Answer set computation of negative two-literal programs based on graph neural networks: Preliminary results

Antonio Ielo[1] and Francesco Ricca[1]

[1]University of Calabria, Italy
ai6chr@gmail.com - ricca@mat.unical.it

Abstract. *Graph neural networks* architectures have been successfully applied to combinatorial optimization problems such as SAT and MAX-CSP. In this work in progress we attempt to adapt similar techniques to Answer Set Programming, in particular to *negative two-literal programs* as they represent a class of programs of practical and theoretical interest whose answer sets have a full graph-theoretical characterization.

Keywords: Answer Set Programming · Graph Neural Networks

1 Introduction

In the recent years deep learning techniques have been successfully applied to the combinatorial optimization domain [3]. This has been possible mainly thanks to the *graph neural network* architectures [12], whose inductive biases (mainly *permutation invariance*) are well suited to exploit the graph encodings of problem instances that frequently and naturally occour in this field. Graph neural networks were recently applied to SAT [14, 13] and CSP [15] providing interesting results to the approximate solution of these problems. As far as we know, such approaches have not been attempted on Answer Set Programming (ASP) [2]. ASP is an established logic-based programming paradigm which has been successfully applied for solving complex problems arising in Artificial Intelligence.

In this paper we approach the application of graph neural networks to ASP. We focus on *negative two-literal* ASP programs [11], a relevant subclass of ASP programs that is able to model all problems in NP. Negative two-literal programs have a natural correspondence with graphs, and thus fit well the graph neural network architecture. In particular, we build on the approach of RUNCSP [15] and adapt the graph neural network architecture to handle ASP programs. Preliminary experimental results show strength and weaknesses of the approach which paves the way for future improvements and applications.

2 Message Passing Graph Neural Networks

In the recent years there has been a surge of interest in machine learning techniques able to process graph data [4], hence an abundance of graph neural network architectures and frameworks have been defined.

We briefly recall the *message passing* framework [6, 7], as it is the most relevant for our work. We refer the reader to [1] for an up-to-date reference on graph neural network architectures and their applications.

Let G be a directed graph. We denote the set of its vertices and the set of its edges as $V(G)$ and $E(G)$ respectively. Let $\{\{A\}\}$ denote the set of multisets over the elements of a set A. Let $N(v) \subseteq V(G)$ denote the neighbourhood of the node $v \in V(G)$, that is the set of nodes $w \in V(G)$ such that $(v, w) \in E(G)$. Let

$$
\begin{aligned}
&M : \mathbb{R}^k \times \mathbb{R}^k \longmapsto \mathbb{R}^k \\
&U : \mathbb{R}^k \times \mathbb{R}^k \longmapsto \mathbb{R}^k \\
&C : \{\{\mathbb{R}^k\}\} \longmapsto \mathbb{R}^k \\
&R : \mathbb{R}^k \longmapsto \mathbb{R}^d
\end{aligned}
$$

be functions, and let $h_v \in \mathbb{R}^k$ be a real-valued vector associated to the vertex $v \in V(G)$. A *message passing round* consists in the following computation:

$$
h_v^{i+1} = U(h_v^i, C(\{\{M(h_v^i, h_{v'}^i) : v' \in N(v)\}\})) \tag{1}
$$

Here h_v^i represents a *state* associated to each node in the i-th iteration of message passing. Neighbouring nodes exchange messages, generated by the *message function* M. Each node aggregates incoming messages through the *aggregating function* C and updates its own state through the *update function* U. Finally, the *readout function* R can be used to project each node state from $\mathbb{R}^k$ to $\mathbb{R}^d$ and represents the output of the neural network on a given node. This computation is iterated up to T times. If the above mentioned functions are differentiable (as those implemented through feedforward neural networks or recurrent neural networks) then the whole system - which we will refer to as *message passing neural network* - can be trained through standard deep learning optimization techniques such as backpropagation [7].

3 Answer Set Programming

We now recall some basic notions regarding ASP.

Syntax. Let C be a set of *constants*, a *predicate* p is a subset of C^k and k is known as the *arity* of P. A *ground atom* is an expression $p(c_1, ..., c_k)$ where p is a predicate of arity k and $(c_1, ..., c_k) \in p$. If a is an atom, $\neg a$ is its *opposite* atom and we say $\neg a$ is the *negation* of a.

A *rule* r is an expression:

$$a_0 \leftarrow a_1, ..., a_k, \neg a_{k+1}, ..., \neg a_n \tag{2}$$

where a_i are ground atoms. a_0 is the *head* of the rule and we denote it by $H(r)$, while $a_1, ..., a_p, \neg a_{p+1}, ..., \neg a_n$ is the *body* of the rule which we denote by $B(r)$. We can partition $B(r)$ into two sets $B^+(r) = \{a_1, ..., a_k\}$ and $B^-(r) = \{a_{k+1}, ..., a_n\}$, respectively the set of *positive* atoms and *negative* atoms in the rule's body.

A rule is *positive* if B^-r is empty, and *negative* if B^+r is empty.

A *(normal) logic program program* is a set of rules. A program is positive (negative) if all its rules are positive (negative).

Semantics. Let $X \subseteq A$ be a set of atoms. We say X *satisfies* a rule r if $B^+(r) \subseteq X$, $B^-(r) \cap X$ is empty implies that $H(r) \in X$. If X satisfies all the rules of a program P then X is a *model* for P. A model is said to be *minimal* if there does not exist a model X' such that $X' \subset X$. Positive logic programs admit a unique minimal model.

Let $X \subseteq A$. The *reduct* of a program P with respect to X is a program P^X that contains a rule $H(r) \leftarrow B^+(r)$ whenever $B^-(r) \cap X$ is empty. An *answer set* is a set X of atoms such that X is a minimal model of P^X. A program is *consistent* if it admits at least one answer set, *incoherent* otherwise.

Supportedness. Let P be a logic program over a set of atoms A. Given an answer set S of P we say that an atom $a \in S$ is *supported* in S if there exists a rule r in P such that $H(r) = a$. It is in the folklore that all atoms in an answers set are supported.

It is known that negative two-literal programs [11, 16] are expressive enough to encode combinatorial problems over graphs of practical interest [8]. Indeed, deciding whether a negative two-literal is consistent is known to be a NP-complete problem [10]. Moreover, as it will be clearer in the following answer sets of negative two literal programs have a natural counterpart correspondence with graphs. For this reason we concentrate our attention on this sub-class of ASP programs.

Negative two-literal programs. *[11, 16]* A *negative two-literal program* (N2LP) is a negative logic program whose rules are all of the form $a \leftarrow \neg b$.

Properties of N2LP. We now recall some relevant properties of N2LP. [16].

Proposition 1. *Each normal logic program is equivalent to a negative two-literal program under answer set semantics.*

Proposition 2. *Let P be a negative two-literal program on (a set of atoms) A. Then S is an answer set of P if and only if the two conditions are satisfied:*

- *If $b_1, b_2 \in A \setminus S$ then $b_1 \leftarrow \neg b_2$ is not a rule in P.*
- *If $a \in S$ there exists $b \in A \setminus S$ such that $a \leftarrow \neg b$ is a rule of P.*

Let P be a negative two-literal program. We can associate to each negative two-literal program a graph $G(P)$, where nodes are the atoms of P and there exists an edge (α, β) whenever P contains a rule of the form $\beta \leftarrow \neg\alpha$.

Recall that $W \subseteq V(G)$ is an *independent set* of a directed graph G if $E(G)$ does not contain an edge (w_1, w_2) with $w_1, w_2 \in W$. If W is an independent set of G and $W \cup \{v\}$ for all $v \in V(G) \setminus W$ is not an independent set of G then we say W is a *maximal independent set*. If for all edges $(a, b) \in E(G)$ we have that $a \in W$ or $b \in W$ we say that W *dominates* G. A *kernel* is a maximal independent set of G that dominates G. Thus, one can restate the latter proposition as follows:

Proposition 3. *Let P be a negative two-literal program, $G(P)$ its associated graph. Then S is an answer set of P if and only if $V(G(P)) \setminus S$ is a kernel of $G(P)$.*

Thus, one can tackle the computation of answer sets of N2LP by devising an approach able to compute kernels of a graph. This is the key observation underlying our approach in which we extend a graph neural network solving the independent set to identify kernels.

Since the answer sets of negative two-literal programs can be characterized in a graph-theoretical way, they represent a natural starting point in exploring feasibility of graph neural networks based techniques. Indeed, no encoding or feature extraction are necessary: directed graphs are negative two-literal programs, and we will talk about graphs/programs and nodes/atoms as it is more convenient, with no ambiguity.

4 A Graph Neural Network for N2LP

The architecture of the network extends RUNCSP [15]. RUNCSP is a message passing neural network architecture that unsupervisedly finds approximate solutions to MAX-CSP problems, by attempting to minimize a loss function that penalizes solutions which violate constraints. In the simplified context of our problem we need to describe the functions M, U, C, R (see Section 2).

The message function (M) is a linear function of the nodes' hidden states. The aggregating function (C) is the average function. The update function (U) is a LSTM cell [17]. The readout function (R) is a linear function followed by a sigmoid activation function to obtain outputs in $[0, 1]$ for node classification. An output close to 1 for a node v means that the corresponding atom is part of the candidate answer set. The loss function can be defined as:

$$\mathcal{L}(P, t) = \frac{1}{|R|} \sum_{(a,b) \in R} -\log(1 - (1 - \psi^t(a)) \cdot (1 - \psi^t(b)) \tag{3}$$

Where R is the set of rules $b \leftarrow \neg a$ of a logic program P and ψ^t represents the neural network's readout function on the t message passing round. The readout function output on an atom embedding is interpreted as the probability that the atom belongs to the *candidate answer set* of P. In the original architecture,

the loss ("constraint loss", as referred to in the original article [15]) of all the T message passing iterations is combined with a *discount factor* in the following way:

$$\mathcal{L}_{tot}(P) = \sum_{t}^{T} \lambda^{T-t} \mathcal{L}(P, t) \tag{4}$$

Intuitively, this penalizes the network for violating constraints in the later message passing rounds, while being less relevant in the earlier rounds. The term $(1-\psi(a)) \cdot (1-\psi(b))$ amounts to the probability that the rule $b \leftarrow \neg a$ is violated in the candidate answer set.

The main difference between RUNCSP and our neural network lies in the loss function: we are not looking for an independent set, as in the RUNCSP Maximal Independent Set experiment [15], but for a kernel of the graph. What we are missing is enforcing supportedness of atoms (in place of the "size" of the candidate answer, like in the original approach). In some preliminary experiments we observed that adding extra terms to the loss function makes the results of training unstable, hence we proceed as follows. Let $\psi(P) \in [0,1]^n$ be the output of our neural networks on a logic program P, it represents the probability that the i-th atom belongs to the candidate answer set. We denote such probability p_i. Thus, we are interested in computing a real vector $S \in [0,1]^n$ where s_i represents the probability that the i-th atom is supported by an atom outside the answer set.

We can express such computation as a message passing operation:

$$s_i = 1 - \prod_{k \in N(v_i)} p_k \tag{5}$$

The product term expresses the probability that *all* the neighbours of v_i are inside the answer set - that is, unable to support v_i. Hence its complement represents the probability that at least one of its neighbours is outside the answer set.

Such computation is based solely on $\psi(P)$ and does not have any learning involved. $s_i \cdot p_i$ represents the probability that v_i belongs to the answer set and at least one of its neighbours does not belong to the answer set - that is, it is supported. We train unsupervisedly on constraint loss, while we enforce supportedness by modulating $\psi(P)$ via S, using their coordinate-wise product for our predicitons. Intuitively, we bias probability of choosing atoms by the likelihood of them being supported.

In practice the notion of atom supportedness in an answer set does not take in account by "how many" rules an atom is supported, hence the above computation is biased towards atoms with an high number of neighbours. For example, suppose v is an atom with 3 neighbours, and each one of them has a probability $p = 0.5$ to be outside the answer set. Then by the above equation, $S(v) = 1.0 - 0.5^3 = 0.875$. In order to solve this issue, we compute S as follows:

$$S_i = \max_{k \in N(i)} (1 - p_k)$$

That is, we compute the likelihood of v_i being supported as the maximum probability of one of its neighbours being ouside of the answer set. In the above numerical example, the probability of v being supported would be only of 0.5 rather than 0.875.

5 Experiment

5.1 Hyperparameters and training

The network is trained on a random stream of Erdős-Rényi [5] graphs with 20 to 50 nodes and average degree between 2.0 and 5.0. Number of nodes and average degree are independently, uniformly sampled for each graph required during the network's training.

All the neural network's hyperparameters (batch size, inference message passing rounds, loss discount factor and training message passing rounds) have been kept the same as described in the original RUNCSP article's reproducibility section [15], with the exception of the learning rate and node embedding size which have been set respectively to $2 \cdot 10^{-5}$ and 256.

The neural network is trained with stochastic weight averaging and an early stop policy based on the loss on a validation set composed of Erdős-Rényi graphs of 150 nodes with average degree that spans from 2.0 to 12.0. The reported experiments' results were performed with the model that best performed on the fixed validation set among all the epochs.

5.2 Evaluation

Our neural network is a binary node classifier that given a negative two-literal logic program P maps its atoms to their probability of being contained in a *candidate answer set* S'. We can obtain S' by choosing nodes above a treshold t, that we fix to 0.5 in our experiments. If P is a consistent program we can compare S' to its answer set S and compute binary classification metrics such as accuracy and F1 score in order to evaluate "how far", in terms of misclassified atoms, our candidate answer set S' is from the real answer set S. P might have multiple answer sets, and we compare our solution with the closest answer set (w.r.t. the minimal cardinality of the symmetric set difference) we obtained running the ASP system clingo. Hence, to evaluate our neural network on a consistent program P:

1. Build the graph $G(P)$.
2. Evaluate the neural network on $G(P)$ to obtain S'
3. Compute closest answer set S to S'
4. Compute accuracy and F1 score for predictions S' with respect to S

We evaluate our network on random graphs of 150 nodes and average degree spanning from 2.0 to 9.0. Note that in this interval one should expect to find hard N2LP programs [11]. Each measure in Table 1 is obtained by averaging binary classification metrics on a sample of 1000 random programs of the given degree by performing the steps previously described. We report our results in terms of F1 score and accuracy.

The *Neural* columns report the results of the experiment using our neural network, while the *Random* column reports the results of the experiment inserting each atom in the answer set with a 0.5 probability.

Our approach outperforms balanced random choices and is very accurate on low-density graphs, however the performance degrades as the underlying graphs get more dense, that is when the logic programs have more rules.

6 Conclusion and future work

In this paper we proposed the first graph neural network approach for ASP programs. The approach builds on the ideas underlying RUNCSP architecture and demonstrates a promising behaviour on random programs. Indeed, it is basically very accurate on low-density graphs, and thus might already be considered an useful tool in this setting. However, the network needs improvements to be used in more general setting. In the future we plan to investigate ways for improving the performance of the network in "dense" graphs and we will study how to apply it for improving ASP solvers performance (e.g., using our approach as an heuristics).

Table 1. Results of the experiment on Erdős-Rényi graphs of 150 nodes.

Avg. Degree	F1 Score		Accuracy	
	Neural	Random	Neural	Random
2.0	0.996	0.372	0.996	0.473
2.5	0.982	0.379	0.978	0.452
3.0	0.974	0.381	0.967	0.432
3.5	0.949	0.386	0.932	0.419
4.0	0.927	0.392	0.899	0.412
4.5	0.906	0.393	0.868	0.401
5.0	0.890	0.396	0.842	0.394
5.5	0.862	0.395	0.811	0.386
6.0	0.837	0.397	0.787	0.379
6.5	0.795	0.396	0.748	0.371
7.0	0.749	0.401	0.709	0.370
7.5	0.667	0.402	0.651	0.365
8.0	0.622	0.404	0.615	0.363
8.5	0.549	0.404	0.564	0.359
9.0	0.498	0.405	0.527	0.356

Acknowledgments. We'd like to thank the PyTorch Geometric Tutorial community [9] - all of this would have been less, less pleasant than it was. A special thank to prof. Davide Bacciu for the discussions on RUNCSP and for the hints in devising graph neural networks.

References

1. Bacciu, D., Errica, F., Micheli, A., Podda, M.: A gentle introduction to deep learning for graphs. Neural Networks (2020)
2. Brewka, G., Eiter, T., Truszczynski, M.: Answer set programming at a glance. Commun. ACM **54**(12), 92–103 (2011)
3. Cappart, Q., Chételat, D., Khalil, E., Lodi, A., Morris, C., Veličković, P.: Combinatorial optimization and reasoning with graph neural networks. arXiv preprint arXiv:2102.09544 (2021)
4. Chami, I., Abu-El-Haija, S., Perozzi, B., Ré, C., Murphy, K.: Machine learning on graphs: A model and comprehensive taxonomy. arXiv preprint arXiv:2005.03675 (2020)
5. Erdos, P., Rényi, A., et al.: On the evolution of random graphs. Publ. Math. Inst. Hung. Acad. Sci **5**(1), 17–60 (1960)
6. Gilmer, J., Schoenholz, S.S., Riley, P.F., Vinyals, O., Dahl, G.E.: Neural message passing for quantum chemistry. In: International conference on machine learning. pp. 1263–1272. PMLR (2017)
7. Hamilton, W.L.: Graph representation learning. Synthesis Lectures on Artifical Intelligence and Machine Learning **14**(3), 1–159 (2020)
8. Huang, G.S., Jia, X., Liau, C.J., You, J.H.: Two-literal logic programs and satisfiability representation of stable models: A comparison. In: Conference of the Canadian Society for Computational Studies of Intelligence. pp. 119–131. Springer (2002)
9. Longa, A., Santin, G., Pellegrini, G.: Pytorch geometric tutorial (2020)
10. Marek, W., Truszczyński, M.: Autoepistemic logic. Journal of the ACM (JACM) **38**(3), 587–618 (1991)
11. Namasivayam, G., Truszczyński, M.: Simple random logic programs. In: International Conference on Logic Programming and Nonmonotonic Reasoning. pp. 223–235. Springer (2009)
12. Scarselli, F., Gori, M., Tsoi, A.C., Hagenbuchner, M., Monfardini, G.: The graph neural network model. IEEE Trans. Neural Networks **20**(1), 61–80 (2009). https://doi.org/10.1109/TNN.2008.2005605, https://doi.org/10.1109/TNN.2008.2005605
13. Selsam, D., Bjørner, N.: Neurocore: Guiding high-performance sat solvers with unsat-core predictions (2019)
14. Selsam, D., Lamm, M., Bünz, B., Liang, P., de Moura, L., Dill, D.L.: Learning a SAT solver from single-bit supervision. In: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019. OpenReview.net (2019), https://openreview.net/forum?id=HJMC_iA5tm
15. Toenshoff, J., Ritzert, M., Wolf, H., Grohe, M.: Run-csp: unsupervised learning of message passing networks for binary constraint satisfaction problems (2019)
16. Wang, K., Wen, L., Mu, K.: Random logic programs: linear model. Theory and Practice of Logic Programming **15**(6), 818–853 (2015)
17. Yu, Y., Si, X., Hu, C., Zhang, J.: A Review of Recurrent Neural Networks: LSTM Cells and Network Architectures. Neural Computation **31**(7), 1235–1270 (07 2019)