12 (2011): 92-103.

[13] Clocksin, W. F., and Mellish C.S. (2003) "Programming PROLOG." Springer Science & Business Media.

[14] Cheeseman, P. C., Kanefsky, B., & Taylor, W. M. (1991) "Where the really hard problems are." Ijcai, Vol. 91, pp. 331-337.