Encoding Selection for Solving Hamiltonian Cycle Problems with ASP

Liu Liu Miroslaw Truszczynski
Department of Computer Science
University of Kentucky
Lexington, KY 40506, USA
liu.liu@uky.edu mirek@uky.edu

It is common for search and optimization problems to have alternative equivalent encodings in ASP. Typically none of them is uniformly better than others when evaluated on broad classes of problem instances. We claim that one can improve the solving ability of ASP by using machine learning techniques to select encodings likely to perform well on a given instance. We substantiate this claim by studying the hamiltonian cycle problem. We propose several equivalent encodings of the problem and several classes of hard instances. We build models to predict the behavior of each encoding, and then show that selecting encodings for a given instance using the learned performance predictors leads to significant performance gains.

1 Introduction

Answer Set Programming (ASP) [3] has been shown to be especially effective on search and optimization problems whose decision versions are in the class NP, including many problems of practical interest [9, 6]. Despite the ease of modeling and the demonstrated potential of ASP, using it poses challenges. In particular, it is unlikely a single solver will emerge that would uniformly outperform other solvers. Consequently, selecting a solver for an instance may mean the difference between solving the problem within an acceptable time and having the solver run "forever." To address the problem, solver selection, portfolio solving, and automated solver parameter configuration have all been extensively studied [17, 10, 14, 16, 12]. The key idea has been to learn instance-driven performance models and use them, given an instance, to select a solver (or a parameter configuration) that might perform well on that instance.

Another challenge is selecting the right encoding. It is well known that problems have alternative equivalent encodings as answer set programs. The problem is that these encodings typically perform differently when run on different instances. Consequently, one can seek program rewriting heuristics to generate better performing programs, or develop methods for encoding selection and encoding portfolio solving, similar to those used in portfolio solving. The first idea has received some attention in recent years [4, 2, 11]. However, the approach to capitalize on the availability of collections of equivalent encodings has not yet been explored.

We pursue here this latter possibility and offer for it a proof of concept. To this end, we study a computationally hard *hamiltonian cycle* (HC) problem. We construct several ASP encodings of the problem as well as a collection of hard instances. We show that using standard machine learning approaches one can build a performance model for each encoding based on its performance data, and that these performance models are effective in guiding a selection of encodings to be used with a particular instance. Our experiments show performance improvements and suggest encoding selection as a technique of improving ASP potential to solve hard problems.

Submitted to:
ICLP 2019

2 Encoding Candidates for the HC Problem

The HC problem has directed graphs as input instances. An instance is given by the lists of nodes and links (edges), represented as ground atoms over a unary predicate `node` and a binary predicate `link`. The code below represents a directed graph with four nodes and six links.

```
node(1..4).
link(1,2).link(1,3).link(2,1).link(3,4).link(4,2).link(4,3).
```

The HC problem imposes constraints on a set of edges selected to form a solution: there has to be exactly one edge leaving each node, exactly one edge entering each node, and every node must be *reachable* from every other node by a path of selected edges. These constraints can be modeled by program rules such as those below.

```
%Select edges
{ hcyc(X,Y) : link(X,Y) }=1 :- node(X).
{ hcyc(X,Y) : link(X,Y) }=1 :- node(Y).
%Define reachability
reach(X,Y) :- hcyc(X,Y).
reach(X,Z) :- reach(X,Y),hcyc(Y,Z).
%Enforce reachability
:- not reach(X,Y),node(X),node(Y).
```

The first two rules model the constraints on the number of selected edges (represented by a binary predicate `hcyc`) leaving and entering each node. The third and the fourth rule together define the concept of reachability by means of selected edges. Finally, the last rule, a constraint, guarantees that every node is reachable from every other node by means of selected edges only.

We can rewrite this encoding to generate its variants. For example, reachability can be modeled by selecting a node, say 1, and requiring that every node in the graph (including 1) is reachable from 1 by a non-trivial path (at least one edge) of selected edges. Another possibility is to change the way we select edges by rewriting the first two rules.

To obtain a collection of several high-performing encodings for the HC problem, we generated 15 encodings based on different constraint representation and rewriting ideas, as discussed above. We ran these encodings on hard instances to the HC problem (cf. Section 3) and selected six encodings based on (1) the percentage of solved instances, and (2) the number of instances for which an encoding yields the fastest solve time. We refer to these encodings as Encoding $1,\ldots,6$.[1]

Table 1 summarizes the performance data for the six encodings on 784 hard instances (we comment later on how the instances were generated). The results on the number of wins, instances solved fastest by the encoding, show that our encodings have complementary strengths. We observe that the oracle always selecting the fastest solver solves about 98.0% of all instances, an improvement of about 16% over the best individual encoding. This indicates that there is much room for intelligent encoding selection methods to improve the performance of ASP on the HC problem.

3 Data Collection

Performance data. A natural and often used class of graphs for experimental studies of algorithm performance is the class of graphs generated randomly from some distribution space. We designed a program to generate graphs with n nodes and e edges at random and searched for areas of hardness. We

[1] All six encodings, the data sets and detailed experimental results can be accessed at `http://cs.uky.edu/~lli259/encodingselection`.

Table 1: Performance of individual encodings and the oracle.

Encoding	Solved Percentage%	Average Solved Runtime	Number of Wins
Encoding 1	82.3	84.1	102
Encoding 2	71.8	46.6	126
Encoding 3	55.3	29.7	110
Encoding 4	76.2	42.9	155
Encoding 5	55.4	31.9	120
Encoding 6	77.4	47.7	151
Oracle	98.0	22.8	

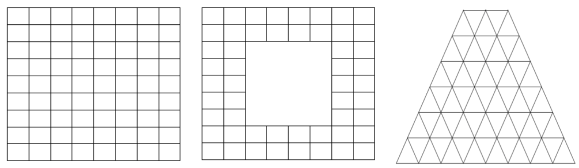

Figure 1: Structured instances: regular grid graph, regular grid graph with hole, regular triangular graph with cutting area

have not identified any such area. Even when we considered graphs with thousands of nodes and the number of edges selected from the *phase transition* range, they could be solved within 10 seconds.[2]

To find classes of graphs for which the HC problem is not easily solved in ASP, we consider structured graphs (structure often is a source of hardness). Starting with some highly structured graph that has a hamiltonian cycle, we remove edges at random until the graph is no longer hamiltonian. In this work, we start the process with grid graphs shown in Figure 1. We set their dimensions and "hole" locations so that to guarantee the existence of a hamiltonian cycle. Our experiments demonstrate that graphs with the number of edges in the phase transition region tend to yield programs that often require hundreds or thousands of seconds, even when the graphs have relatively few nodes (of the order of hundreds).

To collect performance data, we generated a large collection of graphs. Then we combined each graph with each of the six encodings, and ran *clasp/gringo*[3] on the resulting programs. We set the cutoff time to 200s. When an instance timed out, we used the penalized runtime as an approximation of its real runtime, computed based on the number k of encodings for which the instance timed out. We then selected instances with runtime between 50s and 200s for at least one encoding (not necessarily the same one). We call these instances *reasonably hard* (those that cannot be solved with any encoding in under 200s are too hard, and those that can be solved with each encoding in under 50s are too easy). Finding reasonably hard instances is time consuming. Our experiments show that only a small fraction falls into this category (cf. Table 2). Thus, typically several graphs need to be generated before a single reasonably hard instance is found. We used this method to build a collection of 784 reasonably hard instances.[4]

Instance features. To use machine learning to construct a predictor of performance for a given instance on a particular encoding, we need informative and easy to compute instance features. In this work,

[2]As e grows (given a fixed n), the likelihood of the graph having a hamiltonian cycle switches from 0 to 1. The region where the switch occurs is called the *phase transition*. For many problems, such as satisfiability of k-CNF formulas, this is where hard problems are located [18].

[3]`https://potassco.org`

[4]The graph instances and the performance data can be downloaded from `http://cs.uky.edu/~lli259/encodingselection`

Table 2: Runtime distribution of Encoding 2 on 500 different instances that are gained through eliminating random edges from a regular square grid graph with length of its side 10.

runtime	<50s	50s ∼200s	200s≤
counts	330	52	118

we considered features of two types: graph features and encoding-based features. Some graph features capture general characteristics of graphs such as the numbers of nodes and edges, or the minimum and the maximum degrees. Other graph features are constructed to reflect aspects of the problem at hand. In our case, they are designed to capture properties of depth-first and breadth-first search trees rooted in nodes of the graph as they inform about reachability from a node.[5]

Encoding-based features of an instance are obtained by means of the program *claspre*[6]. It extracts *static* and *dynamic* features of ground ASP programs while solving them for a short amount of time. In order to obtain *claspre* features of a graph instance, we combine the instance with our six encodings and then pass the resulting ground programs to *claspre*. In total, there are 569 features in 13 groups, one group of graph features and six pairs of groups of *claspre* encoding-based static and dynamic features. When learning performance predictors (the details are in the next section), we used a narrowed down set of features to avoid overfitting and retain features that are informative for the HC problem.

4 Encoding Selection with Machine Learning

The goal of encoding selection is to identify encodings that promise good performance for a given instance. Our work is based on the performance data and instance features computed and collected for the data set of 784 reasonably hard instances. We use this data to build regression models for the six encodings we constructed as representations of the HC problem. Specifically, we build k-nearest neighbor (KNN), decision tree (DT) and random forest (RF) regressors. All these models are directly imported from python *scikit-learn* package.

To select informative features we perform in-group individual feature selection followed by the feature group selection. We start with an empty feature set, randomly add one or more features, and then test the average performance of selected features to decide whether to keep them or not.

To evaluate a particular set of selected features, we randomly divide our data into the training set (80%) and the test set (20%), train models using the training set, and test the performance of encoding selection result on the test set. We partition training data into 10 bins and use 10-fold cross-validation to improve the generalization performance. The approach we described in this section results in a set of 42 features (cf. Table 3), 27 graph features and 14 *claspre* static features, all obtained with Encoding 1. Our results show that *clapsre* dynamic features are not as informative as graph features and *claspre* static features. We note, however, that because we used the greedy feature selection, our set of features may not be optimal and better selections may be possible.

5 Experimentation

Hardware. Our experiments were conducted on a computer with four cores, each with Intel i7-7700 3.60GHz CPU and 16GB RAM, running under 64-bit 18.04.2 LTS (Bionic Beaver) Ubuntu system. The

[5] All features we used are available at `http://cs.uky.edu/~lli259/encodingselection`

[6] `https://potassco.org/labs/claspre/`

Table 3: Selected instance features for encoding selection

ratio_node_edge	avg_depth_beam	Frac_Binary_Rules_hc1
ratio_bi_edge	dfs_1st_back_depth	Frac_Ternary_Rules_hc1
avg_out_degree	sum_of_choices_along_path	Free_Problem_Variables_hc1
avg_in_degree	depth_avg_dfs_backjump	Problem_Variables_hc1
ratio_of_odd_out_degree	depth_back_to_root	Assigned_Problem_Variables_hc1
ratio_of_even_out_degree	depth_back_to_any	Constraints_hc1
ratio_of_odd_in_degree	depth_one_path	Rules_hc1
ratio_of_even_in_degree	min_depth_bfs	Frac_Normal_Rules_hc1
ratio_of_odd_degree	max_depth_bfs	Frac_Cardinality_Rules_hc1
ratio_of_even_degree	avg_depth_bfs	Frac_Choice_Rules_hc1
ratio_out_degree_less_than_3	min_depth_beam	Frac_Binary_Constraints_hc1
ratio_in_degree_less_than_3	max_depth_beam	Frac_Ternary_Constraints_hc1
ratio_degree_less_than_3	avg_depth_beam	Frac_Other_Constraints_hc1
avg_depth_bfs	Frac_Unary_Rules_hc1	

solver used is *clasp*[7] version 3.3.2 with default parameter setting. The grounding tool is *gringo*[8] version 4.5.4. We choose 200 CPU seconds as cutoff time.

Analysis of encoding selection results. We performed encoding selection experiments using different machine learning methods on the training set and test set described earlier. The models were trained with the narrowed down set of features and hyper-parameters set to values obtained via the standard hyper-parameter tuning process. The results are shown in Table 4.

The test set contains 156 instances randomly selected from the original data set of 784. The results show that the encodings we used in the experiment have complementary strengths. The oracle solves 98.7%, or 154 out of 156 instances. Compared with Encoding 1, the always-select-best selection method solves 14.7% more instances. We note that 98.7% is the upper bound for the performance that could be achieved by encoding selection.

Our best predictor based on the decision tree method, solves 96.2% of instances, or 150 out of 156. This is very close to 98.7% of solved instances for the always-select-best oracle. Even the worst performing of three machine learning models, based on the *k*-nearest neighbors algorithm, solves 92.9% of instances, much better than any individual encoding. Overall, we find that each of the three machine learning methods we studied provide promising results in terms of the percentage of the solved instances.

6 Summary

We applied machine learning techniques to build performance prediction models for a collection of six encodings of the HC problem that showed complementary strengths on our data set. We designed features to characterize problem instances (some of them problem independent and some reflecting the problem of interest). Finally, we applied three kinds of regression models to construct performance predictors. We used these predictors to select an encoding for an instance to run it on an ASP solver. The results showed performance gain over individual encodings. Moreover, the encoding selection approach came very close to the always-select-best oracle in terms of solved instances. We conclude that the encoding selection method improves the solving capabilities of ASP solvers.

[7] `https://potassco.org`
[8] `https://potassco.org`

Table 4: Test result of encoding selection experiment

	Solved Percentage%	Average Solved Runtime	Number of Wins
Single encoding performance			
Encoding 1	84.0	82.4	25
Encoding 2	71.2	44.0	29
Encoding 3	56.4	30.7	20
Encoding 4	78.8	38.6	28
Encoding 5	57.1	35.4	26
Encoding 6	79.4	48.1	26
Oracle performance			
Oracle	98.7	21.1	
Encoding selection			
Encoding selection (KNN)	92.9	40.2	
Encoding selection (DT)	96.2	42.2	
Encoding selection (RF)	93.6	41.7	

7 Future Work

It is unsatisfactory that the running time of our encoding selection approach is higher (about two times higher in our experiments) than the optimal time of the always-select-best oracle. Closing that gap is a challenge that will require more accurate runtime prediction. One way to attack the problem is to identify more informative features, especially domain-specific features. Also, feature selection can remove irrelevant and redundant attributes and has huge impacts on the performance of machine learning models.

Another factor that affects the performance of solving an ASP problem is a specific solver used to perform the search. The default parameter configuration may be good overall but more often than not will not be optimal on specific instances. Work on parameter configuration, such as ParamILS [13], has shown that a well-chosen parameter configuration can help achieve a performance improvement of over one order of magnitude. Our next step is to combine encoding selection with parameter configuration.

The encoding candidates in our experiments were created and modified manually. However, it is desirable to automate the process, that is, to generate candidates with a tool that is able to analyze an encoding and rewrite it into several equivalent and efficient forms.

Acknowledgements

This work was funded by the NSF under the grant IIS-1707371.

References

[1] M. Alviano, C. Dodaro, W. Faber, N. Leone & F. Ricca (2013): *WASP: A Native ASP Solver Based on Constraint Learning*. In Pedro Cabalar & Tran Cao Son, editors: *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR-2013*, *LNCS* 8148, Springer, pp. 54–66, doi:10.1007/978-3-642-40564-8_6.

[2] M. Bichler, M. Morak & S. Woltran (2016): *lpopt: A Rule Optimization Tool for Answer Set Programming*. In Manuel V. Hermenegildo & Pedro López-García, editors: *Proceedings of the 26th International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR-2016*, *LNCS* 10184, Springer, pp. 114–130, doi:10.1007/978-3-319-63139-4_7.

[3] G. Brewka, T. Eiter & M. Truszczynski (2011): *Answer set programming at a glance*. Commun. ACM 54(12), pp. 92–103, doi:10.1145/2043174.2043195.

[4] M. Buddenhagen & Y. Lierler (2015): *Performance Tuning in Answer Set Programming*. In Francesco Calimeri, Giovambattista Ianni & Miroslaw Truszczynski, editors: Proceedings of the 13th International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR-2015, LNCS 9345, Springer, pp. 186–198, doi:10.1007/978-3-319-23264-5_17.

[5] M. Denecker, J. Vennekens, S. Bond, M. Gebser & M. Truszczynski (2009): *The Second Answer Set Programming Competition*. LPNMR 2009, p. 637654, doi:10.1007/978-3-642-04238-6_75.

[6] E. Erdem, M. Gelfond & N. Leone (2016): *Applications of Answer Set Programming*. AI Magazine 37(3), pp. 53–68, doi:10.1609/aimag.v37i3.2678.

[7] M. Gebser, R. Kaminski, B. Kaufmann & T. Schaub (2012): *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan&Claypool Publishers, doi:10.2200/S00457ED1V01Y201211AIM019.

[8] M. Gebser, L. Liu, G. Namasivayam, A. Neumann, T. Schaub & M. Truszczynski (2007): *The First Answer Set Programming System Competition*. LPNMR 2007, p. 317, doi:10.1007/978-3-540-72200-7_3.

[9] M. Gebser, M. Maratea & F. Ricca (2017): *The Design of the Seventh Answer Set Programming Competition*. In Marcello Balduccini & Tomi Janhunen, editors: Proceedings of the 14th International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR-2017, LNCS 10377, Springer, pp. 3–9, doi:10.1007/978-3-319-61660-5_1.

[10] C. P. Gomes & B. Selman (2001): *Algorithm portfolios*. Artif. Intell. 126(1-2), pp. 43–62, doi:10.1016/S0004-3702(00)00081-3.

[11] N. Hippen & Y. Lierler (2019): *Automatic Program Rewriting in Non-Ground Answer Set Programs*. In José Júlio Alferes & Moa Johansson, editors: Proceedings of PADL 2019, LNCS 11372, Springer, pp. 19–36, doi:10.1007/978-3-030-05998-9_2.

[12] H. H. Hoos, M. T. Lindauer & T. Schaub (2014): *claspfolio 2: Advances in Algorithm Selection for Answer Set Programming*. Theory and Practice of Logic Programming 14(4-5), pp. 569–585, doi:10.1017/S1471068414000210.

[13] F. Hutter, H. H. Hoos, K. Leyton-Brown & T. Stützle (2009): *ParamILS: An Automatic Algorithm Configuration Framework*. J. Artif. Intell. Res. 36, pp. 267–306, doi:10.1613/jair.2861.

[14] P. Kerschke, H. H. Hoos, F. Neumann & H. Trautmann (2019): *Automated Algorithm Selection: Survey and Perspectives*. Evolutionary Computation 27(1), pp. 3–45, doi:10.1162/evco_a_00242.

[15] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri & F. Scarcello (2006): *The DLV system for knowledge representation and reasoning*. ACM Trans. Comput. Log. 7(3), pp. 499–562, doi:10.1145/1149114.1149117.

[16] M. Maratea, L. Pulina & F. Ricca (2014): *A Multi-engine Approach to Answer-set Programming*. Theory and Practice of Logic Programming 14(6), pp. 841–868, doi:10.1017/S1471068413000094.

[17] J. R. Rice (1976): *The Algorithm Selection Problem*. Advances in Computers 15, pp. 65–118, doi:10.1016/S0065-2458(08)60520-3.

[18] B. Selman, D. G. Mitchell & H. J. Levesque (1996): *Generating Hard Satisfiability Problems*. Artif. Intell. 81(1-2), pp. 17–29, doi:10.1016/0004-3702(95)00045-3.

[19] L. Xu, F. Hutter, H. H. Hoos & K. Leyton-Brown (2008): *SATzilla: Portfolio-based Algorithm Selection for SAT*. J. Artif. Intell. Res. 32, pp. 565–606, doi:10.1613/jair.2490.