Deep learning for the generation of heuristics in answer set programming: a case study of graph coloring

Carmine Dodaro[1][0000−0002−5617−5286], Davide Ilardi[2][0000−0001−5111−8998], Luca Oneto[2][0000−0002−8445−395X], and Francesco Ricca[1][0000−0001−8218−3178]

[1] DeMaCS, University of Calabria, Italy
name.surname@unical.it
[2] DIBRIS, University of Genova, Italy,
name.surname@unige.it

Abstract. Answer Set Programming (ASP) is a well-established declarative AI formalism for knowledge representation and reasoning. ASP systems were successfully applied to both industrial and academic problems. Nonetheless, their performance can be improved by embedding domain-specific heuristics into their solving process. However, the development of domain-specific heuristics often requires both a deep knowledge of the domain at hand and a good understanding of the fundamental working principles of the ASP solvers. In this paper, we investigate the use of deep learning techniques to automatically generate domain-specific heuristics for ASP solvers targeting the well-known graph coloring problem. Empirical results show that the idea is promising: the performance of the ASP solver WASP can be improved.

Keywords: answer set programming, deep learning, heuristics, graph coloring

1 Introduction

Answer Set Programming (ASP) [5] is a well-established declarative AI formalism for knowledge representation and reasoning. ASP is a popular paradigm for solving complex problems mainly because it combines high modeling power with efficient solving technology [7]. The rich language, the intuitive semantics and the availability of efficient solvers are the key ingredients of the success of ASP on solving several industrial and academic problems [10].

Modern ASP solvers employ an extended version of the Conflict-Driven Clause Learning (CDCL) algorithm [16]. As a matter of fact, the performance of a CDCL solver heavily depends on the adoption of heuristics that drive the search for solutions. Among these, the heuristic for the selection of the branching literal (i.e., the criterion determining the literal to be assumed true at a given stage of the computation) can dramatically affect the overall performance of an implementation [8]. As default strategies, ASP implementations feature very good

general purpose heuristics belonging to the family of VSIDS [9]. However, they may fail to compute solutions of the hardest problems in a reasonable amount of time. Nonetheless, it is well-known that the performance of ASP solvers can be improved by embedding domain-specific heuristics into their solving process [2, 8, 11], and this is particularly true in the case of real-world industrial problems [24]. However, the development of domain-specific heuristics often requires both a deep knowledge of the domain at hand and a good understanding of the fundamental working principles of the ASP solvers. Thus, one might wonder whether it is possible to ease the burden of the ASP developer by leaving the task of defining proper heuristics to a machine that can learn effective heuristics from the observation of the behavior of solvers on instances from the same domain. A first positive answer to this question was provided by Balduccini in [3] who proposed the DORS framework, where the solver learns domain-specific heuristics while solving instances of a given domain [3]. The DORS framework was implemented in SMODELS, yielding interesting performance improvements. However, DORS was tailored for DPLL-style algorithms and we are not aware of any attempt to experiment with automatic learning of domain heuristics in modern solvers. Starting from the observation that the recent success of AI technology was largely propelled by the developments in deep neural networks [4], which proved to be very effective tools for solving tasks where the presence of humans was considered fundamental; we decided to investigate the use of deep learning techniques to automatically generate domain-specific heuristics for CDCL-based ASP solvers.

This paper presents our first results on employing neural networks to improve the performance of an ASP solver, and to this end, we targeted the well-known graph coloring problem as a use case. The heuristic is learned by observing the behavior of the ASP solver WASP [1] on a test set of instances randomly sampled from a population, where each sample corresponds to an ASP instance. The proposed neural network model takes inspiration from previous experiments conducted by Selsam and Bjørner in [22] and possesses a particular structure specifically designed for being invariant to permutations between literals and their negations, between literals belonging to the same rule and, finally, between rules themselves. The neural network is then trained on the test set, and the resulting model is used to alter the initial values of the heuristic counters used by WASP default heuristics so to make the most promising choices first. Empirical results show that the idea is promising: the performance of WASP can be improved by plugging-in automatically-generated neural domain heuristics.

2 Background

2.1 Graph coloring problem

The graph coloring problem consists of assigning colors to nodes of a graph, such that two connected nodes do not share the same color. More formally, let C be a set of colors and let $G = (N, L)$ be an undirected graph, where N is a set of natural numbers representing the nodes of G, and $L \subseteq N \times N$ be a set of

links between nodes in N. The graph coloring problem consists of finding a total function $col : N \mapsto C$ such that $col(n_1) \neq col(n_2)$ for each $(n_1, n_2) \in L$. The following example shows an ASP encoding of the graph coloring problem. Note that the encoding represents a simplified version of the one used in the recent ASP Competitions [7].

Example 1 (ASP encoding of the graph coloring problem.). Let C be a set of colors and $G = (N, L)$ be a graph. Let Π_G^C be the following program:

$$\begin{aligned}
col(n, c) &\leftarrow \sim ncol(n, c) && \forall n \in N, c \in C \\
ncol(n, c) &\leftarrow col(n, c_2), c \neq c_2 && \forall n \in N, c \in C \\
ncol(n, c) &\leftarrow col(n_2, c) && \forall (n_2, n) \in L \ s.t. \ n_2 < n, c \in C \\
colored(n) &\leftarrow col(n, c) && \forall n \in N, c \in C \\
\bot &\leftarrow \sim colored(n) && \forall n \in N
\end{aligned}$$

If $C = \{b, g\}$ and $G = (\{1, 2\}, \{(1, 2)\})$, then Π_G^C is the following program:

$$\begin{aligned}
r_1 &: col(1, b) \leftarrow \sim ncol(1, b) & r_2 &: col(1, g) \leftarrow \sim ncol(1, g) \\
r_3 &: col(2, b) \leftarrow \sim ncol(2, b) & r_4 &: col(2, g) \leftarrow \sim ncol(2, g) \\
r_5 &: ncol(1, b) \leftarrow col(1, g) & r_6 &: ncol(1, g) \leftarrow col(1, b) \\
r_7 &: ncol(2, b) \leftarrow col(2, g) & r_8 &: ncol(2, g) \leftarrow col(2, b) \\
r_9 &: ncol(2, b) \leftarrow col(1, b) & r_{10} &: ncol(2, g) \leftarrow col(1, g) \\
r_{11} &: colored(1) \leftarrow col(1, b) & r_{12} &: colored(1) \leftarrow col(1, g) \\
r_{13} &: colored(2) \leftarrow col(2, b) & r_{14} &: colored(2) \leftarrow col(2, g) \\
r_{15} &: \bot \leftarrow \sim colored(1) & r_{16} &: \bot \leftarrow \sim colored(2)
\end{aligned}$$

Π_G^C admits two solutions, i.e., $\{col(1, g), col(2, b), ncol(1, b), ncol(2, g), colored(1), colored(2)\}$ and $\{col(1, b), col(2, g), ncol(1, g), ncol(2, b), colored(1), colored(2)\}$ corresponding to the ones of the graph coloring problem. ◁

Moreover, in the following, an ASP program modeling the graph coloring problem is *coherent* if it admits a solution, i.e. there is a function *col* satisfying the requirements, otherwise it is *incoherent*.

2.2 Stable model search

Modern algorithms for computing stable models of a given ASP program Π employ a variant of the CDCL algorithm [16], whose idea is to build a stable model step-by-step starting from a set of literals S initially empty.

During the execution of the algorithm, some of the literals to be added in S (called branching literals) are selected according to a heuristic. Modern implementations use the MINISAT [9] heuristic (or one of its variants), whose key idea is to associate each atom to an *activity* value, that is initially set to 0. This value is incremented by a value *inc*, whenever the atom (or its corresponding literal) is used to compute a learned constraint. Then, after each learning step, the value of *inc* is multiplied by a constant greater than 1, to promote variables that occur in recently-learned constraints. When a branching literal must be selected, the heuristic chooses $\sim a$ (denoted as *negative polarity*), where a is the undefined atom with the highest activity value (ties are broken randomly).

2.3 Deep neural networks

Machine Learning (ML) comprises by now a huge number of algorithms to tackle several different problems [12, 23]. In particular, ML algorithms observe a given data set and refine iteratively their understanding of it through measurable error's estimation. The data set is characterized by an input space $\mathcal{X} \subseteq \mathbb{R}^D$ and an output space $\mathcal{Y}$. The goal consists in determining the unknown function f, associating input and output spaces $f : \mathcal{X} \to \mathcal{Y}$. If $\mathcal{Y}$ is not empty the problem can be defined *supervised* and it determines an ease error estimation [23].

Within this context, the supervised training approach is applied to a popular ML algorithm called Deep Neural Network (DNN) [12], that can be seen as a black-box model able to extract and approximate the function f^*, governing the data set under analysis. The DNN, often called also Deep Feedforward Neural Network or Multilayer Perceptrons (MLPs), maps $y = f(\mathbf{x}; \theta)$, where $\mathbf{x}$ is the set of input features and θ the set of parameters that need to be learned in order to better approximate the function f [12]. The neural adjective takes inspiration from neuroscience, since the simplest unit of such a model (i.e., the neuron) is connected to previous and following units similarly to biological neurons [12]. The neurons are organized in layers, whose number determines the depth of the network under development [12]. The number of neurons per layer defines the width of the model and neurons belonging to the same layer act in parallel [12]. Each unit computes the weighted sum of the previous layer's outputs in addition to an optional bias value [12]. The result will be then processed by a function, also called activation function to emulate the firing activity of the biological neuron, and will be passed to the next layer or to the model's output according to its position within the network. The equation $y = \sigma(\mathbf{w}^T\mathbf{x} + b)$ summarizes the computations performed by each neuron, where $\mathbf{w}$ represents the vector of weights connecting each neuron with previous layer's ones, $\mathbf{x}$ is the vector of inputs coming from previous layer, b corresponds to the bias, σ identifies the activation function characterizing the current neuron and all its layer's neighbors and, finally, y is the neuron's output. As described above, y can be passed as input to following neurons or can be directly interpreted as the model's output. Typically, in a binary classification scenario [23] like the one proposed in this paper, the DNN is asked to determine if the given input belongs or not to a specific class. Consequently, the output space $\mathcal{Y} \in \{0, 1\}$ [23]. Even though DNNs and MLPs give the human practitioners the possibility not to identify the precise function to estimate the desired non-linearity, as it can be inferred from the universal approximation theorem [15], it is still their responsibility to design the architecture and to tune properly its hyperparameters $\mathcal{H}$ through a Model Selection (MS) procedure [20]. It is fundamental to perform a reasoned MS and to choose properly the values to be assigned to the hyperparameters in order to obtain reasonable results and a good level of generalization. Thereafter, the resulting model will pass through an Error Estimation (EE) phase [20], during which its performances will be evaluated on a specific test set.

The width and the depth of the model, the activation functions of the various layers and the connections between neurons all fall into the architectural

parameters to be defined. In addition to these, developers should choose the proper optimization algorithm, the size of the samples' batch to be processed before a backward propagation phase [21], the learning rate, the cost function and the regularization approach to adopt in order to guarantee a good level of generalization and to prevent under and overfitting [13].

3 Generation of domain-specific heuristics in ASP

In this section, we describe the main challenges to face in order to automatically generate domain-specific heuristics, that are: finding a suitable representation of ASP instances in order to be used by deep learning algorithms, which usually operate on matrices (Section 3.1); generating a meaningful set of training instances (Section 3.2); creating a deep learning model to generate the heuristics (Section 3.3); embedding the heuristics into an ASP solver (Section 3.4).

3.1 Representation of ASP instances

In order to create a representation of the input program that is suitable for the deep learning model, we used a variant of the matrix representation used in NeuroCore [22]. In particular, a given program Π is represented as a $|\Pi| \times 2 \cdot |atoms(\Pi) \cup \{\bot\}|$ sparse matrix denoted with letter $\mathcal{G}$, where the rows are the rules of Π and the columns are all literals occurring in Π (including $\bot$ and $\sim\bot$). Then, a triple $(r,\ell,\text{-}1)$ represents that the literal ℓ occurs in the head of rule r; a triple $(r,\ell,1)$ represents that the literal ℓ occurs in the body of rule r; and a triple $(r,\ell,0)$ represents that the literal ℓ does not occur in r.

Example 2. Consider again program Π_G^C of Example 1. The first row of $\mathcal{G}$ is represented by the following triples: $(r_1,col(1,b),\text{-}1)$, $(r_1,\sim ncol(1,b),1)$, and $(r_1,\ell,0)$ for each other literal ℓ occurring in Π_G^C. Similarly, the last row of $\mathcal{G}$ is represented by the following triples: $(r_{16},\bot,\text{-}1)$, $(r_{16},\sim colored(2),1)$, and $(r_{16},\ell,0)$ for each other literal ℓ occurring in Π_G^C.

3.2 Generation of the training set

Deep learning algorithms operate on a set of labeled examples, referred to as *training set*. In our setting, the training set is composed by a set of tuples (Π, I), where Π represents an instance of the graph coloring problem, and I is a stable model of Π. The generation of a meaningful set of training instances is a challenging problem since deep learning algorithms require huge sets of examples to be successfully trained. Moreover, instances must be *easily* solvable for the ASP solver, since it is required to compute one stable model. Note that in principle one could also enumerate a fixed number of stable models, however in our preliminary experiments we observed this was not beneficial for the solver.

Our generation strategy is as follows. Given a graph $G = (N, L)$, a set C of colors, and a positive number k; we build a set of programs $\mathcal{P}$ representing the

training set, where each program in the set is a smaller portion of G. In particular, as first step, we block L and we randomly select $n\%$ of the nodes in N (with $n \in \{10, 20, 30, 40, 50\}$). For each value of n, we randomly generate k new graphs, whose corresponding programs are added to Π. Similarly, we block N and we randomly select $l\%$ of the links in L (with $l \in \{10, 20, 30, 40, 50\}$). As before, for each value of l, we randomly generate k new graphs, whose corresponding programs are added to Π. Finally, we randomly select $n\%$ of the nodes in N and $l\%$ of the links in L (with $n, l \in \{10, 20, 30, 40, 50\}$, and for each combination of n and l we randomly generate k new graphs. Hence, this strategy $35 \cdot k$ programs starting from a single input graph. In order to generate the training set, we considered all the sixty instances submitted to a recent ASP Competition [7] and we set the value of k to 100, for a total of 210 000 training instances.

3.3 Generation of the deep learning model

In this section we provide the details for training a DNN model to learn the heuristic characterizing a set of graph coloring instances expressed according to the ASP formalism. After the tuning phase, the resulting model is then queried to estimate the best initial configuration to be submitted to the WASP solver to enhance the CDCL branching routine and, consequently, the solving process.

The DNN model designed in this context takes inspiration from the NeuroCore architecture proposed by Selsam and Bjørner in [22]. Despite the different targets, NeuroCore model shows distinctive characteristics that can fit this paper's needs. Recalling section 3.1 and Example 2, we know that we have to deal with matrix representations. NeuroCore is able to manage problems of such matrix form thanks to its architecture, comprising three different MLPs:

$$\mathbf{R}_{\text{update}} : \mathbb{R}^{2d} \rightarrow \mathbb{R}^{d}, \mathbf{L}_{\text{update}} : \mathbb{R}^{3d} \rightarrow \mathbb{R}^{d}, \mathbf{V}_{\text{proj}} : \mathbb{R}^{2d} \rightarrow \mathbb{R}$$

where d is a fixed parameter and identifies the embedding associated with each atom and rule during model's iterations. In a nutshell, at each training step the model goes through T iterations of message passing, during which the rules' and literals' embeddings are continuously updated. The MLPs involved within these operations are $\mathbf{R}_{\text{update}}$ and $\mathbf{L}_{\text{update}}$, respectively. For the sake of clarity, the term *embedding* is usually exploited by practictioners to identify the vector exploited to translate a feature or a variable characterizing a data set in order to make the training process easier. In this context, we build a mono-dimensional vector with size d to represent each rule and each literal to be ingested by the latter MLPs. At each iteration, the output matrices of $\mathbf{L}_{\text{update}}$ and $\mathbf{R}_{\text{update}}$ are recursively combined and concatenated with the matrix $\mathcal{G}$, introduced in section 3.1. These continuous combinations are necessary to guarantee the robustness of the DNN model against rules' and literals' permutations, allowed in this context. Moreover, the embedding's exploitation is crucial to manage ASP programs with different number of atoms and rules, since, in this way, the number of neurons of the different MLP's layers involved can be fixed and the only varying dimension is the number of row of the input matrices. This does not represent a problem

since each program is managed singularly as a batch. Consequently, each batch represents the set of embeddings characterizing the literals or rules belonging to the same ASP program. After T iterations, $\mathbf{L}_{\text{update}}$'s output is horizontally split and the two even sub matrices, corresponding to each literal's and its negated correspondent's embeddings respectively, are vertically merged in order to build a matrix V, whose dimensions are $n_v \times 2d$ and where each row intuitively corresponds to a atom. At this point, V goes through the last MLP $\mathbf{V}_{\text{proj}}$ and $\hat{v}$ is finally obtained, which consists of a numerical score for each atom and it is finally passed to the softmax function to build a suitable probability distribution over the atoms. Concerning the embedding's size d, the number of iterations T and depth and width of the MLPs, the original values assigned by Selsam and Bjørner in [22] have been kept and are the following: 4 Iterations (T); 80 Embedding (d); 2 $\mathbf{C}_{\text{update}}$ layers; 2 $\mathbf{L}_{\text{update}}$ layers; 4 $\mathbf{V}_{\text{proj}}$ layers; 80 hidden layers neurons. The activation function exploited between each MLP's hidden layer is ReLU and the optimization algorithm adopted for training purpose is the ADAM one [17] with a constant learning rate of 10^{-4}. The considerations regarding $\mathbf{V}_{\text{proj}}$'s output layer and $\hat{v}$ interpretation need a further explanation. This paper's aim consists in determining a promising heuristic starting point for the solver's activity, which means that a value between 0 (*false*) and 1 (*true*) should be assigned to every literal of the instance under analysis. Moreover, it is fundamental to underline that literals corresponding to candidate colors for the same node are inevitably correlated and mutually exclusive. Due to this reason, the model should be able to assign a value of 1 exclusively to one of such literals in order to avoid contradictory scenarios. Consequently, $\mathbf{V}_{\text{proj}}$'s output activation function is kept linear and the softmax function is selectively applied to each group of atoms referring to the same node. Thereafter, the maximum value within each group is identified and assigned the value of 1, while 0s are assigned indistinctly to the remaining literals.

Furthermore, it is worth noting that, differently from Selsam and Bjørner's attempt in [22], the shape of the training instances, referring to the number of literals characterizing each of them, has not been fixed to a unique value. The data set considered in this context includes instances with varying sizes in the range comprised between 510 and 6032. It is feasible thanks to the NeuroCore architecture that is able to manage different shape instances through embedding representation. Nonetheless, it complicates the training process and poses important challenges to the generalization search.

3.4 Integration of the deep learning model in WASP

The integration of the domain-heuristic in WASP is based on the algorithm reported as Algorithm 1. In a nutshell, the algorithm takes as input a program Π and a set of parameters (namely, k_1, k_2, k_3, h_1, and h_2, such that $0 < k_i < 1$ $(i = [1..3])$, and h_1, $h_2 \in \mathbb{N}$, $h_1 > h_2$) and returns as output a set of heuristic assignments for the atoms of the form $col(\_, \_) \in atoms(\Pi)$. Such assignments will be used later on by WASP as initial activities of the atoms. In more details, it first invokes the deep learning model to obtain the predictions

Algorithm 1: Integration of the heuristic

Input : A program Π, parameters k_1, k_2, k_3, h_1, and h_2
Output: A set of pairs $\mathcal{H}$

```
1  H := ∅;
2  (Pr, Conf) := DeepLearning(Π);  // Returns predictions and confidences
3  N := {node | col(node, _) ∈ atoms(Π)};              // Nodes of the graph
4  for node ∈ N do
5      S := {col(node, _) | col(node, _) ∈ atoms(Π)};
6      (first, second) := ComputeAtomWithMaxConfidence(S, Conf);
7      if Pr(first) is true and Conf(first) ≥ k1 then
8          H := H ∪ {(first, h1)} ;
9          diff := Conf(first) − Conf(second);
10         sum := (∑_{p∈S} Conf(p)) − diff;
11         if diff ≤ k2 and Conf(second) > k3 · sum then
12             H := H ∪ {(second, h2)};
13 return H;
```

(Pr) and confidences ($Conf$) for the atoms of the form $col(\_, \_)$ (line 2), where a prediction can be either *true* (if the atom must be selected as positive) or *false* (if the atom must be selected as negative), and a confidence is a positive (decimal) number less than 1, where for a given node n the sum of the confidences of the atoms of the form $col(n, \_)$ is equal to 1. Then, the algorithm computes the set N of the nodes of the graph by processing the program Π (line 3; in particular, a node n is added to the set if an atom of the form $col(n, \_)$ occurs in Π). Later on, for each node n in N, the algorithm collects the set of atoms, say S, of the form $col(n, \_)$ (line 5). Then, it computes the two atoms in S associated to the highest confidences, say *first* the atom with the highest value, and *second* the other one (line 6). At this point, if the prediction of *first* is *true* and its confidence is greater than a given threshold (k_1), then the atom *first* is associated to the initialization h_1 (line 8). Moreover, an additional check is performed to provide a heuristic score also for the atom *second*. In particular, if the difference between the confidence associated to *first* and the one associated to *second* is less than or equal to a given threshold (k_2) and the confidence of *second* is greater than a threshold (k_3) times the sum of the confidences of all other atoms in S, then the atom *second* is associated to the initialization h_2 (line 12). Then, the default polarity of the MINISAT heuristic is set to positive for atoms in $\mathcal{H}$. Intuitively, for each node, the atom with the highest confidence (*first*) is used only if its confidence is greater than k_1. In this way, if the deep learning model is not sufficiently confident about the color to assign to the node then the heuristic is not applied to the node. Similarly, the atom with the second highest confidence (*second*) is used only if its confidence is similar to the one of *first* (i.e., their difference is smaller than k_2) and is greater than the confidence of all other atoms multiplied by k_3. Finally, the initialization of the activities of *first* and *second* to h_1 and h_2 permits the solver to select first the most promising

atoms, and then thanks to the decay of the MINISAT heuristic the activities are progressively reduced if the atoms are not used during the search.

4 Experiment

Hardware and software settings. With respect to the NeuroCore's-related model introduced in section 3.3, the 210 000 instances obtained after the data generation process of 3.2 have been randomly split into training, validation and test sets. More specifically, 60% of the instances have been picked to build the training set and the remaining 40% has been equally divided between validation and test sets. The training has been performed on a NVIDIA A100 Tensor Core GPU, dividing the training samples in batches of 128 instances and applying the backward propagation algorithm in relation to the binary cross entropy (BCE) loss measured on each batch. The stopping criterion adopted to this extent has been designed to monitor the BCE loss on the validation set and to interrupt the execution in case of consecutive lack of improvements.

Then, the performance of WASP without heuristics (referred to as WASP-DEFAULT) have been compared with the ones of WASP with the domain heuristics introduced as Algorithm 1. In particular, we experimented with different values of k_1, k_2, k_3, h_1, and h_2. In the following, we report the two sets obtaining the best performance overall, where $k_1 = 0.15$, $k_2 = 0.15$, $k_3 = 1.0$, $h_1 = 10$, and $h_2 = 5$ for the first strategy and $k_1 = 0.15$, $k_2 = 0.35$, $k_3 = 1.0$, $h_1 = 10$, and $h_2 = 5$ for the second one, that are referred to as WASP-STRAT1 and WASP-STRAT2, respectively. All the variants of WASP have been executed on all the sixty instances of the graph coloring problem submitted to a recent ASP Competition [7]. Note that the training set is built on random subgraphs of the input ones used in the ASP Competition, thus the experiment is not executed on instances used during the training of the deep learning model. Time and memory limit were limited to 1200 seconds and 8 GB, respectively.

Results deep learning. Table 1 reports DNN trained model's performances measured on the test set. Recalling model's generation of section 3.2, outputs can be interpreted as the confidences of the model in stating that the value of 1 can be assigned to a specific literal. The model has been evaluated in terms of TOP N accuracy, where $N \in \{1, 2, 3\}$, and it corresponds to the ratio between the predicted and expected 1s among the first N most confident estimations. Moreover, the percentage of predicted 1s for increasing confidence $C \in \{20, 30, 40, 50\}$ % is measured. As expected, the percentage accuracy increases in agreement with N and C, with approximately 80% for $N = 3$ and 70% for $C \geq 50\%$. The same confidence levels is not guaranteed for all the instances under analysis, as it is underlined by the performances measured for decreasing values of C and N. Nonetheless, it is fundamental to keep in mind the complexity of the proposed target, continuously managing graphs with different shapes.

Result on ASP instances. Table 2 reports the results of the comparison of the different approaches implemented in WASP, where for each heuristic, we show

Accuracy (%)						
TOP N			Confidence $\geq C\%$			
$N=1$	$N=2$	$N=3$	$C=20$	$C=30$	$C=40$	$C=50$
39.59	63.62	79.55	39.59	51.66	63.08	70.74

Table 1. DNN' EE on the test in terms of TOP N and Confidence Accuracies.

	Coherent			Incoherent		
Heuristic	#	solved	PAR10	#	solved	PAR10
WASP-DEFAULT	14	12	1744.90	24	**24**	**449.43**
WASP-STRAT1	14	13	914.97	24	20	2393.44
WASP-STRAT2	14	**14**	**82.94**	24	19	2837.65

Table 2. Comparison of the different heuristics on ASP competition instances.

	130 nodes			135 nodes			140 nodes		
Heuristic	#	solved	PAR10	#	solved	PAR10	#	solved	PAR10
WASP-DEFAULT	60	**60**	**15.24**	89	77	1707.73	85	76	1322.17
WASP-STRAT1	60	**60**	32.95	89	**78**	**1609.21**	85	76	1338.02
WASP-STRAT2	60	**60**	31.80	89	**78**	1620.13	85	**78**	**1066.18**

Table 3. Comparison of the different heuristics on generated instances.

the number of solved instances, and the PAR10. We recall that the PAR10 is the average solving time where unsolved instances are counted as $10 \cdot$ timeout. PAR10 is a metric commonly used in machine learning and SAT communities, as it allows to consider both coverage and solved time. As a first observation, we mention that the call to the deep learning model requires on average less than one second, thus it has no negative impact on the performance of the domain-specific heuristics. Then, we observe that both WASP-STRAT1 and WASP-STRAT2 are faster than WASP-DEFAULT on coherent instances, solving 1 and 2 more instances, respectively. Additionally, WASP-STRAT2 has a PAR10 equals to 82.94 and it is approximately 21 times lower than the one of WASP-DEFAULT. The same result cannot be obtained for incoherent instances, where WASP-DEFAULT solves 4 and 5 instances more than WASP-STRAT1 and WASP-STRAT2, and also with a much lower PAR10. This result is expected since only coherent instances were used during the training and also since the heuristic is oriented towards finding a stable model. As an additional experiment, we generated, starting from the set of known incoherent instances, another set of coherent instances by randomly removing a certain number of links from the input instance. Table 3 reports the results of such an experiment, where we classified instances into three sets according to the number of nodes, i.e., 130 nodes, 135 nodes, and 140 nodes. Interestingly, domain-specific heuristics are not effective on instances with 130 nodes, which are solved quite fast by the default version of WASP. However, on instances with 135 and 140 nodes the domain-specific WASP-STRAT2 outperforms

WASP-DEFAULT solving three more instances overall and being faster in terms of PAR10. Finally, concerning the usage of memory, we observe that all the tested approaches never exceed memory limits.

5 Related Work

Several ways of combining domain heuristics and ASP are proposed in the literature. In [3], a technique which allows learning of domain-specific heuristics in DPLL-based solvers is presented. The basic idea is to analyze off-line the behavior of the solver on representative instances from the domain to learn and use a heuristic in later runs. A declarative approach to definition of domain specific heuristics in ASP is presented in [11]. The techniques presented in this paper might be also applied in combination with such a framework by properly setting the *_heuristic* predicate. Andres et al. in [2] and Dodaro et al. in [8] proposed domain-specific heuristics for tackling hard problems. However, their approach was based on the implementation of the heuristic made by a domain expert. ML-solutions have been also adopted to predict the best solver for a given instance [6, 14, 19]. We are not aware of any attempt to experiment with automatic learning of domain heuristics in modern CDCL-based solvers.

In the context of SAT, our work is related to the one of Selsam and Bjørner [22] and their system NeuroCore. In particular, our deep learning model takes inspiration of their proposal. Nonetheless, in our model we do not fix the shape of the training instances to a unique value. Another important difference is that our training set contains coherent instances only, whereas the one used by NeuroCore is instead based on (minimal) unsatisfiable cores. This difference was motivated by the fact that the computation of (minimal) unsatisfiable cores is not currently supported by state-of-the-art ASP solvers. The integration of such techniques can be also beneficial in the context of domain-specific heuristics. Moreover, the deep learning model introduced by NeuroCore is periodically queried during the search to re-configure the branching heuristic. However, our preliminary experiments (not included in the paper for space reason) show that considering learned constraints deteriorate the performance of the solver, since multiple calls to the deep learning model on larger and larger programs were counterproductive. Wu in [25] pointed out the lack of efficiency of CDCL algorithm in solving formulae of even moderate sizes, e.g. 300 to 500 variables involved, and proposes to take advantage of ML techniques to train a model able to wisely assign initial values to branching variables in order to prevent possible conflicts and to find a solution in relatively short time. After the experimental phase, it was observed a consistent decrease in the number of conflicts. However, the computational time required to perform the preprocessing phase was non-negligible compared to the timing necessary to run the enhanced version of the solver taken as a benchmark. Moreover, Liang et al. in [18] proposed a ML-based approach to predict the so called *Literal Block Distance* (LBD), defined as the number of different decision levels of the variables in the clause. They choose to exploit an Adam SGD algorithm that autonomously triggers a restart if the next LBD exceeds the

linear sample mean for 3.08 standard deviations (i.e. the 99.9^{th} percentile). The experiments show that the proposed approach performs coherently with state-of-the-art methods. Xu et al. in [26] proposed a ML-based strategy to evaluate 3-SAT instances on the phase transition. In particular, they trained a model on a 3-SAT dataset comprising instances with varying number of variables in the range $100 - 600$. They initially opt for a random forest algorithm with the aim of discriminating between SAT or UNSAT instances basing on 61 cheap-to-compute features. Then, they progressively simplify their model and reduce the number of features considered still achieving reasonable performances.

6 Conclusion

In this paper, we presented a strategy based on deep learning to automatically generate domain-specific heuristics. In particular, we focus on one single benchmark, i.e. the graph coloring problem. This choice was motivated by the fact that *(i)* the encoding does not include advanced features such as aggregates, choice rules, and weak constraints; *(ii)* the problem allows to control the hardness of the instance by either reducing the number of nodes and/or the number of links. Moreover, the training set used to automatically generate the heuristics contains coherent instances only and, as expected, this lead to poor performance on incoherent ones. As future work, alternative strategies consist of exploiting minimal unsatisfiable cores, or automatically tuning parameters used by Algorithm 1.

Acknowledgments

This work is partially supported by MUR under PRIN project "exPlaInable kNowledge-aware PrOcess INTelligence" (PINPOINT), CUP H23C22000280006 and by MISE under project MAP4ID "Multipurpose Analytics Platform 4 Industrial Data", N. F/190138/01-03/X44.

References

1. Alviano, M., Dodaro, C., Leone, N., Ricca, F.: Advances in WASP. In: Proc. of LPNMR. LNCS, vol. 9345, pp. 40–54. Springer (2015). https://doi.org/10.1007/978-3-319-23264-5_5
2. Andres, B., Biewer, A., Romero, J., Haubelt, C., Schaub, T.: Improving Coordinated SMT-Based System Synthesis by Utilizing Domain-Specific Heuristics. In: Proc. of LPNMR. LNCS, vol. 9345, pp. 55–68. Springer (2015). https://doi.org/10.1007/978-3-319-23264-5_6
3. Balduccini, M.: Learning and using domain-specific heuristics in ASP solvers. AI Commun. **24**(2), 147–164 (2011). https://doi.org/10.3233/AIC-2011-0493
4. Bengio, Y., LeCun, Y., Hinton, G.E.: Deep learning for AI. Commun. ACM **64**(7), 58–65 (2021). https://doi.org/10.1145/3448250
5. Brewka, G., Eiter, T., Truszczynski, M.: Answer set programming at a glance. Commun. ACM **54**(12), 92–103 (2011). https://doi.org/10.1145/2043174.2043195

6. Calimeri, F., Dodaro, C., Fuscà, D., Perri, S., Zangari, J.: Efficiently coupling the I-DLV grounder with ASP solvers. TPLP **20**(2), 205–224 (2020). https://doi.org/10.1017/S1471068418000546
7. Calimeri, F., Gebser, M., Maratea, M., Ricca, F.: Design and results of the fifth answer set programming competition. Artif. Intell. **231**, 151–181 (2016). https://doi.org/10.1016/j.artint.2015.09.008
8. Dodaro, C., Gasteiger, P., Leone, N., Musitsch, B., Ricca, F., Schekotihin, K.: Combining answer set programming and domain heuristics for solving hard industrial problems (application paper). TPLP **16**(5-6), 653–669 (2016). https://doi.org/10.1017/S1471068416000284
9. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: Proc. of SAT. LNCS, vol. 2919, pp. 502–518. Springer (2003). https://doi.org/10.1007/978-3-540-24605-3_37
10. Erdem, E., Gelfond, M., Leone, N.: Applications of answer set programming. AI Mag. **37**(3), 53–68 (2016). https://doi.org/10.1609/aimag.v37i3.2678
11. Gebser, M., Kaufmann, B., Romero, J., Otero, R., Schaub, T., Wanko, P.: Domain-specific heuristics in answer set programming. In: Proc. of AAAI. AAAI Press (2013)
12. Goodfellow, I., Bengio, Y., Courville, A.: Deep learning. MIT press (2016)
13. Hawkins, D.M.: The problem of overfitting. Journal of chemical information and computer sciences **44**(1), 1–12 (2004)
14. Hoos, H.H., Lindauer, M., Schaub, T.: claspfolio 2: Advances in algorithm selection for answer set programming. TPLP **14**(4-5), 569–585 (2014)
15. Hornik, K., Stinchcombe, M.B., White, H.: Multilayer feedforward networks are universal approximators. Neural Networks **2**(5), 359–366 (1989)
16. Kaufmann, B., Leone, N., Perri, S., Schaub, T.: Grounding and solving in answer set programming. AI Mag. **37**(3), 25–32 (2016). https://doi.org/10.1609/aimag.v37i3.2672
17. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
18. Liang, J.H., Oh, C., Mathew, M., Thomas, C., Li, C., Ganesh, V.: Machine learning-based restart policy for CDCL SAT solvers. In: Proc. of SAT. LNCS, vol. 10929, pp. 94–110. Springer (2018)
19. Maratea, M., Pulina, L., Ricca, F.: A multi-engine approach to answer-set programming. TPLP **14**(6), 841–868 (2014). https://doi.org/10.1017/S1471068413000094
20. Oneto, L.: Model selection and error estimation in a nutshell. Springer Cham (2020)
21. Rumelhart, D., Hinton, G., Williams, R.: Learning representations by back-propagating errors. Nature **323**(6088), 533–536 (1986)
22. Selsam, D., Bjørner, N.S.: Guiding high-performance SAT solvers with unsat-core predictions. In: Proc. of SAT. LNCS, vol. 11628, pp. 336–353. Springer (2019). https://doi.org/10.1007/978-3-030-24258-9_24
23. Shalev-Shwartz, S., Ben-David, S.: Understanding machine learning: From theory to algorithms. Cambridge university press (2014)
24. Taupe, R., Friedrich, G., Schekotihin, K., Weinzierl, A.: Solving configuration problems with ASP and declarative domain specific heuristics. In: Proc. of CWS/ConfWS. vol. 2945, pp. 13–20. CEUR-WS.org (2021)
25. Wu, H.: Improving sat-solving with machine learning. In: Proc. of SIGCSE. pp. 787–788. ACM (2017)
26. Xu, L., Hoos, H.H., Leyton-Brown, K.: Predicting satisfiability at the phase transition. In: Proc. of AAAI. AAAI Press (2012)