Graph Neural Networks for Scheduling of SMT Solvers

1st Jan Hůla
University of Ostrava
Ostrava, Czechia

2nd David Mojžíšek
University of Ostrava
Ostrava, Czechia

3rd Mikoláš Janota
Czech Technical University in Prague
Prague, Czechia

***Abstract*—This paper develops an approach to the scheduling of solvers in the domain of Satisfiability Modulo Theories (SMT) using a Graph Neural Network (GNN). In contrast to related methods, GNNs do not require manual feature design as they enable discovering relevant features in the raw data. We train them to predict the effectivity of individual solvers on a given problem. Rather than choosing only one solver with the best prediction, we schedule the solvers by ordering them according to the predicted runtime and dividing the overall runtime into all solvers uniformly. We compare our approach to several baselines. In the selected benchmarks, we show a substantial improvement over these baselines in terms of the number of solved problems and overall solving time.**

I. Introduction

In recent years, there has been a lot of interest in the use of machine learning (ML) for problems dealing with combinatorial search. Such problems are known to be NP-complete or harder, and typically there is no single best algorithm to deal with them. Instead, various algorithms behave differently on different problem distributions. For a given problem distribution, we can settle with a single best-on-average algorithm or try to choose the best one from a portfolio on a per instance basis.

The previous statement also holds in the domain of Satisfiability Modulo Theories (SMT). Different heuristics used in individual solvers or their configurations may produce different per-instance behaviour in terms of runtime or even in the ability to find a successful solution. For that reason, there has been an interest to design portfolios of solvers together with predictors which select a solver. This per-instance behaviour is hard to understand for a human, but we may try to predict it using ML methods. These predictors are conditioned on various features of the SMT formula.

In the past, most of the approaches have been based on manually designed features which reflected the intuition of a domain expert. These features can vary from very complex, such as the ones given by linear programming relaxation of the formula, or various runtime statistics of a particular solver [1], to very simple, such as counts of occurrence of various symbols within the formula [2].

Instead of designing better features, we follow the recent trend of learning features from raw data together with the predictions in an end-to-end fashion. In our case, we use a Graph Neural Network (GNN) [3] that takes as input an SMT formula graph representation and predicts the efficiency of available solvers. Using the predictions from the GNN, we construct a schedule which runs individual solvers until a timeout or solution is reached. We compare our approach to several baselines and show an substantial improvement in terms of the PAR-2 score.

We also demonstrate a significant drawback of choosing only the solver with the best prediction, which was done in a previous work [2], by showing that it is outperformed by a random schedule which splits the time across the individual solvers and runs them in an random order.

To summarize, the paper has the following main contributions.

- It applies GNN to predict the performance of SMT solvers on a given instance. To the best of our knowledge, this is the first application of GNN in the context of SMT.
- The proposed approach schedules the solvers rather than just picking the best one, which further improves the robustness of the approach.

II. Satisfiability Modulo Theories

Solvers for *Satisfiability Modulo Theories (SMT)* are the driving force behind software verification, software testing, or software synthesis, among others [4]–[7]. These applications often require repeated queries to an SMT solver. This means that quick response times of the solver are paramount.

An SMT solver receives as input a formula and responds if it is satisfiable or not. Since the problem is generally undecidable, solvers often timeout or give up.

The language and the semantics of the given formula depends on the theories being used (such as theory of non-linear integer arithmetic) and are standardized in the *SMT-LIB standard* [8].

Various SMT solvers support various theories and their combinations. Furthermore, not all solvers support all the features of the SMT language. This alone makes the choice of the right solver for a given instance a nontrivial task.

III. Problem Statement

Let $S = \{s_1, \ldots, s_l\}$ be a set of l SMT solvers which we have at our disposal. Our goal is to produce an effective algorithm which on per-instance basis creates a schedule from the set S.

More formally, we want to obtain a function f_θ (parametrized by learnable parameters θ) which takes a representation of an SMT formula q and the set S as an input

and outputs an ordered tuple $f_\theta(q, S) = ((i_1, t_1), \ldots, (i_n, t_n))$ where i_j's are indices of selected solvers and t_j's are times assigned to these solvers, such that $\sum_j t_j = t_{max}$, where t_{max} is the maximum time we are willing to spend on a problem. In other words, for a given formula q, $f_\theta(q, S)$ represents a schedule of chosen solvers for this formula.

Given a formula q and its schedule $f_\theta(q, S)$, we measure how long it takes to solve the formula using this schedule. We denote this measurement by $M(q, f_\theta(q, S))$ and set it to a constant number t_{pen} (with $t_{pen} > t_{max}$) if the formula was not solved under the time limit t_{max}.

We assume that the problems/formulas we want to solve come from an unknown distribution P and that we have a finite set of independent and identically distributed samples $Q = \{q_1, \ldots, q_m\}$ where $q_i \sim P$. Our task may be posed as the following optimization problem:

$$\theta^* = \arg\min_\theta \int M(q, f_\theta(q, S))\, dP(q)$$

Because the distribution P is unknown, we can only try to minimize an approximation to the objective function given by the m samples in Q:

$$\hat{\theta}^* = \arg\min_\theta \frac{1}{m} \sum_{q_i \in Q} M(q_i, f_\theta(q_i, S)). \tag{1}$$

The empirical objective function $\frac{1}{m}\sum_{q_i \in Q} M(q_i, f_\theta(q_i, S))$ can be also used to compare different scheduling functions f_θ. The learnable parameters of this function cannot be directly optimized with respect to the objective function by gradient-based methods, because it involves discrete choices. We will therefore construct the function f_θ in two stages, by first learning to approximate the runtimes of individual solvers conditioned on the problem and then constructing the final schedule based on the predicted runtimes.

IV. Graph Neural Networks

GNNs are neural networks which process inputs structured as a graph. For this reason, GNNs became popular for processing all kinds of formal structures such as logical expressions, which are naturally represented as trees or directed acyclic graphs.

Additionally, meta-information for nodes can be encoded as a feature vector of a fixed size. In our case, we encode the symbols used within the expression. For the sake of keeping the alphabet small, we abstract away specific numeric values of variables and function names and encode the resulting symbols as one hot vectors. Those representations serve as initial feature vectors for nodes.

Each layer of a GNN updates the feature vectors of all nodes by transforming and aggregating the feature vectors of its neighbour nodes.

Let $G = (V, E)$ be a graph with a feature vector x_v assigned to each node ($x_v \in \mathbb{R}^m$ where m is the length of the alphabet of possible symbols). Furthermore, let $N(v)$ denote the neighbourhood of a node v, i.e. the set of all nodes adjacent to v. In each propagation step $k = 1, 2, 3 \ldots$, a new feature vector $h_v^{(k)}$ (s.t. $h_v^{(0)} = x_v$) is computed in the following way [9]:

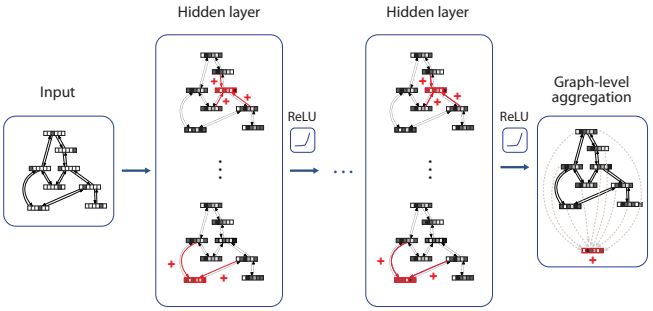

Fig. 1. Evaluation of the GNN on an input graph to obtain a single feature vector. The vectors highlighted in red are the once being aggregated.

$$a_v^{(k)} = \mathsf{AGGREGATE}^{(k)}\left(\left\{h_u^{(k-1)} \mid u \in N(v)\right\}\right)$$

$$h_v^{(k)} = \mathsf{COMBINE}^{(k)}\left(h_v^{(k)-1}, a_v^{(k)}\right)$$

Often these two steps are integrated and the aggregation is done over $N(v) \cup \{v\}$. For example, a simple node update rule used in the basic Graph Convolutional Network model (GCN) [10] has the following form:

$$h_u^{(k)} = \sigma\left(W^{(k)} \sum_{v \in N(u) \cup \{u\}} \frac{h_v^{(k-1)}}{\sqrt{|N(u)||N(v)|}}\right)$$

The matrix $W^{(k)} \in \mathbb{R}^{dim(h_u^k) \times dim(h_u^{k-1})}$ contains trainable parameters, the square root in the denominator scales the vectors according to a degree of the respective node, and σ is the application of a non-linear activation function (such as ReLU). We stress that $W^{(k)}$ is shared by all nodes in layer k but may differ across layers, allowing successive change of feature vector sizes.

In our initial experiments, we tried more advanced architectures, but in the final implementation we use the basic GCN described above because it produced no or only negligible improvements. GNNs can either be used to produce *node-level predictions* or *graph-level predictions*. In the case of graph-level predictions, we have to aggregate the feature vectors for every node within a graph to one output feature vector, which is then used to make the final prediction.

The process for obtaining one feature vector from the input graph is depicted in Figure 1. For the final prediction, we use a simple Multi-layer perceptron and train the whole network using backpropagation. In our case, we train the network to predict the runtimes of individual solvers.

The advantage of using a GNN is that processed graphs are allowed to be of different size, as transformations are applied locally and aggregation operator does not require a specific number of inputs.

V. Solver Scheduling

As mentioned in Section III, we aim to obtain a problem-dependent scheduler that minimizes Equation 1. The objective function in Equation 1 cannot be optimized directly by gradient-based methods, and therefore we solve the problem in two stages. First, we train our GNN to predict the runtimes of individual solvers and then create a schedule based on these predictions.

In our experiment, we tested a very simple schedule which divides the overall runtime into all solvers uniformly and orders the solvers according to the predicted runtimes. We also tried to schedule the solvers with times proportional to the predicted runtimes but this did not lead to a significant improvement and therefore we report the results for the simpler schedule.

VI. Experiments

A. Datasets

We test our approach and the other baselines on 4 representative benchmarks. As our goal is to show that we can improve upon the best individual solver and the other baselines on a given benchmark, we choose benchmarks with a noticeable gap between the best individual solver and the virtual best solver. The virtual best solver represents an upper bound of what could be achieved. It simply selects the best solver for each problem independently. When selecting the benchmarks, we have also taken into consideration the number of problems on a benchmark because our GNN contains a large number of learnable parameters and therefore could overfit on small datasets. A summary of the chosen benchmarks follows.

QF-NRA Is the logic of non-linear real arithmetic without quantifiers (QF stands for quantifier free). It involves arithmetic operations on real numbers. Since it is non-linear multiplications between two unknowns are allowed.

UFNIA The logic combines two theories, uninterpreted functions (UN) and non-linear integer arithmetic (NIA). The formulas may contain quantifiers. The logic is highly useful in software verification but at the same time is undecidable.

UFNIA-CONF It is customary that in competitions solvers run multiple strategies in sequence to tackle any given problem. This is especially true for formulas containing quantifiers where a number of different approaches exist, which exhibit strong degree of orthogonality [11]. To further diversify our experimental results, we have collected the different strategies that the solver CVC5 uses to solve UFNIA formulas in the competition.[1]

B. Data processing

For the processing of individual SMT formulas, we used the PySMT library [12]. Concretely, we used the built-in SMT-LIB parser and our custom printer which outputs the formula as a directed acyclic graph (DAG) with nodes labeled by possible symbols. The alphabet of possible symbols is handled by PySMT, which abstracts away details such as function names or specific numeric values. To create the final form of the graph which goes as an input to the GNN, we augment the DAG we obtain from PySMT with edges going backwards. That is, for every edge (a, b) we add edge (b, a) to the graph. This process is depicted in Figure 2.

[1] https://github.com/cvc5/cvc5/blob/master/contrib/competitions/smt-comp/run-script-smtcomp-current

Benchmark name	# of problems	# of solvers	Timeout (s)
QF_NRA	2654	9	2400
UFNIA	5659	7	2400
UFNIA-CONF	5659	23	60

TABLE I
Number of problems and solvers per benchmark.

C. Training the GNN

We collect all examples on a given benchmark and split them into training and testing samples as described in the next section. When searching for hyperparameters, we set aside 15% of the training set and use it as a validation set.

As a loss function for training, we use a *l2 loss* and rescale the ground truth times using a logarithm of base 2. As mentioned previously, we used GCN for our experiments. We set the number of graph convolutional layers to $k = 6$ and the hidden representation vector size to 150 for all of them with exception of the final GCN layer size set to 700. To aggregate feature vectors from all nodes into one feature vector, we use the max aggregation operator. It takes element-wise maximum across all feature vectors. This feature vector goes as an input to a 2-layer MLP with the size of the hidden layer equal to 350. The output size is equal to the number of solvers on a given benchmark.

To optimize the weights of the network, we use the ADAM optimizer [13] with the learning rate set to 0.001 and a batch size of 16.

D. Evaluation protocol

For the evaluation of individual models, we run the same procedure as Scott et al. [2]. That is, we shuffle the dataset and split it into training and testing subsets five times ($K = 5$) so that 80 % of the data is used for training and 20 % is used for testing. This is the same procedure as used in K-fold cross-validation. Testing sets from all 5 folds are disjoint and cover the whole dataset. For each fold, the model is always initialized with random weights, trained for 100 epochs and then evaluated on the respective testing set. This ensures that we obtain predictions for every problem within the dataset.

To compare individual solvers and schedules, we compute their PAR-2 score. This score is the sum of runtimes over all problems within the dataset. For problems that are not solved, the runtime is set to twice the timeout. To compute the runtime for various schedules, we iterate over individual solvers within the schedule and check if the problem would be solved by the given solver within the interval dedicated to it. If the problem is solved by the n-th solver within the schedule by using time t of its current interval and the length of each previous interval

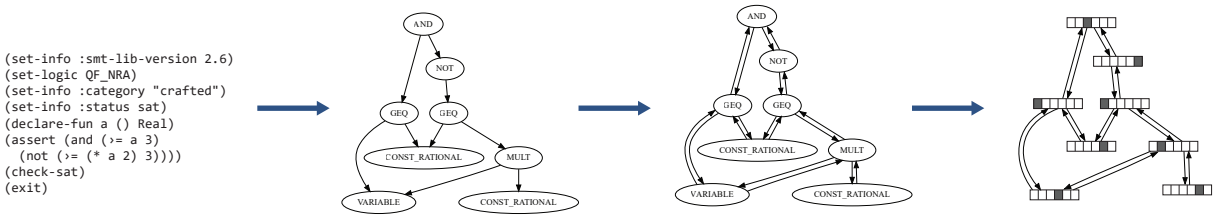

Fig. 2. The steps conducted during the creation of the input graph from a given SMT formula. Parsing yields a directed graph, which is then augmented by edges going in reverse direction. Finally, symbols on nodes are encoded to one-hot vectors.

i would be s_i, we would set the runtime to $t + \sum_{i=1}^{n-1} s_i$. If the problem is not solved by any solver within the interval dedicated to it, we set its runtime to twice the timeout.

E. Description of the tested approaches

The results for the following baselines and our approach are presented in the Table II.

Best solver. Is a single solver with the best PAR-2 score. We present our results as an relative improvement over the best solver.

Virtual best solver (VBS). This is a hypothetical algorithm that selects the best solver for each problem separately. It represents an upper bound for a possible improvement.

BOW-single and GNN-single. These two baselines choose the best solver according to the predictions of the model and run it until the timeout. They differ only in the ML model. Both models are trained to regress the ground truth runtimes and we select the solver with the shortest predicted runtime.

BOW-single uses a model similar to the one used in MachSMT. It uses bag-of-words features, which contain the counts of each symbol within the formula. Depending on the benchmark, the size of the feature vector is in the range 12–20. For the regression, we use LightGBM [14] and a grid search to find suitable hyperparameters.

GNN-single uses the same model described in the Section VI-C.

Random schedule. Random schedule divides the whole available time to all available solvers and runs them in a random order. We found out that for 2 of the tested benchmark sets, this simple schedule outperforms the approach which selects only the solver with the best prediction. This shows serious drawbacks of the results presented in MachSMT and also the importance of scheduling more solvers. Random schedule is effective because many problems are solved in a short time by at least one solver.

Solver ordering (GNN). An improvement over choosing only the predicted best solver is to choose k best solvers and split the available time across them. In the extreme case, we can choose all available solvers, split the available time across them uniformly, and order them by the predicted time starting with the solver with the shortest predicted time. For the predictions, we use the same model as in GNN-single.

Benchmark		**QF-NRA**	**UFNIA**	**UFNIA-CONF**
Best Solver	solver	Z3	CVC4	-
	solved	2120	3093	2494
VBS	solved	2516	3339	3118
BOW single	PAR-2 impr.	117.10%	-0.64%	0.32%
	solved	2343	3074	2586
GNN single	PAR-2 impr.	231.19%	1.8%	56.73%
	solved	2403	3085	2644
Random schedule	PAR-2 impr.	269.33%	-21.62%	69.59%
	solved	2494	3053	2812
Solver ordering	PAR-2 impr.	**913.05%**	**8.30%**	**88.59%**
	solved	2494	3053	2812

TABLE II
COMPARISON OF EVALUATED APPROACHES. THE PAR-2 IMPROVEMENT IS RELATIVE TO THE BEST SOLVER.

F. Results

The results of our experiments are visible in the Table II. The PAR-2 improvement is relative to the best solver and it takes into consideration only the problems which were solved by at least one solver. Our schedule clearly outperforms the other approaches. We can also notice that for QF-NRA and UFNIA-CONF, the random schedule outperforms GNN-single and BOW-single, which demonstrates the importance of a schedule.

VII. RELATED WORK

Algorithm selection and scheduling [15] is recognized as an important topic as a consequence of the need for reliable and fast problem handling in practical applications. Using machine learning methods for selection of a solver from a portfolio was popularized by Leyton-Brown et al. [16]. In the literature, models which predict the runtime of individual algorithms are usually called Empirical Hardness Models [17].

In terms of the goal and used benchmarks, our work is most similar to MachSMT [2]. The main difference is that they try to select only one solver from the whole portfolio and use bag-of-words as the representation of a formula. We found out that in many cases, the best solver according to the prediction of the model does not solve the formula at all and that schedules, including the random one, work better. We also train the feature extraction together with the final regressor end-to-end, and therefore our approach may discover more complex and useful features.

There are many other approaches that demonstrate the possibility of using Graph Neural Networks to extract the

features of various formulas and learn the final prediction in an end-to-end fashion. In the domain of SAT solvers, Selsam et al. [18] train a GNN to predict the satisfiability of a formula. The trained network can later be used to find a solution for new formulas. The same problem is studied by Cameron et al. [19] who used different architectural choices for the GNN. In the following work, Selsam et al. uses a GNN to guide a SAT solver [20]. The GNN was trained to predict the unsatisfiable core of a given formula and these predictions were then used for variable selection inside the SAT solver. Wang et al. [21] use a GNN to embed and predict the relevance of logical formulas in the task of premise selection.

For an overview of various use cases of ML methods for combinatorial problems and algorithm selection, see the following survey papers: [22]–[24]. For a more specific overview focused on GNNs see [25].

VIII. Conclusion

This paper presents an application of GNNs for SMT solver scheduling. We showed that GNNs can be successfully used as Empirical Hardness Models. Their main advantage, in comparison to other ML methods, is that they can be used without manual feature engineering. We also showed the benefits of using a schedule. In our experiments, we compared our approach to several baselines and demonstrated significant improvements in terms of the number of solved problems and overall solving time. In future work, we plan to focus on meta-learning of GNNs to quickly adapt to new problem distributions and on the augmentation and subsampling strategies for individual formulas.

IX. Acknowledgement

This scientific article is part of the RICAIP project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 857306. The results were supported by the Ministry of Education, Youth and Sports within the dedicated program ERC CZ under the project POSTMAN no. LL1902. David Mojzisek was supported by the Czech Science Foundation project 20-06390Y.

References

[1] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "SATzilla: portfolio-based algorithm selection for SAT," *J. Artif. Intell. Res.*, vol. 32, pp. 565–606, 2008. [Online]. Available: https://doi.org/10.1613/jair.2490

[2] J. Scott, A. Niemetz, M. Preiner, S. Nejati, and V. Ganesh, "MachSMT: a machine learning-based algorithm selector for SMT solvers," in *TACAS*, 2020.

[3] W. L. Hamilton, "Graph representation learning," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 14, pp. 1–159, 2020.

[4] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2009, pp. 825–885.

[5] A. Reynolds, V. Kuncak, C. Tinelli, C. W. Barrett, and M. Deters, "Refutation-based synthesis in SMT," *Formal Methods Syst. Des.*, vol. 55, no. 2, 2019.

[6] L. de Moura and N. Bjørner, "Applications and challenges in satisfiability modulo theories," in *Workshop on Invariant Generation (WING)*, vol. 1. EasyChair, 2012, pp. 1–11.

[7] P. Godefroid, M. Y. Levin, and D. A. Molnar, "SAGE: whitebox fuzzing for security testing," *Commun. ACM*, vol. 55, no. 3, pp. 40–44, 2012.

[8] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB Standard: Version 2.0," in *SMT Workshop*, 2010.

[9] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" in *ICLR*, 2019.

[10] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[11] A. Reynolds, H. Barbosa, and P. Fontaine, "Revisiting enumerative instantiation," in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 10806, 2018, pp. 112–131.

[12] M. Gario and A. Micheli, "PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms," in *SMT Workshop*, 2015.

[13] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *ICLR*, Y. Bengio and Y. LeCun, Eds., 2015.

[14] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T. Liu, "LightGBM: A highly efficient gradient boosting decision tree," in *NeurIPS*, 2017.

[15] S. Kadioglu, Y. Malitsky, A. Sabharwal, H. Samulowitz, and M. Sellmann, "Algorithm selection and scheduling," in *International Conference on Principles and Practice of Constraint Programming*, 2011.

[16] K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, and Y. Shoham, "A portfolio approach to algorithm selection," in *IJCAI*, vol. 3, 2003, pp. 1542–1543.

[17] K. Leyton-Brown, E. Nudelman, and Y. Shoham, "Empirical hardness models: Methodology and a case study on combinatorial auctions," *J. ACM*, vol. 56, no. 4, pp. 22:1–22:52, 2009.

[18] D. Selsam, M. Lamm, B. Bünz, P. Liang, L. de Moura, and D. L. Dill, "Learning a SAT solver from single-bit supervision," in *International Conference on Learning Representations*, 2019.

[19] C. Cameron, R. Chen, J. Hartford, and K. Leyton-Brown, "Predicting propositional satisfiability via end-to-end learning," in *AAAI*, 2020.

[20] D. Selsam and N. Bjørner, "Guiding high-performance SAT solvers with unsat-core predictions," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2019, pp. 336–353.

[21] M. Wang, Y. Tang, J. Wang, and J. Deng, "Premise selection for theorem proving by deep graph embedding," in *NeurIPS*, 2017.

[22] Y. Bengio, A. Lodi, and A. Prouvost, "Machine learning for combinatorial optimization: a methodological tour d'horizon," *European Journal of Operational Research*, 2020.

[23] P. Kerschke, H. H. Hoos, F. Neumann, and H. Trautmann, "Automated algorithm selection: Survey and perspectives," *Evolutionary computation*, vol. 27, no. 1, pp. 3–45, 2019.

[24] E.-G. Talbi, "Machine learning into metaheuristics: A survey and taxonomy of data-driven metaheuristics," *ACM Computing Surveys*, 2020.

[25] Q. Cappart, D. Chételat, E. Khalil, A. Lodi, C. Morris, and P. Veličković, "Combinatorial optimization and reasoning with graph neural networks," *arXiv preprint arXiv:2102.09544*, 2021.