Efficiently Finding Higher-Order Mutants

Chu-Pan Wong*
Carnegie Mellon University
USA

Jens Meinicke*
Carnegie Mellon University
USA

Leo Chen*
Carnegie Mellon University
USA

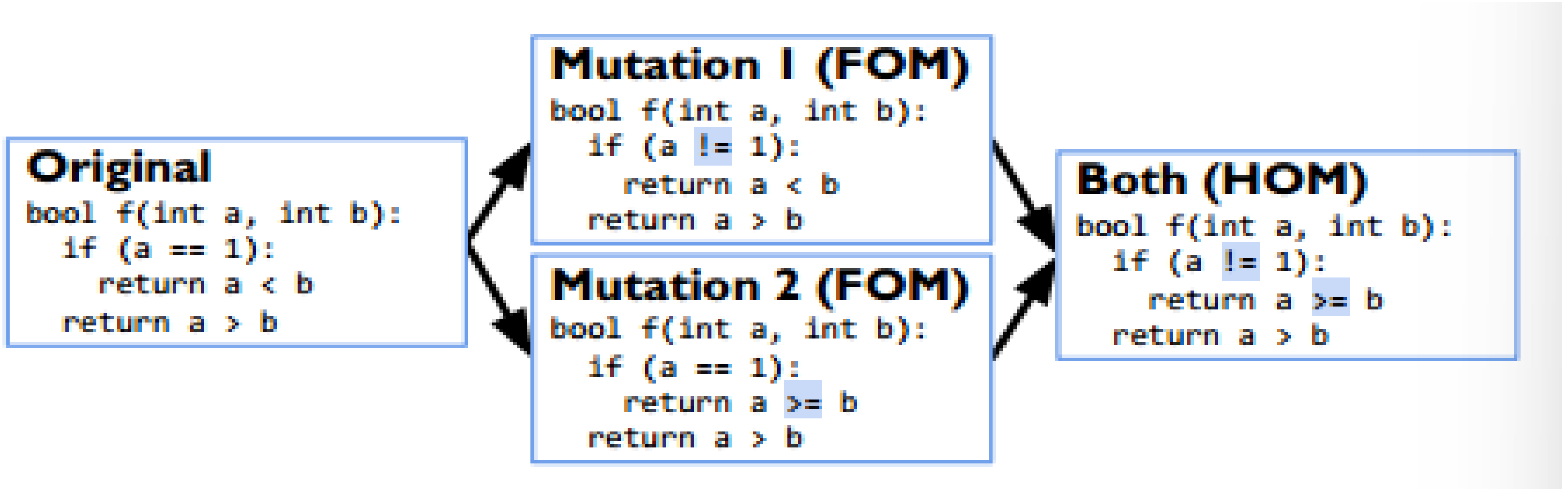
João P. Diniz
Federal University of Minas Gerais
Brazil

Christian Kästner
Carnegie Mellon University
USA

Eduardo Figueiredo
Federal University of Minas Gerais
Brazil

背景知识介绍

- Mutant:使用变异算子对源程序进行细微变动
- First-Order Mutant（FOM):只在一处进行变异
- High-Order Mutant（HOM):两个或多个一阶Mutant的组合



```
Original
bool f(int a, int b):
  if (a == 1):
    return a < b
  return a > b
```

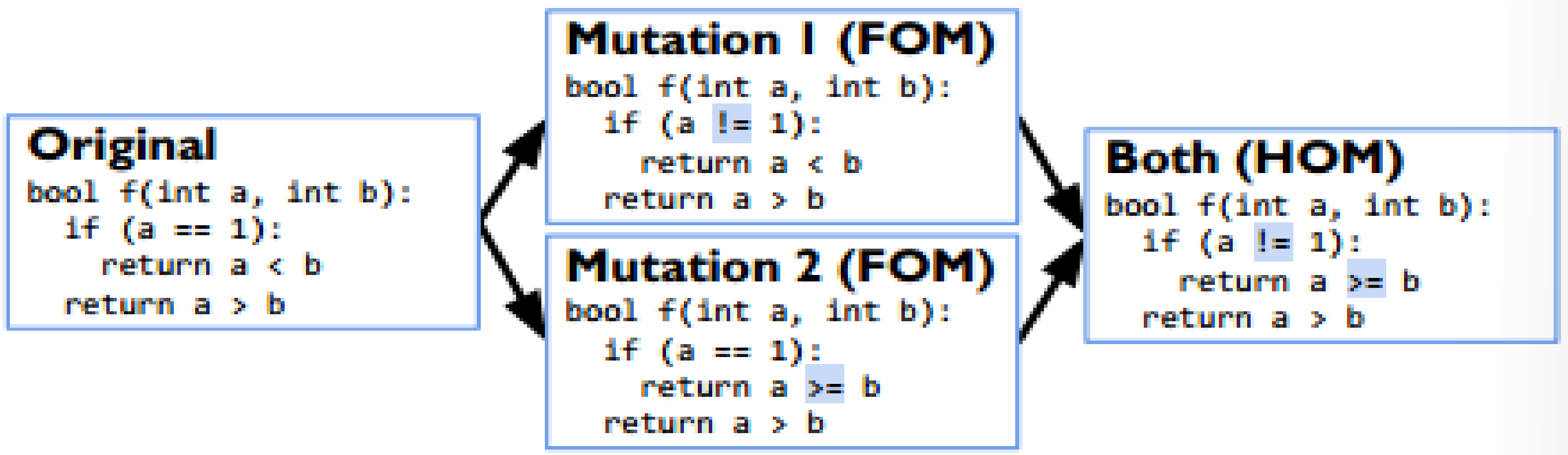
```
Mutation 1 (FOM)
bool f(int a, int b):
  if (a != 1):
    return a < b
  return a > b
```

```
Mutation 2 (FOM)
bool f(int a, int b):
  if (a == 1):
    return a >= b
  return a > b
```

```
Both (HOM)
bool f(int a, int b):
  if (a != 1):
    return a >= b
  return a > b
```

- 如果一个mutant可以被一个测试用例T所识别出来，那么就可以说这个mutant被T杀死

例如（original: f(1,2)->return(a<b),mutant1:f(1,2)->return(a>b)）


Original
```
bool f(int a, int b):
  if (a == 1):
    return a < b
  return a > b
```
Mutation 1 (FOM)
```
bool f(int a, int b):
  if (a != 1):
    return a < b
  return a > b
```
Mutation 2 (FOM)
```
bool f(int a, int b):
  if (a == 1):
    return a >= b
  return a > b
```
Both (HOM)
```
bool f(int a, int b):
  if (a != 1):
    return a >= b
  return a > b
```

- **由于耦合性，普通HOM容易被杀死**

  O:f(0,3)-> return a>b,mutant1:f(0,3)->return a<b, hom:f(0,3)->return a>=b;

  O:f(1,1)-> return a<b,mutant2:f(1,1)->return a>=b, hom:f(1,1)->return a>b;

SSHOM

普通的HOM很难代表真实世界的bug，于是有人提出了Strong Subming High-Order Mutant(SSHOM)，这是HOM的一个子集，更难被杀死。

- **T1->M1**
- **T2->M2**
- **T3->M3**
- **T4->HOM(M1,M2,M3组成)**

HOM是一个SSHOM当且紧当

$$T4 \subseteq T1 \cap T2 \cap T3$$

Strict-SSHOM:

$$T4 \subset T1 \cap T2 \cap T3$$

论文总体框架

- 1.首先提出一种方法：search_var，在中小型程序中寻找SSHOM
- 2.分析SSHOM的特征
- 3.基于分析的特征，提出另一种优先搜索策略：search_pri，可应用于大型程序系统
- 4.对比实验
- 其它方法
  - Search_gen：基于基因搜索的方法，是目前效果最好的。
  - Search_bf：暴力搜索。

Search_var

- Search_var使用变分执行作为一种黑盒技术来探索，它找出所有的FOM的组合已即杀死它们的测试用例。然后返回这些FOM组合的命题公式(HOM)和测试用例。
- 变分执行通过动态跟踪由突变引起的程序状态差异来运行程序。

从概念上讲，一次变分执行相当于以暴力搜索的方式来运行一阶突变的所有组合，但由于在运行时共享类似的执行，它通常要快得多。

为了识别SSHOM，编码三个标准。（ft:由t杀死的FOM组合的命题公式，由变分执行产生；Γ(m,t)：一阶变异m被t所杀死。

$$\bigvee_{t\in T} f_t$$

$$\bigwedge_{t\in T} (f_t \Rightarrow \bigwedge_{m\in M} (\neg m \vee \Gamma(m,t))$$

$$\bigvee_{t\in T} (\neg f_t \wedge \bigwedge_{m\in M} (\neg m \vee \Gamma(m,t)))$$

特征

- 超过90%的所有SSHOM和Strict-SSHOM由最多4个FOM组成。
- 当一个SSHOM由两个以上的一阶突变体组成时，这些一阶突变体的一个子集很可能也形成了一个SSHOM。（N+1)
- 许多SSHOM和Strict-SSHOM由被同一组测试用例杀死的一阶突变体组成(即组成SSHOM的FOM更容易被相同的测试用例杀死)
- 对于大多数SSHOM，所有组成的FOM都属于同一类，甚至通常在同一个方法中。这可能是因为接近的一阶突变具有更高的数据流或控制流相互作用的机会。

Search_pri

- Search_pri避免了变分执行的开销，它也是通过执行测试套件来判断候选的HOM是否是SSHOM, 基于先前确定的典型特征进行优先排序，然后以优先排序顺序来对候选者进行检验是否是SSHOM.
- 进行优先排序时考虑惩罚因子

$$penalty = \omega_1 \cdot order + \omega_2 \cdot testDiff - \omega_3 \cdot isN1$$

w1=1,w2=1,w3=15

testDiff:只能杀死组成候选者的FOM的子集的测试用例的数量。

考虑dist呢?

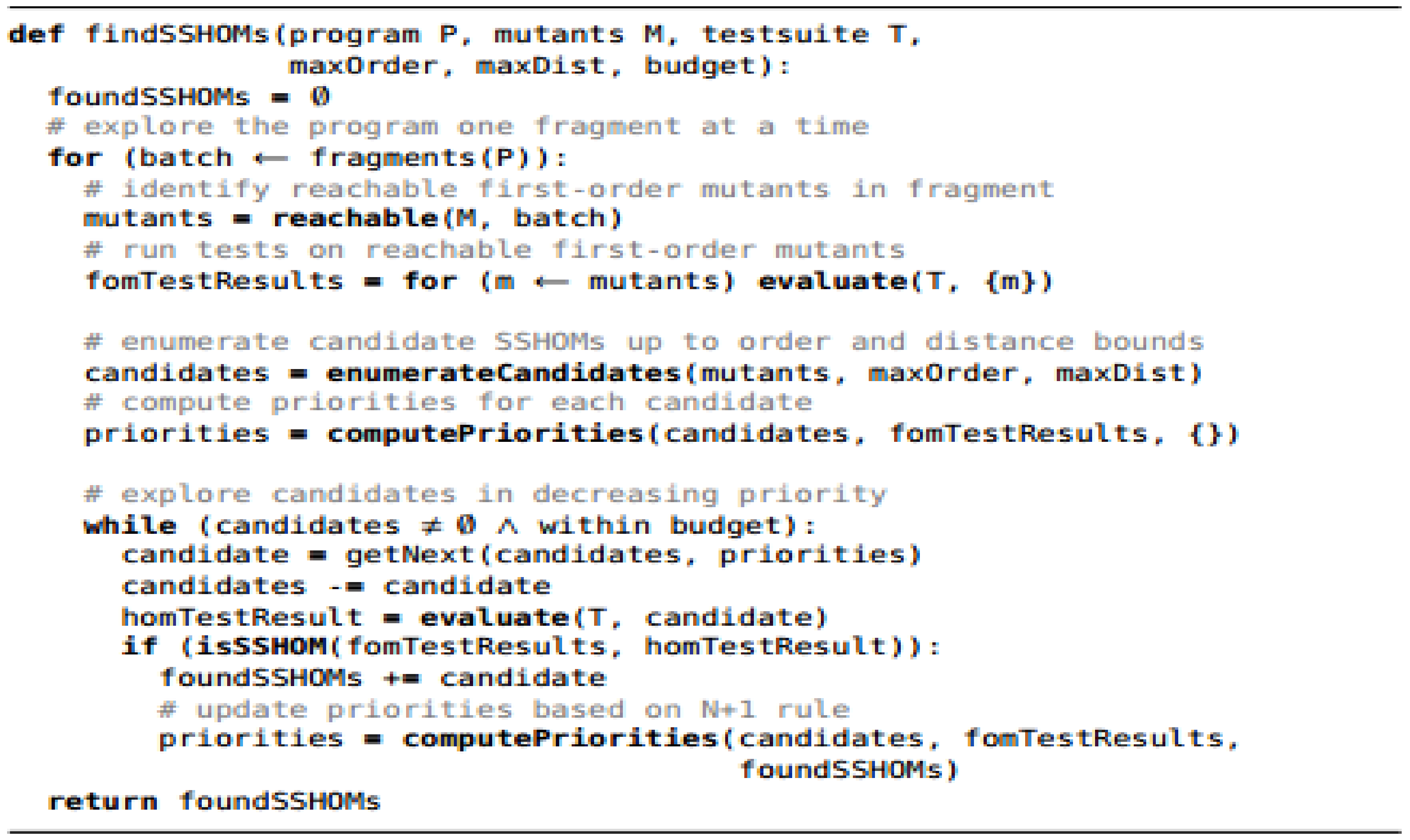
testDiff:只能杀死组成候选者的FOM的子集的测试用例的数量。

- **T1(t1,t2,t3,t4)->M1**
- **T2(t2,t3,t4)->M2**
- **T3(t3,t4)->M3**
- **T4(t4)->HOM(M1,M2,M3组成)**

$\mathbf{T4} \subseteq T1 \cap T2 \cap T3$

testDiff=2

Search_pri



```
def findSSHOMs(program P, mutants M, testsuite T,
               maxOrder, maxDist, budget):
  foundSSHOMs = ∅
  # explore the program one fragment at a time
  for (batch ⟵ fragments(P)):
    # identify reachable first-order mutants in fragment
    mutants = reachable(M, batch)
    # run tests on reachable first-order mutants
    fomTestResults = for (m ⟵ mutants) evaluate(T, {m})

    # enumerate candidate SSHOMs up to order and distance bounds
    candidates = enumerateCandidates(mutants, maxOrder, maxDist)
    # compute priorities for each candidate
    priorities = computePriorities(candidates, fomTestResults, {})

    # explore candidates in decreasing priority
    while (candidates ≠ ∅ ∧ within budget):
      candidate = getNext(candidates, priorities)
      candidates -= candidate
      homTestResult = evaluate(T, candidate)
      if (isSSHOM(fomTestResults, homTestResult)):
        foundSSHOMs += candidate
        # update priorities based on N+1 rule
        priorities = computePriorities(candidates, fomTestResults,
                                       foundSSHOMs)
  return foundSSHOMs
```

实验及评估

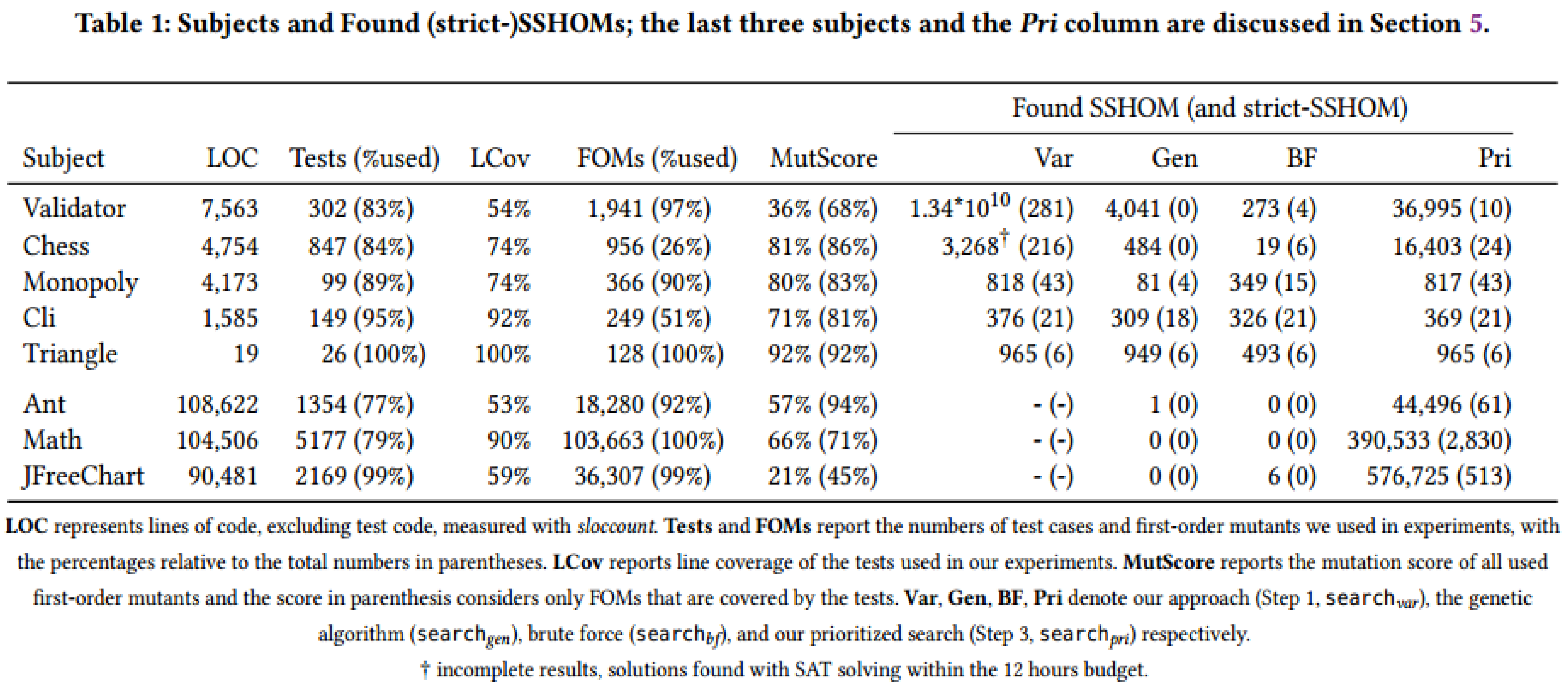


Table 1: Subjects and Found (strict-)SSHOMs; the last three subjects and the *Pri* column are discussed in Section 5.

						Found SSHOM (and strict-SSHOM)			
Subject	LOC	Tests (%used)	LCov	FOMs (%used)	MutScore	Var	Gen	BF	Pri
Validator	7,563	302 (83%)	54%	1,941 (97%)	36% (68%)	$1.34*10^{10}$ (281)	4,041 (0)	273 (4)	36,995 (10)
Chess	4,754	847 (84%)	74%	956 (26%)	81% (86%)	3,268† (216)	484 (0)	19 (6)	16,403 (24)
Monopoly	4,173	99 (89%)	74%	366 (90%)	80% (83%)	818 (43)	81 (4)	349 (15)	817 (43)
Cli	1,585	149 (95%)	92%	249 (51%)	71% (81%)	376 (21)	309 (18)	326 (21)	369 (21)
Triangle	19	26 (100%)	100%	128 (100%)	92% (92%)	965 (6)	949 (6)	493 (6)	965 (6)
Ant	108,622	1354 (77%)	53%	18,280 (92%)	57% (94%)	- (-)	1 (0)	0 (0)	44,496 (61)
Math	104,506	5177 (79%)	90%	103,663 (100%)	66% (71%)	- (-)	0 (0)	0 (0)	390,533 (2,830)
JFreeChart	90,481	2169 (99%)	59%	36,307 (99%)	21% (45%)	- (-)	0 (0)	6 (0)	576,725 (513)

LOC represents lines of code, excluding test code, measured with *sloccount*. **Tests** and **FOMs** report the numbers of test cases and first-order mutants we used in experiments, with the percentages relative to the total numbers in parentheses. **LCov** reports line coverage of the tests used in our experiments. **MutScore** reports the mutation score of all used first-order mutants and the score in parenthesis considers only FOMs that are covered by the tests. **Var**, **Gen**, **BF**, **Pri** denote our approach (Step 1, $\texttt{search}_{var}$), the genetic algorithm ($\texttt{search}_{gen}$), brute force ($\texttt{search}_{bf}$), and our prioritized search (Step 3, $\texttt{search}_{pri}$) respectively.

† incomplete results, solutions found with SAT solving within the 12 hours budget.

实验及评估

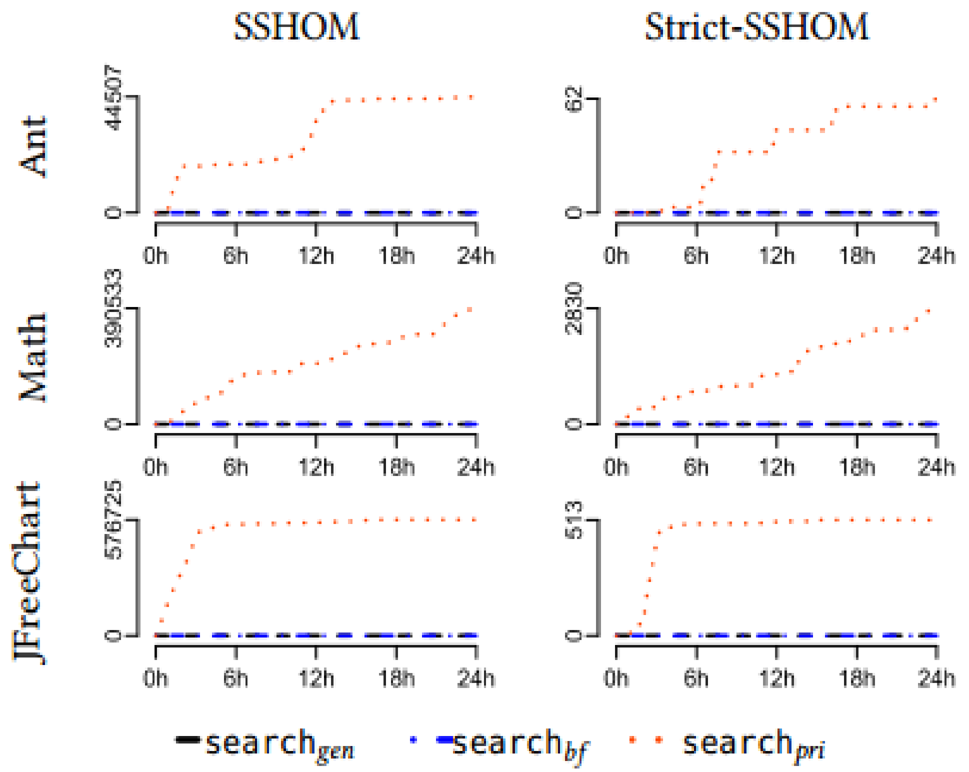
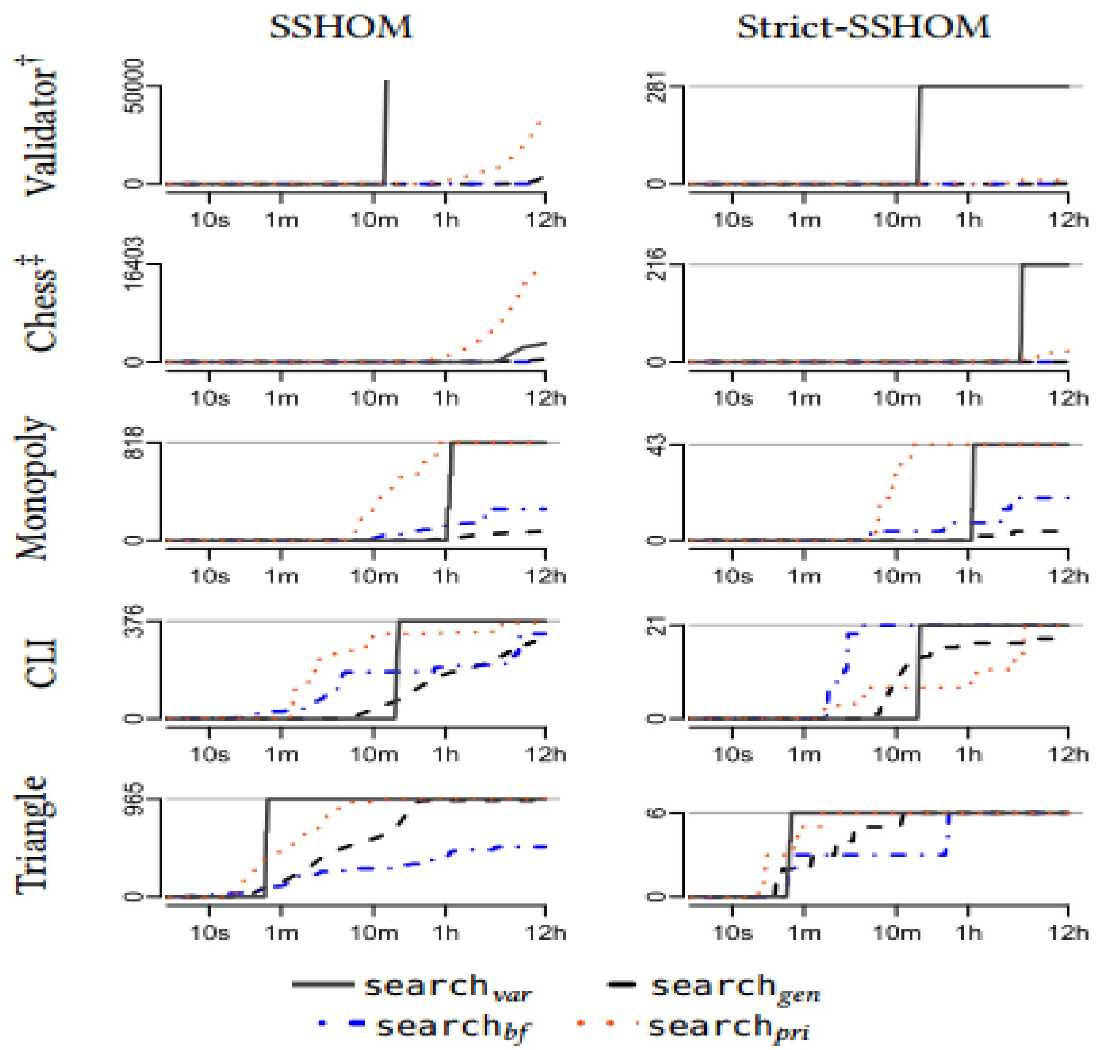


Figure 6: (Strict-)SSHOMs found over time, averaged over 3 executions. Note that time is plotted in a linear scale as SSHOMs are found consistently over time due to batching.



† We cap the plot for Validator since there are 13.4 billion SSHOMs; ‡ we could not enumerate all nonstrict-SSHOMs for Chess due to the difficulty of the SAT problem and report only those found within the time limit

Figure 4: (Strict-)SSHOMs found over time in each subject system, averaged over 3 executions. Note that time is plotted in log scale as most SSHOMs are found within the first hour.

总结

为了找到SSHOM,论文首先执行变分执行，提出Search_var在中小型程序中找到所有的SSHOM，然后分析其特征，并基于这些特征提出了一种新的优先搜索策略Search_pri，来在大型程序上寻找SSHOM。

- Search_var在中小型程序上寻找SSHOM的任务中表现优异
- Search_pri无论在中小型程序还是在大型程序中都表现优异，是至今为止表现最先进的寻找SSHOM的方法。