Universiteit Leiden

Master Computer Science

Algorithm Selection for SAT
Using Graph Neural Networks

Name:	Hadar Shavit
Student ID:	3092593
Date:	25/09/2023
Specialisation:	Artificial Intelligence
1st supervisor:	Marie Anastacio
2nd supervisor:	Bram M. Renting
3rd supervisor:	Prof. dr. Holger H. Hoos
2nd reader:	dr. Jan N. van Rijn

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

The Boolean Satisfiability problem (SAT) is a prominent $\mathcal{NP}$-hard problem with important practical applications. Many high-performance SAT solvers exist, but none of them dominates the others across the board; therefore, selecting the best algorithm on a per-instance basis can bring significant improvements in solving efficiency. Existing algorithm selection methods for SAT rely on hand-crafted features, but in many applications, deep-learning techniques have been demonstrated to be able to work well on raw input. Inspired by this, we aim to use a graph neural network (GNN) to improve algorithm selection for SAT. We first use graph neural networks for features-free algorithm selection. We apply neural architecture search on a rich configuration space and build our algorithm selector based on the resulting architecture. Our feature-free approach outperforms the previously known method and performs similarly to the best feature-based baselines. This demonstrates that graph neural networks can learn meaningful features of SAT instances for effective use in per-instance algorithm selection. In the second part of this thesis, we explore the usage of graph neural networks with hand-crafted features. We use a neural network that takes graph representation and hand-crafted features. We also show a method to extract the features from the graph neural network and use them with a well-known algorithm selection method. We show that our extracted features can improve the performance of the algorithm selection.

Contents

Contents **3**

1 Introduction **5**

2 Background **7**
2.1 Boolean satisfiability problem 7
2.1.1 CNF encoding 7
2.1.2 Complexity 8
2.1.3 SAT in real life 8
2.1.4 SAT solvers 8
2.1.5 Pre-processing 9
2.1.6 Performance measure 9
2.2 Graph representation of SAT formula 10
2.3 Hand-crafted features of SAT 11
2.4 Algorithm Selection 13
2.4.1 Performance metrics 14
2.4.2 Benchmarks and competitions 15
2.5 Graph neural networks 15

3 Related Work **18**
3.1 Features-based algorithm selection 18
3.2 Features free algorithm selection 19
3.3 Graph neural networks for SAT 20
3.3.1 Predicting satisfiability 20
3.3.2 Heuristics 20

4 Features-free algorithm selection **22**
4.1 Methodology 22
4.1.1 Scenario generation 22
4.1.2 Neural architecture search 23
4.2 Results 27
4.2.1 Neural architecture search 27
4.2.2 Hyperparameters importances 28
4.2.3 Comparison to other methods 29

5 Combining graph neural networks with hand-crafted features **35**

5.1 Combined neural network . . . 36
5.1.1 Neural architecture search results . . . 36
5.1.2 Hyperparameters importances . . . 37
5.2 AutoFolio with graph neural network features . . . 39
5.3 Experimental setup . . . 42
5.3.1 ASlib and CSSC Scenarios . . . 42
5.4 Results . . . 43
5.4.1 Comparison of methods on the generated scenario . . . 43
5.4.2 ASlib and CSSC scenarios . . . 48

6 Discussion, conclusions and future work 59
6.1 Discussion and reflection . . . 59
6.2 Conclusions and furture work . . . 60

Bibliography 62

A AutoGluon for Algorithm Selection 69

B Full Scatter Plots 71

Chapter 1

Introduction

The Boolean Satisfiability problem (SAT) is a well-known problem that attracts substantial research attention. The importance of this problem has two prominent reasons. First, it is the first problem that was proven to be $\mathcal{NP}$-Complete (Cook, 1971), bringing theoretical interest to it. Moreover, other problems, such as planning, hardware verification, and software verification, can be expressed as SAT. Therefore, SAT is a good starting point for solving important real-world problems and researching theoretical computer science.

Given the importance of SAT, various solvers were developed. These include tree-search-based solvers, local-search solvers, and more. As SAT is an $\mathcal{NP}$-complete problem, there is no known polynomial time solver for it. SAT solvers use heuristics to find a solution.

It was found that there is performance complementarity for SAT solvers, *i.e.*, no solver dominates over all instances. This property can be exploited by using algorithm selection, an idea first introduced by Rice (1976). In the framework, the goal is to select the best algorithm for a problem instance using the characteristics of a problem instance. Choosing the right solver for each instance can reduce the total running time and the number of timeouts. The state-of-the-art methods for algorithm selection utilise hand-crafted features of the SAT formula as input for a machine-learning model to select the best solver.

Using a deep learning approach that takes raw input to perform the prediction can outperform learning a model based on the hand-crafted features with a machine learning model.A popular example is pattern recognition tasks in computer vision. AlexNet (Krizhevsky et al., 2012) outperformed the hand-crafted features by a large margin on the ILSVRC-2012 challenge (Russakovsky et al., 2015). Following AlexNet, the state-of-the-art methods for image classification are based on deep learning, with advances in architectures and training procedures that further improve the performance.

Following the success of deep learning to process images, Loreggia et al. (2016) generated image representation of Boolean formulas using text-to-image transformation. Then, a convolutional neural network was used to predict the best solver for each instance. Their method, however, did not outperform the algorithm selection method that use hand-crafted features. A possible reason for this is that images are not permutation invariant. Therefore, the same formula with a different variable order will have a different image representation and a different prediction.

The performance of deep learning on image-based tasks inspired the usage of deep learning for other inputs, such as graphs. Graphs are a useful representation for some problems as they can represent relationships between different objects without the need to define their order (*i.e.* graphs are permutations invariant). A common application of graph neural networks is within chemistry, where molecules are represented as graphs.

SAT formulas can be represented as graphs as both are permutation invariant. This property has been used to extract features of SAT formulas. For example, (Xu et al., 2008) calculated the degree of a variables graph, where every variable has a node in the graph and an edge connects two variables that appear in the same clause. Ansótegui et al. (2019) extracted structural information about the formula using graph representation.

In recent years, graph neural networks were also applied in the context of the Boolean satisfiability problem. The first occurrence was a features-free prediction of the satisfiability of a formula from the graph representation (Selsam et al., 2018). Then, various other approaches were developed, including improvements of existing solvers (Selsam and Bjørner, 2019; Wang et al., 2021), searching for a satisfying assignment (Kurin et al., 2020) and more.

In this thesis, our aim is to improve the algorithm selection for SAT using graph neural networks. We first try a features-free approach to predict the best solver given an instance, using a graph representation of the formula. We then extract features of the formula found by the trained neural network. We combine those features with the known hand-crafted features and examine whether our new features can improve over the existing ones on various algorithm selection tasks. We find that features-free approaches can reach comparable results to existing algorithm selection methods.

Our contributions are as follows:

- We introduce a features-free approach to algorithm selection for SAT using a graph neural network.
- We introduce a method to extract features from a SAT formula using graph neural networks.
- We generate a new scenario of industrial-like SAT instances created by the Community Attachment (Giráldez-Cru and Levy, 2016) instance generator. We also compute the running times of various solvers on them, as well as hand-crafted features.
- We create algorithm selection scenarios based on special instance distributions from Configurable SAT Solver Challenge (CSSC). We then evaluate our methods on those scenarios.

In the following chapters, we describe the background of SAT, graph neural networks and algorithm selection. In Chapter 3, we describe the related work in the field. In Chapter 4, we describe our method for features-free algorithm selection using graph neural networks. In Chapter 5, we show the usage of the features extracted using graph neural networks to improve the existing hand-crafted features. Finally, in Chapter 6, we conclude our work and discuss future work.

Chapter 2

Background

In this chapter, we describe the background of the Boolean satisfiability problem and graph neural networks. In Section 2.1 we start by describing the SAT problem, CNF encoding, usage, and solvers. Following the fundamentals of SAT, we present the graph representations of it in Section 2.2 and the hand-crafted features Section 2.3. In Section 2.4 we define the algorithm selection for SAT. Finally, in Section 2.5, we describe graph neural networks and show various types of graph neural networks.

2.1 Boolean satisfiability problem

In the Boolean satisfiability problem (Karp, 1972; Franco and Martin, 2021), also known as SAT, given a Boolean formula, we are interested to know if there is an assignment for all variables which satisfies the formula.

Example 2.1.1. An example of a Boolean formula is:

$$(x_1 \lor x_2 \lor \neg x_3) \land (\neg x_1 \lor x_2 \lor x_3) \land (x_1 \lor \neg x_2 \lor x_3)$$

This formula is satisfiable since the assignment $x_1 = \text{False}, x_2 = \text{False}, x_3 = \text{False}$ satisfies the formula.

2.1.1 CNF encoding

While a Boolean formula can have any form (e.g., no special structure of the variables and clauses), encoding them in conjunctive normal form (CNF) (Prestwich, 2021; Davis and Putnam, 1960) is common. In CNF, the formula is a conjunction of clauses, and each clause is a disjunction of literals:

$$\land_{j=1}^{m}(\lor_{i=1}^{n_j} l_{ij}) \tag{2.1}$$

where l_{ij} is a literal, which is either a variable or its negation, the number of variables is n, and the number of clauses is m. Every formula can be converted to CNF. Because of these features and the readability of the CNF, it is the most common encoding read by SAT solvers.

The DIMACS CNF format (Johnson and Trick, 1996) is commonly used as an input for the SAT solvers. This is a file format that contains the CNF formula. This format is widely used by the annual SAT competitions (Balyo et al., 2022) and other available benchmarks, such as SATLIB (Hoos and Stützle, 2000). In this format, the first line describes the number of variables and the number of clauses. The following lines describe the clauses, where each is a sequence of integers, and the last integer is 0. The integers represent the literals, where the absolute value of the integer is the variable number, and the sign represents the negation.

Example 2.1.2. For the Boolean formula in Example 2.1.1, the DIMACS CNF form is:

```
p cnf 3 3
1 2 -3 0
-1 2 3 0
1 -2 3 0
```

Special cases of the CNF form are the k-CNF representations. In those representations, every clause has exactly k literals. Converting any Boolean formula to 3-CNF form is possible (Sipser, 2012).

2.1.2 Complexity

A formula with n variables has 2^n possible assignments. SAT formulas can have thousands and millions of variables, so a simple, brute-force approach that checks all possible assignments is impossible due to the high running time required. If the formula can be converted to 2-CNF form (so that every clause contains two literals), then the problem is solvable in a polynomial time (Krom, 1967). However, in other cases, the SAT problem is known as an $\mathcal{NP}$-Complete problem (Cook, 1971), so no known polynomial time algorithm can solve it. Due to the simplicity of the SAT problem, it is used as a base for many other problems. The 3-SAT problem is used to prove that many other problems are $\mathcal{NP}$-Complete (Karp, 1972). Those problems include the node cover problem (a minimal subset of nodes in a graph that cover all edges), and the clique problem (a subset of nodes in a graph that are all connected to each other).

2.1.3 SAT in real life

In addition to the theoretical interest of researching efficient solutions for the SAT problem, there are many practical applications of it. The most important industrial application is in the hardware verification field (Clarke et al., 2001; Biere et al., 2021) where hardware models, formulated as finite-state machines, are verified on properties such as illegal states that can lead to a system crash. SAT solvers can also be used for other fields, such as software verification, planning, cryptography and more (Franco and Martin, 2021). Although, in the last decades, machine learning gathered much attention from the public and researchers, SAT solvers have influenced and allowed this success by improving the quality of the hardware and software machine learning relies on (Fichte et al., 2023).

2.1.4 SAT solvers

SAT solvers can be classified into two main groups: complete and incomplete algorithms. Complete algorithms always return an answer, although they can take a long time to do so. On the other hand, incomplete solvers do not guarantee a solution but can be faster in some cases compared to complete methods.

The most common type of complete SAT solvers is based on the DPLL algorithm (Davis et al., 1962; Darwiche and Pipatsrisawat, 2021). The DPLL algorithm is a backtracking algorithm that attempts to find a satisfying assignment for the formula. The algorithm begins with an empty assignment and then tries to assign values to variables. If the assignment is not satisfying, it backtracks and attempts different values for the same variable. If all values have been tried, it backtracks to the previous variable and tries different values for it. If all values have been tried for all variables, the algorithm concludes that the formula is unsatisfiable. If the algorithm reaches an assignment that satisfies the formula, it returns that the formula is satisfiable.

A well-known improvement to the DPLL algorithm is the Conflict-Driven-Clause-Learning (CDCL) algorithm (Marques Silva and Sakallah, 1997). The CDCL algorithm is an enhancement of the DPLL algorithm that utilizes conflict analysis to find a satisfying assignment. The CDCL algorithm is the most commonly used algorithm in SAT solvers today. The conflict analysis is also referred to as the branching heuristic. The branching heuristic is a function that selects the next variable to assign a value to. The most common branching heuristic is the VSIDS (Moskewicz et al., 2001) heuristic, which assigns a score to each variable and then selects the one with the highest score. The scores are updated after each conflict, with scores for conflicting variables increasing and scores for other variables decreasing. Modern SAT solvers use the CDCL algorithm with various branching heuristics and conflict analysis improvements.

Incomplete algorithms are typically based on local search algorithms, such as WalkSAT (Selman et al., 1993). This algorithm greedily flips the truth values of variables to increase the number of satisfiable clauses. It focuses the search by selecting variables from unsatisfied clauses. However, it may become trapped in a local optimum and may not find a solution even if one exists.

2.1.5 Pre-processing

Pre-processing of the SAT formula is an essential part of the SAT-solving process (Biere et al., 2021). It is a set of techniques that are applied to the formula before it is given to the SAT solver. Using preprocessing techniques, it is possible to simplify the formula quickly, which can speed up the search done by the SAT solver. While there are many techniques for preprocessing, the first technique that greatly improved the performance was introduced by the SATELITE preprocessor (Eén and Biere, 2005). The technique is called bounded variable elimination (BVE) and is performed for each variable by adding all the resolvents (clauses that can be inferred from the clauses that contain the variable and its negation) to the formula, and then we remove all the original clauses. Doing this process repeatedly can increase the size of the formula exponentially. Therefore, the technique is employed only for variables that do not increase the number of clauses. In addition to this technique, many other techniques are being used to pre-process to get a smaller, easier-to-solve formula such as unit propagation (Davis and Putnam, 1960) and subsumption (Zhang, 2005).

2.1.6 Performance measure

The performance of SAT solvers is measured by their running time. The total running time is usually computed as Penalized Average Running Time (PAR), where a constant k is used to penalise timeouts by a factor of k:

$$\mathrm{PAR}_k(I, A) = \frac{1}{|I|} \cdot \sum_{i \in I} r_k(i, A) \tag{2.2}$$

$$r_k(i, A) = \begin{cases} t_i & t_i < T \\ T \cdot k & \text{otherwise,} \end{cases} \tag{2.3}$$

Where I is the set of instances, and t_i is the running time of the solver on instance i if the running time is less than the time limit, otherwise, it is k times the time limit. Common k values are 2, such as used by the SAT competitions (Froleyks et al., 2021) or the Sparkle SAT challenge 2018 (Luo and Hoos, 2018), and PAR$_1$0, which was used by the OASC (Lindauer et al., 2017) and the ICON challenge (Kotthoff et al., 2017).

2.2 Graph representation of SAT formula

There are many ways to express an SAT formula as a graph. Some of the SATZilla features are based on a graph representation. The most basic representation is the Literal-Clause Graph (LCG), where every literal clause has a node, and an edge connects every clause with the literals that it contains. A downside of this graph is its size, as we represent each variable by two nodes. Also, this means that each variable has two representations (one for each literal). A literal clause graph can be seen in Figure 2.1. A more compact form of this graph is the Variable-Clause Graph (VCG), which has a node for each variable and an edge between a variable and a clause if the clause contains the variable. It is possible to include the polarity of the variable in each clause by adding a value for each edge, depending on the polarity. An example for a literal clause graph can be seen in Figure 2.2. An even more compact representation is the Variables Graph (VG), which contains only variables and an edge between two variables if they appear in the same clause. The value of each edge is 1 if they both appear in the same polarity. Otherwise, it is -1. This graph is the most compact but contains no information about the clauses. An example of the variables graph can be seen in Figure 2.3.

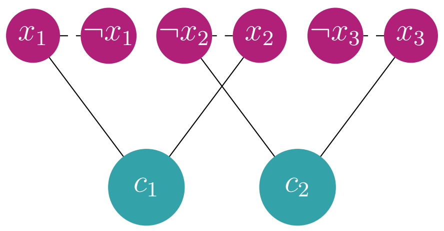


Figure 2.1: Literal-clause graph (LCG) for the formula $(x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$. The graph contains six literals (representing three variables) and two clauses. The literal nodes are coloured turquoise, and the clause nodes are coloured green. x_1 and x_2 are connected to the node c_1. x_2 and $\neg x_3$ are connected to the node c_2, x_2 in a positive form and x_3 in a negative form. Two literals of the same variables are connected with a dashed edge.

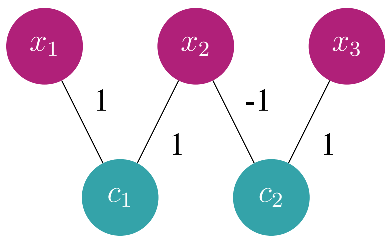


Figure 2.2: Variables-clause graph (VCG) for the formula $(x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$. The graph contains three variables and two clauses. The variable nodes are coloured green, and the clause nodes are coloured turquoise. x_1 and x_2 are connected to the node c_1 in a positive form. x_2 and x_3 are connected to the node c_2, x_2 in a positive form and x_3 in a negative form.

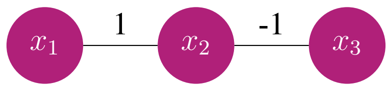


Figure 2.3: Variables graph (VG) for the formula $(x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$. The graph contains only the three variables. x_1 and x_2 are connected with a positive weight as they appear in the first clause without negation. x_2 and x_3 are connected with a negative weight as they appear in the second clause, one with a negation and not the other.

2.3 Hand-crafted features of SAT

Features of the SAT problem have been studied for a long time. The features can be used for algorithm selection (Xu et al., 2008), running time prediction (Hutter et al., 2014b) and also speed up algorithm configuration (Hutter et al., 2011). Moreover, various papers showed that it is possible to predict whether an SAT instance is satisfiable or not based on those features (Xu et al., 2012a).

The first feature we describe is the SATZilla features, which are the most commonly used features for SAT. Those features contain a few feature groups:

- Problem Size Features: These features describe the basic characteristics of the formula, such as the number of clauses and variables before and after pre-processing and the ratio of variables and clauses.
- Variable-Clause Graph Based Features: The Variable-Clause Graph (VCG) is a graph representing the SAT formula with a node for each variable and clause and a link between every variable's node and the clauses it is part of. The features are calculated by extracting the degree statistics of the variables and clause nodes.
- Variable Graph Based Features: The Variable Graph (VG) contains a node for each variable and a link between two variables if they appear in the same clause. The features contain the degree statistics of the VG. It also contains the diameter of the graph, which is the longest and shortest path between every node and every other node in the graph.
- Clause Graph features: The Clause Graph (CG) contains a vertex for each clause and a link between two clauses if they share a variable. The features contain the degree statistics of the CG. For this graph, the weighted clustering coefficient is also calculated.

- Balance Features The balance features contain statistics on the positive and negative literals in each clause, positive and negative occurrences of each variable and fractions of unary, binary and trinary clauses.

- Proximity to Horn formula A clause with at most one positive literal is known as a *Horn* clause. A formula: that has only Horn clauses the Horn formula. Horn formulae are an important subclass of SAT formulae, as those are based on logic programming. The statistics that are used as features are the fraction of Horn clauses and the number of occurrences in a Horn clause of each variable.

- DPLL Probing features: As explained in Section 2.1.4, DPLL is a common algorithm to solve the SAT problem. The extracted features are based on running the DPLL algorithm on many depths and measuring the number of unit propagations. Also, using the DPLL algorithm, it is possible to estimate the size of the search space, which is another feature.

- Linear Programming (LP): Those features are based on solving an integer linear programming (ILP) relaxed version of the formula. Given a formula that contains variables x_i and clauses C_j, the ILP is defined as follows:

$$v(x_i) = x_i \tag{2.4}$$

$$v(\neg x_i) = 1 - x_i \tag{2.5}$$

$$v(C_j) = \sum_{l \in C_j} v(l) \tag{2.6}$$

  The goal is to maximize the following objective function:

$$\sum_{j=1}^{n} v(C_j) \tag{2.7}$$

  Under the following constraints:

$$\forall C_i : \sum_{l \in C_i} v(l) \geq 1 \tag{2.8}$$

$$\forall x_i : 0 \leq v(x_i) \leq 1 \tag{2.9}$$

  Where x_i is a variable, C_j is a clause and v is the value of a vairable or a clause. In other words, if we give each variable a value between 0 and 1, the goal is to maximize the total value of all clauses under the constraint that each clause should have a positive value. Then, the extracted features are the objective function value, the fraction of the variables set to zero or one and the variable integer slack statistics. It should be noted, however, that these features might take a long time to be computed.

- Local Search Based: SAPS and GSAT are local-search SAT solvers. The features are extracted by running them many times until a local optimum cannot be escaped after some steps. Then, statistics about those runs are extracted, such as the minimum fraction of UNSAT clauses in a run, the number of steps to the best local minimum in each run, the average improvement per local-search step, the fraction of improvements due to the first local minimum and the coefficient of variation of the number of unsatisfied clauses in each local minimum.

- Clause Learning: Those features are gathered by running ZCHAFF_RAND, a DPLL-based SAT solver. The features are based on running it for two seconds and extracting the number and length of learned clauses.
- Survey Propagation: Features that are based on the variables' bias. The positive bias is the fraction of a solution of the formula where the variable appears positively. The negative bias is defined similarly. The survey is the bias of all variables. The features are based on estimations of the survey, and then computing the statistics of the confidence. The confidence is defined as

$$\text{conf} = \max(\frac{P_{\text{true}}(i)}{P_{\text{false}}(i)}, \frac{P_{\text{false}}(i)}{P_{\text{true}}(i)}) \tag{2.10}$$

  Where $P_{true}(i)$ is the probability that variable i is true, $P_{false}(i)$ is the probability that variable i is false and conf is the confidence. Similarly, there are statistics about the probability of the variable being unconstrained. Those probabilities are extracted using VARSAT (Hsu et al., 2008).
- Timing Features: describe the running times of each feature extraction phase above.

In total, there are 138 features. Some features are fast to calculate, like the problem size features. However, sometimes, the time required to calculate all features is high (more than 60 seconds).

2.4 Algorithm Selection

It has been observed that no single algorithm dominates all others on all problem instances for many high-performance computing applications, including solving $\mathcal{NP}$-Hard problems. This is also known as the performance complementarity of algorithms (Leyton-Brown et al., 2003; Luo and Hoos, 2018). This property can be exploited by choosing the best algorithm for each instance and creating a meta-solver that can utilise the strengths of all solvers.

The algorithm selection problem was introduced approximately 50 years ago by Rice (1976). However, only in the last two decades has it achieved better performance than the single best solver (Kerschke et al., 2018). It first gained attention within the SAT community with the introduction of SATZilla (Xu et al., 2008), which won numerous medals in the SAT competitions since 2007. Subsequently, many other methods for algorithm selection were developed for SAT and other problems, including machine learning problems.

In this thesis, we use the term algorithm selection for *offline* algorithm selection. This means an offline training part of the algorithm selection framework is used for unseen instances. The other type of algorithm selection is *online* algorithm selection, which is learning to select algorithms while solving new instances. We also perform the algorithm selection on a per-instance basis, where we select an algorithm for each instance separately.

Formally, per-instance algorithm selection (Rice, 1976) aims to find, for instance, set I and a set of algorithms $\mathbf{A}$, an algorithm selector $S : I \to \mathbf{A}$. S should optimise the performance with respect to a performance metric $m : \mathbf{A} \times I \to \mathbb{R}$.

The algorithm selection framework has two parts. First, we train the algorithm selector, as shown in Figure 2.4. We start from a set of problem instances and algorithms. We then extract features from the instances and measure the running times of the algorithms. We then use them to train a model (or a set of

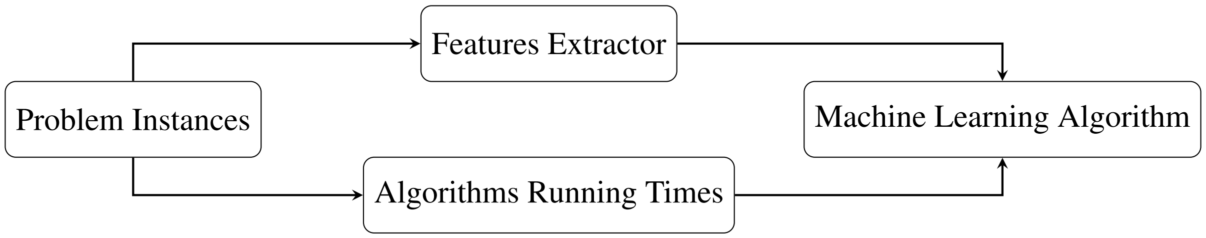


Figure 2.4: Training scheme of algorithm selection. We use a features extractor to get the features of the training instances as well as the running times of the algorithms on the training instances. Then, we train a machine learning algorithm to perform algorithm selection.

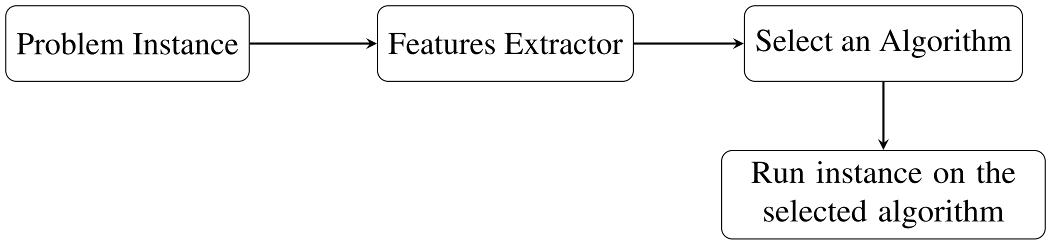


Figure 2.5: Inference of algorithm selection. We extract the features of the instance, select an algorithm using the model trained in Figure 2.4, and run the instance on the selected algorithm.

models) that predicts the best algorithm for each instance. The second part is using the trained model on new instances, as seen in Figure 2.5. At this stage, we extract features from the new instances and use the trained model to select the best algorithm for each instance and run it. Various ways to extend this framework have been proposed in the last few years. For example, Pulatov et al. (2022) showed that it is possible to extend this framework by adding algorithm features.

An approach that is related and commonly used together with algorithm selection is scheduling. In scheduling, instead of selecting a single algorithm to run, we select multiple algorithms to run and assign them order and execution time. While this approach can have several advantages over algorithm selection, as we do not rely on a single algorithm to solve the instance, it has a major drawback as it can never achieve the performance of the best algorithm. The latest algorithm selection methods combine both approaches by using a short pre-solving schedule followed by an algorithm that is being selected.

2.4.1 Performance metrics

There are many ways to measure the performance of algorithm selection (Amadini et al., 2023). Most of them rely on the definition of the Single Best Solver (SBS), which is the fastest solver on the training instances. The Virtual Best Solver (VBS), also known as the Oracle, represents the optimal meta solver that chooses the solver that works best for each instance.

While evaluating the performance of algorithm selection methods based on the running time is possible, it is missing the relation to the VBS. In this thesis, we will focus on measuring the performance using the closed gap metric, which is computed as follows:

$$\text{CG} = \frac{m_{SBS}(I) - m_S(I)}{m_{SBS}(I) - m_{VBS}(I)} \tag{2.11}$$

Where $m_A(I)$ is the performance of the solver A on instance set I and CG is the closed gap. Due to performance complementarity, the SBS and VBS are different; therefore, $m_{SBS}(I) - m_{VBS}(I) \geq 0$. For SAT, the evaluation metric is PAR$_k$. Using the closed gap metric, a score of 0 means that the algorithm selection is as good as the SBS, and a score of 1 means that the algorithm selection is as good as the VBS. Therefore, the maximum score is 1, and a negative score means the algorithm selection approach is worse than the single best solver. In practice, the score should be between 0 and 1; the higher the score, the better the algorithm selection.

2.4.2 Benchmarks and competitions

In algorithm selection, a *scenario* is a set of instances, algorithms and features. A scenario can also have feature costs, which indicate the running time required to extract the features. A scenario also contains a 10-fold cross-validation split of the instances to allow the comparison between different approaches. A commonly used collection of algorithm scenarios is the ASLib (Bischl et al., 2016) library, which contains many algorithm selection *scenarios* from various fields (SAT, QBF, MIP, Machine Learning, etc.). Those scenarios are in a unified format that is easy to use for new algorithm selection approaches. The format contains the features' computation time, feature values, and algorithm running times. This library was developed as part of the Configuration and Selection of Algorithms (COSEAL) organisation.

The Inductive Constraint Programming (ICON) Challenge on Algorithm Selection (Kotthoff et al., 2017) and the Open Algorithm Selection Challenge (OASC) (Lindauer et al., 2017) were both competitions between various algorithm selectors on various benchmarks from ASLib. Both competitions calculated the performance of the algorithm selector based on the PAR$_{10}$ metric, where they included feature computation time. They also allowed a pre-solving schedule.

2.5 Graph neural networks

Graph neural networks (Bronstein et al., 2021) are a type of neural network used to process graphs. An undirected graph is defined as:

$$G = (V, E) \tag{2.12}$$

$$V = \{1, 2, \cdots n\} \tag{2.13}$$

$$E = \{\{i, j\} \mid i, j \in V\} \tag{2.14}$$

where V is the set of nodes, n is the number of nodes. E is the set of edges, and the edge $\{i, j\}$ connects nodes i and j.

The input for graph neural networks is a graph representation of a problem. Every node is represented with a vector of features. For node i, its features vector is h_i. It is also possible to have edge features. For edge $\{i, j\}$, its features vector is $e_{i,j}$. The graph neural network is composed of a few *message-passing* layers, which propagate messages between the nodes and edges. The output can be per node (i.e., node classification), per edge (i.e., edge classification), or the whole graph (i.e., graph classification). For whole graph prediction, an aggregation mechanism is required to aggregate all the representations of the nodes and edges into a single vector that represents the whole graph.

The scheme of a graph neural network is a message-passing scheme, which can be seen in Figure 2.6. In this scheme, the network iteratively passes messages between the nodes over the edges. The messages are aggregated in the nodes, and then the node representations are updated based on the aggregated

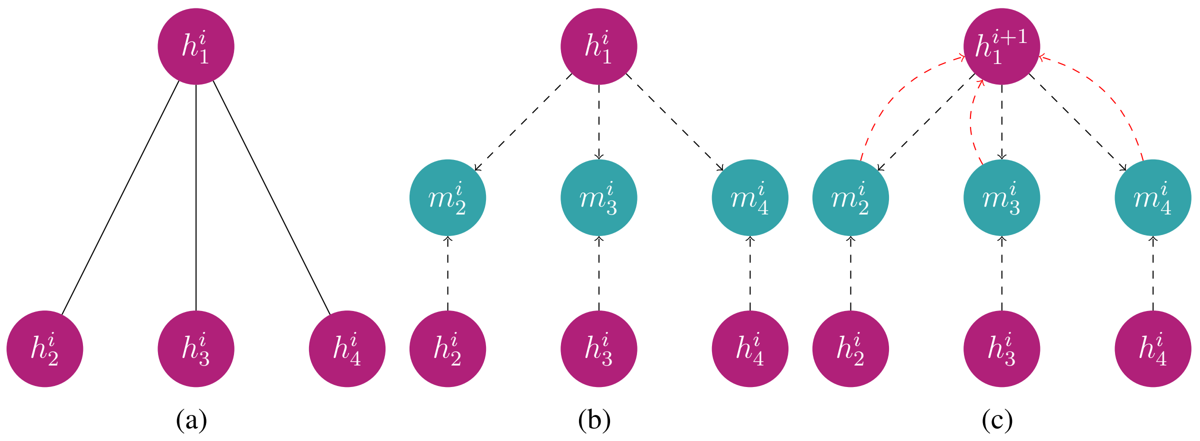


Figure 2.6: The scheme of a graph neural network. The network iteratively passes messages between the nodes and the edges. (a) Each message passing iteration starts from the representation for each node. In this diagram, we compute the representation of node 1. Each edge has a sender node (where the edge starts from) and a receiver node (where the edge ends). (b) The messages are computed using the sender and (optionally) the receiver nodes' representations. (c) The messages are aggregated to compute the new representation for each node.

messages. This process is then done for several layers, and then the final representation of the nodes, and, optionally the edges, is used for the prediction.

There are many message-passing layers, such as:

- GCN (Kipf and Welling, 2017) performs symmetric normalization over the neighbourhood node's features. This layer generalizes the convolutional layer used in computer vision to graph-structured data. The convolutional layer is defined as:

$$h_i^{l+1} = \text{Act}(\text{L}(\frac{1}{\sqrt{\text{deg}_i}\sqrt{\text{deg}_j}} \sum_{j \in N_i} e_{j,i} h_j^l)) \tag{2.15}$$

  Where h_i^l is the representation of node i in the lth message passing iteration. $e_{j,i}$ is the edge features between node j and i, deg_i is the degree of node i, Act is an activation function and L is a linear layer. The input to the linear layer is the symmetric average of the sum of the neighbourhood nodes' features, weighted using the edges features.

- GraphConv (Morris et al., 2019) is a simple graph neural network operator that similarly to the GCN layer calculates the sum of the adjacent nodes' features. However, the GraphConv layer also adds the node's features to the sum. The GraphConv layer is defined as:

$$h_i^{l+1} = \text{Act}(\text{L}_1(h_i^l) + \text{L}_2(\sum_{j \in N_i} e_{j,i} h_j^l)) \tag{2.16}$$

  Where h_i^l is the representation of the node i in the lth message passing iteration. $e_{j,i}$ is the edge between node j. L_1 and L_2 are linear layers, Act is an activation function.

- GAT (Veličković et al., 2018) – The attention mechanism computes a weight for each data point (usually by using a neural network), thus increasing the weight of some data points. This layer utilises an attention mechanism to aggregate the neighbourhood node's features.

$$h_i^{l+1} = \text{concat}_{k=1}^{K}(\text{Act}(\sum_{j \in N_i} \alpha_{ij}^k \text{L}_k(h_j^l))) \tag{2.17}$$

$$\alpha_{ik}^k = \frac{\exp(\hat{\alpha}_{ij}^k e_{ij})}{\sum_{j' \in N(i)} \exp(\hat{\alpha}_{ij'}^k e_{ij})} \tag{2.18}$$

$$\hat{\alpha}_{ij}^k = Act(\text{L}_k(h_i^l), \text{L}_k(h_j^l)) \tag{2.19}$$

Where h_i^l is the representation of node i in the lth message passing iteration α_{ij}^k is the attention coefficient. Act is an activation function and L is a linear layer, and concat is a concatenation operation.

- GIN (Xu et al., 2018) is a layer based on the Weisfeiler-Lehman Isomorphism Test (WL). This test checks whether two graphs are isomorphic (have the same structure). While it might classify two different graphs as isomorphic, it is considered an efficient solution approximation. The test is done by iteratively aggregating neighbouring nodes' representations. The WL test inspires the GIN layer. It specialises in learning graph structures. The layer is defined as follows:

$$h_i^{l+1} = \text{Act}(\text{L}_1((1+\epsilon)h_i^l + \sum_{j \in N(i)} h_j^l)) \tag{2.20}$$

Where h_i^l is the representation of node i in the lth message passing iteration, and Act is an activation function and L is a linear layer. The input to the L is the sum of the neighbourhood node's features and the node's features. The node's feature is scaled by $(1+\epsilon)$, where ϵ is a learnable parameter.

- GraphSAGE (Hamilton et al., 2017) is similar to the GCN layer. While GCN uses the neighbouring nodes' representations to compute the new representation, this layer also uses the updated node's representation. It also uses the normal mean of the messages instead of the symmetric mean. The layer is defined as follows:

$$h_i^{l+1} = \text{Act}(\text{L}(\text{concat}(h_i^l, \frac{\sum_{j \in N_i} h_j^l}{|N(i)|}))) \tag{2.21}$$

Where h_i^l is the representation of node i in the lth message passing iteration. Act is an activation function and L is a linear layer. The input to the L is the concatenation of the node's feature and the mean of the neighbourhood node's features.

Chapter 3

Related Work

Several works have studied the problem of selecting the best algorithm for a given problem instance. In Section 3.1, we present details of the algorithm selection task, commonly used approaches to solve it, and the different techniques to characterise the problem instances. In Section 3.2, we show features-free methods that do not require any feature extraction; finally, in Section 3.3, we present the most recent works that use graph neural networks in the context of the Boolean satisfiability problem.

3.1 Features-based algorithm selection

There are various approaches to implementing the algorithm selection framework:

- SATZilla 2008 (Xu et al., 2008) uses Ridge Regression to estimate the algorithms' running time. Then, the best algorithm is chosen based on the estimated running times. To further improve the performance, SATZilla uses a short pre-solving schedule that can solve most easy instances, and before the computation of the features, it estimates how much time it will take, using some basic features. It uses the backup solver if the estimated time is too long (or the actual feature extraction takes too long). Otherwise, it computes the features and chooses the best algorithm.
- 3S (also known as ASAPv1) (Kadioglu et al., 2011) uses an improved k nearest neighbours classifier (kNN). First, they use a weighted kNN, where the weight of each neighbour is the inverse of the distance. Second, they cluster the training instances. Then, given a test instance, they determine the value of k (the number of neighbours to use) based on the closest cluster. Finally, they use various mixed integer programming (MIP) based scheduling to create a pre-solving schedule based on the neighbour instances solvers' running times. For example, when using weighted kNN, they create a schedule based on the k closest instances, the solvers' (known) running times.
- SATZilla 2012 (Xu et al., 2012b) improved the previous success of SATZilla 2008 by using a more advanced machine learning algorithm, Random Forest, and also by using more features. They also used pairwise classification instead of regression. This way, each random forest classifier "votes" for the best algorithm out of two. The algorithm with the most votes is chosen.
- CSHC (Malitsky et al., 2013) starts by clustering the training instances. The clustering starts with one cluster containing all instances and then iteratively splitting clusters, such as the instances within the clusters maximally agreeing on the best solver for them. Also, two clusters can be merged in

case it reduces the cross-validation error. Given a new instance, it is assigned to the cluster to which it is the most similar.

- SNNAP (Collautti et al., 2013) combines running time prediction with the nearest neighbour approach by first using the random forest to predict the running time of each SAT solver on the new instance, and then using the predicted running times to choose the best solver using kNN, the features of each instance are the running times of the solvers.

- AutoFolio (Lindauer et al., 2015) combines many methods together. From features preprocessing methods, via the machine learning algorithm (SVM, Random Forest, kNN and more) to how the algorithm is chosen (pairwise classification, regression, etc.). To select the best method, AutoFolio uses the SMAC library. Version 2 of AutoFolio, which is implemented in Python 3, has decreased search space size by removing some machine learning algorithms and other options from the search space. Also, pairwise regression was added. Another major change is using SMAC3 (Lindauer et al., 2022) instead of SMAC2 (Hutter et al., 2011).

- ASAP (Gonard et al., 2017) uses black-box optimization (CMA-ES) to minimise the total running time of both the pre-solving schedule and the selector performance. First, a budget is identified for the pre-schedule by using the knee detection heuristic. Then, a random forest regression is built to predict the running time of each solver. The pre-solving schedule is optimised to work well with the trained performance model (*i.e.* optimise the performance of the whole selector). Finally, the performance model is re-trained using the information gathered from the pre-schedule (*i.e.*, whether each solver solved the instance within the assigned time budget).

- AS - ASL (Malone et al., 2017) uses the success of AutoML frameworks, more specifically, AutoSklearn. These frameworks get a classification or regression task and produce a machine-learning model automatically by using multiple machine-learning algorithms, hyperparameters and clustering approaches. This way, it saves time and human effort and has good performance. The AS-ASL approach uses these advantages to create an AutoML-based algorithm selection approach using multi-class classification.

- Pairwise regression (Kerschke et al., 2018) predicts the difference between the running times of two algorithms on a given instance. Then, each solver's sum of the differences is calculated, and the solver with the highest sum is chosen.

- SUNNY (Liu et al., 2022) uses a similar nearest neighbours approach like 3S by selecting k instances that are closest to the new instance. It then creates a schedule based on the number of instances solved by each solver, ordering them by the average running time.

3.2 Features free algorithm selection

Previously, features-free algorithm selection for SAT was explored by Loreggia et al. (2016) by using a convolutional neural network. In order to transform the formula into an image, the authors used the DIMACS CNF file and transformed it into a black-and-white image using the ASCII values of each character. This results in N pixels. To reach a square shape, they reshaped the vector to a matrix with size $\sqrt{N} \times \sqrt{N}$ and rescaled the square to 128×128. For prediction, the authors used a binary classification of the solvers based on whether the solver could solve the instance given the time limit. The proposed method performed better than the SBS method, but significantly worse than hand-crafted features.

Graves-CPA (Leeson and Dwyer, 2022) uses a graph neural network to perform algorithm selection of software verifiers. The graph representation of is based on the abstract syntax tree (AST) of the problem. Graves utilises a few graph attention message passing layers, followed by a jumping knowledge layer that concatenates the outputs of all the layers. Then, there is an attention mechanism to aggregate the representations of all the nodes, followed by an MLP to perform the prediction. Graves-CPA was applied only to software verification problems.

Seiler et al. (2020) used image representation of travelling salesperson problem to perform algorithm selection. The image representation was obtained by using point clouds, minimum spanning trees and k-nearest neighbour graphs. The authors used a convolutional neural network to predict the best solver. However, the results of the method were not better than the hand-crafted features.

3.3 Graph neural networks for SAT

In the last few years, a few attempts have been made to solve the Boolean satisfiability problem using deep learning. Those papers use a graph representation of the Boolean formula and a graph neural network to either predict satisfiability or enhance existing CDCL or local search solvers.

3.3.1 Predicting satisfiability

Predicting the satisfiability of a Boolean formula has been explored in the past. Using the SATZilla features, it is possible to predict the satisfiability of a formula. Using graph neural networks, some methods tried to remove the need for hand-crafted features. For example, NeuroSAT (Selsam et al., 2018) used a literal-clause graph representation of the formula and a graph neural network to predict the satisfiability of the formula. They showed that the network could learn to predict the satisfiability of the formula with an accuracy of 85% on random formulas, and out of the formulas that were predicted as satisfiable, they could decode a satisfying assignment in 70% of the cases. A major drawback of their method is that it was trained and tested on very small formulas with up to 40 variables (SAT competition formulas can have millions of variables). Interestingly, the neural network that was trained to predict satisfiability can also to find a satisfying assignment in the majority of the cases that the formula was satisfiable. Other papers (Han, 2020; Li and Si, 2022) proposed more advanced architectures and various changes to the training scheme to improve performance. However, all of these methods could only solve small formulas (less than 100 variables) and have generalisation problems to out-of-distribution formulas.

3.3.2 Heuristics

In addition to predicting the satisfiability of the formula, another type of graph neural network can predict a heuristic to enhance existing DPLL solvers. These solvers choose the next assignment to check based on various heuristics, as explained in Section 2.1.4.

First, NeuroCore (Selsam and Bjørner, 2019) used a similar architecture to NeuroSAT, with MLPs instead of LSTMs to replace the activity score (heuristic) of Glucose and MiniSAT. It performs online inference of the graph neural network model combined with the status of the search process to calculate the heuristic. This resulted in an increase in the number of solved formulas by 10% and a decrease in the number of timeouts by 20%.

Later, NeuroComb (Wang et al., 2021) was proposed with a different architecture to compute heuristic values of important clauses and variables. NeuroComb computes the *static values* before the solving process. It then alternates between using the heuristics of CDCL solvers (referred to as *dynamic values* as they change throughout the search process) and the computed static values to choose the next assignment to explore.

The NeuroComb architecture comprises a few *dense* blocks, each containing a few message-passing layers. In each block, the output of each message passing layer is concatenated with its input to allow a better flow of information through the graph. This idea is similar to the jumping knowledge that is used in the graph neural network.

The message passing layers layer used in NeuroComb is defined as:

$$h_i^{l+1} = \text{MLP}_1(\text{concat}(h_i^l, \text{mean}_{j \in N(i)} \text{MLP}_2(h_j^l e_{ij}))) \tag{3.1}$$

where h_i^l is the hidden state of node i in layer l, $N(i)$ is the set of neighbours of node i, e_{ij} is the edge between nodes i and j, and MLP_1 and MLP_2 are MLPs. This layer is similar to GCN, with a few changes. First, it computes the messages using an MLP on the neighbour nodes' representation, then it computes the mean of all the messages and concatenates it with the node's representation. Finally, an MLP is applied (and optionally, an activation function). This layer calculates the mean of the messages of the neighbouring nodes, concatenating with the receiver node's representation. Based on this, the new representation is calculated.

Chapter 4

Features-free algorithm selection

In this chapter, we explore the usage of graph neural networks for features-free algorithm selection. In Section 4.1 we describe the dataset we used to train the graph neural network and describe the neural architecture search we performed to get a good performing architecture. In Section 4.2 we show the results of the neural architecture search and compare it to the SATZilla features.

4.1 Methodology

4.1.1 Scenario generation

Preliminary experiments indicated that using graph neural networks with various graph representations of SAT formulas from the 2011 and 2020 SAT competitions often exceeds the memory limit of the GPU, which can mainly be attributed to the formula size. While it is possible to process larger graphs using sampling methods (Hamilton et al., 2017), it remains outside the scope of this thesis. In addition, the number of available formulas per SAT scenario is usually less than 1000.

We generate a larger set of new, smaller formulas that fit into the GPU memory. To generate a new dataset of SAT formulas small enough for the GPU memory yet hard enough to be solved by the solvers, we use the Community Attachment industrial-like SAT formula generator. This generator generates formulas using the *modularity* of the graph of the variables. A highly modular graph is a graph that has a community structure, *i.e.*, most of the variables are in clauses with other variables from the same community. This means there is a partition of the nodes to communities (groups of nodes), where most edges connect nodes from the same community. The modularity of the graph is measured using Q, which means the fraction of edges connecting nodes in the same community with respect to a random graph with the same number of nodes and degrees.

We used the Community Attachment SAT generator to create 3000 3-SAT formulas. Next, we split the instances into three sets: training, validation, and testing sets containing 2100, 450 and 450 instances, respectively. Each formula was generated by sampling a random configuration of hyperparameters from the ranges described in Table 4.1.

After generating the formulas, we ran seven SAT solvers on them among the best performing of past SAT competitions:

- MiniSAT (Eén and Sörensson, 2004) is a simple CDCL solver that is purposed for easy understanding and extension. It uses the VSIDS heuristic

Table 4.1: Hyperparameters of the Community Attachment SAT generator. The hyperparameters are sampled uniformly for each instance.

Hyperparameter	**Range**
# of variables	$[800, 1600]$
Clause-to-variable ratio	$[4.1, 4.3]$
Modularity	$[0.5, 0.9]$
# of communities	$[30, 50]$

- CryptoMiniSAT (Soos et al., 2009) is a MiniSAT base solver that is optimized to solve cryptographic problems. The optimization includes a special XOR clause handling common in cryptographic scenarios.
- Spear (Babic and Hutter, 2007) is a highly configurable DPLL-based SAT solver. Its heuristic is based on the Boolean constraint propagation (BCP) with B-cubing.
- glucose (Audemard and Simon, 2009) is based on MiniSAT with the Literal Block Distance (LBD) heuristic.
- SparrowToRiss (Balint and Manthey, 2014) starts by running the local-search solver Sparrow (Balint et al., 2011) for a limited time (the default value is 150 seconds), followed by the CDCL solver Riss (Manthey, 2010).
- Kissat (Biere et al., 2020) is the current state-of-the-art SAT solver that won 2020, 2021 and 2022 SAT competitions. Kissat is a reimplementation of CaDiCal, a CDCL-based solver that interleaves between two queues, one based on the VSIDS heuristic and the other based on a smoother version of this heuristic. Moreover, CaDiCal includes occasional local searches. Kissat is a reimplementation of CaDiCal, with more efficient data structures and algorithms.

We used AClib (Hutter et al., 2014a) to run the solvers, with the addition of Kissat, which AClib does not contain. The timeout for each solver is 5000 seconds. We ran the solvers using a compute cluster, and each solver ran on one CPU core. The cluster has 34 nodes, and each node has two Intel(R) Xeon(R) CPU E5-2683 v4 2.1GHz with 16 cores each and 94 GB of memory.

Features computation

After creating the dataset of the SAT formulas and gathering the running times of the solvers, we computed the features of the formulas. We used the SATZilla 2012 features and the code provided by Xu et al. (2012b). The features are described in Section 2.3.

4.1.2 Neural architecture search

We perform a neural architecture search (NAS) to find a graph neural network architecture suitable for the task of performing algorithm selection of SAT solvers. We create a search space containing the input graph, the neural network architecture and the prediction type.

Input graph

We use either a variables clause graph or a variables graph for the input graph. A detailed description of the graphs can be found in Section 3.3. In addition, there is an option to add a *connecting node* to the variable clause graph – an additional node that has a connection to all the nodes representing a clause. This common node can be used to improve the graph's connectivity, thus allowing information to pass more easily through the graph, as it reduces the maximal distance between nodes to 4 (e.g., two variable nodes that do not have a common clause). It requires only a few message-passing layers for a message from one node to arrive in all other nodes. The graph can represent the raw formula or the formula after SATELITE preprocessing (Eén and Biere, 2005). As most SAT solvers use a preprocessor before the actual solving, this can allow us to extract more relevant features. A similar step was done by SATZilla (Xu et al., 2008) during the features computation stage. We also allow the option to not use the edge features, as in preliminary experiments, we found that, in some cases, it can result in better performance.

Neural network architecture

The first hyperparameter of the graph neural network is the hidden size of the message-passing layers, which is preserved throughout the whole architecture. The neural network architecture comprises several *blocks* (between 1 and 10 blocks). All the blocks have the same number of message-passing layers (between 1 and 5). At the end of each block, there can be a jumping knowledge layer (Xu et al., 2018). If there is a jumping knowledge layer, then there is a pooling layer – another message-passing layer that takes the concatenated features from the jumping knowledge layer and processes it to the original hidden layer size. This block type is inspired by NeuroComb (Wang et al., 2021) and Graves (Leeson and Dwyer, 2022). The jumping knowledge layer can help the neural network learn more complex graph structures, as it creates a new representation that contains the graph representation from various ranges. This can improve the structure learning of the graph.

The message passing layer can be either GCN (Kipf and Welling, 2017), SAGE (Hamilton et al., 2017), GraphConv (Veličković et al., 2018), GIN (Xu et al., 2018), NeuroComb-like Wang et al. (2021) or GAT (Veličković et al., 2018). The search space includes GIN, which can learn graph structures better than other message-passing layers (Xu et al., 2018). The search space also includes the NeuroComb-like layer, a special layer invented for a variables-clause graph representation of SAT formulas.

After the main graph processing stage (the message-passing blocks), there is an option to add a jumping knowledge layer that aggregates all the outputs from all the blocks. Moreover, each message passing layer can be surrounded by a skip connection, similar to the skip connection that was used by Li et al. (2021). While the jumping knowledge layer concatenates the outputs of two (or more) layers, which is similar to the DenseNet architecture (Huang et al., 2017), the skip connection adds them, like in ResNet (He et al., 2016). Before each message passing layer, there is an activation function which can be one of the following: ReLU (Nair and Hinton, 2010), LeakyReLU (Maas et al., 2013), ELU (Clevert et al., 2015), GELU (Hendrycks and Gimpel, 2016), CELU (Barron, 2017). There is also an option to include layer normalization before each message passing layer. Finally, the number of layers in the MLP can be selected between 1 and 5.

Aggregation

As algorithm selection is a graph classification task, we must aggregate all the node representations into a single prediction. To do that, we have a few options:

Table 4.2: Hyperparameters of the graph neural network architecture

Hyperparameter	Values
Hidden Layer Size	$[16, 256]$
Message Passing Layer	{GCN, SAGE, GraphConv, GAT, GIN, NeuroComb}
Activation Function	{ReLU, LeakyReLU, ELU, GELU, CELU}
#Blocks	[1, 10]
#Message Passing Layers per Block	[1, 5]
Skip Connection after Message Passing	{True, False}
Jumping Knowledge in Each Block	{True, False}
Final Jumping Knowledge	{True, False}
#MLP Layers	[1, 5]

- Mean: the mean of all the node representations.
- Max: the maximum of all the node representations.
- Sum: the sum of all the node representations.
- Attention: using attention mechanism to aggregate the node representations.
- Softmax: uses the softmax function to perform the aggregation, similar to the aggregation operator used by Li et al. (2021).
- Common Node: if a variables clause graph with a connecting node is used, we can take this node's representation as the graph representation.
- Per Node Prediction: using MLP on each node separately to get the prediction and then aggregating the results using mean aggregation.
- Memory-Based Pooling (Khasahmadi et al., 2019): a memory-based pool is an attention-based layer that learns to coarsen the graph based on soft cluster assignments from different heads of the attention module. We use between 1 to 5 layers, with 5 to 100 attention heads per layer and between 4 and 512 clusters per layer (except the final layer, which has one cluster). There is also an option to add an activation between the memory-based pooling layers.

Prediction type

Inspired by AutoFolio (Lindauer et al., 2015), we allow the search process to select between a few prediction types:

- Solving probability: each solver has a binary output, 1 if it can solve the instance, 0 otherwise. The selected solver is the solver with the highest probability of solving the instance. The prediction type was used before by Loreggia et al. (2016).
- Multi-class classification: this prediction type treats algorithm selection as a simple multi-class classification problem, where the output classes are the solvers, and the correct label is the solver

with the lowest running time. This prediction type can be weighted, where the weight is the standard deviation of the running time of the solvers. This prediction type was used before by Lindauer et al. (2015).

- Pairwise classification: in a pairwise classification problem, each pair of solvers has an output, and the correct label is the pair of solvers with the lowest running time. This prediction type was used before by Lindauer et al. (2015).
- Running time regression: this prediction type treats each solver independently by predicting the log running time of each solver similarly to SATZilla 2007 (Xu et al., 2008). Then, the selected solver is the solver with the lowest predicted running time.

Each prediction type can use instance weights, which gives higher importance to instances with higher standard deviations between the running times of the solvers.

Training Procedure

We also search for the optimal training procedure. We choose between the AdamW (Loshchilov and Hutter, 2018) or RAdam (Lindauer et al., 2019) optimisers, their learning rate and weight decay. We use a batch size of 4, with gradient accumulation. The reason for this batch size is to balance between the size of the neural network that can be trained on the GPU and the training speed (bigger batches allow faster training). The number of batches per gradient step is also a hyperparameter and can be between 1 to 64 (so the number of training examples per gradient step is between 4 and 256). There is also an option to set dropout (Srivastava et al., 2014) after each message-passing layer. We perform model selection based on the running time of the trained selectors on the validation set and take the model with the lowest value.

Search process

We employed SMAC3 (Lindauer et al., 2022) version 2.0, utilising the multi-fidelity facade to find good-performing architecture within our search space. This multi-fidelity facade is similar to the BOHB algorithm (Falkner et al., 2018). BOHB employs varying budgets (in terms of the number of epochs) for each training process to reduce computational costs. It also generates new configurations through Bayesian optimisation, implemented as random forests in the SMAC framework.

We run SMAC with four parallel processes, each training a different configuration on a separate GPU. Each configuration was trained three times with different random seeds. During each run, the network was trained for a variable number of epochs, ranging from 10 to 100, depending on the budget allocated by SMAC. For each configuration, we calculate the average running times of the models trained using the three random seeds. The goal is to minimise the average running time.

To prevent computationally expensive configurations, we constrained each configuration to train for a maximum of one hour, irrespective of the assigned budget. We run SMAC for six days, conducting three independent runs. For each of the three architectures found, we train them for 100 epochs and select the one with the lowest average running time on the validation set, as the final model found by the NAS.

The entire NAS process was repeated three times, optimising for three different metrics: PAR_1, PAR_2, and PAR_{10}.

4.2 Results

In this section, we describe the results of the experiments for features-free algorithm selection. We start by showing the neural network architectures found (Section 4.2.1), then in Section 4.2.2 we show the importance of the hyperparameters during the neural architecture search process. Finally, in Section 4.2.3 we compare our graph neural network to other algorithm selection methods on three performance metrics.

4.2.1 Neural architecture search

The architectures found in the neural architecture search processes are described in Table 4.3. We can see that all architectures use the compact variables graph that allows more efficient message propagation through the graph. Moreover, PAR_1 uses the formulas with preprocessing while the others do not. This can indicate that the preprocessing changes the structure of the formula in a way that is more important for timeouts. We can also see that all architectures are relatively shallow, with less than 10 message-passing layers. The found architectures also use dropout with low learning rate, which can indicate that the data is noisy and the network needs to be regularized. The noise is due to the high variance of the solvers running times.

Table 4.3: Hyperparameter values for the architecture found by the NAS processes.

Hyperparameter	PAR_1	PAR_2	PAR_{10}
Graph Type	Variable Graph	Variable Graph	Variable Graph
SATELITE Preprocessing	True	False	False
Edge Weights	False	True	True
Hidden Layer Size	48	96	128
Message Passing Layer	NeuroComb	GraphConv	GraphConv
Activation Function	GELU	GELU	LeakyReLU
#Blocks	2	1	1
#Message Passing Layers per Block	5	5	3
Jumping Knowledge in Each Block	True	True	False
Final Jumping Knowledge	True	True	True
Skip Connection	False	False	True
Layer Normalization	True	False	False
Dropout	0.111	0.414	0.456
Aggregation	Memory Pooling	Per Node Prediction	Memory Pooling
#Memory Pooling Layers	3	N A	4
#Attention Heads per Layer	$[30, 20, 75]$	N A	$[30, 30, 30, 35]$
#Clusters per Layer	$[92, 132, 444]$	N A	$[40, 56, 8, 48]$
#MLP Layers	2	3	3
Prediction Type	Pairwise Class.	Pairwise Class.	Solve Proba.
Optimiser	RAdam	RAdam	RAdam
Learning Rate	$21e^{-3}$	$161e^{-4}$	$15e^{-4}$
Weight Decay	0.014	$1.72e^{-5}$	$1.18e^{-5}$
Outer Batch Size	28	39	52

4.2.2 Hyperparameters importances

To evaluate the importance of the many components used during the neural architecture search process, we used DeepCAVE (Sass et al., 2022), a tool to visualise and analyse AutoML processes. Specifically, we used DeepCAVE to visualise the importance of the different components of the neural architecture search process. We use the local hyperparameter importance metric (LPI) (Biedenkapp et al., 2019) to quantify the importance. The results are presented in Figures 4.1, 4.2, 4.3. We can see that when optimising for a different objective, other hyperparameters are more important, although some hyperparameters are important for all objectives, like the number of blocks, graph type and the type of message passing layer. We can also see the edge weights are important for PAR_1 but not for PAR_2 and PAR_{10}.

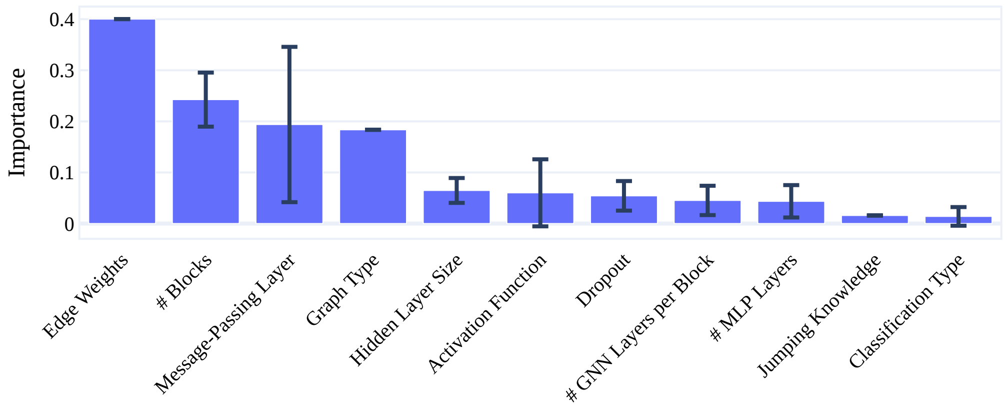


Figure 4.1: Local hyperparameter importance for the PAR_1 metric. Edge weights are the most important hyperparameter, followed by the number of blocks, the type of message passing layer and the graph type. The rest of the hyperparameters are less important.

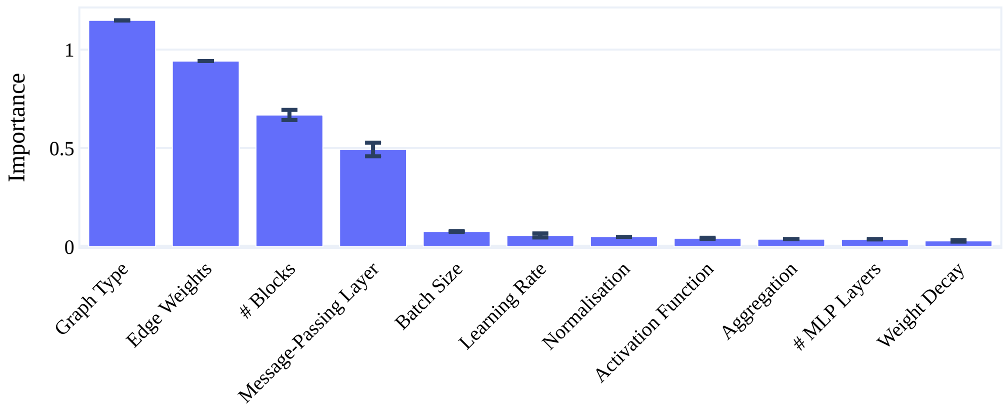


Figure 4.2: Local hyperparameter importance for the PAR_2 metric. Graph type is the most important hyperparameter, followed by the edge weights, number of blocks and the type of message-passing layer. The rest of the hyperparameters are less important.

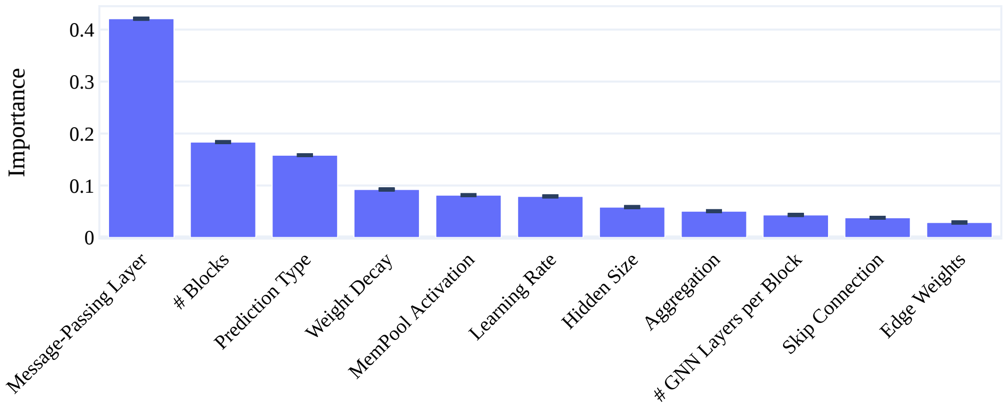


Figure 4.3: Local hyperparameter importance for the PAR_{10} metric. The type of message-passing layer is the most important hyperparameter. It is followed by the number of blocks and prediction type. The rest of the hyperparameters are less important.

4.2.3 Comparison to other methods

We compare our graph neural network approach with the features-free method proposed by Loreggia et al. (2016). We re-implement this method using Pytorch, and used the RAdam optimizer instead of SGD with momentum, as RAdam (Lindauer et al., 2019) is a modern optimizer that outperforms SGD on various benchmarks. We tune the learning rate of the optimizer by running it on a grid of learning rates: $[1-10]e^{-[2-6]}$. We also compare our method to AutoFolio (Lindauer et al., 2015) that uses the SATZilla hand-crafted features. We run each method using 13 different random seeds on the generated scenario. We run each method for PAR_1, PAR_2, and PAR_{10} metrics. We calculated statistical significance and established a ranking using the Autorank library (Herbold, 2020). The statistical significance is calculated between the closed gap metrics of the methods over the different random seeds.

In Figure 4.4a, we can see the closed gap measured using the PAR_1 metric over 13 random seeds. GNN is our method, CNF2IMG is the features-free convolutional neural network of (Loreggia et al., 2016). We can see that AutoFolio has the highest closed gap values, followed by GNN. CNF2IMG has the worst performance, with all values below both AutoFolio and GNN. Figure 4.4b shows each method's number of solved instances. We can see that GNN has the lowest number of solved instances, followed by CNF2IMG and then AutoFolio. Both GNN and CNF2IMG have the same highest value of solved instances, although AutoFolio has a higher median and lowest value. This suggests that our graph neural network method is strong on instances with low running times, but is worse than AutoFolio in reducing timeouts. This finding is confirmed by Figure 4.4c, where we can see a scatter plot of the mean running time of each instance. GNN is superior in instances with lower running time. However, when the instance running time is bigger than 10 seconds, it is worse than AutoFolio. In Figure 4.4d we can see a scatter plot of the mean running times of the instances using GNN and CNF2IMG. We can see for most instances, GNN is better.

We checked the statistical significance of the closed gap using AutoRank with a significance of 0.05. The Friedman test found that the performance differences of the methods are statistically significant. The Nemenyi test then found that there is no statistical significance between AutoFolio and GNN, and those two methods are significantly better than CNF2IMG. The rankings and significance can be seen in Figure 4.4e.

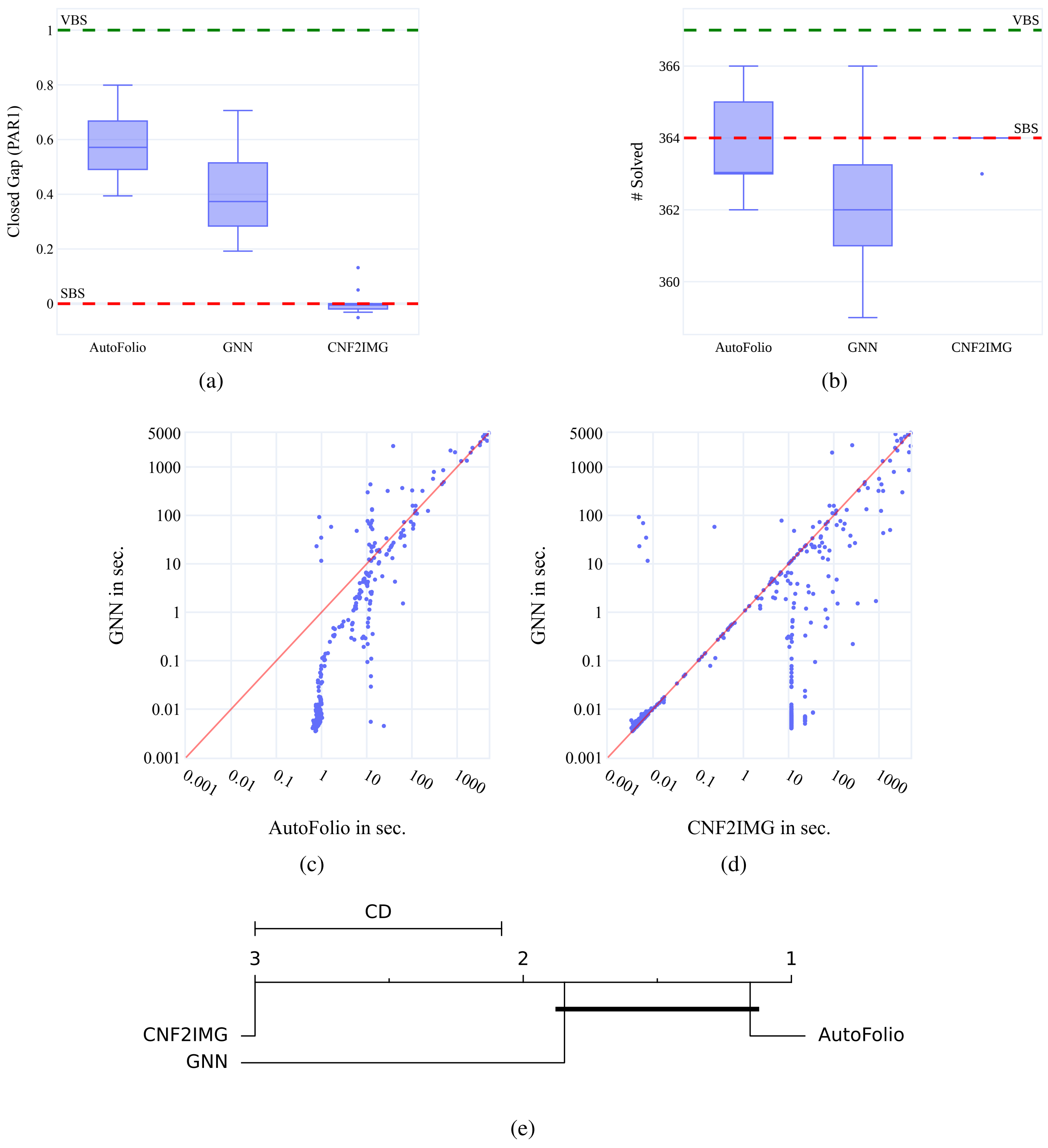


Figure 4.4: Results of PAR_1 experiments. (a) Closed gap of the methods. (b) Number of solved instances. (c) Scatter plot of the mean running times of GNN and AutoFolio. (d) Scatter plot of the mean running times of GNN and CNF2IMG. (e) Critical Difference diagram of the rankings of the three methods based on PAR_1 scores.

In Figure 4.5 we can see the results for the PAR_2 metric. GNN has the highest median value and is the only method that has only positive closed gap values (*i.e.* always better than the single best solver). AutoFolio's maximal value is the highest. CNF2IMG performs the worst. Figure 4.5b shows the number of solved instances per method. AutoFolio has the highest mean and maximal values. GNN and CNF2IMG has the same median, although GNN sometimes can solve more instances. This suggests that similarly to PAR_1, the graph neural network has better running times on instances that do not have time out. In Figure 4.5c we can see a scatter plot of the mean running times of the instances when using both GNN and AutoFolio. We can see that our graph neural network is better on all instances with short running time, while AutoFolio is better in reducing timeouts. Figure 4.5d shows the mean running times of GNN and CNF2IMG. We can see that GNN is better on most instances.

We checked the statistical significance of the closed gap using AutoRank with a significance of 0.05. The Friedman test found that the performance differences of the methods are statistically significant. The Nemenyi test then found that there is no statistical significance between AutoFolio and GNN, and those two methods are significantly better than CNF2IMG. The rankings and significance can be seen in Figure 4.5e.

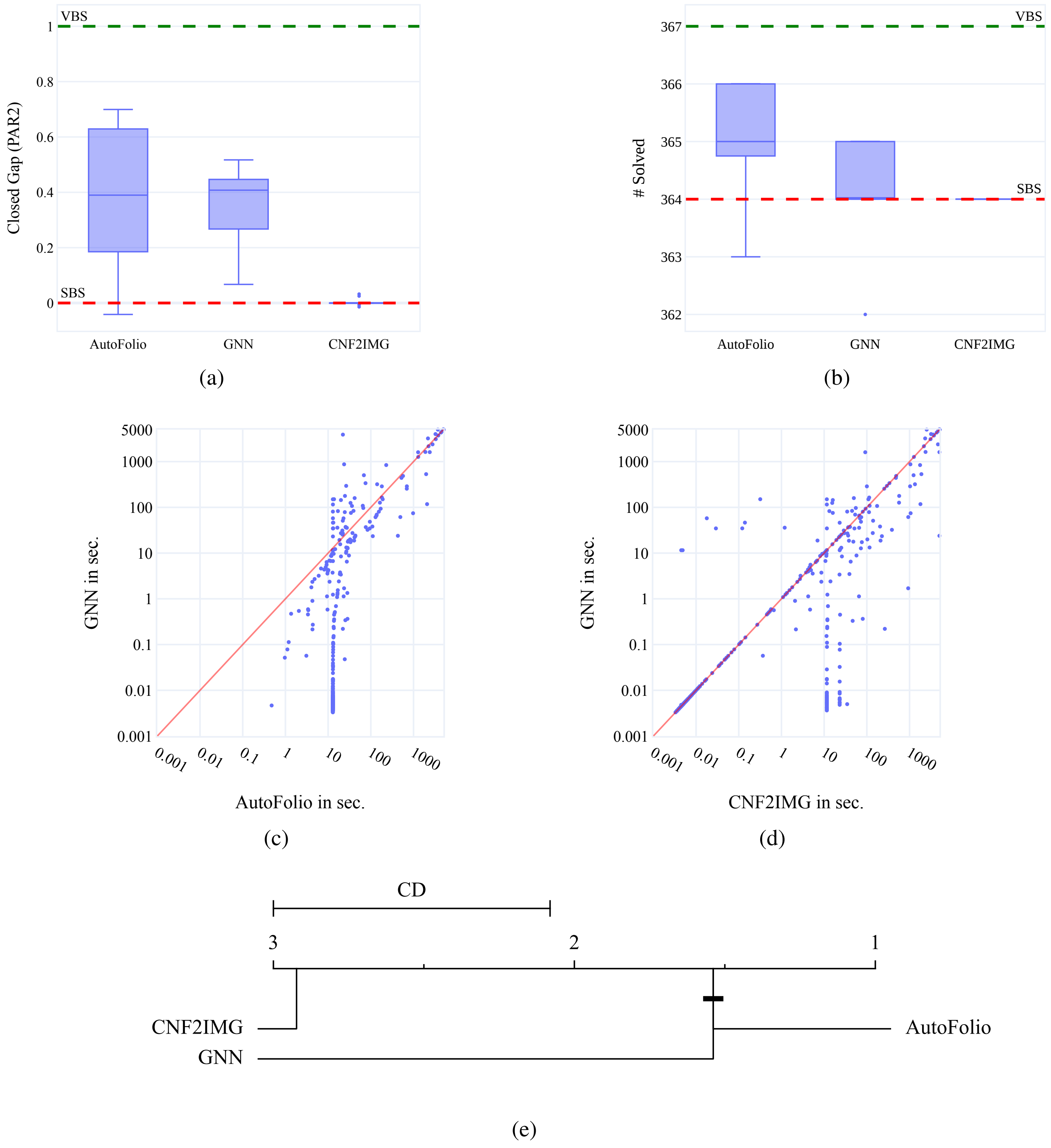
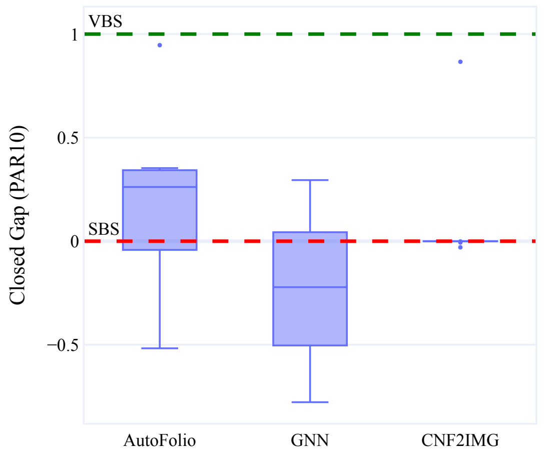


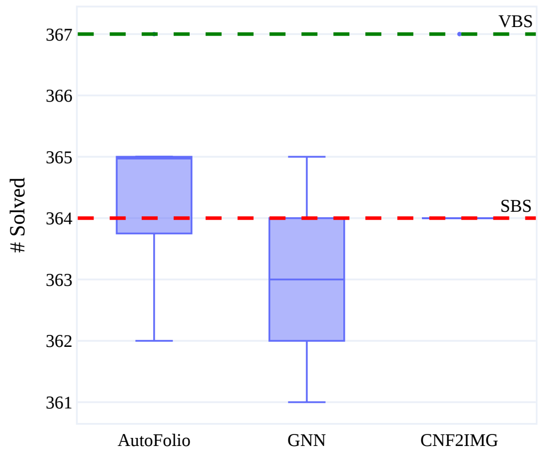
Figure 4.5: Results of PAR$_2$ experiments. (a) Closed gap of the methods. (b) Number of solved instances. (c) Scatter plot of the mean running times of GNN and AutoFolio. (d) Scatter plot of the mean running times of GNN and CNF2IMG. (e) Critical Difference diagram of the rankings of the three methods based on PAR$_2$ scores.

The closed gap results for PAR$_{10}$ are shown in Figure 4.6a, where we can see that AutoFolio has the highest median value while our graph neural network has the worst median performance, with worse performance than the single best solver. Figure 4.6b shows that AutoFolio has the highest number of solved instances among all methods, while GNN has the worst. In Figure 4.6c we can see that similarly to other PAR scores, GNN is better than AutoFolio on instances with short running times. On PAR_{10} the penalty for timeouts is the highest, hence the worse performance of GNN. In Figure 4.6d we can see that on many instances, CNF2IMG is better than GNN, contrary to the previous experiments.

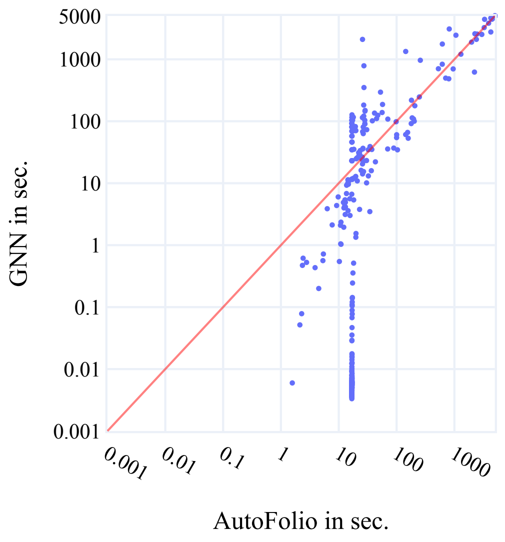
The Friedman test found no statistical significance between the performance differences of the methods for PAR_{10}.



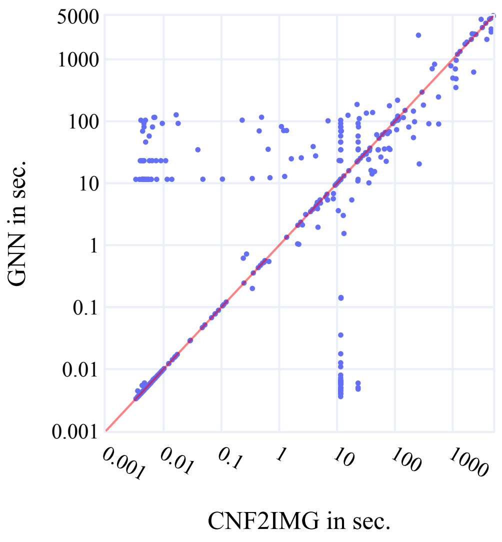
(a)



(b)



(c)



(d)

Figure 4.6: Results of PAR_{10} experiments. (a) Closed gap of the methods. (b) Number of solved instances. (c) Scatter plot of the mean running times of GNN and AutoFolio. (d) Scatter plot of the mean running times of GNN and CNF2IMG.

Overall, we can see that the graph neural network has strong performance in reducing the running times of instances with relatively short running times, while it is unable to significantly reduce timeouts.

This is why on PAR_1 and PAR_2 it is better than CNF2IMG, and close to AutoFolio, while on PAR_{10} it is the worst method.

Chapter 5

Combining graph neural networks with hand-crafted features

Following the features-free algorithm selection, we check whether using both the SATZilla features, and the features extracted using a graph neural network can improve the performance of algorithm selection methods. In this chapter, we start with a neural network approach that uses both hand-crafted features and a graph neural network. Then, we extract the features from the graph neural network to use with AutoFolio.

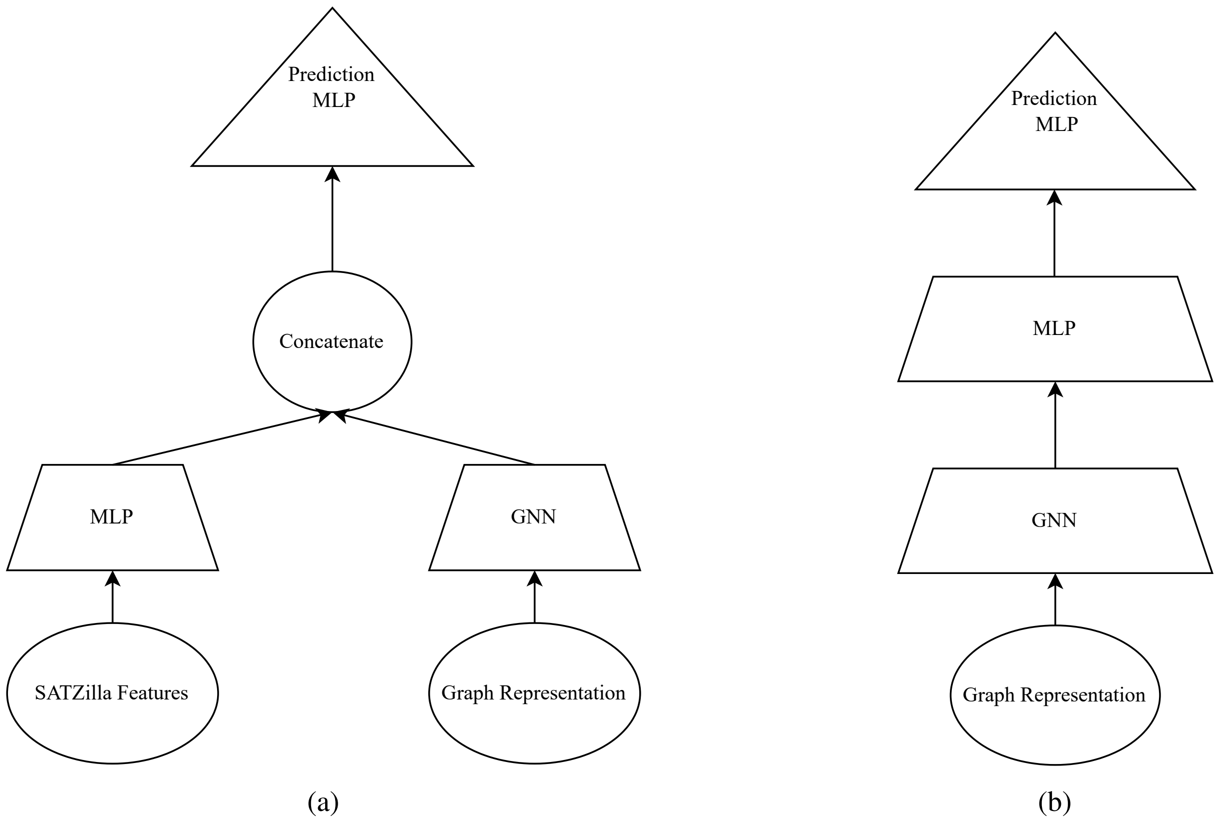


Figure 5.1: (a) Overview of the architecture for combined prediction. Both SATZilla features and the graph are processed, and the extracted features are then combined for prediction. (b) Overview of the downsampling MLP. The graph neural network extracts features that are then downsampled using MLP.

5.1 Combined neural network

Our first method is a neural network that uses both the SATZilla features and the graph representation. We combine the SATZilla features with the graph neural network by processing the SATZilla features using two layers of MLP with 16 features in each layer, similar to the architecture used by Eggensperger et al. (2018). We then append the graph-neural-network features (*i.e.*, the features after the message-passing part of the network), with the processed SATZilla features and use an MLP for the prediction. An overview of the architecture can be seen in Figure 5.1a. We run neural architecture search similarly to Chapter 4 to find an architecture for this purpose. We run the neural architecture search on the generated scenario with three repetitions per PAR score. We run it for PAR_1, PAR_2, PAR_{10}.

5.1.1 Neural architecture search results

The architectures found in the neural architecture search processes can be seen in Table 5.1. We can see that all architectures use the compact variables graph that allows more efficient message propagation through the graph. Moreover, PAR_1 uses the formulas with preprocessing while the others do not, this can indicate that the preprocessing changes the structure of the formula in a way that is more important for timeouts. We can also see that all architectures are relatively shallow with less than 10 message-passing layers. The found architectures also use dropout with low learning rate, which can indicate that the data is noisy and the network needs to be regularized. The noise is due to the high variance of the solvers running times.

Table 5.1: Hyperparameter values for the architecture found by the NAS processes.

Hyperparameter	PAR_1	PAR_2	PAR_{10}
Graph Type	Variable Graph	Variable Graph	Variable Graph
SATELITE Preprocessing	True	True	True
Edge Weights	False	True	False
Hidden Layer Size	112	48	208
Message Passing Layer	GraphSAGE	GraphSAGE	GraphSAGE
Acitvation Function	GELU	ELU	ReLU
#Blocks	10	8	3
#Message Passing Layers per Block	5	3	2
Jumping Knowledge in Each Block	False	True	True
Final Jumping Knowledge	True	True	False
Skip Connection	True	False	True
Layer Normalization	True	True	True
Dropout	0.367	0.078	0.476
Aggregation	Attention pooling	Per Node Prediction	Memory Pooling
#Memory Pooling Layers	N A	N A	4
#Attention Heads per Layer	N A	N A	$[65, 30, 50, 50]$
#Clusters per Layer	N A	N A	$[500, 96, 100, 4]$
#MLP Layers	2	4	2
Prediction Type	Pairwise Class.	Pairwise Class.	Pairwise Class.
Optimiser	AdamW	AdamW	AdamW
Learning Rate	$7.68e^{-5}$	$3.14e^{-4}$	$2.02e^{-05}$
Weight Decay	$1.41e^{-5}$	$1.13e^{-5}$	$5.22e^{-4}$
Outer Batch Size	53	27	22

5.1.2 Hyperparameters importances

We calculate the important hyperparameters in the neural architecture search using DeepCAVE (Sass et al., 2022). We measure the importance using the Local Hyperparameter Importance metric (LPI) (Biedenkapp et al., 2019). In Figure 5.2 we can see the important hyperparameters for PAR_1, where using the common node has the greatest impact on the performance, followed by layer normalisation, the type of message passing layer and the hidden layer size. Figure 5.3 shows the importance of the hyperparameters for PAR_2. We can see that similarly to PAR_1, the common node is the most important feature, followed by layer normalisation. However, for this PAR value, layer normalisation has higher importance. The third most important hyperparameter is the type of message-passing layer. For PAR_{10}, we can see the important hyperparameters in Figure 5.4. Similarly, the common node is the most important hyperparameter and layer normalisation is the second most important. The hidden layers size and the message passing layer type are the third and fourth most important hyperparameters.

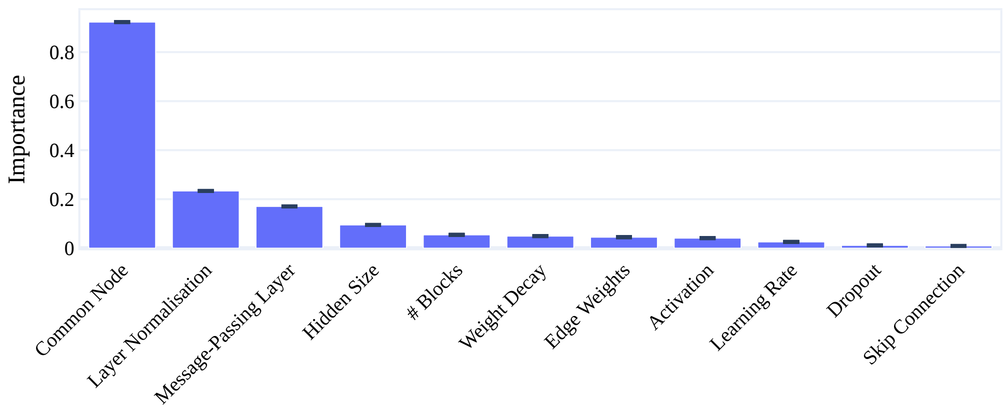


Figure 5.2: Local hyperparameter importance for the PAR$_1$ metric. Common node is the most important hyperparameter.

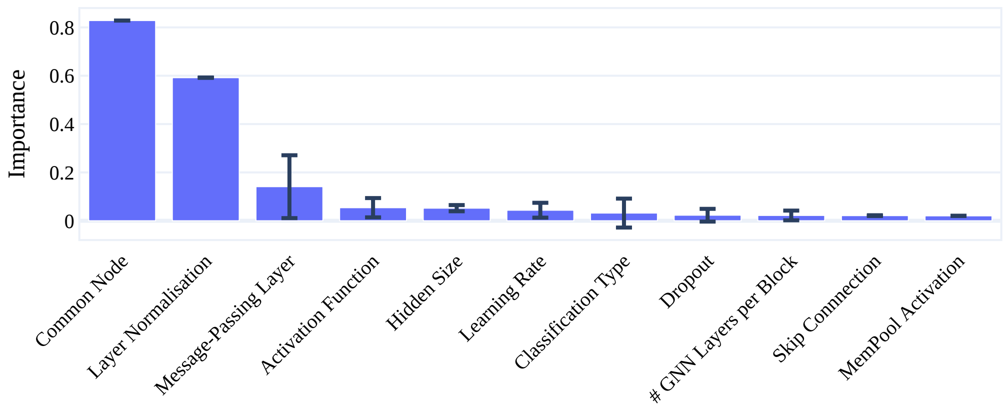


Figure 5.3: Local hyperparameter importance for the PAR$_2$ metric. Common node is the most important hyperparameter, followed by layer normalisation.

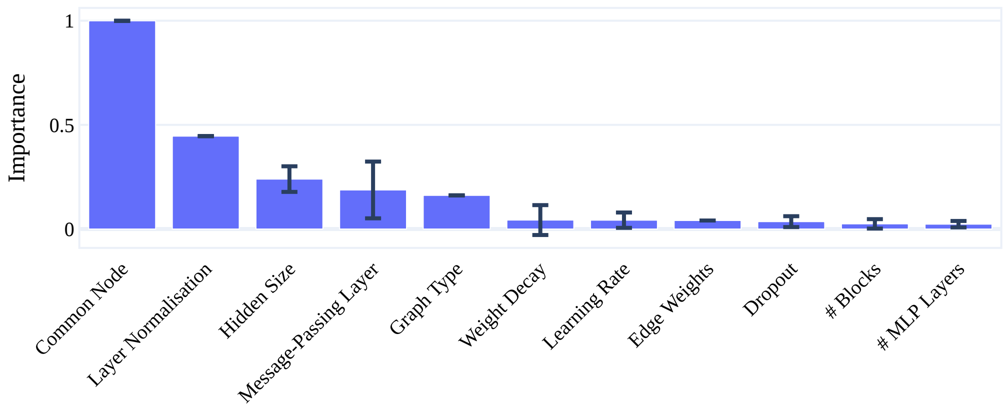


Figure 5.4: Common node is the most important hyperparameter. Common node is the most important hyperparameter, followed by layer normalisation.

5.2 AutoFolio with graph neural network features

We also check whether the features extracted by the graph neural network can improve the performance of AutoFolio with the SATZilla features for algorithm selection. To check that, we use the features-free neural network architectures found by the neural architecture search, as described in Section 4.2.1. We train the neural network using 13 random seeds and take the best model, measured by the total running time on the validation set. To get the features from a model, we remove the MLP (by replacing it with the identity function), and use the model's output for each instance as the extracted features.

In preliminary experiments, we found that using the features from the graph neural network together with the SATZilla features in AutoFolio results in significantly worse performance than using only the SATZilla features for algorithm selection on the scenario we generated in Section 4.1.1. A possible reason is that the dimension of the graph neural network features is high. For example, the number of graph neural network features for PAR$_1$ is 96 features, while there are 138 SATZilla features. The additional features can harm the performance of the Random Forest used in AutoFolio, as it decreases the rate of samples to features. Then, distinguishing between the important and unimportant features is harder.

Instead of using the features extracted by the graph neural network directly, we use the output of the MLP before the final prediction. We tested different MLPs with different numbers of features. Our goal is to process the features using the MLP to reduce the dimension of the extracted features. An overview of the feature extraction can be seen in Figure 5.1b. We used the following MLPs for PAR$_1$ and PAR$_2$:

- [2 layer] 96 $\rightarrow 48 \rightarrow 21$
- [3 layer] 96 $\rightarrow 48 \rightarrow 24 \rightarrow 21$
- [4 layer] 96 $\rightarrow 48 \rightarrow 24 \rightarrow 16 \rightarrow 21$
- [5 layer] 96 $\rightarrow 48 \rightarrow 24 \rightarrow 16 \rightarrow 8 \rightarrow 21$
- [6 layer] 96 $\rightarrow 48 \rightarrow 24 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 21$
- [6 layer] 96 $\rightarrow 48 \rightarrow 24 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 21$

for PAR$_{10}$, we used the following MLPs:

- [2 layer] 128 $\rightarrow 64 \rightarrow 21$
- [3 layer] 128 $\rightarrow 64 \rightarrow 48 \rightarrow 21$
- [4 layer] 128 $\rightarrow 64 \rightarrow 48 \rightarrow 24 \rightarrow 21$
- [5 layer] 128 $\rightarrow 64 \rightarrow 48 \rightarrow 24 \rightarrow 16 \rightarrow 21$
- [6 layer] 128 $\rightarrow 64 \rightarrow 48 \rightarrow 24 \rightarrow 16 \rightarrow 8 \rightarrow 21$
- [7 layer] 128 $\rightarrow 64 \rightarrow 48 \rightarrow 24 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 21$
- [8 layer] 128 $\rightarrow 64 \rightarrow 48 \rightarrow 24 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 21$

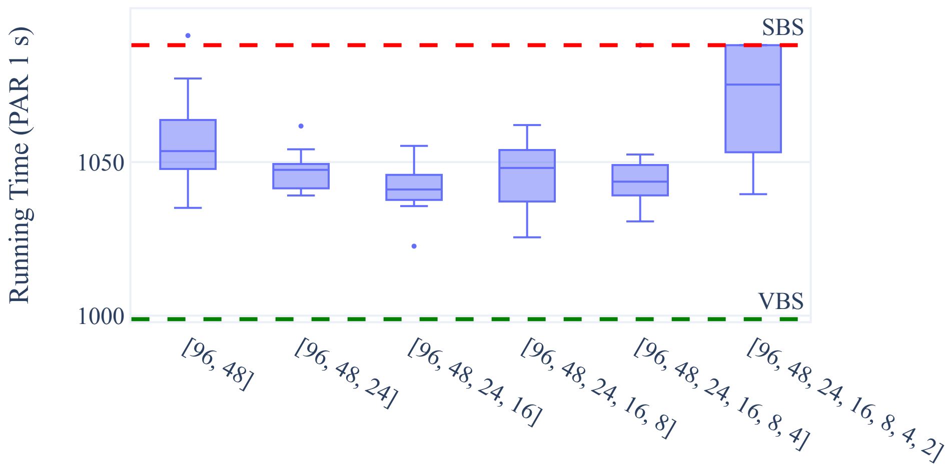


Figure 5.5: Running times of the different MLPs tested for PAR_1. The 4-layer architecture has the lowest running time.

In this list, every number represents an MLP layer with the indicated number of neurons. The leftmost number is the first layer after the message passing part, and the rightmost is the output layer. The different MLPs tested for PAR_{10} are because of the different output shapes of the message-passing part. The output shape of the message passing part is determined by the hidden size of the network and the jumping knowledge layer before the MLP. We train from scratch every architecture for 13 different random seeds on the generated scenario. The result of the MLPs experiments can be found in Figure 5.5. The architecture with the lowest running time for PAR_1 is the 4-layer architecture, which ends with 16 features before the predictions. For PAR_2 the results can be seen in Figure 5.6. The 2-layers architecture has the lowest running time. For PAR_{10} the results can be seen in Figure 5.7. The 2-layers architecture has the lowest running time. It should be noted that the better results of the architectures with fewer layers are because they have fewer parameters, making the training process faster.

We end with three architectures, one for every PAR score. We can then extract for every instance three graph neural network feature sets, for PAR_1, PAR_2 and PAR_{10}. Following the feature extraction, we create three feature sets for each PAR score:

- Graph neural network features of a specific PAR score.
- SATZilla and graph neural network features of a specific PAR score.
- SATZilla and graph neural network features of all PAR score (*i.e.*, all available features).

Those feature sets can be used with any features-based algorithm selection method. Similarly to Chapter 4, we use AutoFolio.

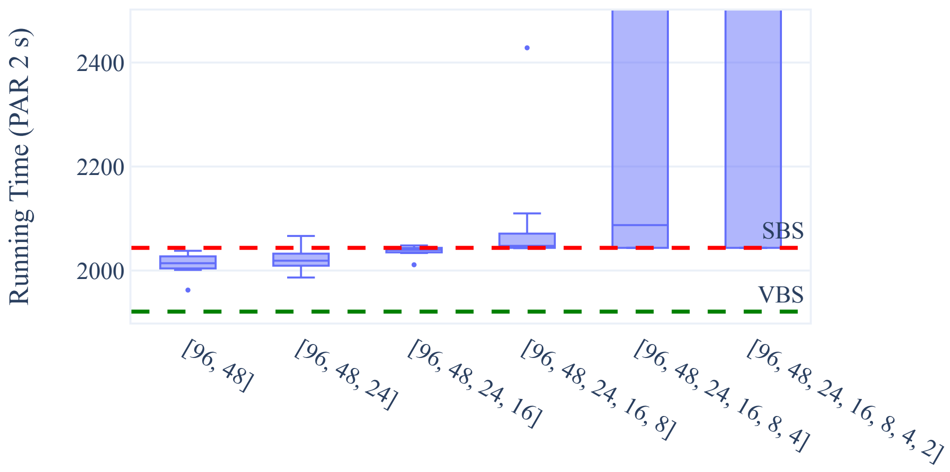


Figure 5.6: Running times of the different MLPs tested for PAR_2. The 2-layers architecture has the lowest running time.

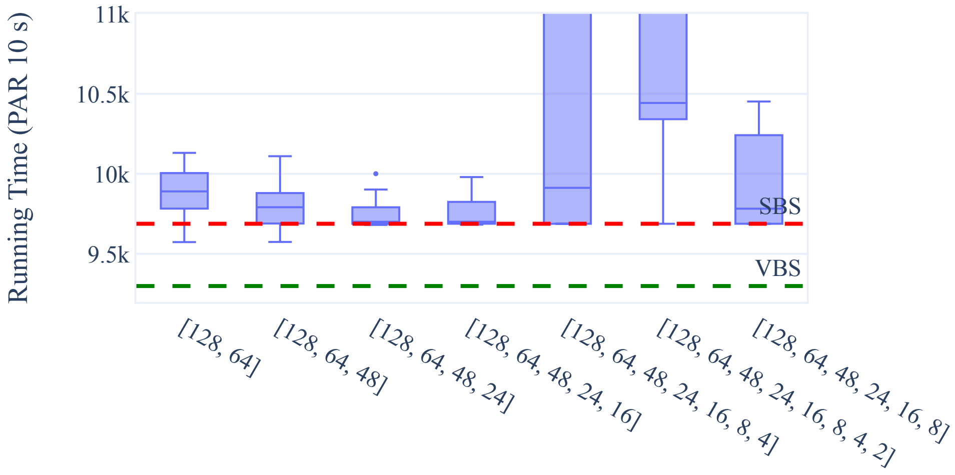


Figure 5.7: Running times of the different MLPs tested for PAR_{10}. The 2-layers architecture has the lowest running time.

Table 5.2: Description of the SAT competition ASLib scenarios used in the experiments.

Scenario	Number of Instances	Number of Features	Number of Solvers
SAT11-HAND	296	115	15
SAT11-RAND	600	115	9
SAT11-INDU	300	115	18
SAT18	353	54	37
SAT20	400	108	67

5.3 Experimental setup

In this chapter, we introduced a combined neural network that uses both the SATZilla features and a graph neural network. We also introduced new feature sets that utilise features extracted from graph neural networks. Those features sets can be used with AutoFolio. We first evaluate all methods on the generated scenario with 13 different random seeds. We also compare to the features-free graph neural networks and AutoFolio with the SATZilla features.

We then evaluate AutoFolio with the different feature sets on SAT scenarios from ASlib and the Configurable SAT Solver Challenge (CSSC) (Hutter et al., 2017). Those scenarios are used to compare the performance of SAT solvers and algorithm selectors. However, due to the size of the instances they contain, we cannot train the graph neural network and the combined neural network on them. Extracting the features from the pre-trained network (on the generated scenario) is possible for most formulas. To extract the graph neural network features, we use a single A100 GPU with 80 GB of vRAM. In the case of out-of-memory, we set the graph neural network features of the instance as missing features. We set the graph neural network features costs as the inference time of the instance.

We run AutoFolio with a time limit of 48 hours, as used in (Lindauer et al., 2015), on various scenarios, as described in the following subsections. We run AutoFolio on a cluster. The cluster has 34 nodes, and each node has two Intel® Xeon® CPU E5-2683 v4 2.1GHz with 16 cores each. Each node also has 94GB of memory. Each CSSC scenario was assigned 5 GB of memory and one core, and each SAT competition scenario was assigned 10 GB of memory and one core, as 5 GB resulted in an out-of-memory error in some cases for those scenarios.

5.3.1 ASlib and CSSC Scenarios

In this section, we describe the ASlib and CSSC scenarios we use. An overview of the ASlib scenarios we used is given in Table 5.2.

The Configurable SAT Solve Challenge (CSSC) 2013 contains a few instance categories (also called benchmarks), each from a different type of instance, such as only hardware verification instances or only software verification instances. This is closer to real-life usage of SAT solvers, where we want to use the SAT solvers on instances in a very specific category. The SAT solvers participating in the competition were configured using automated algorithm configuration methods such as SMAC separately for each benchmark to get a well-performing configuration for each instance distribution. We use the instances and running time of the CSSC 2013 to create new scenarios in ASlib format. An overview of the ASlib scenarios we used is given in Table 5.3.

Table 5.3: Description of the CSSC ASLib scenarios used in the experiments.

Scenario	Category	Number of Instances	Number of Features	Number of Solvers
K3	Random	550	80	10
BMC08	Industrial	1109	80	10
IBM	Industrial	684	126	10

Table 5.4: Overview of the different features sets.

PAR	Name	SATZilla	GNN PAR_1	GNN PAR_2	GNN PAR_{10}
1	AutoFolio SATZilla	X			
	AutoFolio GNN		X		
	AutoFolio Both	X	X		
	AutoFolio All	X	X	X	X
2	AutoFolio SATZilla	X			
	AutoFolio GNN			X	
	AutoFolio Both	X		X	
	AutoFolio All	X	X	X	X
10	AutoFolio SATZilla	X			
	AutoFolio GNN				X
	AutoFolio Both	X			X
	AutoFolio All	X	X	X	X

5.4 Results

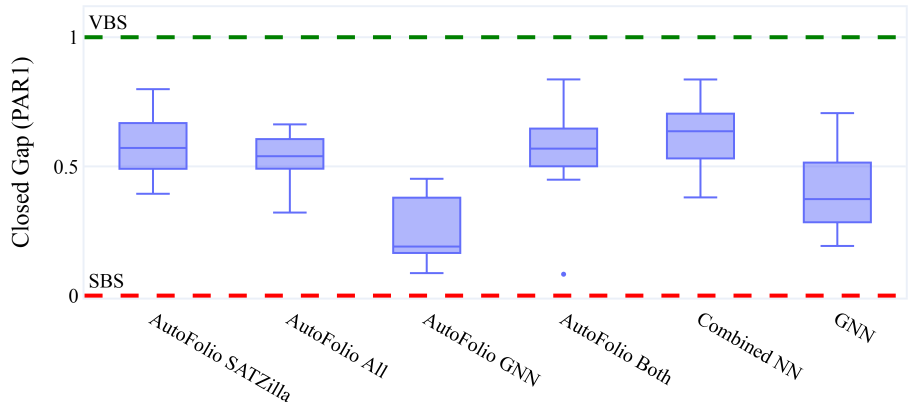
In this section, we show the results of our experiments with graph neural networks and the SATZilla features. We show the results on the generated dataset in Section 5.4.1, and the results on the ASlib and CSSC scenarios in Section 5.4.2.

5.4.1 Comparison of methods on the generated scenario

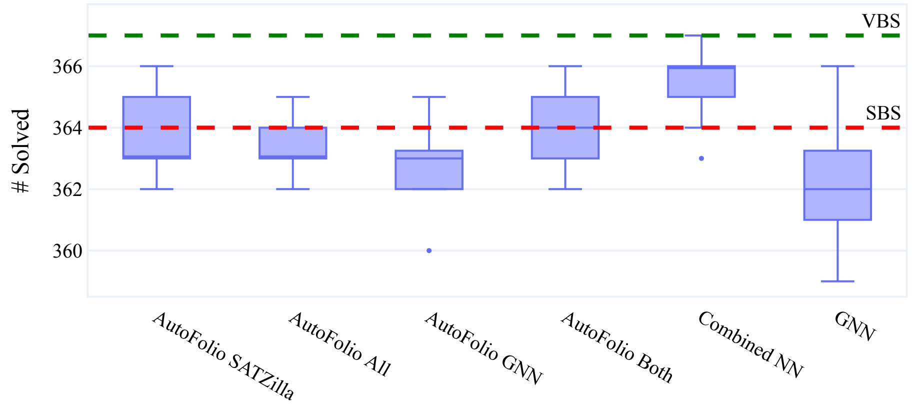
In this subsection, we compare the methods on the generated scenario. We compare AutoFolio with the SATZilla features (AutoFolio SATZilla), AutoFolio with SATZilla features and graph neural neural network features for the specific PAR score (AutoFolio Both), AutoFolio with all available feature sets, including SATZilla features and the three graph neural network features sets (AutoFolio All), AutoFolio only with the graph neural network features for the specific PAR score (GNN) and the neural network that uses both graph neural network and the SATZilla features (Combined NN). An overview of the feature sets can be seen in Table 5.4.

Results for PAR_1

Figure 5.8a shows the closed gap of the various methods on the generated scenario for PAR_1. We can see that all methods have positive closed gap values. The method with the highest median closed gap is the Combined NN, followed by AutoFolio Both. This shows that for PAR_1, the GNN and the SATZilla



(a)



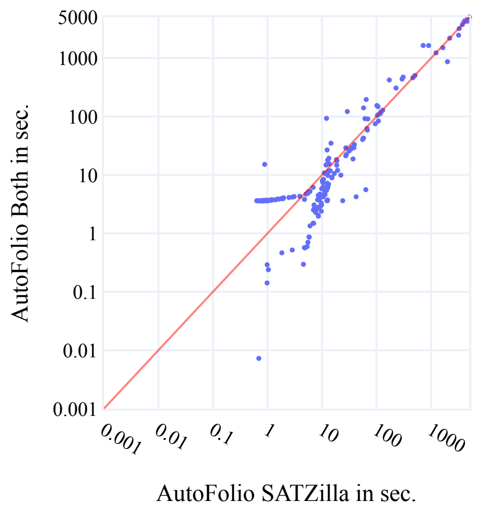
(b)

Figure 5.8: Results of PAR_1 experiments. (a) Closed gap of the methods. (b) Number of instances solved.

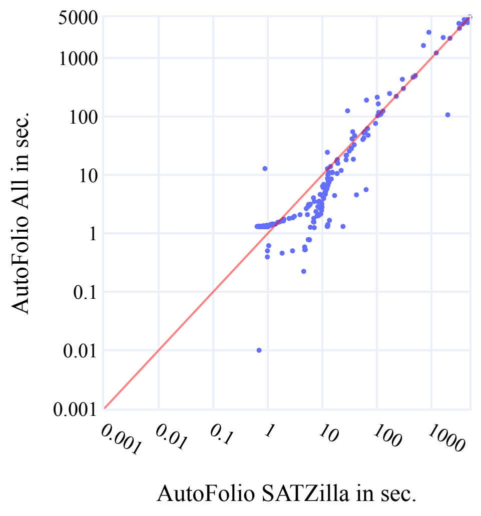
features can provide the best results. AutoFolio All has worse performance, which can be attributed to the larger configuration space. We can also see that GNN performs better than AutoFolio GNN, which means that using the graph neural network features in an end-to-end way leads to better performance.

In Figure 5.8b we can see the number of solved instances for PAR_1. We can see that the Combined NN has the best performance and is the only method to reach the number of solved instances as the virtual best solver. It should be noted, however, that the methods are optimised for PAR_1, which means that timeouts are not penalised. Figure 5.9 shows scatter plots of the mean running times of each instance. We can see that the methods based on AutoFolio have similar patterns, with most points around the diagonal. When comparing the Combined NN to AutoFolio with the SATZilla features, we can see that the neural network is superior on instances with short running times but works worse on the other instances.

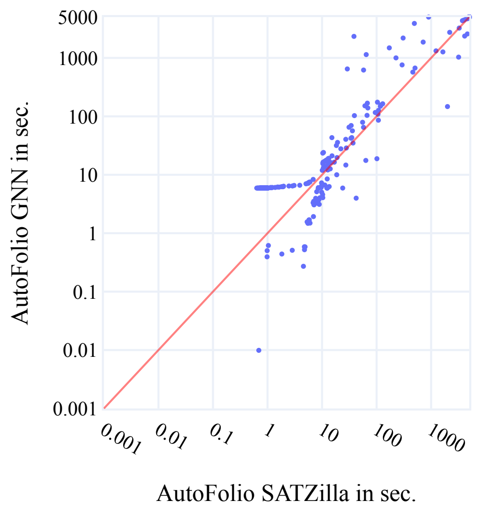
We check the statistical significance of the closed gap values using AutoRank with a significance level of 0.05. The ANOVA test found statistical significance between the performance differences of the methods. The posthoc Tukey HSD test found no significance between AutoFolio All, AutoFolio Both, AutoFolio SATZilla, AutoFolio SATZilla, AutoFolio Both, AutoFolio SATZilla and Combined. The other methods are significantly different from each other.



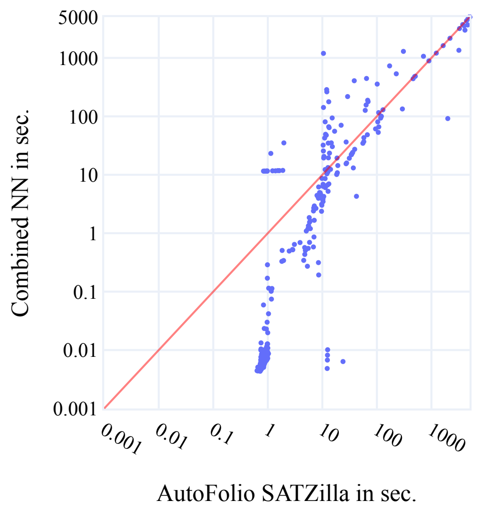
(a)



(b)



(c)



(d)

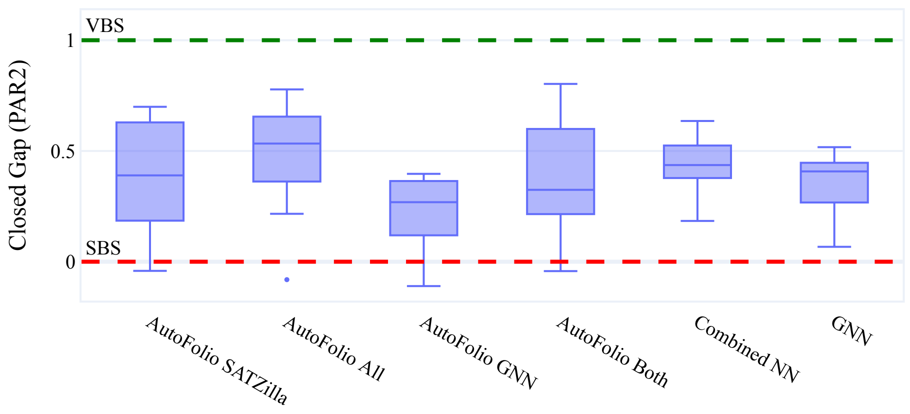
Figure 5.9: Scatter plots of the mean running time of the methods. optimised for PAR_1

Results for PAR_2

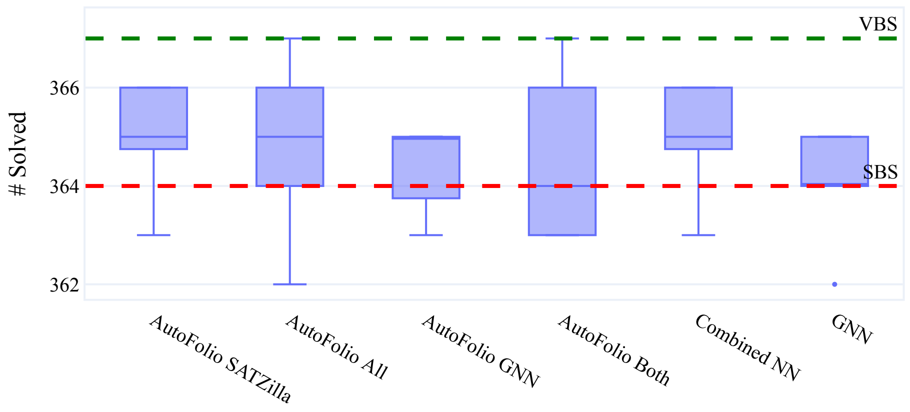
We can see the closed gap value for PAR_2 in Figure 5.10a. For this PAR score, AutoFolio, with all the available features, has the highest median PAR score, while AutoFolio Both has a lower median score, but a higher maximal closed gap. AutoFolio All also has a higher median score than AutoFolio SATZilla. The combined neural network has a higher median closed gap value than the graph neural network alone. Moreover, the combined neural network has a higher median closed gap to AutoFolio with only the SATZilla features. The GNN performs better than the graph neural network features with AutoFolio.

In Figure 5.10b we can see the number of solved instances for PAR_2. We can see that AutoFolio Both and All have the highest number of solved instances. The combined neural network has the same number of solved instances as AutoFolio SATZilla, which is better than using the graph neural network alone. Using the graph neural network features with AutoFolio has a higher median number of solved instances.

The mean running time of each instance for PAR_2 can be seen in Figure 5.11. We can see that for



(a)



(b)

Figure 5.10: Results of PAR_2 experiments. (a) Closed gap of the methods. (b) Number of instances solved.

the three methods that are based on AutoFolio, the instances' running times are around the diagonal. For AutoFolio Both and AutoFolio All, many instances are below the diagonal, which means that the methods are faster than using only the SATZilla features. For AutoFolio GNN, the instances are close to the diagonal with no clear pattern. The instances with short running times for the combined NN are below the diagonal, similar to the feature-free algorithm selection. For instances with higher running times, AutoFolio SATZilla is better.

We check the statistical significance of the closed gap values using AutoRank with a significance level of 0.05. The ANOVA test found statistical significance between the performance differences of the methods. The posthoc Tukey HSD test found no significance between AutoFolio All, AutoFolio Both, AutoFolio SATZilla, and Combined NN. Also, there is no significance between GNN, AutoFolio Both, AutoFolio SATZilla, and Combined NN. The other methods are significantly different from each other.

Results for PAR_{10}

We can see the closed gap value for PAR_{10} in Figure 5.12a. For this PAR score, no method achieves only positive closed gap values. GNN is the worst method, but using the graph neural network features with AutoFolio results in better performance that is comparable to other methods. The combined neural network has the best median performance, while AutoFolio Both has the highest maximal closed gap.

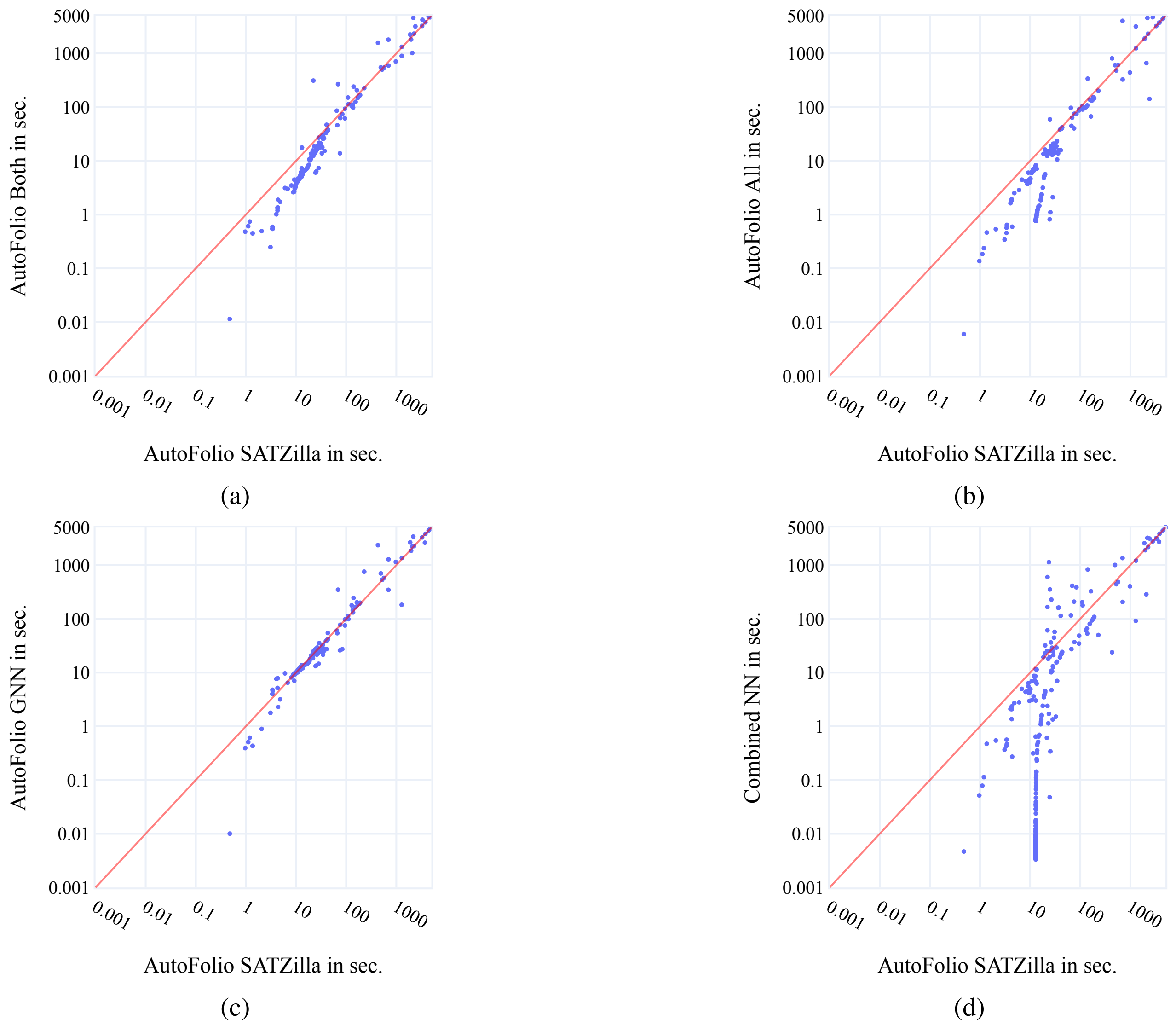
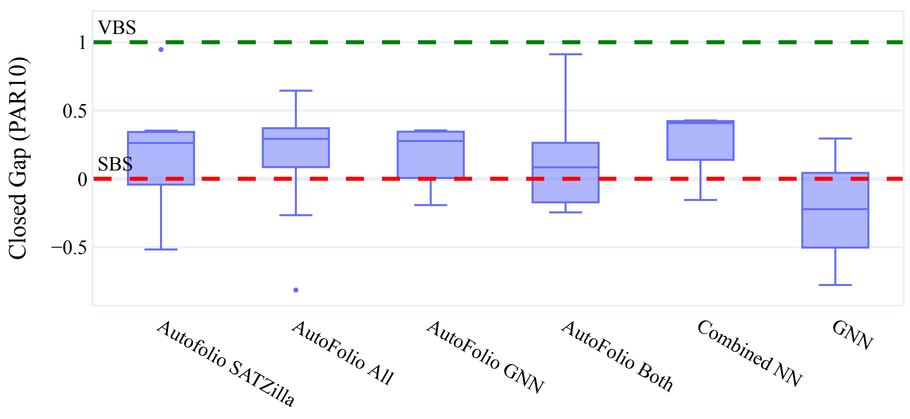


Figure 5.11: Scatter plots of the mean running time of the methods, optimised for PAR_2.

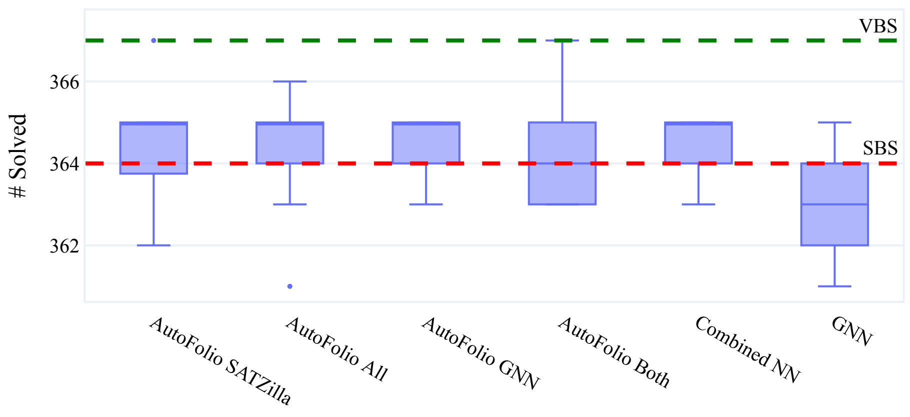
The number of solved instances per method can be seen in Figure 5.12b. AutoFolio SATZilla, AutoFolio All, AutoFolio GNN and the combined NN have the highest median number of solved instances, while the GNN alone has the lowest.

The mean running time of each instance for PAR_{10} can be seen in Figure 5.13. Similarly to other PAR scores, using the graph neural network features results in better performance for instances with short running times, while for instances with longer running times. The Combined neural network performs better than AutoFolio, with only the SATZilla features in most instances.

We check the statistical significance of the closed gap values using AutoRank with a significance level of 0.05. The Friedman test found statistical significance between the performance differences of the methods. The post hoc Nemenyi test found no significance between GNN, AutoFolio Both, Autofolio SATZilla, AutoFolio GNN, and AutoFolio All. Also, there is no significance between AutoFolio Both, Autofolio SATZilla, AutoFolio GNN, AutoFolio All, and Combined NN. The other methods are significantly different from each other.



(a)



(b)

Figure 5.12: Results of PAR_{10} experiments. (a) Closed gap of the methods. (b) Number of instances solved.

5.4.2 ASlib and CSSC scenarios

In this section, we compare the methods on the ASlib and CSSC scenarios containing instances from SAT competitions and challenges. We compare AutoFolio with the SATZilla features (AutoFolio SATZilla), AutoFolio with SATZilla features and graph neural network features for the specific PAR score (AutoFolio Both), AutoFolio with all available feature sets, including SATZilla features and the three graph neural network features sets (AutoFolio All) and AutoFolio only with the graph neural network features for the specific PAR score (AutoFolio GNN). For each PAR score, we run AutoFolio twice, with and without taking into account the feature costs in the calculation of the total running time of the selector. We do this as the feature extraction using graph neural networks can take a long time, even using a GPU (more than 60 seconds). For the generated scenario, the feature extraction is brief (less than a second).

Results for PAR_1 with features costs

In Table 5.5 we can see the performance of AutoFolio with the four feature sets. The runs are optimised for PAR_1 taking into account the features extraction time. We can see that the graph neural network features alone are worse than the other feature groups that use the SATZilla features. However, in the IBM scenario, the graph neural network features provide the best results. On the other scenarios, it provides

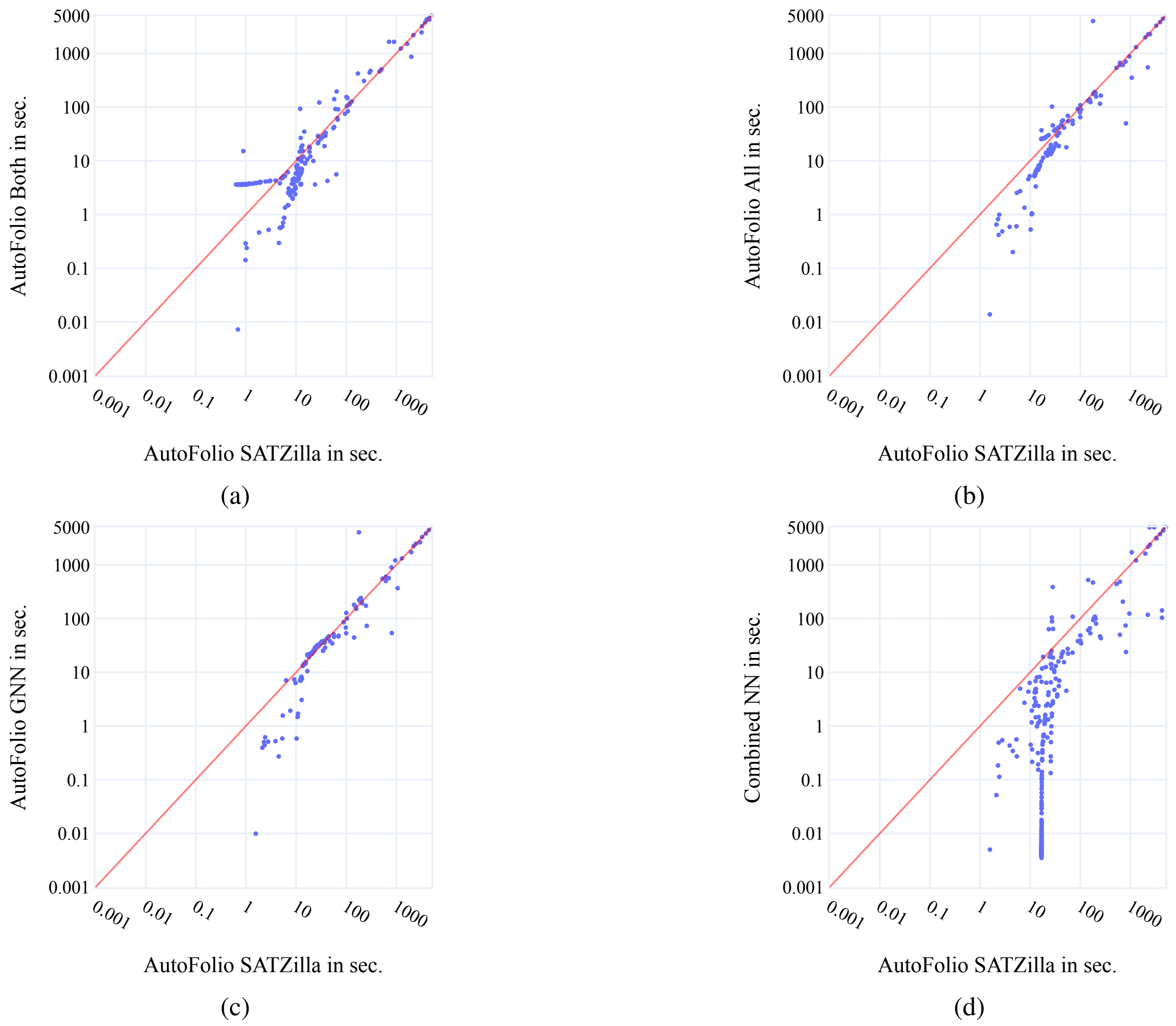


Figure 5.13: Scatter plots of the mean running time of the methods, optimised for PAR_{10}.

mostly positive closed gap values (*i.e.* better than the single best solver). In three scenarios, using the graph neural network features with the SATZilla features provides the best results. Using all available features, including the three groups of graph neural network features, provides the best results on four scenarios. This shows that the graph neural network features are transferable between different scenarios, and in some cases, can complement the SATZilla features.

Using AutoRank, the Friedman test found statistical significance between the performance differences of the methods. The post hoc Nemenyi test found no significance between GNN, Both, SATzilla, and All.

Figure 5.14 and Figure 5.15 shows scatter plots of the running times of AutoFolio using the different features sets, compared to the SATZilla features. In those figures, we show the scatter plots for SAT11-HAND and CSSC-BMC08. Scatter plots for all scenarios are available in Appendix B. We can see that for SAT11-RAND, the graph neural network features alone are better on instances with short running times, while the SATZilla features are better on instances with longer running times and in reducing timeouts. Using both the SATZilla features and all available features also provides better results on instances with

Scenario	AutoFolio SATzilla	AutoFolio GNN	AutoFolio Both	AutoFolio All
SAT11-HAND	0.73	0.67	**0.76**	0.72
SAT11-RAND	**0.93**	0.69	**0.93**	**0.93**
SAT11-INDU	**0.43**	-0.08	0.4	0.38
SAT18-EXP	0.58	0.24	0.64	**0.65**
SAT20-MAIN	0.23	0.09	0.23	**0.29**
CSSC-BMC08	0.33	0.01	**0.4**	0.37
CSSC-IBM	0.82	**0.84**	0.82	0.83
CSSC-K3	**0.73**	0.68	0.71	**0.73**

Table 5.5: Closed gap values of AutoFolio on various algorithm selection scenarios for PAR_1 (with feature costs). SATZilla represents only using the SATZilla features, GNN represents using only the graph neural network features, Both represent when using the SATZilla features and the graph neural network features for PAR_1, All represents when using the SATZilla features and the graph neural network features for all PAR scores (*i.e.* all available features). Bold denotes the maximal value for the scenario.

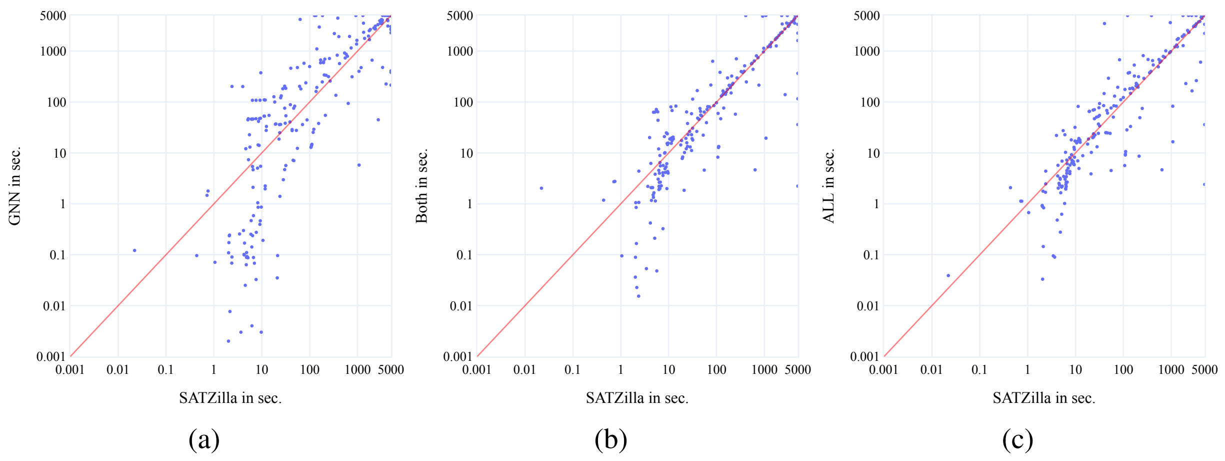


Figure 5.14: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT11-HAND scenario (with feature costs), optimised for PAR_1.

short running times. There is no clear pattern for the other instances. For BMC08, the instances are scattered around the diagonal line, which means that the running times of AutoFolio using the different feature sets are similar. The graph neural network features alone have more instances with higher running times than the SATZilla features.

Results for PAR_1 without features costs

We can see the closed gap values optimised for PAR_1 without features costs in Table 5.6. We can see that the graph neural network features alone have positive closed gap values in all scenarios, while when taking the features extraction time into account in the SAT11-INDU there is a negative closed gap. This can happen due to high feature extraction time relative to the time required to solve the instances. We can also see that in most scenarios, using the graph neural network features (either all of them or the ones optimised for PAR_1) with the SATZilla features can result in better performance.

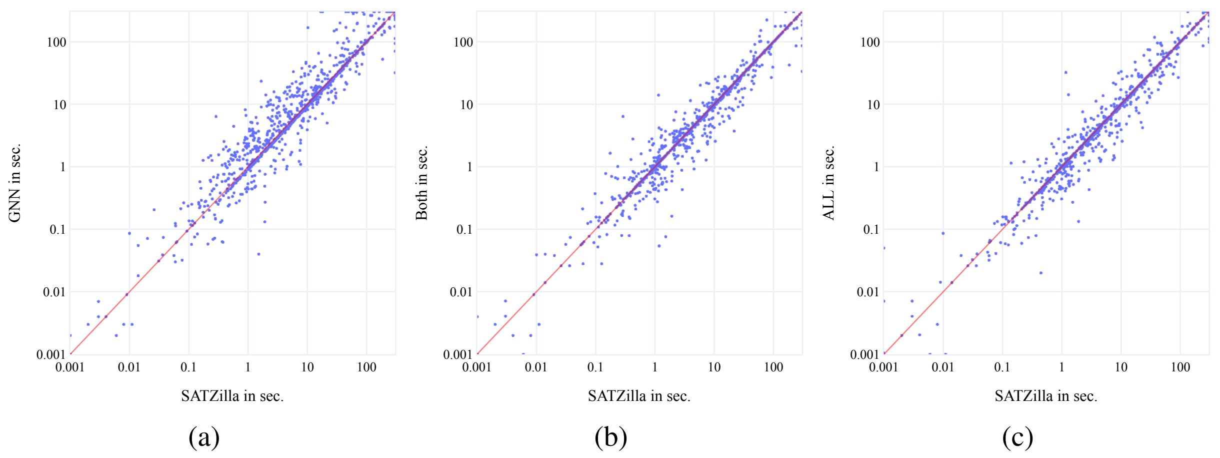


Figure 5.15: Scatter plots of the running times of SATZilla, GNN, Both and all features on the CSSC-BMC08 scenario (with feature costs), optimised for PAR_1.

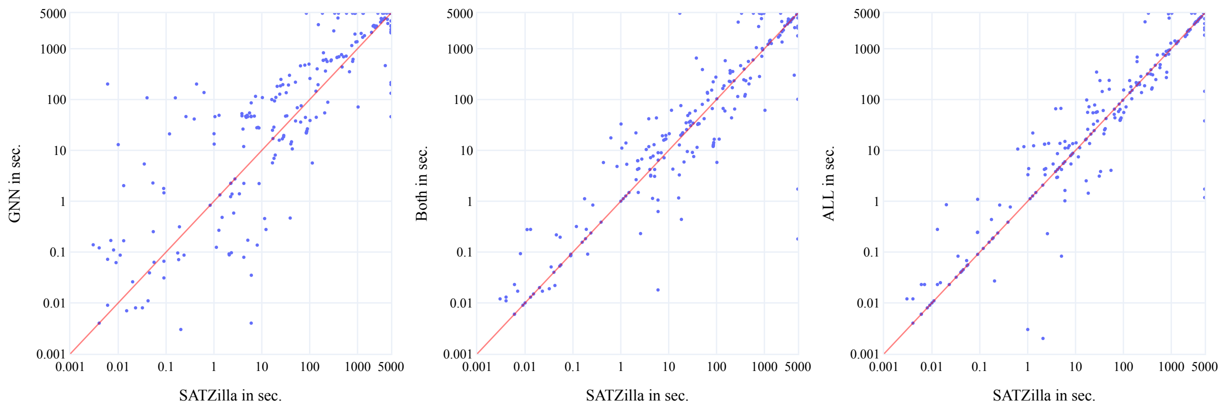
Scenario	AutoFolio SATzilla	AutoFolio GNN	AutoFolio Both	AutoFolio All
SAT11-HAND	0.76	0.66	0.73	**0.78**
SAT11-RAND	**0.94**	0.68	0.92	**0.94**
SAT11-INDU	0.39	0.09	**0.48**	0.26
SAT18-EXP	0.61	0.2	**0.65**	0.6
SAT20-MAIN	**0.34**	0.06	0.27	0.28
CSSC-BMC08	**0.56**	0.05	0.52	0.47
CSSC-IBM	0.91	0.78	0.92	**0.93**
CSSC-K3	0.74	0.73	0.76	**0.78**

Table 5.6: Closed gap values of AutoFolio on various algorithm selection scenarios for PAR_1 (without feature costs). SATZilla represents only using the SATZilla features, GNN represents using only the graph neural network features, Both represent when using the SATZilla features and the graph neural network features for PAR_1, All represents when using the SATZilla features and the graph neural network features for all PAR scores (*i.e.* all available features). Bold denotes the maximal value for the scenario.

Using AutoRank, the Friedman test found statistical significance between the performance differences of the methods. The post hoc Nemenyi test found no significance between Both, SATzilla, and All.

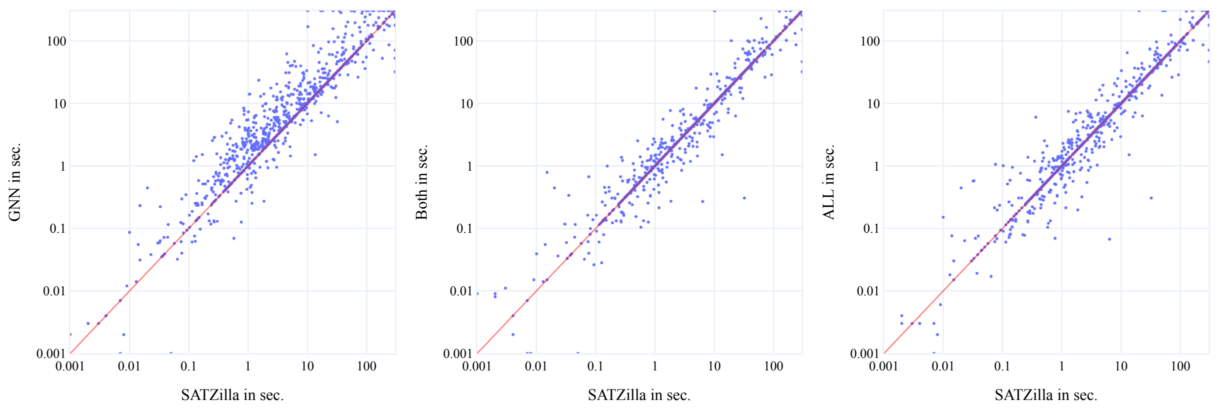
In Figure 5.16, we can see a scatter plot of the running times of the instances when using the four different instance sets on the SAT11-HAND scenario. There is no clear pattern of the points in the scatter plots. When comparing GNN and SATZilla, the points are more sparse, while in the other plots, the points are more concentrated around the diagonal line. This means that the running times of the methods are similar, as shown in the closed gap results. Figure 5.17 we can see scatter plots of the running times of the instances on the BMC08 scenario. Scatter plots for all scenarios are available in Appendix B.

Using AutoRank, the Friedman test found statistical significance between the performance differences of the methods. The post hoc Nemenyi test found no significance between Both, SATzilla, and Multi.



(a) (b) (c)

Figure 5.16: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT11-HAND scenario (without feature costs), optimised for PAR_1.



(a) (b) (c)

Figure 5.17: Scatter plots of the running times of SATZilla, GNN, Both and all features on the CSSC-BMC08 scenario (without feature costs), optimised for PAR_1.

Results for PAR_2 with features costs

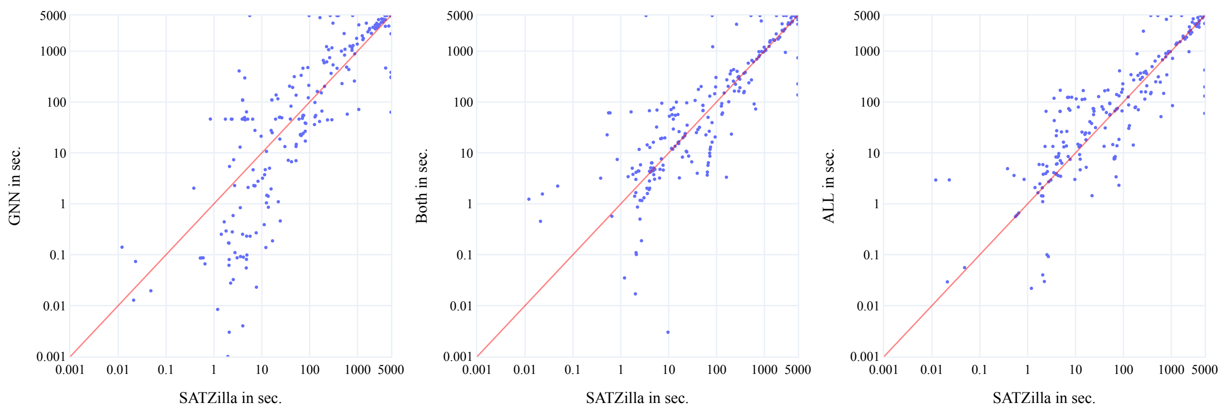
We can see the closed gaps of the four feature sets optimised of PAR_2 when using features costs in Table 5.7. GNN has positive closed gap values in six out of eight scenarios. Using the graph neural network features with AutoFolio results in better performance in three scenarios. Using all available features results in better performance in two scenarios.

Using AutoRank, the Friedman test found statistical significance between the performance differences of the methods. The post hoc Nemenyi test found no significance between GNN and All and the groups All, Both, and SATzilla.

In Figure 5.18, we can see a scatter plot of the running times of the instances when using the four different instance sets on the SAT11-HAND scenario. The points are scattered around with no clear pattern. When comparing Both and All to SATZilla, the points are closer to the diagonal compared to GNN. Figure 5.19 we can see scatter plots of the running times of the instances on the BMC08 scenario. We can see that using only the graph neural network features results in worse performance over the SATZilla

Scenario	AutoFolio SATzilla	AutoFolio GNN	AutoFolio Both	AutoFolio All
SAT11-HAND	**0.76**	0.66	0.71	0.75
SAT11-RAND	**0.95**	0.76	0.94	**0.95**
SAT11-INDU	**0.44**	-0.05	0.33	0.4
SAT18-EXP	0.57	0.13	0.6	**0.61**
SAT20-MAIN	**0.24**	-0.03	**0.24**	0.2
CSSC-BMC08	0.33	0.14	**0.36**	0.32
CSSC-IBM	0.85	0.86	**0.88**	0.85
CSSC-K3	**0.75**	0.48	0.7	0.57

Table 5.7: Closed gap values of AutoFolio on various algorithm selection scenarios for PAR_2 (with feature costs). SATZilla represents only using the SATZilla features, GNN represents using only the graph neural network features, Both represent when using the SATZilla features and the graph neural network features for PAR_2, All represents when using the SATZilla features and the graph neural network features for all PAR scores (*i.e.* all available features). Bold denotes the maximal value for the scenario.



(a) (b) (c)

Figure 5.18: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT11-HAND scenario (with feature costs), optimised for PAR_2.

features for most instances. When comparing using both SATZilla features and graph neural network features, and all available feature sets, the points are around the diagonal line with no clear pattern. Scatter plots for all scenarios are available in Appendix B.

Results for PAR_2 without features costs

Table 5.8 shows the closed gap values optimised for PAR_2 without features costs. We can see that the graph neural network features alone have positive closed gap values in all but one scenario. Using the graph neural network features with the SATZilla features results in the best performance in six out of eight scenarios. Using all available features results in the best performance in three scenarios. The better-closed gap values of the methods that use the graph neural network features when not using the features costs can suggest that extracting those features takes a relatively long time.

Using AutoRank, the Friedman test found statistical significance between the performance differences

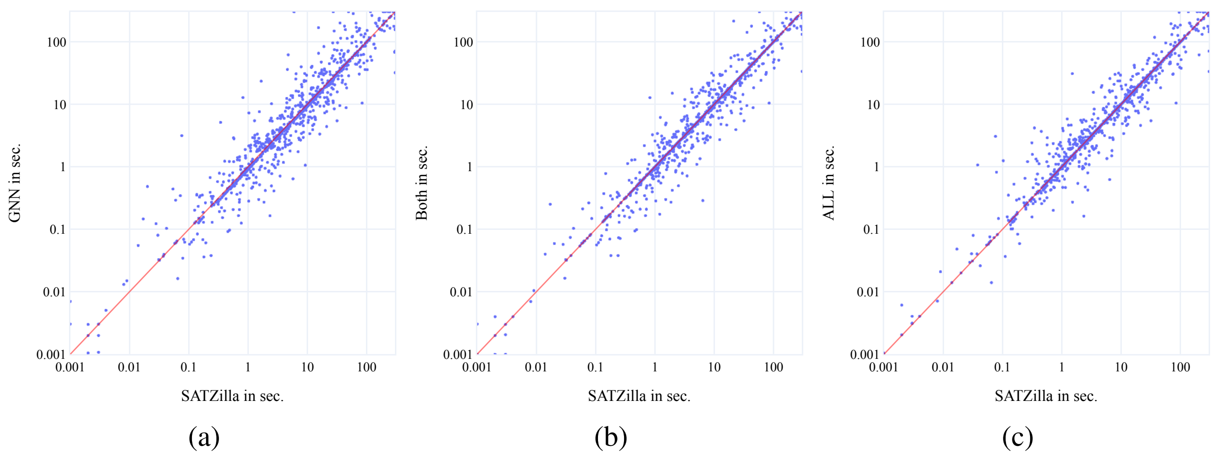


Figure 5.19: Scatter plots of the running times of SATZilla, GNN, Both and all features on the CSSC-BMC08 scenario (with feature costs), optimised for PAR_2.

Scenario	AutoFolio SATzilla	AutoFolio GNN	AutoFolio Both	AutoFolio All
SAT11-HAND	0.74	0.68	**0.81**	0.75
SAT11-RAND	**0.95**	0.72	**0.95**	**0.95**
SAT11-INDU	0.34	-0.04	**0.41**	0.39
SAT18-EXP	0.58	0.19	**0.59**	**0.59**
SAT20-MAIN	0.27	0.06	0.29	**0.31**
CSSC-BMC08	**0.57**	0.15	0.54	0.44
CSSC-IBM	**0.95**	0.87	**0.95**	0.9
CSSC-K3	0.69	0.46	**0.77**	0.67

Table 5.8: Closed gap values of AutoFolio on various algorithm selection scenarios for PAR_2. SATZilla represents only using the SATZilla features, GNN represents using only the graph neural network features, Both represent when using the SATZilla features and the graph neural network features for PAR_2, All represents when using the SATZilla features and the graph neural network features for all PAR scores (*i.e.* all available features). Bold denotes the maximal value for the scenario.

of the methods. The post hoc Nemenyi test found no significance between GNN and SATzilla and the groups SATzilla, Multi, and Both.

We can see scatter plots of the running times of AutoFolio with the four feature sets on the SAT11-HAND scenario in Figure 5.20. When comparing the SATZilla features and the graph neural network features, we can see that the points are scattered around, with more points in the upper part of the graph, suggesting better performance for the SATZilla features set. For the other two groups, there is no clear pattern. For the BMC08 scenario, the scatter plots can be seen in Figure 5.21. When comparing GNN and SATZilla, for most instances, SATZIlla is better. When comparing SATZilla with All and Both, the points are close to the diagonal.

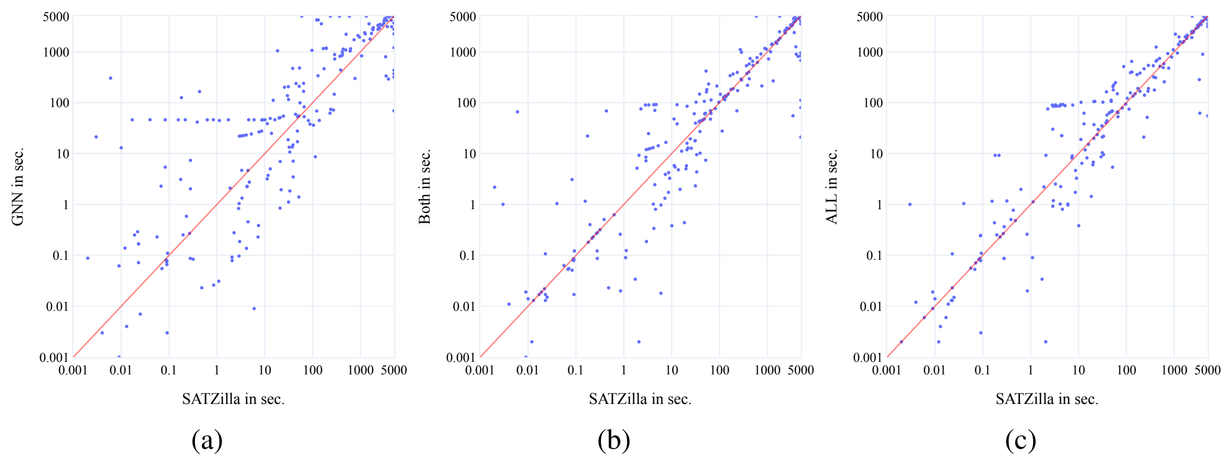


Figure 5.20: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT11-HAND scenario (without feature costs), optimised for PAR_2.

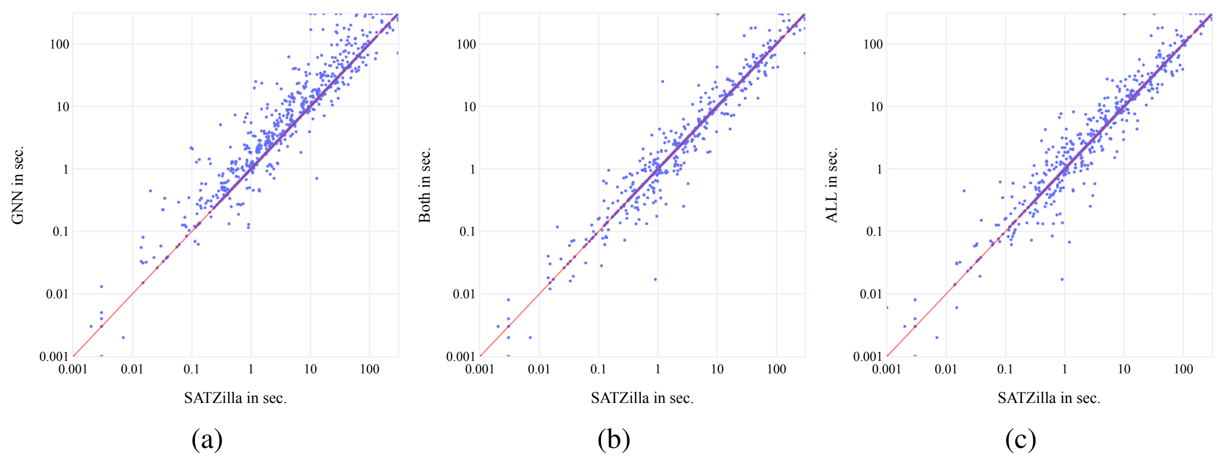


Figure 5.21: Scatter plots of the running times of SATZilla, GNN, Both and all features on the CSSC-BMC08 scenario (without feature costs), optimised for PAR_2.

Results for PAR_{10} with features costs

We can see the closed gap values of AutoFolio with the various feature sets when optimised for PAR_{10} with features costs in Table 5.9. We can see that the graph neural network features alone have positive closed gap values in six out of eight scenarios. It also has the best performance in the IBM scenario. The SATZilla features set is the best on most scenarios, while on three scenarios, using a combination of the SATZilla with graph neural network features results in better performance.

Using AutoRank, the Friedman test found statistical significance between the performance differences of the methods. The post hoc Nemenyi test found no significance between GNN, Both, and All and between Both, All, and SATzilla.

In Figure 5.22, we can see a scatter plot of the running times of the instances when using the four different instance sets on the SAT11-HAND scenario. We can see that using the GNN features results in better performance for instances with shorter running times but worse performance for the other instances. For the other feature sets, the instances are around the diagonal. For the BMC08 scenario, the scatter

Scenario	AutoFolio SATzilla	AutoFolio GNN	AutoFolio Both	AutoFolio All
SAT11-HAND	**0.75**	0.7	0.68	0.74
SAT11-RAND	0.94	0.79	0.94	**0.95**
SAT11-INDU	0.34	-0.02	**0.41**	0.37
SAT18-EXP	**0.61**	0.25	0.56	0.6
SAT20-MAIN	**0.28**	0.07	0.23	0.26
CSSC-BMC08	0.14	-0.09	0.05	**0.25**
CSSC-IBM	**0.95**	**0.95**	0.82	0.93
CSSC-K3	**0.47**	0.13	0.1	0.46

Table 5.9: Closed gap values of AutoFolio on various algorithm selection scenarios for PAR_{10} (with feature costs). SATZilla represents only using the SATZilla features, GNN represents using only the graph neural network features, Both represent when using the SATZilla features and the graph neural network features for PAR_{10}, All represents when using the SATZilla features and the graph neural network features for all PAR scores (*i.e.* all available features). Bold denotes the maximal value for the scenario.

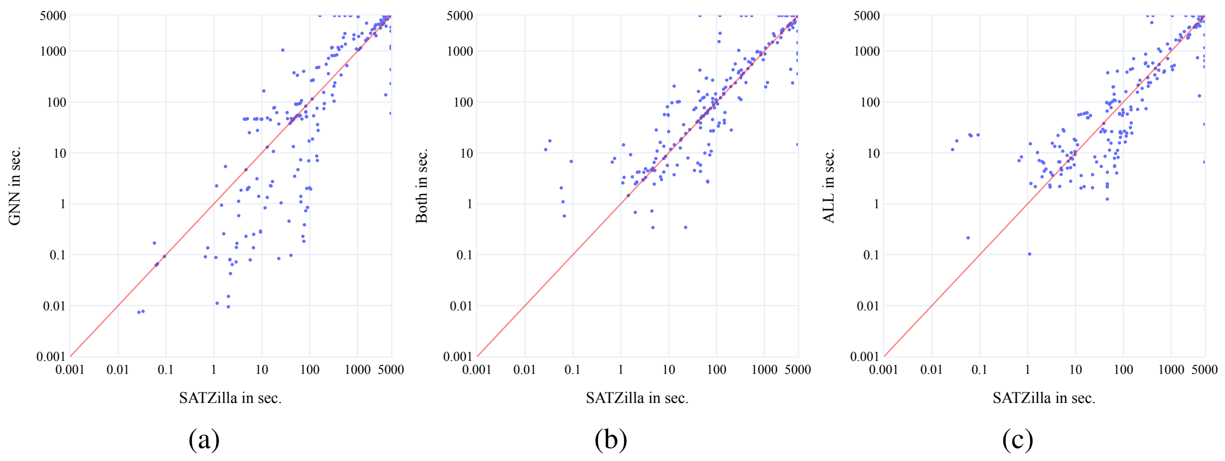


Figure 5.22: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT11-HAND scenario (with feature costs), optimised for PAR_{10}.

plots can be seen in Figure 5.23. We can see that all three methods that use the GNN features have better performance on instances with shorter running times. we can also see many instances that have timeout when using only the GNN features but are solved when using the SATZilla features.

Results for PAR_{10} without features costs

The closed gap values of the four feature sets optimised for $PAR_1 0$ without features costs can be seen in Table 5.10. We can see that the graph neural network features alone have positive closed gap values in all scenarios except CSSC-K3. Using the graph neural network features with the SATZilla features results in better performance on more than half of the scenarios. This is similar to other PAR scores, when not using the features extraction time results in better closed gap values for the features sets when using the graph neural network features.

Using AutoRank the Friedman test found statistical significance between the performance differences

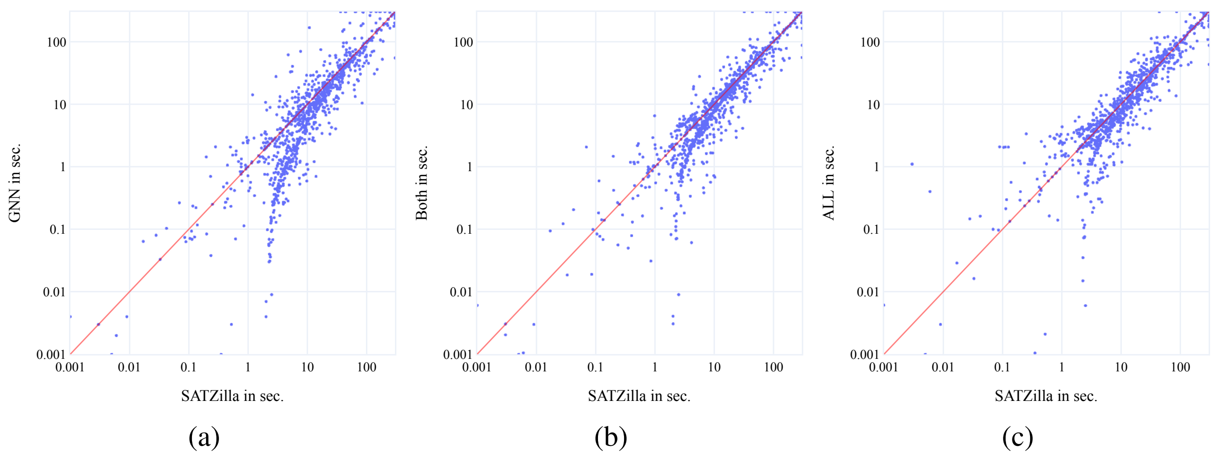


Figure 5.23: Scatter plots of the running times of SATZilla, GNN, Both and all features on the CSSC-BMC08 scenario (with feature costs), optimised for PAR_{10}.

Scenario	AutoFolio SATzilla	AutoFolio GNN	AutoFolio Both	AutoFolio All
SAT11-HAND	**0.77**	0.69	**0.77**	0.76
SAT11-RAND	**0.94**	0.77	**0.94**	**0.94**
SAT11-INDU	**0.47**	0.08	0.42	0.24
SAT18-EXP	**0.63**	0.27	0.52	0.6
SAT20-MAIN	0.16	0.05	0.15	**0.32**
CSSC-BMC08	0.15	0.04	0.25	**0.26**
CSSC-IBM	0.94	0.93	0.94	**0.97**
CSSC-K3	0.22	-0.56	0.18	**0.86**

Table 5.10: Closed gap values of AutoFolio on various algorithm selection scenarios for PAR_{10}. SATZilla represents only using the SATZilla features, GNN represents using only the graph neural network features, Both represent when using the SATZilla features and the graph neural network features for PAR_{10}, All represents when using the SATZilla features and the graph neural network features for all PAR scores (*i.e.* all available features). Bold denotes the maximal value for the scenario.

of the methods. The post hoc Nemenyi test found no significance between Both, SATzilla, and All.

In Figure 5.22, we can see scatter plots of the running times of the instances when using the four different instance sets on the SAT11-HAND scenario. The pattern is similar to other PAR scores, where the points are scattered around. Using only GNN features results in more timeouts. For the BMC08 scenario, the scatter plots can be seen in Figure 5.23. Using only the GNN methods results in more instances with higher running times. For the other two groups, the points are scattered around the diagonal line.

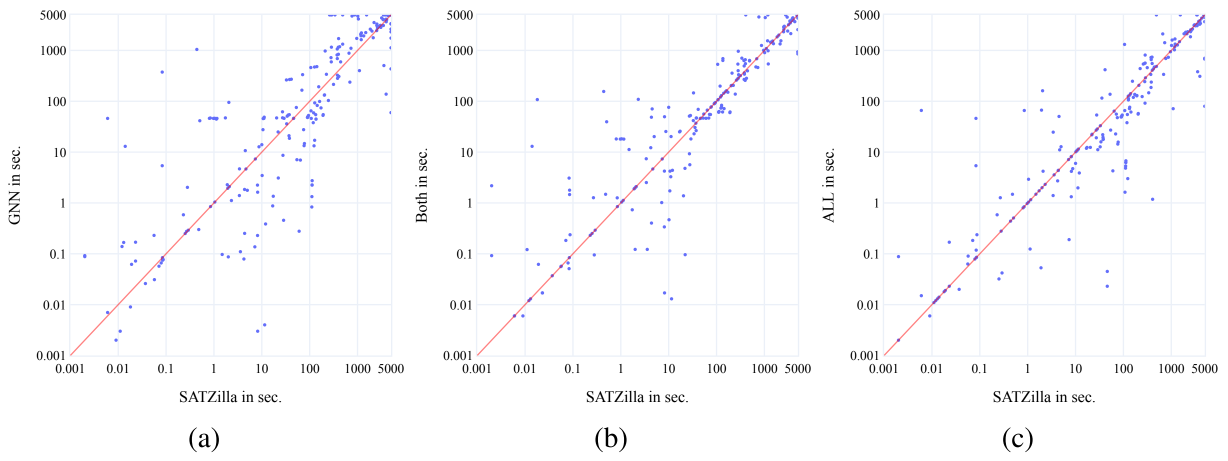


Figure 5.24: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT11-HAND scenario (without feature costs), optimised for PAR_{10}.

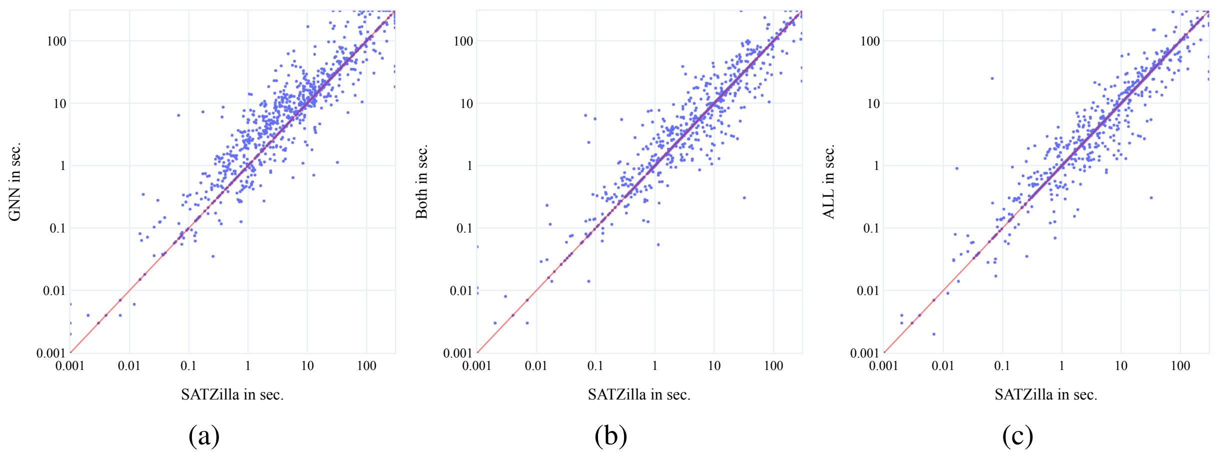


Figure 5.25: Scatter plots of the running times of SATZilla, GNN, Both and all features on the CSSC-BMC08 scenario (without feature costs), optimised for PAR_{10}.

Chapter 6

Discussion, conclusions and future work

In this final chapter, we start in Section 6.1 by discussing the work performed in this thesis, including a reflection and the limitations. We then describe future work and conclude the thesis in Section 6.2.

6.1 Discussion and reflection

This thesis starts with features-free algorithm selection. In the early stage of the work, we found that most existing algorithm selection scenarios include formulas that do not fit into GPU memory. We, therefore, generated a new scenario. We generated the scenario using a well-known industrial-like SAT instance generator. However, many formulas we generated have a very short running time on all solvers (less than a second) or ended as an unsolvable instance. A better way to generate the scenario would be to remove all the unsolvable formulas or have a very short running time. This can allow us to have more "interesting" instances, as the unsolvable instances are not interesting for algorithm selection. In addition, more solvers could be used to generate the scenario.

The main limitation of our proposed features-free approach is the size of the graph and, consequently, the size of the instances we can handle. We can overcome this limitation by using graph sampling approaches such as GraphSAGE (Hamilton et al., 2017). Another method to overcome this limitation is to use a cutting-edge GPU with higher memory capacity.

We then used a graph neural network to extract features from the SAT formulas. Using neural architecture search, we found a good graph neural network architecture for the algorithm selection. Our search space was limited to reduce the computational resources required, as a larger search space requires more evaluations to find a good-performing architecture. For example, we did not fully search for an MLP architecture. We did this due to resource limitations. It is possible that a better architecture exists in the search space that we did not explore.

The running times of the SAT solvers can have high variance. This makes the data noisy, and as observed, high regularisation was used during the training (*i.e.* dropout, large batch size). A possible way to overcome it is to use multiple runs of the same algorithm with different random seeds and use those different runs in the predictions. Neural networks have been shown previously to be able to learn such distribution of SAT solvers' running time (Eggensperger et al., 2018).

Finally, we used the graph neural network features to complement the SATZilla features. We found that the graph neural network features can improve the performance of the algorithm selection, but that the high extraction time required to get the graph neural network features is not worth it. We believe it is possible to reduce the extraction time by optimising the inference of the graph neural network (Jeong et al., 2022).

We also extracted the features using a modified graph neural network. We performed the modification manually after the neural architecture search. We could change the neural architecture search to search for a graph neural network with a bottleneck to extract fewer features. We could also perform a multi-objective neural architecture search to optimise the accuracy and the number of extracted features.

6.2 Conclusions and furture work

Algorithm selection of SAT solvers is an important problem. In this thesis, we looked into using graph-neural networks for algorithm selection in SAT. We started by performing features-free algorithm selection and reached comparable results to features-based algorithm selection. We found that our method outperforms the other features-free method. We believe that this is because our method uses a graph representation of the SAT formula, which is a more natural representation as it is permutations invariant to permutations of the variables and the clauses.

Following features-free algorithm selection, we looked into combining the graph neural network with existing hand-crafted features to improve algorithm selection. We explored both a neural network that uses the graph representation and hand-crafted features. We also extracted the features from the graph neural network and used them together with the hand-crafted features in a known algorithm selection framework. We found that the graph neural network features can improve the performance of the algorithm selection. However, the high extraction time required to get the graph neural network features makes it not worth it for now. It is possible to reduce the extraction time by optimising the inference of the graph neural network (Jeong et al., 2022). This can make it more practical to use the graph neural network features.

We further showed that the graph neural network features are meaningful in scenarios that the graph neural network was not trained on. This shows that our learned features are transferable to other instance distributions, which can imply that the graph neural network captures information about the general structure of SAT formulas. In future work, it is possible to explore training on other scenarios that contain large formulas to possibly improve the performance of the algorithm selection

Although our combined neural network did not significantly outperform the state-of-the-art algorithm selection method, we believe it is possible to improve the performance by using multi-modal learning techniques (Gat et al., 2020). This is because the graph neural network has more parameters than the MLP used to process the SATZilla features. This makes the training unbalanced and biased towards the SATZilla features, which are easier to learn from.

Our algorithm selection framework was exclusively utilized for SAT. Nevertheless, we believe using the frameworks for other $\mathcal{NP}$-Hard problems, such as the travelling-salesperson problem (TSP) and mixed integer programming (MIP), is possible.

We observed that the neural networks can reach a closed gap similar to AutoFolio on PAR_1 and PAR_2. However, for PAR_{10} the performance is worse. This can be attributed to the features that the graph neural network extracts, which might be less informative about timeouts, as the combined neural network performed very well on this metric. In future work, it is possible to explore different message-passing layers that will capture different features.

To conclude, we believe that graph neural networks are a promising direction for algorithm selection of SAT solvers. We introduced the first features-free algorithm selection method that can reach comparable results to the well-studied features-based algorithm selection approaches. Additionally, it was demonstrated that the graph neural network features can enhance the performance of the state-of-the-art algorithm selection method.

Acknowledgements

We thank the fruitful discussion and suggestions from Frank Hutter, Lennart Purucker, Thijs Simons and Kevin Tierney. This work was (partially) performed using the compute resources from the Academic Leiden Interdisciplinary Cluster Environment (ALICE) provided by Leiden University. This work also used the GRACE cluster provided by the Leiden Institute of Advanced Computer Science (LIACS).

Bibliography

Amadini, R., Gabbrielli, M., Liu, T., and Mauro, J. (2023). On the Evaluation of (Meta-)solver Approaches. *Journal of Artificial Intelligence Research*, 76:705–719.

Ansótegui, C., Bonet, M. L., Giráldez-Cru, J., Levy, J., and Simon, L. (2019). Community Structure in Industrial SAT Instances. *Journal of Artificial Intelligence Research*, 66:443–472.

Audemard, G. and Simon, L. (2009). Predicting learnt clauses quality in modern SAT solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*, pages 399–404.

Babic, D. and Hutter, F. (2007). Spear theorem prover. *Proceedings of the 2007 SAT competition*, 2007.

Balint, A., Fröhlich, A., Tompkins, D., and Hoos, H. (2011). Sparrow 2011.

Balint, A. and Manthey, N. (2014). SparrowToRiss. *SAT Competition 2014: Solver and Benchmark Descriptions*, pages 77–78.

Balyo, T., Froleyks, N., Heule, M. J., Iser, M., Järvisalo, M., and Suda, M. (2022). Proceedings of SAT Competition 2022 : Solver and Benchmark Descriptions.

Barron, J. (2017). Continuously Differentiable Exponential Linear Units. *arXiv:1511.07289*.

Biedenkapp, A., Marben, J., Lindauer, M., and Hutter, F. (2019). CAVE: Configuration Assessment, Visualization and Evaluation. In Battiti, R., Brunato, M., Kotsireas, I., and Pardalos, P. M., editors, *Learning and Intelligent Optimization*, Lecture Notes in Computer Science, pages 115–130, Cham. Springer International Publishing.

Biere, A., Fazekas, K., Fleury, M., and Heisinger, M. (2020). CADICAL, KISSAT, PARACOOBA, PLINGELING and TREENGELING Entering the SAT Competition 2020. In *Proceedings of SAT Competition 2020 : Solver and Benchmark Descriptions*. Department of Computer Science, University of Helsinki.

Biere, A., J&#228, Rvisalo, M., and Kiesl, B. (2021). Chapter 9. Preprocessing in SAT Solving. In *Handbook of Satisfiability*, pages 391–435. IOS Press.

Bischl, B., Kerschke, P., Kotthoff, L., Lindauer, M., Malitsky, Y., Fréchette, A., Hoos, H., Hutter, F., Leyton-Brown, K., Tierney, K., and Vanschoren, J. (2016). ASlib: A benchmark library for algorithm selection. *Artificial Intelligence*, 237:41–58.

Bronstein, M. M., Bruna, J., Cohen, T., and Veličković, P. (2021). Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges.

Clarke, E., Biere, A., Raimi, R., and Zhu, Y. (2001). Bounded model checking using satisfiability solving. *Formal methods in system design*, 19:7–34.

Clevert, D.-A., Unterthiner, T., and Hochreiter, S. (2015). Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). *Proceedings of the 4th International Conference on Learning Representations (ICLR).*

Collautti, M., Malitsky, Y., Mehta, D., and O'Sullivan, B. (2013). SNNAP: Solver-Based Nearest Neighbor for Algorithm Portfolios. In Blockeel, H., Kersting, K., Nijssen, S., and Železný, F., editors, *Proceedings of the 12th Joint European Conference on Machine Learning and Knowledge Discovery in Databases (ECML/PKDD)*, pages 435–450.

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC)*, pages 151–158.

Darwiche, A. and Pipatsrisawat, K. (2021). Chapter 3. Complete Algorithms. In *Handbook of Satisfiability*, pages 101–132. IOS Press.

Davis, M., Logemann, G., and Loveland, D. (1962). A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397.

Davis, M. and Putnam, H. (1960). A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215.

Eén, N. and Biere, A. (2005). Effective Preprocessing in SAT Through Variable and Clause Elimination. In Bacchus, F. and Walsh, T., editors, *Proceedings of the 8th International Conference onTheory and Applications of Satisfiability Testing (SAT)*, pages 61–75.

Eén, N. and Sörensson, N. (2004). An Extensible SAT-solver. In Giunchiglia, E. and Tacchella, A., editors, *Proceedings of the 4th International Conference onTheory and Applications of Satisfiability Testing (SAT)*, pages 502–518.

Eggensperger, K., Lindauer, M., and Hutter, F. (2018). Neural Networks for Predicting Algorithm Runtime Distributions. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, pages 1442–1448, Stockholm, Sweden. International Joint Conferences on Artificial Intelligence Organization.

Erickson, N., Mueller, J., Shirkov, A., Zhang, H., Larroy, P., Li, M., and Smola, A. (2020). AutoGluon-Tabular: Robust and Accurate AutoML for Structured Data.

Falkner, S., Klein, A., and Hutter, F. (2018). BOHB: Robust and efficient hyperparameter optimization at scale. In *International Conference on Machine Learning*, pages 1437–1446. PMLR.

Fichte, J. K., Berre, D. L., Hecher, M., and Szeider, S. (2023). The Silent (R)evolution of SAT. *Communications of the ACM*, 66(6):64–72.

Franco, J. and Martin, J. (2021). Chapter 1. A History of Satisfiability. In *Handbook of Satisfiability*, pages 3–74. IOS Press.

Froleyks, N., Heule, M., Iser, M., Järvisalo, M., and Suda, M. (2021). SAT Competition 2020. *Artificial Intelligence*, 301:103572.

Gat, I., Schwartz, I., Schwing, A., and Hazan, T. (2020). Removing bias in multi-modal classifiers: Regularization by maximizing functional entropies. *Advances in Neural Information Processing Systems*, 33:3197–3208.

Giráldez-Cru, J. and Levy, J. (2016). Generating SAT instances with community structure. *Artificial Intelligence*, 238:119–134.

Gonard, F., Schoenauer, M., and Sebag, M. (2017). ASAP.V2 and ASAP.V3: Sequential optimization of an Algorithm Selector and a Scheduler. In *Proceedings of the Open Algorithm Selection Challenge*, pages 8–11. PMLR.

Hamilton, W., Ying, Z., and Leskovec, J. (2017). Inductive Representation Learning on Large Graphs. *Proceedings of the 31st International Conference on Neural Information Processing Systems (NeurIPS)*.

Han, J. M. (2020). Enhancing SAT solvers with glue variable predictions. *ArXiv*.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778. IEEE Computer Society.

Hendrycks, D. and Gimpel, K. (2016). Bridging Nonlinearities and Stochastic Regularizers with Gaussian Error Linear Units. *arXiv:1607.06450*.

Herbold, S. (2020). Autorank: A Python package for automated ranking of classifiers. *Journal of Open Source Software*, 5(48):2173.

Hoos, H. H. and Stützle, T. (2000). SATLIB: An Online Resource for Research on SAT. In *SAT 2000*, pages 283–292. IOS Press.

Hsu, E. I., Muise, C. J., McIlraith, S. A., and Beck, J. C. (2008). Applying probabilistic inference to heuristic search by estimating variable bias. In *Proceedings of the 1st International Symposium on Search Techniques in Artificial Intelligence and Robotics (at AAAI 2008), Chicago, IL, USA, July*, pages 13–14. Citeseer.

Huang, G., Liu, Z., Maaten, L. V. D., and Weinberger, K. Q. (2017). Densely Connected Convolutional Networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2261–2269. IEEE Computer Society.

Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2011). Sequential Model-Based Optimization for General Algorithm Configuration. In Coello, C. A. C., editor, *Learning and Intelligent Optimization*, Lecture Notes in Computer Science, pages 507–523, Berlin, Heidelberg. Springer.

Hutter, F., Lindauer, M., Balint, A., Bayless, S., Hoos, H., and Leyton-Brown, K. (2017). The Configurable SAT Solver Challenge (CSSC). *Artificial Intelligence*, 243:1–25.

Hutter, F., López-Ibánez, M., Fawcett, C., Lindauer, M., Hoos, H. H., Leyton-Brown, K., and Stützle, T. (2014a). AClib: A benchmark library for algorithm configuration. In *Learning and Intelligent Optimization: 8th International Conference, Lion 8, Gainesville, FL, USA, February 16-21, 2014. Revised Selected Papers 8*, pages 36–40. Springer.

Hutter, F., Xu, L., Hoos, H. H., and Leyton-Brown, K. (2014b). Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence*, 206:79–111.

Jeong, E., Kim, J., and Ha, S. (2022). TensorRT-Based Framework and Optimization Methodology for Deep Learning Inference on Jetson Boards. *ACM Transactions on Embedded Computing Systems*, 21(5):51:1–51:26.

Johnson, D. J. and Trick, M. A. (1996). *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, Workshop, October 11-13, 1993*. American Mathematical Society, USA.

Kadioglu, S., Malitsky, Y., Sabharwal, A., Samulowitz, H., and Sellmann, M. (2011). Algorithm selection and scheduling. In *Principles and Practice of Constraint Programming–CP 2011: 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings 17*, pages 454–469. Springer.

Karp, R. M. (1972). Reducibility among Combinatorial Problems. In Miller, R. E., Thatcher, J. W., and Bohlinger, J. D., editors, *Complexity of Computer Computations*, pages 85–103. Springer US, Boston, MA.

Kerschke, P., Kotthoff, L., Bossek, J., Hoos, H. H., and Trautmann, H. (2018). Leveraging TSP Solver Complementarity through Machine Learning. *Evolutionary Computation*, 26(4):597–620.

Khasahmadi, A. H., Hassani, K., Moradi, P., Lee, L., and Morris, Q. (2019). Memory-Based Graph Networks. In *Proceedings of the 7th International Conference on Learning Representations (ICLR)*.

Kipf, T. N. and Welling, M. (2017). Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the 5th International Conference on Learning Representations (ICLR)*.

Kotthoff, L., Hurley, B., and O'Sullivan, B. (2017). The ICON Challenge on Algorithm Selection. *AI Magazine*, 38(2):91–93.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 26th International Conference on Neural Information Processing Systems (NeurIPS)*.

Krom, M. R. (1967). The Decision Problem for a Class of First-Order Formulas in Which all Disjunctions are Binary. *Mathematical Logic Quarterly*, 13(1-2):15–20.

Kurin, V., Godil, S., Whiteson, S., and Catanzaro, B. (2020). Can Q-learning with graph networks learn a generalizable branching heuristic for a SAT solver? In *Proceedings of the 34th International Conference on Neural Information Processing Systems (NeurIPS)*, pages 9608–9621.

Leeson, W. and Dwyer, M. B. (2022). Graves-CPA: A Graph-Attention Verifier Selector (Competition Contribution). In Fisman, D. and Rosu, G., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 440–445, Cham. Springer International Publishing.

Leyton-Brown, K., Nudelman, E., Andrew, G., McFadden, J., and Shoham, Y. (2003). Boosting as a Metaphor for Algorithm Design. In Rossi, F., editor, *Principles and Practice of Constraint Programming – CP 2003*, Lecture Notes in Computer Science, pages 899–903, Berlin, Heidelberg. Springer.

Li, G., Xiong, C., Qian, G., Thabet, A., and Ghanem, B. (2021). DeeperGCN: All You Need to Train Deeper GCNs.

Li, Z. and Si, X. (2022). NSNet: A General Neural Probabilistic Framework for Satisfiability Problems. In *Proceedings of the 35th International Conference on Neural Information Processing Systems (NeurIPS)*.

Lindauer, M., Eggensperger, K., Feurer, M., Biedenkapp, A., Deng, D., Benjamins, C., Ruhkopf, T., Sass, R., and Hutter, F. (2022). SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization. *Journal of Machine Learning Research*, 23(54):1–9.

Lindauer, M., Hoos, H. H., Hutter, F., and Schaub, T. (2015). AutoFolio: An Automatically Configured Algorithm Selector. *Journal of Artificial Intelligence Research*, 53:745–778.

Lindauer, M., van Rijn, J. N., and Kotthoff, L. (2017). Open Algorithm Selection Challenge 2017: Setup and Scenarios. In *Proceedings of the Open Algorithm Selection Challenge*, pages 1–7. PMLR.

Lindauer, M., van Rijn, J. N., and Kotthoff, L. (2019). The algorithm selection competitions 2015 and 2017. *Artificial Intelligence*, 272:86–100.

Liu, T., Amadini, R., Gabbrielli, M., and Mauro, J. (2022). Sunny-as2: Enhancing SUNNY for Algorithm Selection. *Journal of Artificial Intelligence Research*, 72:329–376.

Loreggia, A., Malitsky, Y., Samulowitz, H., and Saraswat, V. (2016). Deep Learning for Algorithm Portfolios. *Proceedings of the AAAI Conference on Artificial Intelligence*, 30(1).

Loshchilov, I. and Hutter, F. (2018). Decoupled Weight Decay Regularization. In *Proceedings of the 6th International Conference on Learning Representations (ICLR)*.

Luo, C. and Hoos, H. H. (2018). Sparkle SAT Challenge 2018.

Maas, A. L., Hannun, A. Y., and Ng, A. Y. (2013). Rectifier Nonlinearities Improve Neural Network Acoustic Models. In *ICML Workshop on Deep Learning for Audio, Speech, and Language Processing (WDLASL)*.

Malitsky, Y., Sabharwal, A., Samulowitz, H., and Sellmann, M. (2013). Boosting Sequential Solver Portfolios: Knowledge Sharing and Accuracy Prediction. In Nicosia, G. and Pardalos, P., editors, *Learning and Intelligent Optimization*, Lecture Notes in Computer Science, pages 153–167, Berlin, Heidelberg. Springer.

Malone, B., Kangas, K., Järvisalo, M., Koivisto, M., and Myllymäki, P. (2017). AS-ASL: Algorithm Selection with Auto-sklearn. In *Proceedings of the Open Algorithm Selection Challenge*, pages 19–22. PMLR.

Manthey, N. (2010). Riss 2010 Solver Description.

Marques Silva, J. P. and Sakallah, K. A. (1997). GRASP—a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '96, pages 220–227, USA. IEEE Computer Society.

Morris, C., Ritzert, M., Fey, M., Hamilton, W. L., Lenssen, J. E., Rattan, G., and Grohe, M. (2019). Weisfeiler and leman go neural: Higher-order graph neural networks. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence*, AAAI'19/IAAI'19/EAAI'19, pages 4602–4609, Honolulu, Hawaii, USA. AAAI Press.

Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., and Malik, S. (2001). Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pages 530–535.

Nair, V. and Hinton, G. E. (2010). Rectified Linear Units Improve Restricted Boltzmann Machines. In Fürnkranz, J. and Joachims, T., editors, *Proceedings of the 27th International Conference on Machine Learning (ICML)*, pages 807–814. Omnipress.

Prestwich, S. (2021). Chapter 2. CNF Encodings. In *Handbook of Satisfiability*, pages 75–100. IOS Press.

Pulatov, D., Anastacio, M., Kotthoff, L., and Hoos, H. (2022). Opening the Black Box: Automated Software Analysis for Algorithm Selection. In *First Conference on Automated Machine Learning (Main Track)*.

Rice, J. R. (1976). The Algorithm Selection Problem. In Rubinoff, M. and Yovits, M. C., editors, *Advances in Computers*, volume 15, pages 65–118. Elsevier.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. (2015). ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3):211–252.

Sass, R., Bergman, E., Biedenkapp, A., Hutter, F., and Lindauer, M. (2022). DeepCAVE: An Interactive Analysis Tool for Automated Machine Learning.

Seiler, M., Pohl, J., Bossek, J., Kerschke, P., and Trautmann, H. (2020). Deep Learning as a Competitive Feature-Free Approach for Automated Algorithm Selection on the Traveling Salesperson Problem. In Bäck, T., Preuss, M., Deutz, A., Wang, H., Doerr, C., Emmerich, M., and Trautmann, H., editors, *Parallel Problem Solving from Nature – PPSN XVI*, Lecture Notes in Computer Science, pages 48–64, Cham. Springer International Publishing.

Selman, B., Kautz, H. A., and Cohen, B. (1993). Local search strategies for satisfiability testing. *Cliques, coloring, and satisfiability*, 26:521–532.

Selsam, D. and Bjørner, N. (2019). Guiding High-Performance SAT Solvers with Unsat-Core Predictions. In Janota, M. and Lynce, I., editors, *Theory and Applications of Satisfiability Testing – SAT 2019*, Lecture Notes in Computer Science, pages 336–353, Cham. Springer International Publishing.

Selsam, D., Lamm, M., Bünz, B., Liang, P., de Moura, L., and Dill, D. L. (2018). Learning a SAT Solver from Single-Bit Supervision. In *Proceedings of the 6th International Conference on Learning Representations (ICLR)*.

Sipser, M. (2012). *Introduction to the Theory of Computation*. Cengage Learning, Australia Brazil Japan Korea Mexiko Singapore Spain United Kingdom United States, 3rd edition edition.

Soos, M., Nohl, K., and Castelluccia, C. (2009). Extending SAT Solvers to Cryptographic Problems. In Kullmann, O., editor, *Proceedings of the 12th International Conference onTheory and Applications of Satisfiability Testing (SAT 2009)*, pages 244–257.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., and Bengio, Y. (2018). Graph Attention Networks. In *Proceedings of the 6th International Conference on Learning Representations (ICLR)*.

Wang, W., Hu, Y., Tiwari, M., Khurshid, S., McMillan, K., and Miikkulainen, R. (2021). NeuroComb: Improving SAT Solving with Graph Neural Networks. *arXiv:2110.14053*.

Xu, K., Hu, W., Leskovec, J., and Jegelka, S. (2018). How Powerful are Graph Neural Networks? In *Proceedings of the 6th International Conference on Learning Representations (ICLR)*.

Xu, L., Hoos, H., and Leyton-Brown, K. (2012a). Predicting Satisfiability at the Phase Transition. *Proceedings of the AAAI Conference on Artificial Intelligence*, 26(1):584–590.

Xu, L., Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2008). SATzilla: Portfolio-based Algorithm Selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606.

Xu, L., Hutter, F., Shen, J., Hoos, H. H., and Leyton-Brown, K. (2012b). SATzilla2012: Improved algorithm selection based on cost-sensitive classification models. *Proceedings of the SAT Challenge 2012*, pages 57–58.

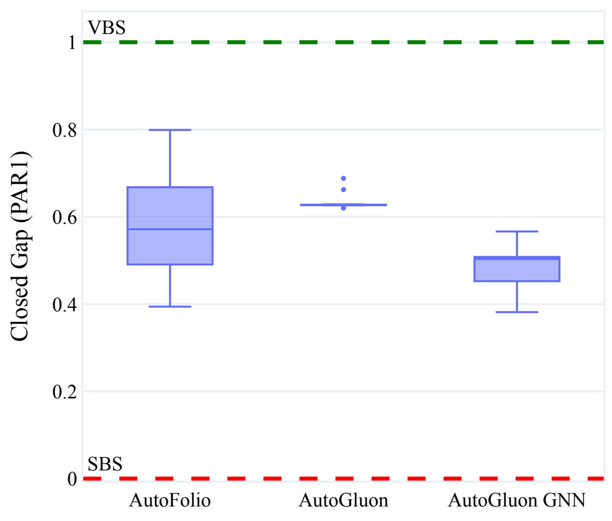
Zhang, L. (2005). On Subsumption Removal and On-the-Fly CNF Simplification. In Bacchus, F. and Walsh, T., editors, *Theory and Applications of Satisfiability Testing*, Lecture Notes in Computer Science, pages 482–489, Berlin, Heidelberg. Springer.

Appendix A

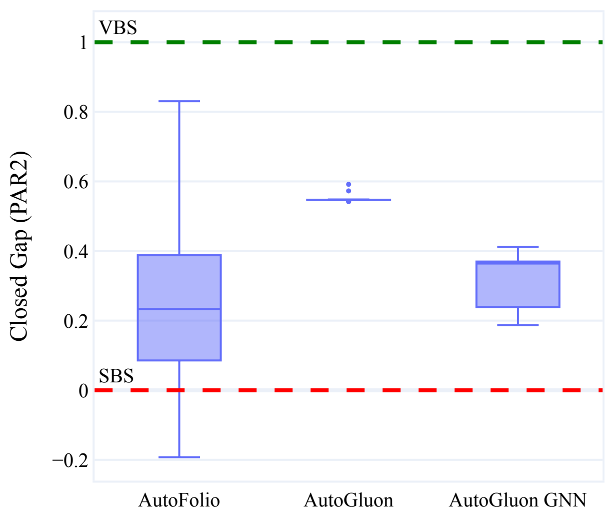
AutoGluon for Algorithm Selection

In addition to experiments with AutoFolio, a state-of-the-art method for algorithm selection, we try to use AutoGluon for algorithm selection. AutoGluon (Erickson et al., 2020) is an AutoML system that can preprocess the input data, select machine learning algorithms to use, and create set of them. It can often achieve better results than other machine learning models. AutoGluon uses various machine learning algorithms, such as XGBoost, random forest, neural networks, and more, in combination with ensembling and various preprocessing approaches. However, AutoGluon is used for general machine learning (i.e. classification or regression tasks) while our goal is to perform algorithm selection. For algorithm selection, multi-class classification is often not the best prediction type (Liu et al., 2022). Inspired by SATZilla 2012 (Xu et al., 2012b), we use AutoGluon to perform pairwise classification. We use one AutoGluon model per pair of solvers and take the solver that gets the most "votes". We also set the samples' weights to be the running times' standard deviation.

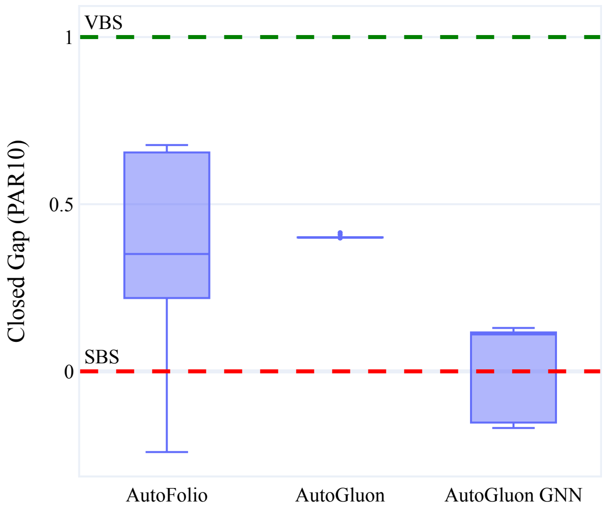
We run AutoGluon on AutoFolio on the features extracted from the network optimised for PAR_1. We extract the features using 24 Intel Xeon Gold 6126 2.6GHz CPU cores and 300 GB of RAM. The closed gap results are presented in Figure A.1. We can see that AutoGluon has the least variance across the different random seeds. On PAR_2 it is the best method. It should be noted that AutoGluon does not have features groups selection, so it cannot remove expensive features groups.



(a)



(b)



(c)

Figure A.1: Closed gap values of AutoGluon and AutoFolio. (a) PAR_1 (b) PAR_2 (c) PAR_{10}.

Appendix B

Full Scatter Plots

In this appendix, we show the scatter plots comparing the running times of the instances on the various AutoFolio variants, as described in Chapter 5.

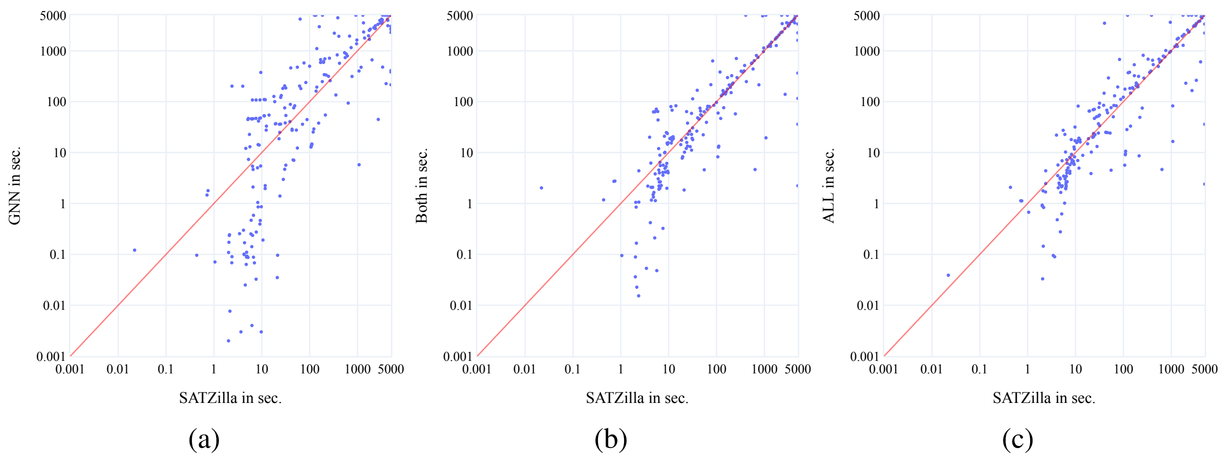


Figure B.1: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT11-HAND scenario (with feature costs), optimised for PAR_1.

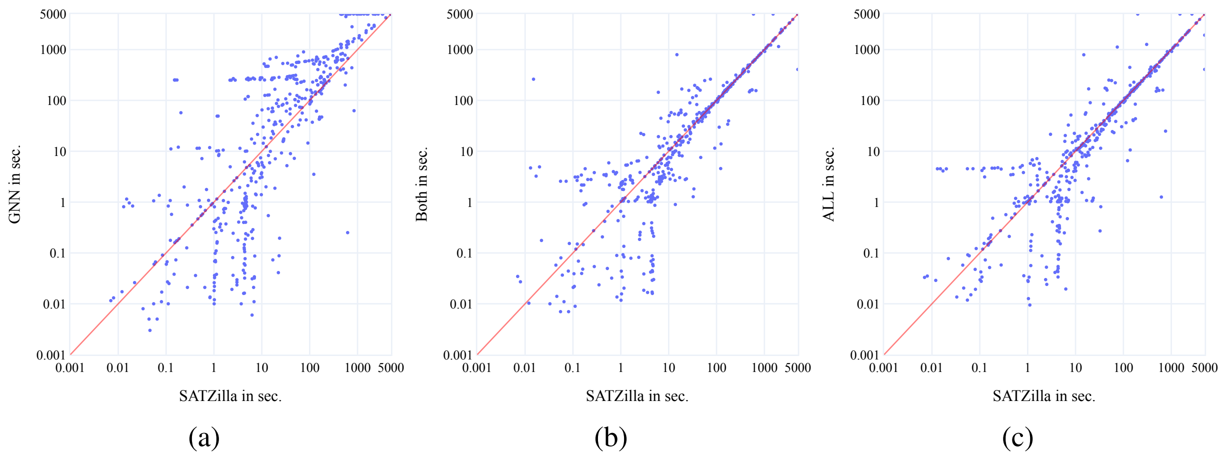


Figure B.2: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT11-RAND scenario (with feature costs), optimised for PAR_1.

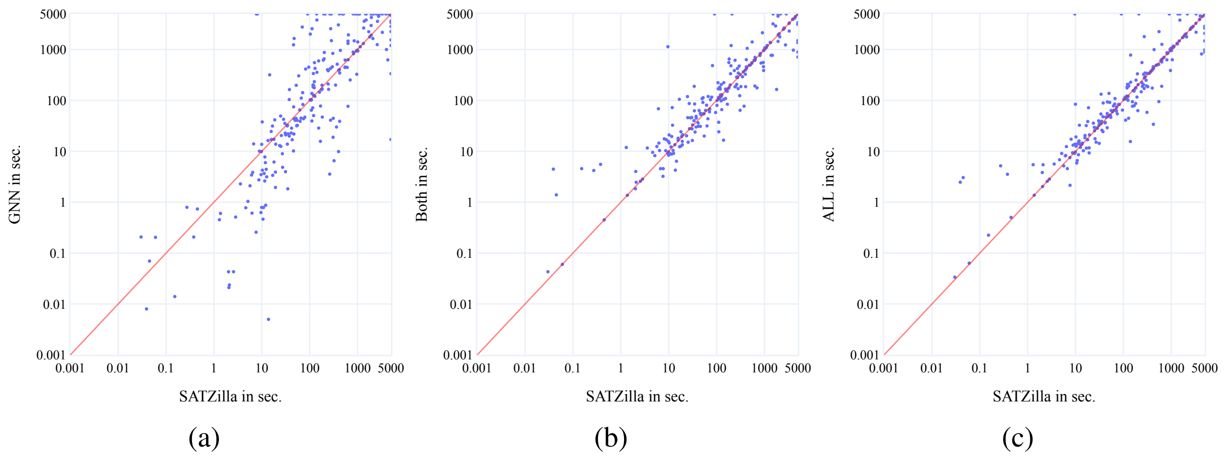
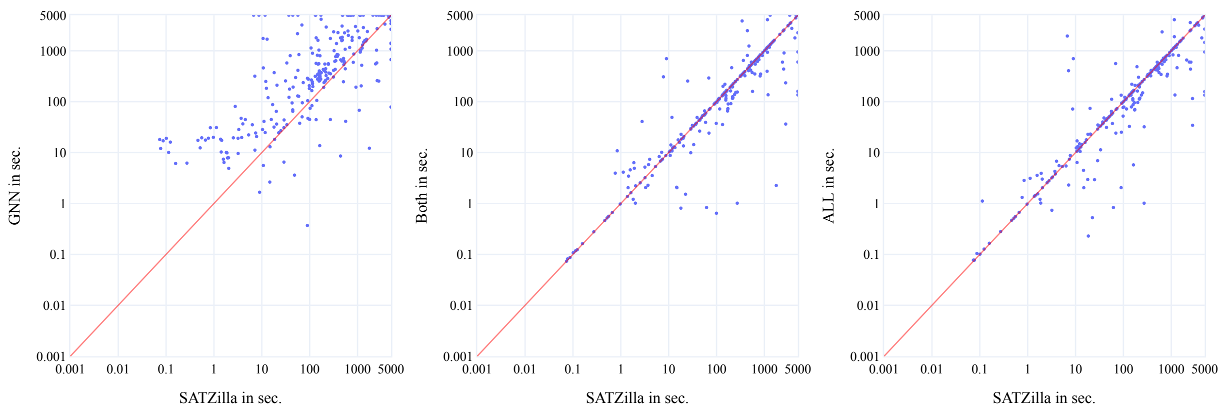
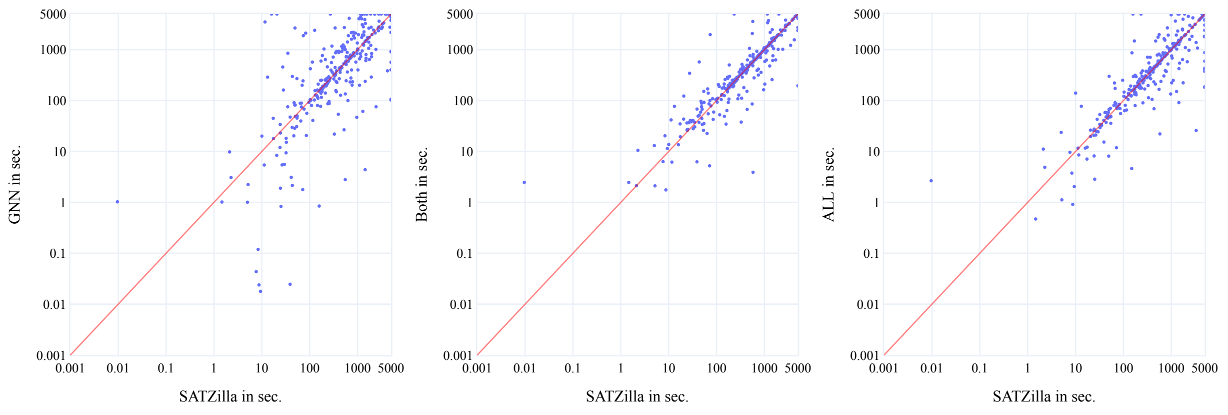


Figure B.3: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT11-INDU scenario (with feature costs), optimised for PAR_1.



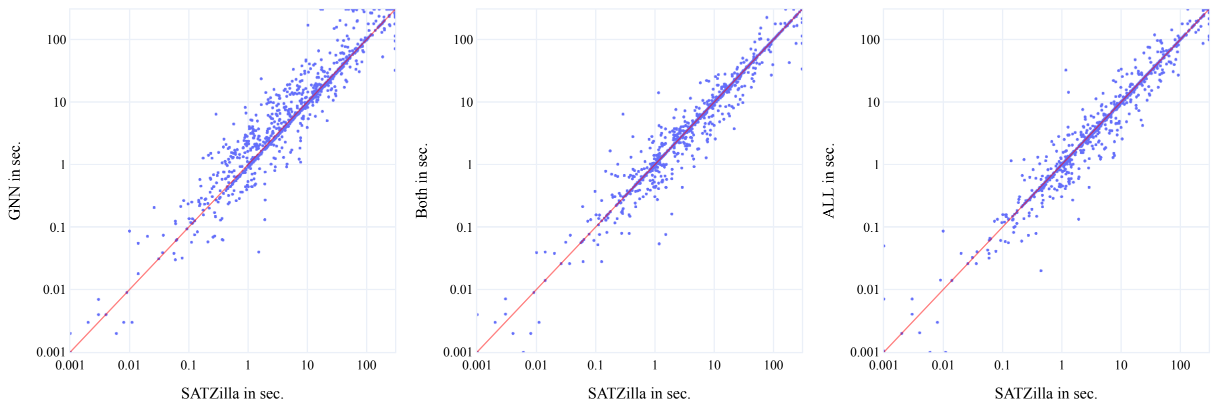
(a) (b) (c)

Figure B.4: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT18-EXP scenario (with feature costs), optimised for PAR$_1$.



(a) (b) (c)

Figure B.5: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT20-MAIN scenario (with feature costs), optimised for PAR$_1$.



(a) (b) (c)

Figure B.6: Scatter plots of the running times of SATZilla, GNN, Both and all features on the CSSC-BMC08 scenario (with feature costs), optimised for PAR$_1$.

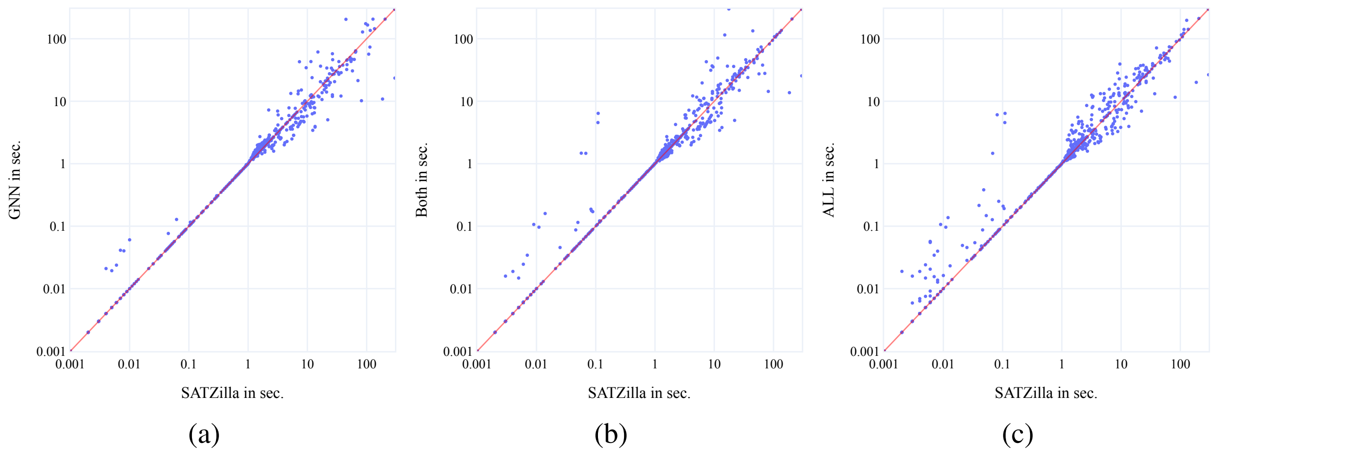


Figure B.7: Scatter plots of the running times of SATZilla, GNN, Both and all features on the CSSC-IBM scenario (with feature costs), optimised for PAR$_1$.

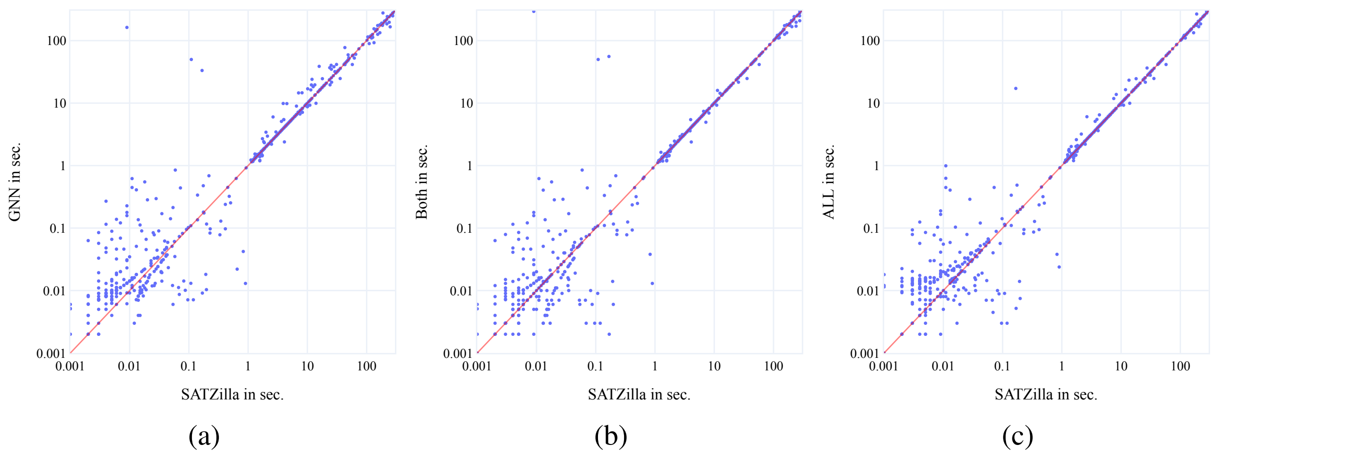


Figure B.8: Scatter plots of the running times of SATZilla, GNN, Both and all features on the CSSC-K3 scenario (with feature costs), optimised for PAR$_1$.

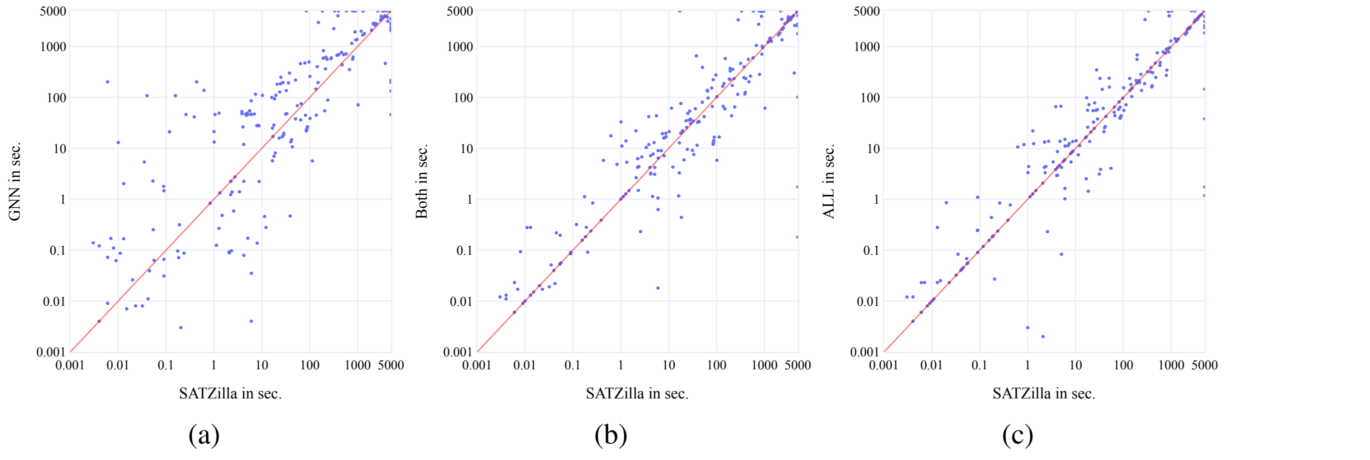


Figure B.9: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT11-HAND scenario (without feature costs), optimised for PAR$_1$.

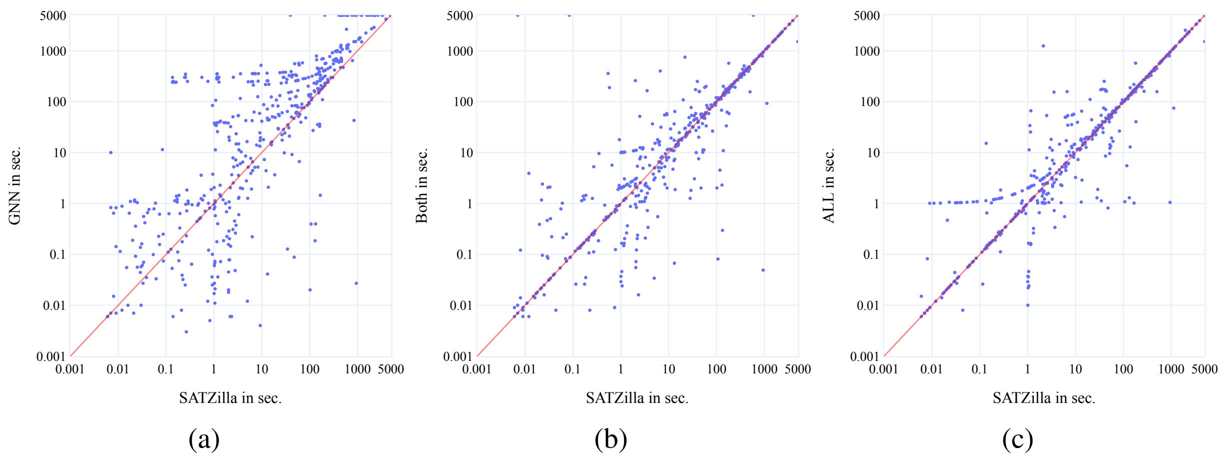


Figure B.10: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT11-RAND scenario (without feature costs), optimised for PAR_1.

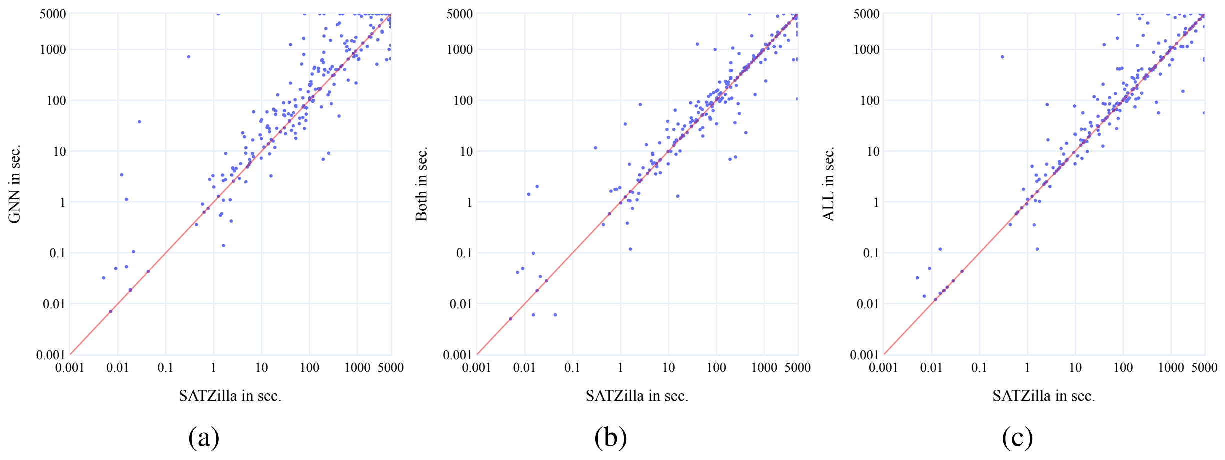


Figure B.11: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT11-INDU scenario (without feature costs), optimised for PAR_1.

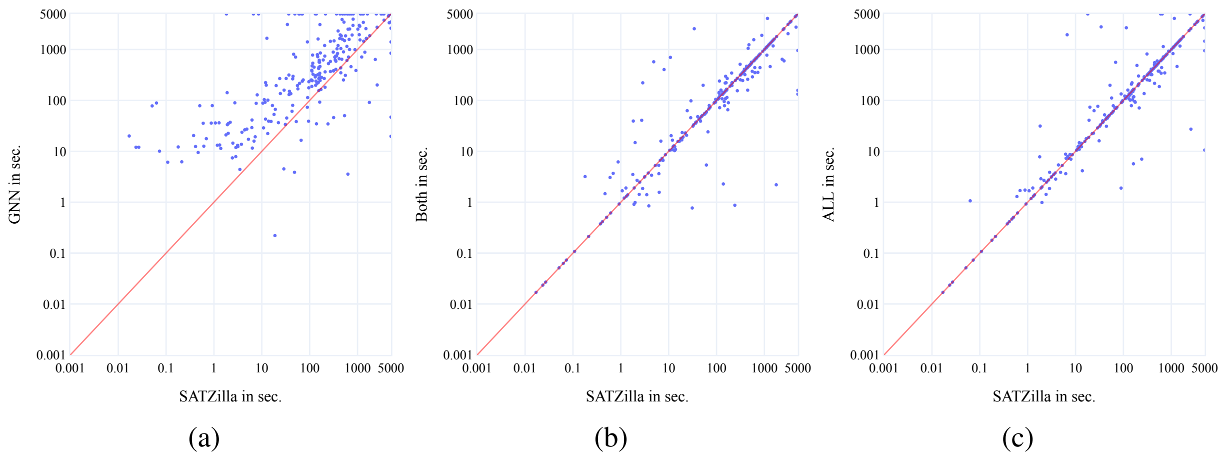


Figure B.12: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT18-EXP scenario (without feature costs), optimised for PAR_1.

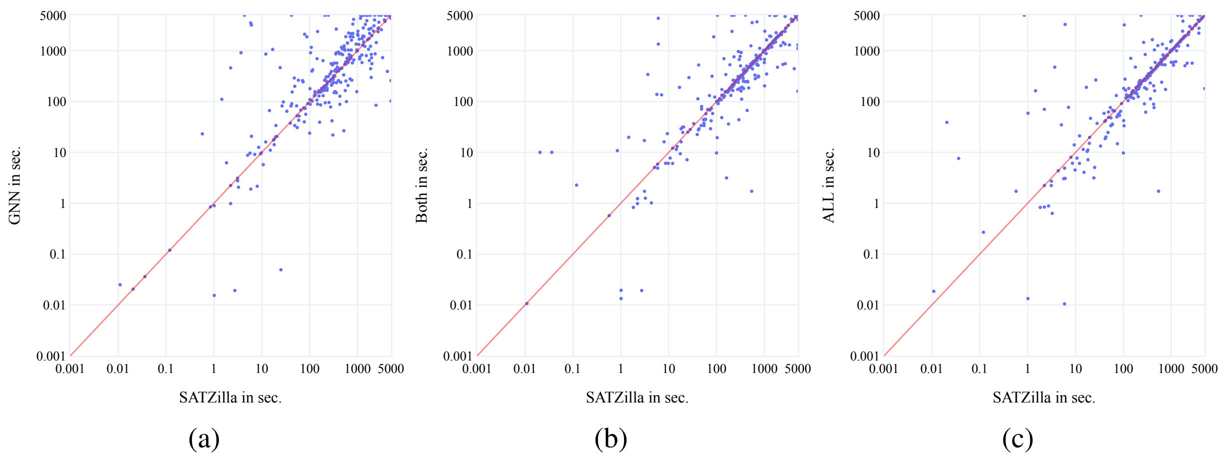


Figure B.13: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT20-MAIN scenario (without feature costs), optimised for PAR$_1$.

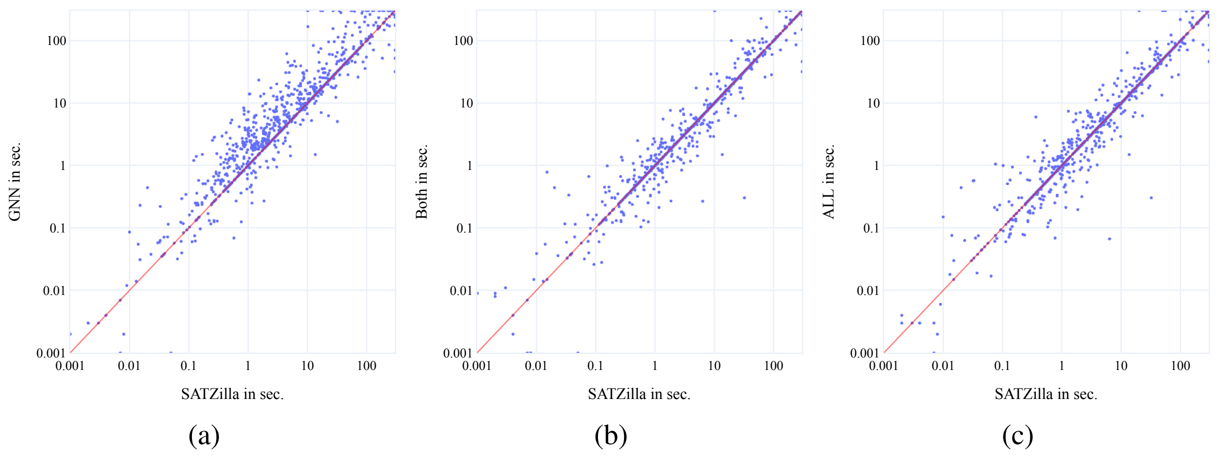


Figure B.14: Scatter plots of the running times of SATZilla, GNN, Both and all features on the CSSC-BMC08 scenario (without feature costs), optimised for PAR$_1$.

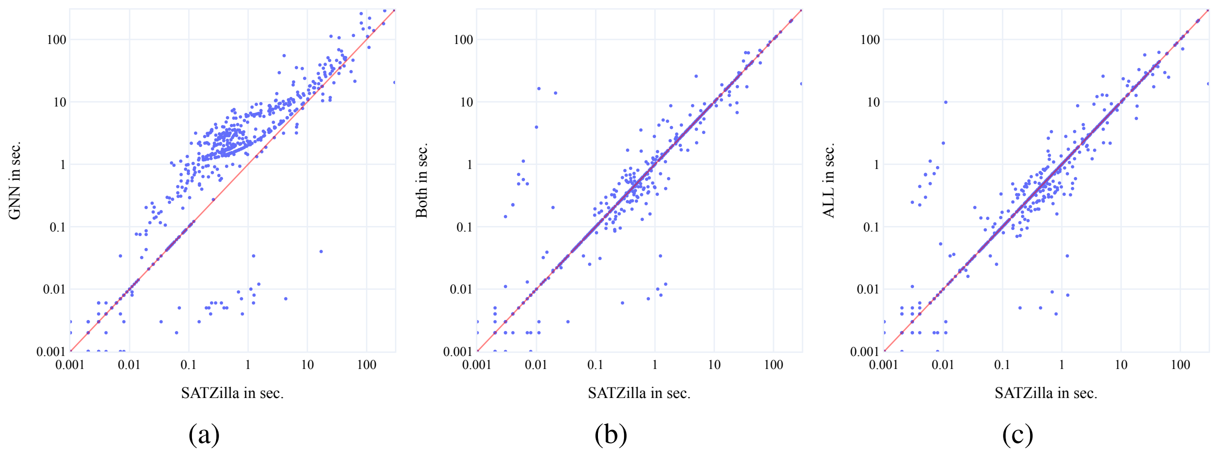
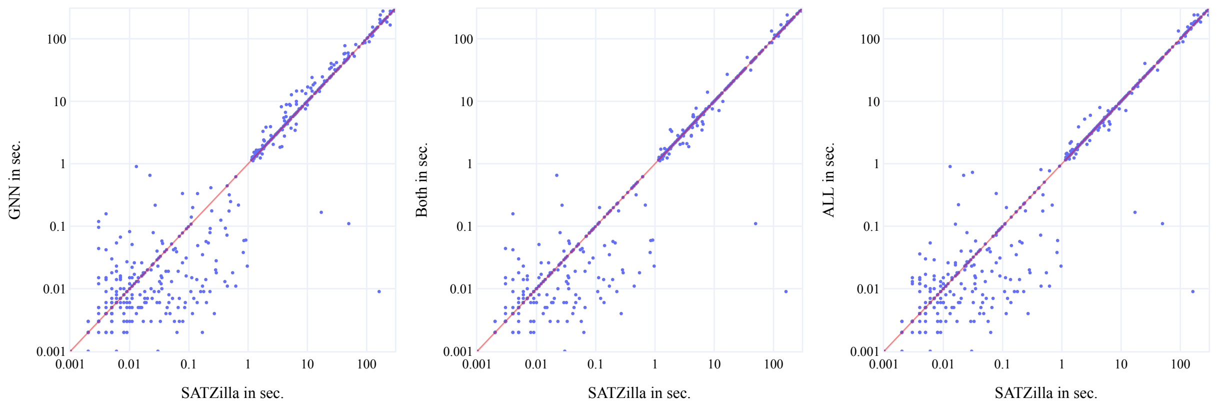
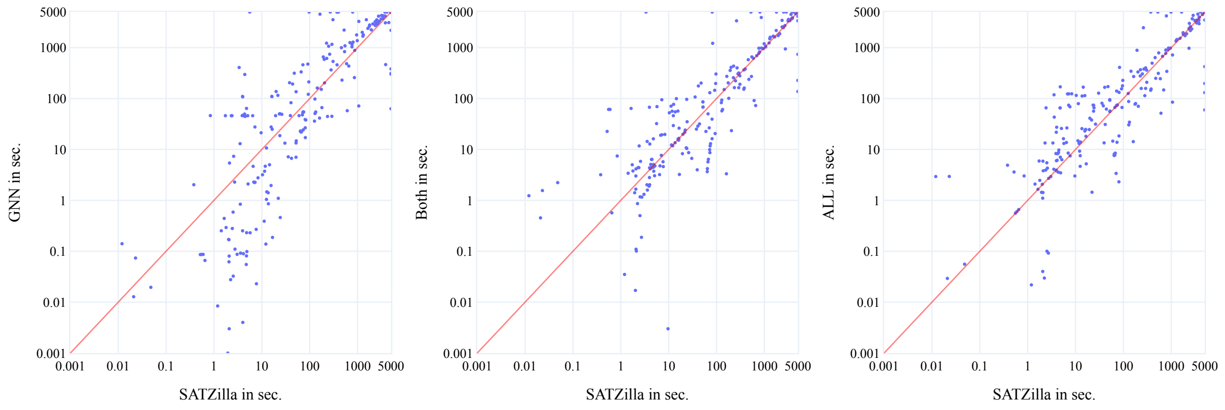


Figure B.15: Scatter plots of the running times of SATZilla, GNN, Both and all features on the CSSC-IBM scenario (without feature costs), optimised for PAR$_1$.



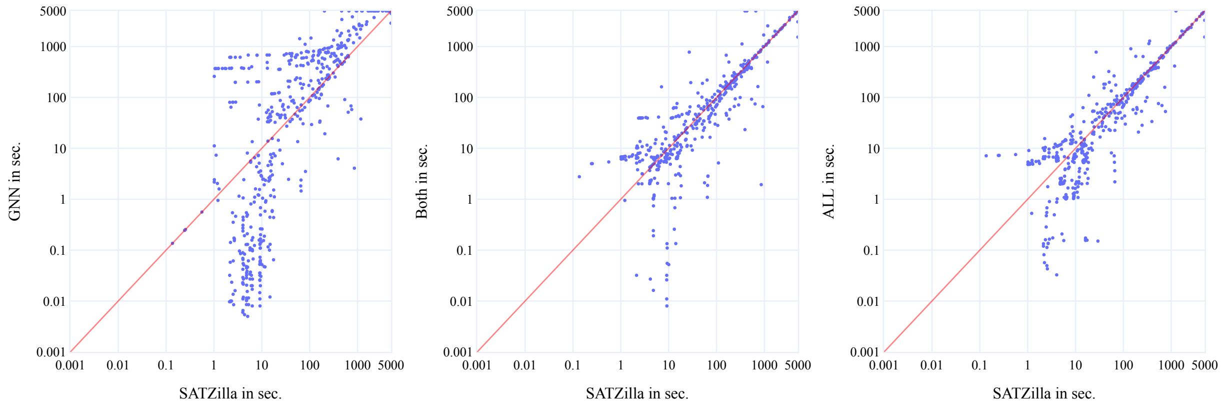
(a) (b) (c)

Figure B.16: Scatter plots of the running times of SATZilla, GNN, Both and all features on the CSSC-K3 scenario (without feature costs), optimised for PAR_1.



(a) (b) (c)

Figure B.17: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT11-HAND scenario (with feature costs), optimised for PAR_2.



(a) (b) (c)

Figure B.18: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT11-RAND scenario (with feature costs), optimised for PAR_2.

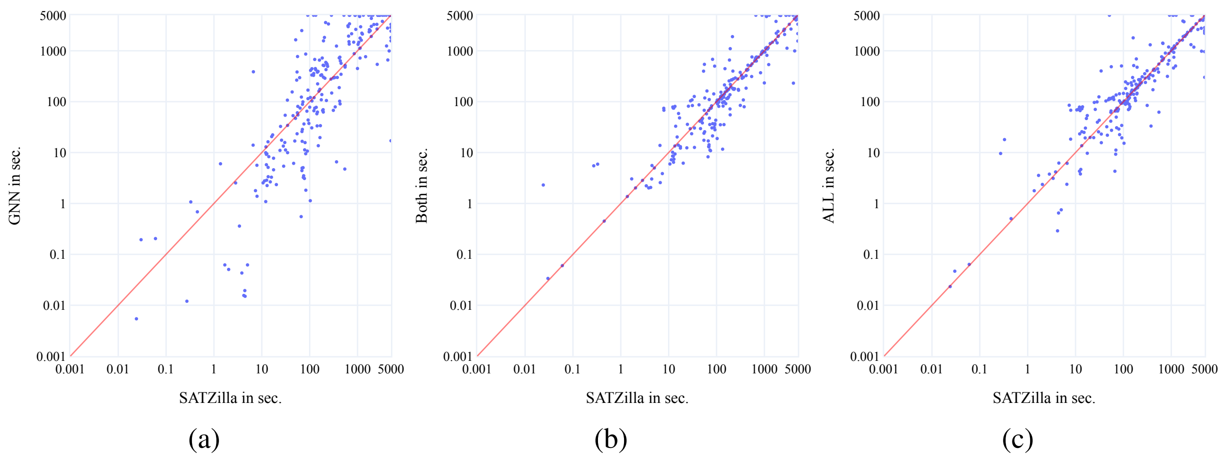


Figure B.19: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT11-INDU scenario (with feature costs), optimised for PAR$_2$.

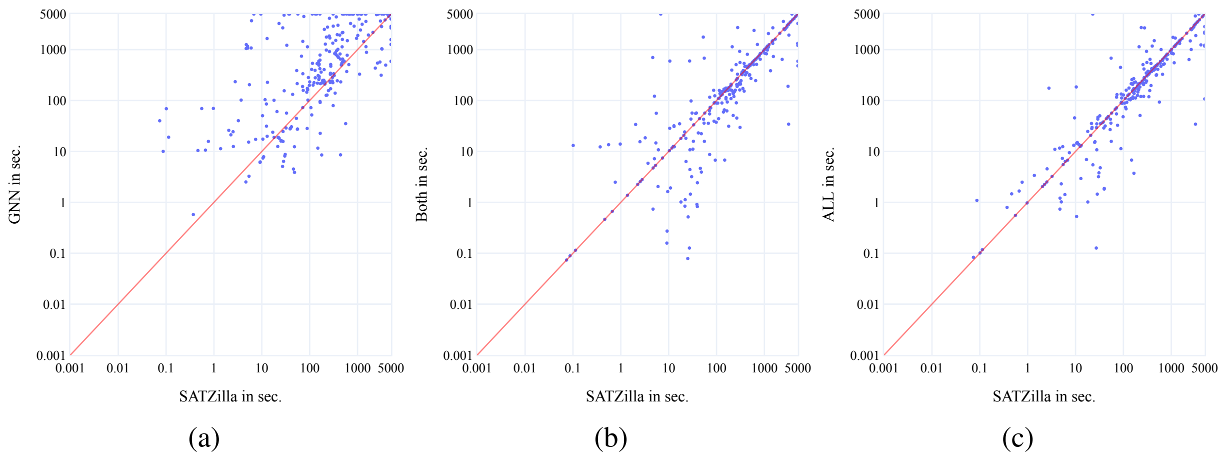


Figure B.20: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT18-EXP scenario (with feature costs), optimised for PAR$_2$.

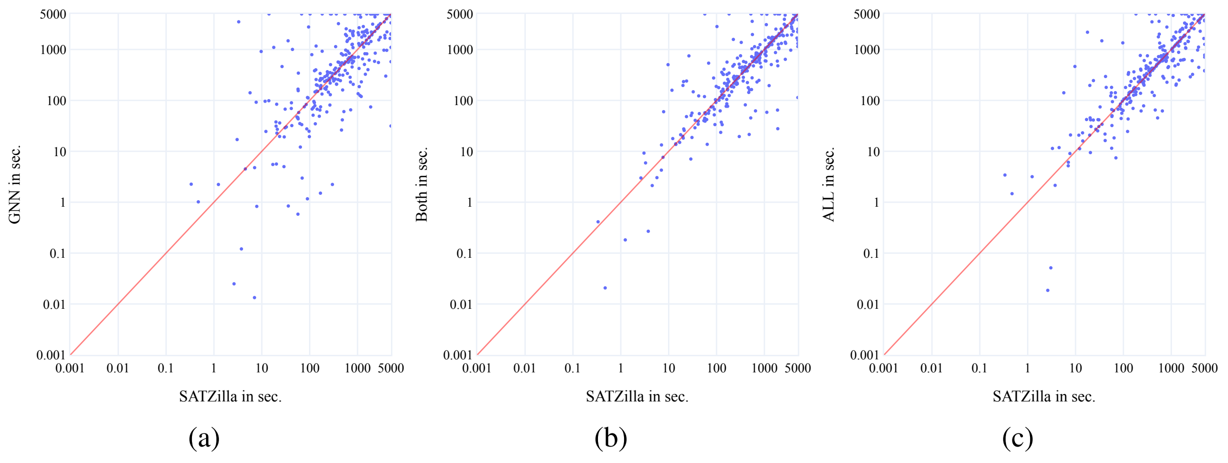


Figure B.21: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT20-MAIN scenario (with feature costs), optimised for PAR$_2$.

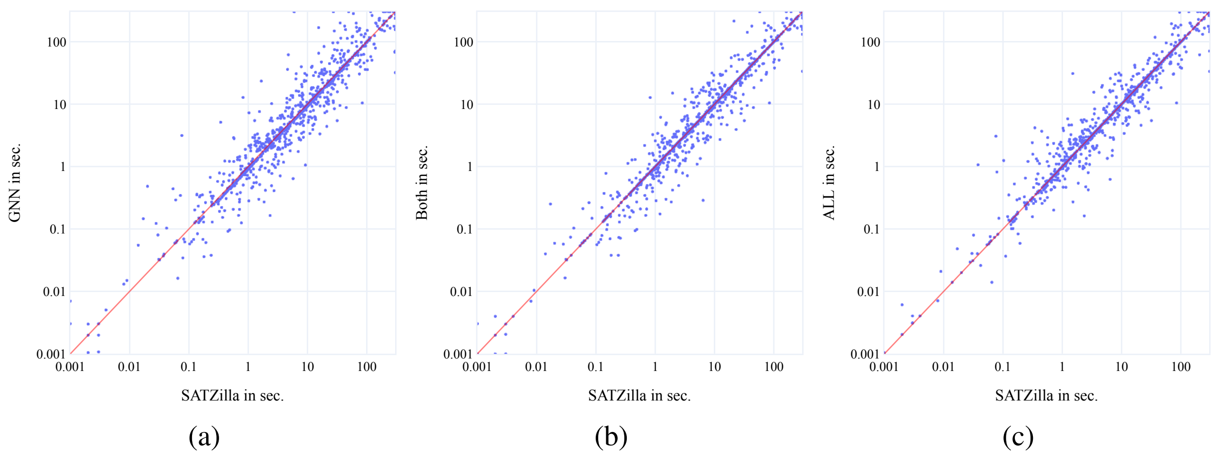


Figure B.22: Scatter plots of the running times of SATZilla, GNN, Both and all features on the CSSC-BMC08 scenario (with feature costs), optimised for PAR$_2$.

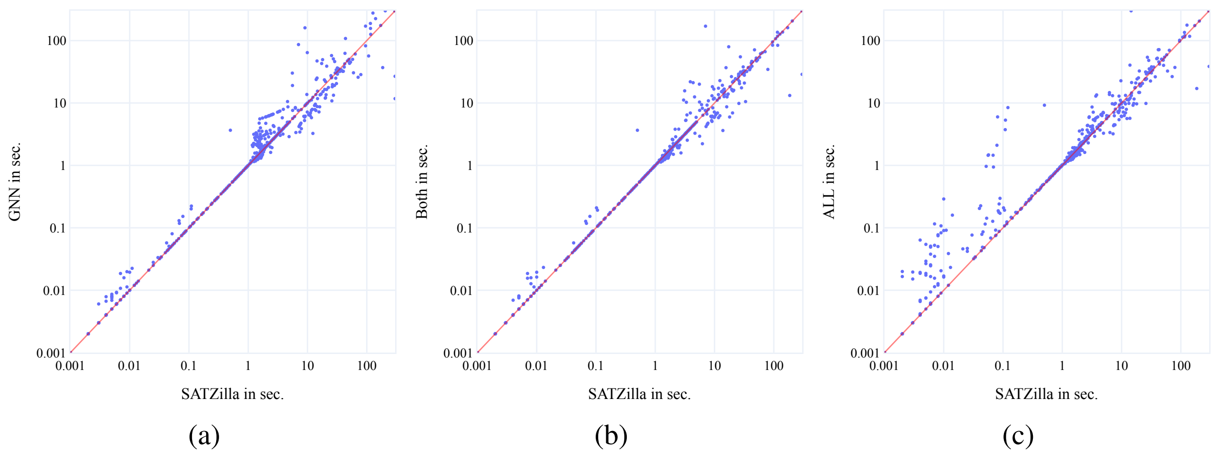


Figure B.23: Scatter plots of the running times of SATZilla, GNN, Both and all features on the CSSC-IBM scenario (with feature costs), optimised for PAR$_2$.

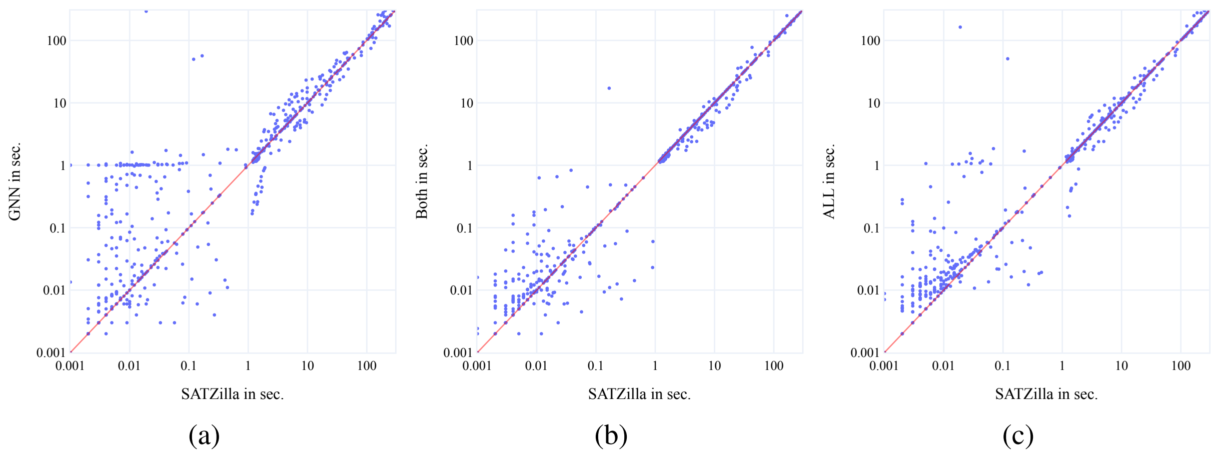


Figure B.24: Scatter plots of the running times of SATZilla, GNN, Both and all features on the CSSC-K3 scenario (with feature costs), optimised for PAR$_2$.

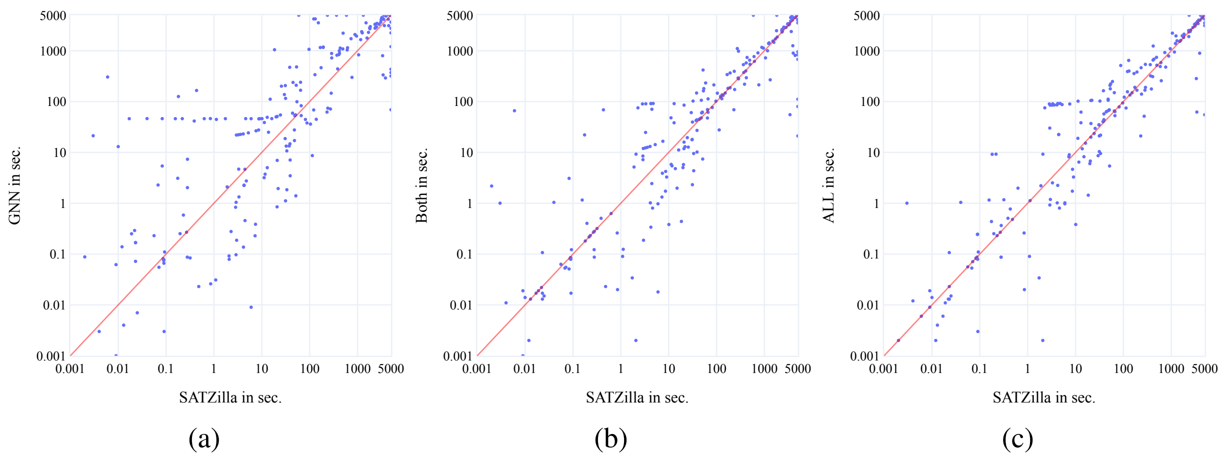


Figure B.25: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT11-HAND scenario (without feature costs), optimised for PAR$_2$.

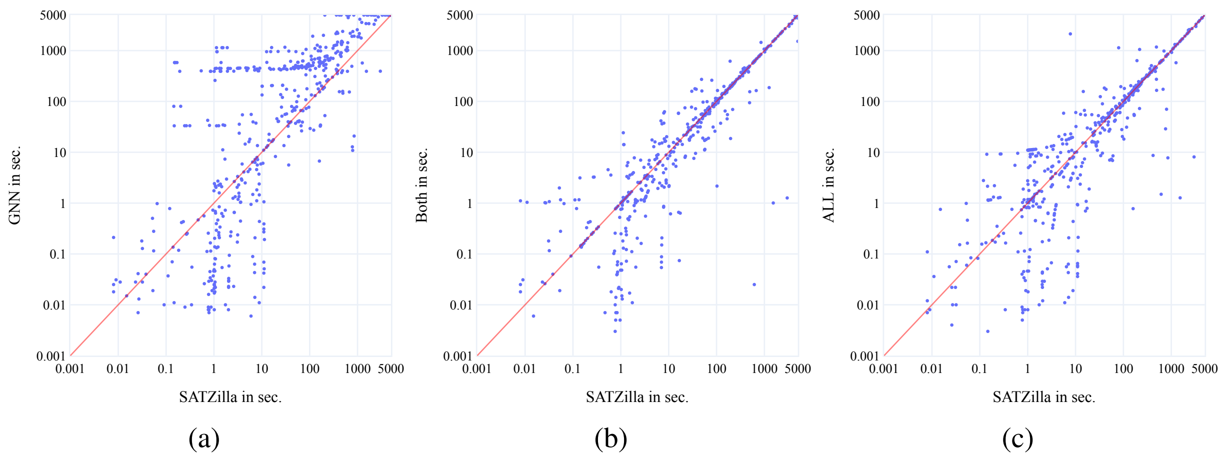


Figure B.26: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT11-RAND scenario (without feature costs), optimised for PAR$_2$.

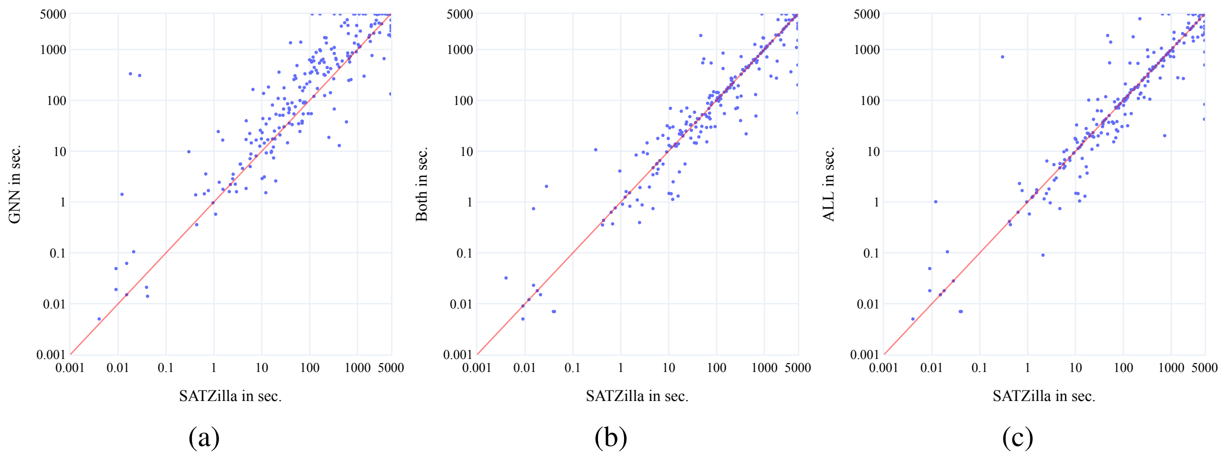
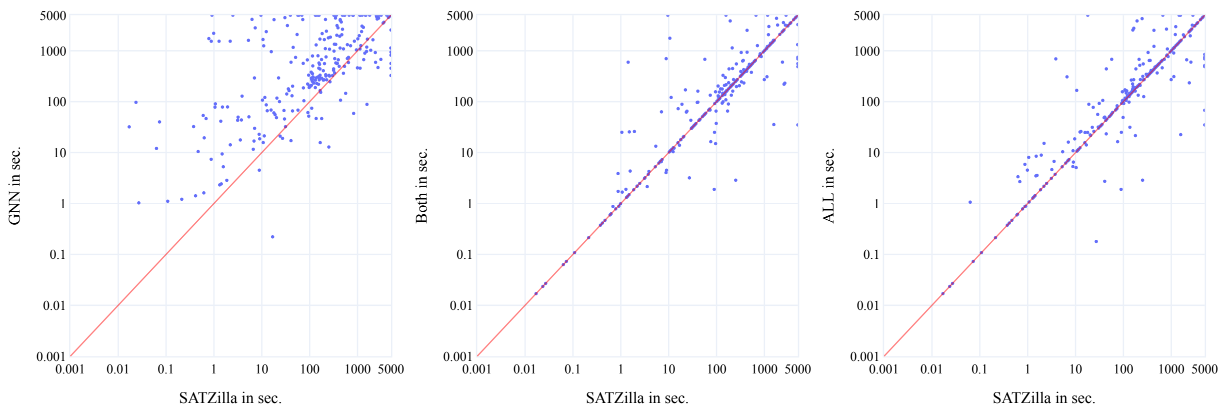
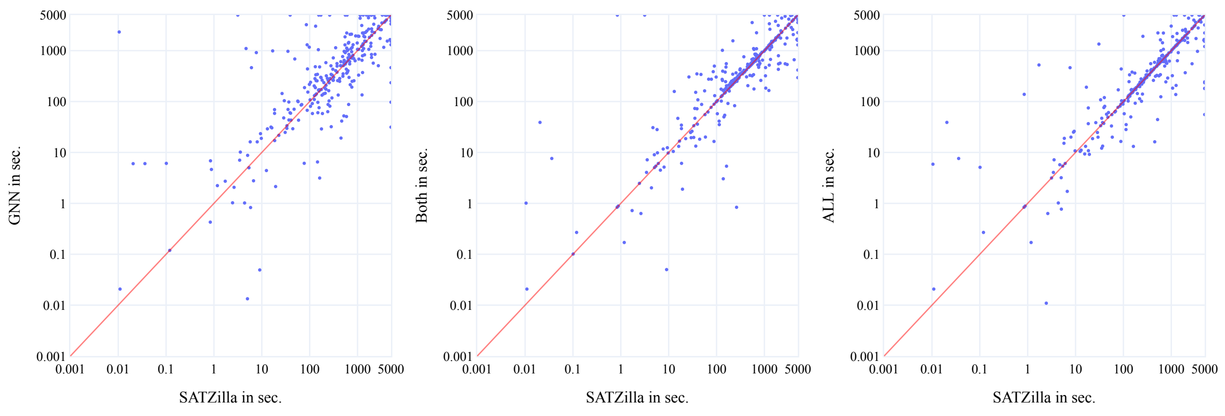


Figure B.27: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT11-INDU scenario (without feature costs), optimised for PAR$_2$.



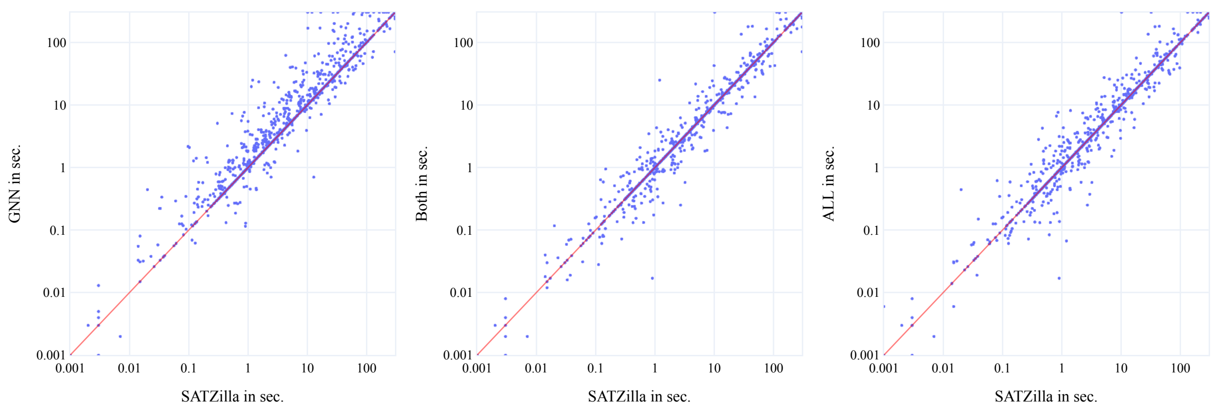
(a) (b) (c)

Figure B.28: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT18-EXP scenario (without feature costs), optimised for PAR$_2$.



(a) (b) (c)

Figure B.29: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT20-MAIN scenario (without feature costs), optimised for PAR$_2$.



(a) (b) (c)

Figure B.30: Scatter plots of the running times of SATZilla, GNN, Both and all features on the CSSC-BMC08 scenario (without feature costs), optimised for PAR$_2$.

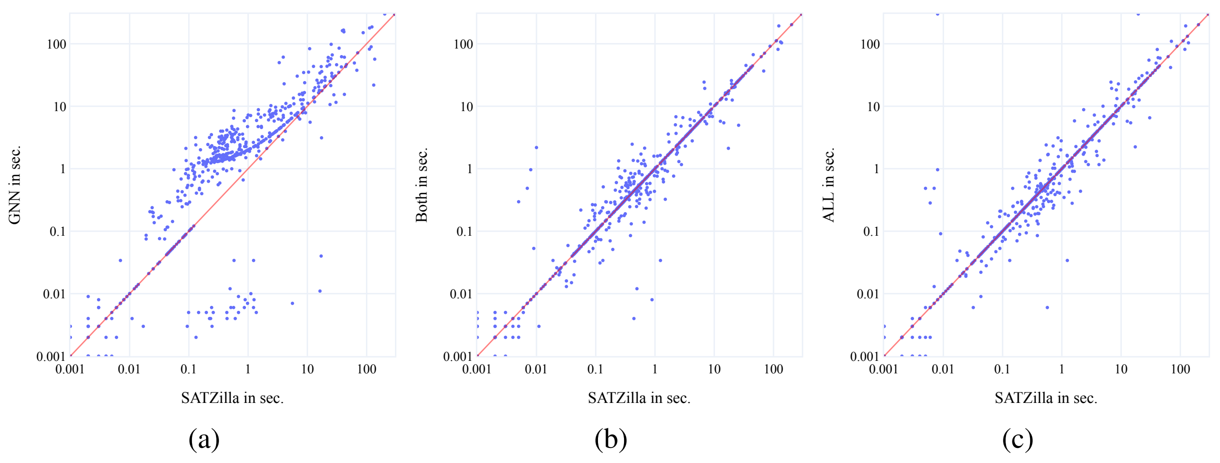


Figure B.31: Scatter plots of the running times of SATZilla, GNN, Both and all features on the CSSC-IBM scenario (without feature costs), optimised for PAR_2.

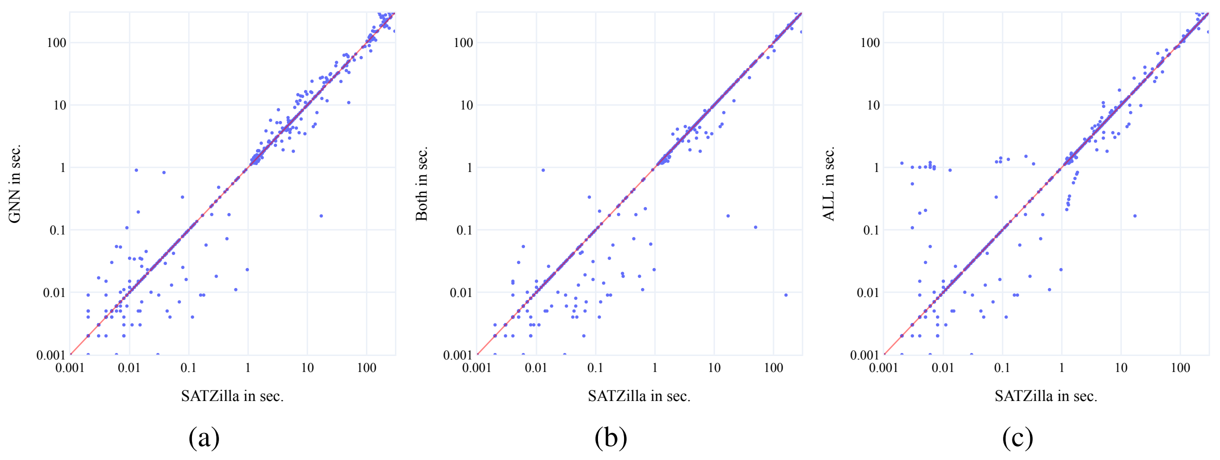


Figure B.32: Scatter plots of the running times of SATZilla, GNN, Both and all features on the CSSC-K3 scenario (without feature costs), optimised for PAR_2.

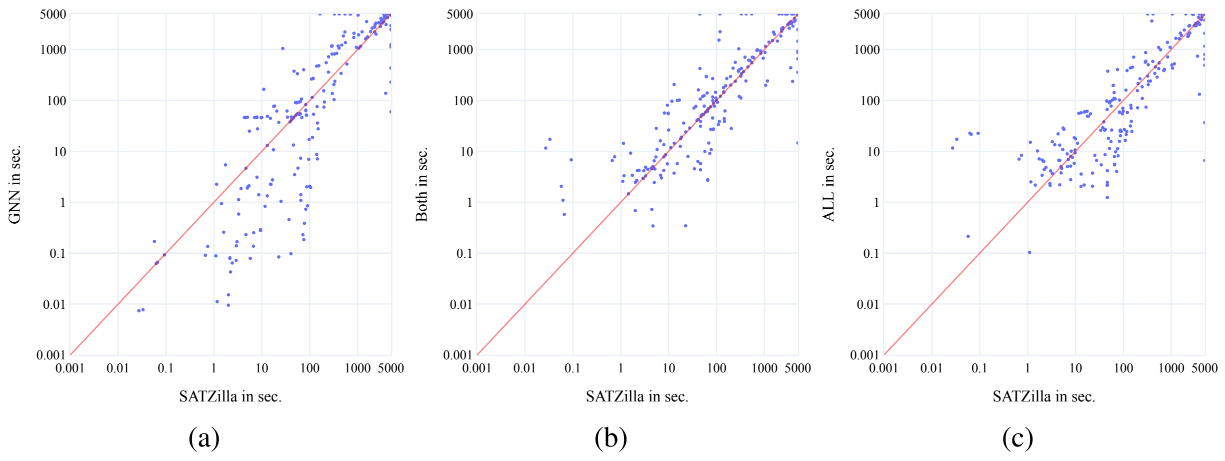


Figure B.33: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT11-HAND scenario (with feature costs), optimised for PAR_{10}.

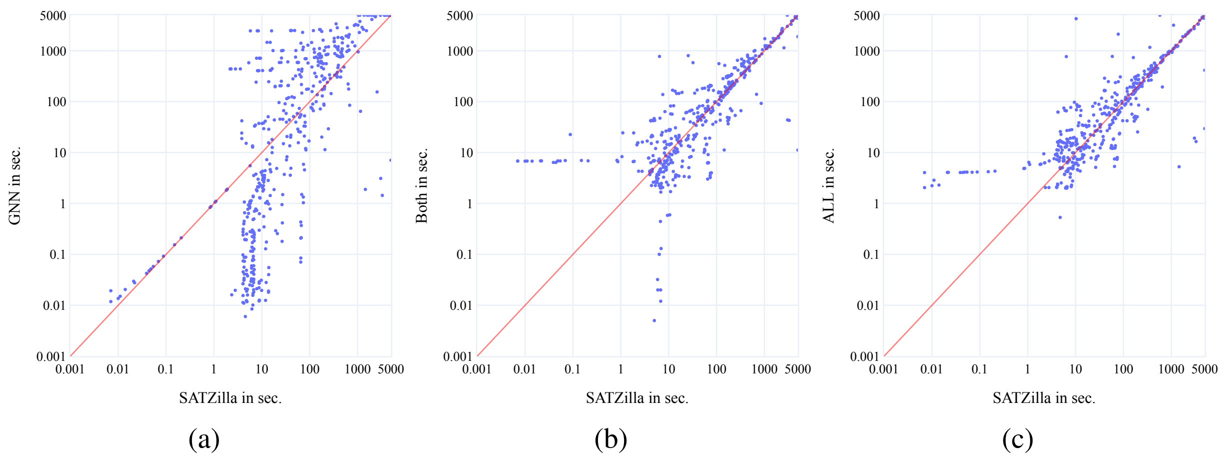


Figure B.34: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT11-RAND scenario (with feature costs), optimised for PAR_{10}.

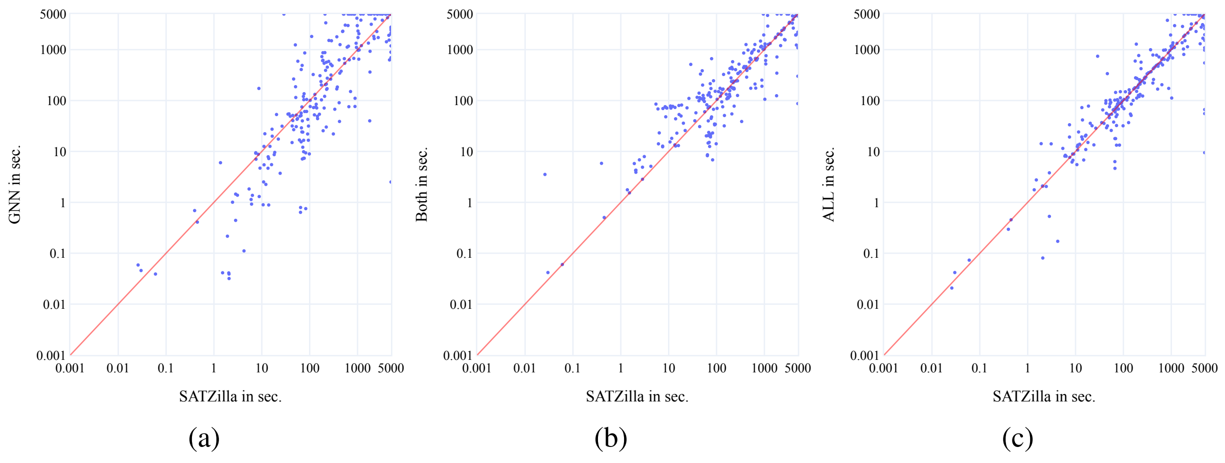


Figure B.35: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT11-INDU scenario (with feature costs), optimised for PAR_{10}.

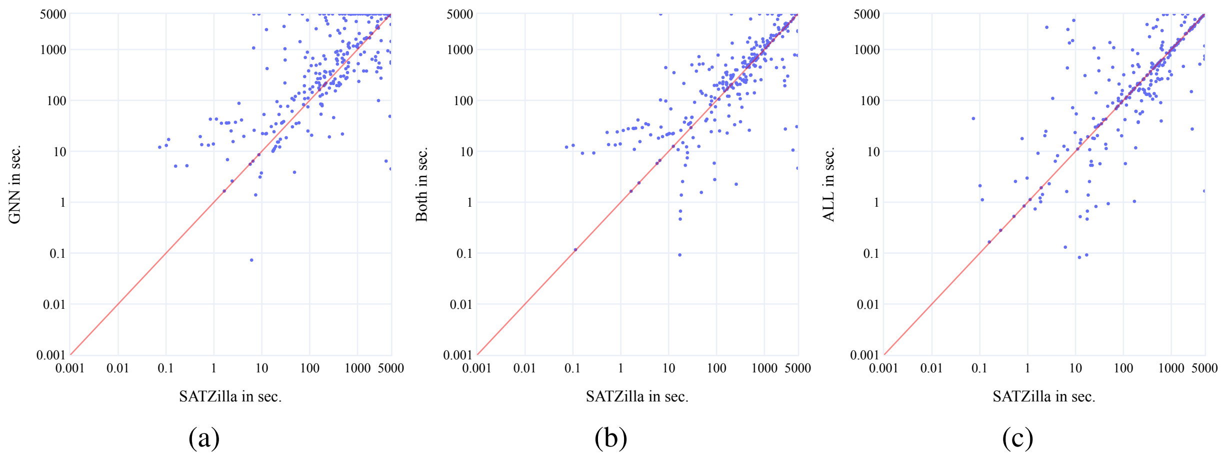
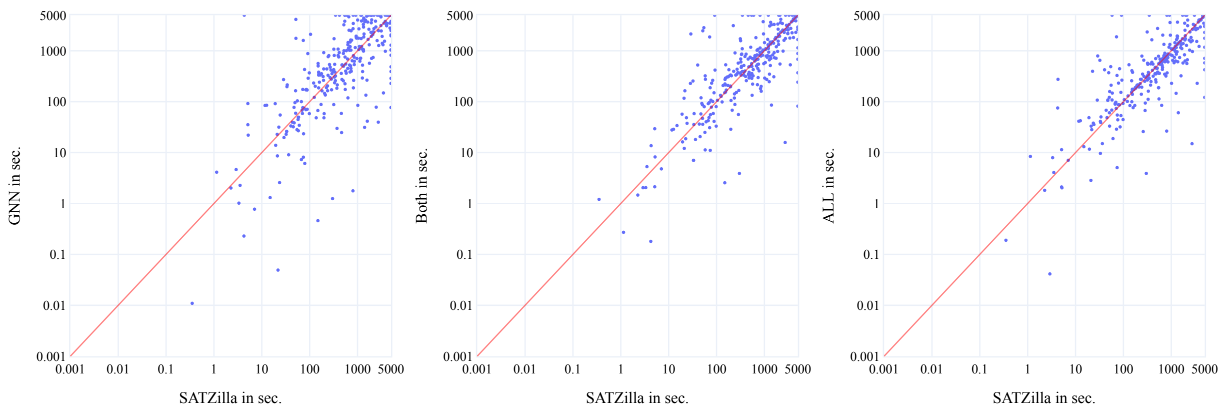
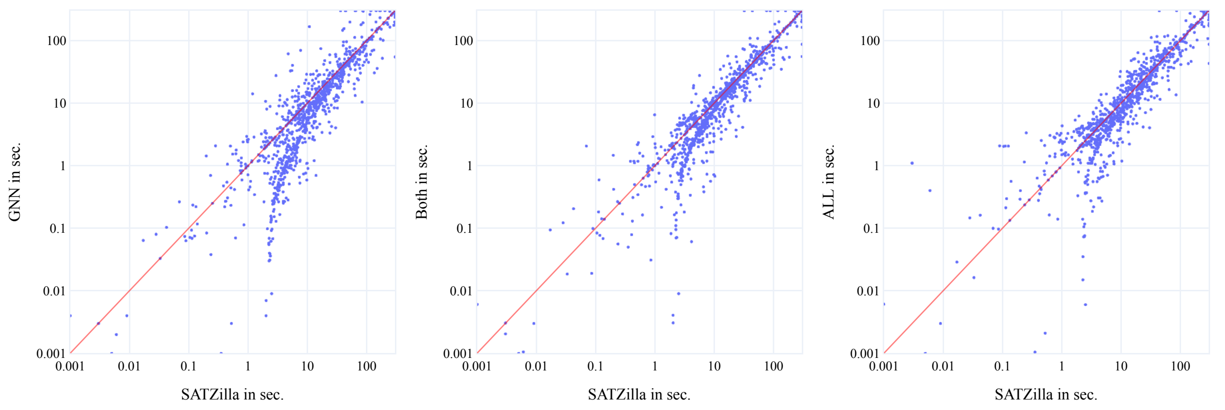


Figure B.36: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT18-EXP scenario (with feature costs), optimised for PAR_{10}.



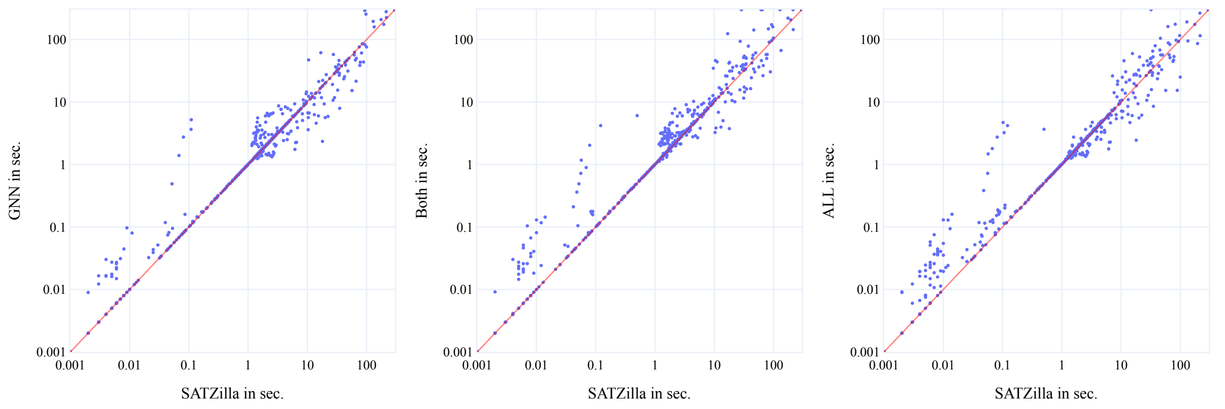
(a) (b) (c)

Figure B.37: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT20-MAIN scenario (with feature costs), optimised for PAR$_{10}$.



(a) (b) (c)

Figure B.38: Scatter plots of the running times of SATZilla, GNN, Both and all features on the CSSC-BMC08 scenario (with feature costs), optimised for PAR$_{10}$.



(a) (b) (c)

Figure B.39: Scatter plots of the running times of SATZilla, GNN, Both and all features on the CSSC-IBM scenario (with feature costs), optimised for PAR$_{10}$.

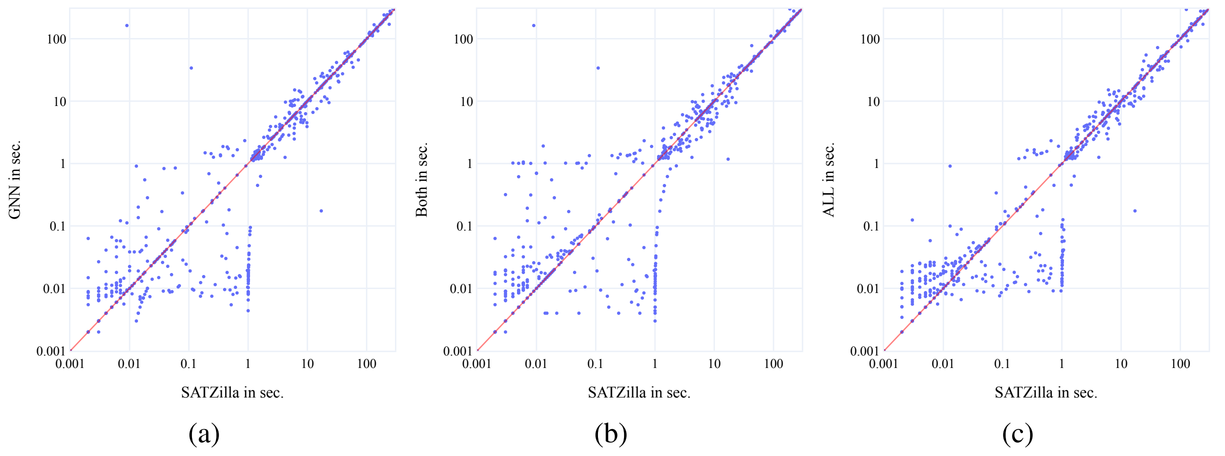


Figure B.40: Scatter plots of the running times of SATZilla, GNN, Both and all features on the CSSC-K3 scenario (with feature costs), optimised for PAR_{10}.

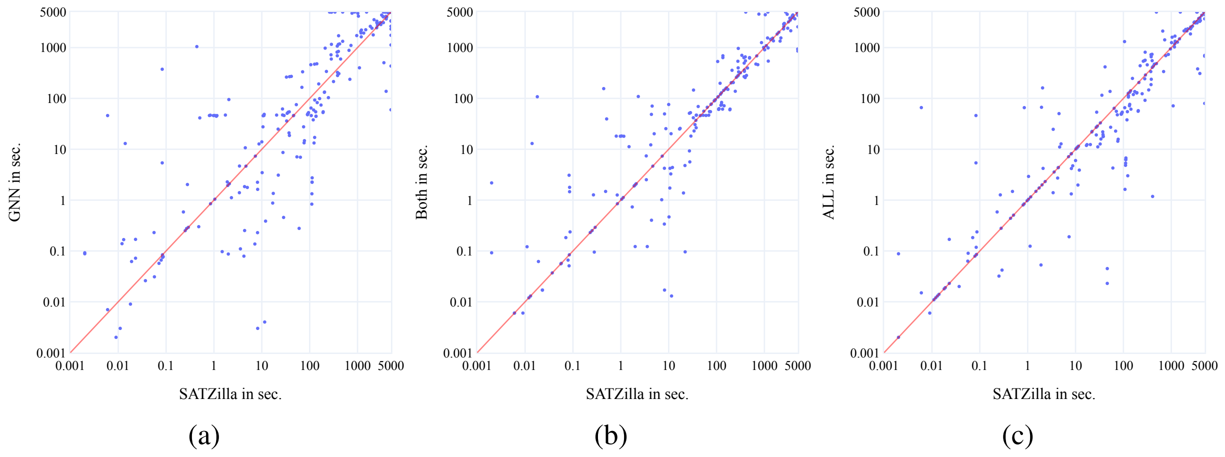


Figure B.41: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT11-HAND scenario (without feature costs), optimised for PAR_{10}.

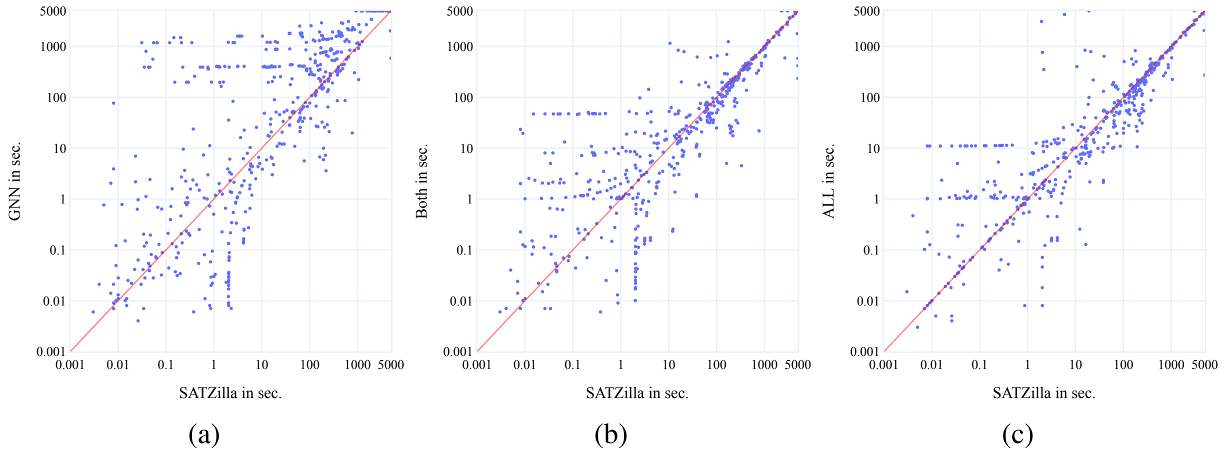
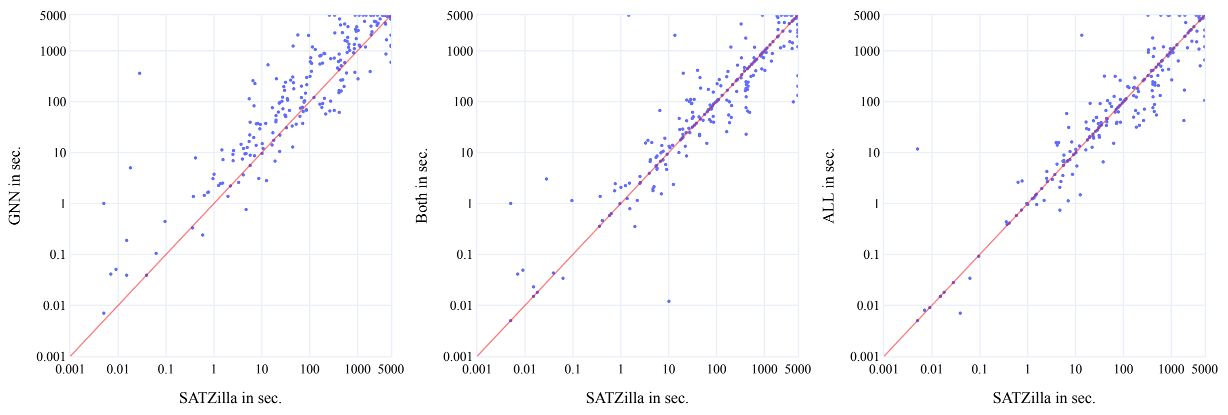
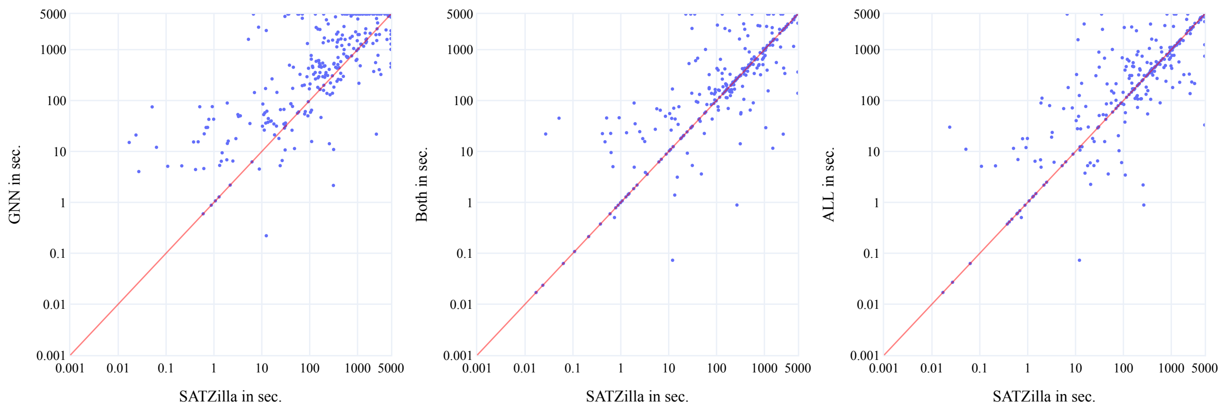


Figure B.42: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT11-RAND scenario (without feature costs), optimised for PAR_{10}.



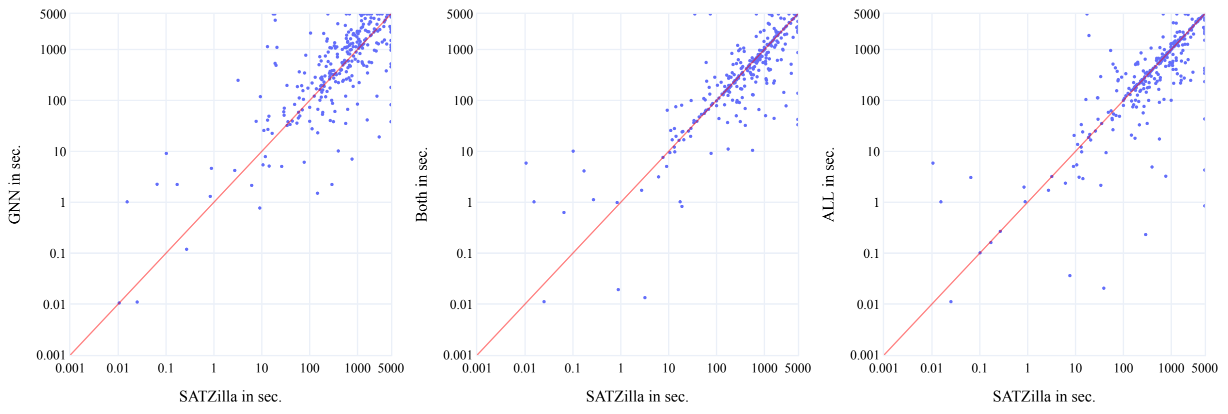
(a) (b) (c)

Figure B.43: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT11-INDU scenario (without feature costs), optimised for PAR$_{10}$.



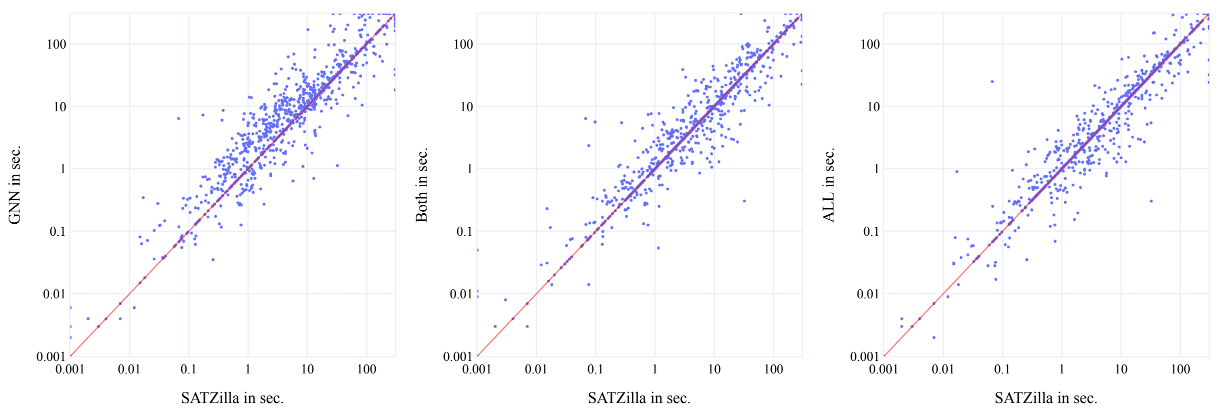
(a) (b) (c)

Figure B.44: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT18-EXP scenario (without feature costs), optimised for PAR$_{10}$.



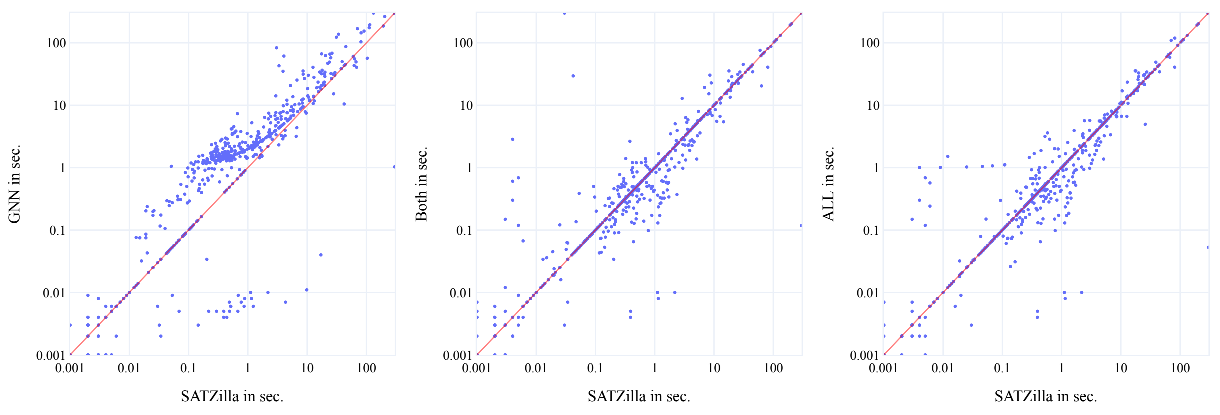
(a) (b) (c)

Figure B.45: Scatter plots of the running times of SATZilla, GNN, Both and all features on the SAT20-MAIN scenario (without feature costs), optimised for PAR$_{10}$.



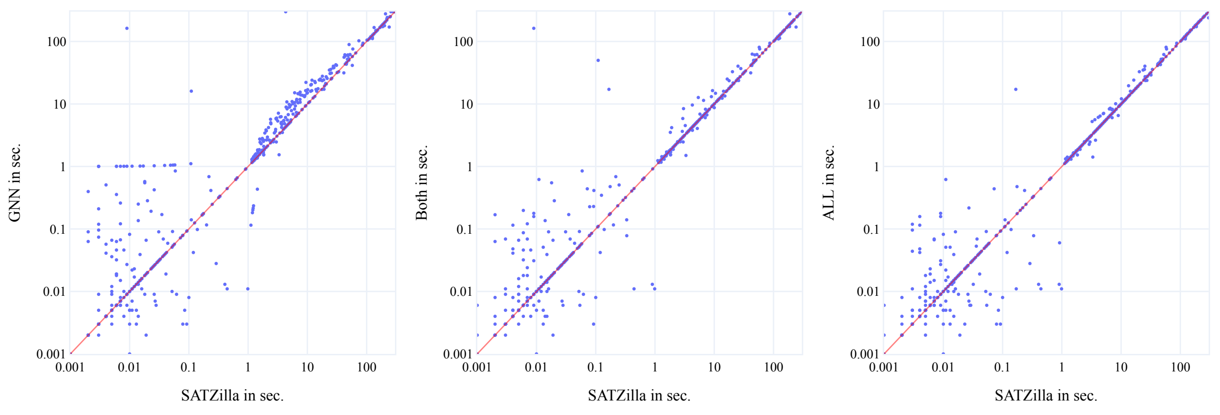
(a) (b) (c)

Figure B.46: Scatter plots of the running times of SATZilla, GNN, Both and all features on the CSSC-BMC08 scenario (without feature costs), optimised for PAR_{10}.



(a) (b) (c)

Figure B.47: Scatter plots of the running times of SATZilla, GNN, Both and all features on the CSSC-IBM scenario (without feature costs), optimised for PAR_{10}.



(a) (b) (c)

Figure B.48: Scatter plots of the running times of SATZilla, GNN, Both and all features on the CSSC-K3 scenario (without feature costs), optimised for PAR_{10}.