Sibyl: Improving Software Engineering Tools with SMT Selection

Will Leeson
Department of Computer Science
University of Virginia
Charlottesville, USA
will-leeson@virginia.edu

Matthew B Dwyer
Department of Computer Science
University of Virginia
Charlottesville, USA
matthewbdwyer@virginia.edu

Antonio Filieri
Department of Computing
Imperial College London
London, UK
a.filieri@imperial.ac.uk

***Abstract*—SMT solvers are often used in the back end of different software engineering tools—e.g., program verifiers, test generators, or program synthesizers. There are a plethora of algorithmic techniques for solving SMT queries. Among the available SMT solvers, each employs its own combination of algorithmic techniques that are optimized for different fragments of logics and problem types. The most efficient solver can change with small changes in the SMT query, which makes it nontrivial to decide which solver to use. Consequently, designers of software engineering tools often select a single solver, based on familiarity or convenience, and tailor their tool towards it. Choosing an SMT solver at design time misses the opportunity to optimize query solve times and, for tools where SMT solving is a bottleneck, the performance loss can be significant.**

In this work, we present Sibyl, an automated SMT selector based on graph neural networks (GNNs). Sibyl creates a graph representation of a given SMT query and uses GNNs to predict how each solver in a suite of SMT solvers would perform on said query. Sibyl learns to predict based on features of SMT queries that are specific to the population on which it is trained – avoiding the need for manual feature engineering. Once trained, Sibyl makes fast and accurate predictions which can substantially reduce the time needed to solve a set of SMT queries.

We evaluate Sibyl in four scenarios in which SMT solvers are used: in competition, in a symbolic execution engine, in a bounded model checker, and in a program synthesis tool. We find that Sibyl improves upon the state of the art in nearly every case and provide evidence that it generalizes better than existing techniques. Further, we evaluate Sibyl's overhead and demonstrate that it has the potential to speedup a variety of different software engineering tools.

***Index Terms*—graph neural networks, satisfiable modulo theories, algorithm selection**

I. Introduction

Satisfiability modulo theories (SMT) is the problem of determining if there exists an assignment to the variables in a logical formula involving predicates from different mathematical theories that makes it true. Software engineering tools, such as model checkers [1]–[5] or symbolic executors [6]–[9], can formulate problems as SMT queries and then rely on SMT solvers to determine if there is a solution to the query. In practice, the SMT solvers' ability to solve queries tend to be a performance bottleneck for these tools [9]–[12].

There are many SMT logics, dealing with combinations of arrays, bit-vectors, integers, reals, strings, etc. This has resulted in a diversity of solvers, some specializing in certain logics or fragments of a logic, and others attempting to be more general. The International Satisfiability Modulo Theories Competition (SMT-COMP) is an annual event which evaluates SMT solvers on hundreds of thousands of queries across the various logics [13]. Solvers are scored based on how many queries they could correctly solve, with ties being decided by time. In SMT-COMP'21, 9 different solvers came in first place in at least one category. For each category, there are many queries where the winner isn't the fastest solver. In the best case, the winner is fastest on 96.7% of the queries. In the worst case, the winner is only the fastest solver on 35.4% of the queries. All this is to say that, even when looking at individual logics, there is no optimal solver, where optimal means correct followed by most efficient in use of resources.

In an effort to utilize the strengths of various SMT solvers, some software engineering tools support multiple solvers [1], [10], [14]. These tools typically only allow the user to select which solver to use across the entire run of the tool on a given input. On a single input, tools may generate millions of queries and its been shown that different solvers may be optimal on different subsets of queries generated on said input [10]. Some tools allow for running a *portfolio* of SMT solvers in parallel and stop once the first solver has solved the query [10]. On simple queries, this is reasonable, but as queries become larger and more complex, this can result in inefficient use of resources. Choosing the optimal solver from a portfolio for a given problem requires both an understanding of how a given tool generates SMT queries and of the strengths of each solver in a portfolio. This is challenging both for SMT experts, who may be unfamiliar with tool-generated queries, and for SE tool experts, who may be unfamiliar with the intrinsic capabilities of multiple SMT solvers.

Algorithm selection is an approach that attempts to exploit the benefits of a diverse set of algorithms to best solve a problem. An algorithm selector for SMT queries would attempt to select the solver which most efficiently uses resources, such as time or memory, to solve a given query. Algorithm selection is a heuristic approach and can be used in several ways. The most basic is choosing the selected solver and running it. If the algorithm selector makes optimal choices and has little overhead, this is ideal. In general, optimal algorithm selection is challenging if the population of problems and the portfolio

of solvers is highly diverse. Another solution would be to choose the top-k predicted solvers selected and run them in parallel [15]–[17]. Resource usage would rise with k, but if the best solver was one of those selected, the problem would still be solved in the optimal time.

Even if an algorithm selector is efficient, there will still be some overhead required to make a selection. SMT solvers can solve some queries in microseconds, meaning selection will often slow down solving. One way to counteract selection latency is to run a solver that is fast on average in parallel with the selected solver. If the query is simple, the fast-on-average solver will solve it quickly and the time overhead of selection does not matter. If the query is complex, however, the selector can choose the best solver for it. The modest degree of parallelism in this approach will use fewer resources than the top-k approach while offering its benefits. In this paper, we evaluate several solvers using both *selection-only* (pure algorithm selection) and *selection-in-parallel* (selection in parallel with a single solver).

The current state-of-the-art, MachSMT [18] and MedleySolver [19], rely on feature engineering, a process which requires developers to extract features they determine, through domain knowledge, are important to various solvers' success. They then use these features to train a model which can perform selection on new queries when they are presented to it. These techniques were tuned by SMT experts to perform well across a range of SMT-Comp benchmarks. As we demonstrate, however, when the fragments of SMT logics that arise in software engineering domains are considered, those features are no longer well-suited to differentiating tool performance.

In this work, we present Sibyl, an algorithm selector for SMT solvers based on graph attention networks (GATs), a variant of graph neural networks (GNNs). Unlike MachSMT and MedleySolver, Sibyl learns how to generate a feature vector, a process known as representation learning. Thus, it can be tailored to a specific domain during the training process. Given an SMT query, Sibyl generates a graph representation of the query and produces scores for each SMT solver in a portfolio of solvers.

We evaluate Sibyl in comparison with the state-of-the-art, and other selection approaches, on three different software engineering problem domains: symbolic execution, bounded model checking, and program synthesis. We find that, while MachSMT outperforms Sibyl on the SMTComp benchmarks, which it was designed for, Sibyl outperforms the state-of-the-art by 37.6% to 159.7% across each studied problem domain.

This paper makes the following contributions: (1) it introduces an algorithm selection technique for SMT solvers based on state-of-the-art graph neural networks; (2) it demonstrates that Sibyl improves on the state-of-the-art by evaluating SMT algorithms selectors on a large, diverse set of benchmarks, selected from four domains; (3) it evaluates the practicality of integrating Sibyl into an SE tool; and (4) it demonstrates, through an ablation study, that each components of Sibyl's design contributes to its performance.

II. Background and Related Work

In the following sections, we provide background information on the use of SMT in software engineering, GNNs, and algorithm selection and summarize the main related work.

A. SMT in Software Engineering

In this paper, we examine SMT solvers in three software engineering scenarios: in a bounded model checker, in a symbolic execution engine, and in a software synthesis tool. In each of these scenarios, the tool generates SMT *queries* to solve the given problem. A query is specified by a *logic* and a *constraint*. The logic defines the theory underpinning the semantic interpretation of the boolean expression representing the constraint. For example, the constraint $x > 0 \wedge 2 * x <= 1$ is satisfiable, e.g., $x = 0.5$, if interpreted in the logic of linear real arithmetics (LRA) while it is unsatisfiable in the logic or linear integer arithmetics (LIA). The SMTLib standard [20] defines a broad set of logics supported by most solvers.

Bounded model checking builds a finite state model of a software execution up to a maximum number of steps, where the transition relation between states is derived from the semantics of the program's instructions [21]. Violations of a desired property – e.g., an assertion condition evaluating to false – are encoded as transitions towards designated error states. The model checker then verifies whether the transition relation allows transitively reaching any error state from the initial one. SMT-based model checkers encode the transition relation as an SMT problem and query a solver to determine the reachability of error states [11], [12], [22]. Building on the semantics of established SMT theories, the transition relation can be encoded compactly, and the solver finds an assignment to the problem's variables that allow reproducing a failing execution (counterexample).

Symbolic execution engines can be used to test programs by executing them not with symbolic inputs which represent all possible inputs of the given type [23]. When the symbolic executor reaches a conditional branch, the execution forks to cover each of the execution branches, and each of the forked executions collect the condition or its negation in their *path condition* – a logical constraint characterizing all the program inputs that would follow the same execution path. Path conditions are checked for satisfiability after each fork to decide if the path is still feasible, and solved to generate a representative test case at the end of a terminating path.

Program synthesis tools attempt to generate a program that conforms to a given specification. For example, a specification could be "Given a list of integers, return the same integers in ascending order". The synthesizer would be expected to output a function which sorts a list of integers. Given a specification, a synthesizer can generate the constraints on the function that must hold for the specification to be met, e.g., $A[i] \leq A[i+1]$. These constraints are then simplified into first-order constraints which can be solved by SMT solvers to verify conformance to the specification or guide the synthesis process [24].

B. Graph Neural Networks

Traditionally, machine learning techniques, such as convolutional neural networks (CNNs) or SVMs, rely on the fact that the data they operate on has a consistent size and ordering [25], e.g., images have a fixed resolution and orientation. In general, graphs have neither of these attributes. They can have any number of nodes and edges and have no canonical order. One could collate all nodes in a graph into a feature vector, but this would lose the information in the edges and overall structure of the graph. Graph neural networks (GNNs) were introduced to capture these characteristics [25].

GNNs operate by propagating information among the nodes of the graph through the edges that connect them via message passing layers. Each node in the graph has a state vector. Initially, this vector represents some information about the node. Over a series of propagation steps, the state of each node is updated using its state vector and the state vectors of its neighboring nodes. After each step, node values are transitively influenced by neighbors further in the graph. This diffusion process allows each node to update its state to reflect the data from other nodes and the structure of the graph.

Once the propagation is completed, the new representation of the graph can be used to perform various tasks. Individual nodes in the graph can be used to perform node-based prediction tasks, such as recommending a connection to a person in a social network [26]. The entire graph can be used to perform graph-based tasks, such as predicting properties of a chemical compound [27]. In order to do so, a pooling layer, a layer which collates the graph into a fixed size, must be used so a traditional prediction technique, like a fully connected neural network, can operate on it.

Graph Attention Networks (GATs) are a type of message passing layer which use *attention mechanisms* to calculate the state value of nodes in a graph [28]. For each node n, the attention mechanism computes a score for each of n's neighbors, which roughly represents the importance assigned to the information coming from a neighbor. This attention score is then used to weight each neighboring node's encoding when they are used to form a new encoding for n. This allows the network to learn which components in an encoding are important when performing the task as the attention mechanism is a learned component, typically a one layer feed forward neural network. For example, if an SMT solver excels at handling queries with quantifiers, the network may highly weight the encoding of the quantifier token. If strictly less than operations do not affect solver performance much, it may learn to associate this token's encoding with a small weight.

Because many aspects of software can be abstracted in graph form, GNNs have been used to accomplish various tasks in software engineering contexts. GNNs have also been used to perform program classification [29]–[31], compiler optimization scheduling [32], and predicting variable names [33].

C. Algorithm Selection

As SMT-COMP has shown [34], there is no optimal SMT solver in general, even when limited to a single logic. This has motivated algorithm selection for SMT solvers. Algorithm selectors seek to improve performance on a problem class by selecting the optimal algorithm for a given problem. In this section, we describe three approaches – rule based, feature engineering, and representation learning – along with the current state-of-the-art algorithm selectors for SMT solvers.

Rule based algorithm selection is a fairly simple approach to selection. An algorithm selector extracts features from the given problem, typically using some lightweight analyses, and composes the portfolio of algorithms to run based on a given set of conditionals [35]. For example, a rule could be "if a query has strings, run solver X". This technique requires the designer of the rules to know the strengths and weakness of each tool in the portfolio in order to be effective. As a result, adding tools to the portfolio is nontrivial.

Feature engineering is a more flexible approach to selection. Once again, the selector extracts features from the problem to create a feature vector. Unlike the previous technique, the developer does not decide the selector's output for a combination of features, instead a machine learning model, such as a linear regression model or a support vector machine (SVM) [18], [19], [36], is trained to predict the best output. In order to do so, the model must be trained using a labeled dataset, where the label is the metric the developer wants the selector to optimize, such as time or memory used. The design of the feature set for models like SVMs can be challenging and there is a risk that too few features leads to high bias in the learned model – and poor generalization – and too many features leads to high variance – and poor accuracy [37].

Representation learning is the most adaptive of these techniques. The developer is relieved of deciding which features to extract. Instead, the selector makes use of an encoder which transforms a representation of the problem, an SMT query in our case, into a feature vector a traditional machine learning model can operate on [38]. The encoder learns how to form a feature vector, allowing it to adapt to the training population. In this paper, we represent SMT queries as graphs and use a GNN to learn an encoding for queries, similar to the algorithm selector introduced by Hůla et al. [39]. A downside of this technique is that it can be hard to decipher why predictions are made. There are techniques which can be used to extract interpretability information from a model. Ying et. al introduce an approach which masks portions of the graph to see how they affect a prediction [40].

MachSMT [18] and **MedleySolver** [19] are feature engineering approaches to algorithm selection for SMT queries that use a random forest [41] with AdaBoosting [42] and multi-armed bandit models [43], respectively. They are the current state-of-the-art in SMT algorithm selection and will act as baselines in our experiments. The developers of each technique have identified a set of syntactic features, overlapping by over 80%, which can be extracted easily. Each were evaluated on SMT-COMP benchmarks and shown to perform well on them. Because these techniques are based on feature engineering, they rely on their chosen features being able to differentiate solvers performance. We show in our evaluation that many of the

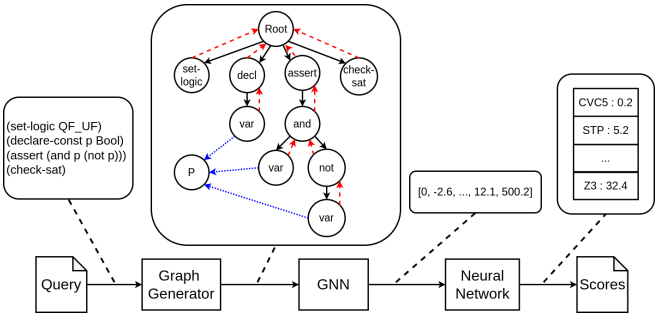

Fig. 1: Sibyl prediction pipeline. Queries are converted to graphs, which a GNN encodes as a feature vector. A simple neural network predicts solver performance using this vector.

features do not appear in SE problem domains. We believe this explains the poor accuracy we observe for these techniques – the large feature counts lead to unnecessarily high variance for these domains. Engineering features for each domain would resolve this, but that is precisely what representation learning does.

Hůla et al. [39] introduce a more adaptive approach which uses graphs to represent queries and a basic GNN to generate vector representations of the graph. Their approach uses a relatively simple message passing layer called graph convolutional networks (GCN). For each node, the representation of its neighboring nodes are scaled by the neighborhood size and then summed together. The resulting value is then multipled by a learned matrix. Besides scaling the value of each node, it does not adapt to the neighborhood. Once the GNN is finished, their approach forms a vector representation by taking the element-wise maximum of all nodes in the graph which is fed to a simple feed-forward network to make a prediction.

III. Approach

MachSMT and MedleySolver use human engineered feature vectors to represent SMT queries. The majority of these features capture "counts" of syntactic entities in a query, e.g., the frequency of different tokens, the ratio of forall or exists quantifiers. While efficient to calculate, these features do not capture query semantics. While they provide some value for prediction, they are likely to miss some information since syntactic counts cannot predict the performance of a single SMT solver, much less multiple solvers, as evidenced by these queries: $x \cdot y = 7919 \wedge x > 1 \wedge y > 1$ and $x \cdot 7919 = y \wedge x > 1 \wedge y > 1$, which have identical syntactic count profiles but take 0.87 and 0.08 seconds, respectively, to solve with Z3 [44].

Beyond this, many of the human engineered features may remain unused depending on the domain of the query. For example, none of the software engineering tools we studied produce queries with floats in them, whereas over a quarter of the engineered features concern floats. Such approaches cannot automatically adapt their representation to a given domain.

In this paper, we introduce Sibyl, a representation learning approach for SMT selection which uses state-of-the-art graph neural networks. Like [39], Sibyl represents queries as a graph and uses a GNN to form a vector representation. However, Sibyl uses a more sophisticated GNN which makes use of several state-of-the-art GNN layers. Instead of a GCN, Sibyl uses GAT layers which learn how to calculate the value of each node in the graph based on its specific neighborhood. Take the following two query fragments as an example: $X \vee \texttt{True}$ and $X \vee Y$. The first fragment is true regardless of the remainder of the query. The second fragment can be can be true or false depending on the values of the variables. The attention mechanism can learn that the `True` constant is more important in the first fragment and the free variable can essentially be ignored during propagation. For the second fragment, it can learn both free variables carry importance.

Instead of taking the elementwise maximum of each node in the graph to form a final representation, Sibyl uses an attention-based pooling operator. This operator learns how to scale the different values in a node's representation and sums the values together, effectively boosting information the network learns is important, and minimizing information it deems superfluous.

In the remainder of this section, we describe the Sibyl technique which consists of three phases, depicted in Figure 1: graph generation, representation generation via a GNN, and prediction via a simple three layer neural network.

A. Graph Generator

Definition III.1. A directed Graph $G = (\mathcal{V}, \mathcal{H}, \mathcal{E})$ where:

- $\mathcal{V} = \{v_0, v_1, \ldots, v_n\}$; v_i is a node
- $\mathcal{H} = \{h_0, h_1, \ldots, h_n\}$; h_i is an encoding of v_i's value
- $\mathcal{E} = \{E_0, E_1, \ldots, E_m\}$; E_i corresponds to the i^{th} edge type in the graph
- $E_i = \{e_0, e_1, \ldots, e_{u_i}\}$; $e_i \in \mathcal{V} \times \mathcal{V}$ is a directed edge

To form a graph $G_{Sibyl} = (\mathcal{V}_{Sibyl}, \mathcal{H}_{Sibyl}, \mathcal{E}_{Sibyl})$, Sibyl's graph generator combines two graphs: the query's abstract syntax tree (AST) and its use-def graph (UD).

The nodes of the AST represent the various operations, operands, constants, and variables in a query and their relationships. Purely structural tokens, such as parentheses or whitespace, are implicit in the graph structure and are ignored. By discarding structural tokens, the GNN is able to propagate information about semantic relationships between fragments of a query more directly with message passing layer since there are fewer intervening tokens.

Rather than encoding variable names in the graph, which would require the learner to relate string attributes of graph nodes, the AST nodes are related by UD edges that make explicit the relationship between the uses of the free variables in a query. The UD has a node for each free variable. We call these nodes "context" nodes as they maintain information about the various contexts in which free variables are used.

$\mathcal{H}_{Sibyl}$ consists of the one-hot encoding of each node in $\mathcal{V}_{Sibyl}$. Let T be the set of possible tokens in an SMT query. Each token $t_i \in T$ is represented by an index $i \in [1, |T|]$. The one hot encoding of t_i, $h(t_i)$, is a vector of length $|T|$ of 0s and a single 1 where $h(t_i) = h(t_j) \Leftrightarrow t_i = t_j$. One-hot

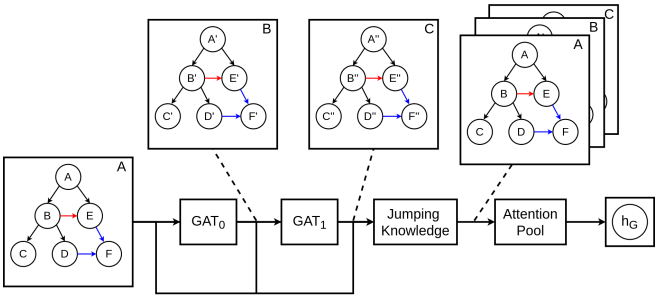

Fig. 2: Sibyl's GNN consists of three types of layers: GAT layers, a jumping knowledge layer, and an attention-based pooling layer. In the experiments, we explore how varying the number of GAT layers affects model performance.

encodings are frequently used in graph-based learning contexts to represent the value of nodes as they are simple to calculate and tend to perform well in practice [36], [45]–[47].

$\mathcal{E}_{Sibyl} = \{E_{TB}, E_{BT}, E_{UD}\}$. E_{TB} is set of edges in the AST traversing the tree from top to bottom. E_{BT} is set of edges in the AST traversing the tree from bottom to top. E_{UD} is a set of edges connecting free variable uses to their corresponding context node. The edges in Sibyl's graphs are directed, following the convention for most GNNs. Undirected graphs can be represented as two directed graphs and the GNN can learn separate weights for each graph.

The edges of the AST allow the GNN to capture the interaction between the various nodes in the graph. E_{TB} allows information in the graph to flow down the tree. This results in the nodes at the bottom of the tree, the operands, having encodings that were formed using both their initial encoding and those of the operators which make use of them. E_{BT} allows the opposite to happen, resulting in the operators being informed by their operands, which may be the results of other operators or terminal nodes. The interplay between these two edge sets also allows information from one operand to flow up to its operator and back down to another operand, effectively informing each term in an equation about each other.

The UD graph has directed edges from use to definition, or context node. This way, the uses of free variables maintain only information of how they are used. The GNN can then propagate the appropriate information from the uses to the context node, allowing context nodes to maintain the information the GNN deems important about every one of its uses.

B. Graph Neural Network

In general, the purpose of a GNN is to transform graph data into a representation that both captures the latent information in the structure of the graph and is amenable to traditional machine learning techniques, in our case a feed forward neural network. The traditional technique can then perform a task, like prediction, it is normally used for.

Visualized in Figure 2, Sibyl uses a GNN consisting of a variable number of GAT layers [28], a jumping knowledge layer (JK) [48], and an attention-based pooling layer [49].

GAT layers are the message passing layer in Sibyl's GNN, allowing information to propagate across edges. During this process, the GAT's attention layer weighs the information that is being propagated. It learns to weigh information based on the current state of the node and the information being "received", in effect weighting the edge between two nodes. GATs are state-of-the-art graph neural networks that are widely used [50]–[52] and have been shown to outperform GCNs [28].

After the GAT layers, the JK layer calculates a final representation for each node in the graph. JK layers allow the network to use the intermediate node representations produced by the message passing layers when calculating a final representation. The JK layer takes in the initial node representations and the representations generated by each GAT layer. Sibyl concatenates each node's intermediate values to form a final representation. JK layers are frequently used in GNN architectures [53]–[55] as it has been shown that representations formed in the earlier layers of message passing may help with a model ability to generalize [56]. We perform an ablation study which showed a 12% increase in Sibyl's accuracy over networks which did not include a JK layer.

Finally, the nodes must be collated into a feature vector of a fixed size. Sibyl uses a attention-based pooling layer. For each node in the graph, the pool calculates an attention score to weight the node's encoding when summing all nodes together. Attention-based pooling techniques have been shown to be effective in both CNNs and GNNs [49], [57], [58]. Like GATs, they can learn to weigh relevant information highly, and give noisy information low weights. By using both GAT layers and an attention-based pool, Sibyl's GNN can weigh the importance of individual nodes and edges when forming a feature vector. This feature vector is fed to a three layer feed-forward neural network which calculates a score for each solver in the portfolio of solvers.

C. Implementation

We created a prototype implementation of the Sibyl technique and made it publicly available at https://anonymous.4open.science/r/sibyl-884E. To generate program graphs, Sibyl uses the AST generated by pySMT [59], defining a visitor to walk the AST to collect its nodes and edges, along with the data required to form the UD graph.

Sibyl uses the machine learning libraries PyTorch [60] and PyTorch Geometric [61] to implement its various machine learning components. PyTorch Geometric is an extension of PyTorch which has efficient implementations of various popular GNNs and allows user to create their own GNNs. As an ablation study in §V shows found Sibyl performs best with 4 GAT layers. With the exception of the ablation study, all Sibyl networks discussed in this paper consist of 4 GAT layers.

IV. Experimental Design

We have designed and executed several experiments that seek to evaluate Sibyl against the current state-of-the-art in SMT algorithm selection along with several baseline selectors. In doing so, we look to answer the following questions.

TABLE I: Dataset statistics. The logics in each theory category can be found in the SMT-COMP description [13].

Dataset	Theory Categories	Queries	Portfolio
BMC	QF_Bitvec, QF_Equality+Bitvec	100,000	[44], [62]–[66]
SymEx	QF_Bitvec, QF_Equality+Bitvec	91,810	[44], [62]–[66]
SyGuS	Equality+LinearArithm	100,000	[44], [64], [66]–[68]

A. Research Questions

RQ1: For each domain, what portion of queries is the SMT solver which is the fastest overall the fastest on individual queries? *The overall fastest solver is fastest on 2.6% to 38.9% of the benchmarks in a domain.*

RQ2: How does Sibyl's predictions compare to other algorithm selectors on software engineering domains? *Sibyl's predictions are 37.6% to 159.7% better than existing selectors.*

RQ3: How does Sibyl's overhead affect its performance? *Sibyl's overhead is non-negligible, but there is evidence to suggest a more efficient implementation will greatly reduce it.*

RQ4: How do the components of Sibyl's GNN contribute to its performance? *When combined, the core GNN components of Sibyl – GAT layers and a jumping knowledge layer – improve Sibyl's performance substantially.*

B. Baseline Techniques

In order to understand the potential benefits and situate Sibyl in the field of SMT algorithm selection, we compare it against four baseline algorithm selectors for SMT. We could could not compare against Hůla et al. as no artifact of their approach is publicly available, which we confirmed with the authors.

Frequently Fastest Static Selector (FFSS). This selector runs the solver which solved the most individual queries in the training set the fastest. This will allows us to evaluate how the solver which most frequently is fastest performs in general.

Overall Fastest Static Selector (OFSS). This selector runs the solver cumulatively fastest on the entire training set. This will act a baseline to determine if selection is beneficial, or if a single solver will suffice.

Virtual Best Selector (VBS). This selector is the theoretical optimal selector, given the portfolio, i.e., for each individual query, it performs as the fastest solver. While such selection is only possible a posteriori, thus not implementable in practice, it allows comparison between selectors to see how much room for improvement exists.

MachSMT. MachSMT is an algorithm selector that uses feature engineering and random forests to perform selection [18].

Medley Solver. Medley Solver is an algorithm selector which uses feature engineering and reinforcement learning to select SMT solvers [19].

C. Datasets and Domain Portfolios

Application domains. To evaluate the selectors, we have identified four domains in which SMT solvers are used: in *competition* (SMT-COMP), *bounded model checking* (BMC), *symbolic execution* (SymEx), and *Syntax-Guided Synthesis* (SyGuS). Here we describe the way in which each dataset was obtained – which are publicly available [34], [69] – and the domain portfolios – a set of SMT solvers the selectors have access to for a given domain.

Selector Strategies. We wish to evaluate selectors with two strategies: *selection-only* and *selection-in-parallel*. The first strategy is pure algorithm selection: given a query, the selector chooses a solver and runs it. In *selection-in-parallel*, a single solver is run at the same time as the selector. If a query is simple, the single solver will most likely solve the query before the selected solver starts running. If the query is complex, the selector should select the optimal tool which will ideally be faster than the single solver. This strategy is motivated by the observation that, in several practical domains, the solver time follows a heavy-tail distribution, where a large proportion of queries are fast to solve, while the remaining, smaller proportion of queries accounts for most of the cumulative execution time. See Figure 5 in the supplementary material for a histogram of the time it took solvers to solve the queries generated in the different application domains. Because BMC has the highest proportion of slow queries, we evaluate it using the *selection-only* strategy. *Selection-only* is best when the domain has a high proportion of complex queries, as *selection-in-parallel* will waste resources running two solvers which may both take a long time solving a query.

SymEx and SyGuS are both strong candidates for *selection-in-parallel*. Because of their skewness towards fast queries, the parallel solver could save on the cost of selection for a large proportion of queries. We evaluate KLEE with the *selection-in-parallel* strategy. Notably, pure selection can still be beneficial in these skewed populations. To demonstrate this, we evaluate SyGuS with the *selection-only* strategy.

Portfolios. For each domain, the portfolio is decided by the logics in the domain's dataset. Each SMT-COMP division contains a number of solvers which were evaluated in the competition. For each dataset, the portfolio consists of the union of solvers in the datasets' divisions. If multiple versions of a tool were evaluated in the competition, we use the version which competed. If none competed, then we use the most recent. For information on participating solvers, see the competition website [34]. Domain portfolios range from 3 to 8 solvers with an average of 6 solvers. Table I shows the theories, size, and domain portfolio for each software engineering dataset. For SMT-Comp domain portfolios, see Table V in the appendix.

Experimental Subjects Selection: **SMT-COMP.** SMT-COMP-21 evaluates SMT tools on over 54,000 queries in 18 categories of logics in the single query track. From the 18 categories, we omit 2, QF_Strings and QF_Equality+NonLinearArith, as they have less than 500 queries. Since we are training machine learning models that require training data, it is important that there are a significant number of examples to both train and evaluate the selectors. We must also omit categories dealing with floating point logics, such as FPArith and QF_FPArith, as the SMT parsing library we use does not support them [59]; this omission is implementation specific, and there is nothing which leads us to believe the technique would not perform similarly with floats. This leaves 14 categories ranging from

547 to 12,549 queries with 3,599 being the average.

The queries in this dataset cover 59 different SMT logics and were generated by the SMT community for the competition. They reflect different SMT applications, as well as hand-crafted queries designed to challenge solvers. This is the most general of the datasets we will use.

BMC. ESBMC is a C bounded model checker based on an SMT solver backend [12]. It generates queries that seek to prove that a specification holds up to a bound and attempts to extrapolate past the bound using K-Induction. These queries must capture the program semantics and what is required to prove the induction step. ESBMC will increase K until it finds a violation, proves the specification holds, or it times out.

SV-COMP is an annual competition which evaluates software verifiers on benchmarks asserting different types of properties [70]. We ran ESBMC on 11,369 C files which come from the SV-COMP'22 dataset. These C files come from the four major categories of SV-COMP: reach safety, memory safety, termination, and overflow safety [70]. ESBMC supports several SMT solvers. In our experiments, we used ESBMC version 6.8.0 with the solver Bitwuzla [62]. ESBMC supports query dumping with Bitwuzla and it produced queries all solvers in the portfolio supported.

For each C file, we ran ESBMC with a 15 minute timeout, as is done in SV-COMP. It generated nearly 700,000 queries, of which we randomly sampled 100,000 queries to create our dataset. The queries fall into the logics in the QF_Bitvec and QF_Equality+Bitvec divisions in SMT-COMP.

SymEx. KLEE is a symbolic execution engine which uses an SMT solver backend to determine the reachability of program execution states [7]. When interesting states are found (such as the evaluation of a conditional branch or the occurrence of an error), KLEE creates a SMT query that captures the path condition leading to the state to confirm its reachability – and possibly generate an input to witness it.

We ran KLEE version 2.2 on a subset of the GNU coreutils and several other real world programs. We selected 20 of the 107 coreutils which we found produced queries which took solvers (on average) no less than 0.5s to solve. These programs are: `basenc`, `cp`, `date`, `dircolors`, `du`, `md5sum`, `mv`, `numfmt`, `ptx`, `seq`, `sha384sum`, `shuf`, `sleep`, `test`, `timeout`, `tr`, `tsort`, `unexpand`, `uniq`, and `wc`. We selected 5 real world programs which KLEE is often evaluated on [7], [10], [71]. These are: `find`, `gawk`, `grep`, `gzip`, and `make`. KLEE was given 5 hours to run on each program. Like ESBMC, KLEE generates queries that fall in the QF_Bitvec and QF_Equality+Bitvec categories in SMT-COMP.

It is important that training data reflect the population a model will be evaluated on. Since we evaluate SymEx with the selector-in-parallel strategy – i.e., running Sibyl in parallel to KLEE's internal solver, STP [65] – we focus the training and evaluation of Sibyl on the 93,996 KLEE generated queries that took STP, KLEE's default solver, more than 0.5s to solve.

SyGuS Syntax-Guided Synthesis (SyGuS) is a program synthesis method that systematically generates candidate programs from a user-specified formal grammar and then verifies the validity of a candidate using a constraints solver, typically an SMT solver [72]. SyGuS-Comp was an annual competition which evaluated program synthesis tools on a variety of benchmarks dealing with several different logics. DryadSynth is a program synthesis tool which competed in the most recent competition, SyGuS-Comp'19, in two of five major categories, Invariant Synthesis and Conditional Linear Integer Arithmetic, the latter of which it won.

We ran DryadSynth on the benchmarks from the Invariant Synthesis and Conditional Linear Integer Arithmetic categories from SyGuS-Comp'19 with a 20 minute timeout. We excluded programs DryadSynth produced errors on. From the remaining 490 benchmarks, DryadSynth generated nearly 2,000,000 SMT queries in the Equality+LinearArith division from SMT-COMP. As with the BMC, we randomly sample 100,000 queries to make up the dataset.

D. Selector Implementation and Training

In order to train and evaluate our selectors, we need to retrieve labels for the datasets. To do so, we ran all queries on StarExec [73] which provides the community compute resources for tasks related to constraint solving. All nodes used to collect labels ran CentOS 7 on a 2.40 GHz Intel(R) Xeon(R) E5-2609 CPU with 256 GB of RAM. SMT-COMP uses the same servers to run the competition, so we used their results, which are publicly available, to generate labels.

Each query is labeled using a Penalized Average Runtime 2 (Par-2) score. If a solver correctly solves a query, the score is the time it took to solve the query. If the solver reports an incorrect response or exceed the given resources, its score is twice the timeout value. In our studies, the timeout value was 1,200s, the same as SMT-COMP'21. Par-X scores are frequently used by algorithm selectors as they are simple to compute and allow selectors to weigh time and correctness, prioritizing correctness [18], [74], [75].

Datasets were split into training and test sets. Due to their small size, the divisions in the SMT-COMP dataset were split using a 80-20 training-test split. The remaining datasets are much larger, so they were split using a 20-80 split. This will help mitigate against over-fitting. Each selector was then trained on the training set using 10-fold cross validation. All reported results are the average of the ten models trained using cross validation and evaluated on the test set.

Training took place on various machines with different specifications. Training resources are not pertinent to any of our research questions or evaluation metrics, so this poses no threat to validity. Sibyl models were trained for 25 epochs with an initial learning rate of 1e-3. If loss did not decrease after three consecutive epochs, the learning rate was reduced by one order of magnitude. Once the learning rate reached 1e-8 or 25 epochs were reached, training ended. Sibyl uses pairwise margin ranking loss to train models**[Anto:** many GNN concepts introduced here. Maybe add a general reference that defines them**]**. MachSMT and MedleySolver were trained according to the repository provided by the authors [18], [19]. Training times range based on query size, with both MachSMT

TABLE II: For each of the domain, there is no solver which is optimal on all queries. All but two solvers are optimal on some number of queries. VBS shows the potential for improvement via selection is great, with a 8580% improvement over the single best solver in the best case and a 175% improvement over the single best solver in the worst case.

(a) SMT-COMP - QF_Bitvec

Solver	Optimal	Par-2 Score
Bitwuzla	0.1698	467,696
CVC5	0.0046	1,682,948
MathSAT	0.0214	2,790,731
STP	0.1819	526,930
Yices2	0.5687	474,132
Z3	0.0535	3,508,911
VBS	1.0000	61,389

(b) SMT-COMP - QF_Equality+Bitvec

Solver	Optimal	Par-2 Score
Bitwuzla	0.3893	108,291
CVC5	0.0146	367,632
MathSAT	0.0065	306,687
Yices2	0.5747	235,774
Z3	0.0149	355,448
VBS	1.0000	16,309

(c) SMT-COMP - Equality+LinearArith

Solver	Optimal	Par-2 Score
CVC5	0.0468	546,587
iProver	0.0009	9,595,641
SMTInterpol	0.0004	4,622,626
U.Elim+MathSAT	0.0000	11,100,209
Vampire	0.0392	857,754
Verit	0.7107	1,422,231
Z3	0.2020	1,420,451
VBS	1.0000	12,332

(d) BMC

Solver	Optimal	Par-2 Score
Bitwuzla	0.1985	172,640
CVC5	0.0017	9,333,074
MathSAT	0.0126	4,516,230
STP	0.0789	24,686,601
Yices2	0.7049	217,680
Z3	0.0034	10,576,990
VBS	1.0000	45,567

(e) SymEx

Solver	Optimal	Par-2 Score
Bitwuzla	0.1358	254,607
CVC5	0.0023	6,692,003
MathSAT	0.0013	1,702,864
STP	0.2049	37,654
Yices2	0.6212	936,809
Z3	0.0344	1,215,535
VBS	1.0000	21,503

(f) SyGuS

Solver	Optimal	Par-2 Score
CVC5	0.0261	166,811
U.Elim+MathSAT	0.9336	297,170
SMTInterpol	0.0000	2,569,827
veriT	0.0003	239,800,801
Z3	0.0400	2,367,390
VBS	1.0000	1,944

and Sibyl taking several hours to train models on the largest datasets. FFSS, OFSS, and VBS require no training. MedleySolver is an online approach, meaning it learns during evaluation. Still, there is an overhead to the tool, which resulted in evaluation taking roughly as long as any other technique to train and evaluate. To evaluate tool overhead, we ran each tool on identical servers running CentOS with one Intel(R) Xeon(R) Bronze 3104 CPU @ 1.70GHz, an Nvidia GTX 1080Ti GPU and 128GB of RAM.

V. Evaluation

A. Solver Performance Diversity

To address **RQ1**, we begin by observing the performance diversity of the various solvers in each domain's portfolio. For algorithm selection to be an interesting problem, there should be a significant difference in performance between solvers on a significant number of problems.

Table II shows the performance diversity of solvers on all queries in the three divisions of SMT-COMP the software engineering domains lie, QF_Bitvec, QF_Equality+Bitvec, and Equality+LinearArith, and the performance of solvers on the queries generated in the software engineering domains. For each table, the first column lists the solver being evaluated, the second is the portion of queries said solver is optimal (correct and fastest), the third is the Par-2 score of the solver on the dataset (lower is better). We omit examples from the that no solver could solve, as this would only inflate every solver's score. This leaves 65,003, 74,667, and 79,981 queries in the BMC, SymEx, and SyGuS test sets, respectively.

There are several patterns accross all six tables: (1) One solver is optimal on over 50% of the queries; (2) A different solver has the lowest Par-2 score; (3) Each solver is optimal on some number of queries, with the exception of U.Elim+MathSAT and SMTInterpol; and (4) VBS substantially improves on any solver in the Domain Portfolio. Moreover, there is no general "best" solver in any domain. Even solvers with rather high Par-2 scores have several scenarios in which they are optimal. Most important is point 4. While VBS is impractical, it demonstrates the potential benefits of selection [18], [19].

> **RQ1:** In the majority of the SMT-Comp domains and every software engineering domain, the solver which can solve the dataset fastest is the optimal solver on less than half of the queries.

B. Selector Evaluation

In this section, we address **RQ2** by evaluating the selectors on each domain's test set. Once again, we omit examples from the test set that no solver could solve.

SMT-COMP. Table IIIa shows how selectors perform on the SMT-COMP QF_Bitvec category. To see how the solvers perform on the remaining divisions, see our repository [69][**Anto:** make sure the reference to the repo is correct]. In terms of Par-2 score, Sibyl performs on average 24.5% worse than MachSMT, 268.0% better than MedleySolver, 12.1% better than FFSS and OFSS, and 233.9% worse than VBS.

In our evaluation, MedleySolver performs noticeably worse then the evaluation performed in [19]. This is most likely due to our timeout values being orders of magnitude different. In their evaluation, the solver timeout value is at 60s, as opposed to ours which is 1,200s (the same as SMT-Comp). As a result, all solvers can solve more problems, making it harder for their reinforcement learning algorithm to learn when solvers do poorly. In our evaluations, we find MedleySolver randomly

TABLE III: MachSMT is the optimal selector in majority of SMT-Comp categories. Across each software engineering domains, however, Sibyl is the optimal feasible selector. MachSMT and MedleySolver were tuned specifically to SMT-Comp. Sibyl learns to calculate feature vectors, allowing it to specialize based on the training data, while

(a) SMT-COMP QF_Bitvec

Selector	Optimal	Par-2 Score
Sibyl	0.73 ± 0.01	63,807 ± 3,081
MachSMT	0.34 ± 0.09	51,256 ± 6,017
Medley Solver	0.14 ± 0.17	234,831 ± 178,681
FFSS	0.60 ± 0.00	71,498 ± 0
OFSS	0.60 ± 0.00	71,498 ± 0
VBS	1.00 ± 0.00	16,734 ± 0

(b) BMC

Selector	Optimal	Par-2 Score
Sibyl	0.60 ± 0.01	113,495 ± 20,711
MachSMT	0.34 ± 0.09	219,186 ± 28,016
Medley Solver	0.38 ± 0.20	1,140,597 ± 1,277,414
FFSS	0.70 ± 0.00	181,007 ± 0
OFSS	0.20 ± 0.00	142,009 ± 0
VBS	1.00 ± 0.00	37,397 ± 0

(c) SymEx

Selector	Optimal	Par-2 Score
Sibyl	0.69 ± 0.01	19,533 ± 1,081
MachSMT	0.30 ± 0.16	27,765 ± 3,791
Medley Solver	0.33 ± 0.25	254,274 ± 253,911
FFSS	0.62 ± 0.00	742,406 ± 0
OFSS	0.21 ± 0.00	30,230 ± 0
VBS	1.00 ± 0.00	17,211 ± 0

(d) SyGuS

Selector	Optimal	Par-2 Score
Sibyl	0.27 ± 0.06	11,359 ± 7,788
MachSMT	0.93 ± 0.01	18,145 ± 15,606
Medley Solver	0.11 ± 0.27	151,459 ± 53,243
FFSS	0.93 ± 0.00	234,377 ± 0
OFSS	0.26 ± 0.00	127,695 ± 0
VBS	1.00 ± 0.00	1,555 ± 0

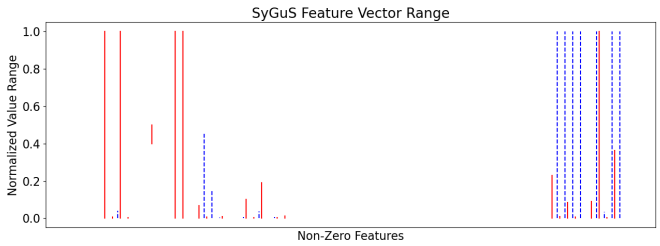

Fig. 3: Normalized engineered feature vector ranges for SyGuS domain, see the supplementary material for BMC and SymEx. Each line represents the normalized range from minimum to maximum values for a given feature for both SE domain generated features (blue, dashed line) and SMT-Comp category generated features (red, solid line). Absent lines mean that feature never appeared in the dataset.

selects a solver, and continues using said solver for 99% of problems, making their solver's performance highly variant.

Of the divisions in SMT-COMP we evaluate the selectors on, Sibyl is the best feasible selector in 2 divisions, QF_Linear RealArith and QF_NonLinearIntArith, on the remaining 12 divisions MachSMT is the best. MachSMT's features were manually engineered for SMT-COMP. As a result, they perform very well on the benchmarks in the dataset. Below, we evaluate the selectors on software engineering domains in which SMT solvers are used to show their performance when trained and evaluated on specific problem classes.

Software Engineering Domains. Table III shows how selectors perform on the three software engineering domains. In terms of Par-2 score, Sibyl performs on average 64.3% better than MachSMT, 1,221.3% better than MedleySolver, 2,031.1% better than FFSS, 469.0% better than OFSS, and 381.8% worse than VBS across the three domains.

In BMC and SymEx, Sibyl makes roughly twice as many optimal choices as the next best, feasible, selector, MachSMT. This reflects in the Par-2 scores where Sibyl is on average 66.7% lower than MachSMT. In SyGuS, however, MachSMT makes nearly 3 times as many optimal selections, yet Sibyl is again better. When Sibyl is making non-optimal choices, it chooses solvers which perform close enough to the optimal solver; on the 7% of non-optimal choices MachSMT makes Sibyl can score 6,786 lower in terms of Par-2 score.

To understand why it is that MachSMT outperforms Sibyl on the SMT-Comp domain but not the Software Engineering domain, we examined the feature vectors from each domain. Figure 3 displays the range of values each feature in the MachSMT's feature vector takes across the entirety of each domain (blue, dashed lines) compared to the SMT-Comp category of the same logic (red, solid line). Each line spans the minimum to maximum value of a given feature, normalized by dividing the values by the maximum value of the feature across all domains. We find that there is a significant difference between the population of features between the SMT-Comp category and a given SE domain.

On average, 35.5% of features in an SMT-Comp category do not appear in the corresponding SE domain. Additionally, each software engineering domain contains on average 8.5% features which do not appear in the SMT-Comp category. In the BMC and SymEx domains, there are 17 features which span 95% of the value range and 22 in the corresponding SMT-Comp categories, with no overlap between these two sets of features. In the SyGuS domain, 8 features span 95% of the value range and in the corresponding SMT-Comp domain there are 12. Once again, there is no overlap between these sets of features. We conjecture that this difference in population results in a worse performance, as the features were selected for a different population.

RQ2: Across all three SE Domains, Sibyl is the best performing, feasible, selector and is on average 63% better than the next best selector.

C. Overhead evaluation

In order to address **RQ3**, we evaluate Sibyl's overhead. That is, we measure the overhead of our implementation of Sibyl to determine the practicality of the technique if it was used by a software engineering tool to select a SMT solver. It would be costly to integrate into existing tools, which are often highly complex with sophisticated solver subsystems, so we chose to study a proxy setting where we use PySMT and study solve times for domain-specific collections of queries.

In practice, software engineering tools build queries on the fly in memory using a programmatic API by traversing a tool-specific representation of the SE problem. To simulate this, we use PySMT's internal representation as a starting point. We compare Sibyl to MachSMT and OFSS as these two are the second fastest solver in at least one category. First, Sibyl and MachSMT generate graphs and feature vectors, respectively,

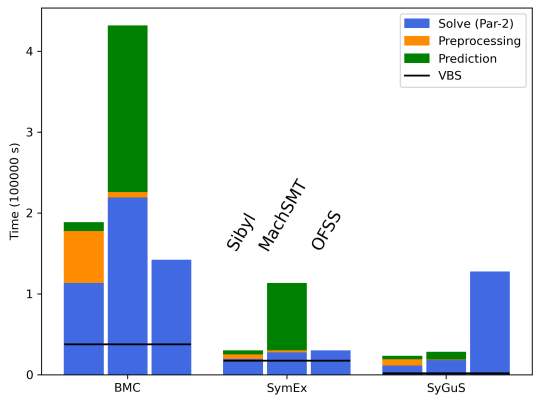

Fig. 4: Overhead evaluation with breakdown of costs of SMT selectors and comparison with OFSS and VBS across three SE domains.

which we call *preprocessing*. Second, they perform *prediction* to select the solver to use. Third, they translate PySMT internal representation to that solvers representation via API calls, which we call *solve*. OFSS performs only *solve*.

Figure 4 shows the performance of each selector on the three software engineering domains. Each bar includes the Par-2 score, preprocessing overhead, and prediction overhead stacked to show the overall performance of the given selector. For each domain, there is a black line indicating the optimal selector performance (VBS). In BMC and SymEx, Sibyl's overhead is high enough that OFSS now solves the test set faster. In the SyGuS domain, Sybil is still marginally better than MachSMT and significantly better than OFSS.

There is variation within and across the domains, but we can observe several broad trends. MachSMT's feature vector extractor is on average 653% faster than Sybil's graph builder. However, Sybil's predictor is 1,151% faster than MachSMT's. This allows Sybil to begin solving faster than MachSMT and, as we have shown, it chooses better solvers.

In the SyGus category, Sibyl is the fastest approach. In the BMC and SymEx categories, Sibyl's overhead is high enough that OFSS is the best selector. This is mainly due to Sibyl's graph building overhead. To investigate whether graph building is an inherent cost, or just due to suboptimal performance in PySMT's pure Python AST walker, we integrated a Sybil graph builder into ESBMC, with graph building implemented in C++ and compiled. We ran it on a corpus of 779,529 queries generated by ESBMC when run on SV-COMP22 and found that graph building took on average 0.012s with a standard deviation of 0.037s. In contrast, the PySMT graph builder took on average 0.801s, with a standard deviation of 2.19s. Extrapolating to the 80,000 queries used in this study graph building cost reduction would make the overall cost of Sibyl 13.1% faster than OFSS.

Further, the VBS line shows the potential improvement that exists for selectors. In the BMC and SyGuS categories, our Python-based prototype implementation of Sibyl has significant room for improvement in predictions which suggests that it could further outperform OFSS in those domains.

TABLE IV: Ablation of Sibyl's GNN evaluated on the SymEx domain. As the number of GAT layers rise, the JK layer becomes more impactful. A network with 3 GAT layers performs best for the SymEx domain.

GAT Layers	Jumping Knowledge	Optimal	Par-2 Score
0	N/A	0.64 ± 0.02	21,063 ± 1,112
1	False	0.66 ± 0.02	21,003 ± 1,761
1	True	0.69 ± 0.01	21,059 ± 1,105
2	False	0.63 ± 0.02	23,148 ± 2,340
2	True	0.69 ± 0.01	20,171 ± 1,207
3	False	0.62 ± 0.02	25,750 ± 6,560
3	True	0.69 ± 0.01	19,233 ± 281
4	False	0.57 ± 0.04	23,980 ± 1,984
4	True	0.69 ± 0.01	19,811 ± 1,093

RQ3: Sibyl outperforms MachSMT. While it's overhead is non-negligeable, it outperforms OFSS in the SyGuS category and has potential to do so also for BMC problems.

D. Ablation Study

In order to answer **RQ4**, we perform an ablation study on Sibyl's GNN. There are two main components to the GNN: GAT Layers and a JK layer. We trained networks on the SymEx domain with 0-4 GAT layers, with and without a JK layer. We chose the SymEx domain as Sibyl performs closest to VBS in this domain. The results are listed in Table IV.

GATs and the JK layer appear to be related in terms of model performance. As GAT layers increase, Par-2 score improves when the JK layer is present and worsens when it is not. Similarly, the benefit of the JK layer tends to become more significant as GAT layers rise. JK layers concatenate the intermediate representations of the graph into a single vector, allowing the model to learn from each intermediate representation. These results imply that there is information the model can use to make accurate predictions from one or more of these intermediate representations, making the JK layer beneficial.

Each additional GAT layer results in a node's representation being calculated from neighbors further away, effectively encoding more of the graph structure in each node. This is more information the model can use when differentiating between queries and looking for similarities.

Depending on the domain, the best network architecture may differ. When determining Sibyl's architecture, we performed a search on the number of GAT layers, and found in general 4 layers performed best across our specialization domains. These results show 3 GAT layers perform marginally better for the SymEx domain. In practice, it can be beneficial to train and evaluate several network architectures to determine which is best for a given population.

RQ4: When combined, both configurable components of Sibyl's GNN have a positive affect on its ability to make predictions.

E. Threats to Validity

A potential threat to the validity of our experiments is the potential for bugs in our implementation. To mitigate this, we perform several sanity checks via assertion statements to ensure certain invariants hold. These include checking that we are recording the SMT graph correctly, labels are between the minimum and maximum Par-2 scores, that the GNN is stable (not producing infinite or NaN values), etc. We used widely adopted libraries to parse SMT inputs and implement our training procedures. Our implementation is available for review at https://anonymous.4open.science/r/sibyl-884E.

Another possible threat to validity is the tools we used to populate our domains. Bounded model checking, symbolic execution, and syntax-guided synthesis are software engineering problems which have garnered much attention. ESBMC and DryadSynth are both competitive tools in their respective domains, performing well in competition [70], [76]–[78]. KLEE is a widely used symbolic execution engine. SMT solving is a known bottleneck in the KLEE pipeline and has been the subject of study and optimization [79]–[81]. This leads us to believe these are strong subjects to study.

VI. Conclusion

We presented Sibyl, an algorithm selection technique based on graph neural networks for SMT solver selection. Sibyl converts SMT queries into graphs which its GNN uses to determine which solver from a portfolio of solvers should be used to solve the query. Because Sibyl learns feature encodings, it requires no domain knowledge to tailor solver selection to the peculiarities of an application domain. A developer need only train a model using data from the domain, without requiring any human feature engineering.

We evaluated Sibyl on over 300,000 benchmarks from SMT-COMP and three datasets comprised of queries generated using three types of software engineering tools: a bounded model checker, a symbolic executor, and a program synthesizer. On these three software engineering domains, we found that selection via Sibyl would solve a sample of queries in these domains between 37.6% to 159.7% better than the state of the art. While our Python-based prototype has a significant overhead for graph building, we demonstrated how such overhead can be drastically reduced integrating graph building within the internal query representation of an analysis tool – ESBMC in our exploratory study – and optimizing it via compilation.

References

[1] M. R. Gadelha, F. R. Monteiro, J. Morse, L. C. Cordeiro, B. Fischer, and D. A. Nicole, "Esbmc 5.0: an industrial-strength c model checker," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 888–891.

[2] H. Lauko, V. Štill, P. Ročkai, and J. Barnat, "Extending divine with symbolic verification using smt," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2019, pp. 204–208.

[3] D. Kroening and M. Tautschnig, "Cbmc–c bounded model checker," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2014, pp. 389–391.

[4] N. Gavrilenko, H. Ponce-de León, F. Furbach, K. Heljanko, and R. Meyer, "Bmc for weak memory models: Relation analysis for compact smt encodings," in *International Conference on Computer Aided Verification*. Springer, 2019, pp. 355–365.

[5] D. Beyer and M. E. Keremoglu, "Cpachecker: A tool for configurable software verification," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 184–190.

[6] M. Zheng, M. S. Rogers, Z. Luo, M. B. Dwyer, and S. F. Siegel, "Civl: formal verification of parallel programs," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 830–835.

[7] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[8] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 380–394.

[9] V. Chipounov, V. Kuznetsov, and G. Candea, "The s2e platform: Design, implementation, and applications," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 1, pp. 1–49, 2012.

[10] H. Palikareva and C. Cadar, "Multi-solver support in symbolic execution," in *International Conference on Computer Aided Verification*. Springer, 2013, pp. 53–68.

[11] L. Cordeiro, B. Fischer, and J. Marques-Silva, "Smt-based bounded model checking for embedded ansi-c software," *IEEE Transactions on Software Engineering*, vol. 38, no. 4, pp. 957–974, 2011.

[12] M. Y. Gadelha, H. I. Ismail, and L. C. Cordeiro, "Handling loops in bounded model checking of c programs via k-induction," *International Journal on Software Tools for Technology Transfer*, vol. 19, no. 1, pp. 97–114, 2017.

[13] H. Barbosa, J. Hoenicke, and A. Hyvarinen, "16th international satisfiability modulo theories competition (smt-comp 2021): Rules and procedures," 2021.

[14] H. Rocha, R. Menezes, L. C. Cordeiro, and R. Barreto, "Map2check: using symbolic execution and fuzzing," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2020, pp. 403–407.

[15] T. Balyo, P. Sanders, and C. Sinz, "Hordesat: A massively parallel portfolio sat solver," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2015, pp. 156–172.

[16] Y. Hamadi, S. Jabbour, and L. Sais, "Manysat: a parallel sat solver," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 6, no. 4, pp. 245–262, 2010.

[17] C. M. Wintersteiger, Y. Hamadi, and L. d. Moura, "A concurrent portfolio approach to smt solving," in *International Conference on Computer Aided Verification*. Springer, 2009, pp. 715–720.

[18] J. Scott, A. Niemetz, M. Preiner, S. Nejati, and V. Ganesh, "Machsmt: a machine learning-based algorithm selector for smt solvers," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2021, pp. 303–325.

[19] N. Pimpalkhare, F. Mora, E. Polgreen, and S. A. Seshia, "Medleysolver: online smt algorithm selection," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2021, pp. 453–470.

[20] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," `www.SMT-LIB.org`, 2016.

[21] R. Jhala and R. Majumdar, "Software model checking," *ACM Computing Surveys (CSUR)*, vol. 41, no. 4, pp. 1–54, 2009.

[22] A. Komuravelli, A. Gurfinkel, and S. Chaki, "Smt-based model checking for recursive programs," *Formal Methods in System Design*, vol. 48, no. 3, pp. 175–205, 2016.

[23] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[24] S. Gulwani, O. Polozov, R. Singh *et al.*, "Program synthesis," *Foundations and Trends® in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017.

[25] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE transactions on neural networks*, vol. 20, no. 1, pp. 61–80, 2008.

[26] W. Fan, Y. Ma, Q. Li, Y. He, E. Zhao, J. Tang, and D. Yin, "Graph neural networks for social recommendation," in *The world wide web conference*, 2019, pp. 417–426.

[27] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *International conference on machine learning*. PMLR, 2017, pp. 1263–1272.

[28] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *stat*, vol. 1050, p. 20, 2017.

[29] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 184–195.

[30] M. Lu, D. Tan, N. Xiong, Z. Chen, and H. Li, "Program classification using gated graph attention neural network for online programming service," *arXiv preprint arXiv:1903.03804*, 2019.

[31] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 261–271.

[32] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, and H. Leather, "Programl: Graph-based deep learning for program optimization and analysis," *arXiv preprint arXiv:2003.10536*, 2020.

[33] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *arXiv preprint arXiv:1711.00740*, 2017.

[34] H. Barbosa, F. Bobot, and J. Hoenicke, "Smt-comp 2021," 2021. [Online]. Available: https://smt-comp.github.io/2021/

[35] D. Beyer and M. Dangl, "Strategy selection for software verification based on boolean features," in *International Symposium on Leveraging Applications of Formal Methods*. Springer, 2018, pp. 144–159.

[36] C. Richter, E. Hüllermeier, M.-C. Jakobs, and H. Wehrheim, "Algorithm selection for software validation based on graph kernels," *Automated Software Engineering*, vol. 27, no. 1, pp. 153–186, 2020.

[37] F. Kuang, S. Zhang, Z. Jin, and W. Xu, "A novel svm by combining kernel principal component analysis and improved chaotic particle swarm optimization for intrusion detection," *Soft Computing*, vol. 19, no. 5, pp. 1187–1199, 2015.

[38] W. L. Hamilton, R. Ying, and J. Leskovec, "Representation learning on graphs: Methods and applications," *arXiv preprint arXiv:1709.05584*, 2017.

[39] J. Hula, D. Mojžíšek, and M. Janota, "Graph neural networks for scheduling of smt solvers," in *2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 2021, pp. 447–451.

[40] Z. Ying, D. Bourgeois, J. You, M. Zitnik, and J. Leskovec, "Gnnexplainer: Generating explanations for graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

[41] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[42] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of computer and system sciences*, vol. 55, no. 1, pp. 119–139, 1997.

[43] J. Vermorel and M. Mohri, "Multi-armed bandit algorithms and empirical evaluation," in *European conference on machine learning*. Springer, 2005, pp. 437–448.

[44] L. d. Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[45] C. Vignac, A. Loukas, and P. Frossard, "Building powerful and equivariant graph neural networks with structural message-passing," *Advances in Neural Information Processing Systems*, vol. 33, pp. 14 143–14 155, 2020.

[46] W. Leeson and M. B. Dwyer, "Algorithm selection for software verification using graph attention networks," *arXiv preprint arXiv:2201.11711*, 2022.

[47] C. Morris, M. Ritzert, M. Fey, W. L. Hamilton, J. E. Lenssen, G. Rattan, and M. Grohe, "Weisfeiler and leman go neural: Higher-order graph neural networks," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, no. 01, 2019, pp. 4602–4609.

[48] K. Xu, C. Li, Y. Tian, T. Sonobe, K.-i. Kawarabayashi, and S. Jegelka, "Representation learning on graphs with jumping knowledge networks," in *International Conference on Machine Learning*. PMLR, 2018, pp. 5453–5462.

[49] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," *arXiv preprint arXiv:1511.05493*, 2015.

[50] X. Wang, H. Ji, C. Shi, B. Wang, Y. Ye, P. Cui, and P. S. Yu, "Heterogeneous graph attention network," in *The world wide web conference*, 2019, pp. 2022–2032.

[51] O. Oktay, J. Schlemper, L. L. Folgoc, M. Lee, M. Heinrich, K. Misawa, K. Mori, S. McDonagh, N. Y. Hammerla, B. Kainz *et al.*, "Attention u-net: Learning where to look for the pancreas," *arXiv preprint arXiv:1804.03999*, 2018.

[52] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE transactions on neural networks and learning systems*, vol. 32, no. 1, pp. 4–24, 2020.

[53] X. Wang, X. He, M. Wang, F. Feng, and T.-S. Chua, "Neural graph collaborative filtering," in *Proceedings of the 42nd international ACM SIGIR conference on Research and development in Information Retrieval*, 2019, pp. 165–174.

[54] V. P. Dwivedi, C. K. Joshi, T. Laurent, Y. Bengio, and X. Bresson, "Benchmarking graph neural networks," *arXiv preprint arXiv:2003.00982*, 2020.

[55] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," *AI Open*, vol. 1, pp. 57–81, 2020.

[56] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" *arXiv preprint arXiv:1810.00826*, 2018.

[57] M. J. Er, Y. Zhang, N. Wang, and M. Pratama, "Attention pooling-based convolutional neural network for sentence modelling," *Information Sciences*, vol. 373, pp. 388–403, 2016.

[58] J. Lee, I. Lee, and J. Kang, "Self-attention graph pooling," in *International conference on machine learning*. PMLR, 2019, pp. 3734–3743.

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

[60] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[61] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[62] A. Niemetz and M. Preiner, "Bitwuzla at the SMT-COMP 2020," *CoRR*, vol. abs/2006.01621, 2020. [Online]. Available: https://arxiv.org/abs/2006.01621

[63] B. Dutertre, "Yices 2.2," in *Computer-Aided Verification (CAV'2014)*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds., vol. 8559. Springer, July 2014, pp. 737–744.

[64] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani, "The MathSAT5 SMT Solver," in *Proceedings of TACAS*, ser. LNCS, N. Piterman and S. Smolka, Eds., vol. 7795. Springer, 2013.

[65] R. Michel, A. Hubaux, V. Ganesh, and P. Heymans, "An smt-based approach to automated configuration," in *SMT Workshop 2012 10th International Workshop on Satisfiability Modulo Theories SMT-COMP*. Citeseer, 2012, p. 107.

[66] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli *et al.*, "cvc5: A versatile and industrial-strength smt solver," *Tools and Algorithms for Construction and Analysis of Systems (TACAS), Lecture Notes in Computer Science*, 2022.

[67] J. Christ, J. Hoenicke, and A. Nutz, "Smtinterpol: An interpolating smt solver," in *International SPIN Workshop on Model Checking of Software*. Springer, 2012, pp. 248–254.

[68] T. Bouton, C. B. de Oliveira, D. Déharbe, P. Fontaine *et al.*, "verit: an open, trustable and efficient smt-solver," in *International Conference on Automated Deduction*. Springer, 2009, pp. 151–156.
[69] Anonymous, "Sibyl datasets," May 2022. [Online]. Available: https://doi.org/10.5281/zenodo.6521827
[70] D. Beyer, "Progress on software verification: Sv-comp 2022," in *Tools and Algorithms for the Construction and Analysis of Systems: 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings, Part II*, vol. 13244. Springer Nature, 2022, p. 375.
[71] J. He, G. Sivanrupan, P. Tsankov, and M. Vechev, "Learning to explore paths for symbolic execution," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2526–2540.
[72] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, *Syntax-guided synthesis*. IEEE, 2013.
[73] A. Stump, G. Sutcliffe, and C. Tinelli, "Starexec: A cross-community infrastructure for logic solving," in *International joint conference on automated reasoning*. Springer, 2014, pp. 367–373.
[74] C. Fawcett and H. H. Hoos, "Analysing differences between algorithm configurations through ablation," *Journal of Heuristics*, vol. 22, no. 4, pp. 431–458, 2016.
[75] R. Amadini, F. Biselli, M. Gabbrielli, T. Liu, and J. Mauro, "Sunny for algorithm selection: a preliminary study," in *CILC*, 2015.
[76] D. Beyer, "Software verification: 10th comparative evaluation (sv-comp 2021)," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2021, pp. 401–422.
[77] R. Alur, D. Fisman, R. Singh, and A. Solar-Lezama, "Sygus-comp 2017: Results and analysis," *arXiv preprint arXiv:1711.11438*, 2017.
[78] K. Huang, X. Qiu, P. Shen, and Y. Wang, "Reconciling enumerative and deductive program synthesis," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 1159–1174.
[79] E. Rakadjiev, T. Shimosawa, H. Mine, and S. Oshima, "Parallel smt solving and concurrent symbolic execution," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 3, 2015, pp. 17–26.
[80] Y. Zhang, Z. Chen, Z. Shuai, T. Zhang, K. Li, and J. Wang, "Multiplex symbolic execution: Exploring multiple paths by solving once," *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 846–857, 2020.
[81] T. Liu, M. Araújo, M. d'Amorim, and M. Taghdiri, "A comparative study of incremental constraint solving approaches in symbolic execution," in *Haifa Verification Conference*. Springer, 2014, pp. 284–299.
[82] M. Barth, D. Dietsch, L. Fichtner, M. Heizmann, and A. Podelski, "Ultimate eliminator at smt-comp 2021," 2021.
[83] G. Reger, M. Suda, and A. Voronkov, "Instantiation and pretending to be an smt solver with vampire," in *Proceedings of the 15th International Workshop on Satisfiability Modulo Theories, CEUR Workshop Proceedings*, vol. 1889, 2017, pp. 63–75.
[84] K. Korovin, "iprover–an instantiation-based theorem prover for first-order logic (system description)," in *International Joint Conference on Automated Reasoning*. Springer, 2008, pp. 292–298.
[85] S. Graham-Lengrand, "Yices-qs, an extension of yices for quantified satisfiability," 2021.
[86] M. Blicha, A. E. Hyvärinen, M. Marescotti, and N. Sharygina, "The opensmt solver in smt-comp 2020," 2021.
[87] S. Cruanes and G. Bury, "Mc2," 2021.
[88] S. CAI, B. LI, and X. ZHANG, "Yicesls on smt comp2021," 2021.
[89] M. Brockschmidt, F. Frohn, C. Fuhs, J. Giesl, J. Hensel, P. Schneider-Kamp, T. Ströder, and R. Thiemann, "Aprove at smt-comp 2020," 2020.
[90] F. Corzilius, G. Kremer, S. Junges, S. Schupp, and E. Ábrahám, "Smt-rat: An open source c++ toolbox for strategic and parallel smt solving," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2015, pp. 360–368.
[91] H. Barbosa, D. El Ouraoui, P. Fontaine, and H.-J. Schurr, "verit and verit+ rasat+ redlog: System description for smt-comp 2019," 2019.

TABLE V: Dataset statistics. The logics in each theory category can be found in the SMT-COMP description [13].

Dataset	Theory Categories	Queries	Portfolio
SV-Comp'21	Arith	1,188	[44], [66]–[68], [82]–[85]
	Bitvec	857	[44], [66], [82], [85]
	Equality	1,409	[44], [63], [66]–[68], [82]–[84]
	Equality+LinearArith	12,549	[44], [64], [66]–[68], [83], [84]
	Equality+MachineArith	547	[44], [66], [82]
	Equality+NonLinearArith	1,600	[44], [66], [67], [82]–[84]
	QF_Bitvec	8,748	[44], [62]–[66]
	QF_Equality	4,229	[44], [63], [64], [66]–[68], [86]
	QF_Equality+Bitvec	3,206	[44], [62]–[64], [66]
	QF_Equality+LinearArith	1,960	[44], [63], [64], [66]–[68], [87]
	QF_LinearIntArith	4,156	[44], [63], [64], [66]–[68], [86], [88]
	QF_LinearRealArith	725	[44], [63], [64], [66]–[68], [86], [87]
	QF_NonLinearIntArith	9,112	[44], [63], [64], [66], [89], [90]
	QF_NonLinearRealArith	2,210	[44], [63], [64], [66], [90], [91]

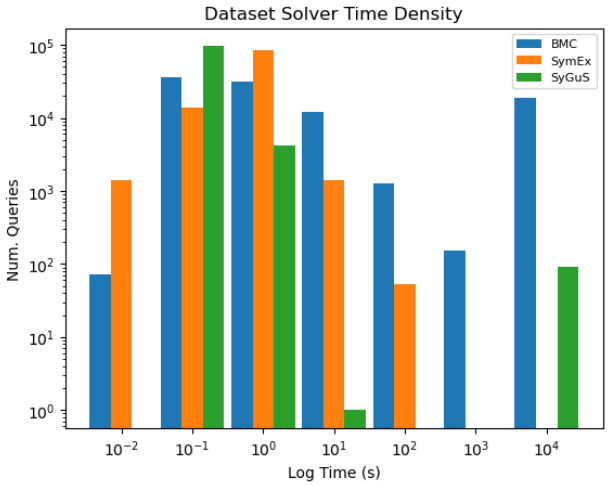

Fig. 5: Query Time Distribution for each software engineering domain. ESBMC has an relatively even mix of fast and slow queries. KLEE and DryadSynth generated far more fast queries than slow queries. DryadSynth generates nearly 100 queries which time out, while KLEE generates none.

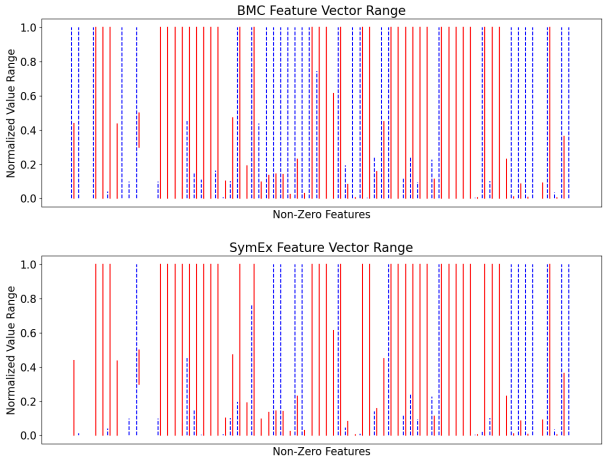

Fig. 6: Normalized engineered feature vector ranges for BMC and SymEx. Each line represents the normalized range from minimum to maximum values for a given feature. Absent lines mean that feature never appeared in the dataset.