SOFTWARE RELIABILITY GROUP

Imperial College London

Pending Constraints in Symbolic Execution for Better Exploration and Seeding

Timotej Kapus • Frank Busse • Cristian Cadar

3rd International KLEE Workshop on Symbolic Execution
15–16 September 2022, London

ASE 2020

Project

https://srg.doc.ic.ac.uk/projects/pending-constraints/

KLEE PR #1334

https://github.com/klee/klee/pull/1334

Pending Constraints in Symbolic Execution for Better Exploration and Seeding

Timotej Kapus
Imperial College London
United Kingdom
t.kapus@imperial.ac.uk

Frank Busse
Imperial College London
United Kingdom
f.busse@imperial.ac.uk

Cristian Cadar
Imperial College London
United Kingdom
c.cadar@imperial.ac.uk

ABSTRACT

Symbolic execution is a well established technique for software testing and analysis. However, scalability continues to be a challenge, both in terms of constraint solving cost and path explosion. In this work, we present a novel approach for symbolic execution, which can enhance its scalability by aggressively prioritising execution paths that are already known to be feasible, and deferring all other paths. We evaluate our technique on nine applications, including *SQLite3*, *make* and *tcpdump* and show it can achieve higher coverage for both seeded and non-seeded exploration.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

Symbolic execution, KLEE

ACM Reference Format:
Timotej Kapus, Frank Busse, and Cristian Cadar. 2020. Pending Constraints in Symbolic Execution for Better Exploration and Seeding. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416589

1 INTRODUCTION

Symbolic execution is a dynamic program analysis technique that has established itself as an effective approach for many software engineering problems such as test case generation [4, 12], bug finding [6, 13], equivalence checking [10, 11], vulnerability analysis [8, 27] and debugging [14, 20].

Even with well-engineered tools like KLEE [4], symbolic execution still faces important scalability challenges. These fall into two broad categories: constraint solving and path explosion. As symbolic execution proceeds, the complexity of constraints and the number of paths typically increase, often making it difficult to make meaningful progress.

In this work, we propose a novel mechanism that aggressively explores paths whose feasibility is known via caching or seeding.

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416589

Our approach tackles both scalability challenges of symbolic execution. On the one hand, it enables more efficient use of solved constraints, thus reducing the burden on the solver. And on the other hand, it provides a meta-search heuristic that gives a way to guide the exploration towards interesting parts of the program.

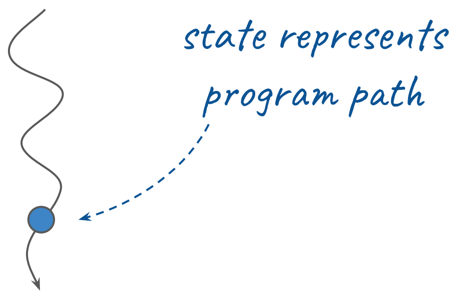
Before presenting our idea, we briefly summarise symbolic execution. We focus here on the *EGT-style* of dynamic symbolic execution [5], embodied in tools such as KLEE [4], which unlike *concolic execution* tools [12, 24], store partially explored paths in memory. Symbolic execution works by running the program on some symbolic inputs, which means they can initially take any value, as they are unconstrained. During execution, if a branch condition depends on a symbolic value, symbolic execution queries an SMT solver for the feasibility of each of the two branches (under the current path condition which is initially empty). If both the `then` and the `else` branches are feasible, it forks the execution exploring both paths and adding the respective branch conditions to each path condition (PC). After every fork, symbolic execution uses a search heuristic to decide what path to explore next. Each path explored in symbolic execution is encoded by a *state* which keeps all the information necessary to resume execution of the associated path (PC, program counter, stack contents, etc.).

The core of our idea revolves around *inverting* the forking process. Instead of doing an (expensive) feasibility check first and then forking the execution, we fork the execution first. The branch condition is then added as a *pending constraint*, which means its feasibility has not been checked yet. We refer to states (or paths) with pending path constraints as *pending states*.

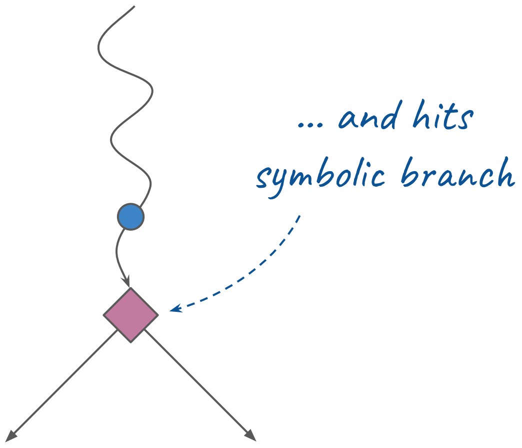
The responsibility for feasibility checking of pending path constraints is passed to the search heuristic. This gives the search heuristic the capability to decide when and for which states it wants to pay the price of constraint solving. For example, it could solve pending states immediately, thus restoring the original algorithm, or could take into account the (estimated) cost of constraint solving in its decisions.

In our approach, we take advantage of an important characteristic of symbolic execution runs: the feasibility of some paths/states can be quickly determined without using a constraint solver. There are two common cases. First, modern symbolic execution systems like KLEE make intensive use of caching and many queries can be solved without involving the constraints solver [1, 4, 26]. Second, symbolic execution is often bootstrapped with a set of seeds from which to start exploration: these can come from regression test suites [18, 19] or greybox fuzzers in hybrid greybox/whitebox fuzzing systems [9, 21, 25]. By aggressively following paths for which feasibility can be quickly determined without using a constraint solver, our approach can minimise the constraint solving

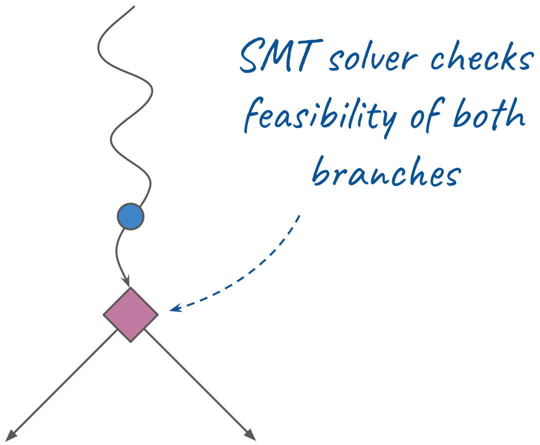
Symbolic Execution



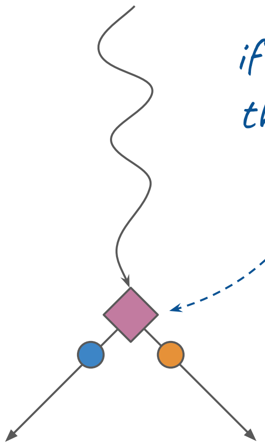
Symbolic Execution



Symbolic Execution

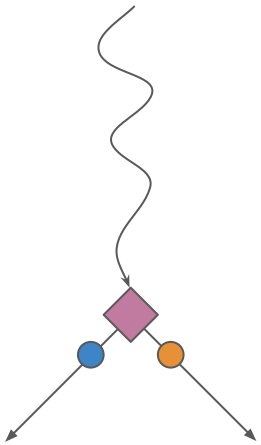


Symbolic Execution



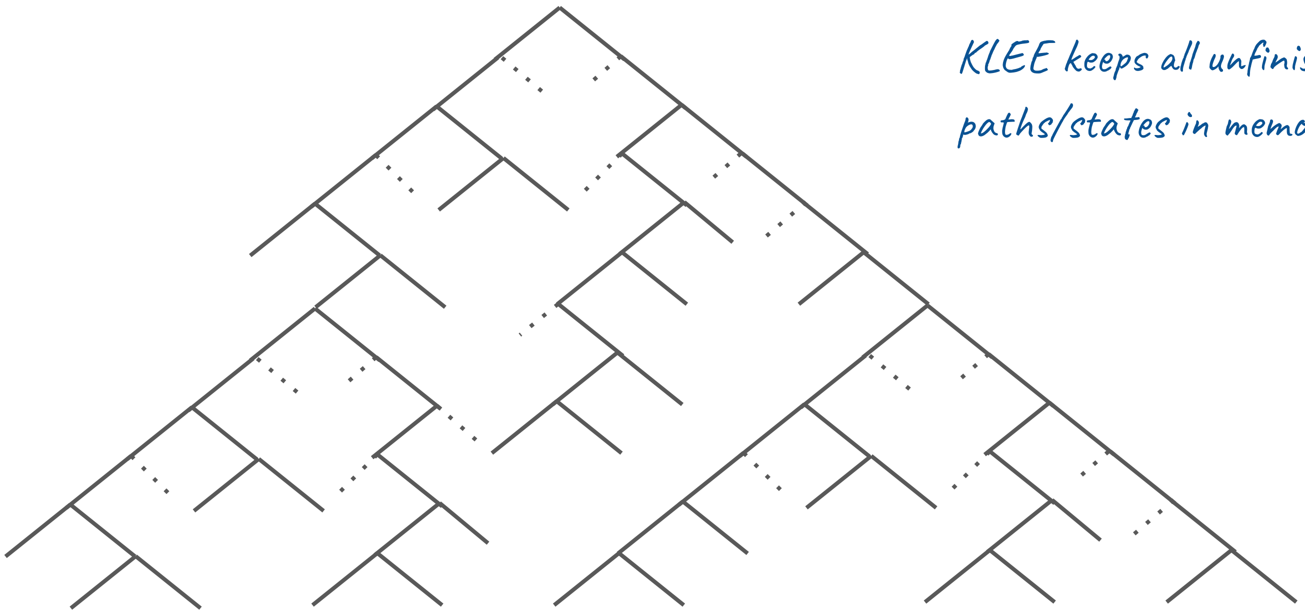
if both branches are feasible
the state is forked to explore
both branches

Symbolic Execution



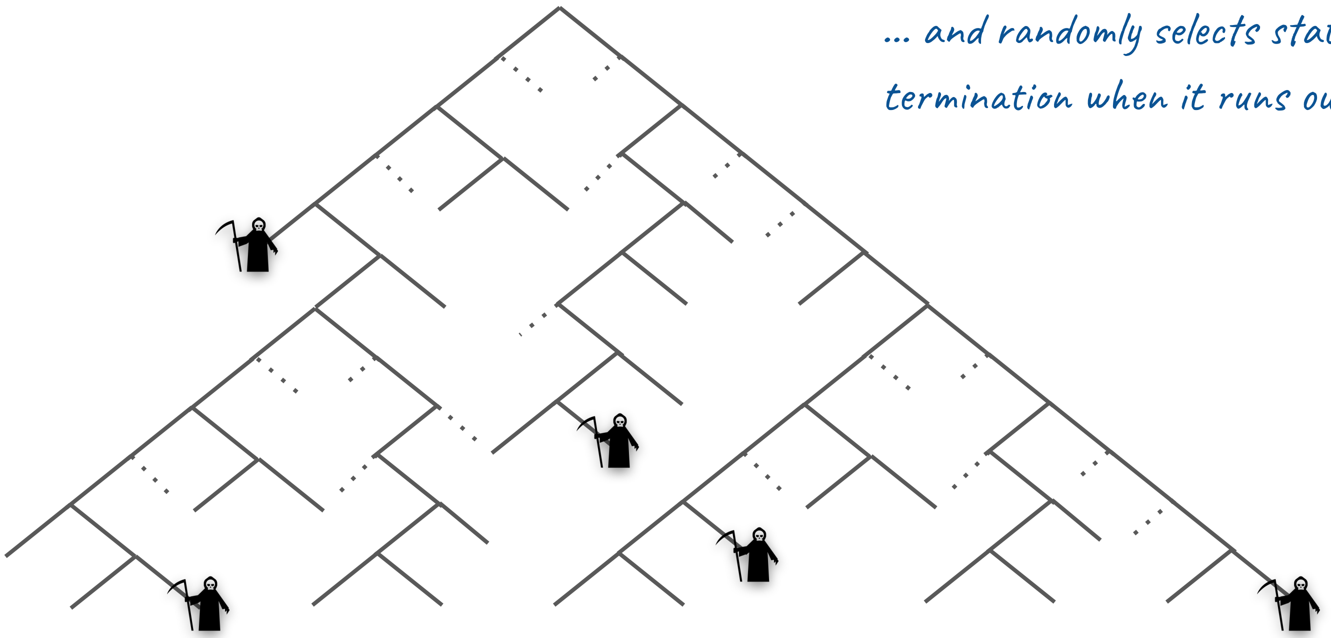
"Searcher" selects next state for exploration.

KLEE's "EGT-style" Execution

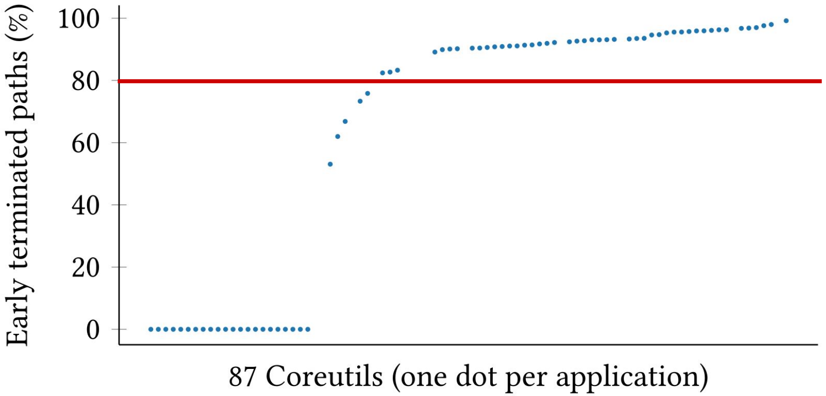


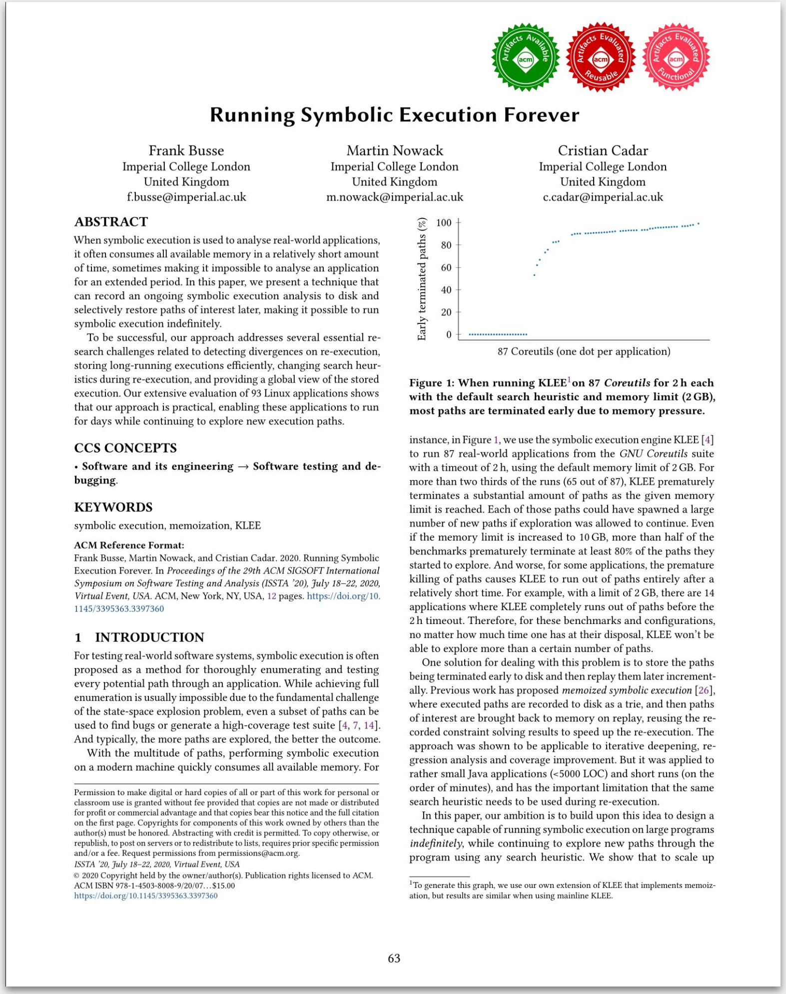
KLEE keeps all unfinished paths/states in memory

KLEE's "EGT-style" Execution



... and randomly selects states for early termination when it runs out of memory





Running Symbolic Execution Forever

Frank Busse
Imperial College London
United Kingdom
f.busse@imperial.ac.uk

Martin Nowack
Imperial College London
United Kingdom
m.nowack@imperial.ac.uk

Cristian Cadar
Imperial College London
United Kingdom
c.cadar@imperial.ac.uk

ABSTRACT

When symbolic execution is used to analyse real-world applications, it often consumes all available memory in a relatively short amount of time, sometimes making it impossible to analyse an application for an extended period. In this paper, we present a technique that can record an ongoing symbolic execution analysis to disk and selectively restore paths of interest later, making it possible to run symbolic execution indefinitely.

To be successful, our approach addresses several essential research challenges related to detecting divergences on re-execution, storing long-running executions efficiently, changing search heuristics during re-execution, and providing a global view of the stored execution. Our extensive evaluation of 93 Linux applications shows that our approach is practical, enabling these applications to run for days while continuing to explore new execution paths.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

symbolic execution, memoization, KLEE

ACM Reference Format:
Frank Busse, Martin Nowack, and Cristian Cadar. 2020. Running Symbolic Execution Forever. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3395363.3397360

1 INTRODUCTION

For testing real-world software systems, symbolic execution is often proposed as a method for thoroughly enumerating and testing every potential path through an application. While achieving full enumeration is usually impossible due to the fundamental challenge of the state-space explosion problem, even a subset of paths can be used to find bugs or generate a high-coverage test suite [4, 7, 14]. And typically, the more paths are explored, the better the outcome.

With the multitude of paths, performing symbolic execution on a modern machine quickly consumes all available memory. For

ACM ISBN 978-1-4503-8008-9/20/07…$15.00
https://doi.org/10.1145/3395363.3397360

Early terminated paths (%) — 100, 80, 60, 40, 20, 0 — 87 Coreutils (one dot per application)

Figure 1: When running KLEE[1] on 87 *Coreutils* for 2 h each with the default search heuristic and memory limit (2 GB), most paths are terminated early due to memory pressure.

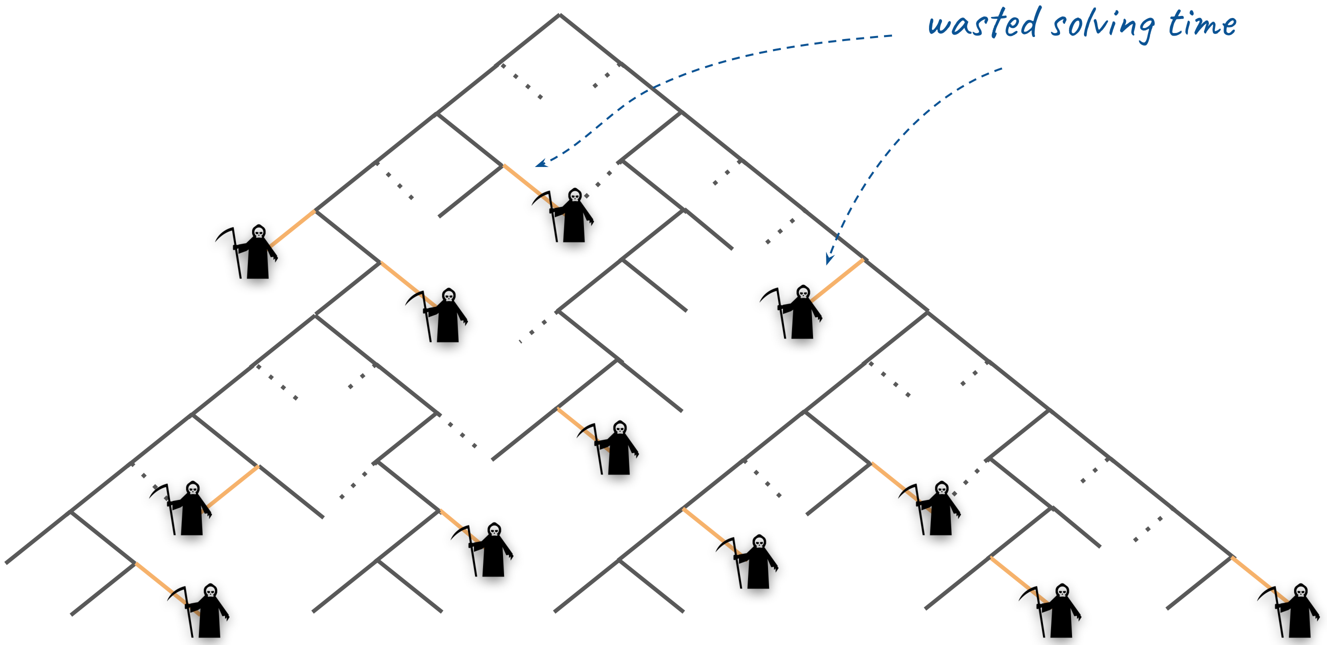
instance, in Figure 1, we use the symbolic execution engine KLEE [4] to run 87 real-world applications from the *GNU Coreutils* suite with a timeout of 2 h, using the default memory limit of 2 GB. For more than two thirds of the runs (65 out of 87), KLEE prematurely terminates a substantial amount of paths as the given memory limit is reached. Each of those paths could have spawned a large number of new paths if exploration was allowed to continue. Even if the memory limit is increased to 10 GB, more than half of the benchmarks prematurely terminate at least 80% of the paths they started to explore. And worse, for some applications, the premature killing of paths causes KLEE to run out of paths entirely after a relatively short time. For example, with a limit of 2 GB, there are 14 applications where KLEE completely runs out of paths before the 2 h timeout. Therefore, for these benchmarks and configurations, no matter how much time one has at their disposal, KLEE won't be able to explore more than a certain number of paths.

One solution for dealing with this problem is to store the paths being terminated early to disk and then replay them later incrementally. Previous work has proposed *memoized symbolic execution* [26], where executed paths are recorded to disk as a trie, and then paths of interest are brought back to memory on replay, reusing the recorded constraint solving results to speed up the re-execution. The approach was shown to be applicable to iterative deepening, regression analysis and coverage improvement. But it was applied to rather small Java applications (<5000 LOC) and short runs (on the order of minutes), and has the important limitation that the same search heuristic needs to be used during re-execution.

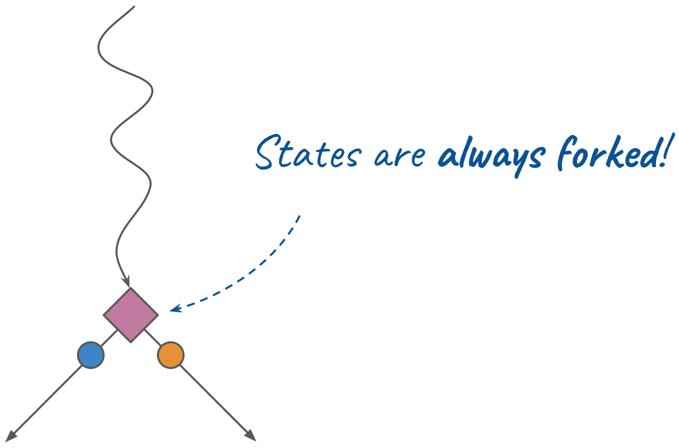
In this paper, our ambition is to build upon this idea to design a technique capable of running symbolic execution on large programs *indefinitely*, while continuing to explore new paths through the program using any search heuristic. We show that to scale up

[1] To generate this graph, we use our own extension of KLEE that implements memoization, but results are similar when using mainline KLEE.

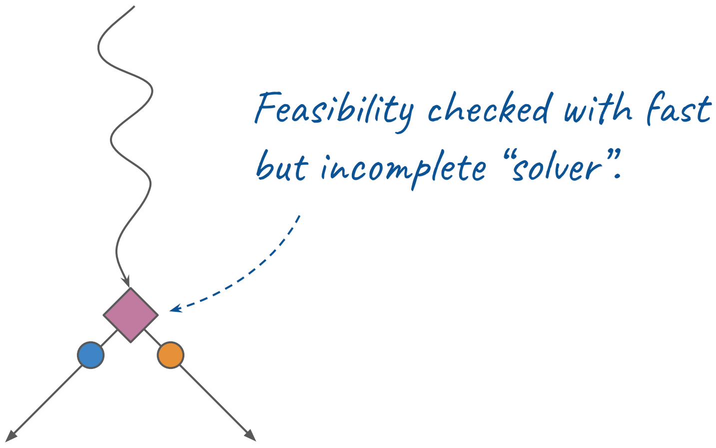
KLEE's "EGT-style" Execution



Symbolic Execution with Pending Constraints

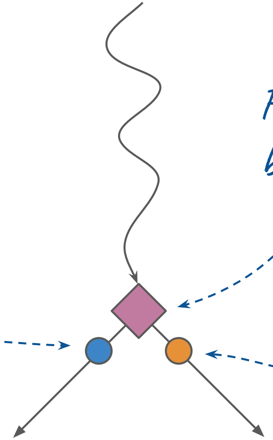


Symbolic Execution with Pending Constraints



Symbolic Execution with Pending Constraints

Feasibility checked with fast but incomplete "solver".



path is feasible

proceed as in normal symbolic execution

path is ??

*do not alter path condition but add constraint as **"pending constraint"***

State Selection

- global state set split into **feasible** and **pending** states
- searchers select from feasible states
- if none left, pending states are **revived**
  - pending constraint is finally checked
  - infeasible states are removed and feasible states are selected

Fast incomplete “solver”

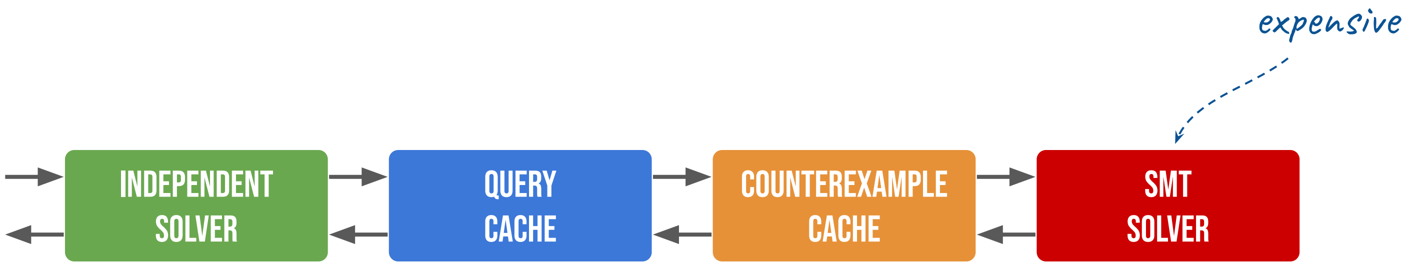
Explore paths that are **known to be feasible**!

Fast incomplete “solver”

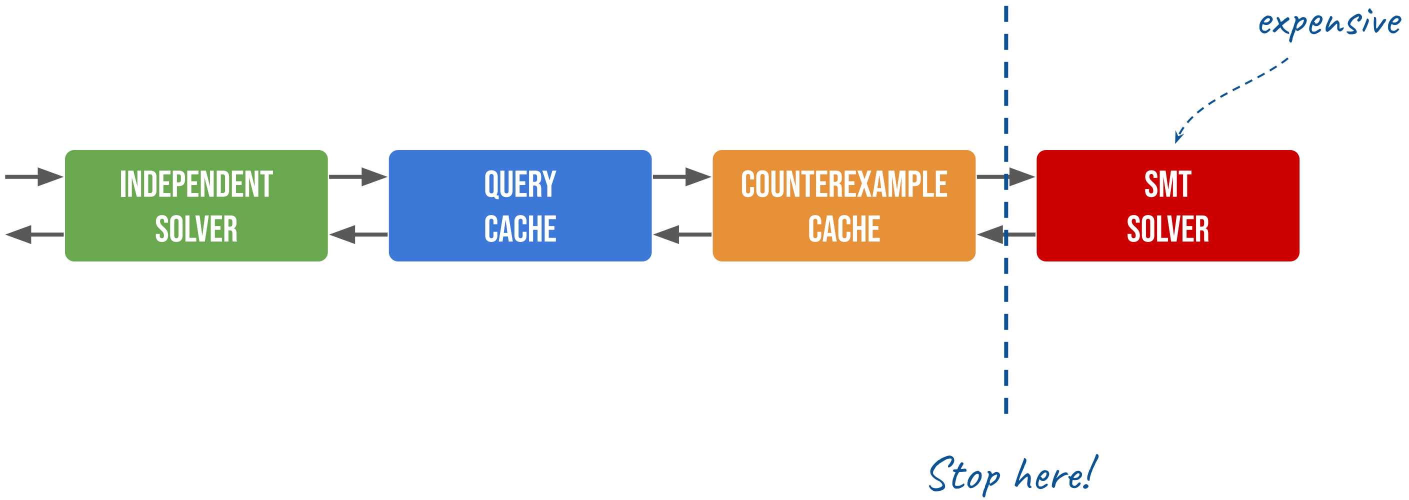
Explore paths that are **known to be feasible**!

- paths where symbolic variables have concrete assignments that satisfy the path condition
    - **seeds** (test cases, production data, fuzzing, …)
    - **cached assignments**

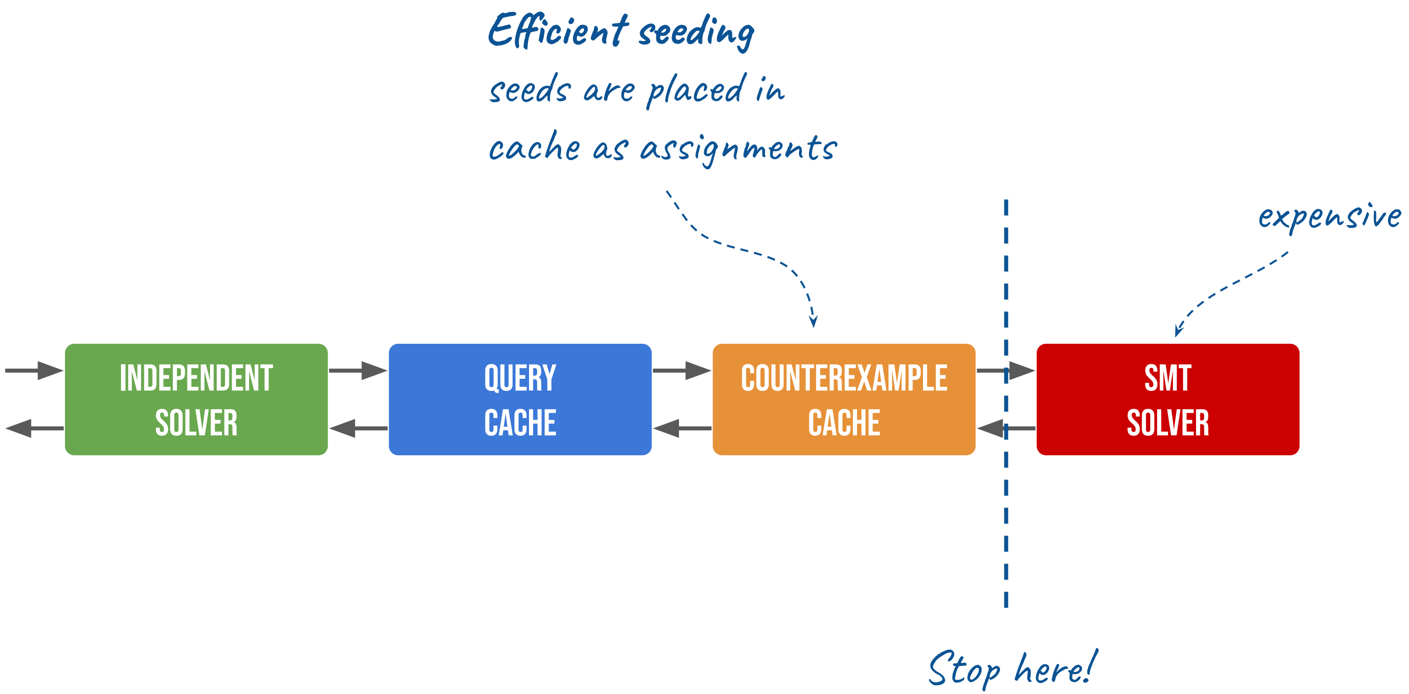
KLEE's solver chain



Fast incomplete “solver”



Seeding



Efficient seeding
seeds are placed in
cache as assignments
expensive
INDEPENDENT SOLVER
QUERY CACHE
COUNTEREXAMPLE CACHE
SMT SOLVER
Stop here!

Example

Solve constraints only when necessary to make progress

Explore paths that are **known to be feasible**

```
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```

```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```

Known assignments

$\varnothing$

`get_sign(`***x***`);`

```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```

Known assignments

$\emptyset$

`get_sign(x);`

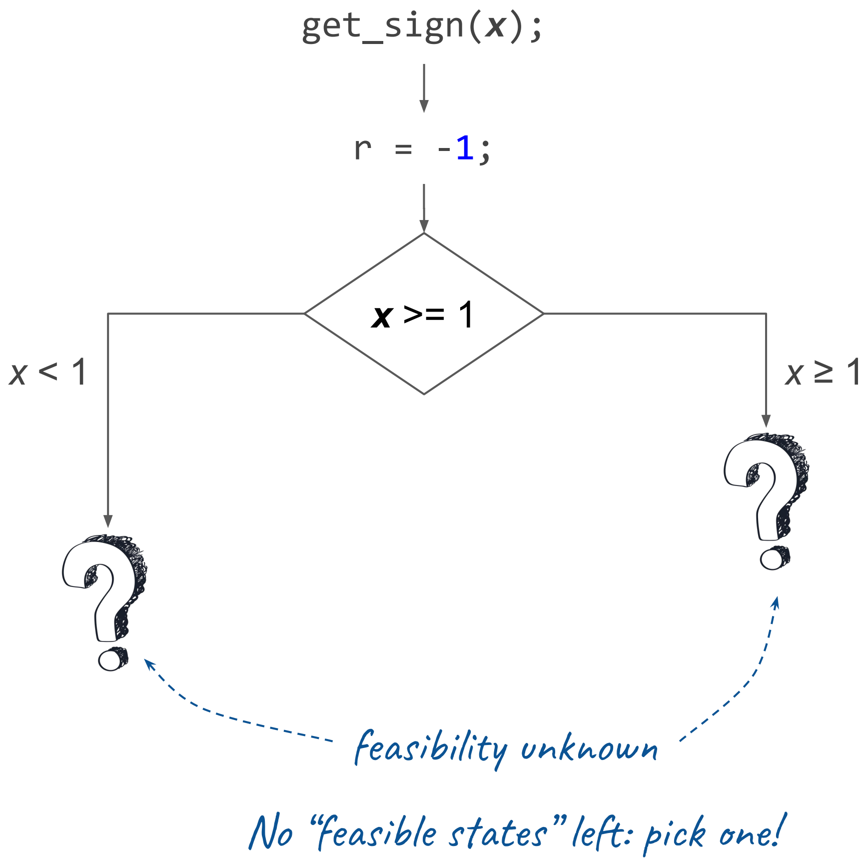
↓

`r = -1;`

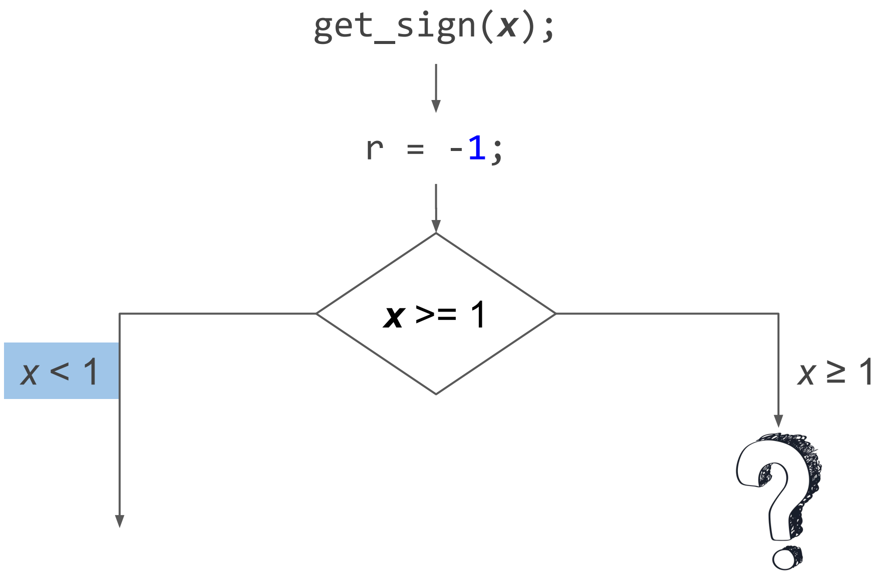
```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```

Known assignments

∅



```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```

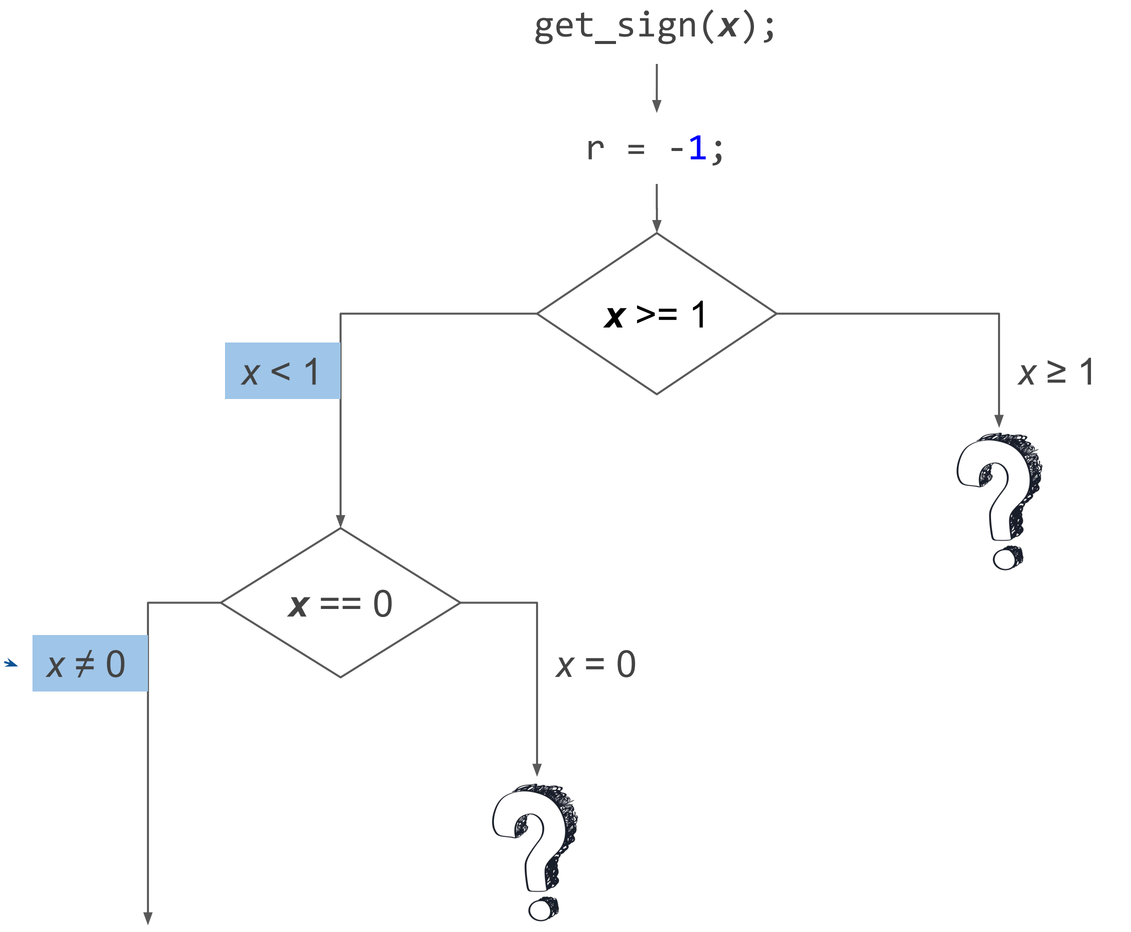


Known assignments

x = -2

```
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```

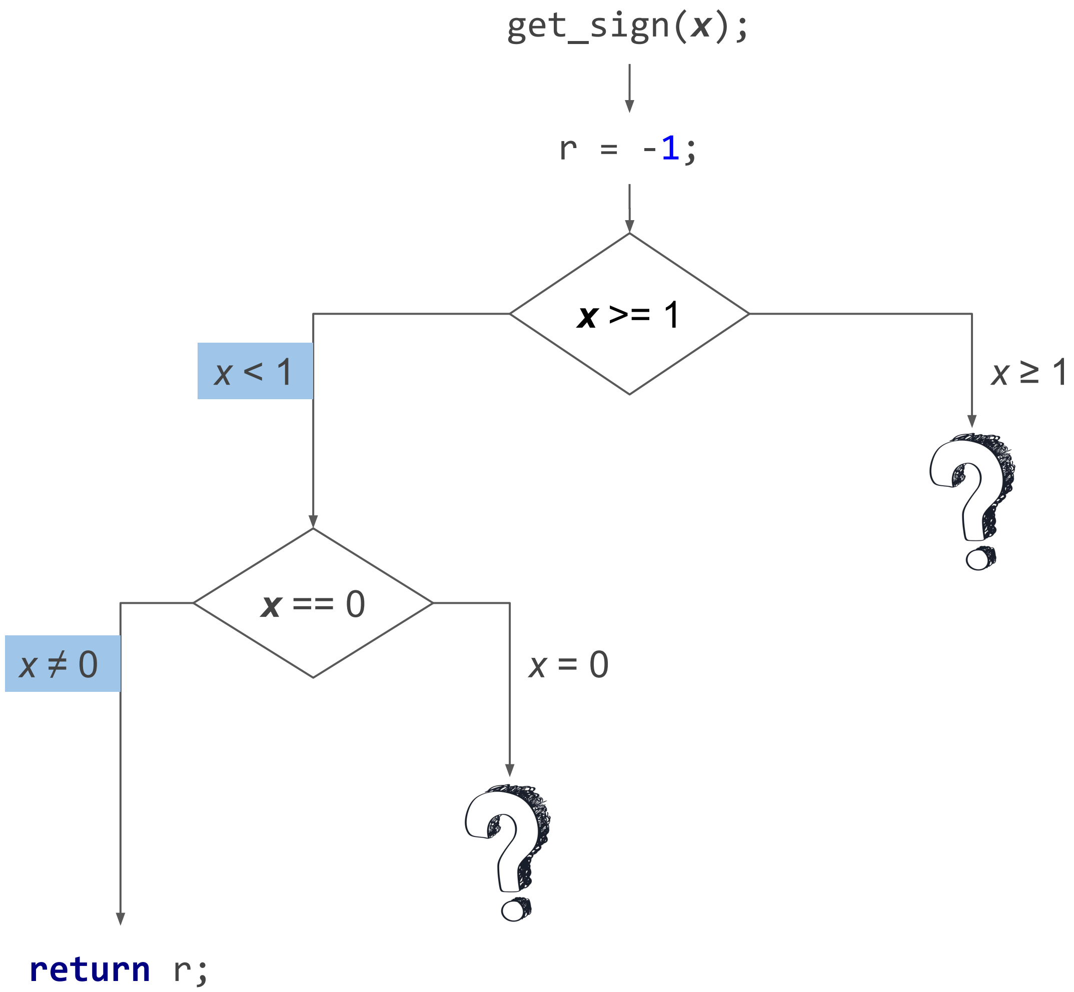
known feasible path



Known assignments

$\boldsymbol{x} = -2$

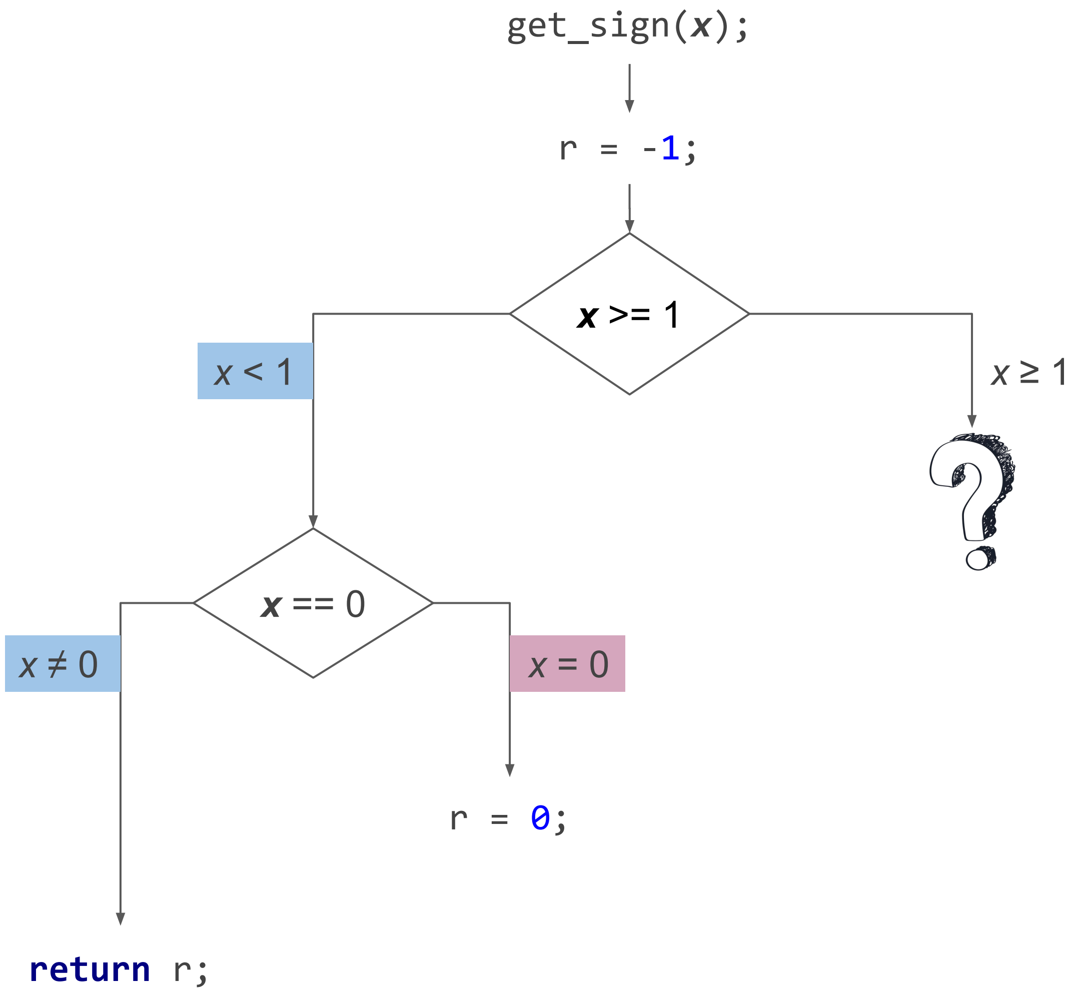
```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```



Known assignments

$x = -2$

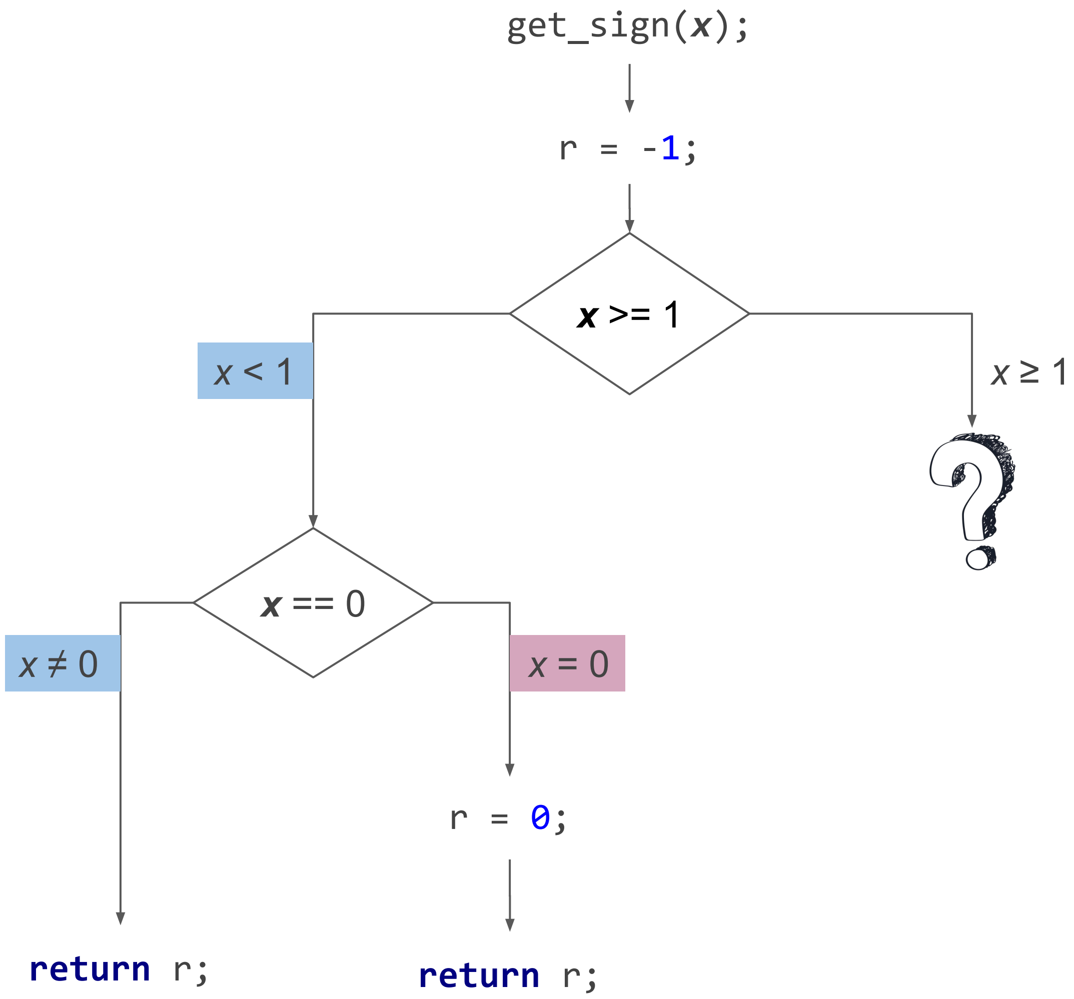
```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```



Known assignments

x = -2
x = 0

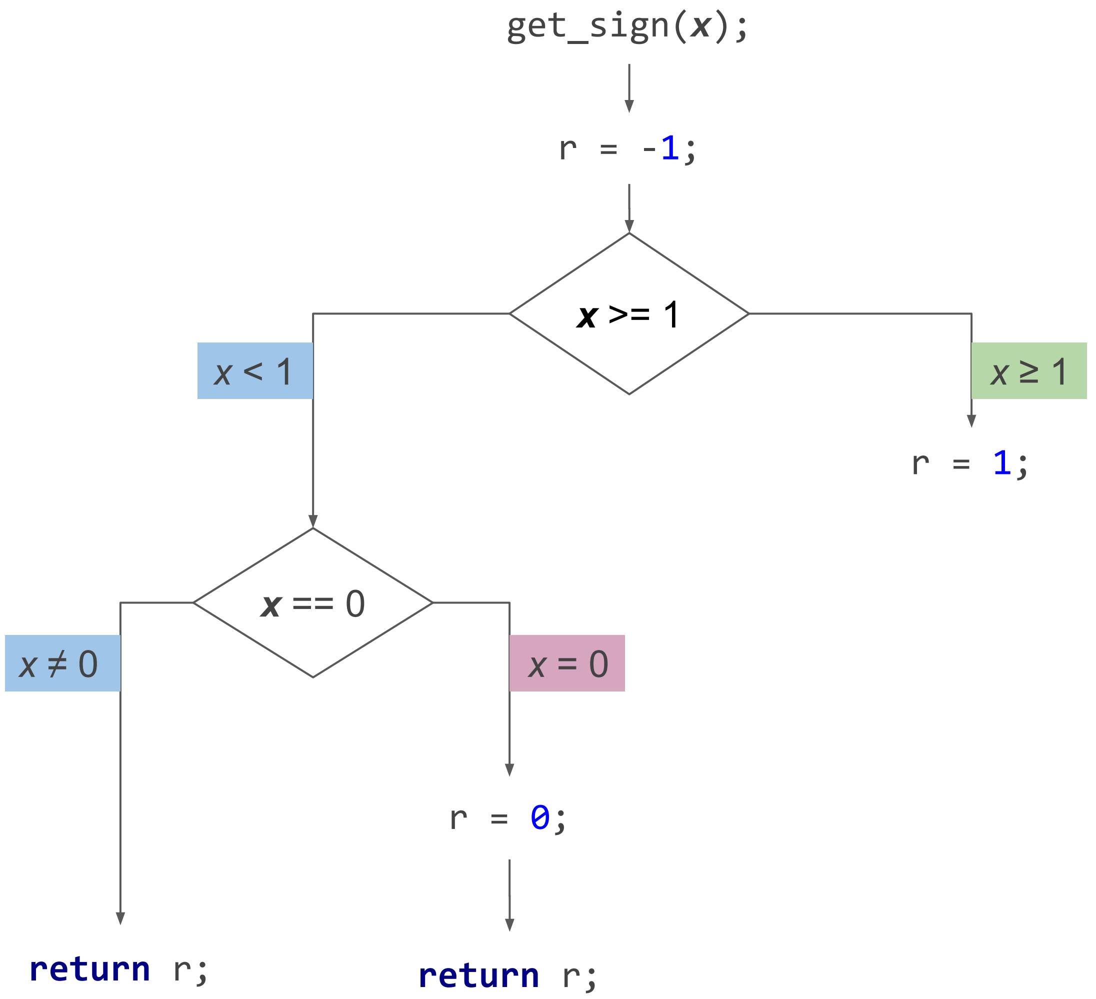
```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```



Known assignments

x = -2
x = 0

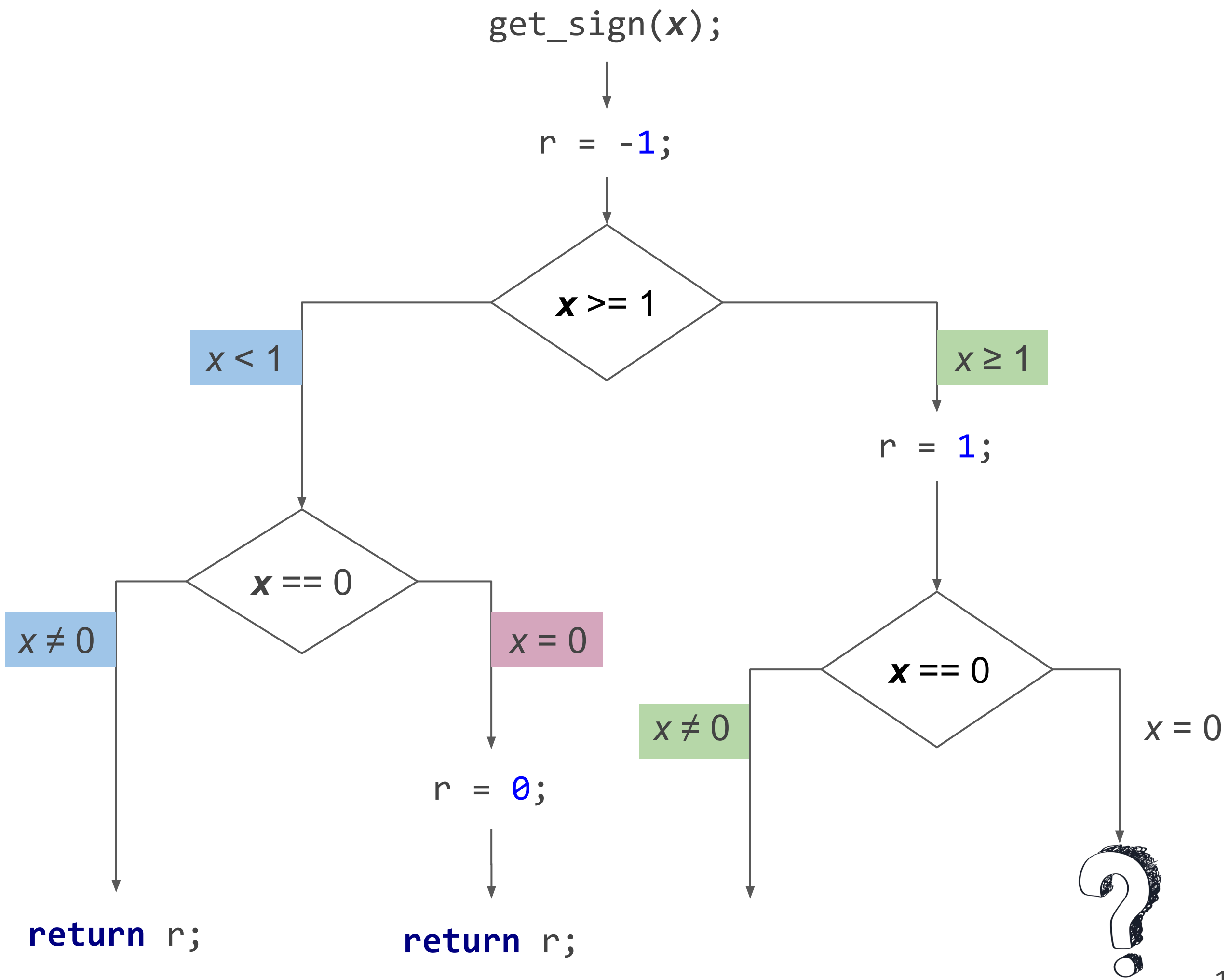
```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```



Known assignments

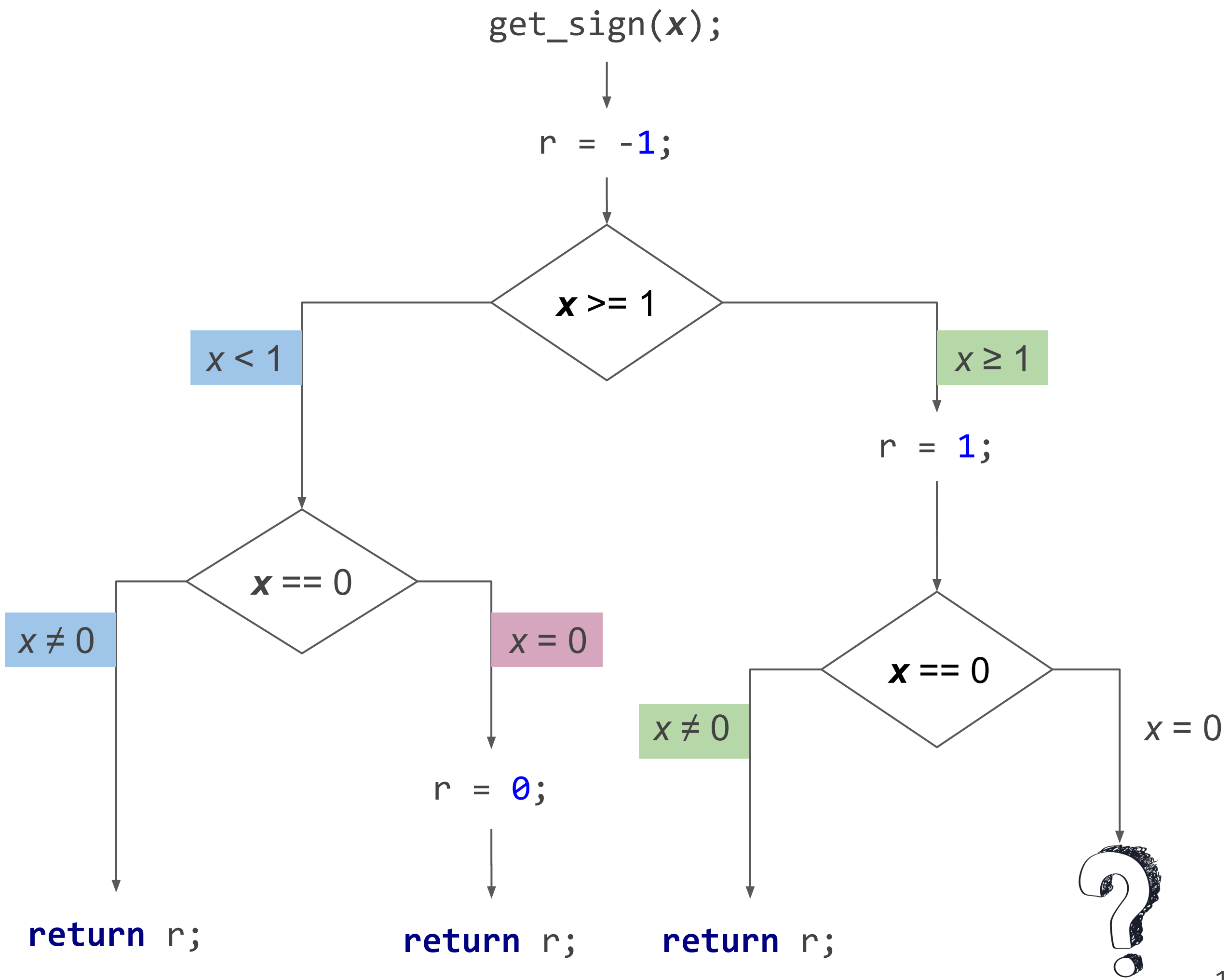
x = -2
x = 7
x = 0

```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```



Known assignments
$x = -2$
$x = 7$
$x = 0$

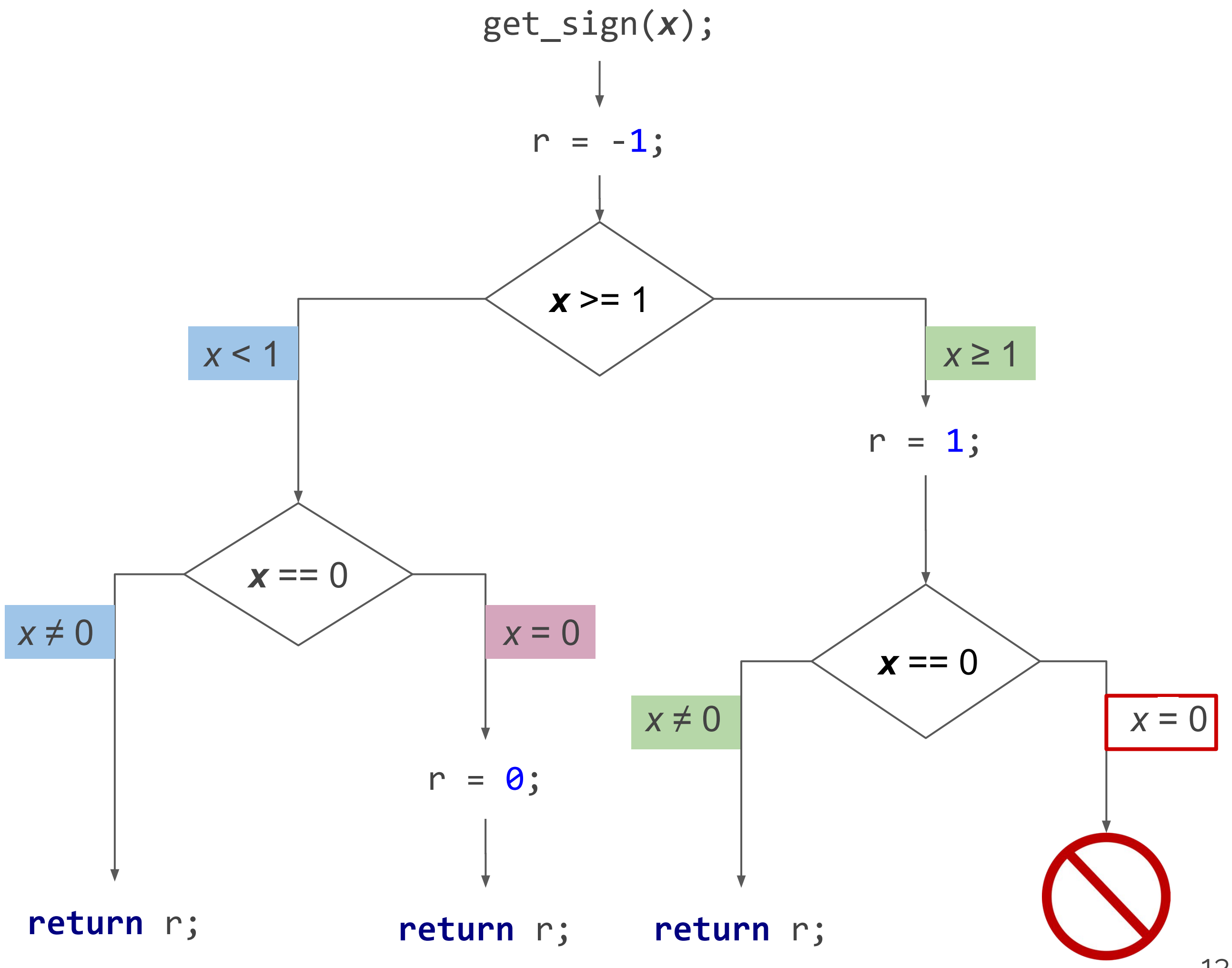
```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```



Known assignments

x = -2
x = 7
x = 0

```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```



Known assignments

$x = -2$
$x = 7$
$x = 0$

Example - Seeding

Reversing md5 hash is hard for SMT solvers

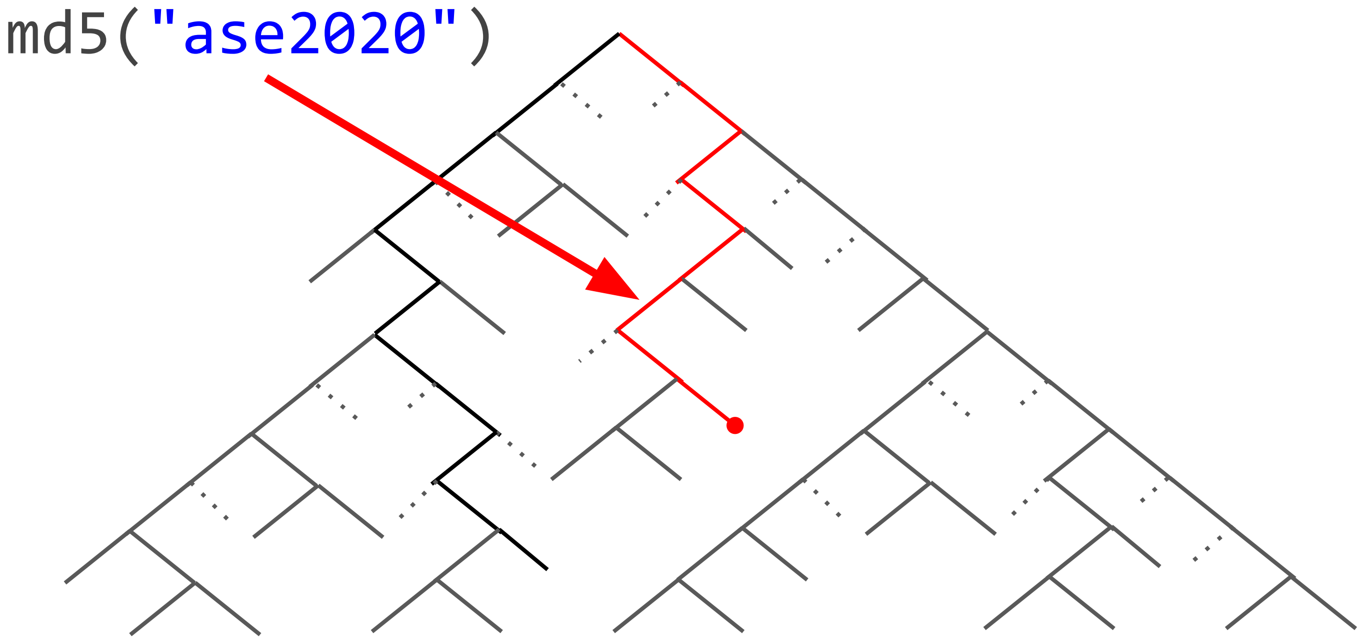
Use

```
1471037522 = md5("ase2020")[0]
```

as seed.

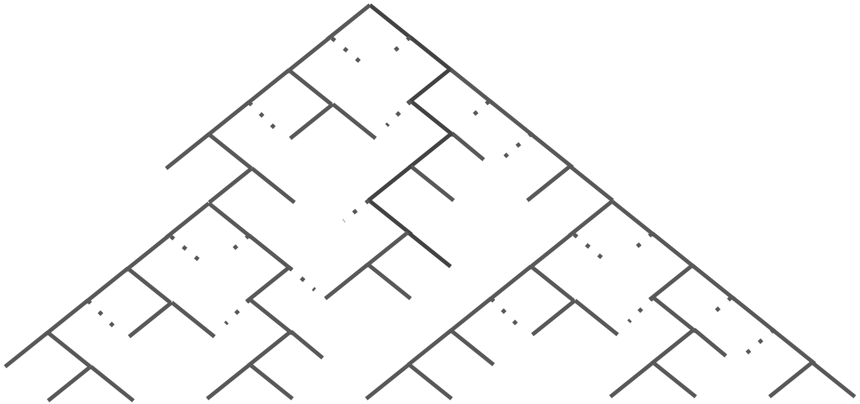
```c
char msg[8] = symbolic;
uint32_t *hash = md5(msg, 8);
assert(hash[0] == 1471037522);
```

Suppose this exploration tree for md5

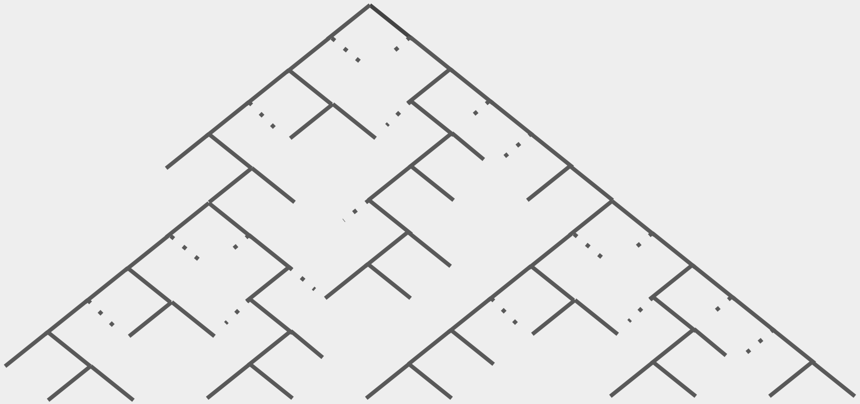


Solver queries: 0

Pending

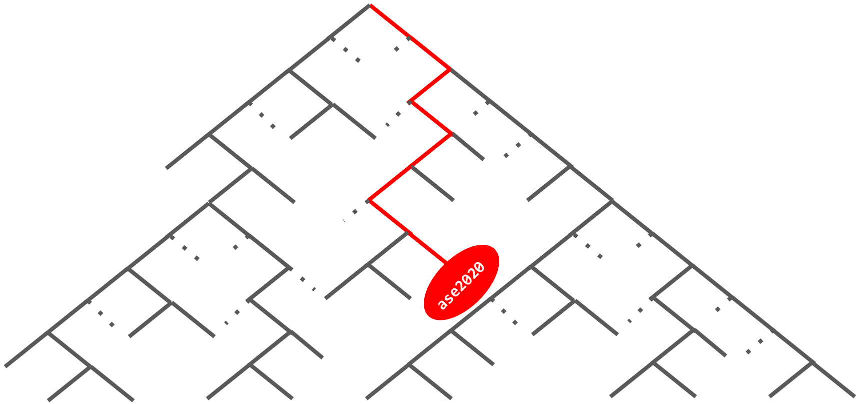


Vanilla

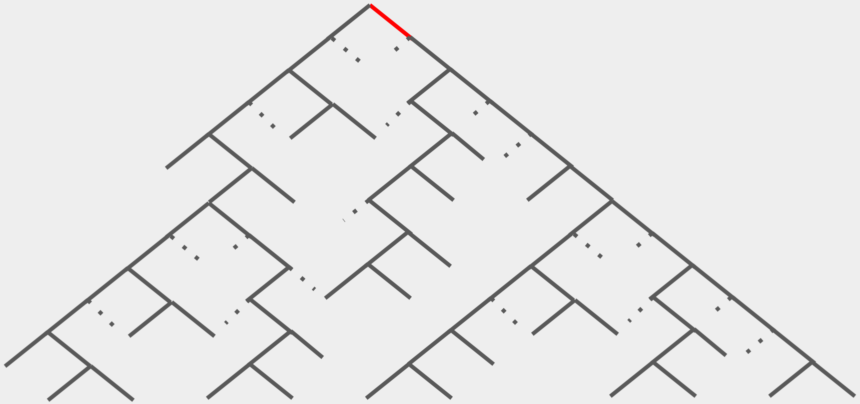


Solver queries: 0

Pending

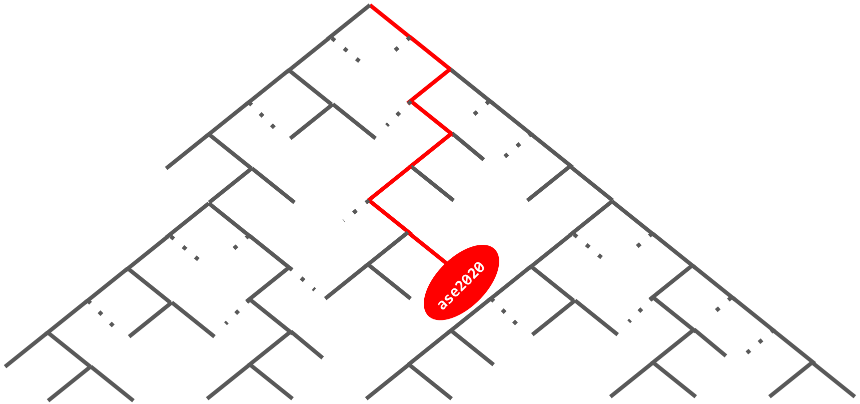


Vanilla

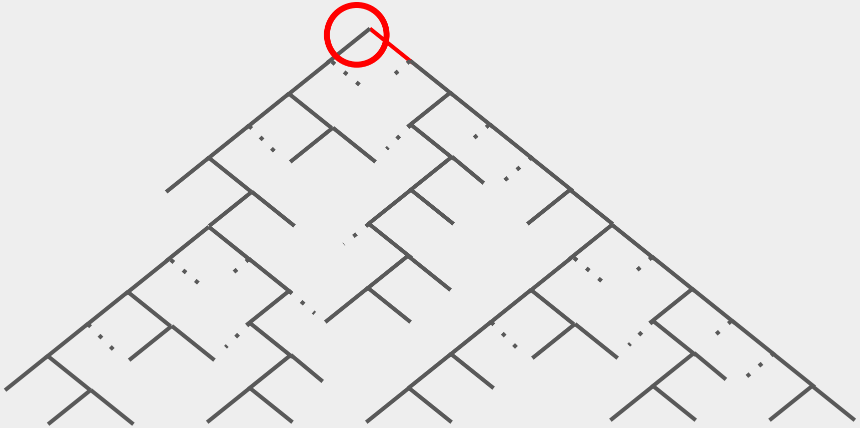


Solver queries: 0

Pending

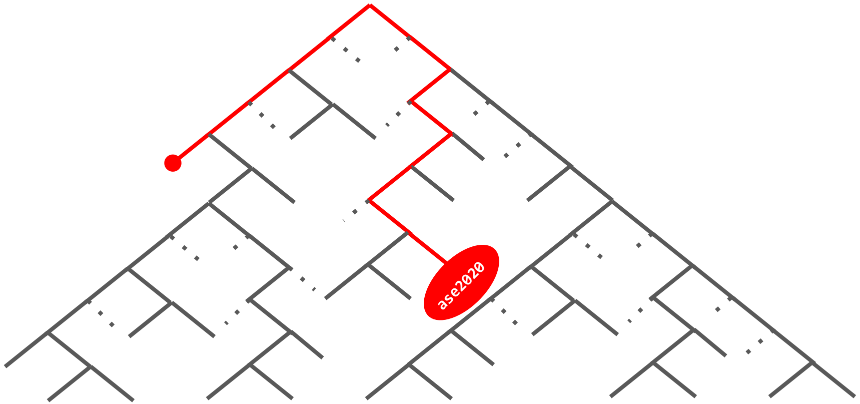


Vanilla

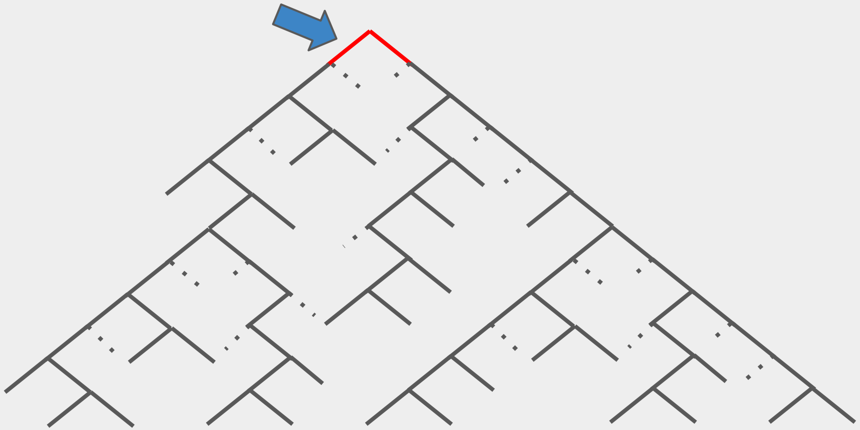


Solver queries: 1

Pending

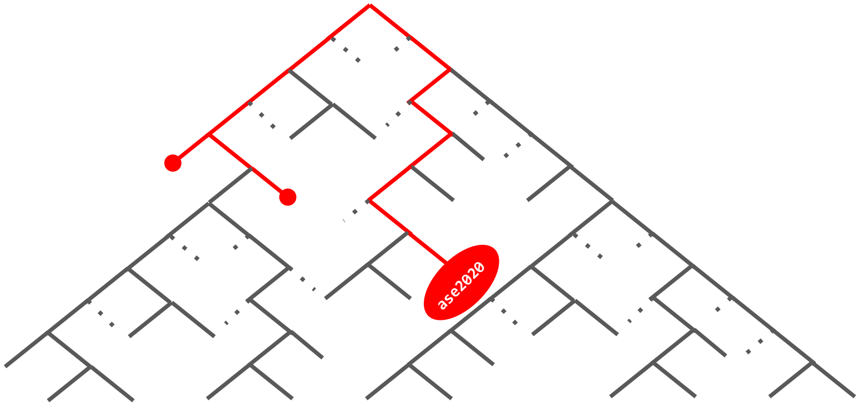


Vanilla

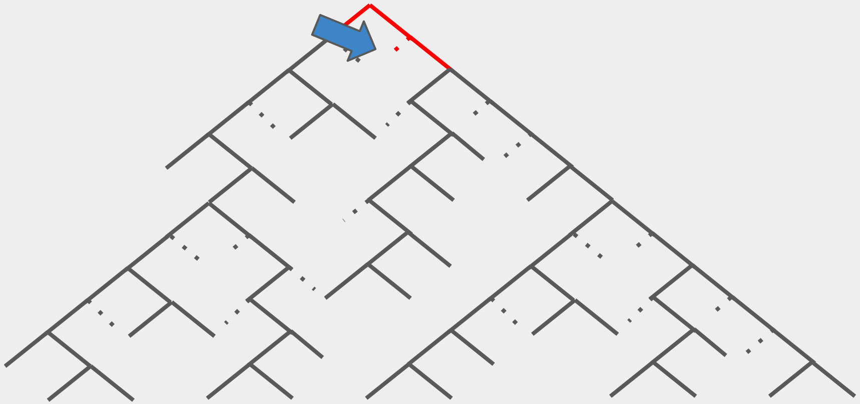


Solver queries: 2

Pending

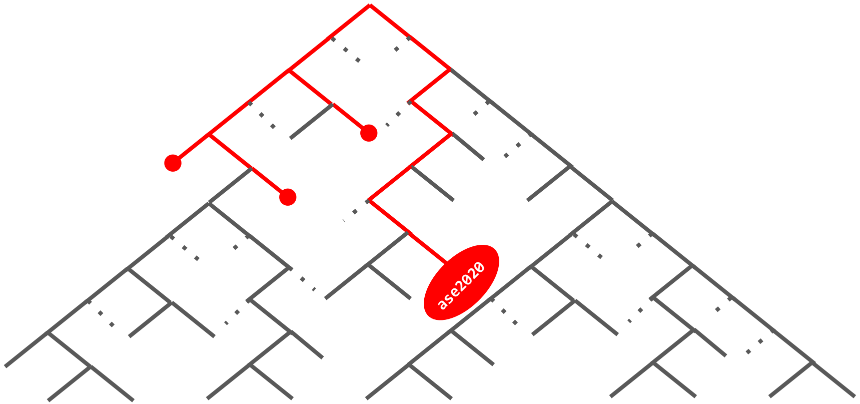


Vanilla

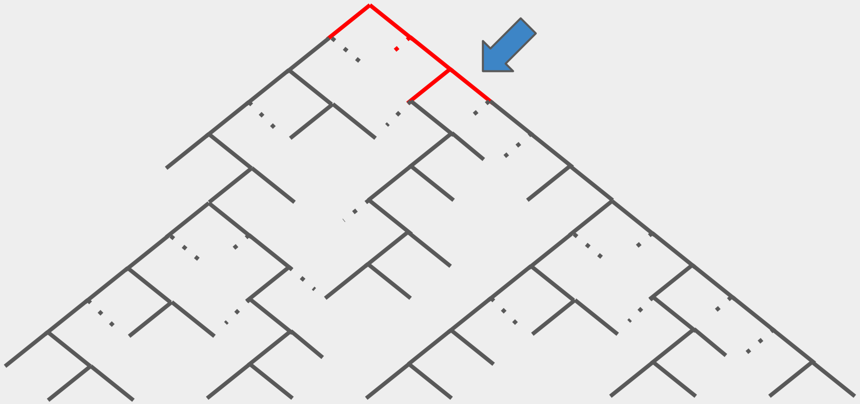


Solver queries: 3

Pending

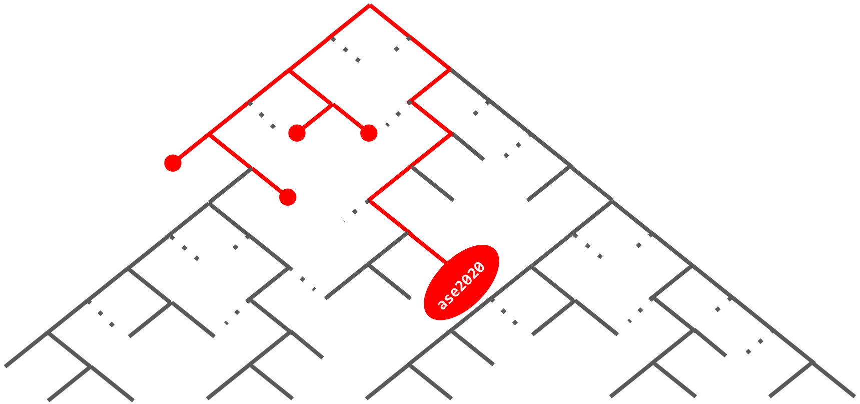


Vanilla

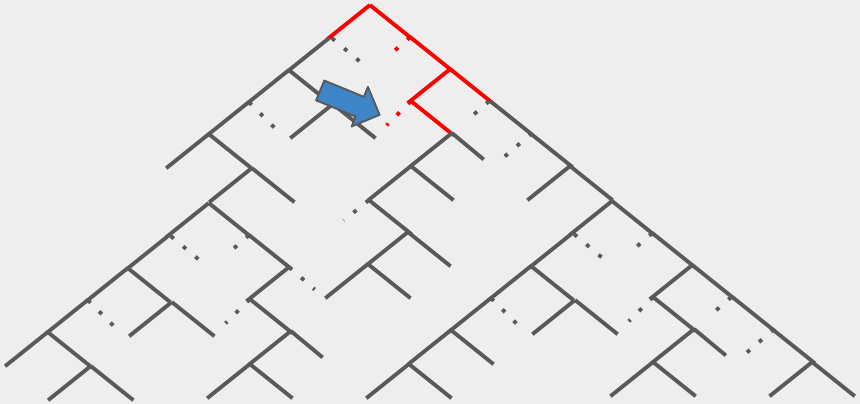


Solver queries: 4

Pending

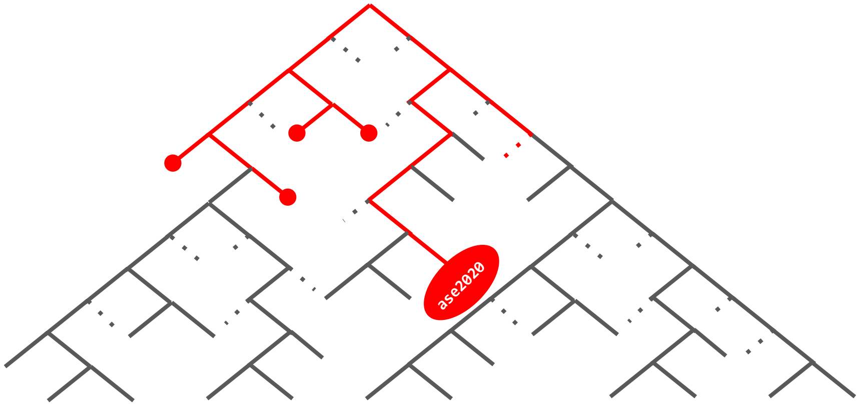


Vanilla

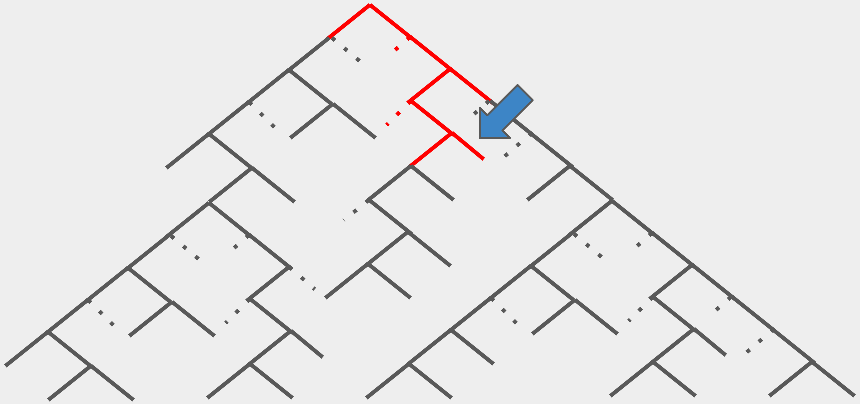


Solver queries: 5

Pending

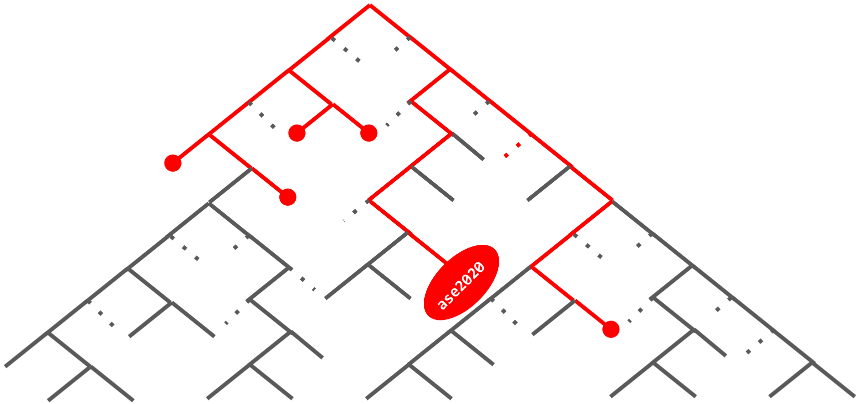


Vanilla

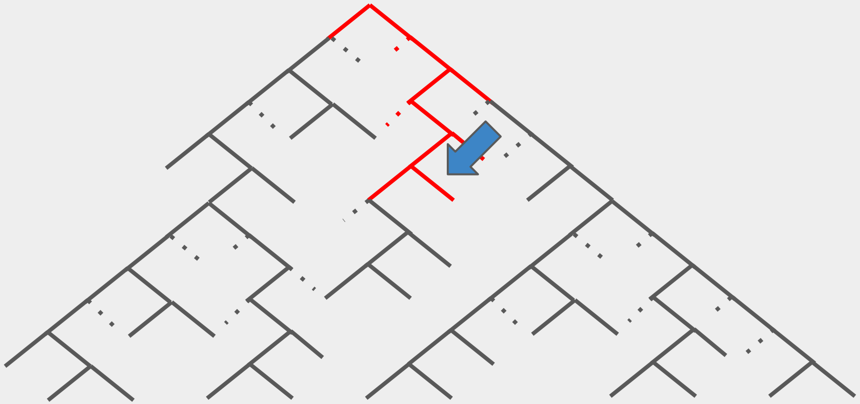


Solver queries: 6

Pending

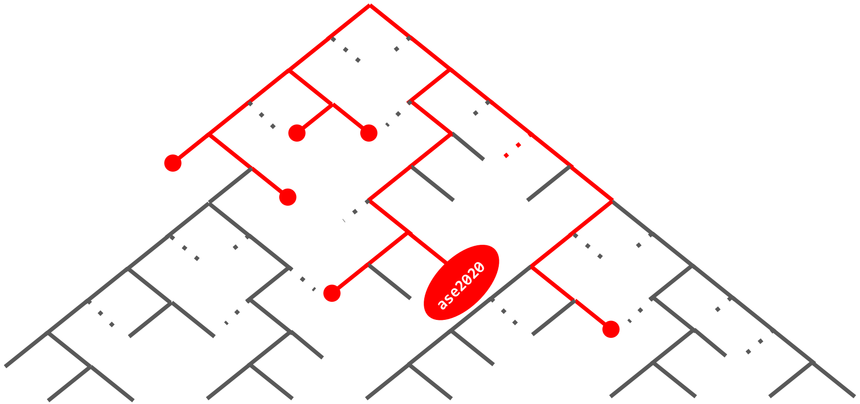


Vanilla

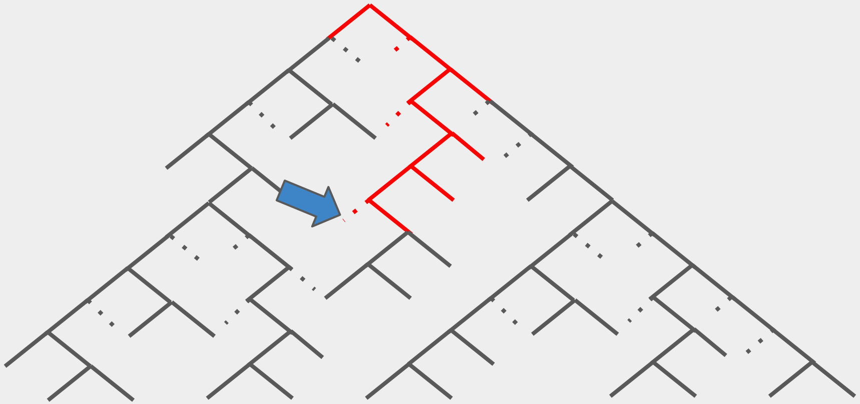


Solver queries: 7

Pending

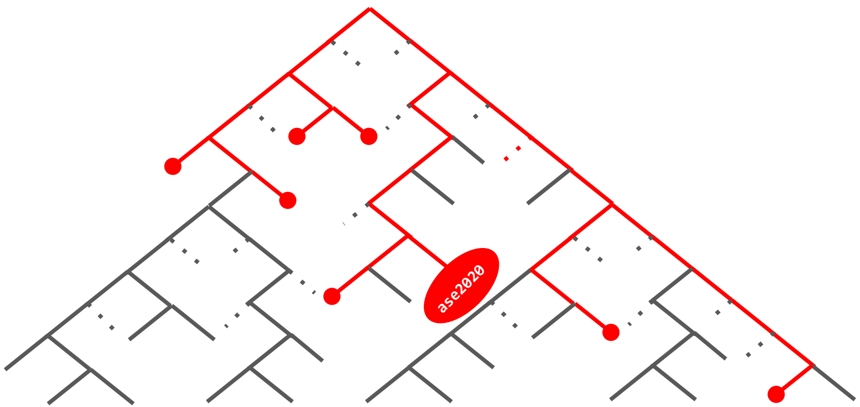


Vanilla

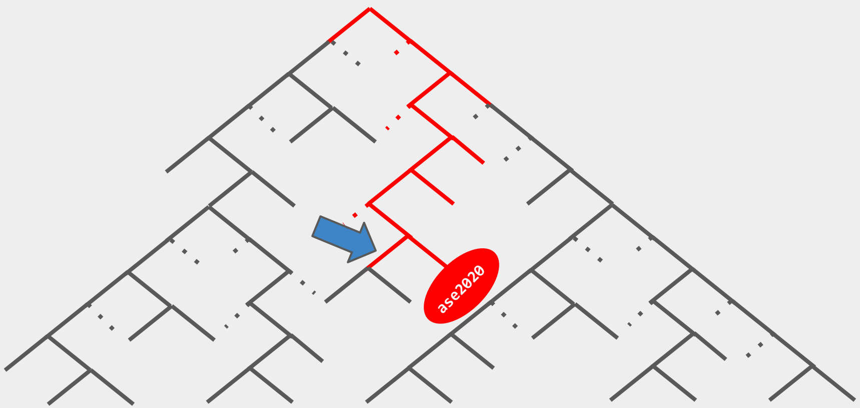


Solver queries: 8

Pending



Vanilla



Why pending constraints?

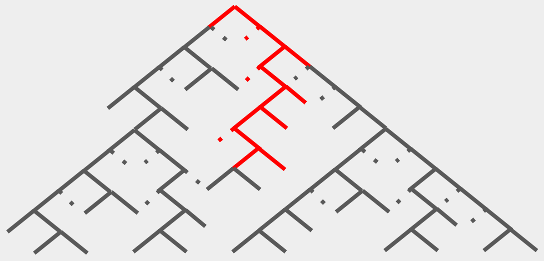
More efficient use of solver solutions

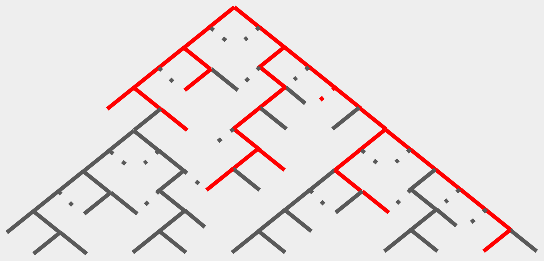
- explore **more instructions per query**
- spend **less time solving infeasible queries**

Allows deeper search tree exploration

Empowering search heuristics

- control over constraint solving
- ZESTI





Why pending constraints?

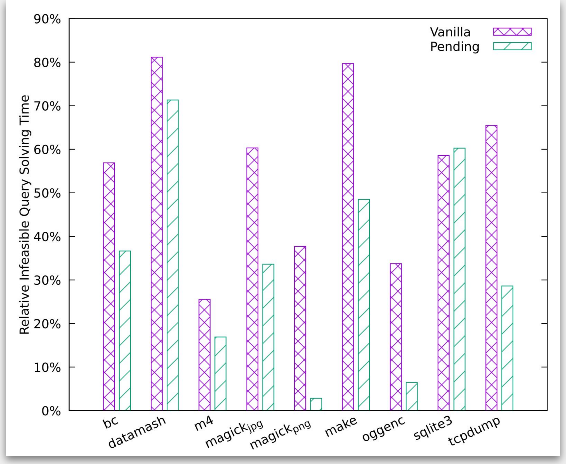
More efficient use of solver solutions

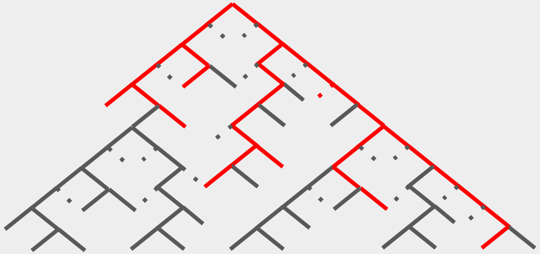
- explore **more instructions per query**
- spend **less time solving infeasible queries**

Allows deeper search tree exploration

Empowering search heuristics

- control over constraint solving





Evaluation

8 real world applications

Hard targets for symbolic execution

2hr runs, 3 searchers, 3 repetitions

24h SQLite study

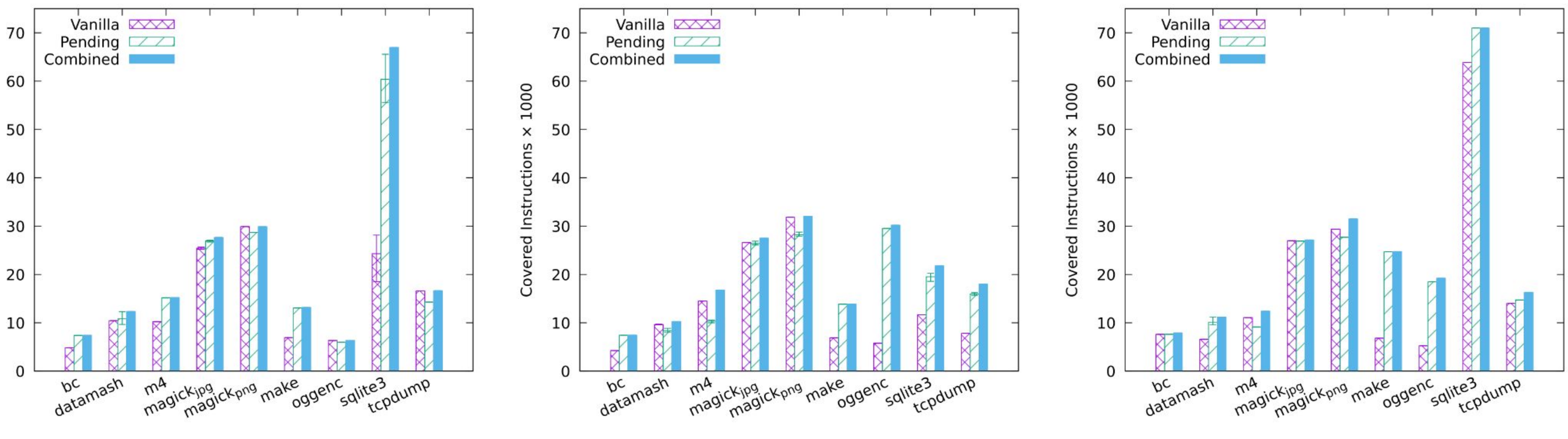
bc

make

datamash

m4

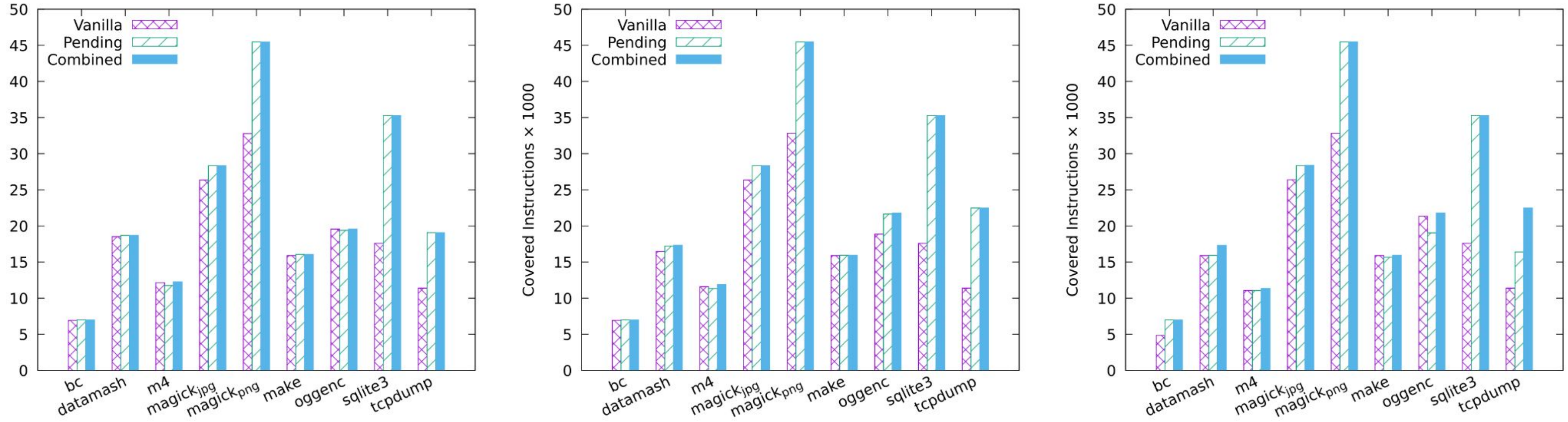
Instruction coverage - non-seeded



(a) Random Path (b) Depth-Biased (c) DFS

35%, 34% resp. 24% more instructions across benchmarks

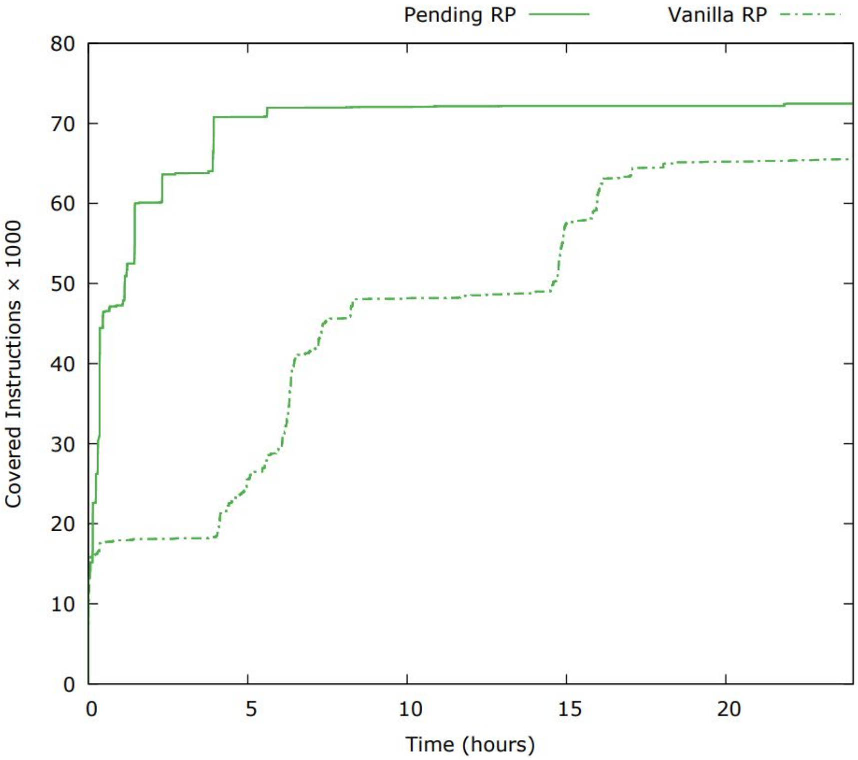
Instruction coverage - seeded



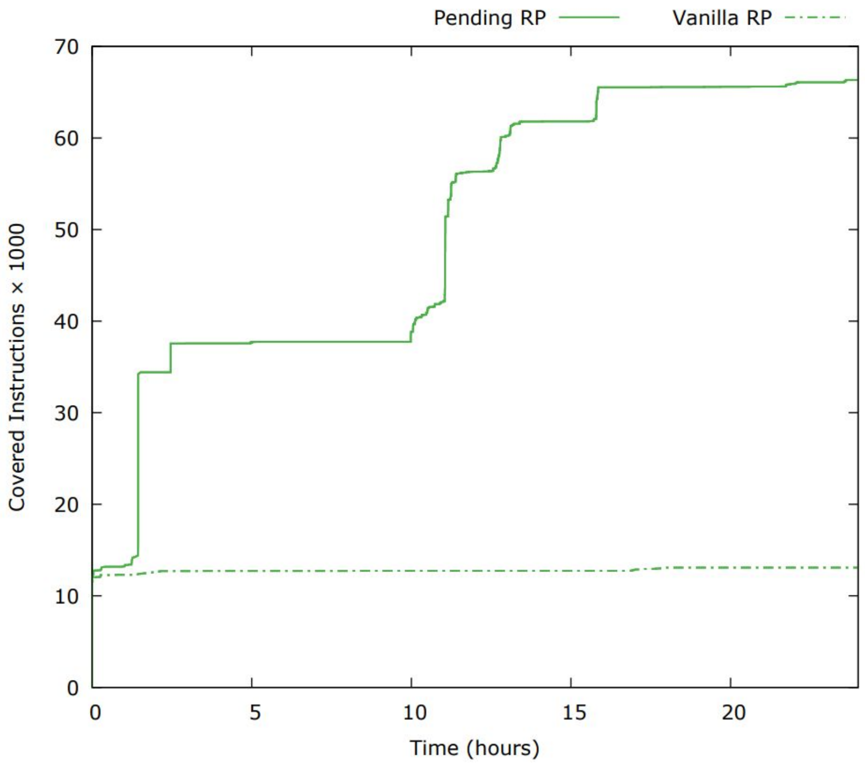
(a) Random Path (b) Depth-Biased (c) DFS

25%, 30% resp. 23% more instructions across benchmarks

SQLite3: 24 hour run - non-seeded (random path)

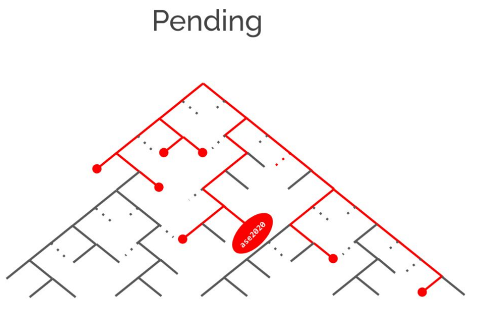


SQLite3: 24 hour run - seeded (random-path)



Pending constraints

- aggressively follow feasible paths and **explore more instructions per query**
- **reduce the constraint solving time**
- could **improve the coverage** for 8 hard programs



Zesti-Reimplementation

Explores sensitive instructions around seed.

Found **2 new bugs**.