A Machine Learning System to Improve the Performance of ASP Solving Based on Encoding Selection

Liu Liu[1], Miroslaw Truszczynski[1], and Yuliya Lierler[2]

[1] University of Kentucky, Lexington KY 40506, USA,
{liu.liu,mirek}@uky.edu,
[2] University of Nebraska at Omaha, Omaha NE 68182, USA,
ylierler@unomaha.edu

Abstract. *Answer set programming* (ASP) has long been used for modeling and solving hard search problems. Experience shows that the performance of ASP tools on different ASP encodings of the same problem may vary greatly from instance to instance and it is rarely the case that one encoding outperforms all others. We describe a system and its implementation that given a set of encodings and a training set of instances, builds performance models for the encodings, predicts the execution time of these encodings on new instances, and uses these predictions to select an encoding for solving.

Keywords: answer set programming, encoding selection, machine learning

1 Introduction

Answer set programming (ASP) is a declarative programming paradigm designed primarily for solving decision problems in NP (in particular, problems that are NP-complete), and their search and optimization variants [2,4]. In ASP, an answer-set program (*AS program*) encoding a problem at hand is separate from the data. The latter is represented as a set of facts and forms a special AS program referred to as an instance. To solve the problem for a particular data instance, one combines the problem encoding with the instance into a single AS program. That program is then processed by answer set programming tools, typically a *grounder* and a *solver* such as, for instance, grounder *gringo* [6] and solver *clasp* [7].

As in other programming systems, a problem specification can be encoded in several ways in ASP. Often, this gives rise to numerous equivalent AS programs encoding the same problem. Extensive experience with ASP accumulated in the past two decades suggests that different AS programs for a given problem may differ significantly in their performance. Namely, it is rarely the case that the same encoding performs best (under a selected grounder-solver tool) across all data instances to the problem. This suggests that the availability of multiple encodings can be turned into an asset that might improve the efficiency of ASP.

Efforts were made to understand how the performance of ASP tools depends on ways AS programs encode the constraints of the problem. Automated encoding rewriting tools [1] were proposed based on tree decomposition techniques on the level of a grounder, some with Machine Learning models [18] to guide the rewriting directions that grounders may follow to produce smaller grounding. Researchers also explored the possibility of exploiting multiple solving algorithms, both outside of ASP and in ASP. *Portfolio solving* and *algorithm selection* [19,10,5,15] emerged from these efforts. The idea that we propose and explore here is to extend the scope of these approaches by taking advantage of multiple equivalent encodings for a problem, not necessarily arranged to minimize the size of the ground program. Specifically, we present a system that supports what we name *encoding portfolio* or *encoding selection* (in the last section, we also briefly discuss *encoding scheduling*). We call this encoding selection platform an ESP. The ESP system exploits collections of equivalent ASP encodings for a problem supplied by the user and can also generate additional encodings by applying simple rewritings to the encodings supplied. Thus, we provide programmers with a tool that automates systematic navigation through available encodings for the problem targeting performance improvements.

The remainder of the paper is organized as follows. Section 2 describes the architecture of the ESP, Section 3 presents a case study to illustrate how it works, and Section 4 concludes with a discussion of future work. Throughout the paper, we list insights and conclusions we arrived at while developing and using ESP. They offer practical tips on utilizing the ESP by ASP practitioners. In our discussion, we use the *hamiltonian cycle* (HC) problem to illustrate functions of the components of the ESP and their operation.

2 The encoding selection platform ESP

Figure 1 shows the architecture and processes involved in the ESP encoding selection platform. The word *Input* in the flowchart indicates input data and parameters to be supplied by the user. In particular, the user provides encodings for a problem to be solved, instances of this problem, and problem specific features, if available. Components shown inside boxes denote processes implemented with the ESP. These include encoding rewriting, performance data collection, encoding candidate generation, feature extraction, machine learning modeling, per-instance encoding selection, and solving. Other annotations point at outcomes of different processes or tools utilized by the system. The ESP uses such tools as encoding rewriting system *AAgg* [3] and feature generator *claspre* [5](*claspre* is a sub-component of portfolio answer set solver *claspfolio*; it is available as a stand alone tool at https://potassco.org/labs/claspre/).

The ESP, a description of the system requirements, and instructions on how to use it are available at http://www.cs.uky.edu/ASPEncodingOptimization/esp/. Although the platform consists of several components, each part can be executed separately. Thus, users can upload encodings and instances and run all the processes, or only run some selected ones.

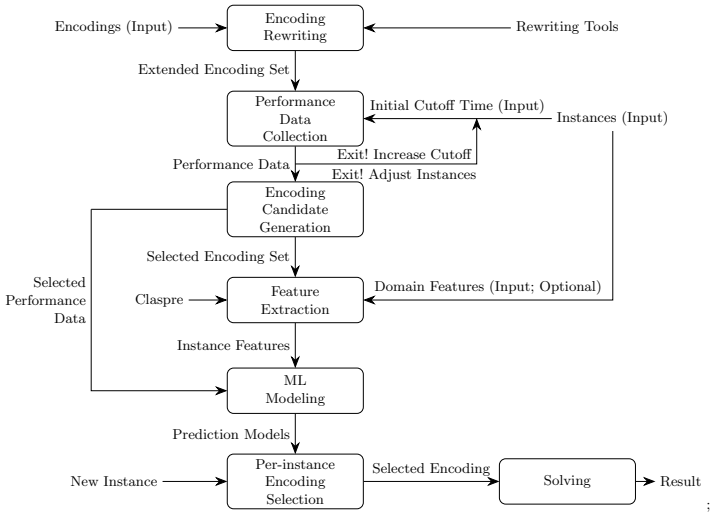

Fig. 1: A flowchart to the encoding selection platform

The ESP exploits the availability of multiple encodings of a problem to assist the user in obtaining performance improvements of ASP-based solutions to the problem. Improved performance means an increased number of instances solved for an application and decreased time spent on these instances. The platform is general purpose and can be applied to arbitrary problems solved by ASP. However, any specific use of the ESP tool assumes a concrete problem at hand. In what follows we often use the letter P to refer to that problem.

2.1 Encoding Rewriting

Encodings The ESP expects a user to supply at least one AS program for a given problem P. However, in most cases, the user will provide several encodings for the problem. The supplied encodings are rewritten by the encoding rewriting tool available in ESP. The *extended set of encodings* is the basis for further processing that aims to select a subset of "promising" encodings to be used when solving new instances of problem P. We comment on how performance data guides the selection process implemented in ESP later in the paper.

To show examples of possible input encodings that the user might supply to the ESP, we consider a well-known Hamiltonian cycle problem. The first encoding for the HC problem follows

```
1 { hpath(X,Y) : link(X,Y) } =1:-node(X).
2 { hpath(X,Y) : link(X,Y) } =1:-node(Y).
3 reach(X) :- hpath(1,X).
4 reach(Y) :- reach(X),hpath(X,Y).
5 :- not reach(X),node(X).
```

Rules 1 and 2 model the requirement that the number of selected edges leaving and entering each node is exactly one. Rules 3 and 4 define the concept of reachability from node 1. Constraint 5 guarantees that every node is reachable from node 1 by means of selected edges only. Another (but equivalent) encoding can be obtained by replacing rule 3 in the encoding above with rule `reach(1).`

Encoding rewriting tools The current version of the ESP employs a non-ground program rewriting tool *AAgg*. It is used to generate additional encodings from those provided by the user. The original version of this system, developed by Dingess and Truszczynski [3], produced rewritings by reformulating applicable rules by means of cardinality aggregates. The version integrated into the platform also supports rewritings that eliminate cardinality constraints. In the future, we will incorporate in the ESP other rewriting tools, such as *Projector* [12] and *Lpopt* [1], and provide an interface for users to incorporate their own tools.

2.2 Performance Data Collection

Instances Benchmark instances must be provided by the user. They are used to extract data on the performance of a solver on each of the selected encodings, to support feature extraction, and to form the training set used by machine learning tools to build encoding performance models. When a solver finds a solution to an instance in a short amount of time no matter what encoding is used, or when the solver times out no matter what encoding is used, the instance offers no insights that could inform encoding selection. Only instances that are not too easy and not too hard are meaningful. We call such instances *reasonably hard*.

More specifically, reasonably hard instances are determined by the time T_e specifying when the execution time is long enough not to view an instance as easy, and the time T_{max} specifying the cutoff time. At present, the user inputs only the cutoff time T_{max}; the system then sets $T_e = T_{max}/7$. How to select the initial value of T_{max} depends on the available computing resources, as well as the time budget for solving incoming instances of the problem at hand.

Once a user provides the ESP with the initial set of instances, and the parameter T_{max}, and the extended set of encodings is produced by rewriting, the ESP computes the *performance data* while automatically adjusting cutoff time T_{max} two times, each time doubling it, if too many time-outs occur. The ESP continues with the next step when the collected performance data suggests that the current instance set contains a sufficient proportion of reasonably hard instances. More specifically, the platform selects randomly a subset of $\min(\max(20, \min(size/10, 100)), size)$ instances to test the hardness (here $size$ denotes the size of the entire input set of instances, which is expected to be greater than 500). All encodings are then run with each selected instance. An instance is easy when all encodings solve it within time T_e. An instance is too hard when it is not solved by any encoding within the cutoff time T_{max}. All other instances are *reasonably hard*. If at least 30% of instances in the selected subset are reasonably hard, the entire input data set is *valid*. If not and also no more than 30% of instances time out on each encoding, the ESP exits and

Instance_id	ham1	ham2	ham3	ham4	ham5	ham6
insttri200_33_1	114.96	0.61	200.00	12.52	2.89	2.14
insttri200_41_2	15.22	49.10	200.00	200.00	0.65	0.49
insttri200_49_1	13.22	0.16	200.00	0.23	200.00	0.62
insttri200_57_1	47.86	200.00	0.45	7.85	200.00	200.00
insttri200_57_2	41.98	200.00	59.55	53.86	0.24	1.08
insttri200_65_2	15.61	1.02	200.00	26.42	45.46	25.65
insttri200_71_10	1.22	200.00	139.17	14.84	200.00	200.00
insttri200_81_8	200.00	38.08	200.00	32.40	200.00	200.00
insttri200_91_5	200.00	74.90	116.11	1.45	40.20	200.00
insttri200_131_10	8.31	132.25	2.85	22.46	42.22	58.86

Table 1: Runtime of valid structured dataset for the HC problem

declares the original input instance set "too easy." Otherwise, the selected subset is "too hard" and the system increases T_{max} by doubling it (and adjusting T_e accordingly). After doubling, the ESP again runs all encodings with all instances. If, with the new values for T_{max} and T_e, the number of reasonably hard instances becomes 30% or more, the ESP stops and declares the original input instance set as valid. Otherwise, the ESP doubles T_{max} one more time and repeats. The possible outcomes are then: "too easy, "too hard," and valid. In the first two cases, the user is informed and asked to adjust T_{max} and the hardness of the input instances accordingly. In the last case, the ESP checks if there are at least 500 reasonably hard instances in the entire input set. If not, the ESP exits and returns to the user the numbers of instances in the set that are easy, hard and reasonably hard, and requests that the user updates the input instance set. (The first phase of the process aims to save time, if the input instance set is too hard, with a high probability the ESP will return this decision without having to process the entire data set.)

We now provide insights into the instance generation/selection process by focusing on the HC domain. Table 1 shows performance data collected by running the *gringo/clasp* tools with six encodings of the HC problem on several instances of that problem, that is, directed graphs. All graphs are generated randomly from a certain space or *model* of graphs. Graphs in the model used in this example are built by removing directed edges from *triangle grid* graphs. Nodes of those graphs are arranged in layers, the first layer has one node, the next two nodes, and so on. The external nodes form a triangle; each internal node is connected by two-directional edges with two neighboring nodes in its own layer, two neighboring nodes in the layer above and two more in the layer below. Such graphs have Hamiltonian cycles. Graphs in our example are subgraphs of a 19-layer triangle grid with 190 nodes. When the number of removed edges is small, the graphs have Hamiltonian cycles with a probability close to 1. As the number of removed edges grows, we reach the point (known as the phase transition [20]), when this probability drops quickly and becomes close to 0. The phase transition region

contains graphs with and without a Hamiltonian cycle with, roughly, the same probability. Moreover, the solving time becomes significant.

Table 1 shows a selection of instances that are reasonably hard (we took $T_{max} = 200$ seconds as the cutoff, and set $T_e = T_{max}/7 = 28.57$ seconds). Building a set of reasonably hard instances (with respect to T_e and T_{max}) may still yield a data set that is relatively easy (when execution times, while greater than T_e do not come close to the cutoff time). An additional requirement one might want to impose on a "good" set of instances is that each encoding must time out on at least some instances in the set. This is the case for the set of instances in Table 1 In addition to a consideration of hardness, a valid instance set must evince complementary performance from the selected encodings. That is, no encoding must be uniformly better than others, in fact, *each* encoding must have its area of strength where it performs better than others. This is the case of the set of instances in Table 1. For example, on the instances *insttri*200_33_1 and *insttri*200_57_1 the *ham 2* and *ham 3* exhibit "opposite" performance: *ham 2* is the winner on the first instance while *ham 3* is the winner on the second one. We can observe that each instance has its own best encoding and the order of per-instance best encodings in the table are 2, 6, 2, 3, 5, 2, 1, 4, 4, 3. In particular, each encoding is the winner on at least one instance. If a dominant encoding exists (performs best on all instances), encoding selection in such case is meaningless. The ESP will inform the user about it.

Building meaningful sets of reasonably hard instances is difficult. They can be derived from the instances submitted to the past ASP competitions [8,9] in the NP category, or can be obtained by building random models of instances (as in our running example above) and finding the right settings for the model's parameters. Incorporating some structure in the model (as in the running example) offers a better chance for meaningful instances as purely random instances without any structure are often quite easy. Finally we note that to support encoding selection a large data set with at least 500 instances is needed.

The concept of an oracle helps evaluate the potential for performance improvements by encoding selection. An *oracle* is a non-deterministic algorithm that always selects the best encoding to run with a given instance. Typically, oracle's performance is much better than the performance of any individual encoding. This is the case for the data set in Table 1. Thus, the task of selecting correct encodings on a per-instance basis becomes meaningful.

Cutoff time penalization Performance data represents the effectiveness of different encodings under a chosen ASP solving tool. It is obtained by processing all encodings with all instances, using a selected solver (such as the *gringo* grounder and the *clasp* solver in some selected configuration). Each individual run should be limited to the selected cutoff time, since some encodings combined with some instances may take a large amount of time before terminating. To assess the quality of an encoding, one must account for timeouts. When an instance reaches timeout, the ESP considers the number of encodings reaching timeout for the instance, and a penalized runtime is given. The ESP uses an approach we call PARX, which takes for the runtime of a timeout instance the

cutoff time multiplied by X, where X is the number of encodings that time out on this instance. For example, when this method is used, for the instances in Table 1, the penalized runtime for *insttri*200_33_1 is 200.00 for *ham3*, and for *insttri*200_41_2 is 400.00 for both *ham3* and *ham4*.

2.3 Encoding Candidate Selection and Feature Extraction

In this stage of the process, the ESP analyzes the performance data obtained for the extended set of encodings. The system selects a subset of the extended encoding set that consists of encodings that are most effective and that together demonstrate *run-time diversity*. At least two and no more than six encodings are selected.

To estimate the effectiveness of the encoding, we assign it a score. The score is affected by the percentage of the solved instances, the number of instances for which the encoding provided the fastest solution, and the average running time on all solved instances.

The selected encodings are organized into groups. Specifically, we consider as a group the entire set of selected encodings, if only two or three encodings were selected. Otherwise, the set of selected encodings has i encodings, where $i = 4, 5$ or 6, and we consider the group of three top-scoring encodings (the scoring is discussed in an earlier section), four top-scoring encodings etc., for the total of $i - 3$ groups (two groups if $i = 4$, three groups if $i = 5$ and four groups if $i = 6$).

To support machine learning of performance prediction models for the selected encodings, we identify instances of problem P with their *feature vectors*. In other words, each instance-encoding pair is mapped into an abstraction captured by a number of properties/features that hold for this pair. Our system relies on two sets of features. First, it exploits features that can be defined based on the generic structure of the propositional program obtained by grounding a given instance-encoding pair. To this end, we take advantage of the system *claspre* [5]. Second, the platform uses domain specific features related to problem P supplied by the user.

Claspre features *Claspre* is a system designed to extract features of ground ASP programs. The extracted features fall into two groups: static and dynamic. Static ones contain features about atoms, rules, and constraints. For instance, they include such program properties as the number of rules, unary rules, choice rules, normal rules, weight rules, negative body rules, binary rules, ternary rules, etc. In total, *claspre* computes 38 static features. To extract dynamic features for a ground program, *claspre* runs *clasp* on it for some short amount of time. *Clasp* returns the information about the solving process. This information is then turned into (dynamic) features of the program. The ESP uses these features for the instance-encoding pair that defined the program processed by *claspre*. These features are based on information collected after each restart performed by *clasp*, with the number of restarts being a parameter of the process. Allowing for more restarts result in features that are usually more accurate to represent a problem, but the process requires extra runtime. Overall, *claspre* computes 25

dynamic features per each restart and the platform uses features for two restarts. However, extremely easy instances have no claspre features since they are solved during the feature extraction process, and no much information can be collected for them.

Domain features *Claspre* features are oblivious to the nature of given problem P represented by the specific instance-encoding pair. Domain features relevant to the nature of P, expressed by properties of an instance to P often provide additional useful characteristics of the instance (note that these features are independent of properties of a particular encoding). For example, if instances for problem P are graphs, possible features may include the number of nodes in a graph, the number of edges, the minimum and maximum degrees, as well as measures reflecting connectivity and reachability properties. Availability of domain features often improves the performance of the platform. The ESP framework provides an interface for the user to supply domain features for their problems at hand. Obviously, the ultimate selection of such features as input to the platform depends on the problem being solved. Indeed, different features may be relevant to, say, problems of graph colorability and Hamiltonian Cycle. In the HC problem, the existence of long paths plays a role, and several features related to this property may be derived from running the depth-first search on the instance. Sample domain specific features for the HC problem [17] follow

- numOfNodes: the number of nodes in a graph;
- avgOutDegree: the average of outdegree of nodes;
- depthDfs1stBackJump: run depth-first search from node 1, return the depth of the first backjump, where the algorithm discovers no new nodes;
- depthBacktoRoot: run depth-first search from node 1, return the depth of a node that has a back edge to node 1;
- minDepthBfs: run breadth-first search from node 1, return the depth of the first node that has no outward edges to new nodes.

We used these features in our running example of the case study of the use of the ESP for tuning performance within the HC domain.

The output of this phase is a table whose rows correspond to instance-encoding pairs and contain the values of all its features.

2.4 Machine Learning Modeling and Solving

The goal of machine learning techniques within this project is to build encoding performance predictors based on performance data and features explained above. Once these predictors are constructed for a problem P at hand, they can be used to select the most promising encoding for processing an input instance of P. To build machine learning models, one can use regression or classification approaches. The former predicts each encoding's performance expressed as the running time, and then selects the most promising one by comparing the predicted performance. The latter method builds a multi-class machine learning model and directly selects the most promising encoding from a collection of candidate encodings. Our earlier experimental analysis (outside of the scope of this

paper) indicates that regression approaches work better than classification. As a result, at present the ESP supports the construction of regression models only.

The set of selected encodings (at least two and at most six arranged into one to four groups, as discussed in Section 2.3) is the basis for machine learning algorithms currently used by the ESP. The ESP performs learning for each of the group based on instance features and instance performance data restricted to encodings in the group. Supervised ML techniques that we use here are trained on ⟨instance features, instance performance⟩ pairs for each encoding in the group. Once a model is trained it yields a mapping from instance features to the estimated performance of a targeted encoding. The ESP builds runtime prediction models for each encoding and selects the encoding with the minimum predicted runtime. We now explain the detailed design below.

Features selection *Claspre* features are collected for instance-encoding pairs. The features representing an instance consist of the features of that instance when paired with all encodings in the group being considered (88 features for each instance-encoding pair possible within the group) and the domain specific features of the instance. This is a large number of features that may cause the poor computational performance of machine learning algorithms. To address this issue, the ESP reduces the number of features by further processing. For *claspre* features, the ESP first performs feature selection inside features related to one encoding. All subsets (from 40% to 70%) of features are selected for each encoding based on standard deviation reduction [11]. These subsets of selected features are trained and validated on different data splits from the whole dataset, and validation results are compared. The subset with the lowest average mean squared error is selected as the selected features for the instance-encoding pair. When the validation results for all encodings within the group are compared, the best subset is selected as the claspre features of the group. A subset of domain specific features is selected separately and then combined with selected claspre features to form the final set of features.

Hyper-parameters tuning At present, the ESP supports three well-known machine learning algorithms: k-Nearest Neighbors (kNN), Decision Tree (for the review of these two methods see, for instance [21]), and Random Forest [13]. In each case, the performance of the algorithm depends on the choice of hyper-parameters (for instance, the number k of nearest neighbors to consider for the kNN method). Hyper-parameters tuning is an important step within training process of machine learning. We implemented the grid-search method for hyper-parameter searching in the ESP and combined it with the 10-fold cross-validation (for the description of k-fold cross validation method see, for instance, [16]) to improve the generalization of the obtained model.

Assessment of learned models The result of the learning (for each group) is the collection of performance models obtained by applying each of the machine learning methods implemented in the ESP. These models are compared by evaluating their performance on the 5-fold cross validation approach. For each round, the platform trains models on the training set, predicts the runtime of the corresponding encoding for instances on the validation set, and selects the

most promising encoding on a per-instance basis. Average solving percentage (primary criteria) and average solved time (secondary criteria, for the case of a tie) for multiply runs are compared for all learned models of all groups, and the best model among them is selected as the solution of the ESP.

Per-instance Encoding Selection and Solving Once the platform computes and selects the model based on the performance of cross-validation results, it will use this model to solve problems provided as new instances. That is, given a new instance, it will apply the encoding selected by the model computed and selected in the machine learning phase. Specifically, the platform extracts features of the instance that are relevant to (are used by) the model, applies the model to select the encoding (the one with the lowest estimated run time is selected), and applies the solver to the instance combined with the selected encoding.

3 Experimental analysis

We tested the performance of the ESP using the Hamiltonian Cycle problem. We now describe the experimental setup and results.

Experimental setup All our experiments were performed on a computer with Intel(R) Core(TM) i7-7700 CPU and 16 GB Memory, running on Linux 5.4.0-91-generic x86_64. The input to the platform consists of *six* HC encodings and *one thousand* structured graph instances. The instance set consists of graphs from four different structures used in our previous work on the HC problem [17]. The cutoff time is initially set to 200 CPU seconds. The system decided that the original cutoff time was appropriate and the cutoff time was not increased.

Each encoding was run on all instances and runtime was recorded. All instances were grounded with *gringo* version 5.2.2 and solved by *clasp* version 3.3.3 with default configurations. Only solving time was counted as runtime, while grounding time was not counted. It took ten days to collect the performance data for all six encodings. Six encodings are ranked according to their performance. They give rise to four encoding groups (top three, top four, top five and top six). For all the instances, *claspre* features are extracted and graph specific features are provided. Out of 1000 originally provided graph instances, the ESP platform determined 775 to be reasonably hard.

The data set is split into the training and the validation set (80% of instances) and the test set (20% of instances). The former is used by the ESP to build models and select the best one. The test set is used in the experiments to evaluate the performance of the platform.

Experimental results The test results are shown in Table 2. Instances from the test set (in other words, instances that ESP has never seen before) are used to compile this table. The assessment of the kind is part of the platform.

The first part of the left table shows the performance of individual encodings: solving percentage (solving%) and average solved runtime (avg_solved_t) are reported. The solving percentage records the percentage of instances each encoding can solve, and the average solved time counts the average runtime for solving these instances. The average solved runtime does not accounts for

	solving%	avg_solved_t
Individual performance		
ham1	61.93	34.09
ham2	74.83	54.31
ham3	74.19	55.37
ham4	58.06	35.63
ham5	**78.70**	**71.35**
ham6	68.38	45.80
Oracle performance		
Oracle	**95.48**	**21.64**
system solution		
RFgroup3	**88.38**	**40.81**

	solving%	avg_solved_t
Other solutions		
DTgroup4	85.16	39.14
RFgroup4	87.09	40.80
kNNgroup4	80.00	40.88
DTgroup3	87.09	36.84
KNNgroup3	80.00	41.68
DTgroup2	73.54	57.74
RFgroup2	78.06	60.81
KNNgroup2	77.41	52.54
DTgroup1	78.06	61.74
RFgroup1	79.35	56.72
KNNgroup1	76.77	57.11

Table 2: Performance of individual encoding, oracle, system solution, and other solutions

unsolved instances, because different penalty methods may result in different average overall runtime. The second part reports the oracle performance, which selects the best encoding for each instance, representing the *upper bound* on what is possible with the encoding selection method. The third part shows the result for the method selected by the ESP. The right part shows the performance of other solutions (intermediate performance models), which are obtained by the system, but not selected as the best solution by ESP. The individual performance shows that the best individual encoding *ham5* can solve 78.70% of all instances. Thus, we can use the performance of this encoding as the *baseline* performance. Even though *ham5* solves the most instances, it does not have the lowest average solved running time. In fact, it has the largest average solved runtime. The encoding *ham1* is the fastest in terms of average solved runtime, but it only solves 61.93% of instances. The oracle results point at the fact that there is a huge performance gain by selecting the best encoding for each instance. It solves 95.48% of instances, with an average solving time of 21.64. Compared with *ham5*, the success percentage of the always-select-best oracle is 16.78 percentage points higher. Overall, the table shows the encodings in the test set have complementary strengths. Each of them can solve a certain fraction of instances, but when combined, they can solve much more.

The system solution with the best cross validation result is RFgroup3, the random forest model based encoding selection from encoding group 3, which consists of top five encoding candidates. When tested on the test set, it solves 88.38% of instances, 9.68 percentage points more than the best individual encoding *ham5*, and is also the best solution among all models. This confirms that the platform is able to generate solutions that improve the performance of ASP. The results also show all other solutions generated using the platform almost overperform the individual best. For example, these machine learning based so-

lutions built for group 4 and group 3, which consists of six and five encoding candidates respectively, all contribute better results than *ham5*. Solutions built for group 2 and group 1 are worse since they are based only on top four and top three encoding candidates. We also observe the group 3, which consists of five encoding candidates, provides better results for corresponding models than other groups.

4 Conclusion and future work

In this article, we described the system ESP that can automatically improve the performance of ASP through encoding rewriting and selection. Many of the processes involved can run separately. This means that one can skip over some parts of the overall process if the necessary inputs for later steps were already computed before. *We view the platform as a valuable tool for the ASP practitioners geared to assist them with performance analysis and encoding selection tasks in a systematic and principled manner.* This paper is meant to assist them in understanding its inner components. Our experiments show that for the HC problem the ESP selects encodings and builds performance prediction models that lead to improvements in ASP solving. Despite this success, the ESP requires more insights into fine-tune machine learning methods for selecting encodings and building accurate performance predicting models. Indeed, our experiments with other problems are mixed. In some cases (for instance, the graceful graph labeling[3]), the ESP performs comparably with the best individual encodings (but not better yet), in some other cases (graph coloring) it performs worse.

Our future work will aim to address the present shortcomings. First, we will expand the encoding rewriting module, where we plan to incorporate additional encoding rewriting tools, to increase the runtime diversity of the encodings the system generates. Further, we plan to develop techniques combining encoding selection with an earlier work on solver selection. In particular, we will study learning models to estimate for a given instance the performance of a pair (*clasp* configuration, problem encoding). Second, we will incorporate into the ESP techniques constructing *schedule* [14] based solutions. In this approach, several encodings are selected to be processed by ASP tools in a certain order and for the total time equal to the cutoff limit, with each encoding receiving a certain share of the time budget.

Acknowledgments

The authors acknowledge the support of the NSF grant IIS 1707371.

References

1. Bichler, M., Morak, M., Woltran, S.: lpopt: A rule optimization tool for answer set programming. Fundamenta Informaticae **177**(3-4), 275–296 (2020)

[3] https://en.wikipedia.org/wiki/Graceful_labeling

2. Brewka, G., Eiter, T., Truszczynski, M.: Answer set programming at a glance. Commun. ACM **54**(12), 92–103 (2011). DOI 10.1145/2043174.2043195
3. Dingess, M., Truszczynski, M.: Automated aggregator - rewriting with the counting aggregate. EPTCS **325**, 96–109 (2020). DOI 10.4204/EPTCS.325.17
4. Erdem, E., Gelfond, M., Leone, N.: Applications of answer set programming. AI Magazine **37**(3), 53–68 (2016). DOI 10.1609/aimag.v37i3.2678
5. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T., Schneider, M., Ziller, S.: A portfolio solver for answer set programming: Preliminary report. In: Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR), pp. 352–357. Springer-Verlag (2011)
6. Gebser, M., Kaminski, R., Konig, A., Schaub, T.: Advances in gringo series 3. In: Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR), pp. 345–351. Springer (2011). DOI 10.1007/978-3-642-20895-9_39. URL http://dx.doi.org/10.1007/978-3-642-20895-9_39
7. Gebser, M., Kaufmann, B., Schaub, T.: Conflict-driven answer set solving: From theory to practice. Artificial Intelligence **187**, 52–89 (2012)
8. Gebser, M., Maratea, M., Ricca, F.: The sixth answer set programming competition. Journal of Artificial Intelligence Research **60**, 41–95 (2017)
9. Gebser, M., Maratea, M., Ricca, F.: The seventh answer set programming competition: Design and results. Theory and Practice of Logic Programming **20**(2), 176–204 (2020). DOI 10.1017/S1471068419000061
10. Gomes, C.P., Selman, B.: Algorithm portfolios. Artif. Intell. **126**(1-2), 43–62 (2001). DOI 10.1016/S0004-3702(00)00081-3
11. Guyon, I., Elisseeff, A.: An introduction to variable and feature selection. J. Mach. Learn. Res. **3**(null), 1157–1182 (2003)
12. Hippen, N., Lierler, Y.: Automatic program rewriting in non-ground answer set programs. In: International Symposium on Practical Aspects of Declarative Languages, pp. 19–36. Springer (2019)
13. Ho, T.K.: Random decision forests. In: Proceedings of 3rd international conference on document analysis and recognition, vol. 1, pp. 278–282. IEEE (1995)
14. Hoos, H., Kaminski, R., Schaub, T., Schneider, M.: aspeed: Asp-based solver scheduling. In: Technical Communications of the 28th International Conference on Logic Programming (ICLP'12). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2012)
15. Hoos, H., Lindauer, M., Schaub, T.: claspfolio 2: Advances in algorithm selection for answer set programming. Theory and Practice of Logic Programming **14**(4-5), 569–585 (2014)
16. Kohavi, R.: A study of cross-validation and bootstrap for accuracy estimation and model selection **14** (2001)
17. Liu, L., Truszczynski, M.: Encoding selection for solving hamiltonian cycle problems with asp. EPTCS **306**, 302–308 (2019). DOI 10.4204/EPTCS.306.35
18. Mastria, E., Zangari, J., Perri, S., Calimeri, F.: A machine learning guided rewriting approach for asp logic programs. Electronic Proceedings in Theoretical Computer Science **325**, 261–267 (2020). DOI 10.4204/EPTCS.325.31
19. Rice, J.R.: The algorithm selection problem. Advances in Computers **15**, 65–118 (1976). DOI 10.1016/S0065-2458(08)60520-3
20. Selman, B., Levesque, D.G.M.H.J.: Generating hard satisfiability problems. Artificial intelligence **81**(1-2), 17–29 (1996)
21. Wu, X., Kumar, V., Quinlan, J.R., Ghosh, J., Yang, Q., Motoda, H., McLachlan, G.J., Ng, A., Liu, B., Philip, S.Y., et al.: Top 10 algorithms in data mining. Knowledge and information systems **14**(1), 1–37 (2008)