LLM-Guided Graph Generation for Structure-Based Local Improvement Methods

Hai Xia HXIA@AC.TUWIEN.AC.AT
Vaidyanathan Peruvemba Ramaswamy VAIDYANATHAN@AC.TUWIEN.AC.AT
Stefan Szeider SZ@AC.TUWIEN.AC.AT
Algorithms and Complexity Group, TU Wien, Austria

Abstract

Large neighborhood search normally selects a random subset of decision variables for iterative optimization. To efficiently solve various problems, researchers tend to design variable selection strategies that take into account structural features across different domains. In this paper, we build an automatic pipeline that is problem-agnostic to all problems in the MiniZinc format. By prompting an LLM with our semantic guidelines, we guide the LLM to produce a graph generator that maps any instance of a problem type to a uniform weighted graph, where nodes represent decision variables and edges represent constraint relationships. These problem-agnostic graphs guide our structure-based local improvement (SLIM) framework for variable selection. Meanwhile, the weighted graph enables all problem instances to share the same generic graph representation, from which the same graph features can be extracted and used for configuration selection. We evaluated our pipeline on instances across 20 MiniZinc competition problems, finding that algorithm selection achieves a 39.6% average problem-weighted win rate against a one-shot Gurobi baseline, more than doubling the best single configuration (19.3%). A post-hoc configuration and a feature ablation indicate a headroom of up to 44.0%, demonstrating that LLM-based semantic generation enables effective automated structure and feature extraction for constraint optimization.

1. Introduction

Large Neighborhood Search (LNS) is one of the most useful metaheuristics for hard combinatorial optimization problems (Shaw, 1998; Pisinger and Ropke, 2019), especially when the problems become larger and more complicated. In each iteration, LNS selects only a subset of the decision variables, which is optimized by an exact solver. After the local instance is optimized, the better local solution is patched back to the original global one. Therefore, the efficiency of the search relies heavily on whether promising and suitable variables are selected for local optimization.

In practice, variable selection (neighborhood selection) is quite flexible. Random selection is the simplest one, which is computationally cheap, but ignores the instance features and the problem structure, wasting solving time on loosely coupled subproblems. Structure-guided variable selection can exploit the variable relations, producing tightly coupled neighborhoods. But what kind of features or structure information should be considered is also a question quite related to the specific problem type, and it needs a lot of expert domain knowledge. For different problem domains, there are different principles for designing efficient Structure-Based Local Improvement Methods (SLIM) (Fichte et al., 2017). In the

last decade, SLIM has been applied to different problem domains, including treewidth computation (Fichte et al., 2017), branchwidth (Lodha et al., 2019), treedepth (Ramaswamy and Szeider, 2020), Bayesian network learning (Ramaswamy and Szeider, 2021; Peruvemba Ramaswamy and Szeider, 2022), graph coloring (Schidler and Szeider, 2023), decision tree optimization (Schidler and Szeider, 2024b), and Maximum Satisfiability (Schidler and Szeider, 2024a). In different problem-specific SLIM algorithms, there are various construction methods for neighborhoods and corresponding variable selection strategies. All these designs are highly engineered by domain experts, taking months of manual work. Besides the algorithm design, efficient algorithm implementation is also not trivial when applied to different problem instances: algorithm configurators typically demand extensive computational budgets for getting better configurations of SLIM algorithms (Ramaswamy et al., 2024), and portfolio-based algorithm selection requires problem-specific features that also need to be curated by experts (Xia and Szeider, 2024).

To alleviate the difficulty of problem-specific application of SLIM, we build a pipeline where we can easily build structure-aware SLIM algorithms for different problems with the help of a Large Language Model (LLM). First, by giving the MiniZinc[1] constraint models to the LLM, we can prompt with our semantic guidelines to guide the LLM to produce a Python program (a *graph generator*) that can map all problem-specific instances to a uniform weighted generic graph. In the generic graph, the nodes correspond to decision variables and take weights and domain sizes. Meanwhile, the edges represent the constraint relations between different nodes, carrying coupling strengths. Therefore, the graph has no problem-specific properties, and it is a problem-agnostic structure representation. Then SLIM can operate on the uniform graph by some basic extraction algorithms (extraction based on breadth-first search (BFS) and weighted random sampling). The generic extraction algorithms can select the neighborhoods with semantic structure information derived from the original constraints. Furthermore, the problem-generic representation also makes it possible to do cross-problem algorithm selection, as we can extract topological and statistical features from the generic graphs of the instances from different problems.

In brief, we have the contributions as follows.

1. We propose using an LLM to produce validated graph generators covering different problems. With the graph generator, generic graphs can be generated in a uniform way for instances from different problems.

2. By using the graph generators, we build problem-agnostic SLIM algorithms easily across different problems without specific expert knowledge.

3. We introduce cross-problem configuration selection using generic graph-based features. From the extensive evaluation on 20 MiniZinc competition problems, we find that our model achieves a 39.6% average problem-weighted win rate against a one-shot Gurobi baseline, more than doubling the best single configuration (19.3%), with a post-hoc ablation analysis indicating potential of up to 44.0%.

1. MiniZinc is a solver-independent constraint modeling language: https://www.minizinc.org

2. Related Work

Large neighborhood search. Shaw (1998) introduced LNS for vehicle routing, showing that structure-aware destruction outperforms random selection. Pisinger and Ropke (2019) survey the area, including adaptive LNS (Ropke and Pisinger, 2006). Recent neural variants learn the destroy/repair policy (Hottung and Tierney, 2022; Johnn et al., 2023) but remain tied to specific problem types. In contrast, we generate problem-agnostic graph structures offline and apply them to any MiniZinc problem type after a one-time generator synthesis, without any per-problem neural training.

Structure-guided local improvement. We build on SLIM, introduced for treewidth by Fichte et al. (2017) and since extended to branchwidth (Lodha et al., 2019), treedepth (Ramaswamy and Szeider, 2020), Bayesian network learning (Ramaswamy and Szeider, 2021; Peruvemba Ramaswamy and Szeider, 2022), graph coloring (Schidler and Szeider, 2023), decision trees (Schidler and Szeider, 2024b), and Maximum Satisfiability (Schidler and Szeider, 2024a). Each application required substantial domain-specific engineering, but we automate this step with an LLM, turning SLIM into a problem-agnostic methodology without needing too much domain knowledge.

Structure from constraint models, and LLMs for optimization. Exploiting constraint-graph topology for solving is classical (Dechter and Pearl, 1989; Gottlob et al., 2002), and recent work learns branching policies from variable–constraint graphs (Gasse et al., 2019). These extract *syntactic* structure, whereas our LLM assigns weights reflecting semantic roles. LLMs have been used to discover programs (Romera-Paredes et al., 2024), as iterative optimizers (Yang et al., 2024), to generate constraint models from natural language (Singirikonda et al., 2025), and to enforce constraints during LLM decoding (Bonlarron et al., 2025). We instead use the LLM as a one-time offline compiler that produces deterministic, auditable generators.

Algorithm selection. Portfolio-based selection is well studied (Xu et al., 2008; Lindauer et al., 2015; Amadini et al., 2014), typically within a single domain using domain-specific features. Because all our instances share one uniform graph representation, a single model selects configurations across 20 heterogeneous problem types without per-problem feature engineering. The supplementary material gives an extended discussion.

3. Methodology

3.1. Pipeline Overview

For each MiniZinc constraint model, we can generate a Python program (a *graph generator*) with the help of the LLM, which can be used for transforming different instances of the problem into a generic graph representation. Therefore, the generic graph can be used for guiding the variable selection in the SLIM framework and enabling algorithm configuration selection in a problem-agnostic fashion according to the uniform graph features. To this end, the whole pipeline has two parts: the *problem-specific* part and the *problem-agnostic* part. Figure 1 illustrates the general mechanism of our framework.

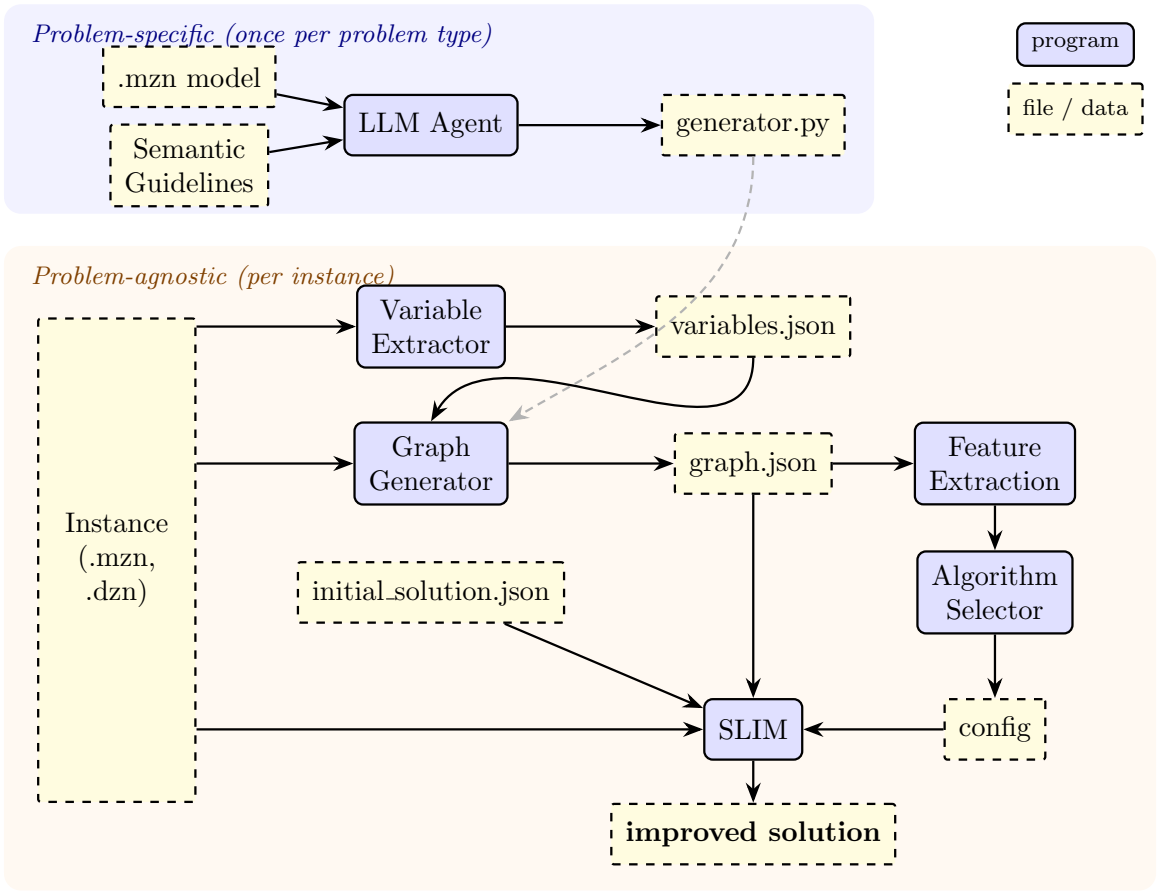

Figure 1: Pipeline overview. Programs (solid border) process files (dashed border). The blue region shows the one-time generator creation by the LLM, and the orange region shows per-instance processing. The instance files (`.mzn` + `.dzn`) feed the Variable Extractor, the Graph Generator, and SLIM (which solves subproblems on the original model). The dashed arrow indicates that `generator.py` is instantiated as the Graph Generator.

In the problem-specific stage, the LLM (Claude Opus 4.5[2]) reads the MiniZinc model (`.mzn`) for a given problem type together with our semantic weight guidelines and produces a *graph generator*, a Python program that understands the semantic structure of that problem's constraints and variables. This is a one-time effort for each problem: we synthesize generators for all problems, and they are applicable to all instances from MiniZinc competitions (2008–2025). Of these, instances across 20 problem types satisfy our benchmarking criteria (see Section 4). Each synthesis for a single problem usually takes within several minutes, freeing experts from curating their own problem-specific SLIM algorithms.

In the problem-agnostic stage, the variable extractor parses the decision variables with their domains and array indices from the MiniZinc instances and the model, producing

2. https://www.anthropic.com/claude

a `variables.json` file. With the parsed `variables.json` file, the LLM-generated graph generator can construct weighted graphs (with the instance information), wherein nodes represent decision variables (with importance weights and domain sizes) and edges indicate the constraint relations (with coupling strengths). Therefore, SLIM can also use the generic graph for selecting variables according to the weighted neighborhoods for local improvements. For example, the BFS-based and the random-based extraction algorithms can operate on the generic graph, which is from a vehicle routing problem (VRP) instance or a scheduling instance. The problem domain information is implicitly contained in the generic graph. And the generic problem-agnostic SLIM algorithm has no idea what domain problem it is solving. Based on the same generic graph representation, the graph features, like the topological and statistical features of the uniform graph, are also extracted and then used for algorithm configuration selection (among 30 SLIM configurations) for better solving efficiency.

3.2. LLM-Guided Graph Generation

The general idea for the graph generation is using the LLM as a *semantic compiler*. With the guidelines set by human researchers, the LLM can generate a Python program (the graph generator), where the instance can be mapped into a weighted graph. The semantic approach can capture the relationships between the variables and constraints. For example, we can set a general range in the prompts regarding the emphasis of different constraints: the variables in an `alldifferent` constraint over 50 elements should be connected with light edges to avoid clique domination.

3.2.1. Problem-Agnostic Graph Format

For each instance, the graph generator can produce a uniform graph representation:

- **Nodes** indicate decision variables. Each node has two properties: a weight $w \in [0, 1]$ (importance to the objective) and a domain size $d \in \mathbb{N}$ (number of possible values).
- **Edges** indicate constraint relations between different decision variables, each with a weight $w \in [0, 1]$ showing how the decision variables are coupled.

The uniform representation allows different components of SLIM and algorithm selection to work in a generic way. For example, the budget computation is counted according to the number of nodes selected and the corresponding domain sizes.

3.2.2. Generation Guidelines

Here are our defined guidelines for the LLM to generate targeted graph generators with semantic information:

1. **Objective-related components should have higher weights** (≥ 0.6), because they usually have direct influence on the final objective values. So these components should have higher weights to be selected by the extraction algorithms during the SLIM optimization phase.

2. **Lower weights for large global constraints**: as there are some global constraints among the decision variables, if we give high weights to all these global constraints, then the weights of different components will be similar, resulting in no preference during the variable selection. Large constraints (e.g., `alldifferent` over $n > 10$ variables) use weight $\max(0.1,\ 1/n)$ to prevent clique domination, while small constraints ($n \leq 5$) retain strong coupling (≥ 0.8).

3. **No isolated nodes**, as the graph-based extraction can only reach the nodes that are connected by different edges. We have to make sure all nodes have the possibility to be selected and optimized without any search space left out.

4. **Bounded weights**: to make the extraction algorithms operate in a uniform way, we set an upper bound for the weights. When there are several constraints linking the same variables, edge weights are aggregated according to $W = 1 - \prod_i (1 - w_i)$, resulting in weights never exceeding 1.

A worked example of the graph-generation process is given in the supplementary material.

3.3. Generic Structure-Based Local Improvement (SLIM)

As we already have a generic graph representation from the graph generator, SLIM can now work on a problem-agnostic uniform weighted graph. Even though SLIM now has no prior knowledge or preferences for different problems, it can extract the variables with some useful information, like the weights and the domain sizes of the nodes, and the coupling weights of the edges in the graph. In each iteration, we use the extraction algorithms to select the neighborhood of variables from the uniform graph, while the remaining variables outside of the neighborhood are frozen. Then the local solver can optimize the smaller subproblem, resulting in better local solutions.

3.3.1. Extraction Methods

Our SLIM has two extraction methods based on the generic graph:

- **BFS extraction** starts from a randomly chosen node according to the node weights, and then expands nodes via edges in a weight-biased random order. These can capture the locality, with considerations of both the importance of the decision variables and the constraint structures of the original problem.

- **LNS extraction** collects random variables using the corresponding node weights, but it has no consideration regarding the locality preferences like BFS does, where a node can be expanded only when it is connected with the current collected component by an edge.

Both of the extractions stop when the current budget is reached: the *budget* parameter b is a threshold bounded by a domain-size-aware metric. Besides the collection of variables to be optimized, we also have another freeze mode, which is for collecting the variables to be frozen instead of to be optimized. We have a detailed description in the supplementary material.

In the whole SLIM working loop (see the supplementary material for the pseudocode), it starts from an initial solution s_0, which is usually a suboptimal solution obtained by heuristics or by running the solver with a short timeout. Then SLIM extracts a neighborhood N from the graph using the extraction method $\mathcal{E}$ (BFS or LNS) with the given budget b. Meanwhile, the freeze mode f determines whether the variables collected should be frozen or should be optimized. To get a better solution, the bounding constraint $\text{obj}(s') \leq \text{obj}(s^*)$ (for minimization; $\geq$ for maximization) is added to the subproblem. Therefore, after the per-iteration timeout t, if the solver returns a feasible solution that is not worse than the current incumbent, the global solution can be updated. SLIM accepts equal-quality solutions, and this allows us to explore different solutions with the same objective value. With this working mechanism, the loop continues until the overall time budget is exhausted or the maximum number of iterations is reached.

In the SLIM framework, there are several configuration parameters having huge influence on the final performance. This is also one of the reasons why we have the following algorithm configuration selection for boosting the performance. For example, the local budget and the local time indicate different preferences for different local structures to be optimized. In our setting, we evaluate 30 configurations in total, including the local budget $b \in \{10, 20, 50, 70, 100, 200\}$, the timeout $t \in \{20, 30, 45, 60\}$ s, and different extraction strategies.

3.4. Algorithm Selection

As we discussed, the 30 SLIM configurations with different strategies have various preferences for solving problem instances. Note that even though there are sophisticated algorithm selectors (Xu et al., 2008; Lindauer et al., 2015; Amadini et al., 2014), they are only applicable to different instances of the same problem. In our setting, we can train a machine learning model to select the best configuration for different instances across problems, as we have a uniform graph representation, from which we can extract the features. We build five simple selection approaches to demonstrate the effectiveness of our pipeline. Any advances in algorithm selection would further improve our results.

There are different methods for training the algorithm selector. Here we only use the regression approach as a representative example to show how the pipeline works (see the supplementary material for the pseudocode). In the training phase, the feature vector $\mathbf{x}_i$ is first extracted from each instance's graph, and per-configuration improvement margins $\mathbf{y}_i$ are the differences between each configuration's improvement and the one-shot baseline improvement. Then we train a multi-output random forest regression model with problem-weighted samples as different problems have quite different numbers of instances. After the cross-validation, the final model is retrained on the full training set. Next, we could evaluate how good the portfolio algorithm is by applying the algorithm predicted by the machine learning model. The other four algorithm selectors proceed with the same workflow, and the only difference is about the training loss objective and what they actually predict during the test. The supplementary material lists the summary of the five different algorithm selectors.

In our feature extraction setting, we design a feature set consisting of 54 features computed from the weighted uniform graph of each instance together with lightweight instance metadata (see the supplementary material). The feature set includes topology statistics,

node and edge weight distributions, domain size statistics and correlations, variable metadata, and numeric instance parameters. As we have the uniform graph format across problems, the same 54 features work identically for all problem instances. Note that the dataset is heavily imbalanced (the resource-constrained project scheduling problem, RCPSP, alone accounts for 38%). To prevent specific problems from dominating the evaluation, we assign problem-weighted sample weights so that each of the problems contributes equally during training as well as the final evaluation on the test set.

In the ablation analysis on the feature set (54 features) and the algorithm portfolio set (30 configurations), we apply greedy backward elimination. The corresponding experimental results are in the supplementary material. The configuration ablation removes configurations whose elimination improves the final performance on the test instances, and feature ablation subsequently removes features from the reduced configuration set. We report this as a post-hoc analysis of the pruning potential.

4. Experimental Evaluation

In this section, we show the experimental analysis of our pipeline on different problems.

4.1. Experimental Settings

As we mentioned in Section 3.1, we collect problems from MiniZinc competitions[3] (2008–2025) resulting in instances across 20 problem types after we filter out unsuitable instances. We have several criteria for the selection: 1. We include instances where Gurobi (a commercial exact solver) finds a feasible solution within 10 minutes but does not prove optimality within 60 minutes. 2. We only include problems with at least 5 qualifying instances. For algorithm selection, we split the instances with a 70:30 ratio, stratified by problem type, resulting in the instance distribution on training and test sets as shown in the supplementary material.

We run all experiments on a Sun Grid Engine cluster with 20 nodes running Ubuntu 18.04 LTS. Each node has two Intel Xeon E5-2640 v4 2.40 GHz CPUs and 160 GB RAM. SLIM and related programs use Python 3.6.9. The Zenodo repository[4] contains materials for reproducing the results.

4.2. SLIM vs One-Shot Gurobi

We compare our problem-agnostic SLIM against the one-shot Gurobi baseline. The one-shot Gurobi means running Gurobi on the original MiniZinc instance with the same total time budget (60 minutes). As Gurobi is a strong industrial commercial solver, it has been widely used on many problems. We compare the solutions generated by Gurobi's default running and the solutions optimized with our problem-agnostic SLIM. As we introduced in Section 3.3, there are different strategies that can be used during running. Therefore, SLIM using Gurobi as the subsolver becomes standard LNS when the variable extraction is random and does not utilize the uniform weights.

3. https://www.minizinc.org/challenge/
4. https://doi.org/10.5281/zenodo.21910103

We show detailed comparison results among the 20 problems in Figure 2. On each problem, we use SLIM to run on all instances with the 30 configurations, and then we can compare the objective values of SLIM with the best-performing configuration against those generated by one-shot Gurobi. From the win/tie/loss percentages on each problem, we can see that on some problems, problem-agnostic SLIM can get high-performing results. On problems including tdtsp, spot5, community-detection, triangular, and opd, SLIM can win over 75% of the instances. Among all 20 problems, there are only two problems on which SLIM gets worse solutions on more than 50% of the instances: rectangle-packing and VRP.

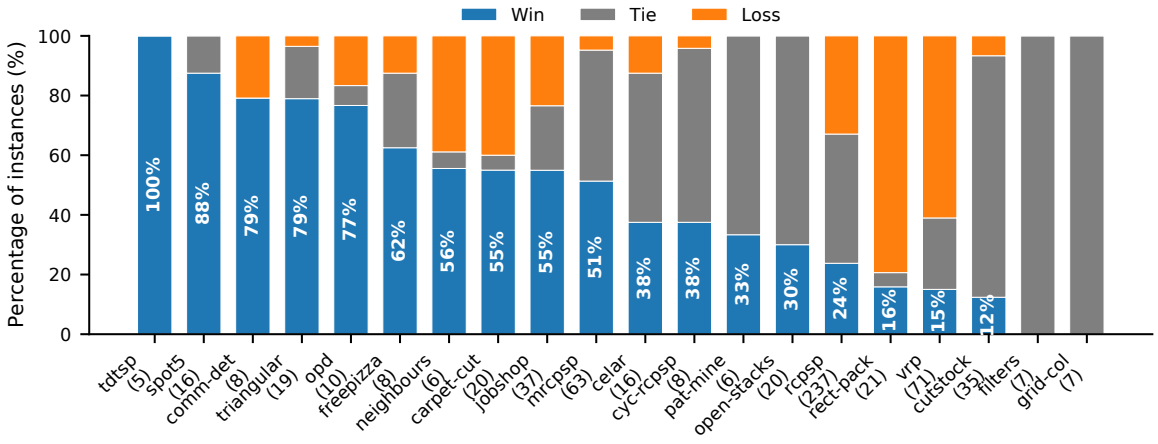

Figure 2: SLIM (best configuration per problem type) vs one-shot Gurobi baseline, averaged across 3 seeds. Problem types are sorted by win rate. Instance counts per type are shown in parentheses.

Even though SLIM failed to get any better solutions on the filters and grid-coloring problems, on all instances of these two problems SLIM can get the same good solutions as one-shot Gurobi. In general, when SLIM is equipped with suitable strategies, it can get substantially better solutions than the one-shot Gurobi. Meanwhile, the results also indicate that the configurations (suitable search strategies) are important, which motivates our problem-agnostic algorithm selection idea in the following section. We could use algorithm selection to learn a good mapping from different instances to the configurations for guiding the structure-guided search.

4.3. Algorithm Selection

We use the aforementioned five algorithm selection approaches across three random seeds (51, 52, 53). In each training run of the algorithm selectors, we use the same train/test instance distribution for a fair comparison.

In Table 1, we show the problem-weighted win rates of all five selection approaches against the best single configuration baseline. All five approaches substantially outperform the baseline across all seeds: the best single configuration reaches only 17.8–22.3% depending on the seed, while every selection approach exceeds 31%. Regarding the performance

across seeds, no single approach dominates others. The two algorithm selection approaches, regression and ensemble, win on two of the three seeds. It indicates that per-configuration improvement margins are more suitable for training the algorithm configuration selectors of the problem-agnostic SLIM. Even though the binary algorithm selection approach is the best-performing selector on only one seed, it is the most stable algorithm selector, whose win rate fluctuates within two percentage points. We can see that for all algorithm selection approaches, the win rate is always below 50%. Actually, it does not mean weak performance among the problems, as they still have a high tie rate for each problem. The detailed win/tie/loss rates are also shown in the supplementary material. In general, for the best-performing approach on each seed, the problem-weighted win rates are tightly clustered (37.9–40.6%) with positive net scores (win minus loss of 14.8–24.5%). And the performance of the portfolio algorithm is well above the best single configuration, and reasonably below the virtual best.

Table 1: Problem-weighted win rate (%) of all selection approaches across 3 seeds.

Approach	Seed 51	Seed 52	Seed 53
regression	**40.1**	40.2	34.2
ensemble	38.3	**40.6**	35.3
binary	36.4	36.9	**37.9**
classification	36.9	39.0	36.2
two_stage	34.3	37.2	31.9
Best single config	17.8	17.7	22.3

5. Conclusion

In this paper, we presented an automatic pipeline where an LLM is used for synthesizing graph generators to transform MiniZinc problem instances into a uniform graph representation. With the problem-generic graph, we can use structure-aware SLIM across different problems. Meanwhile, we can also extract the features from the uniform graph to automatically select the best-performing configurations for SLIM. By extensive evaluation on the 20 problems, we found that the problem-agnostic SLIM can get stronger results compared with one-shot Gurobi.

There are also limitations. When the problem-agnostic SLIM is applied to specific problems, SLIM is not able to outperform one-shot Gurobi even with the best search strategies. This is probably due to the LLM-guided graph generation. The generators, while validated, remain LLM-produced approximations of the true constraint semantics, and our evaluation covers only MiniZinc competition benchmarks. In the future, we may have more targeted prompts to guide the LLM to transform the instances into uniform graphs with other types of information, like some probing topological features and so on. Furthermore, we can even add some information that can be dynamically updated during the search. For example, after specific components have been optimized many times, we can use tabu search or adaptively adjust the weights to escape the repetitive optimization.

Acknowledgements

This project is also partially supported by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101034440, and by the Austrian Science Fund (FWF) within the Cluster of Excellence Bilateral Artificial Intelligence (10.55776/COE12) and 10.55776/P36420.

References

Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. SUNNY: a lazy portfolio approach for constraint solving. *Theory Pract. Log. Program.*, 14(4-5):509–524, 2014. doi: 10.1017/S1471068414000179. URL https://doi.org/10.1017/S1471068414000179.

Alexandre Bonlarron, Florian Régin, Elisabetta De Maria, and Jean-Charles Régin. Large language model meets constraint propagation. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2025, Montreal, Canada, August 16-22, 2025*, pages 10036–10044. ijcai.org, 2025. doi: 10.24963/IJCAI.2025/1115. URL https://doi.org/10.24963/ijcai.2025/1115.

Rina Dechter and Judea Pearl. Tree clustering for constraint networks. *Artif. Intell.*, 38(3): 353–366, 1989. doi: 10.1016/0004-3702(89)90037-4. URL https://doi.org/10.1016/0004-3702(89)90037-4.

Johannes Klaus Fichte, Neha Lodha, and Stefan Szeider. SAT-based local improvement for finding tree decompositions of small width. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10491 of *Lecture Notes in Computer Science*, pages 401–411. Springer, 2017. doi: 10.1007/978-3-319-66263-3_25. URL https://doi.org/10.1007/978-3-319-66263-3_25.

Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 15554–15566, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/d14c2267d848abeb81fd590f371d39bd-Abstract.html.

Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci.*, 64(3):579–627, 2002. doi: 10.1006/JCSS.2001.1809. URL https://doi.org/10.1006/jcss.2001.1809.

André Hottung and Kevin Tierney. Neural large neighborhood search for routing problems. *Artif. Intell.*, 313:103786, 2022. doi: 10.1016/J.ARTINT.2022.103786. URL https://doi.org/10.1016/j.artint.2022.103786.

Syu-Ning Johnn, Victor-Alexandru Darvariu, Julia Handl, and Jörg Kalcsics. GRAPH reinforcement learning for operator selection in the ALNS metaheuristic. In Bernabé

Dorronsoro, Francisco Chicano, Grégoire Danoy, and El-Ghazali Talbi, editors, *Optimization and Learning - 6th International Conference, OLA 2023, Malaga, Spain, May 3-5, 2023, Proceedings*, volume 1824 of *Communications in Computer and Information Science*, pages 200–212. Springer, 2023. doi: 10.1007/978-3-031-34020-8_15. URL https://doi.org/10.1007/978-3-031-34020-8_15.

Marius Lindauer, Holger H. Hoos, Frank Hutter, and Torsten Schaub. Autofolio: An automatically configured algorithm selector. *J. Artif. Intell. Res.*, 53:745–778, 2015. doi: 10.1613/JAIR.4726. URL https://doi.org/10.1613/jair.4726.

Neha Lodha, Sebastian Ordyniak, and Stefan Szeider. A SAT approach to branchwidth. *ACM Trans. Comput. Log.*, 20(3):15:1–15:24, 2019. doi: 10.1145/3326159. URL https://doi.org/10.1145/3326159.

Vaidyanathan Peruvemba Ramaswamy and Stefan Szeider. Learning large Bayesian networks with expert constraints. In James Cussens and Kun Zhang, editors, *Proceedings of the Thirty-Eighth Conference on Uncertainty in Artificial Intelligence, UAI 2022*, volume 180 of *Proceedings of Machine Learning Research*, pages 1592–1601. PMLR, 2022. URL https://proceedings.mlr.press/v180/peruvemba-ramaswamy22a.html.

David Pisinger and Stefan Ropke. Large neighborhood search. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, pages 99–127. Springer International Publishing, Cham, 2019. ISBN 978-3-319-91086-4. doi: 10.1007/978-3-319-91086-4_4. URL https://doi.org/10.1007/978-3-319-91086-4_4.

Vaidyanathan Peruvemba Ramaswamy and Stefan Szeider. MaxSAT-Based postprocessing for treedepth. In Helmut Simonis, editor, *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings*, volume 12333 of *Lecture Notes in Computer Science*, pages 478–495. Springer, 2020. doi: 10.1007/978-3-030-58475-7_28. URL https://doi.org/10.1007/978-3-030-58475-7_28.

Vaidyanathan Peruvemba Ramaswamy and Stefan Szeider. Turbocharging treewidth-bounded bayesian network structure learning. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 3895–3903. AAAI Press, 2021. doi: 10.1609/AAAI.V35I5.16508. URL https://doi.org/10.1609/aaai.v35i5.16508.

Vaidyanathan Peruvemba Ramaswamy, Stefan Szeider, and Hai Xia. The power of collaboration: Learning large bayesian networks at scale. In *36th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2024, Herndon, VA, USA, October 28-30, 2024*, pages 371–378. IEEE, 2024. doi: 10.1109/ICTAI62512.2024.00061. URL https://doi.org/10.1109/ICTAI62512.2024.00061.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg,

Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625(7995): 468–475, 2024. doi: 10.1038/S41586-023-06924-6. URL https://doi.org/10.1038/s41586-023-06924-6.

Stefan Ropke and David Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transp. Sci.*, 40(4):455–472, 2006. doi: 10.1287/TRSC.1050.0135. URL https://doi.org/10.1287/trsc.1050.0135.

André Schidler and Stefan Szeider. Sat-boosted tabu search for coloring massive graphs. *ACM J. Exp. Algorithmics*, 28:1.5:1–1.5:19, 2023. doi: 10.1145/3603112. URL https://doi.org/10.1145/3603112.

André Schidler and Stefan Szeider. Structure-guided local improvement for maximum satisfiability. In Paul Shaw, editor, *30th International Conference on Principles and Practice of Constraint Programming, CP 2024, Girona, Spain, September 2-6, 2024*, volume 307 of *LIPIcs*, pages 26:1–26:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024a. doi: 10.4230/LIPICS.CP.2024.26. URL https://doi.org/10.4230/LIPIcs.CP.2024.26.

André Schidler and Stefan Szeider. SAT-based decision tree learning for large data sets. *J. Artif. Intell. Res.*, 80:875–918, 2024b. doi: 10.1613/JAIR.1.15956. URL https://doi.org/10.1613/jair.1.15956.

Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In Michael J. Maher and Jean-Francois Puget, editors, *Principles and Practice of Constraint Programming - CP98, 4th International Conference, Pisa, Italy, October 26-30, 1998, Proceedings*, volume 1520 of *Lecture Notes in Computer Science*, pages 417–431. Springer, 1998. doi: 10.1007/3-540-49481-2_30. URL https://doi.org/10.1007/3-540-49481-2_30.

Akash Singirikonda, Serdar Kadioglu, and Karthik Uppuluri. Text2zinc: A cross-domain dataset for modeling optimization and satisfaction problems in minizinc. *CoRR*, abs/2503.10642, 2025. doi: 10.48550/ARXIV.2503.10642. URL https://doi.org/10.48550/arXiv.2503.10642.

Hai Xia and Stefan Szeider. SAT-Based tree decomposition with iterative cascading policy selection. In Michael J. Wooldridge, Jennifer G. Dy, and Sriraam Natarajan, editors, *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2024, February 20-27, 2024, Vancouver, Canada*, pages 8191–8199. AAAI Press, 2024. doi: 10.1609/AAAI.V38I8.28659. URL https://doi.org/10.1609/aaai.v38i8.28659.

Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Satzilla: Portfolio-based algorithm selection for SAT. *J. Artif. Intell. Res.*, 32:565–606, 2008. doi: 10.1613/JAIR.2490. URL https://doi.org/10.1613/jair.2490.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=Bb4VGOWELI.