

Enhancing Loop-Invariant Synthesis via Reinforcement Learning

Takeshi Tsukada
Chiba University, Japan
tsukada@math.s.chiba-u.ac.jp

Taro Sekiyama
National Institute of Informatics, Japan
tsukiyama@acm.org

Hiroshi Unno
University of Tsukuba, Japan
uhiro@logic.cs.tsukuba.ac.jp

Kohei Suenaga
Kyoto University, Japan
ksuenaga@fos.kuis.kyoto-u.ac.jp

Abstract—*Loop-invariant synthesis is the basis of every program verification procedure. Due to its undecidability in general, a tool for invariant synthesis necessarily uses heuristics. Despite the common belief that the design of heuristics is vital for the effective performance of a verifier, little work has been performed toward obtaining the optimal heuristics for each invariant-synthesis tool. Instead, developers have hand-tuned the heuristics of tools.*

This study demonstrates that we can effectively and automatically learn a good heuristic via reinforcement learning for an invariant synthesizer PCSat. Our experiment shows that PCSat combined with the heuristic learned by reinforcement learning outperforms the state-of-the-art solvers for this task. To the best of our knowledge, this is the first work that investigates learning the heuristics of an invariant synthesis tool.

I. INTRODUCTION

Static formal verification, which is a method of verifying software before their execution based on mathematically robust theories, is gaining more attention owing to the increasing impact of software malfunctions. For its application to real-world software, its efficiency is of paramount importance.

One of the most important static-verification problems is the *partial correctness* verification: given a program c and logical formulae φ_{pre} and φ_{post} , deciding whether φ_{pre} and φ_{post} are the correct *precondition* and *postcondition* of c , respectively. A verification procedure needs to either prove or disprove the following: if c is executed from an initial state that satisfies φ_{pre} and terminates, the final state satisfies φ_{post} .

We explain the partial correctness verification using the following program c_1 :

while $y > 0$ **do** $x \leftarrow x + 1; y \leftarrow y - 1$ **done**.

In this program, $x \leftarrow e$ is a command that updates the value of x to the value of e . This program repeatedly increments x and decrements y as long as y is positive. y is supposed to be initialized with some non-negative number z . Suppose that we are given $\varphi_{pre} := x = 0 \wedge y = z \wedge z \geq 0$ and $\varphi_{post} := x = z$. The partial correctness verification problem for c_1 is to show the following property: if c_1 starts execution with x being 0, z being non-negative, and y being equal to z , and then terminates, then x is equal to z at the end. Proving that it

holds for any initial values that satisfy the precondition is not a trivial task, and this is the goal of static formal verification.

It is known that a key to solving a partial correctness problem is discovering an appropriate *loop invariant*, or simply an *invariant* [1]. A formula φ_{inv} is called an invariant if it holds every time a program execution reaches the entry of the **while** loop. For example, in c_1 , a formula $x + y = z \wedge y \geq 0$ is an invariant. Owing to the importance of an invariant, methods for solving an *invariant synthesis problem (ISP)* have received considerable research interest in the programming verification community. Unfortunately, the ISP is an undecidable problem in general, as it easily follows from the undecidability of the partial correctness verification [1]; therefore, a method for solving the ISP must use a *heuristic* to search for an invariant.

Template-Based Invariant Synthesis.

Template-based invariant synthesis is one of the popular strategies for invariant synthesis. In this method, a tool controls the search space for solutions to a given ISP instance by using a data structure called a *template*.

Let us briefly explain how PCSat [2, 3], a template-based invariant synthesis tool, works to exemplify the behavior of a template-based invariant-synthesis procedure; we explain the behavior in detail in Section III. PCSat is based on a method called *Counter-Example-Guided Inductive Synthesis (CEGIS)* [4]. A tool based on this method repetitively *guesses* an invariant and *proves* or *refutes* the correctness of the guess by using an SMT solver such as Z3 [5]; this repetition is called a CEGIS loop. If a guess is refuted, then PCSat refines the guess based on the counterexample.

PCSat chooses a guess from a set of candidate invariants expressed by a template, which is a predicate that contains parameters. PCSat obtains a guess by instantiating the parameters with concrete values so that the instantiated guess does not contradict the counterexamples obtained so far.

If there is no possible instantiation of the parameters, which means that the candidate invariant set is empty, PCSat *extends* the template to a more expressive one and starts the CEGIS loop again. In a template extension, PCSat chooses the next template heuristically. The heuristic computes the next template from the internal state of PCSat, which includes

information on how a template is extended last time and the responses from the SMT solver during the CEGIS loop.

Template-based tools heuristically update a template during a run to expand the search space. We applied RL to learn an effective heuristic for updating a template. We reformulated the invariant-synthesis procedure of PCSat as an RL problem.

The quality of a heuristic is as crucial as designing a smart invariant-synthesis procedure for the performance of an invariant synthesis tool. However, the program verification research community has not devoted significant attention in this regard; instead, the heuristics of an invariant synthesis tool is manually tuned by a developer, sometimes in an ad-hoc manner. Indeed, the heuristic currently used in PCSat is the one that is hand-tuned by the leading member of the research team.

Our Contribution.

We demonstrate the usefulness of machine learning to discover an effective heuristic for an invariant synthesis tool. We apply reinforcement learning (RL) [6] to learn a heuristic of an invariant-synthesis tool. The performance of PCSat with the learned heuristic measured on standard benchmarks outperforms that of state-of-the-art solvers including the original PCSat and CVC4 [7, 8, 9]. This result implies that (1) improving the heuristic of an invariant-synthesis tool is as effective as improving the synthesis algorithm, and (2) RL is a helpful tool to find a good heuristic.

Concretely, to evaluate the effectiveness of our approach, we conducted several experiments using the benchmark of the invariant synthesis track of a synthesis competition SyGuS-Comp [10]. First, we evaluated whether RL can learn an effective policy to solve a set of ISP instances. To this end, we learned a policy using the benchmark called XC included in SyGuS-Comp 2019 and evaluated it on the same dataset. PCSat combined with the obtained policy outperformed the original PCSat and CVC4, the winner of SyGuS-Comp 2019. This result indicates that RL is indeed helpful for learning a heuristic to solve a *given* set of invariant synthesis problems efficiently.

We also confirmed that a heuristic learned using a dataset generalizes well to another dataset; we learned a policy with SyGuS-Comp 2018 and evaluated it with XC. The resulting policy also outperforms the original PCSat and CVC4. In particular, the policy learned with the original PCSat with the SyGuS-Comp 2018 dataset improves the performance of the newest version of PCSat on the XC dataset, which already outperforms CVC4.

Notice that we *do not* use RL to simply tune a global parameter of PCSat, which could be done using the existing techniques for automated hyperparameter optimization [11, 12]. Instead, we used RL for learning a good heuristic, which is a function from internal states to template-extending actions that leads to an efficient template-based invariant synthesis. This is much more non-trivial than simply applying an existing automated configuration-tuning tool to improve the performance of PCSat; we needed, for example, to design the state space

on which RL works that abstracts the behavior of PCSat for successfully applying RL to our problem..

STRUCTURE OF THE PAPER.

The rest of this paper is structured as follows. Section I-A reviews the related work. Section II sets up the notations (Section II-A) and describes how the ISP can be formalized as a constraint-solving problem (II-B). Section III explains CEGIS (Section III-A) and a template-based approach to CEGIS (Section III-B); we also describe the heuristic to decide which template to be used there (Section III-C). Section IV introduces how we reduce the template-update heuristic learning to an RL problem. Section V summarizes the experiments for the evaluation of our methods. We conclude in Section VI.

A. Related Work

Learning loop invariants.: Our work is based on CEGIS [4], a data-driven approach to constraint solving that can also be applied to the ISP. CEGIS repeats the guess-and-refute loop described above. We will overview how CEGIS works in Section III-A, where we explain how the ISP can be solved via CEGIS.

Whereas decision-tree learning has been a popular method to guess a candidate solution from gathered data in CEGIS [13, 14, 15, 16, 17], PCSat uses a template-based approach [18, 19]; we will explain this approach in Section III-B in detail. Although the template-based approach is advantageous in that it adaptively adjust atomic formulae to be used in a candidate solution, it requires careful manual tuning of the heuristic for deciding the shape of a candidate solution. We overcome this challenge by learning an effective heuristic leveraging RL.

Si et al. [20, 21] proposed CODE2INV, a framework to learn loop invariants with deep learning and RL. CODE2INV uses graph neural networks to encode the information of a program and synthesizes loop invariants using a syntax-directed decoder, leveraging RL to guide the encoder-decoder process. In contrast to their approach that applied machine learning to synthesize an invariant directly from a program, we applied RL to improve the heuristic used in an existing program-verification tool. Our approach resulted in a solver that outperforms the state-of-the-art tools. Their study and ours also differ in the evaluation of the obtained solver. Si et al. evaluated the performance of their solver in the numbers of queries to Z3 [5], an automated theorem prover adopted by most of the program verification tools, instead of measuring the actual running time. Although the running cost of Z3 is dominant in most program verification, CODE2INV conducts online learning of a neural network, which incurs non-trivial additional cost to the performance. On the contrary, the performance of our solver was evaluated in the wall-clock time.

Learning heuristics for theorem proving.: The usefulness of learning heuristics for theorem proving has been demonstrated in various automated theorem proving techniques including CDCL for SAT [22, 23, 24], strategies for SMT [25], connection tableau [26], and incremental determinization for

QBF [24, 27]. These previous studies applied machine learning to enhance the proof search. We cannot directly apply these theorem proving techniques to the ISP since we need to synthesize a solution to the ISP, an appropriate predicate over integers, in addition to deciding the validity of a given formula.

Learning vector embedding of programs and logical formulae. Extensive research has been conducted on learning embedding of programs and logical formulae through neural networks such as LSTMs [28, 29, 30], tree-based neural networks [31, 32, 33, 34], graph neural networks [35, 36, 37, 38], and a path-based attention model [39]. The present work does not learn nor use embedding of invariants or programs. It is an interesting future direction to investigate whether using embedding serves for any further improvement.

II. PRELIMINARIES

A. Notations

We write \mathbf{X} for a finite sequence X_1, \dots, X_n . We use symbols φ and ψ for first-order formulae over integers. We often write $\varphi(\mathbf{x})$ to express that φ may depend on the variables \mathbf{x} ; if it does not contain any free variables, it is called a *ground* predicate. For this φ and integers \mathbf{n} , we write $\varphi(\mathbf{n})$ for the (ground) predicate obtained by substituting \mathbf{n} to \mathbf{x} in φ .

In this paper, we use a constraint, denoted by symbols $\Phi(F(\mathbf{x}))$ and $\mathcal{E}(F(\mathbf{x}))$, that contains a predicate variable $F(\mathbf{x})$. For example, $\forall x. F(x) \implies F(x-1)$ is a constraint that contains a predicate variable F ; this constraint holds if we set F to $x \leq 0$ whereas it does not hold if we set F to $x \geq 0$. We write $\Phi(\varphi)$ for a predicate that obtained by substituting φ to F in Φ .

We use symbol σ for a mapping from variables to integers; this mapping represents an assignment of values to variables. We write $\sigma(x)$ for a value of x in σ and $\sigma(\varphi)$ for the predicate obtained by replacing every variable in φ with its values in σ . For example, if $\sigma := \{x \mapsto 0, y \mapsto 1\}$, then $\sigma(y) = 1$; $\sigma(x < 5 \wedge y > 5) = (0 < 5 \wedge 1 > 5)$, which does not hold.

B. Invariant Synthesis as CHC solving

As we mentioned in Section I, the key to solving a partial correctness verification problem is to find an appropriate invariant. One can check that $\varphi_{inv} := x + y = z \wedge y \geq 0$ is an invariant of the program c_1 in Section I as follows. When a program execution reaches the loop entry for the first time, φ_{inv} holds because the φ_{pre} holds there and φ_{pre} implies φ_{inv} . It is easy to observe that, if the program execution reaches the loop entry and the loop body $c_{body} := (x \leftarrow x+1; y \leftarrow y-1)$ is executed once, in which case $y > 0$ holds, φ_{inv} remains to hold after the execution of c_{body} . Therefore, the condition φ_{inv} is an invariant of this loop. Furthermore, if the program execution reaches the end of c_1 , in which case $y \leq 0$ holds since, otherwise, the loop should have been executed at least one more time, the postcondition $x = z$ holds since $\varphi_{inv} \wedge y \leq 0$ implies $x = z$.

Then, we can observe that the condition for a predicate to be an invariant of c_1 is that it satisfies the condition

$\forall xyz. \Phi_1(F(x, y, z))$ over the predicate variable F , where $\Phi_1(F(x, y, z))$ is defined as follows.

$$\begin{aligned} x = 0 \wedge y = z \wedge z \geq 0 &\implies F(x, y, z) && \wedge \\ y > 0 \wedge F(x, y, z) &\implies F(x+1, y-1, z) && \wedge \\ y \leq 0 \wedge F(x, y, z) &\implies x = z. \end{aligned} \quad (1)$$

The first conjunct ensures that $F(x, y, z)$ is implied by the initial condition $x = 0 \wedge y = z \wedge z \geq 0$; the second conjunct ensures that the predicate $F(x, y, z)$ is preserved by one iteration of the loop if the guard condition ($y > 0$) of the loop holds; and the third conjunct ensures that $F(x, y, z)$ implies the postcondition $x = z$ if the loop terminates. The loop invariant $x + y = z \wedge y \geq 0$ satisfies $\forall xyz. \Phi_1(F(x, y, z))$.

In general, it is well known that an ISP is equivalent to solving the constraint of the form $\Phi(F(\mathbf{x}))$ over a predicate variable F , where $\Phi(F(\mathbf{x}))$ is defined by

$$\begin{aligned} \varphi_{pre} &\implies F(\mathbf{x}) && \wedge \\ \bigwedge_{i=0}^N (\varphi_i \wedge F(\mathbf{x}) &\implies F(\mathbf{e})) && \wedge \\ \varphi' \wedge F(\mathbf{x}) &\implies \varphi_{post}, \end{aligned}$$

\mathbf{x} is a finite sequence of variables, and \mathbf{e} is a finite sequence of expressions. A constraint of this form is called a *linear Constrained Horn Clause (linear CHC)* (or, simply *CHC* in this paper). A *solution* to the above constraint is a logical formula $\varphi_{sol}(\mathbf{x})$ to be substituted to F such that $\forall \mathbf{x}. \varphi_{sol}(\mathbf{x})$ is valid.

III. TEMPLATE-BASED CEGIS FOR ISP

A. CounterExample Guided Inductive Synthesis (CEGIS) for ISP

CounterExample Guided Inductive Synthesis (CEGIS) [4] is a popular approach to CHC solving. It solves a given CHC $\forall \mathbf{x}. \Phi(F(\mathbf{x}))$ via interactions between a *learner* (L) and a *teacher* (T). Its high-level workflow is described as follows.

- L maintains a set $\mathcal{E}(F(\mathbf{x}))$ of ground constraints (i.e., constraints over F that do not contain any program variable); $\mathcal{E}(F(\mathbf{x}))$ is called *example instances*. The set $\mathcal{E}(F(\mathbf{x}))$ consists of constraints that a solution needs to satisfy. Then, L synthesizes a candidate solution $\psi(\mathbf{x})$ such that every constraint in $\mathcal{E}(\psi(\mathbf{x}))$ is valid where $\mathcal{E}(\psi(\mathbf{x}))$ is the set obtained by substituting $\psi(\mathbf{x})$ to every occurrence of $F(\mathbf{x})$. Notice that L does not access the CHC $\forall \mathbf{x}. \Phi(F(\mathbf{x}))$. L then sends $\psi(\mathbf{x})$ to T .
- T checks whether $\psi(\mathbf{x})$ is a real solution to $\forall \mathbf{x}. \Phi(F(\mathbf{x}))$ by checking whether $\neg \Phi(\psi(\mathbf{x}))$ is satisfiable or not where $\Phi(\psi(\mathbf{x}))$ is a predicate obtained by substituting $\psi(\mathbf{x})$ to $F(\mathbf{x})$ in $\Phi(F(\mathbf{x}))$. Most solvers implement this step by querying the satisfiability of $\Phi(\psi(\mathbf{x}))$ to an off-the-shelf solver such as Z3. If it is unsatisfiable, then it follows that $\forall \mathbf{x}. \Phi(\psi(\mathbf{x}))$ is valid; therefore, $\psi(\mathbf{x})$ is a real solution to the CHC $\forall \mathbf{x}. \Phi(F(\mathbf{x}))$. Otherwise, there is a vector of constants \mathbf{r} such that $\neg \Phi(\psi(\mathbf{r}))$ holds; therefore, $\forall \mathbf{x}. \Phi(\psi(\mathbf{x}))$ does not hold. Then, T sends $\Phi(F(\mathbf{r}))$ to L as the example instance to be satisfied; notice that

$\Phi(F(\mathbf{r}))$ is a ground constraint over F . L adds it to the example instance set $\mathcal{E}(F)$ and proposes a new candidate solution again.

For example, the CHC c_1 is solved by CEGIS as follows.

- L starts from $\mathcal{E}(F) = \emptyset$ and proposes $F(\mathbf{x}) = \top$ as the initial solution. T discovers that $(x, y, z) = (0, 0, 1)$ is a counterexample (cex) to c_1 since it falsifies the last conjunct; then L receives $\neg F(0, 0, 1)$ as a new example instance, which is equivalent to $\Phi(F(0, 0, 1))$, and adds it to $\mathcal{E}(F)$.
- Suppose L then proposes $x = z$ as a candidate, which satisfies $\neg F(0, 0, 1)$. Then, T discovers a cex $(0, 1, 1)$ since it falsifies the first conjunct. Therefore, $F(0, 1, 1)$ (equivalent to the first conjunct) and $F(0, 1, 1) \implies F(1, 0, 1)$ (equivalent to the second conjunct) are added to $\mathcal{E}(F)$.
- Then, suppose L proposes $x + y = z$. T discovers a cex $(0, -1, -1)$. Therefore, $\neg F(0, -1, -1)$, which is equivalent to the last conjunct, is added to $\mathcal{E}(F)$.
- Suppose L proposes $x + y = z \wedge y \geq 0$. T finds that there is no cex to this solution; therefore, this is a real solution to c_1 .

B. Template-Based Approach to CEGIS

In each step of a CEGIS-based CHC solving, a learner needs to find a candidate solution that satisfies all the constraints in the current example instance $\mathcal{E}(F)$. One of the strategies to implement the candidate-solution discovery, which is also used by PCSat [2, 3], is called a *template-based* approach.

A template-based learner works as follows. It maintains an example instance $\mathcal{E}(F)$ and a template of a solution $\psi(\mathbf{a}, \mathbf{x})$, which contains *parameters* \mathbf{a} in addition to the program variables \mathbf{x} . Since $\mathcal{E}(F)$ consists of ground constraints on F , by substituting $\psi(\mathbf{a}, \mathbf{x})$ to F in $\mathcal{E}(F)$, the learner obtains constraints $C(\mathbf{a}) := \mathcal{E}(\psi(\mathbf{a}, \mathbf{x}))$ over the parameters \mathbf{a} . If $C(\mathbf{a})$ is satisfiable and α is a satisfying assignment to \mathbf{a} , then $\psi(\alpha, \mathbf{x})$ is a candidate solution that satisfies every constraint in $\mathcal{E}(F)$; if this is the case, then the learner sends $\psi(\alpha, \mathbf{x})$ to the teacher. The satisfiability of $C(\mathbf{a})$ can be checked by using an SMT solver. Otherwise, the learner heuristically updates the template and use it to discover a new candidate solution.

Notice that a template $\psi(\mathbf{a}, \mathbf{x})$ determines a *set* of candidate solutions, members of which is an instantiation of \mathbf{a} in $\psi(\mathbf{a}, \mathbf{x})$ to concrete values. Therefore, a template in the template-based approach determines the search space for a candidate solution of a given constraint. A template is said to be more *expressive* than another if the set of formulae that is obtained by instantiating the parameters in the latter is a subset of the former.

There is a trade-off between the expressiveness of a template and the efficiency of a learner [40]. If one uses an expressive template, there is more chance that there is a true solution that can be obtained by instantiating this template. However, the SMT constraint $C(\mathbf{a})$ generated by using an expressive template tends to be complex, which incurs performance

Algorithm 1 Template-based learner

```

1:  $\psi(\mathbf{a}, \mathbf{x}) \leftarrow$  the initial template
2:  $\mathcal{E}(F) \leftarrow \emptyset$ 
3: while Timeout is not reached do
4:    $C(\mathbf{a}) \leftarrow \mathcal{E}(\psi(\mathbf{a}, \mathbf{x}))$ 
5:    $r_1 \leftarrow \text{SMT}(C(\mathbf{a}))$ 
6:   if  $r_1 = \text{Sat}(\sigma)$  then
7:     Instantiate parameters
8:      $\alpha \leftarrow \sigma(\mathbf{a})$ 
9:     Send the candidate solution to the teacher
10:     $r_2 \leftarrow \text{Teacher}(\psi(\alpha, \mathbf{x}))$ 
11:    Check whether  $\psi_1$  is a real solution
12:    if  $r_2 = \text{Valid}$  then
13:      Return  $\psi(\alpha, \mathbf{x})$  as a real solution
14:    else if  $r_2 = \text{Cex}(\mathcal{E}'(F))$  then
15:      Cex is found; new example instance is received.
16:       $\mathcal{E}(F) \leftarrow \mathcal{E}(F) \cup \mathcal{E}'(F)$ 
17:    end if
18:    else if  $r = \text{Unsat}(I)$  then
19:       $\psi(\mathbf{a}, \mathbf{x}) \leftarrow \text{ChangeTempl}(\psi(\mathbf{a}, \mathbf{x}), I)$ 
20:    end if
21: end while

```

degradation of SMT solving and therefore a learner. In deciding which template to be used, it is crucial to find a sweet spot that addresses this trade-off.

Algorithm 1 is a typical template-based learner for CHC solving. This procedure uses the following subprocedures:

- $\text{SMT}(C(\mathbf{a}))$: Decides whether the predicate $C(\mathbf{a})$ is satisfiable or not by an SMT solver. If it is satisfiable, then it returns $\text{Sat}(\sigma)$ where σ is a value assignment to \mathbf{a} in $C(\mathbf{a})$ such that $\sigma(C(\mathbf{a}))$ is valid. If not, it returns $\text{Unsat}(I)$, where I is a collection of various information on the behavior of the decision procedure (e.g., consumed time and memory). I also includes explanations why $C(\mathbf{a})$ is unsatisfiable such as an unsat core, which is an unsatisfiable subconstraint of $C(\mathbf{a})$. We write $I.\text{uc}$ for the unsat core contained in I .
- $\text{Teacher}(\varphi(\mathbf{x}))$: Sends the candidate solution $\varphi(\mathbf{x})$ to the teacher and let it decide whether it is a real solution. If $\varphi(\mathbf{x})$ is indeed a solution, then the teacher returns *Valid*. Otherwise, the teacher returns $\text{Cex}(\mathcal{E}(F))$, where $\mathcal{E}(F)$ is the new example instance to be satisfied by the learner.
- $\text{ChangeTempl}(\psi(\mathbf{a}, \mathbf{x}), I)$: Heuristically updates the template $\psi(\mathbf{a}, \mathbf{x})$ to a new one based on the information I returned by the SMT solver.

C. Learner Implemented in PCSat

PCSat [2, 3] is one of the tools for CHC solving based on template-based CEGIS. It uses a family of template $\psi_{M,N,P,Q}(\mathbf{a}, \mathbf{x})$, in which each template is determined by the values M, N, P and Q ; here, $\mathbf{x} := (x_1, \dots, x_L)^T$. The parameter \mathbf{a} used in these templates consists of the following parameters.

- *Coefficient parameters* $(\mathbf{a}^{(ij)})_{1 \leq i \leq M, 1 \leq j \leq N}$ where $\mathbf{a}^{(ij)} := (a^{(ij)})$ for each i and j .
- *Constant parameters* $(c^{(ij)})_{1 \leq i \leq M, 1 \leq j \leq N}$ for each i and j .

The template family $\psi_{M,N,P,Q}(\mathbf{a}, \mathbf{x})$ is defined as follows using these parameters:

$$\bigvee_{i=1}^M \bigwedge_{j=1}^N (\mathbf{a}^{(ij)})^T \mathbf{x} \geq c^{(ij)} \wedge \bigwedge_{i=1}^M \bigwedge_{j=1}^N \sum_{k=1}^L |a_k^{(ij)}| \leq P \wedge \bigwedge_{i=1}^M \bigwedge_{j=1}^N |c^{(ij)}| \leq Q.$$

We explain the above definition in the following

- The subformula $\bigvee_{i=1}^M \bigwedge_{j=1}^N \mathbf{a}^{(ij)} \mathbf{x} \geq c^{(ij)}$ is a boolean combination of linear inequalities $\mathbf{a}^{(ij)} \mathbf{x} \geq c^{(ij)}$ expressed in a disjunctive normal form (DNF). The parameter M (resp. N) is the number of disjunctions (resp. conjunctions in each disjunct) in this template; therefore, the larger they are, the more expressive the template is.
- The subformula $\bigwedge_{i=1}^M \bigwedge_{j=1}^N \sum_{k=1}^L |a_k^{(ij)}| \leq P$ bounds the sum of the absolute values of every coefficient in each linear inequality (i.e., the L^1 norm of each $\mathbf{a}^{(ij)}$). The bound P is a natural number or ∞ ; if $P = \infty$, then the coefficients may be any number. The larger P is, the more expressive the template is.
- The subformula $\bigwedge_{i=1}^M \bigwedge_{j=1}^N |c^{(ij)}| \leq Q$ bounds the absolute value of every constant $c^{(ij)}$ in each linear inequality. The bound Q is a natural number or ∞ . The larger Q is, the more expressive the template is.

PCSat starts with a template obtained by $(M, N, P, Q) = (1, 1, 1, 0)$. Then, it decides how to update the template from the unsat core $I.uc$ as follows. One of M and N is chosen alternatively and incremented by 1; PCSat first chooses N . If $I.uc$ mentions P , then P is incremented by 1. If $I.uc$ mentions Q and $Q < 3$, then Q is incremented by 1; if $Q \geq 3$, then Q is set to ∞ .

IV. TEMPLATE-UPDATE HEURISTIC LEARNING AS AN RL PROBLEM

This section describes how we reduce the problem of template-update strategy learning for *ChangeTempl* as a reinforcement learning (RL) problem.

The goal of reinforcement learning is to control a system, usually called the *environment*, to achieve a long-term goal. In our setting, the environment is an implementation of Algorithm 1 that calls an external function *ChangeTempl*. The controller, usually called the *agent*, is the function *ChangeTempl* in this setting; it tells the environment the shape of the template that should be tried in the next loop. The long-term goal is to find a solution of a given CHC (preferably in a short time).

In this formulation, the problem of finding a good template-update heuristics is to learn a well-behaved agent. We model the interaction between Algorithm 1 and the function

ChangeTempl as a Markov decision process and exploit learning algorithms for this setting.

A. Preliminaries: Reinforcement Learning

The environment is specified by a Markov decision process.

A *Markov decision process* is specified by the following data:

- a set \mathcal{S} of *states*,
- an *initial state* $s_0 \in \mathcal{S}$,
- a subset $\mathcal{R} \subseteq \mathbb{R}$ of reals, called *rewards*,
- a finite set \mathcal{A} of *actions*, and
- a *dynamics function* $p : \mathcal{S} \times \mathcal{A} \times \mathcal{R} \times (\mathcal{S} \cup \{\star\}) \rightarrow [0, 1]$.

We write $p(r, s' | s, a)$ for $p(s, a, r, s')$. The dynamics function specifies a probability distribution for each $(s, a) \in \mathcal{S} \times \mathcal{A}$, that means,

$$\sum_{s' \in \mathcal{S} \cup \{\star\}} \sum_{r \in \mathcal{R}} p(r, s' | s, a) = 1, \quad \text{for every } (s, a) \in \mathcal{S} \times \mathcal{A}.$$

\star is a special symbol meaning termination; if the current state is s and the action is a , then $\sum_{r \in \mathcal{R}} p(r, \star | s, a)$ is the probability of termination at this time step.

The agent's behaviour is described by a *policy*, which is a function $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ such that $\sum_{a \in \mathcal{A}} \pi(s, a) = 1$ for every $s \in \mathcal{S}$. If the current state is s , the agent chooses a as the next action with probability $\pi(s, a)$. We write $\pi(a | s)$ for $\pi(s, a)$.

A pair of a Markov decision process and a policy probabilistically generates a sequence

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, \dots,$$

where a_i is a sample of $\pi(- | s_i)$ and s_{i+1}, r_{i+1} is a sample drawn from the distribution specified by $p(-, - | s_i, a_i)$. We assume that the interaction terminates at step n , i.e. $s_n = \star$. Then, the *return* is the sum of the rewards $r_1 + r_2 + \dots + r_n$.

The goal of reinforcement learning is to find a policy that maximize the *expected return*. Several efficient algorithms are known for this problem. In this paper, we use *Q-learning* [41] and (*first-visit on-policy*) *Monte Carlo* [42]; see a textbook [6] for the details of these algorithms. These algorithms always converge to the optimal policy under a certain assumption.

An important feature of these algorithms is that they do not require the complete description of the dynamics function. The algorithms are applicable if one can generate state-action-reward sequences $s_0, a_0, r_1, s_1, a_1, r_2, s_2, \dots$ following a given policy π . In particular, these algorithms are applicable to "non-Markov" processes, in which (r_{i+1}, s_{i+1}) depends not only on (s_i, a_i) but also on other data, although there is no theoretical guarantee. It is known that the Monte Carlo method tends to work better than Q-learning for non-Markov settings.

B. Template-Update Process as Markov Decision Process

In our setting, the environment E is an implementation of Algorithm 1 except for the call to *ChangeTempl*. The agent A is responsible for choosing the next template, namely choosing the new template from the current state of the solver. As described below, A can observe an abstraction of a concrete state of the solver.

The formal definitions of the states, the actions, and the rewards used in our setting are as follows.

a) States: We set \mathcal{S} to the concrete states of PCSat. Our RL algorithm explores only policies that are expressible as a function from *abstracted* states to the actions to make the policy-learning tractable. Concretely, an abstracted state is a tuple of the Boolean values, each of which expresses the current status of E as follows.

- b_0 (resp. b_1)
True iff. $M \leq N$ (resp. $M < N$).
- b_2 (resp. b_3)
True iff. $P \geq 2$ (resp. $P \geq 5$).
- b_4 (resp. b_5)
True iff. $Q \geq 2$ (resp. $Q \geq 5$).
- b_6
True iff. the set $\mathcal{E}(F)$ has not been augmented between the previous call to *ChangeTempl* and the current call.
- f_1, f_2, f_3
These values summarize the information on the unsat core $I.uc$: f_1 is true iff. $I.uc$ mentions M or N ; f_2 iff. $I.uc$ mentions P ; f_3 iff. $I.uc$ mentions Q .

Our design of the state space is inspired by the heuristic that is used in the current implementation of PCSat described in Section III-C. The flag b_1 can be used to alternate incrementing M and N . PCSat also uses the values of P and Q to increase P and Q gradually; to keep the state space finite in our setting, we abstract the values of P and Q using b_2, \dots, b_5 . The information expressed using flags f_1, \dots, f_3 is also used in PCSat to decide which of M, N, P, Q to be increased. One exception is the flag b_6 , which is not used in the current PCSat heuristic. We include this flag inspired by CODE2INV [20, 21], which uses the number of example instances satisfied by the current template as a part of reward. We abstract the number by the flag b_6 to keep the state space finite.

Note that, by the above abstraction of states, the agent can access only the limited information on the concrete state of the solver E . For example, the agent does not know anything about the CHC that E is now solving nor about counterexamples that E has been obtained so far. Even the current template shape is only partially informed to the agent. Therefore, the MDP observable from the agent A in our setting, (i.e., the MDP where \mathcal{S} is replaced with the abstracted states) does not satisfy the Markov property.

b) Actions: Recall that a template is specified by a tuple

$$(M, N, P, Q) \in \mathbb{N} \times \mathbb{N} \times (\mathbb{N} \cup \{\infty\}) \times (\mathbb{N} \cup \{\infty\}).$$

An action is a tuple

$$(m, n, p, q) \in \{0, 1\} \times \{0, 1\} \times \{0, 1, \infty\} \times \{0, 1, \infty\}.$$

If the current template shape is (M, N, P, Q) and the action is (m, n, p, q) , then the next template shape is $(M + m, N + n, P + p, Q + q)$ where $i + \infty = \infty + i = \infty + \infty = \infty$. This is based on the actions of the PCSat heuristics.

c) Rewards: The agent A receives reward $-T$, where T is the duration between the last invocation of A by E and the current time. Hence, the sum of the rewards is $-T_{total}$, where $-T_{total}$ is the time spent for the run. Therefore, the earlier a solution is found, the more reward is given; the smallest reward is given if Algorithm 1 timeouts.

V. EXPERIMENTS

This section presents the empirical evaluation of our approach. We implemented our idea on the top of the template-based invariant synthesis of PCSat. In the rest of this section, PCSat stands for the original version. We developed several solvers using the proposed approach, which varies in the training datasets and the learning algorithms; a solver with heuristics learned by a learning algorithm Alg using a training dataset $DataSet$ is named $Alg^{DataSet}$. PCSat and $Alg^{DataSet}$ differ in template-updating heuristic for invariant synthesis.

a) Research questions.: We aim at addressing the following research questions through the experiments.

- RQ1** Does a choice of heuristics for template update affect the performance of an invariant-synthesis tool?
- RQ2** Can a better heuristic than the hand-tuned PCSat be learned?
- RQ3** Does a learned heuristic perform well for unseen problems?
- RQ4** Does the design of state space affect the performance of learned heuristics?

b) Task and datasets.: We use the problems in Inv-Track of a competition SyGuS-Comp [10] as the benchmark of our experiments; Inv-Track is a competition for invariant-synthesis tools in SyGuS-Comp. A tool for this competition is supposed to return one of the following three answers: SAT, which indicates that the tool successfully synthesized an answer to the given problem; UNSAT, which indicates that there is no solution to the given problem; and TIMEOUT, which indicates the tool fails to solve the given problem within a specified time limit. A tool is required to give an evidence when it answers SAT; in Inv-Track, this evidence is an invariant.

For our experiments, we use two benchmark suites taken from Inv-Track: SYGUS2018 (resp. XC), which is taken from SyGuS-Comp 2018 (reps. SyGuS-Comp 2019) consisting of 127 (resp. 276) problems. There is no overlapping problems between these two benchmark suites. The problems can be found at <https://github.com/SyGuS-Comp/benchmarks>.

c) Configuration of experiments.: As learning methods, we used first-visit on-policy Monte Carlo (MC) [42] and on-policy Q-learning (QL) [41]. The design of states, actions, and rewards followed that described in Section IV, except for the experiment in Section V-C where we conducted an ablation study on the design of the state space to answer RQ4.

Each epoch, we generated an episode from each problem in a training benchmark set using the ϵ -greedy¹ action selection with $\epsilon = 0.05$. We set the time limit of every episode (i.e.,

¹Technical terms of reinforcement learning that are not explained in this paper are used in this and the next paragraphs, in order to appropriately describe the experimental setting. See a textbook [6] for a reference.

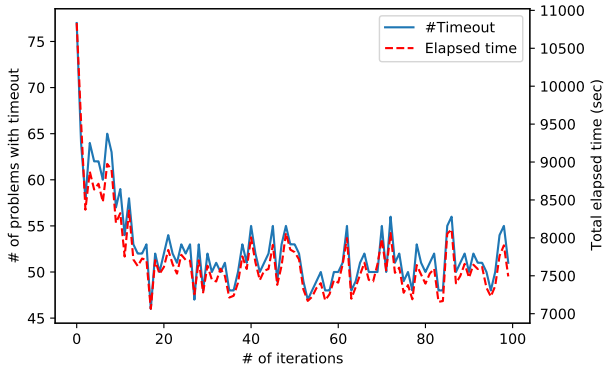


Fig. 1. The learning with Monte Carlo on XC. The blue solid line draws the number of problems to which TIMEOUT is answered and the red dashed line draws the total elapsed time (i.e., the number opposite in sign to the total reward) for each iteration.

problem solving) to 120 seconds when we learn a policy. The discount factor is set to 1.0, and the learning rate is 0.1. For each experiment, we iterated the entire training benchmark set 100 times. These hyperparameters have not been tuned carefully.

Among the learned action-value functions, we chose the best heuristic with respect to the training dataset (for each configuration); for example, MC^{18} is the solver with the best heuristic among those appearing in the learning process of the algorithm MC over the training dataset SYGUS2018. In the experiments below, the performance of each heuristic is evaluated with the greedy policy under the time limit of 600 seconds.

All the experiments are conducted on PCSat with SMT solver Z3 (version 4.8.9) on 2.8GHz Intel(R) Xeon(R) CPU, 64 GB RAM; the source code of PCSat is in the supplementary material.

We use three baselines for comparison: PCSat, LoopInvGen [40], and CVC4 [7, 8, 9]. The last two tools are the winners of SyGuS-Comp 2018 and 2019, respectively.²

A. Experiments for RQ1 and RQ2

To address RQ1 and RQ2, we compared the performance between PCSat and MC^{XC} . This experiment used the Monte Carlo method as a learning method and used the benchmark XC for both learning and evaluation.³

Figure 1 presents how the learning proceeds. It indicates that the learning converges after 20 iterations.

Table I and Figure 2 summarize the comparison result. In the table, the cell of row R and column #C denotes the number of problems for which implementation R produces answer C. The column “Total time” shows the total elapsed times spent to answer all the problems in XC. The numbers for CVC4

TABLE I
COMPARISON OF MC^{XC} AND THE BASELINES ON XC.

	#SAT	#UNSAT	#TIMEOUT	Total time (seconds)
MC^{XC}	221	17	38	25366
PCSat	166	15	95	62961
LoopInvGen	186	—	—	—
LoopInvGen*	174	9	93	57033
CVC4	215	—	—	—
CVC4*	174	18	84	54047

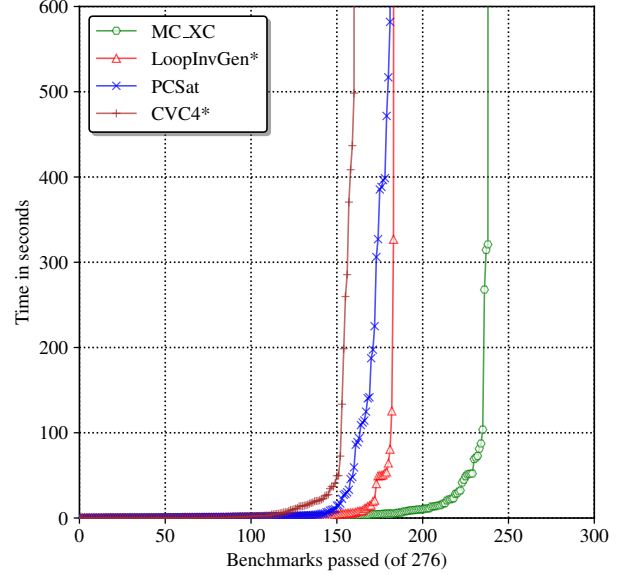


Fig. 2. Comparison of the cumulative times among MC^{XC} and the baselines (RQ1 and RQ2).

and LoopInvGen are taken from the competition report [44]; the numbers of UNSAT and TIMEOUT, as well as the total running time, are not made public in the report. Notice that the machine configuration of the competition is not the same as that of PCSat and MC^{XC} ; the competition was conducted on the platform StarExec.⁴ For a fair comparison, we remeasured the performance of LoopInvGen and CVC4 under the same configuration as that we use; the results are shown in the rows for LoopInvGen* and CVC4*.⁵

The table shows that the learned heuristics successfully enhance the performance of PCSat; the number of the solved problems (i.e., the number of problems with answer SAT or UNSAT) increases from 181 of PCSat to 238 of MC^{XC} . Therefore, a choice of heuristics can significantly affect the performance of an invariant-synthesis tool (RQ1) and a more effective heuristic than the hand-tuned one is learnable by RL (RQ2).

⁴<https://www.starexec.org/starexec/public/about.jsp>

⁵Unfortunately we have not succeeded in reproducing the results in the competition report, perhaps because of differences of the versions and configurations of the solvers. For a fair comparison, we shall use the results in the competition report if we are only interested in #SAT; if the running time is needed, we use the results of our reproductive experiments, but we emphasise that they should be regarded only as a guide.

²We also tried to use CLN2INV [43], which is a deep-learning-based invariant-synthesis tool, as a baseline. However, we excluded it from our baseline since the implementation that is made public is not fully automated. It uses a manually-given hint to solve some problems.

³We also have tried Q-learning, but the learned heuristics have not shown as good performance as those learned with the Monte Carlo method.

TABLE II
GENERALIZATION OF LEARNING ON SYGUS2018 TO XC.

	#SAT	#UNSAT	#TIMEOUT	Total time (seconds)
MC ¹⁸	190	17	69	49965
QL ¹⁸	204	18	54	35277

Figure 2 plots a series of maximum numbers of problems that each tool can answer for specified times. It shows that, while the baseline tools immediately return answers or end at timeout for most of the problems, MC^{XC} address more problems for acceptable times.

Figure 3 shows the results of one-to-one comparisons between MC^{XC} and each of the baselines. It draws the time taken by each tool for every problem in XC. The x- and y-axes of each plot represent the times in seconds on logarithmic scale for a baseline and MC^{XC}, respectively. The problems of SAT are plotted by the blue dots and those of UNSAT are plotted by the red circles. We can find that (1) there exist few problems on which the performance of MC^{XC} is worse than that of PCSat and (2) the problems at which MC^{XC} and LoopInvGen* or CVC4* work well are varying, though MC^{XC} can solve problems at which the baselines end at timeout.

B. Experiment for RQ3

Next, to address RQ3, we evaluated whether a heuristic learned on SYGUS2018 generalized well to the performance on XC. Table II and Figure 4 show the result.

We can observe that the learned heuristic in Table II outperforms PCSat in Table I. It is also noteworthy that the performance of QL¹⁸ is comparable to that of the state-of-the-art invariant-synthesis tool CVC4 (in Table I). Hence, we conclude that a heuristic that generalizes well to unseen problems can be learned via RL.

Figure 5 shows the comparison result between PCSat and QL¹⁸ with the best heuristic. Compared with Figure 3a, the plots in Figure 5 seem more scattered and the relative performance of QL¹⁸ is not as good as that of MC^{XC} in Section V-A. This would be because this section uses different datasets for learning and evaluation. Section V-A uses XC for both learning and evaluation. Therefore, the result in Figure 3a would be overfitted to XC. By contrast, since the heuristics in this section are learned on SYGUS2018, their performance on XC could be more deviated.

C. Experiment for RQ4

To address RQ4, we conducted the experiments with the variations of the state space. We created the variations by dropping state flags that do not directly encode the information of a template. b_6, f_1, f_2, f_3 are such flags: b_6 only denotes whether new example instances have been augmented; f_1, f_2, f_3 only encode the information of an unsat core.

The result is shown in Table III and Figure 6. The column “Dropped” shows the flags dropped from the state space; N/A means that no flag is dropped. Notice that the results in the

TABLE III
ABLATION STUDY ON THE STATE SPACE DESIGN.

	Dropped	#SAT	#UNSAT	#TIMEOUT	Total time
MC	N/A	190	17	69	49965
MC	b_6	214	18	44	31930
MC	f_1, f_2, f_3	216	18	42	30897
MC	b_6, f_1, f_2, f_3	184	15	77	53730
QL	N/A	204	18	54	35277
QL	b_6	173	16	87	57235
QL	f_1, f_2, f_3	186	16	74	50460
QL	b_6, f_1, f_2, f_3	190	17	69	48815

TABLE IV
COMPARISON OF THE NEWEST VERSION OF PCSAT AND ITS ENHANCEMENT WITH THE LEARNED POLICY.

	#SAT	#UNSAT	#TIMEOUT	Total time (seconds)
CVC4	215	—	—	—
NewPCSat	244	18	14	14496
NewMC ^{XC}	247	18	11	8870
NewQL ¹⁸	245	18	13	10307

first rows of MC and QL are the same as those in Table II, respectively.

For MC, the performance of dropping one of b_6 or f_1, f_2, f_3 is significantly better than that of dropping all the flags. Perhaps surprisingly, the performance is second-worst among the four configurations if all the flags are used. This result indicates the possibility that using all the flags causes overfitting to XC, while dropping all of b_6, f_1, f_2, f_3 results in underfitting. We can find that the design of state spaces affects the performance, but how to refine state spaces is left for future work.

In contrast, for QL, it is more difficult to read out a general trend on the performance in terms of the change of the state space. We suspect that a potential reason for this is that our model is not a Markov decision process. Our model cannot observe key information to identify states of PCSat, such as the information on a program being verified, the current example instances, and the queries to the underlying SMT solver Z3. Furthermore, it is known that the behavior of Z3 often depends on the queries that have been issued so far. Hence, the performance of PCSat depends on both the current and the previous hidden states. Unfortunately, Q-learning is known to fail to converge on such a model [45]. Thus, although the performance of QL is better than that of PCSat and competitive with that of CVC4 in some configurations, we consider it difficult to read out a principle on the state design from the experimental results with QL.

We therefore conclude that the state design significantly affects the performance of learned heuristics, but we need to choose a suitable learning algorithm for a model to evaluate the result appropriately.

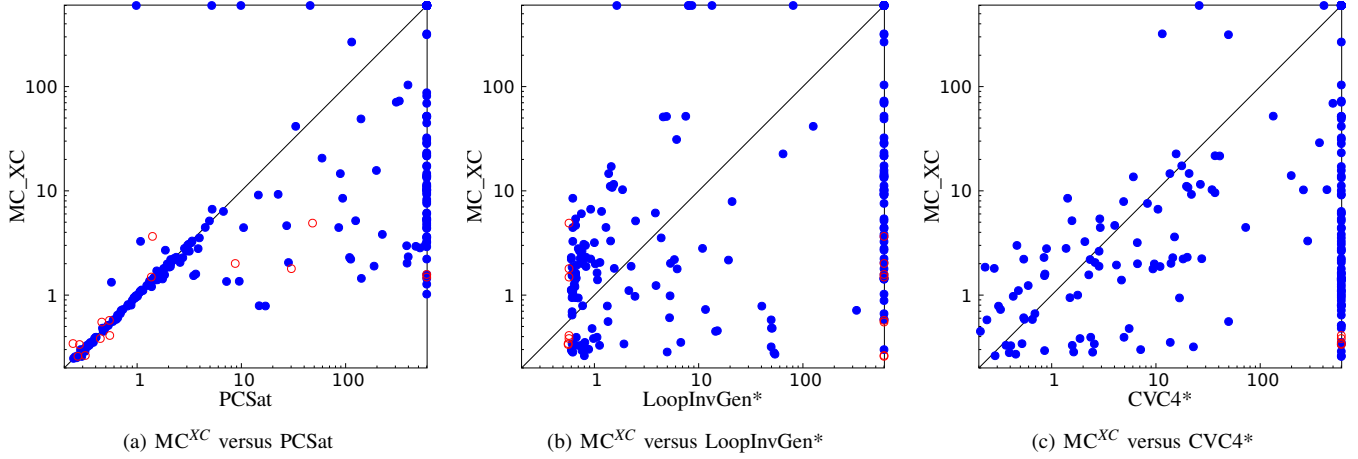


Fig. 3. Comparison of the elapsed time for each problem between MC^{XC} and each of the baselines (RQ1 and RQ2).

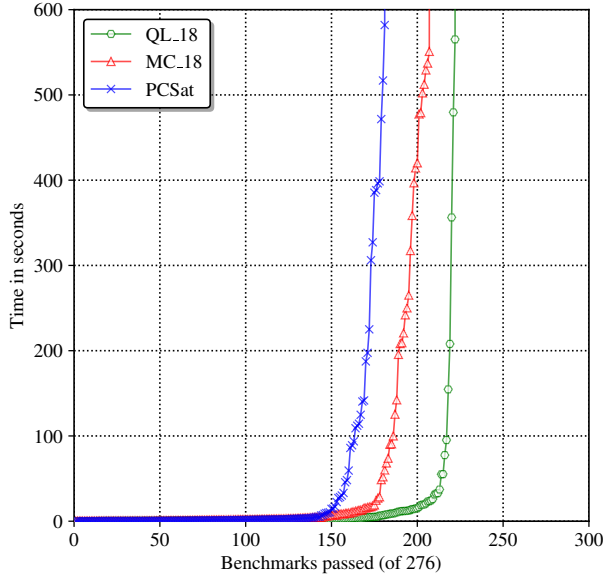


Fig. 4. Comparison of the cumulative times among MC^{18} , QL^{18} , and PCSat (RQ3).

D. Effectiveness of the Learned Heuristic for the Newest Version of PCSat

A recent update of PCSat significantly improves the performance; we call the new version of PCSat NewPCSat. A major change is an introduction of a preprocessing called *redundant argument filtering* [46] to an input CHC before passing it to the CEGIS loop. This preprocessing eliminates the redundant arguments from a CHC to make the constraint simple expecting the CEGIS phase becomes efficient. This preprocessing is indeed effective for XC; as shown in Table IV, NewPCSat significantly outperforms CVC4, the winner of SyGuS-Comp 2019 (which provides XC). It is natural to ask whether the heuristics learned on the old version of PCSat also works well for the new optimized version.

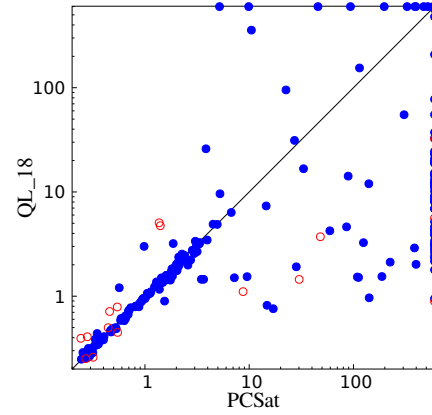


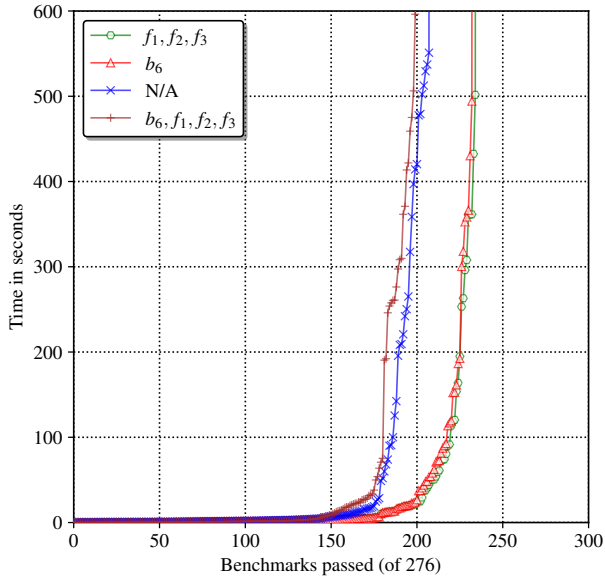
Fig. 5. Comparison of the elapsed time for each problem between QL^{18} and PCSat (RQ3).

We measured the performance of NewPCSat combined with MC^{XC} (the best heuristic learned in Section V-A) and QL^{18} (the best heuristic learned in Section V-B); we call these solvers New MC^{XC} and New QL^{18} , respectively. Note that the heuristics are learned with the old version of PCSat and evaluated with the new version.

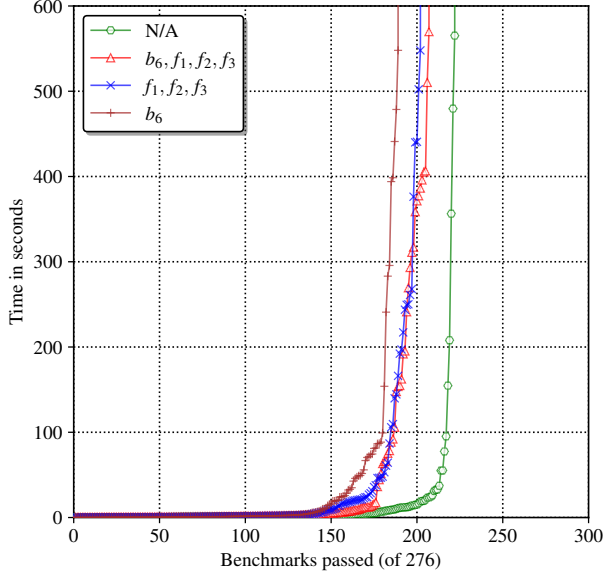
The results are summarized in Table IV and Figure 7. Table IV shows that (1) New MC^{XC} and New QL^{18} solve slightly more problems than NewPCSat and (2) New MC^{XC} and New QL^{18} solve the entire problems 38.8% and 28.9% faster than NewPCSat, respectively. Therefore, the learned heuristics work well under the new environment.

Figures 8 (a), (b), and (c) share the same trend. This means that the choice of heuristics is orthogonal to the optimization (at least in this case).

In summary, the experiment shows the robustness of learned heuristics and that the solver New QL^{18} with the heuristic learned by reinforcement learning outperforms the state-of-the-art solvers, namely CVC4 and the latest version of PCSat.



(a) MC



(b) QL

Fig. 6. Comparison of the cumulative times for MC and QL with various state abstractions (RQ4).

VI. CONCLUSION

We applied reinforcement learning to learn a heuristic for an invariant-synthesis tool PCSat, which is a solver based on the template-based CEGIS. The heuristic decides how to expand the search space for an invariant based on the states of PCSat. We demonstrated that our method indeed learns an effective heuristic by showing that PCSat combined with the learned heuristic (and the recent optimization) outperforms the state-of-the-art solver CVC4. We also showed that learned heuristics are robust in that they work well even for the new optimized version of PCSat.

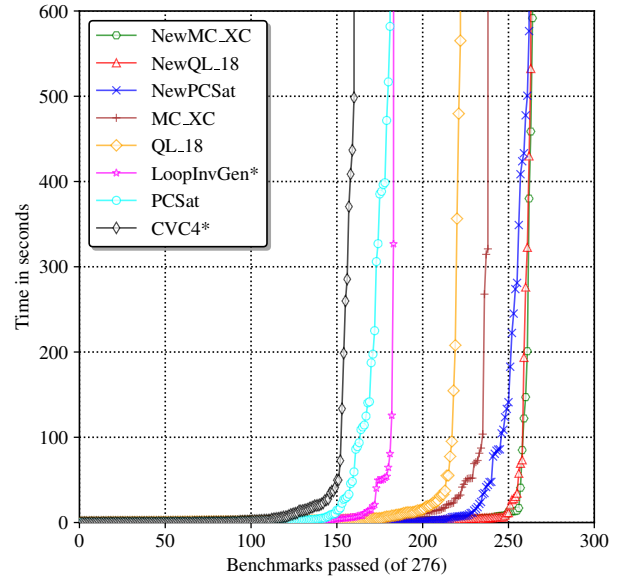


Fig. 7. Comparison of the cumulative times with the new optimized PCSat.

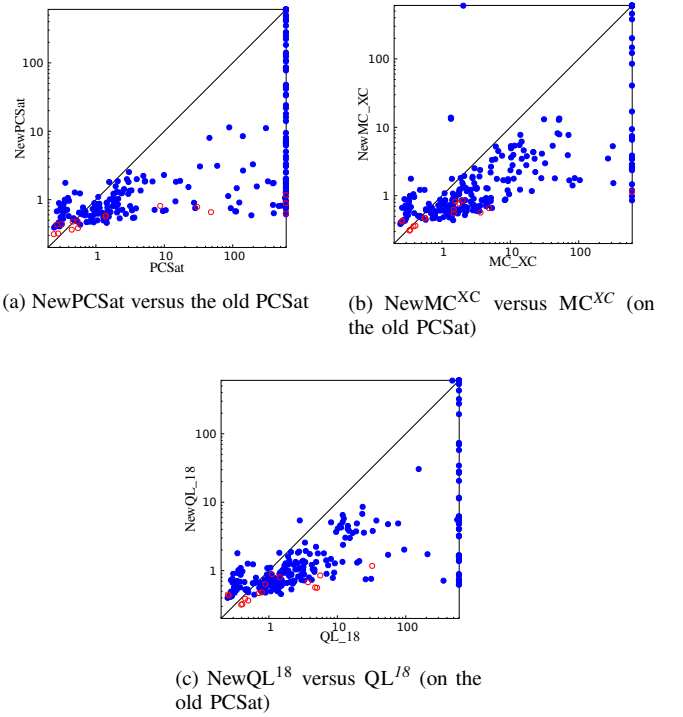


Fig. 8. Comparison of the elapsed time for each problem with the new optimized PCSat.

In the present study, we have focused on programs that have only one loop. In general, we need to deal with a program with multiple nested loops to solve the partial correctness verification problem in general. We are studying how to handle a broader class of constraints than linear CHCs to extend our methods to generic programs.

We are also planning to study how to *systematically* improve

a hand-tuned solver based on a learned heuristic, whose usefulness is signified by our experience. One direction that we are looking at is leveraging the research on interpretability of machine learning [47, 48, 49] to extract useful information from an effective heuristic and to feedback to the implementation of a hand-tuned solver.

REFERENCES

- [1] G. Winskel, *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.
- [2] Y. Satake, H. Unno, and H. Yanagi, “Probabilistic inference for predicate constraint satisfaction,” in *Proceedings of AAAI 2020*, 2020.
- [3] H. Unno, Y. Satake, T. Terauchi, and E. Koskinen, “Program verification via predicate constraint satisfiability modulo theories,” *CoRR*, vol. abs/2007.03656, 2020. [Online]. Available: <https://arxiv.org/abs/2007.03656>
- [4] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, “Combinatorial sketching for finite programs,” in *ASPLOS XII*. ACM, 2006, pp. 404–415.
- [5] L. de Moura and N. Bjørner, “Z3: An efficient SMT solver,” in *TACAS ’08*, ser. LNCS, vol. 4963. Springer, 2008, pp. 337–340.
- [6] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. The MIT Press, 2018.
- [7] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, “CVC4,” in *CAV ’11*. Springer, 2011, p. 171–177.
- [8] A. Reynolds, H. Barbosa, A. Nötzli, C. Barrett, and C. Tinelli, “cvc4sy: Smart and fast term enumeration for syntax-guided synthesis,” in *CAV ’19*. Cham: Springer, 2019, pp. 74–83.
- [9] H. Barbosa, A. Reynolds, D. Larraz, and C. Tinelli, “Extending enumerative function synthesis via SMT-driven classification,” in *FMCAD ’19*, Oct 2019, pp. 212–220.
- [10] R. Alur, D. Fisman, S. Padhi, A. Reynolds, R. Singh, and A. Udupa, “SyGuS,” accessed on 2-Feb-2021. [Online]. Available: <https://sygus.org>
- [11] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, “Paramils: An automatic algorithm configuration framework,” *J. Artif. Intell. Res.*, vol. 36, pp. 267–306, 2009.
- [12] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, “Optuna: A next-generation hyperparameter optimization framework,” in *KDD ’19*, A. Teredesai, V. Kumar, Y. Li, R. Rosales, E. Terzi, and G. Karypis, Eds. ACM, 2019, pp. 2623–2631.
- [13] P. Ezudheen, D. Neider, D. D’Souza, P. Garg, and P. Madhusudan, “Horn-ICE learning for synthesizing invariants and contracts,” *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 131:1–131:25, Oct. 2018.
- [14] P. Garg, D. Neider, P. Madhusudan, and D. Roth, “Learning invariants using decision trees and implication counterexamples,” in *POPL ’16*. ACM, 2016, pp. 499–512.
- [15] S. Krishna, C. Puhersch, and T. Wies, “Learning invariants using decision trees,” *CoRR*, vol. abs/1501.04725, 2015. [Online]. Available: <http://arxiv.org/abs/1501.04725>
- [16] H. Zhu, S. Magill, and S. Jagannathan, “A data-driven CHC solver,” in *PLDI ’18*. ACM, 2018, pp. 707–721.
- [17] A. Champion, T. Chiba, N. Kobayashi, and R. Sato, “ICE-based refinement type discovery for higher-order functional programs,” in *TACAS ’18*, ser. LNCS, vol. 10805. Springer, 2018, pp. 365–384.
- [18] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori, “A data driven approach for algebraic loop invariants,” in *ESOP ’13*, ser. LNCS, vol. 7792. Springer, 2013, pp. 574–592.
- [19] P. Garg, C. Löding, P. Madhusudan, and D. Neider, “ICE: A robust framework for learning invariants,” in *CAV ’14*. Springer, 2014, pp. 69–87.
- [20] X. Si, H. Dai, M. Raghothaman, M. Naik, and L. Song, “Learning loop invariants for program verification,” in *NeurIPS ’18*. Curran Associates, Inc., 2018, pp. 7762–7773.
- [21] X. Si, A. Naik, H. Dai, M. Naik, and L. Song, “Code2Inv: A deep learning framework for program verification,” in *CAV ’20*. Springer, 2020, pp. 151–164.
- [22] J. H. Liang, C. Oh, M. Mathew, C. Thomas, C. Li, and V. Ganesh, “Machine learning-based restart policy for CDCL SAT solvers,” in *SAT ’18*. Springer, 2018, pp. 94–110.
- [23] D. Selsam, M. Lamm, B. Bünz, P. Liang, L. de Moura, and D. L. Dill, “Learning a SAT solver from single-bit supervision,” in *ICLR ’19*. OpenReview.net, 2019.
- [24] D. Selsam and N. Bjørner, “Guiding high-performance SAT solvers with unsat-core predictions,” in *SAT ’19*, ser. LNCS, vol. 11628. Springer, 2019, pp. 336–353.
- [25] M. Balunovic, P. Bielik, and M. T. Vechev, “Learning to solve SMT formulas,” in *NeurIPS ’18*, 2018, pp. 10338–10349.
- [26] C. Kaliszyk, J. Urban, H. Michalewski, and M. Olsák, “Reinforcement learning of theorem proving,” in *NeurIPS ’18*, 2018, pp. 8836–8847.
- [27] G. Lederman, M. Rabe, S. Seshia, and E. A. Lee, “Learning heuristics for quantified boolean formulas through reinforcement learning,” in *ICLR ’20*, 2020.
- [28] S. Iyer, I. Konstantas, A. Cheung, and L. Zettlemoyer, “Summarizing source code using a neural attention model,” in *ACL ’16*. The Association for Computer Linguistics, 2016.
- [29] G. Irving, C. Szegedy, A. A. Alemi, N. Eén, F. Chollet, and J. Urban, “DeepMath - deep sequence models for premise selection,” in *NIPS ’16*, 2016, pp. 2235–2243.
- [30] C. Kaliszyk, F. Chollet, and C. Szegedy, “HolStep: A machine learning dataset for higher-order logic theorem proving,” in *ICLR ’17*. OpenReview.net, 2017.
- [31] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, “Convolutional neural networks over tree structures for programming language processing,” in *AAAI ’16*. AAAI Press, 2016, pp. 1287–1293.
- [32] S. M. Loos, G. Irving, C. Szegedy, and C. Kaliszyk, “Deep network guided proof search,” in *LPAR ’17*, ser. EPIc Series in Computing, vol. 46. EasyChair, 2017, pp. 85–105.
- [33] R. Evans, D. Saxton, D. Amos, P. Kohli, and E. Grefenstette, “Can neural networks understand logical entailment?” in *ICLR ’18*. OpenReview.net, 2018.

- [34] T. Sekiyama and K. Suenaga, “Automated proof synthesis for the minimal propositional logic with deep neural networks,” in *APLAS '18*, ser. LNCS, vol. 11275. Springer, 2018, pp. 309–328.
- [35] M. Allamanis, M. Brockschmidt, and M. Khademi, “Learning to represent programs with graphs,” in *ICLR '18*. OpenReview.net, 2018.
- [36] M. Wang, Y. Tang, J. Wang, and J. Deng, “Premise selection for theorem proving by deep graph embedding,” in *NIPS '17*, 2017, pp. 2786–2796.
- [37] M. Kusumoto, K. Yahata, and M. Sakai, “Automated theorem proving in intuitionistic propositional logic by deep reinforcement learning,” *CoRR*, vol. abs/1811.00796, 2018.
- [38] A. Paliwal, S. M. Loos, M. N. Rabe, K. Bansal, and C. Szegedy, “Graph representations for higher-order logic and theorem proving,” in *AAAI '20*. AAAI Press, 2020, pp. 2967–2974.
- [39] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, “Code2vec: Learning distributed representations of code,” *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, Jan. 2019.
- [40] S. Padhi, T. D. Millstein, A. V. Nori, and R. Sharma, “Overfitting in synthesis: Theory and practice,” in *CAV '19*, ser. LNCS, vol. 11561. Springer, 2019, pp. 315–334.
- [41] C. J. C. H. Watkins, “Learning from delayed rewards,” Ph.D. dissertation, University of Cambridge, 1989.
- [42] S. P. Singh and R. S. Sutton, “Reinforcement learning with replacing eligibility traces,” *Mach. Learn.*, vol. 22, no. 1-3, pp. 123–158, 1996.
- [43] G. Ryan, J. Wong, J. Yao, R. Gu, and S. Jana, “CLN2INV: learning loop invariants with continuous logic networks,” in *ICLR '20*. OpenReview.net, 2020. [Online]. Available: <https://openreview.net/forum?id=HJIfuTEtvB>
- [44] R. Alur, D. Fisman, S. Padhi, A. Reynolds, R. Singh, and A. Udupa, “The 6th competition on syntax-guided synthesis,” accessed on 2-Feb-2021. [Online]. Available: <https://sygus.org/comp/2019/results-slides.pdf>
- [45] S. J. Majeed and M. Hutter, “On q-learning convergence for non-markov decision processes,” in *IJCAI '18*. ijcai.org, 2018, pp. 2546–2552.
- [46] M. Leuschel and M. H. Sørensen, “Redundant argument filtering of logic programs,” in *LOPSTR '97*. Springer, 1997, pp. 83–103.
- [47] M. T. Ribeiro, S. Singh, and C. Guestrin, “‘why should I trust you?’: Explaining the predictions of any classifier,” in *KDD '16*, 2016.
- [48] V. Petsiuk, A. Das, and K. Saenko, “RISE: randomized input sampling for explanation of black-box models,” in *BMVC 2018*, 2018. [Online]. Available: <http://bmvc2018.org/contents/papers/1064.pdf>
- [49] S. M. Lundberg and S.-I. Lee, “A unified approach to interpreting model predictions,” in *NIPS '17*, 2017.