《Towards General Loop Invariant Generation: A Benchmark of Programs with Memory Manipulation》

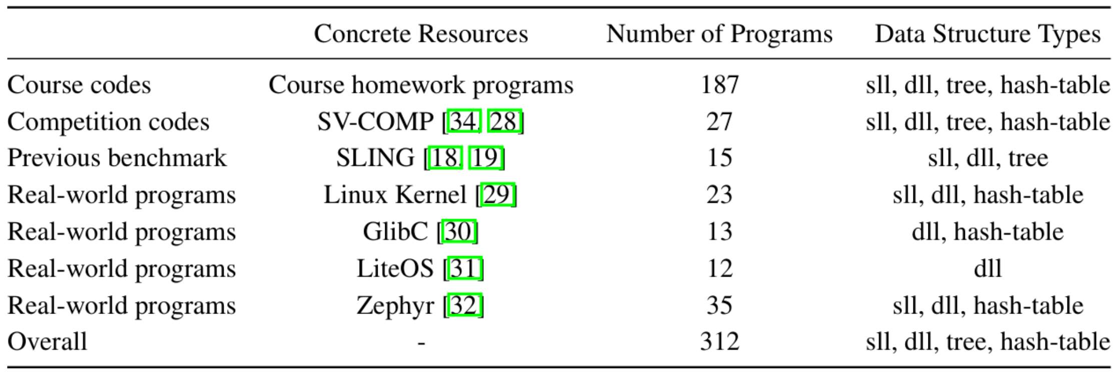
——Shanghai Jiao Tong University. *NeurIPS' 24* Track on Datasets and Benchmarks.

Main contribution:

The core content of this article is a survey about generating loop invariant in program verification, specially those programs **involve complicated data structure and memory manipulation**. This article comes up with **a new baseline benchmark** called LIG-MM and **a new framework** combines LLM with symbolic execution, aiming at strengthening the ability to verify program automatically.

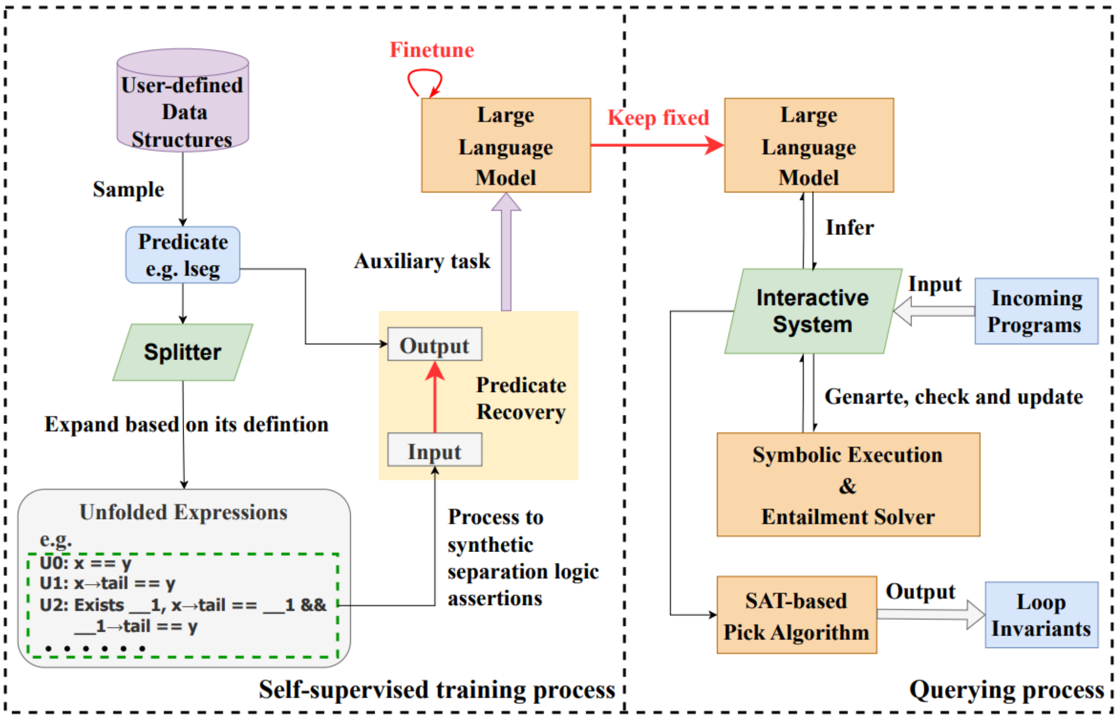
Method：

- **benchmark:** collecting **312 programs** from college assignments, SV-COMP, previous baseline and real-world software system like linux kernel. All program involves complicated data structure and memory manipulation.



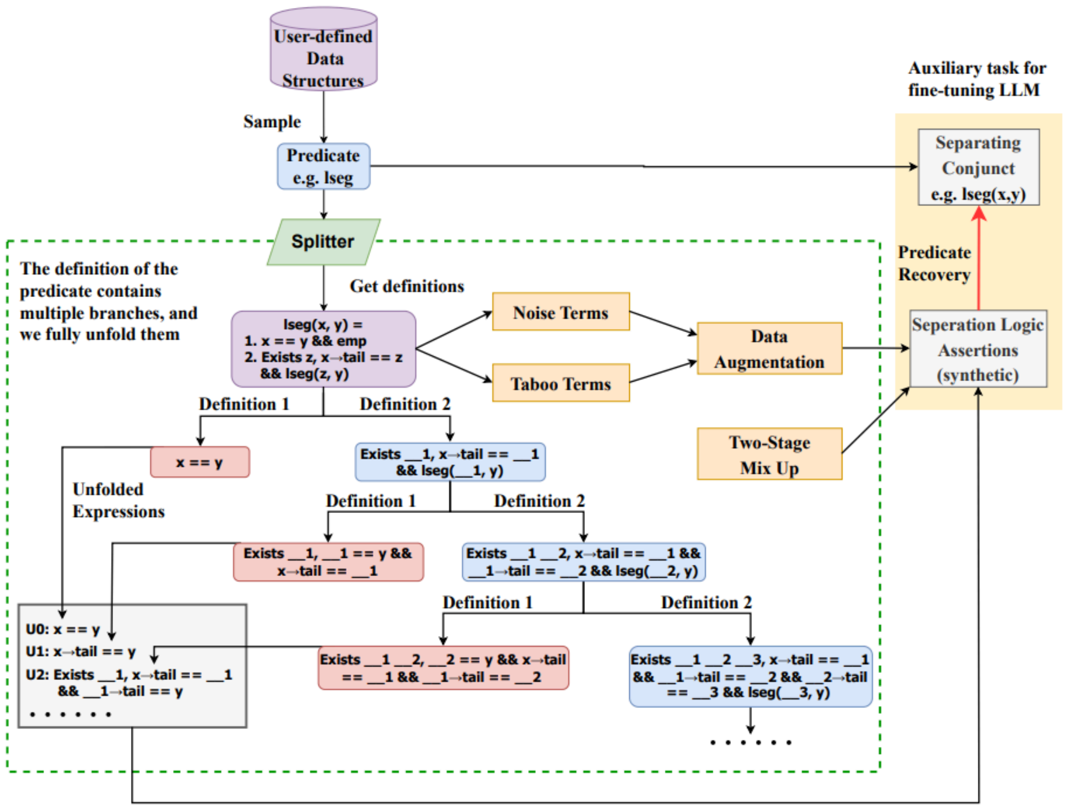
	Concrete Resources	Number of Programs	Data Structure Types
Course codes	Course homework programs	187	sll, dll, tree, hash-table
Competition codes	SV-COMP [34, 28]	27	sll, dll, tree, hash-table
Previous benchmark	SLING [18, 19]	15	sll, dll, tree
Real-world programs	Linux Kernel [29]	23	sll, dll, hash-table
Real-world programs	GlibC [30]	13	dll, hash-table
Real-world programs	LiteOS [31]	12	dll
Real-world programs	Zephyr [32]	35	sll, dll, hash-table
Overall	-	312	sll, dll, tree, hash-table

- **framework:**



Two stages:

1. offline training: aiming at **creating enough labeled data** for training by designing an auxiliary task.



User-defined Data Structures
Sample
Predicate e.g. lseg
Splitter
Auxiliary task for fine-tuning LLM
Separating Conjunct e.g. lseg(x,y)
Predicate Recovery
Seperation Logic Assertions (synthetic)
The definition of the predicate contains multiple branches, and we fully unfold them
Get definitions
lseg(x, y) = 1. x == y && emp 2. Exists z, x→tail == z && lseg(z, y)
Noise Terms
Taboo Terms
Data Augmentation
Two-Stage Mix Up
Definition 1
Definition 2
x == y
Exists __1, x→tail == __1 && lseg(__1, y)
Unfolded Expressions
Exists __1, __1 == y && x→tail == __1
Exists __1 __2, x→tail == __1 && __1→tail == __2 && lseg(__2, y)
U0: x == y
U1: x→tail == y
U2: Exists __1, x→tail == __1 && __1→tail == y
. . . . . .
Exists __1 __2, __2 == y && x→tail == __1 && __1→tail == __2
Exists __1 __2 __3, x→tail == __1 && __1→tail == __2 && __2→tail == __3 && lseg(__3, y)
. . . . . .

- input: `x->tail == z && lseg(z, y) {&& listrep(y) && y->tail == t && listrep(t)}`
- output: `lseg(x, y)`

2. online query:

input: `lseg(x, y)`

output: `loop invariant x == y -> emp ||`

SAT-based Pick Algorithm：

假设我们有以下候选断言集合：

$$\{A_1 : x > 0, A_2 : x \geq 0, A_3 : x < 0, A_4 : x \leq 0\}$$

目标是选择一个最小的断言集合，使得每个断言都能被覆盖。

1. **构造覆盖性约束**：
   - A_1 被 A_2 蕴含：(x_2)
   - A_2 被 A_2 蕴含：(x_2)
   - A_3 被 A_4 蕴含：(x_4)
   - A_4 被 A_4 蕴含：(x_4)
2. **构造最小性约束**：
   - 最小化 $x_1 + x_2 + x_3 + x_4$
3. **求解 SAT 问题**：
   - 使用 SAT 求解器求解上述约束，得到一个满足条件的布尔变量赋值。

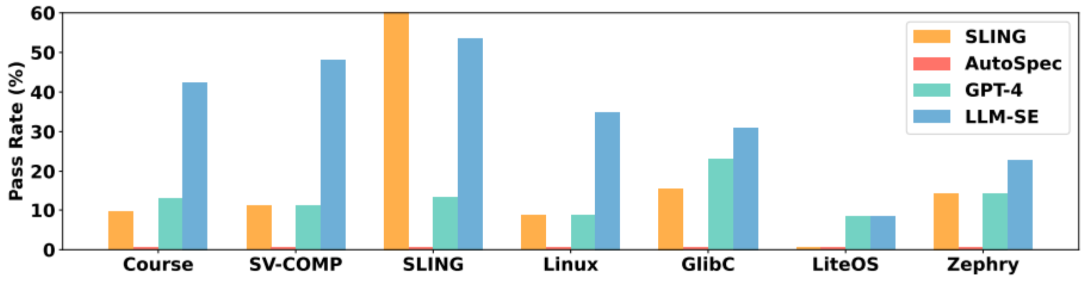
假设求解器返回 $x_2 = \text{true}$ 和 $x_4 = \text{true}$，则选择的断言集合为 $\{A_2, A_4\}$。

Experiment:

- **Set up:** Hugging face —— 350M CodeGen
- **Result:**

Table 2: Experimental results on our LIG-MM benchmark, 312 programs in total. Pass rates are reported as Pass Rate @ N, where N is the number of attempts to generate loop invariants.

Method	SLING [18]	AutoSpec [27]	GPT-4 [40]	LLM-SE (ours)
Pass Rate @ 1	12.82%	0.00%	12.50%	**38.78%**
Pass Rate @ 8	21.79%	0.00%	17.68%	**42.95%**



《Enhancing Automated Loop Invariant Generation for Complex Programs with Large Language Models》

——Shanghai Jiao Tong University.

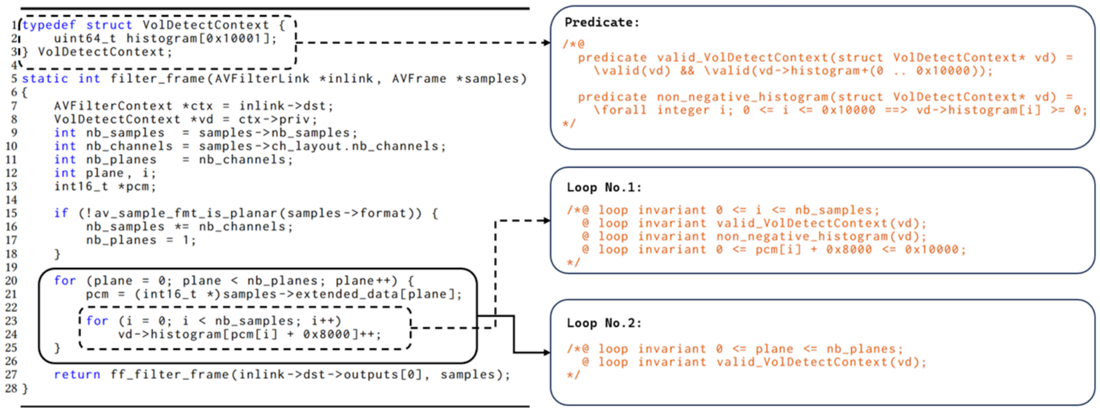
Main contribution:

Dataset + Workflow

Method:

- Dataset:

  with data structure: LIG-MM; SV-COMP24; FFmpeg(audio and video processing library)



```c
typedef struct VolDetectContext {
    uint64_t histogram[0x10001];
} VolDetectContext;

static int filter_frame(AVFilterLink *inlink, AVFrame *samples)
{
    AVFilterContext *ctx = inlink->dst;
    VolDetectContext *vd = ctx->priv;
    int nb_samples  = samples->nb_samples;
    int nb_channels = samples->ch_layout.nb_channels;
    int nb_planes   = nb_channels;
    int plane, i;
    int16_t *pcm;

    if (!av_sample_fmt_is_planar(samples->format)) {
        nb_samples *= nb_channels;
        nb_planes = 1;
    }

    for (plane = 0; plane < nb_planes; plane++) {
        pcm = (int16_t *)samples->extended_data[plane];

        for (i = 0; i < nb_samples; i++)
            vd->histogram[pcm[i] + 0x8000]++;
    }

    return ff_filter_frame(inlink->dst->outputs[0], samples);
}
```

```
Predicate:
/*@
  predicate valid_VolDetectContext(struct VolDetectContext* vd) =
    \valid(vd) && \valid(vd->histogram+(0 .. 0x10000));

  predicate non_negative_histogram(struct VolDetectContext* vd) =
    \forall integer i; 0 <= i <= 0x10000 ==> vd->histogram[i] >= 0;
*/
```

```
Loop No.1:
/*@ loop invariant 0 <= i <= nb_samples;
  @ loop invariant valid_VolDetectContext(vd);
  @ loop invariant non_negative_histogram(vd);
  @ loop invariant 0 <= pcm[i] + 0x8000 <= 0x10000;
*/
```

```
Loop No.2:
/*@ loop invariant 0 <= plane <= nb_planes;
  @ loop invariant valid_VolDetectContext(vd);
*/
```

Figure 3: An illustrative example of C code from FFmpeg

  w/o data structure: AutoSpec (Frama-c; Sy-GuS, OOPSLA-13, SV-COMP22)

- Workflow:

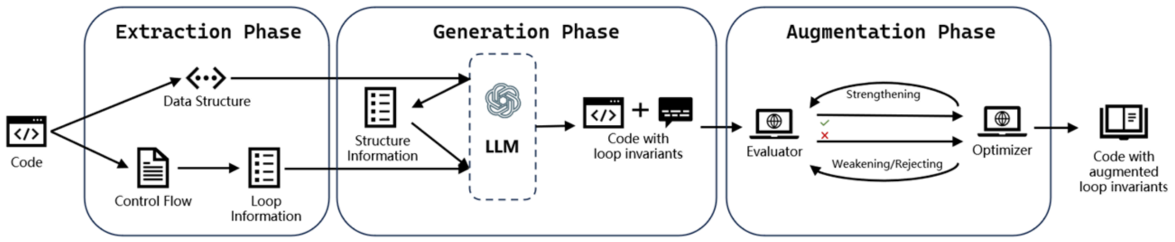


Figure 1: the Workflow of ACInv

- Extraction Phase: Mainly focus on **Data Structure Info, Loop Info, Key variant Info**, using static analysis techniques.

  Data Structure Info: Scan the AST of the program, and for each data structure, extract its member variant as well as its type.

  Loop Info: Iterate through the CFG of the program, and for each loop, extract its loop condition and loop body.

Key variant Info: Extract key invariant from loop body, including those variants been assigned, parameters, nested-loop...

Packaging Infos above as Structure Information (described as a string) pass on to Generation Phase.

- Generation Phase:

  For Data Structure: there are some predicate templates of different data structure predefined by human.

```
/*@
inductive is_linked_list{L}
(struct Node *p1, struct Node *p2) {
    case empty{L}: \forall struct Node *p1, *p2;
        (p1 == p2) ==> is_linked_list(p1, p2);
    case non_empty{L}: \forall struct Node *p1, *p2;
        (
            \valid(p1) && is_linked_list(p1->next, p2)
        ) ==> is_linked_list(p1, p2);
}
*/
```

  For Loop invariant Generation: the prompt composed of loop description, variant info, ds info, pre loop invariant.

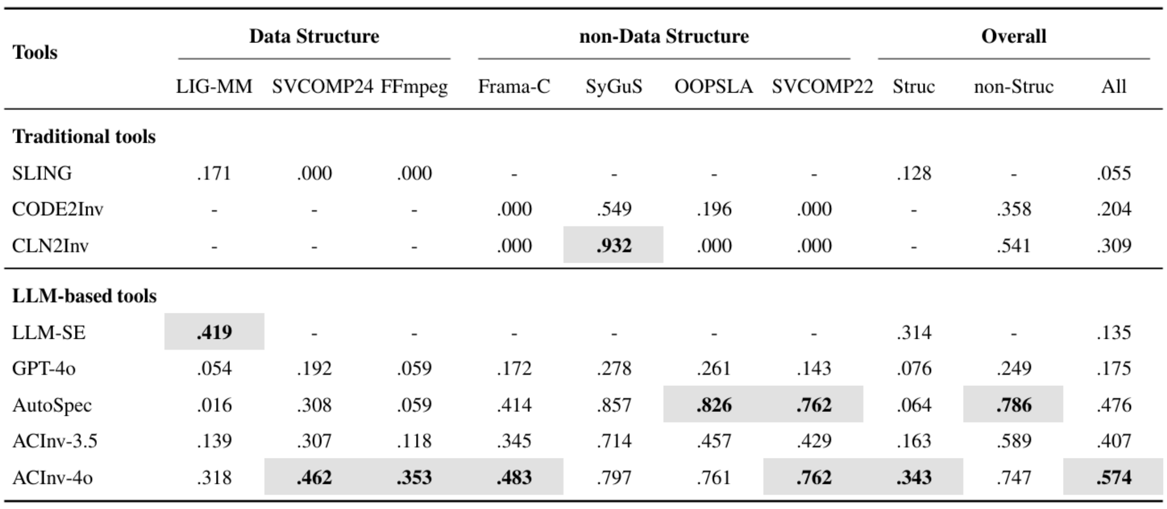
- Augmentation Phase:

  Evaluator: assess the correctness of loop invariants.

  Optimizer(LLM): strengthening or weakening current loop invariant according to the evaluator's result.

Experiment:

Table 1: Comparison of Performance on Multiple Datasets



Tools	Data Structure			non-Data Structure				Overall		
	LIG-MM	SVCOMP24	FFmpeg	Frama-C	SyGuS	OOPSLA	SVCOMP22	Struc	non-Struc	All
Traditional tools										
SLING	.171	.000	.000	-	-	-	-	.128	-	.055
CODE2Inv	-	-	-	.000	.549	.196	.000	-	.358	.204
CLN2Inv	-	-	-	.000	**.932**	.000	.000	-	.541	.309
LLM-based tools										
LLM-SE	**.419**	-	-	-	-	-	-	.314	-	.135
GPT-4o	.054	.192	.059	.172	.278	.261	.143	.076	.249	.175
AutoSpec	.016	.308	.059	.414	.857	**.826**	**.762**	.064	**.786**	.476
ACInv-3.5	.139	.307	.118	.345	.714	.457	.429	.163	.589	.407
ACInv-4o	.318	**.462**	**.353**	**.483**	.797	.761	**.762**	**.343**	.747	**.574**

4.3 Ablation Study

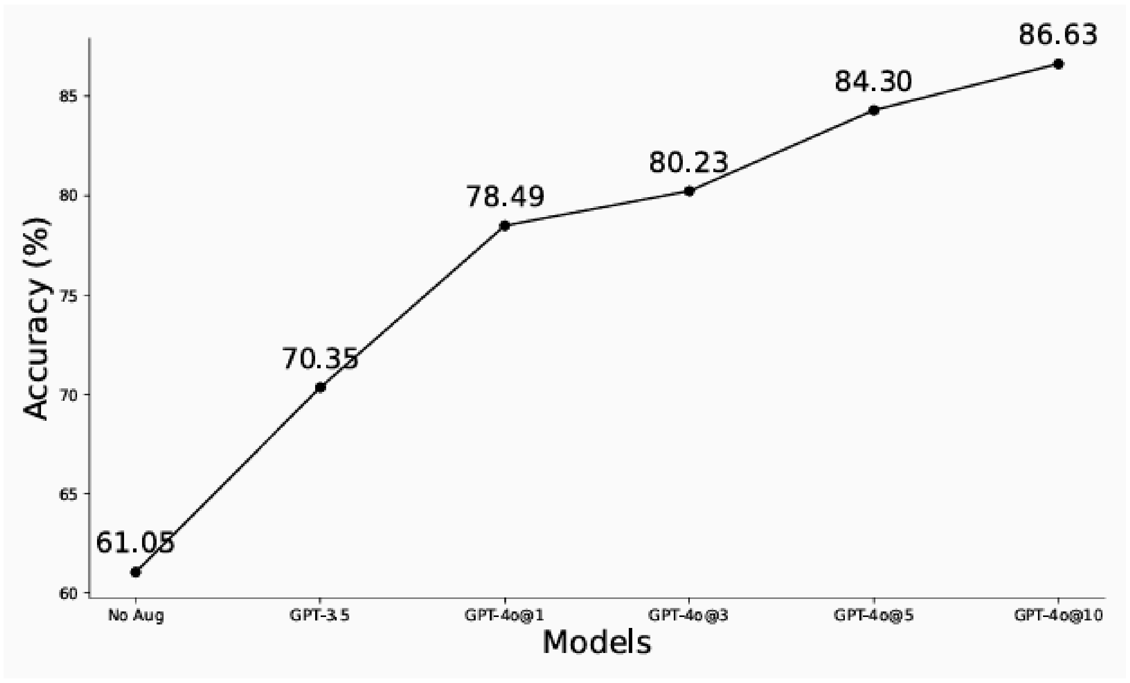


Figure 4: Performance of Augmentation of ACInv

《Code Repair with LLMs gives an Exploration-Exploitation Tradeoff》

——Cornell University, Shanghai Jiao Tong University. NeurIPS' 24

Main Idea: Balance the exploration and exploitation in code repair task's iteration.

The article proposes the **REx algorithm** to address the exploration-exploitation trade-off by modeling the improvement process as a multi-armed bandit (MAB) problem. The methodology is as follows:

- **The multi-armed bandit problem**: Each program is considered as an “arm”, and **“pulling” an arm corresponds to improving a program.** After each improvement, **a reward (0 or 1) is given depending on whether the newly generated program satisfies the specification.**
- **Thompson Sampling:** A Thompson Sampling strategy is used to select the next program to improve. The strategy selects the next program based on the success rate of each program's improvement (modeled by the Beta distribution), taking into account the number of times each program has been improved.

- **Heuristic function:** each program is evaluated for goodness using a heuristic function h(ρ) based on the proportion of programs that satisfy the specification.

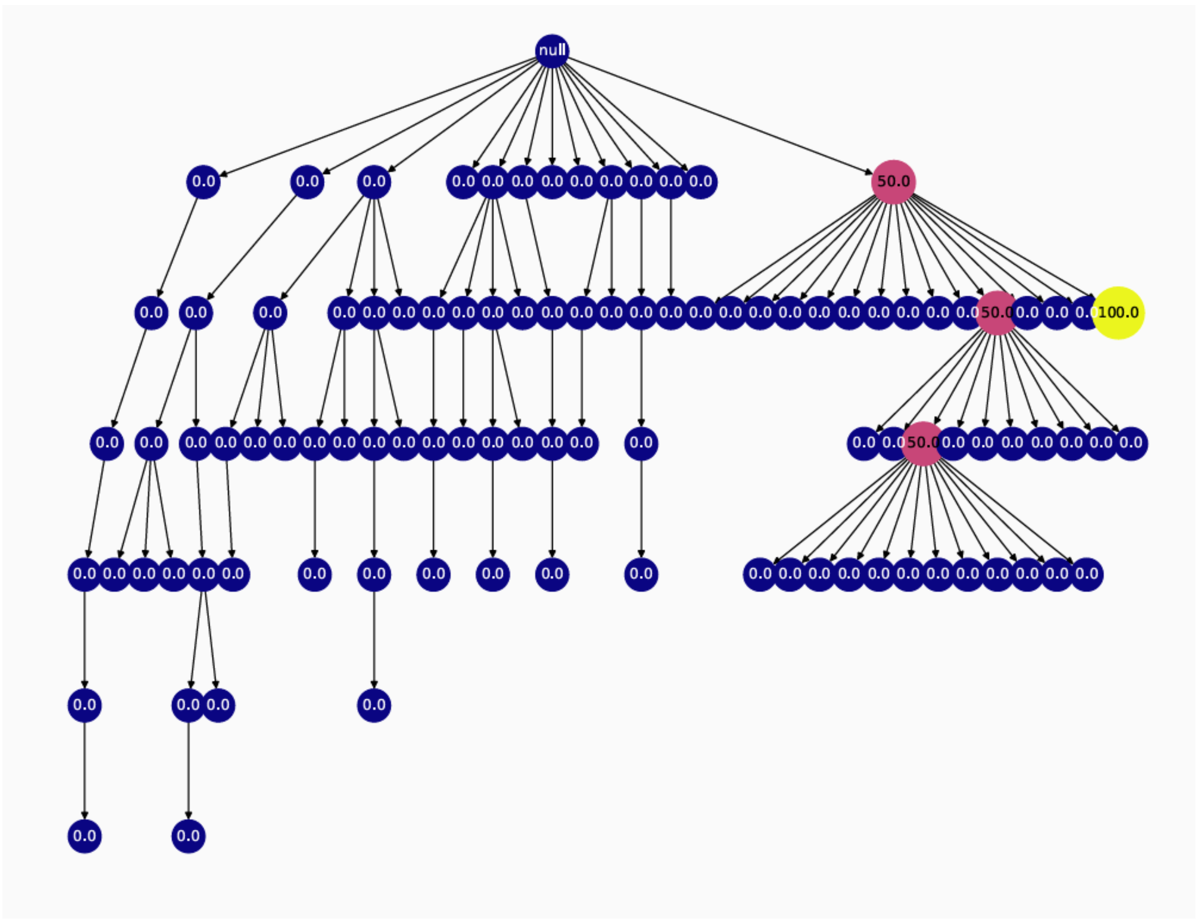


Figure 16: Search Tree

Dataset:

Index	Program	Loop #	InvType	G-CLN [21]	GSpacer [26]	Loopy-GPT4 [41]	**REx**
1	cohencu	1	NL	✓	-	-	-
2	cohendiv	1	NL	✓	-	✓	✓
3		2	NL	-	-	✓	✓
4	dijkstra	1	Linear	✓	✓	-	✓
5		2	NL	✓	-	-	-
6	divbin	1	Linear	✓	✓	-	✓
7		2	NL	✓	-	✓	✓
8	egcd	1	NL	✓	-	-	✓
9	egcd2	1	NL	-	-	-	✓
10		2	NL	-	-	✓	✓
11	egcd3	1	NL	-	-	-	✓
12		2	NL	-	-	✓	✓
13		3	NL	-	-	-	✓
14	fermat1	1	NL	✓	-	-	✓
15		2	Linear	-	-	-	✓
16		3	Linear	-	-	✓	✓
17	fermat2	1	NL	✓	-	-	✓
18	freire1	1	NL	-	-	-	-
19	freire2	1	NL	✓	-	-	✓
20	geo1	1	NL	✓	-	-	-
21	geo2	1	NL	✓	-	-	-
22	geo3	1	NL	✓	✓	-	-
23	hard	1	NL	✓	-	✓	✓
24		2	NL	✓	-	✓	✓
25	knuth	1	NL	-	-	-	-
26	lcm1	1	NL	-	-	-	✓
27		2	NL	✓	✓	-	✓
28		3	NL	✓	✓	-	✓
29	lcm2	1	NL	-	-	-	-
30	mannadiv	1	NL	✓	-	✓	✓
31	prod4br	1	NL	-	-	-	✓
32	prodbin	1	NL	✓	-	✓	✓
33	ps2	1	NL	✓	-	✓	✓
34	ps3	1	NL	✓	-	-	✓
35	ps4	1	NL	✓	-	-	✓
36	ps5	1	NL	-	-	-	-
37	ps6	1	NL	✓	-	-	-
38	sqrt	1	NL	✓	-	-	✓
	Total correct			24	5	11	28

Table 2: Benchmark evaluation on the NLA programs. For programs with multiple loops, each loop is labeled with a number, top to down, from outer to inner.