Fast Numerical Program Analysis with Reinforcement Learning

March, 2021

CAV 2018

Challenge

- A key challenge in static analysis is coming up with effective and general approaches that can decide where and how to lose precision during analysis for best tradeoff between performance and precision.

Their Work

- They offer a new approach for dynamically losing precision based on reinforcement learning (RL).

- The key idea is to learn a policy that determines when and how the analyzer should lose the least precision at an abstract state to achieve best performance gains.

Basic Idea

- Imagine that a static analyzer has at each program state two available abstract transformers: the precise but slow T_p and the fast but less precise T_f. Ideally, the analyzer would decide adaptively at each step on the best choice that maximizes speed while producing a final result of sufficient precision.

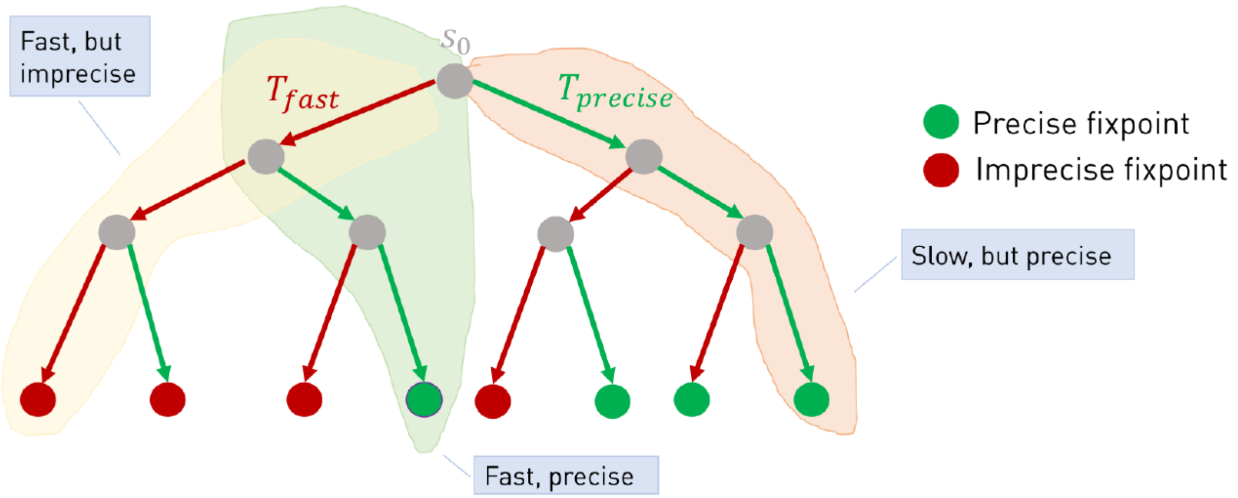


Figure 4.1: Policies for balancing precision and speed in static analysis.

Basic Idea

- They use RL to discover such a policy automatically.

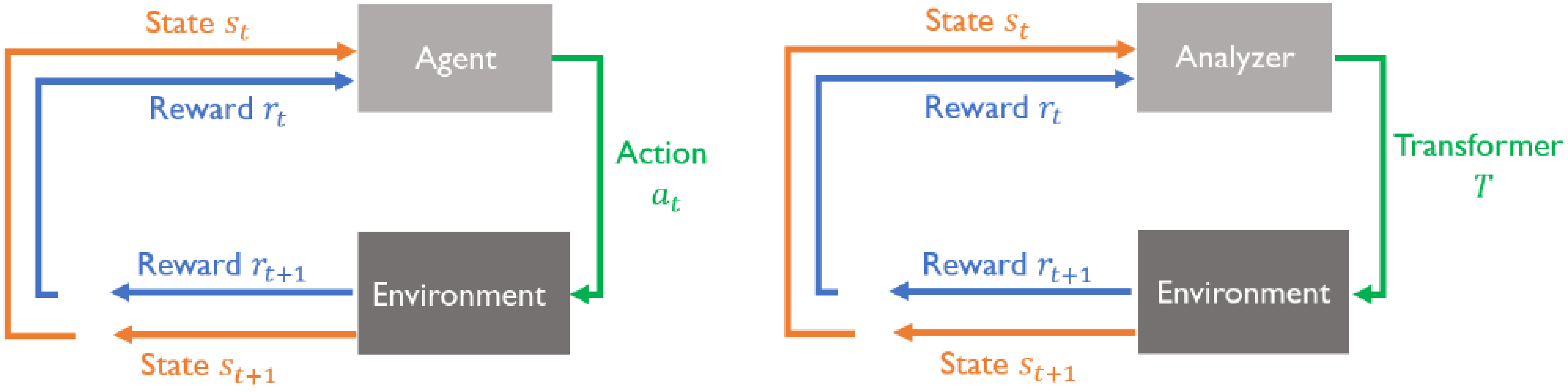


Figure 4.2: Reinforcement learning for static analysis.

Polyhedra Analysis

- Let $\chi = \{x_1, x_2, \ldots, x_n\}$ be the set of n program variables where each variable $x_i \in Q$ take a rational value.

- Constraints and generator representation:
  - An abstract element $P \subseteq Q^n$ in the Polyhedra domain is a conjunction of linear constrains $\Sigma_{i=1}^{n} a_i x_i \leq c$ between the program variables where $a_i \in Z, c \in Q$.

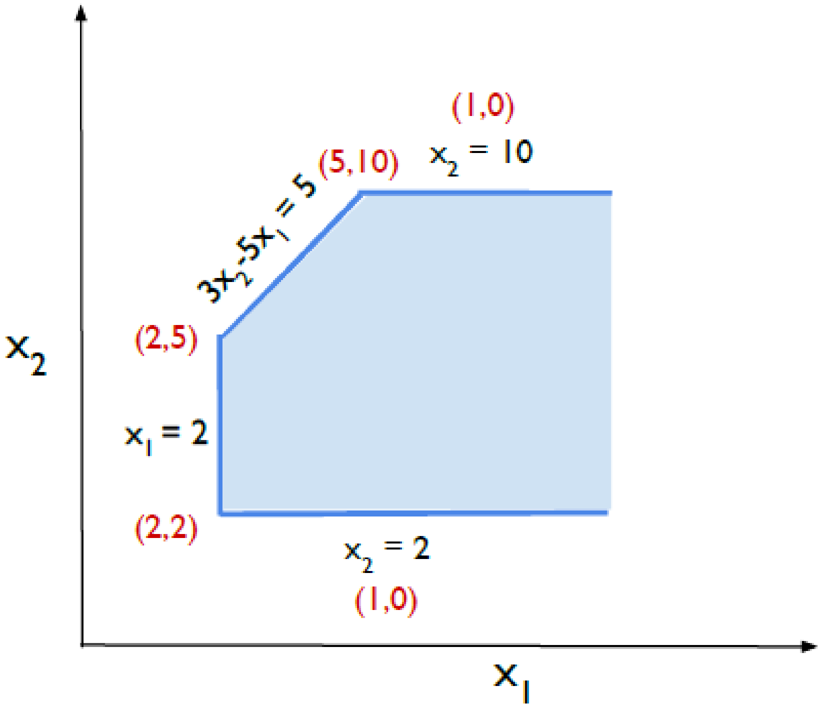
Polyhedra Domain

- Polyhedra domain is commonly used in static analysis to derive invariants that hold for all executions of the program starting from a given initial state.
  - Prove safety properties in programs like the absence of buffer overflow, division by zero and others.

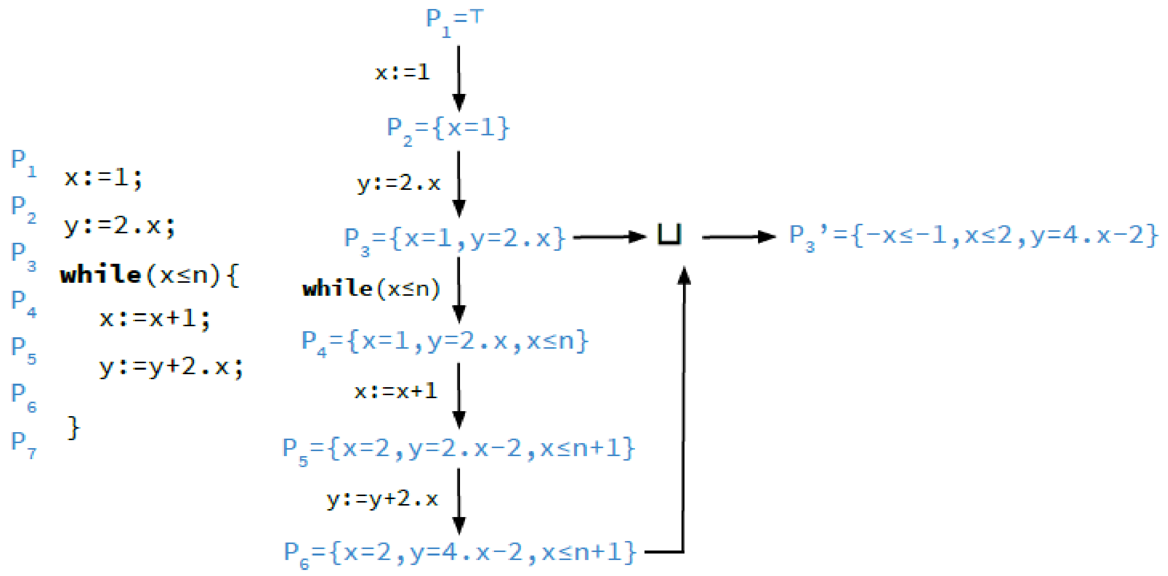
- Transformers:
  - Model the effect of various program statements such as assignments and conditionals as well as control flow such as loops and branches on the program states approximated by polyhedral.
  - Inclusion test: this transformer tests if $P \sqsubseteq Q$ for the given polyhedral P and Q.
  - Equality test;
  - Join: this transformer computes $P \sqcup Q$, i.e., the convex hull of P and Q.
  - Meet;
  - Widening

Polyhedra Analysis

- Example:
- Constraints representation:
  - $C_P = \{-x_1 \leq -2, -x_2 \leq -2, x_2 \leq 10, 3x_2 - 5x_1 \leq 5\}.$
- Generator representation:
  - $G_P = \{vertices,\ rays,\ lines\} = \{\{(2,2),(2,5),(5,10)\}, \{(1,0),(1,0)\},\ \emptyset\}$



Polyhedra Domain Analysis: Example



- The analysis proceeds iteratively by selecting the polyhedron at a given line, then applying the transformer for the statement at that program point on that polyhedron and producing a new polyhedron.
- The analysis terminates when a fixpoint is reached.
- At the fixpoint, the polyhedron P_l represents invariants that hold for all executions of the program before executing the statement at line l.

Polyhedra Analysis

- The main bottleneck for the Polyhedra analysis is the join transformer ($\sqcup$).

- Polyhedra domain analysis was sped up by orders of magnitude, without approximation, using the idea of online decomposition.

Online Decomposition

- The set of variables χ in a given polyhedron P can be partitioned as $\pi_P = \{\chi_1, \ldots \chi_r\}$ into blocks χ_t. P can be decomposed into a set of smaller Polyhedra $P(\chi_t)$ called factors.

- Example:
- Consider the set $\chi = \{x_1, \ldots x_6\}$ and the polyhedron $P = \{2x_1 - 3x_2 + x_3 + x_4 \leq 0, x_5 = 0\}$.
- Here $\pi_P = \{\{x_1, x_2, x_3, x_4\}, \{x_5\}, \{x_6\}\}$ is a possible partition of χ with factors $P(\chi_1) = \{2x_1 - 3x_2 + x_3 + x_4 \leq 0\}$, $P(\chi_2) = \{x_5 = 0\}$, $P(\chi_3) = \emptyset$.

Block Splitting

- The optimal partition for an element P is denoted with π_P.
  - It may be too expensive to compute the optimal partition.
  - The online decomposition in often computes a cheaply computable *permissible* partition $\bar{\pi}_Z \sqsupseteq \pi_Z$.

- The cost of a decomposed abstract transformer applied on P depends on the sizes of the blocks in the permissible partition $\bar{\pi}_P$, and more specifically, on the size of the largest such block.

- It is desirable to bound this size by a $threshold \in N$.

Block Splitting

- First identifying all blocks $\chi_t \in \bar{\pi}_P$ with $|\chi_t| > threshold$ that the transformer requires and then removing constraints from $P(\chi_t)$ until it decomposes into blocks of sizes < *threshold*.

Example 4.2.1. Consider the following polyhedron and *threshold* $= 4$

$$\begin{aligned} \mathcal{X}_t &= \{x_1, x_2, x_3, x_4, x_5, x_6\}, \\ P(\mathcal{X}_t) &= \{x_1 - x_2 + x_3 \leqslant 0, x_2 + x_3 + x_4 \leqslant 0, x_2 + x_3 \leqslant 0, \\ &\qquad x_3 + x_4 \leqslant 0, x_4 - x_5 \leqslant 0, x_4 - x_6 \leqslant 0\}. \end{aligned}$$

We can remove $\mathcal{M} = \{x_4 - x_5 \leqslant 0, x_4 - x_6 \leqslant 0\}$ from $P(\mathcal{X}_t)$ to obtain the constraint set $\{x_1 - x_2 + x_3 \leqslant 0, x_2 + x_3 + x_4 \leqslant 0, x_2 + x_3 \leqslant 0, x_3 + x_4 \leqslant 0\}$ with partition $\{\{x_1, x_2, x_3, x_4\}, \{x_5\}, \{x_6\}\}$, which obeys the threshold.

Block Splitting

- First identifying all blocks $\chi_t \in \bar{\pi}_P$ with $|\chi_t| > threshold$ that the transformer requires and then removing constraints from $P(\chi_t)$ until it decomposes into blocks of sizes < *threshold*.

Example 4.2.1. Consider the following polyhedron and *threshold* $= 4$

$$
\begin{aligned}
\mathcal{X}_t &= \{x_1, x_2, x_3, x_4, x_5, x_6\},\\
P(\mathcal{X}_t) &= \{x_1 - x_2 + x_3 \leqslant 0, x_2 + x_3 + x_4 \leqslant 0, x_2 + x_3 \leqslant 0,\\
&\quad x_3 + x_4 \leqslant 0, x_4 - x_5 \leqslant 0, x_4 - x_6 \leqslant 0\}.
\end{aligned}
$$

We could also remove $\mathcal{M}' = \{x_2 + x_3 + x_4 \leqslant 0, x_3 + x_4 \leqslant 0\}$ from $P(\mathcal{X}_t)$ to get the constraint set $\{x_1 - x_2 + x_3 \leqslant 0, x_2 + x_3 \leqslant 0, x_4 - x_5 \leqslant 0, x_4 - x_6 \leqslant 0\}$ with partition $\{\{x_1, x_2, x_3\}, \{x_4, x_5, x_6\}\}$, which also obeys the threshold.

Block Splitting

Example 4.2.1. Consider the following polyhedron and *threshold* $= 4$

$$\mathcal{X}_t = \{x_1, x_2, x_3, x_4, x_5, x_6\},$$
$$P(\mathcal{X}_t) = \{x_1 - x_2 + x_3 \leqslant 0, x_2 + x_3 + x_4 \leqslant 0, x_2 + x_3 \leqslant 0, x_3 + x_4 \leqslant 0, x_4 - x_5 \leqslant 0, x_4 - x_6 \leqslant 0\}.$$

We can remove $\mathcal{M} = \{x_4 - x_5 \leqslant 0, x_4 - x_6 \leqslant 0\}$ from $P(\mathcal{X}_t)$ to obtain the constraint set $\{x_1 - x_2 + x_3 \leqslant 0, x_2 + x_3 + x_4 \leqslant 0, x_2 + x_3 \leqslant 0, x_3 + x_4 \leqslant 0\}$ with partition $\{\{x_1, x_2, x_3, x_4\}, \{x_5\}, \{x_6\}\}$, which obeys the threshold.

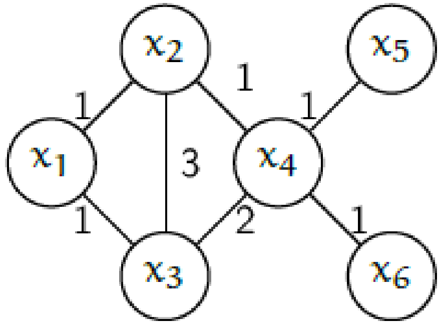


Figure 4.3: Graph G for $P(\mathcal{X}_t)$ in Example 4.2.1

Merging of Blocks

- The basic objective when approximating is to ensure that the maximal block size remains below a chosen threshold.
- Besides splitting to ensure this, there can also be a benefit of merging small blocks. The merging itself does not change precision, but the resulting transformer may be more precise when working on larger blocks.

1. *No merge:* None of the blocks are merged.
2. *Merge smallest first:* We start merging the smallest blocks as long as the size stays below the threshold. These blocks are then removed and the procedure is repeated on the remaining set.
3. *Merge large with small:* We start to merge the largest block with the smallest blocks as long as the size stays below the threshold. These blocks are then removed and the procedure is repeated on the remaining set.

Merging of Blocks

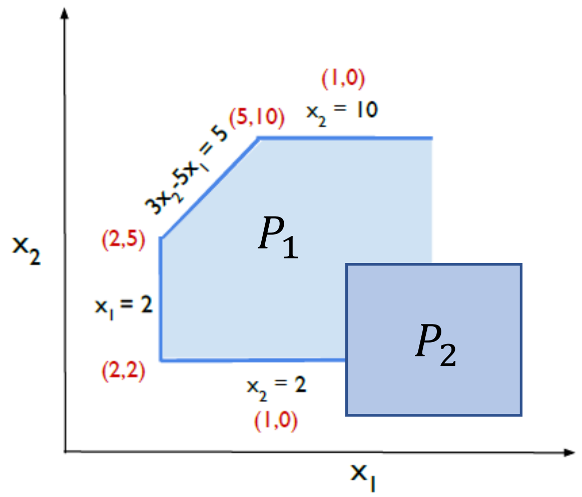
- We can apply merging to obtain larger blocks $\chi_m \leq threshold$ to increase the precision of subsequent join.

Example 4.2.4. Consider *threshold* $= 5$ and $\overline{\pi}_P$ with block sizes $\{1, 1, 2, 2, 2, 2, 3, 5, 7, 10\}$. Merging smallest first yields blocks $1 + 1 + 2$, $2 + 2$, $2 + 3$ leaving the rest unchanged. The resulting sizes are $\{4, 4, 5, 5, 7, 10\}$. Merging large with small leaves $10, 7, 5$ unchanged and merges $3 + 1 + 1$, $2 + 2$, and $2 + 2$. The resulting sizes are also $\{4, 4, 5, 5, 7, 10\}$ but the associated factors are different (since different blocks are merged), which will yield different results in following transformations.

Approximation for Polyhedra Join

- Let $\bar{\pi}_{common} = \bar{\pi}_{P_1} \sqcup \bar{\pi}_{P_2}$ be a common permissible partition for the inputs P_1, P_2 of the join transformer.
- Then a permissible partition for the output is obtained by keeping all blocks $\chi_t \in \bar{\pi}_{common}$ for which $P_1(\chi_t) = P_2(\chi_t)$ int the output partition, and fusing all remaining blocks into one.
- Formally, $\bar{\pi}_O = \{N\} \cup U$.

$$\mathcal{N} = \bigcup\{\mathcal{X}_k \in \overline{\pi}_{\text{common}} : P_1(\mathcal{X}_k) \neq P_2(\mathcal{X}_k)\}, \quad \mathcal{U} = \{\mathcal{X}_k \in \overline{\pi}_{\text{common}} : P_1(\mathcal{X}_k) = P_2(\mathcal{X}_k)\}.$$



Approximating the Polyhedra Join

Algorithm 4.3 Approximation algorithm for Polyhedra join

function `approximate_join`$((P_1, \overline{\pi}_{P_1}), (P_2, \overline{\pi}_{P_2}), \textit{threshold})$
    Input:
        $(P_1, \overline{\pi}_{P_1}), (P_2, \overline{\pi}_{P_2}) \leftarrow$ decomposed inputs to the join
        threshold $\leftarrow$ Upper bound on size of $\mathcal{N}$
    $O := \bigcup\{P_1(\mathcal{X}_k) : \mathcal{X}_k \in \mathcal{U}\}$
    $\overline{\pi}_O := \mathcal{U}$ ▷ *initialize output partition*
    $\mathcal{B} := \{\mathcal{X}_k \in \overline{\pi}_{P_1} \sqcup \overline{\pi}_{P_2} : \mathcal{X}_k \subseteq \mathcal{N}\}$
    $\mathcal{B}_t := \{\mathcal{X}_t \in \mathcal{B} : |\mathcal{X}_t| > \text{threshold}\}$

▷ *join factors for blocks in $\mathcal{B}_t$ and split the outputs*
    for $\mathcal{X}_t \in \mathcal{B}_t$ *do*
        $P' := P_1(\mathcal{X}_t) \sqcup P_2(\mathcal{X}_t)$
        $(\mathcal{C}, \overline{\pi}) :=$ `block_split`$(\mathcal{X}_t, \mathcal{C}_{P'}, \textit{threshold})$
        for $\mathcal{X}_{t'} \in \overline{\pi}$ *do*
            $\mathcal{G}(\mathcal{X}_{t'}) :=$ `conversion`$(\mathcal{C}(\mathcal{X}_{t'}))$
            $O := O \cup (\mathcal{C}(\mathcal{X}_{t'}), \mathcal{G}(\mathcal{X}_{t'}))$
        end for
        $\overline{\pi}_O := \overline{\pi}_O \cup \overline{\pi}$
    end for

▷ *merge blocks $\in \mathcal{B} \setminus \mathcal{B}_t$ via a merge algorithm and apply join*
    $\text{m\_algo} :=$ `choose_merge_algorithm`$(\mathcal{B} \setminus \mathcal{B}_t)$
    $\mathcal{B}_m :=$ `merge`$(\mathcal{B} \setminus \mathcal{B}_t, \text{m\_algo})$
    for $\mathcal{X}_m \in \mathcal{B}_m$ *do*
        $O := O \cup (P_1(\mathcal{X}_m) \sqcup P_2(\mathcal{X}_m))$
        $\overline{\pi}_O := \overline{\pi}_O \cup \{\mathcal{X}_m\}$
    end for
    return $(O, \overline{\pi}_O)$
end function

Need for RL

- The above algorithm shows how to approximate the join transformer: different choices of threshold, splitting and merge strategies yield a range of transformers with different performance and precision depending on the inputs.

- Determining the suitability of a given choice on an input is highly non-trivial and thus we use RL to learn it.

Reinforcement Learning for Polyhedra Analysis

- The instantiation:
  - Extracting the RL state s from the abstract program state numerically using a set of features.
  - Defining actions a as the choices among the threshold, merge and split methods.
  - Defining a reward function r favoring both high precision and fast execution,
  - Defining the feature functions $\phi(s, a)$ to enable Q-learning.

Reinforcement Learning for Polyhedra Analysis

- States.
- Considering 9 features for defining a state s for RL.
- The first 7 features capture the asymptotic complexity of the join on the input polyhedra P_1 and P_2.
  - The number of blocks, the distribution of their sizes, and the number of generators.
- The precision of the inputs is captured by considering the number of variables $x_i \in \chi$ with finite upper and lower bound.
- They use bucketing to reduce the state space size by clustering states with similar precision and expected join cost.
- The RL state s is then a 9-tuple consisting of the indices of buckets where each index indicates the bucket that $\psi_i's$ return value falls into.

Reinforcement Learning for Polyhedra Analysis

- States.

Table 2. Features for describing RL state s ($m \in \{1, 2\}, 0 \leq j \leq 8, 0 \leq h \leq 3$).

Feature ψ_i	Extraction complexity	Typical range	n_i	Buckets for feature ψ_i
$\lvert\mathcal{B}\rvert$	$O(1)$	1–10	10	$\{[j+1, j+1]\} \cup \{[10, \infty)\}$
$\texttt{min}(\lvert\mathcal{X}_k\rvert : \mathcal{X}_k \in \mathcal{B})$	$O(\lvert\mathcal{B}\rvert)$	1–100	10	$\{[10 \cdot j + 1, 10 \cdot (j+1)]\} \cup \{[91, \infty)\}$
$\texttt{max}(\lvert\mathcal{X}_k\rvert : \mathcal{X}_k \in \mathcal{B})$	$O(\lvert\mathcal{B}\rvert)$	1–100	10	$\{[10 \cdot j + 1, 10 \cdot (j+1)]\} \cup \{[91, \infty)\}$
$\texttt{avg}(\lvert\mathcal{X}_k\rvert : \mathcal{X}_k \in \mathcal{B})$	$O(\lvert\mathcal{B}\rvert)$	1–100	10	$\{[10 \cdot j + 1, 10 \cdot (j+1)]\} \cup \{[91, \infty)\}$
$\texttt{min}(\lvert\bigcup \mathcal{G}_{P_m(\mathcal{X}_k)}\rvert : \mathcal{X}_k \in \mathcal{B})$	$O(\lvert\mathcal{B}\rvert)$	1–1000	10	$\{[100 \cdot j + 1, 100 \cdot (j+1)]\} \cup \{[901, \infty)\}$
$\texttt{max}(\lvert\bigcup \mathcal{G}_{P_m(\mathcal{X}_k)}\rvert : \mathcal{X}_k \in \mathcal{B})$	$O(\lvert\mathcal{B}\rvert)$	1–1000	10	$\{[100 \cdot j + 1, 100 \cdot (j+1)]\} \cup \{[901, \infty)\}$
$\texttt{avg}(\lvert\bigcup \mathcal{G}_{P_m(\mathcal{X}_k)}\rvert : \mathcal{X}_k \in \mathcal{B})$	$O(\lvert\mathcal{B}\rvert)$	1–1000	10	$\{[100 \cdot j + 1, 100 \cdot (j+1)]\} \cup \{[901, \infty)\}$
$\lvert\{x_i \in \mathcal{X} : x_i \in [l_m, u_m] \text{ in } P_m\}\rvert$	$O(ng)$	1–25	5	$\{[5 \cdot h + 1, 5 \cdot (h+1)]\} \cup \{[21, \infty)\}$
$\lvert\{x_i \in \mathcal{X} : x_i \in [l_m, \infty) \text{ in } P_m\}\rvert + \lvert\{x_i \in \mathcal{X} : x_i \in (-\infty, u_m] \text{ in } P_m\}\rvert$	$O(ng)$	1–25	5	$\{[5 \cdot h + 1, 5 \cdot (h+1)]\} \cup \{[21, \infty)\}$

Reinforcement Learning for Polyhedra Analysis

- Actions.
- An action a is a 3-tuple $(th, r_{algo},\ m_{algo})$:
- $th \in \{1,2,3,4\}$ depending on $threshold \in [5,9], [10,14], [15,19], or\ [20, \infty)$.
- $r_{algo} \in \{1,2,3\}$: the choice of a constraint removal, i.e., splitting method.
- $m_{algo} \in \{1,2,3\}$: the choice of merge algorithm.

Reinforcement Learning for Polyhedra Analysis

- Reward.
- After applying the approximated join transformer according to action a_t in state s_t, they compute the precision of the output polyhedron $P_1 \sqcup P_2$.
- The reward is defined by: $r(s_t, a_t, s_{t+1}) = 3n_s + 2n_b + n_{hb} - \log_{10}(cyc)$.

  - n_s: number of variables x_i with singleton interval, i.e., $x_i \in [l, u], l = u$.
  - n_b: number of variables x_i with finite upper and lower bounds, i.e., $x_i \in [l, u], l \neq u$.
  - n_{hb}: number of variables x_i with either finite upper or finite lower bounds, i.e., $x_i \in (-\infty, u]$ or $x_i \in [l, \infty)$.

Reinforcement Learning for Polyhedra Analysis

- Q-function.
- Define binary feature functions ϕ_{ijk} for each $(state, action)$ pair:

$$\phi_{ijk}(s, a) = 1 \iff s(i) = j \text{ and } a = a_k$$

- The Q-function is a linear combination of state action features ϕ_{ijk}:

$$Q(s, a) = \sum_{i=1}^{9} \sum_{j=1}^{n_i} \sum_{k=1}^{36} \theta_{ijk} \cdot \phi_{ijk}(s, a).$$

Reinforcement Learning for Polyhedra Analysis

- Q-learning
- Q-learning is performed with input parameters instantiated as explained above and summarized in Table 3.
- Each episode consists of a run of Polyhedra analysis on a benchmark in D. They run the analysis multiple times on each program in D and update the Q-function after each join by calling Q-LEARN.

Table 3. Instantiation of Q-learning to Polyhedra static analysis.

RL concept	Polyhedra Analysis Instantiation
Agent	Polyhedra analysis
State $s \in \mathcal{S}$	As described in Table 2
Action $a \in \mathcal{A}$	Tuple $(th,\ r\_algo,\ m\_algo)$
Reward function r	Shown in (3)
Feature ϕ	Defined in (4)
Q-function	Q-function from (5)

Reinforcement Learning for Polyhedra Analysis

- Q-learning
- Q-learning is performed with input parameters instantiated as explained above and summarized in Table 3.
- Each episode consists of a run of Polyhedra analysis on a benchmark in D. They run the analysis multiple times on each program in D and update the Q-function after each join by calling Q-LEARN.

Algorithm 1 Q-learning algorithm

function Q-LEARN($\mathcal{S}, \mathcal{A}, r, \gamma, \alpha, \phi$)
    Input:
        $\mathcal{S} \leftarrow$ set of states, $\mathcal{A} \leftarrow$ set of actions, $r \leftarrow$ reward function
        $\gamma \leftarrow$ discount factor, $\alpha \leftarrow$ learning rate
        $\phi \leftarrow$ set of feature functions over $\mathcal{S}$ and $\mathcal{A}$
    Output: parameters θ
    $\theta =$ Initialize arbitrarily (which also initializes Q)
    for each episode **do**
        Start with an initial state $s_0 \in \mathcal{S}$
        for $t = 0, 1, 2, \ldots, \mathit{length}(\mathit{episode})$ **do**
            Take action a_t, observe next state s_{t+1} and $r(s_t, a_t, s_{t+1})$
            $\theta := \theta + \alpha \cdot (r(s_t, a_t, s_{t+1}) + \gamma \cdot \mathtt{max}_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \cdot \phi(s_t, a_t)$
    return θ

Reinforcement Learning for Polyhedra Analysis

- Obtaining the learned policy.
- After learning over the dataset D, the learned approximating join transformer in state s_t chooses an action according $p^*(s) = \mathrm{argmax}_{\mathrm{a} \in A} \mathrm{Q}(\mathrm{s}, \mathrm{a})$ by selecting the maximal value over all actions.

Experimental Evaluation

- Poly-RL.
- Compare the performance and precision of Poly-RL against the state-of-the-art ELINA, which uses online decomposition for Polyhedra analysis.
- Benchmarks: they chose the Linux Device Drivers category in SVCOMP, known to be challenging for Polyhedra analysis as to prove properties in these programs one requires Polyhedra invariants.

Experimental Evaluation

Table 4. Timings (seconds) and precision of approximations (%) w.r.t. ELINA.

Benchmark	#Program Points	ELINA time	Poly-RL time	Poly-RL precision	Poly-Fixed time	Poly-Fixed precision	Poly-Init time	Poly-Init precision
wireless_airo	2372	877	6.6	100	6.7	100	5.2	74
net_ppp	680	2220	9.1	87	TO	34	7.7	55
mfd_sm501	369	1596	3.1	97	1421	97	2	64
ideapad_laptop	461	172	2.9	100	157	100	MO	41
pata_legacy	262	41	2.8	41	2.5	41	MO	27
usb_ohci	1520	22	2.9	100	34	100	MO	50
usb_gadget	1843	66	37	60	35	60	TO	40
wireless_b43	3226	19	13	66	TO	28	83	34
lustre_llite	211	5.7	4.9	98	5.4	98	6.1	54
usb_cx231xx	4752	7.3	3.9	≈100	3.7	≈100	3.9	94
netfilter_ipvs	5238	20	17	≈100	9.8	≈100	11	94

Conclusion

- Given a training dataset of programs, they first learn a policy over analysis runs of these programs.
- Then they use the resulting policy during analysis of new unseen programs.
- The experimental results on a set of realistic programs (e.g., Linux device drivers) show that their RL-based Polyhedra analysis achieves substantial speed-up over a heavily optimized state-of-the-art Polyhedra library.