PySE: Automatic Worst-Case Test Generation by Reinforcement Learning

Jinkyu Koo, **Charitha Saumya**, Milind Kulkarni, and Saurabh Bagchi
Electrical and Computer Engineering

Purdue University

{kooj, cgusthin, milind, sbagchi}@purdue.edu

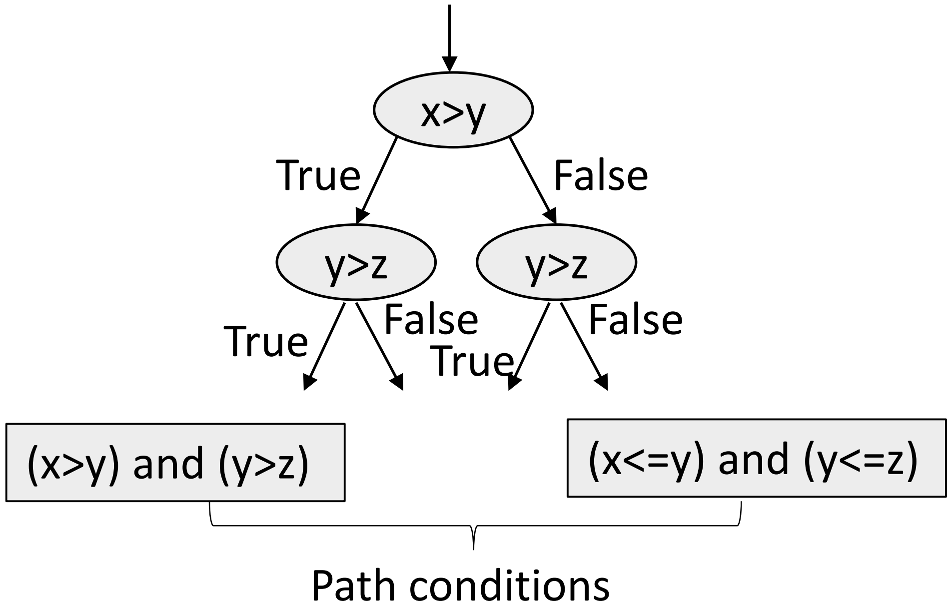
Stress testing

- Stress testing
  - Testing the software beyond its normal operational capacity, and investigates the behavior of a program when subjected to heavy loads.
- The goal of such tests
  - To identify performance bottlenecks
  - To identify algorithmic complexity attacks
  - To identify scale-dependent bugs
- The key challenge
  - How to find the input that can lead to the worst-case complexity.

Symbolic execution

- Runs a program using symbolic variables as inputs, instead of concrete values.
- Can explore all the possible execution paths, including the ones of worst-case complexity.
- On each path that is executed, symbolic execution collects a set of symbolic conditions, called a path condition.
- Then, it invokes a constraint solver, such as OpenSMT [7] or Z3 that generates concrete test input values.
- Path explosion: the number of paths to search increase exponentially with the size of the input.

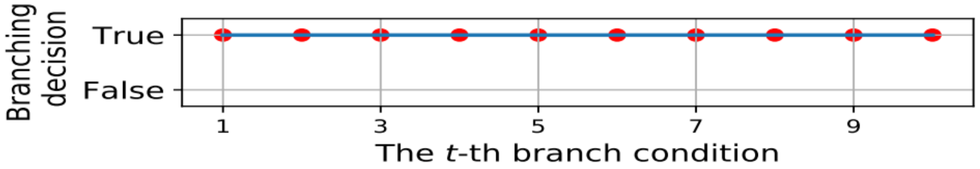
```
def function(x, y, z):
    ....
    if x>y:
        ....
    if y>z:
        ....
    ....
```



WISE-like algorithms

- WISE[1] and SPF-WCA[2]
  - Learn a branching policy that results in a path of the worst-case complexity for small input sizes by using exhaustive search, and
  - Then apply the learned branching policy to perform a guided search for a large input size.

The worst-case branching policy

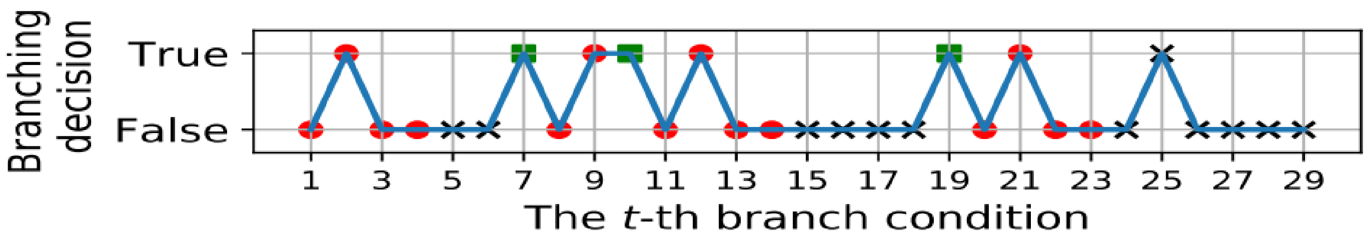


Insertion sort: always True

[1] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. Wise: Automated test generation for worst-case complexity. ICSE ’09
[2] Kasper Luckow, Rody Kersten, and Corina Pasareanu. Symbolic complexity analysis using context-preserving histories. ICST’17

Limitations of WISE-like algorithms

- Assumes a continuous program behavior across scales
  - Some conditional blocks are activated only when the input size is larger than a certain threshold.
- Irregular branching policy

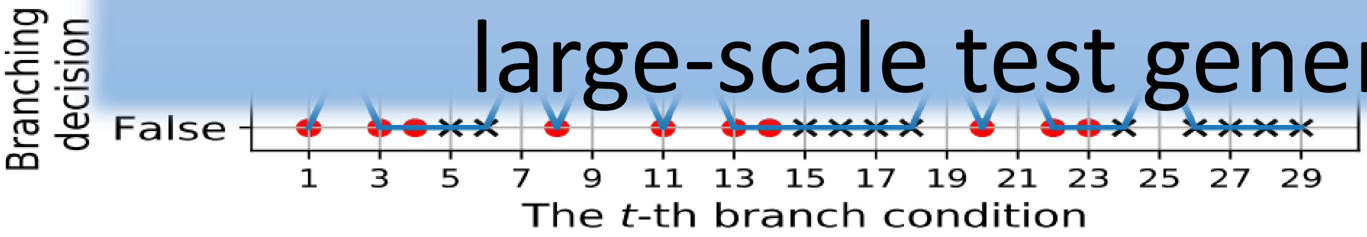


Dijkstra implemented with min-priority queue: no simple way to describe the worst-case branching policy

Limitations of WISE-like algorithms

- Assumes a continuous program behavior across scales
  - Some conditional blocks are activated only when the input size is larger than a certain threshold.
- Irre

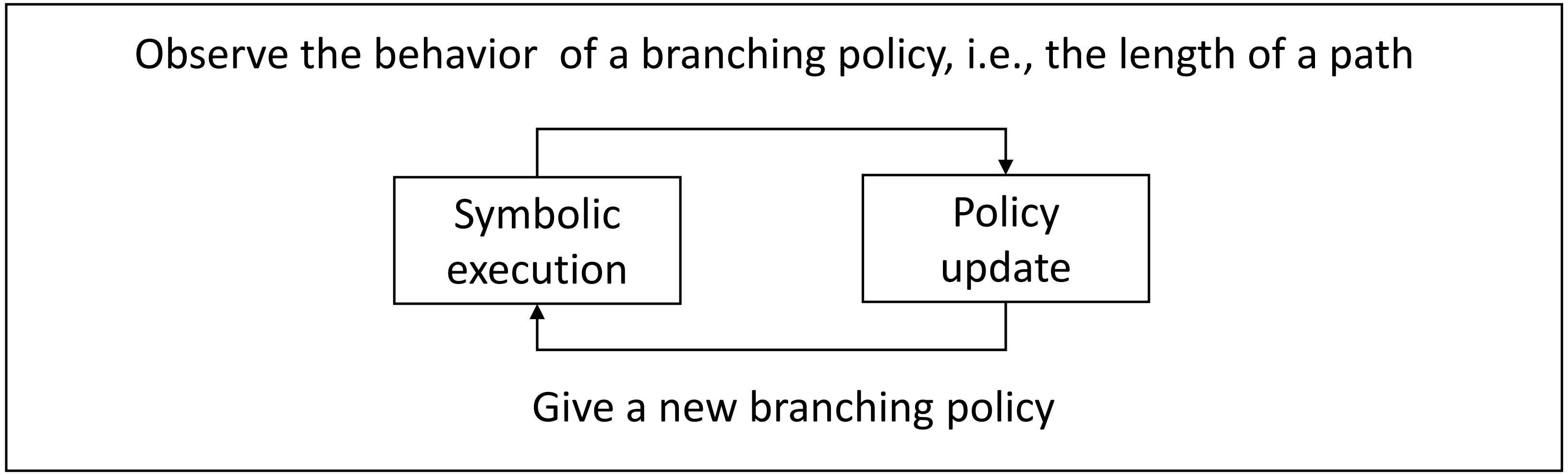
Can we avoid/minimize these issues of white-box based approaches for large-scale test generation?



in- priority queue: no simple way to describe the worst-case branching policy

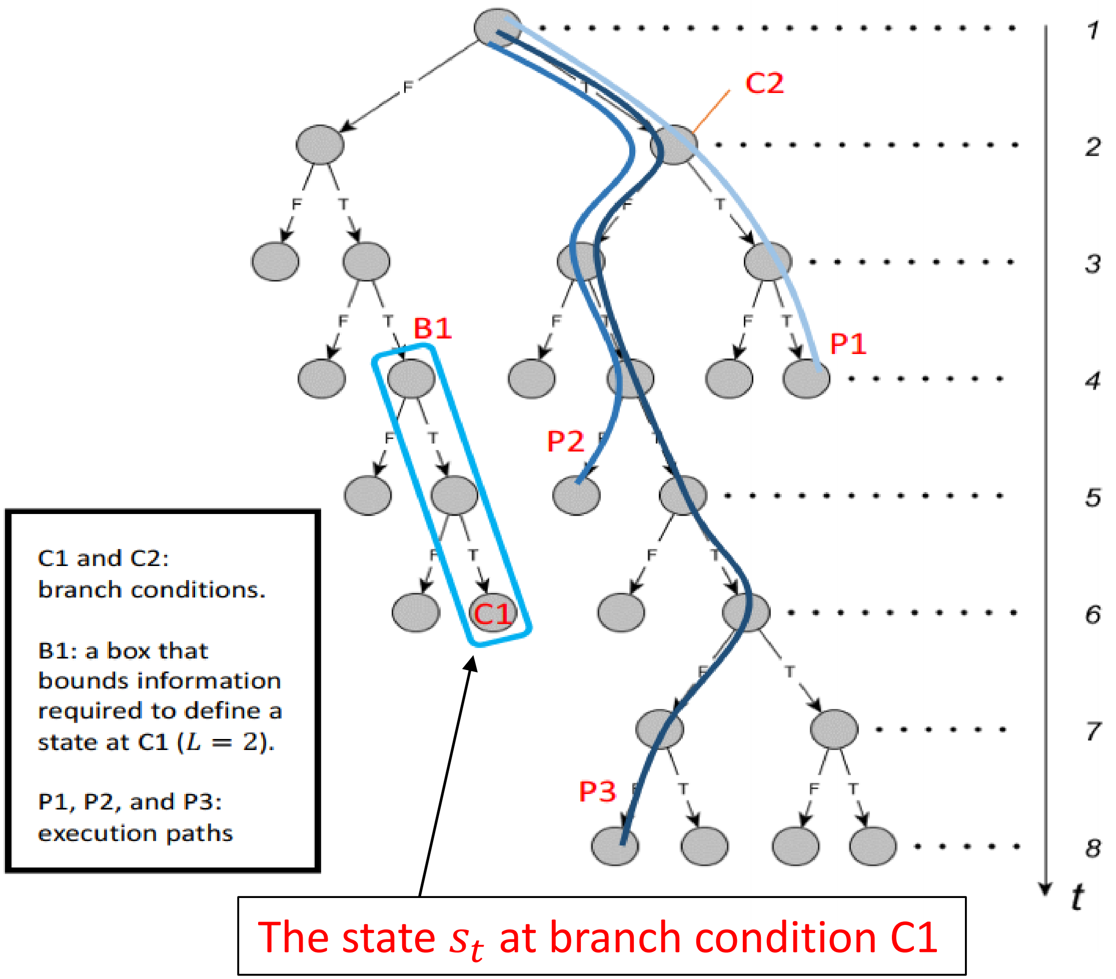
PySE Solution approach

- PySE: learns the worst-case branching policy using Q-learning, a model-free reinforcement learning.
  - Uses symbolic execution to collect behavioral information of a given branching policy
  - Updates the policy based on Q-learning.

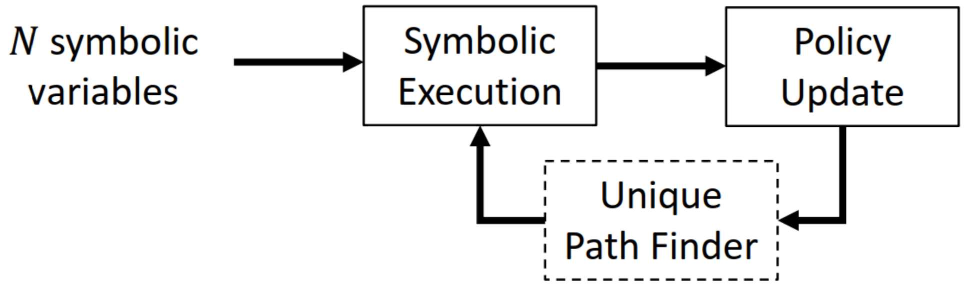


The main objective of PySE

- To find out a **branching policy** $\boldsymbol{\pi(s_t)}$for a given **state** $\boldsymbol{s_t}$at the t-th branch condition that it encounters while a program is being symbolically executed.
  - The branching policy $\pi(s_t)$ determines a **branching decision** $\boldsymbol{a_t} = \pi(s_t) \in \{True, False\}$, which we also call **action**.
  - The state s_t mainly consists of the current branch condition, previous L branch conditions, and actions taken there.
    - ✓ L: the **history length**
  - The branching policy $\pi(s_t)$ continue evolving in such a way that the **length of an execution path** increases.



Workflow of PySE



- Step 1: (SYMBOLIC EXECUTION)
  - Execute a program by the branching policy $\pi(s_t)$.
  - Collect resulting behavioral information such as which branch points the program visits, actions taken at each branch, and feasibilities of the actions.
- Step 2: (POLICY UPDATE)
  - Update the branching policy $\pi(s_t)$ in a way that an undesirable action that caused a program to terminate quickly can be avoided in the future.
    - ✓ Q-learning

Branching policy $\pi(s_t)$

- Design the branching policy $\pi(s_t)$ as:

$$\pi(s_t) = \arg\max_{a_t} Q(s_t, a_t)$$

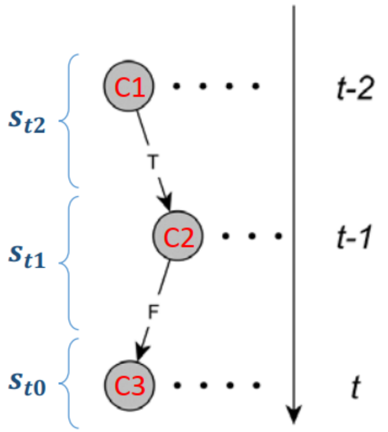
- $Q(s_t, a_t)$ is made from an artificial neural network (ANN), whose inputs are $s_t$and its output layer produces two values, $Q(s_t, True)$ and $Q(s_t, False)$.
  - $\pi(s_t)$= $True$ if $Q(s_t, True) \geq Q(s_t, False)$.
  - $\pi(s_t)$= $False$ if $Q(s_t, True) < Q(s_t, False)$.

State representation

- $s_t = (s_{t0}, s_{t1}, \ldots, s_{tL})$
  - s_{tl}: an integer vector encoding the $(t-l)$-th branch condition and the action taken there.

<u>Examples of branch conditions</u>

C1: line number 150, C2: line number 140, C3: line number 89



F1	F2	F3	
Level	Line number	Action	
$t-2$	0	1	s_{t2}
$t-1$	1	0	s_{t1}
t	2	-1	s_{t0}

Encoding

$$s_t = (s_{t0}, s_{t1}, s_{t2})$$

- Encoding of a state when $L = 2$.
  - F2: unique identifier for each branch point (e.g. line number)
  - F3: action taken at the branch point (1 = TRUE , 0 = FALSE)

How to update the branching policy (1/3)

- Symbolic execution takes action a_t at a given state s_t and observes its consequence.
  - Whether the execution path is still feasible.
  - Feasibility can be checked by using a constraint solver like Z3.
- Depending on the feasibility, the consequence of the action a_t at the state s_t is scored by a **reward** r_t:
  - $r_t = 1$ if feasible, and $r_t = P$ if not feasible.
  - $P = -20$ so that the infeasible decision is more distinguishable from the feasible one.

How to update the branching policy (2/3)

- We want $\pi(s_t)$ to converge to the optimal branching policy $\pi^*(s_t)$ that maximizes the expected sum of future rewards, $E\left(\sum_{k=t}^{T} r_k \,\middle|\, s_t\right)$.
  - *T* denotes the last branch condition before a program terminates normally or falls in an infeasible path condition.
  - Thus, equivalently, it maximizes the length of a feasible execution path.
- Define the optimal action-value function $Q^*(s_t, a_t)$ as the maximum expected sum of future rewards, after taking action $a_t$at a state s_t:

$$Q^*(s_t, a_t) = \max_{\pi} E\left(\sum_{k=t}^{T} r_k \,\middle|\, s_t\right)$$

- $Q^*(s_t, a_t)$ can be re-written recursively as:

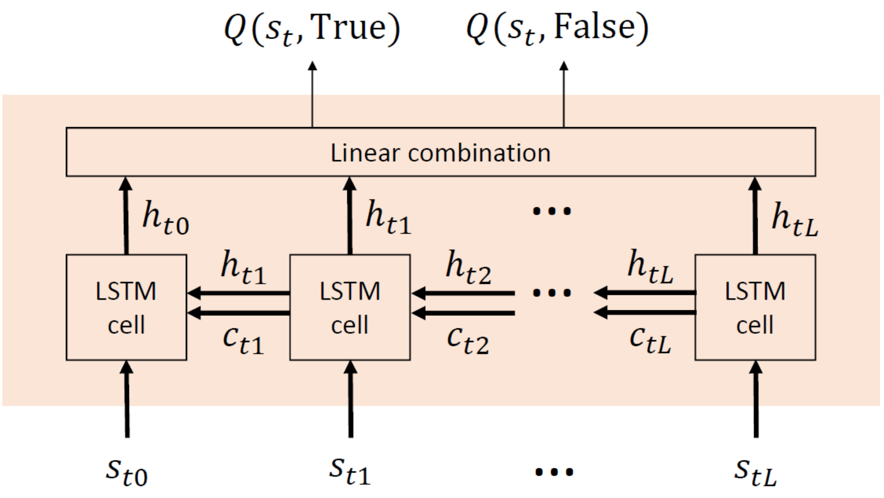
$$Q^*(s_t, a_t) = E(r_t + \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1}))$$

How to update the branching policy (3/3)

- We try to learn $Q^*(s_t, a_t)$ by a sample mean $Q(s_t, a_t)$:
$$Q(s_t, a_t) \leftarrow (1-\alpha)Q(s_t, a_t) + \alpha(r_t + \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}))$$
  - α is called a learning rate.
  - By the law of large numbers, $Q(s_t, a_t)$ can converge to $Q^*(s_t, a_t)$ after iterations for a sufficiently small value of α.
  - Such an update for learning $Q(s_t, a_t)$ without knowing the underlying probability distribution model is referred to as Q-learning in the reinforcement learning literature.

Q-network architecture

- In practice, updating $Q(s_t, a_t)$ separately for each (s_t, a_t) is unattainable.
  - This is because the state is a multi-dimensional integer vector and thus the number of possible states can be too large.
- Thus, a function approximator is commonly used to estimate the function $Q(s_t, a_t)$ with the limited number of observations for state-action pairs.
- PySE also represents $Q(s_t, a_t)$ by using an ANN-based function approximator, which we refer to as a **Q-network**.



Algorithm of PySE

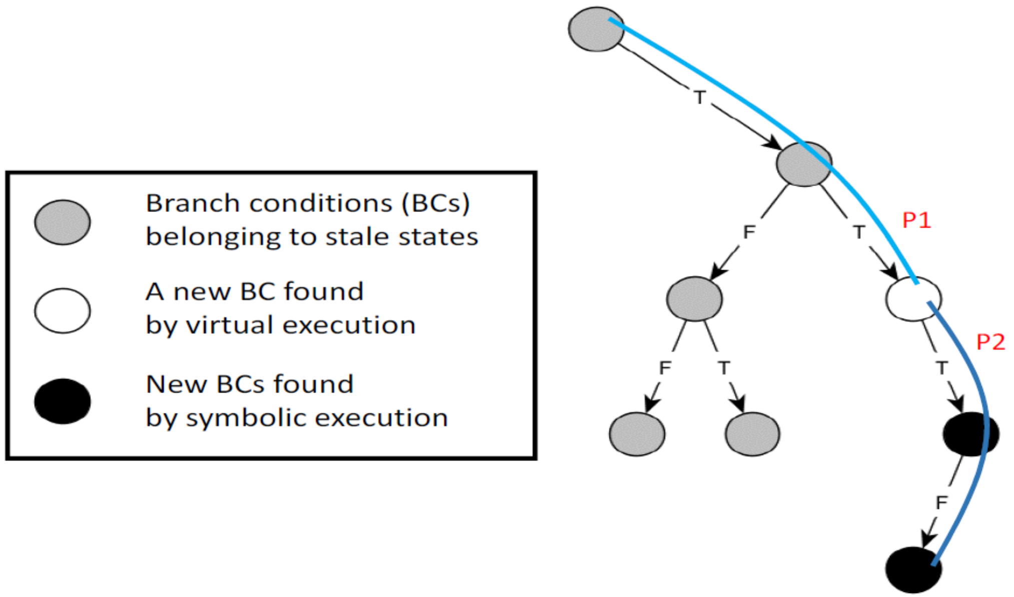
Algorithm 1 Basic mode of PySE

procedure SYMBOLIC EXECUTION
  for t from 1 to T **do**
    Choose a number u randomly over $[0, 1]$.
    if $u < \epsilon$ **then**
      Choose a_t randomly. ▷ ϵ-greedy.
    else
      $a_t = \pi(s_t)$.
    Execute a_t, and observe r_t and s_{t+1}.
    if the **experience** $e_t = (s_t, a_t, r_t, s_{t+1})$ is new **then**
      Add e_t in E.
  Delete old experiences in E to keep $|E| \leq N_e$.
procedure POLICY UPDATE
  Sort experiences in E in a random order.
  for i from 1 to $|E|$ **do**
    Read the i-th experience from E.
    Update weights.

- Exploration of a new path by ϵ-greedy strategy.
  - With ϵ probability, take random action instead of $\pi(s_t)$.
- Symbolic Execution step collects what is called the experience.
  - $e_t = (s_t, a_t, r_t, s_{t+1})$
- Policy Update step uses these experiences to update the Q-network.

Unique Path Finder (UPF)

- UPF attempts to help us gather at least one new experience in each symbolic execution step.
- Virtual execution:
  - Defined as a sequence of state transitions using $\pi(s_t)$ with an ϵ- greedy strategy over an observed computation tree, which means a computation tree built up by all of observed experiences.
  - Namely, the virtual execution is not an execution of a real program, but a simulation of state transitions among states that have been already observed.
  - Such a simulation takes negligible time to run.



Unique Path Finder that discovers a prefix (P1) of a brand-new execution path by virtual execution, which is a run over a computation tree built by observed experiences. Symbolic execution that follows is guided by the prefix P1 and finds out the remaining (P2) of the new execution path.

Experiments

- Class 1 programs:
  - The worst-case branch behavior is continuous and follows a simple pattern like “always True” or “always False”
  - These are the programs where WISE is effective, and SPF-WCA works exactly the same as WISE.
- Class 2 programs:
  - Some or all of branch points have a **irregular branch behavior** in the worst case.
  - the worst-case-leading decision at a branch point can change depending on the scale (N), or the time (t) that the branch point is visited.
  - WISE cannot handle Class 2 programs efficiently.
  - SPF-WCA can be effective for some of them, *i.e.*, when the pattern can be expressed in terms of the history-length

Class 1 example

Benchmark 1: Biopython parewise2: Smith-Waterman [39]	(N, longest path length)		(3,9)	(4,12)	(5,15)	(10,30)	(20,60)	(30,90)	(100,300)
	Exhaustive search	Paths	127	511	2047	-	-	-	-
		Time	0:04	0:18	1:14	-	-	-	-
	WISE	Paths	1	1	1	1	1	1	1
		Time	0:00	0:00	0:00	0:00	0:00	0:00	0:01
	PySE	Paths	1	1	1	1	1	1	2
		Time	0:02	0:02	0:02	0:02	0:02	0:02	0:13

- Exhaustive search: search time exponentially grows
- WISE: small-scale tests predict the worst-case at a larger scale.
- PySE: finds the worst-case within a few trials.

Class 2 example (1/2)

	(N, longest path length)		(3,3)	(4,3)	(5,3)	(10,9)	(20,18)	(30,30)	(100,99)
GNU grep : Boyer-Moore	Exhaustive search	Paths	4	4	4	40	1093	88573	-
		Time	0:00	0:00	0:00	0:01	0:31	43:39	-
	WISE	Paths	4	4	4	40	1093	88573	-
		Time	0:00	0:00	0:00	0:01	0:32	44:24	-

- WISE cannot handle: GNU grep's worst-case branching behavior shows an irregular pattern

Class 2 example (2/2)

	(N, longest path length)		(3,3)	(4,3)	(5,3)	(10,9)	(20,18)	(30,30)	(100,99)
GNU grep : Boyer-Moore	SPF-WCA trained at N=3,4	Paths	1	1	1	9	243	19683	-
		Time	0:00	0:00	0:00	0:00	00:07	10:20	-
	SPF-WCA trained at N=6,7	Paths	1	1	1	1	1	1	1
		Time	0:00	0:00	0:00	0:00	0:00	0:00	0:00
	PySE pre-trained at N = 5	Paths	2	2	2	2	2	3	276
		Time	0:11	0:11	0:11	0:11	0:12	0:20	48:21
	PySE pre-trained at N = 10	Paths	2	2	2	1	2	3	82
		Time	0:11	0:11	0:11	0:02	0:12	0:20	13:03

- SPF-WCA may handle, but its performance is sensitive to the length of history.
- PySE can handle it and the length of history is not critical.

Concluding remarks

- PySE uses symbolic execution to run a program and collects behavioral information.
- PySE then updates a branching policy using the collected behaviors based on a reinforcement learning framework.
- By iterating the symbolic execution and policy update, PySE gradually increases the length of an execution path towards a path of the worst-case complexity.
- In various Python programs and scales, PySE can effectively find a path of worst-case complexity and has benefits against exhaustive search and WISE-like algorithms.

Thank you!