# 从数据中挖掘符号规则

## 大模型驱动的新型机器学习框架

2026-06-27

熊英飞 北京大学

# 1 概论

# 大模型正在替代自动化算法

| | | | |
|---|---|---|---|
| **大模型生成代码**<br>直接生成成千上万行代码，完成软件项目。 | → | **传统程序合成**<br>仅能合成单行小表达式或局部代码片段。 |
| **大模型生成证明**<br>直接完成公开数学问题的证明。 | → | **传统求解器**<br>只能在已知决策过程的小范围内证明。 |
| **大模型查找漏洞**<br>Mythos 已经对世界网络安全构成威胁 | → | **传统程序分析**<br>因为误报高居不下在实践中应用有限。 |
| **大模型生成优化**<br>对程序进行全面改写并大幅提升性能 | → | **传统编译优化**<br>只能进行局部小改动，很少影响复杂度。 |

# 软件自动化的未来是什么样的?

① 只有大模型

② 大模型和符号算法混合

③ 只有符号算法

# 为什么是符号算法?

**经过充分研究的问题上，理论最优算法远优于大模型**

- 大模型进行四则运算远不如计算器
- 在多数有判定过程的问题上，大模型也远不如传统方法

### 为什么现在大模型解得更好?

因为很多场景的自动化策略和算法**还没有被研究出来**。

### 如何能得到这些算法?

- 科学研究的一般规律：从特殊到一般，从一般到特殊。
- 从大量样例分析出合适的自动化策略，并形式化为符号规则

### 为什么之前没有这样做?

- 面临场景过多，**人力不够**
- 大模型时代，这一切都能交给大模型来做。
- 核心是设计合适的流程驱动大模型工作

# 另一个证据

Why Expert Systems Fail.
Michael Z. Bell, *The Journal of the Operational Research Society*, 1985.
DOI: 10.2307/2582480

## 专家系统失败的常见原因

- 专家没空
- 专家无法表述规则
- 专家不想表述规则
- 没有专家
- 缺乏常识
- 知识表示语言不够强
- 难以测试
- 用户不信任

**符号智能失败的核心原因:**
**没有足够多的专家来写规则。**

# 2 从证明数据挖掘符号证明算法 [arxiv:2605.08694]

# 动机

## 大模型证明智能体

- 混合大模型和符号求解器
- 展现出强大的证明能力
- 大模型带来昂贵的证明成本
- 在中小安全攸关软件企业难以使用

## 如何增强证明智能体?

- 大量已有研究关注增强大模型
- 但符号求解器（如：CoqHammer）同样值得增强
  - 如果增强能力落在大模型之外，则提升智能体整体能力
  - 如果增强能力落在大模型之内，则降低智能体整体开销
- 中小安全攸关软件企业往往只能使用符号求解器

# LLM2LTAC 方法概览

### 离线阶段

给定一个证明训练集，大模型分析每个证明，提取其中用到的证明策略

→

### 证明策略

大模型将证明策略编码成 Rocq Tactics

→

### 在线阶段

提取出的符号策略和 CoqHammer 交替运用，完成证明

# 离线阶段：提取

**证明准备**

提取定理、证明脚本、相关定义和引理

```
Lemma append_neutral_r :
forall (i : positive),
  append i xH = i.
Proof.
  induction i; simpl; congruence.
Qed.
```

**分析证明过程**

LLM 逐步解释证明过程

```
### Step-by-step explanation
1. ** induction i**
- ** Effect :** Performs induction on the binary positive integer i. So the induction generates three subgoals:
- Base case: i = xH
- Inductive case for xO p with IH for p
  ...
2. **simpl**
...
```

**识别通用策略**

要求 LLM 从中证明过程中识别出通用的证明策略

```
## General Proof Strategies
### 1. **Structural Induction with Recursive Data Types**
This pattern is widely applicable to theorems about recursive data structures like lists, trees, and natural numbers, where properties can be proven by following the structure's definition.
...
```

**Ltac 生成**

将自然语言策略用 Rocq 的 Ltac 语言编写为可执行策略

```
Ltac
structural_induction_recursive :=
  match goal with
  | [ |- forall p : positive, _ ] ⇒
    induction p; simpl; try
congruence
  | [ |- forall n : nat, _ ] ⇒
  ...
```

# 离线阶段：质量检查

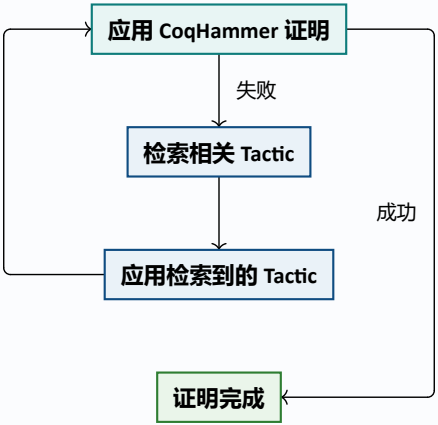
## 有效性测试

- 编译器检查策略有效性
- 如果没有通过编译则请 LLM 修复

## 泛化性测试

- 在一个验证集上检测策略泛化性
- 只保留让 a%-b% 的证明状态发生改变的策略
  - 低于 a% 说明泛化性不足
  - 高于 b% 往往是没有实际意义的策略，比如重命名变量

# 在线阶段：整体流程



# 在线阶段：检索

### 问题

- 直接检索：从证明状态检索 Ltac 策略
- **失败**：检索算法基于文本相似性，但 Ltac 程序文本密度低。

### 解决方案

- 首先将目标证明目标匹配到源定理
- 再从定理找到所关联的 Ltac 策略

# 实验验证

## 实验设置

- 训练集：Rocq 标准库
- 测试集：来自四个真实项目的定理：CompCert、Coq-Art、Ext-Lib、Vfa
- 提取模型：DeepSeek V3.2，GPT 5.2

## 实验结果

- 显著提升符号求解器的能力
- 集成到大模型智能体之后，智能体的求解能力和词元效率均得到提升

**+23.87%**

CoqHammer 求解数提升

**+9.90%**

集成到 Claude Code 后求解数提升

**−10.51%**

集成到 Claude Code 后词元消耗降低

# 3 融合符号规则和大模型的自动代码优化 [TOSEM26]

# 研究动机：一个优化示例

```
R_API const char* r_reg_32_to_64(RReg* reg,
    const char* rreg32) {
  // ...
  for (i = 0; i < R_REG_TYPE_LAST; ++i) {
    r_list_foreach(reg->regset[i].regs, iter, item) {
      if (!r_str_casecmp(rreg32, item->name)
          && item->size == 32) {
        j = item->offset;
        break;
      }
    }
  }
}
```

```
R_API const char* r_reg_32_to_64(RReg* reg,
    const char* rreg32) {
  // ...
  for (i = 0; i < R_REG_TYPE_LAST; ++i) {
    r_list_foreach(reg->regset[i].regs, iter, item) {
      if (item->size == 32 //交换了顺序
          && !r_str_casecmp(rreg32, item->name)) {
        j = item->offset;
        break;
      }
    }
  }
}
```

通过交换子条件的顺序，避免开销较大的运算

# 2.2 现有方法局限性与本研究的思路

### 基于规则的方法

难以对所有场景都精确覆盖

- Clang-tidy 不支持“交换条件顺序”
- CLion 会对所有条件都建议交换

### 纯 LLM 端到端优化

- 策略空间和代码空间都很庞大
- LLM 难以定位优化位置和对应策略，往往只能给出通用建议

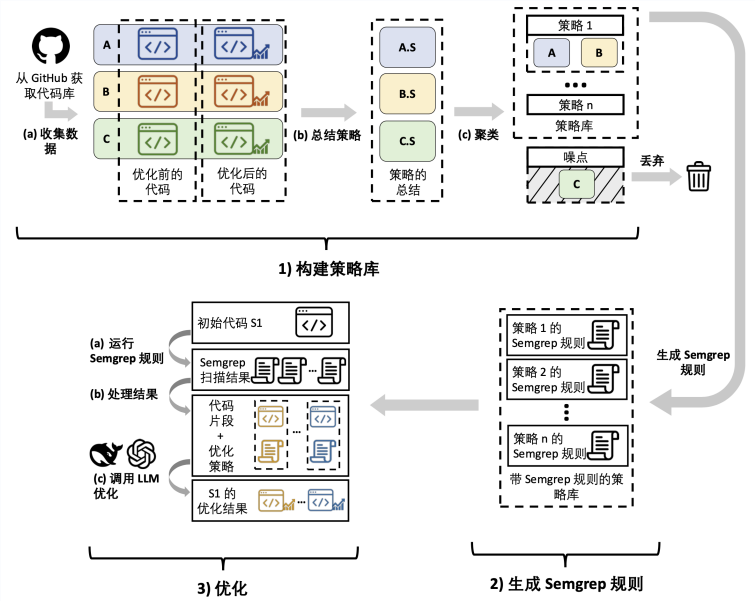
### 检索增强方法

- 指定位置和样例后模型优化效果提升
- 但基于文本相似度检索样例往往返回无关结果

### 本研究核心思路

**静态分析匹配优化位置和对应策略：** 由 LLM 生成的 Semgrep 规则在代码中匹配可优化位置和对应优化策略，再由 LLM 在策略提示下生成具体优化代码。

# SemOpt 框架概述


从 GitHub 获取代码库
(a) 收集数据
A
B
C
优化前的代码
优化后的代码
(b) 总结策略
A.S
B.S
C.S
策略的总结
(c) 聚类
策略 1
A
B
策略 n
策略库
噪点
C
丢弃
1) 构建策略库
生成 Semgrep 规则
策略 1 的 Semgrep 规则
策略 2 的 Semgrep 规则
策略 n 的 Semgrep 规则
带 Semgrep 规则的策略库
2) 生成 Semgrep 规则
初始代码 S1
(a) 运行 Semgrep 规则
Semgrep 扫描结果
(b) 处理结果
代码片段 + 优化策略
(c) 调用 LLM 优化
S1 的优化结果
3) 优化

# 策略库构建

## 数据来源

- GitHub 活跃开源仓库（100+ stars，近 5 年活跃）
- C/C++：30,342 个代码库，35,668 个优化提交
- Python：9,316 个代码库，10,084 个优化提交

## 策略提炼流程

1. LLM 为每个提交生成 3 条策略描述
2. 选择与其他描述平均相似度最高的描述
3. 使用 DBSCAN 算法对策略做聚类

| 指标 | C/C++ | Python |
| --- | --- | --- |
| 代码库 | 30,342 | 9,316 |
| 优化提交 | 35,668 | 10,084 |
| 聚类数 | 155 | 55 |
| 平均簇大小 | 13.4 | 19.9 |

# 实验结果：复现程序员的历史优化

## C/C++ 代码优化（151 个任务）

| 方法 | 精确匹配 | 语义等价 |
| --- | --- | --- |
| 直接提示 | 1.32% | 5.96% |
| RAG | 16.56% | 23.84% |
| RAPGen | 0.00% | 1.99% |
| SemOpt | **27.81%** | **42.38%** |
| Clang-tidy | 2.65% | 2.65% |

## Python 代码优化（150 个任务）

| 方法 | 精确匹配 | 语义等价 |
| --- | --- | --- |
| 直接提示 | 4.00% | 9.33% |
| RAG | 16.00% | 28.67% |
| RAPGen | 4.00% | 10.00% |
| SemOpt | **25.33%** | **46.00%** |

## 实验结果

- 相比 RAPGen，SemOpt 实现 **14 至 21 倍**的相对提升
- 相比直接提示，精确匹配提升 **4 至 21 倍**，语义等价提升 **3 至 7 倍**
- **89.86%** 的优化建议经验证可保持语义且产生性能收益

# 2.8 实验验证：跨代码库泛化能力

## 评测设计

评估方法在面对**从未见过的代码库中的优化任务**时的性能表现

- **标准模式**：排除与目标代码完全相同的提交
- **降级模式**：排除同一代码库的所有历史知识

| 方法 | 模式 | DeepSeek 语义等价 | 变化 |
|---|---|---|---|
| RAG | 标准 | 36 (23.84%) | |
| RAG | 降级 | 10 (6.62%) | **下降 72%** |
| SemOpt | 标准 | 64 (42.38%) | |
| SemOpt | 降级 | 63 (41.72%) | **仅下降 1.7%** |

## 核心洞察

通过抽取结构化策略并转化为规则化检测，SemOpt 实现了从特定项目知识到通用优化模式的有效迁移。

# 2.9 真实场景验证

C/C++ 项目优化结果

| 项目 | 最大提升 | 平均提升 | ≥5% 用例数 | 状态 |
|---|---|---|---|---|
| RocksDB | 5.04% | 2.41% | 1/2 | PR 审查中 |
| Redis | 6.14% | 1.68% | 3/21 | ✓ 已接受 |
| gRPC | 35.48% | 3.10% | 2/16 | PR 审查中 |
| LevelDB | 218.07% | 10.40% | 2/22 | 社区认可 |
| spdlog | 20.00% | 3.27% | 6/45 | ✓ 已接受 |

Python 项目优化结果

| 项目 | 最大提升 | 平均提升 | ≥10% 用例数 |
|---|---|---|---|
| click | 479.90% | 143.25% | 11/11 |
| flask | 66.07% | 56.67% | 8/8 |
| jinja2 | 71.43% | 48.60% | 7/8 |
| requests | 85.45% | 40.19% | 8/9 |
| scrapy | 61.77% | 50.26% | 4/4 |

所产生的部分优化效果用 Cursor+GPT5.4 High 长时间多轮尝试也无法达到

# 4 符号策略挖掘统一框架 [进行中工作]

# 符号策略挖掘问题

- 给定训练集，如何从训练集中得到高效的符号策略集合，并集成到系统中完成任务？
  - LLM2LTAC：从训练集中挖掘证明策略，生成 Ltac 规则，提升 CoqHammer 的证明能力
  - SemOpt：从训练集中挖掘代码优化机会匹配策略，生成 Semgrep 规则，指导大模型进行优化

# 符号策略挖掘步骤

**训练集**

任务实例与人类经验

→

**NL 描述**

LLM 抽取规则意图

→

**筛选**

合并相似策略，剔除无效策略

→

**符号规则**

生成可执行规则

→

**系统集成**

利用策略完成任务

**LLM2Ltac**

历史证明 → 证明策略描述 → 筛选 → Ltac 策略 → 和 CoqHammer 集成

**SemOpt**

历史优化 → 优化策略描述 → 聚类 → Semgrep 规则 → 指导大模型优化

# 开放问题

### 规则质量

如何降低 LLM 生成规则的幻觉、过拟合和漏报?

### 规则选择

在线阶段如何检索、排序和组合大量候选规则?

### 验证闭环

如何检查生成的策略的质量?

### 演化机制

能否保证系统随训练集数据增多能力单向增强?

# 5 总结

# 其他进行中的工作

- 自动证明模型和智能体
  - 命令式语言自动证明模型[arxiv:2604.05820]
  - Rocq 自动证明智能体[arxiv:2604.19558]
    - 目前 Rocq 上的 SOTA 智能体，甚至超过 QCP 官方 Skill
  - QCP、C*、FramaC 自动证明智能体
- 大模型驱动的形式化验证流程[ICSE26 Distinguished]
  - 完整的形式化基础设施要求较高：形式化规约、完整语义、完全实现
  - 指导大模型严格按形式化方法论证程序性质
    - 比大模型直接判断更准
    - 可在自然语言规约、无完整语义等场景使用
- 经验证的优化策略挖掘
  - SemOpt 目前只是挖掘匹配规则
  - 后续工作：挖掘匹配和优化规则
  - 设计新的编程语言来完成这一工作

# 总结

- 在大量问题上，符号算法仍然是最终追求的目标
  - 包括约束求解问题
- 在完全替代大模型之前，符号算法的发展也能提升混合系统的效果和降低资源消耗
- 现有符号算法的主要问题是缺乏足够多的人力来覆盖所有场景
- 用大模型挖掘策略是一条可行之路
- 很多开放问题有待解决

欢迎大家一起来研究！