软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(8):3161−3179 [doi: 10.13328/j.cnki.jos.007592] [CSTR: 32375.14.jos.007592]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于并行探索的大模型缺陷定位增强*

秦意浩, 王尚文, 林 博, 陈立前, 刘万伟, 毛晓光

(国防科技大学 计算机学院, 湖南 长沙 410005)

通信作者: 王尚文, E-mail: wangshangwen13@nudt.edu.cn

摘 要: 软件缺陷定位是软件工程领域的重要问题. 近年来, 基于大语言模型的缺陷定位方法在缺陷定位任务中展现出较好前景. 现有方法仅为大语言模型维护单一决策路径, 导致搜索范围有限, 缺陷定位效果不够理想. 针对此问题, 提出一种基于并行探索的大模型缺陷定位增强方法 PRIME. 通过设计缺陷位置的并行探索机制提高大语言模型的搜索范围, 并结合节点重要性评估方法对大语言模型预测的多个候选缺陷位置进行排序, 形成优化后的缺陷定位结果. 通过与其他缺陷定位方法的对比分析以及全面的消融实验和参数影响分析, 验证所提方法可以有效增强大语言模型的缺陷定位性能. PRIME 在 Top-1 指标上较现有方法的提升幅度超过 18%, 其在 *MAP* 和 *MRR* 指标上的性能提升分别可达 15% 和 25%.

关键词: 大语言模型; 缺陷定位; 节点重要性评估

中图法分类号: TP311

中文引用格式: 秦意浩, 王尚文, 林博, 陈立前, 刘万伟, 毛晓光. 基于并行探索的大模型缺陷定位增强. 软件学报, 2026, 37(8): 3161–3179. http://www.jos.org.cn/1000-9825/7592.htm

英文引用格式: Qin YH, Wang SW, Lin B, Chen LQ, Liu WW, Mao XG. Boosting LLM-based Fault Localization with Parallel Exploration. Ruan Jian Xue Bao/Journal of Software, 2026, 37(8): 3161–3179 (in Chinese). http://www.jos.org.cn/1000-9825/7592.htm

Boosting LLM-based Fault Localization with Parallel Exploration

QIN Yi-Hao, WANG Shang-Wen, LIN Bo, CHEN Li-Qian, LIU Wan-Wei, MAO Xiao-Guang

(College of Computer Science and Technology, National University of Defense Technology, Changsha 410005, China)

Abstract: Software fault localization is a critical issue in software engineering. In recent years, fault localization methods based on large language models (LLMs) have demonstrated a promising prospect in fault localization tasks. However, existing methods maintain only a single decision path for LLMs, which limits the search scope and results in suboptimal fault localization performance. To this end, this study proposes PRIME, an enhanced fault localization method for LLMs based on parallel exploration. The search scope of LLMs is broadened by designing a parallel exploration mechanism for fault locations. Furthermore, multiple candidate fault locations predicted by LLMs are ranked by combining a node importance evaluation method to generate optimized fault localization results. By conducting comparative analysis with other fault localization methods, comprehensive ablation experiments and parameter influence analysis, it is verified that the proposed method can effectively enhance the fault localization performance of LLMs. Compared with the existing methods, PRIME improves the Top-1 metric by over 18%, and its performance improvements in *MAP* and *MRR* metrics can reach 15% and 25%, respectively.

Key words: large language model (LLM); fault localization; node importance evaluation

软件缺陷定位技术是指在软件开发和维护过程中, 可以自动识别并定位代码中导致程序行为异常或不符合预期的缺陷所在位置的方法和工具. 近年来, 随着大语言模型 (large language model, LLM) 技术的发展, 基于 LLM 的缺陷定位方法可以对软件错误信息进行分析, 并自主地调用预定义的函数从软件代码仓库中寻找可能存在缺陷的

* 基金项目: 国家自然科学基金 (62402506, 62474196); 国防科技大学研究基金 (ZK24-05)
本文由“大模型与软件工程”专题特约编辑聂长海教授、王璐副教授、王莹教授、姜艳杰副教授、张敏灵教授推荐.
收稿时间: 2025-09-07; 修改时间: 2025-10-28; 采用时间: 2025-12-08; jos 在线出版时间: 2025-12-24
CNKI 网络首发时间: 2026-03-04

代码位置, 在减少开发人员调试时间、提高软件质量和可靠性等方面展现出良好的发展前景.

包含缺陷的软件仓库常具备海量的源代码, 而 LLM 可以接受的输入文本有限, 且在输入内容过多时容易造成性能下降[1]. 因此, 直接将整个软件的源代码提供给 LLM 往往是不可行的, 如何从软件仓库中提取与软件缺陷联系紧密的上下文 (context) 信息成为一个重要的问题. 如图 1 所示, 借助 LLM 自身的决策和函数调用能力, 现有的基于 LLM 的缺陷定位方法采取的一种主流解决方案为: 将失败测试用例源代码等缺陷信息提供给 LLM, 并定义“获取指定类信息”“获取指定方法信息”等可供 LLM 调用的函数, LLM 可以通过多轮函数调用自主地获取上下文信息, 并最终定位至可疑的源码位置. 然而, 现有方法存在一个严重的局限性, 即 LLM 在定位过程中只探索单条决策路径, 能够浏览到的上下文信息十分有限. 图 1 中展示了一个对应的例子, 其中实线标记部分表示 LLM 已经探索的单条决策路径, 虚线标记部分表示 LLM 还未探索的路径. 从图 1 中可以看出, 在只维护一条推理路径的情况下, LLM 检查了 findInvocations 方法, 发现该方法并非缺陷位置, 随即终止缺陷定位过程. 在这种情况下, LLM 会失去探索其他分支路径的机会, 从而使 LLM 无法寻找到真实缺陷方法 hasSameMethod.

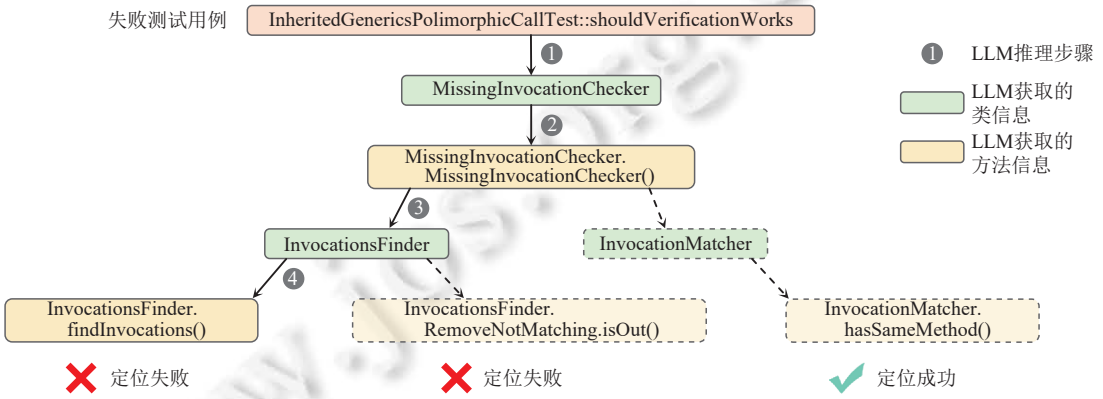


图 1 基于大语言模型的软件缺陷定位方法示意图

在基于 LLM 的软件缺陷定位任务中, 特别是在软件系统规模庞大且缺陷位置隐蔽的情况下, 如何进一步提高 LLM 能够探索到的代码范围, 从而提高缺陷定位的成功率? 针对这一挑战, 本文提出了一种基于并行探索的大模型缺陷定位增强方法 PRIME. 为解决现有方法决策路径单一的问题, PRIME 将缺陷定位任务定义为一个搜索问题, 通过在每一推理步骤对 LLM 的回答进行多次采样和多样性驱动的动作选择, PRIME 支持将单一搜索路径拓展为可以并行推理的多条搜索路径, 旨在增大 LLM 寻找到错误代码位置的成功率. 然而, 更多的搜索路径意味着定位方法将输出更多的可疑代码位置, 这可能会为开发人员确认真实的错误位置带来额外的工作量. 为进一步解决此挑战, PRIME 引入了基于节点重要性评估的定位结果优化算法. 具体而言, PRIME 首先将定位结果中的每个可疑代码位置映射到方法调用图的单个节点, 通过子图提取重塑节点之间的拓扑结构; 随后, PRIME 通过节点频率和节点在子图中的中心性属性计算每个节点的重要性; 最后, PRIME 根据重要性得分对所有候选缺陷位置进行排序, 为用户提供优化后的缺陷定位结果. 本文的主要贡献如下.

(1) 提出一种新的软件缺陷定位增强策略, 充分利用 LLM 推理时的计算资源, 通过并行探索扩展 LLM 的决策路径, 并结合多样性驱动的动作选择扩大 LLM 搜索范围, 有效提高 LLM 识别真实缺陷位置的成功率.

(2) 提出一种新的软件缺陷定位结果优化策略, 通过分析多个可疑代码位置的频率属性和在软件方法调用图中的拓扑结构, 有效地识别出更有可能存在缺陷的程序位置, 减少对定位结果进行二次确认的工作量.

(3) 通过在 691 个真实 Java 软件缺陷上的详细实验分析, 证实本文所提出的方法优于已有方法, 能够显著增强大语言模型的缺陷定位性能. PRIME 的源代码仓库链接为 https://github.com/IntHelloWorld/PRIME.

本文第 1 节介绍基于 LLM 的软件缺陷定位相关方法和研究现状. 第 2 节介绍本文所涉及的基本概念, 包括大语言模型及其函数调用能力, 结合搜索算法的人工智能系统, 节点重要性评估. 第 3 节介绍本文提出的基于并行探索的大模型缺陷定位增强方法 PRIME. 第 4 节通过对比实验、消融实验和参数影响分析验证本文方法的有效性.

第 5 节总结全文.

1 基于大语言模型的软件缺陷定位相关工作

作为软件工程领域的一个重要问题, 软件缺陷定位任务一直受到研究者们的广泛关注. 研究者们提出了多种多样的软件缺陷定位方法, 包括基于程序频谱的缺陷定位、基于程序依赖关系的缺陷定位、基于深度学习的缺陷定位等. 然而, 传统的定位方法存在明显的局限性, 基于程序频谱的缺陷定位需要通过运行大量开发人员编写的测试用例来统计覆盖信息, 存在运行开销大的问题; 基于程序依赖关系的缺陷定位和基于深度学习的缺陷定位从历史数据中学习缺陷的特征或模式, 而对于新的项目而言, 其历史缺陷数据是非常稀缺的, 这往往导致此类方法在新项目中的泛化性较差. 近年来, 随着大语言模型的出现, LLM 为开发者提供了前所未有的工具, 极大地提升了软件开发的效率. 目前, LLM 已经在代码生成[2,3]、测试用例生成[4−6]、缺陷修复[7−9]等许多软件工程任务中得到广泛应用.

在软件缺陷定位任务上, 最近有相关工作提出了几种基于 LLM 的软件缺陷定位方法, 大致可以分为两类: 面向固定工作流的定位方法和面向 LLM 自主决策的定位方法. 面向固定工作流的定位方法是指人工地将缺陷定位拆分为包含多个特定步骤的固定工作流, 并在每一步骤中调用 LLM 来完成指定任务. Qin 等人[10]提出了一种基于标准操作流程的缺陷定位方法, 该方法将缺陷定位任务分为 3 个彼此衔接的阶段: 错误理解、代码导航和错误确认, 在每一步骤中, 他们为 LLM 注入特定的上下文信息和来自前序阶段的中间结果, 指导 LLM 逐步找到可疑的方法级缺陷位置. Xia 等人[11]提出了另一种定位工作流, 该方法首先将整个代码仓库组织为仓库目录、代码文件骨架、具体代码行的 3 层结构, 随后结合基于语义相似度的检索技术和 LLM 的分析自顶向下逐步定位至行级别的缺陷位置. 随着 LLM 的任务规划能力和工具使用 (即函数调用) 能力的不断增强, 面向 LLM 自主决策的定位方法应运而生, 并逐渐成为主流. 为了有效地在大型软件仓库中定位缺陷并克服 LLM 上下文长度的限制, Kang 等人[12]最先提出让 LLM 使用函数调用来主动搜索代码库中与缺陷相关的信息, 此外, 他们还发现通过提供自然语言的解释, 基于 LLM 的定位方法有助于帮助开发人员更好地理解为什么某个位置被认为是缺陷源, 从而有效提升自动化定位工具的使用体验. Xu 等人[13]提出了一个两阶段的框架, 在第 1 阶段, 该方法首先利用基于程序频谱的缺陷定位技术缩小缺陷代码的搜索空间, 生成一个可疑方法的候选列表; 在第 2 阶段, 利用 LLM 深入检查第 1 阶段建议的方法代码片段, 以完善缺陷定位结果. 该方法能灵活地利用多种错误相关信息, 并通过构建基于开源 LLM 的智能体来定位缺陷. Jiang 等人[14]提出了一种由 LLM 驱动的、针对函数级缺陷的定位方法 CoSIL, 该方法通过模块调用图减少搜索空间, 并通过迭代搜索函数调用图获取相关上下文, 使用上下文剪枝来控制搜索方向并有效管理上下文, 以应对 LLM 上下文窗口长度的限制.

总的来看, 现有的基于 LLM 的软件缺陷定位方法都涉及一个共同的步骤, 即需要从软件仓库的海量代码中收集与缺陷相关的信息, 并在此基础上搜索可能导致缺陷的代码位置. 然而, 不论是面向固定工作流还是面向 LLM 自主决策的定位方法, 往往只为 LLM 维护单一的搜索或决策路径, 其在代码仓库搜索上的广度和深度都存在严重不足. 针对这一问题, 本文的研究旨在提高缺陷位置搜索效率, 从而提高基于 LLM 的定位方法找到缺陷代码位置的成功率.

2 基本概念

2.1 大语言模型及其函数调用能力

大语言模型是指基于深度学习, 特别是基于 Transformer 架构的人工智能模型. LLM 可以被视为一个强大的文本序列概率分布学习器. 给定一个文本序列 $S=(w_1,w_2,\ldots,w_T)$, 其中每个 w_t 代表一个词元 (token), LLM 的目标是学习该序列的联合概率分布 $P(S)$, 这个联合概率可以被分解为一系列条件概率的乘积:

$$P(S)=P(w_1,w_2,\ldots,w_T)=\prod_{t=1}^{T}P(w_t|w_1,w_2,\ldots,w_{t-1}) \tag{1}$$

其中, $P(w_t|w_1,w_2,\ldots,w_{t-1})$ 表示在已知前 $t-1$ 个词元 (即上下文) 的条件下, 下一个词元为 w_t 的概率.

大语言模型通过在海量文本语料库上进行训练, 利用其深层神经网络来逼近这个条件概率函数. 在文本生成任务中, 模型接收一个初始文本序列 (即提示词 prompt) 作为上下文, 然后以自回归 (auto-regressive) 的方式逐个预测并生成最有可能出现的下一个词元, 从而构成连贯、有意义的完整文本. 在每一步 t, 模型从其预测的概率分布中采样或选择概率最高的词元 $\widehat{w}_t$, 并将其追加到现有上下文中, 用于下一步的预测, 这个过程可以形式化地描述如下:

$$\widehat{w}_t \sim P_\theta(w_t|w_1,w_2,\ldots,w_{t-1}) \tag{2}$$

传统的 LLM 主要局限于其内部知识库, 其知识存在截止日期, 且无法执行精确的数学计算或与外部世界进行实时交互. 为了突破这些局限性, 函数调用 (function calling) 范式应运而生. 函数调用能力使大语言模型能够充当一个智能的“调度中心”, 模型本身不直接执行代码, 而是通过理解用户的自然语言意图, 判断是否需要借助外部工具或 API 来完成任务, 并生成符合预定格式的、可由机器执行的函数调用指令. 该过程大致可分为以下 4 个阶段.

(1) 意图识别与工具选择: 模型接收用户输入 X_{prompt}, 并访问一个预先定义好的可用函数集合 $F=\{f_1,f_2,\ldots,f_n\}$. 每个函数 f_i 都有明确的描述、参数和格式要求. 模型分析 X_{prompt} 以确定最适合解决当前问题的函数 $f_i \in F$.

(2) 参数提取与格式化输出: 一旦选定函数 f_i, 模型会从 X_{prompt} 中提取执行该函数所需的具体参数值 $(v_1,v_2,\ldots,v_k)$. 随后, 模型并不生成自然语言回复, 而是生成一个结构化的、通常为 JSON 格式的对象 C_{call}, 该对象包含了函数名和参数键值对. 这个过程可以表示为:

$$C_{\text{call}} = LLM(X_{\text{prompt}},F) = \{\text{function\_name}: f_i,\ \text{arguments}: \{p_1: v_1, p_2: v_2,\ldots,p_k: v_k\}\} \tag{3}$$

(3) 外部执行与结果返回: 外部的执行环境 (如代码解释器、API 网关) 接收并解析 C_{call}, 执行实际的函数调用 $f_i(v_1,v_2,\ldots,v_k)$, 并获得执行结果 R_{result}.

(4) 结果整合与最终应答: 执行结果 R_{result} 被返回给大语言模型, 作为新的上下文信息. 模型将原始问题和函数执行结果进行整合, 最终生成一段面向用户的自然语言回答 Y_{final}:

$$Y_{\text{final}} = LLM(X_{\text{prompt}},R_{\text{result}}) \tag{4}$$

通过这一机制, 大语言模型极大地扩展了其能力边界, 能够获取实时信息、执行精确计算、操作外部数据库或软件, 从而显著提升了其在复杂、动态任务中的实用性和事实准确性.

2.2 结合搜索算法的人工智能系统

人工智能 (artificial intelligence, AI) 系统存在着丰富的运用搜索和规划算法的历史. 广度优先搜索、深度优先搜索和 A*搜索[15]等搜索算法在 AI 系统中得到广泛应用. Newell 等人[16]将目标导向的行为视为在可能状态空间中的搜索过程. Tash 等人[17]提出了在有限搜索范围内进行的规划算法, 并采用扩展策略, 根据信息价值的启发式知识来优化计划, 他们的研究表明, 这种方法使智能体能够对时间压力和世界中的随机性做出适当的反应. 1997 年, 基于大规模并行树搜索的人工智能国际象棋引擎 Deep Blue[18]击败了世界冠军 Kasparov, 证明了机器在具有明确规则、信息完全透明的复杂决策游戏中, 可以通过强大的计算能力超越人类. 在多人无限注德州扑克游戏中, Pluribus 人工智能系统[19]可以基于多次自我博弈构建的蓝图策略, 并针对当前对手的特定行动进行有限的实时搜索, 以做出更精细、更具针对性的决策. 2019 年, Pluribus 在一场实验中, 与多位世界顶级的职业扑克选手进行了 6 人无限注德州扑克的比赛, 并取得了决定性的胜利, 证明了 AI 不仅能处理信息完全的对弈, 也能在充满未知和欺诈的不完美信息环境中做出超越人类的决策.

在深度学习领域, 结合神经网络组件的搜索算法在多个任务中实现了超人水平的表现. DeepMind 公司开发的围棋人工智能程序 AlphaGo[20]通过使用蒙特卡洛树搜索 (Monte Carlo tree search, MCTS) 算法, 从庞大的可能性中, 更有效地探索和评估最佳的落子选择. Gray 等人[21]通过采取 one-step lookahead 搜索算法, 在著名策略桌面游戏 Diplomacy 中达到了最好的成绩. 一些最近的研究[22,23]表明, 将搜索算法应用于大语言模型可以引入对多条推理路径的探索, 从而显著提高 LLM 在需要进行复杂规划任务上的表现.

2.3 节点重要性评估

在复杂网络分析中, 评估网络中节点的重要性是一个核心且基础的问题. 节点的重要性或称为中心性 (centrality)

并非一成不变, 而是取决于其在网络拓扑结构中所处的位置和扮演的角色. 一个重要的节点可能拥有大量的连接, 或位于众多信息传播路径的关键位置上, 或能够快速地将信息传递给网络中的其他节点. 为了从不同维度量化这种重要性, 研究者们提出了一系列经典的节点重要性评估算法.

(1) 度中心性 (degree centrality): 度中心性是最直观、最简单的节点重要性度量指标. 它认为, 一个节点的直接连接越多, 其在网络中的影响力就越大. 对于一个无向网络中的节点 v_i, 其度中心性 $C_D(v_i)$ 直接定义为其度 k_i. 为了在不同规模的网络之间进行比较, 通常会进行归一化处理. 在一个包含 N 个节点的网络中, 归一化后的度中心性如下:

$$C'_D(v_i) = \frac{k_i}{N-1} \tag{5}$$

(2) 介数中心性 (betweenness centrality): 介数中心性衡量的是一个节点在网络中扮演“桥梁”角色的程度. 它计算的是网络中所有节点对之间的最短路径中, 经过该节点的路径数量所占的比例. 一个节点的介数中心性越高, 它对网络中信息的流通和控制能力就越强. 其数学表达式为:

$$C_B(v_i) = \sum_{s \neq i \neq t} \frac{\sigma_{st}(v_i)}{\sigma_{st}} \tag{6}$$

其中, s 和 t 是网络中除 v_i 外的任意两个节点, σ_{st} 表示从节点 s 到节点 t 的最短路径总数, 而 $\sigma_{st}(v_i)$ 表示这些最短路径中经过节点 v_i 的数量. 高介数的节点一旦被移除, 可能会导致网络分割或显著增加其他节点间的通信成本.

(3) 特征向量中心性 (eigenvector centrality): 特征向量中心性的核心思想是, 一个节点的重要性不仅取决于其连接数量, 更取决于其邻居节点的重要性. 与一个高度重要的节点相连, 比与许多不重要的节点相连更能提升自身的重要性. 设节点 v_i 的中心性为 x_i, 它正比于其所有邻居节点中心性的总和.

$$x_i = \frac{1}{\lambda} \sum_{j \in N(i)} x_j \tag{7}$$

其中, $N(i)$ 是节点 v_i 的邻居节点集合, λ 是一个常数.

上述算法从不同角度刻画了节点的重要性. 度中心性关注局部影响力, 介数中心性关注节点对信息流的控制力, 特征向量中心性则关注节点的“背景”或“声望”. 在不同的网络分析任务中, 需要根据网络特性和研究目标选择最合适的评估算法, 有时甚至需要结合多种算法进行综合评估, 以获得对节点重要性更全面、准确的认识.

3 基于并行探索的大模型缺陷定位增强方法 PRIME

为了解决现有缺陷定位方法搜索范围小、定位准确率较低的问题, 本文提出一种基于并行探索的大模型缺陷定位增强方法 PRIME, 旨在提高现有定位技术的方法级 (在面向对象程序语言中, “方法”是一段依附于类存在的、用于执行特定功能并可通过名称重复调用的代码块, 为避免歧义, 本文剩余部分使用 method 进行指代) 缺陷定位准确率, 如图 2 所示, PRIME 的工作流程由 2 步组成.

在第 1 步中, 首先通过轻量化的程序插桩技术跟踪失败测试用例在执行过程中覆盖的所有类和 method, 并识别 method 之间的动态调用关系, 构建程序动态调用图. 整个软件系统规模庞大, 有时可能包含数十万行代码, 从整个软件系统中直接定位缺陷位置是十分困难的. 程序动态调用图可以让 LLM 只关注于失败测试用例所触发的软件功能, 从而极大地压缩了搜索空间, 有助于提高缺陷定位的准确率. 在获得程序动态调用图后, 指导 LLM 通过多轮调用预先定义的函数在图上搜索缺陷位置, 其中在每次 LLM 进行推理时, 通过对 LLM 回复进行多次采样, 允许其产生不同的搜索路径.

在第 2 步中, 引入节点重要性评估方法来度量多个候选缺陷位置的可疑程度差异, 对候选缺陷位置进行排序, 获得最终的缺陷定位结果. 特别的, 由于候选缺陷位置在程序动态调用图中所对应的多个关键节点之间不一定存在关联边, 需要通过基于最短路径的子图提取算法重建关键节点之间的拓扑关系, 形成连通子图. 此外, 由于 LLM 在不同的推理路径中可能会产生相同的候选缺陷位置, 这种现象被称为 LLM 的自一致性 [24], 如果一个候选缺陷位置出现频率更高, 说明 LLM 更有可能认为该位置可疑, 因此需要额外考虑每个关键节点的频率属性. 对于每个

关键节点 (即候选缺陷位置), 其重要性值由节点频率属性值和节点中心性属性值线性加权融合而成, 用于表示候选缺陷位置的可疑程度.

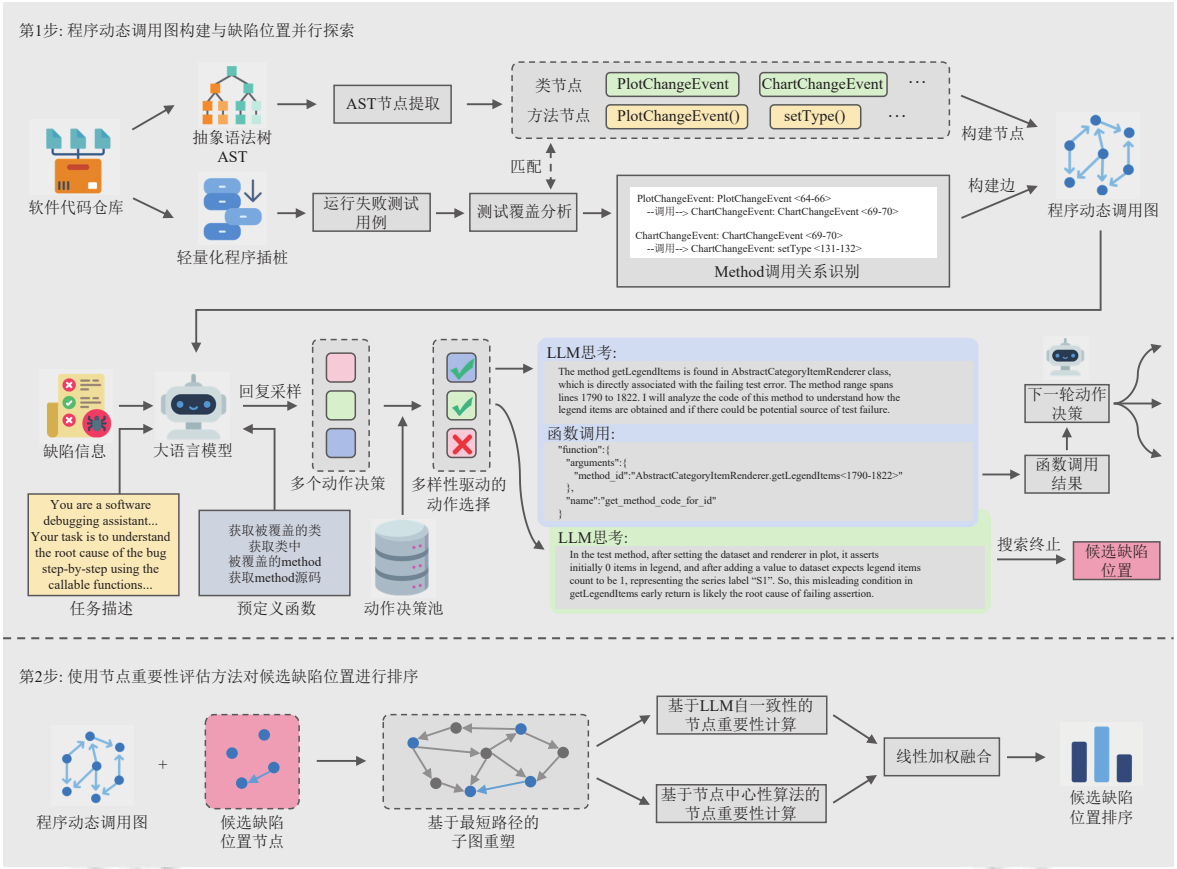


图 2 PRIME 流程图

3.1 问题定义

软件缺陷定位的本质是一个信息搜寻与推理的过程. 在任意时刻, LLM 都无法直接观测到系统内部的完整状态 (例如, 缺陷的精确位置、所有变量的实时值等), 而只能通过执行特定的诊断工具或查询代码库来获得片面的观测结果. 为了系统性地解决软件缺陷定位的复杂性, 本文将该任务构建为一个由 LLM 作为决策引擎的部分可观测马尔可夫决策过程 (partially observable Markov decision process, POMDP). 此框架形式化地描述了一个智能体如何在不完全信息的环境下, 通过一系列的探索性动作来逐步收集信息, 并最终收敛至一个目标状态, 即成功定位缺陷. 我们将由 LLM 驱动的缺陷定位任务定义为一个六元组 $M=(S,A,T,\Omega,O,\pi)$, 包含状态空间 S、动作空间 A、状态转移函数 T、观测空间 R、策略函数 π. 接下来详细阐述每个组成部分的具体定义.

(1) 状态空间 S

状态 $s\in S$ 是对整个存在缺陷的软件环境在某一时刻的完整、真实的描述, 状态 s 对于 LLM 是部分不可见的. 一个状态 s 至少包含以下核心要素:

$$s=(C,F_{\text{fail}},H) \tag{8}$$

其中, C 代表整个缺陷软件的源代码仓库. F_{fail} 为初始缺陷信息, 在本文中包括运行失败的测试用例源代码、失败测试抛出的异常信息以及完整的堆栈跟踪 (stack trace). H 表示 LLM 与环境交互的历史记录, 即一个“观测-动作”序

列, $H_t=(a_0,o_1,a_1,o_2,\ldots,a_{t-1},o_t)$, 它隐式地包含了 LLM 已经获取的所有上下文信息. 基于以上定义, LLM 的初始状态 s_0 为一个包含缺陷的代码库 C 和失败测试用例产生的初始缺陷信息 F_{fail}, 历史记录为空集. 目标状态 s_{goal} 则是 LLM 成功识别到缺陷代码位置的状态.

(2) 动作空间 A

动作空间 A 定义了 LLM 为收集上下文信息所执行的所有操作, 这些动作被设计为一系列结构化的函数调用 (function call), 模拟了人类开发者在调试时所采取的步骤. 本文参照现有研究[12]为 LLM 提供了多个可以采取的动作 (详细内容可见第 3.2 节). 在每个决策步骤 t, LLM 基于其内部的知识选择一个动作 $a_t\in A$ 来执行.

(3) 状态转移函数 T

在此任务中, 动作的执行通过预定好的程序来实现, 因此状态转移函数 $T: S\times A\to S$ 被定义为确定性的, 这意味着在给定当前状态 s_t 和 LLM 执行的动作 a_t 后, 系统将转移到一个唯一确定的新状态 s_{t+1}. 由于代码库 C 和初始缺陷信息 F_{fail} 在一次调试会话中通常是固定的, 因此对 LLM 决策引擎而言, 状态转移函数 T 的作用主要是更新历史记录 H, 即 $s_{t+1}=(C,F_{\text{fail}},H_t\cup\{a_t,o_{t+1}\})$. 其中 o_{t+1} 是执行动作 a_t 后产生的新观测.

(4) 观测空间 Ω 与观测函数 O

观测空间 Ω 是 LLM 执行动作后可能收到的所有上下文信息的集合. 每个观测 $o\in\Omega$ 都是对缺陷软件环境整体状态 s 的一个片面反映. 观测的形式与动作直接相关, 如图 2 中所示, 调用 get_method_code_for_id 工具会返回指定 method 的完整源代码. 在本文中, 观测函数 $O: S\times A\to\Omega$ 同样是确定性的, 它描述了在状态 s_t 下执行动作 a_t 会产生哪一个确定的观测 o_{t+1}.

(5) 策略函数 π

POMDP 框架的目标是学习一个最优策略 $\pi^*: B(S)\to A$, 该策略将当前的信念状态 $b(s)$ (即关于当前真实状态的概率分布) 映射到一个最优动作. 在基于 LLM 的缺陷定位任务中, LLM 本身就扮演了策略函数 π 的角色. LLM 的上下文窗口 (context window) 充当了维护信念状态的载体. 我们将初始任务指令 I、故障上下文 F_{fail} 以及整个“动作-观测”历史 $H_t=(a_0,o_1,a_1,o_2,\ldots,a_{t-1},o_t)$ 拼接成一个详尽的提示词 (prompt) 输入至 LLM, LLM 可以基于这个提示生成下一步的动作 a_t.

$$a_t=\pi_{\text{LLM}}(Prompt(F_{\text{fail}},H_t)) \tag{9}$$

这个过程可以被看作是对最优策略 π^* 的近似. LLM 通过其预训练知识库中蕴含的关于代码缺陷模式的知识, 来推断哪一个动作最有希望成功定位缺陷位置.

3.2 程序动态调用图构建与缺陷位置并行探索

(1) 程序动态调用图构建

在复杂软件系统中, 组件之间复杂的调用关系是导致缺陷难以定位的关键因素之一. 静态分析虽然能勾勒出程序的完整调用结构, 但无法反映特定运行场景下的实际执行路径, 往往引入大量与缺陷无关的冗余信息. 因此, 直接在整个缺陷软件中搜索缺陷位置相对困难. 为了精准地表征与特定测试用例相关的程序行为, 本文首先提出了一种结合静态分析与动态追踪的混合方法, 用于构建面向具体缺陷场景的程序动态调用图.

本文将一个程序动态调用图 $G=(V,E)$ 定义为一个包含两类节点和两类边的异构有向图, 其中 V 是图中节点的有限集合, 由类节点集 V_C 和 method 节点集 V_M 的并集构成, 即 $V=V_C\cup V_M$, 每个类节点 $v_c\in V_C$ 代表一个类, 每个类节点记录了类名 (如 org.jfree.chart.AbstractCategoryItemRenderer)、文件路径 (如 src/org/jfree/chart/AbstractCategoryItemRenderer.java)、类是否被失败测试用例覆盖等信息. 每个 method 节点 $v_m\in V_M$ 代表一个 method, 记录其所属的类名、method 名、method 源代码、文件路径、是否被覆盖、method 所位于的代码行范围 (如第 25–28 行) 等信息. E 是图中边的有限集合, 由归属关系边集 E_B 和动态调用关系边集 E_C 的并集构成, 即 $E=E_B\cup E_C$. $E_B\subseteq V_C\times V_M$ 是归属关系边集 (belongs-to edges). 一条边 $e_b=(v_c,v_m)\in E_B$ 表示 method 节点 v_m 在静态结构上“属于”类节点 v_c. $E_C\subseteq V_M\times V_M$ 是动态调用关系边集 (call edges). 一条边 $e_c=(v_{m,i},v_{m,j})\in E_C$ 表示在失败测试用例的执行过程中, method $v_{m,i}$ 实际调用了 method $v_{m,j}$. 程序动态调用图的构建主要包括以下 3 个部分.

● 基于抽象语法树的静态特征与归属关系提取. 从源代码中提取所有潜在的图节点 (V_C 和 V_M 的候选集) 以及它们之间固有的归属关系 (E_B 的候选集), 采用静态分析技术, 遍历软件代码仓库中的源文件, 并为每个文件构建抽象语法树 (abstract syntax tree, AST)[25]. 通过深度优先遍历 AST, 系统地识别所有的类定义, 构成类节点集 V_C. 在每个类定义内部, 识别所有 method 的定义, 构成 method 节点集 V_M. 在提取节点的同时, 类与 method 之间的结构关系也随之确定. 对于每一个被识别出的 method 节点 $v_m \in V_M$ 及其所在的类节点 $v_c \in V_C$, 构建一条归属关系边 (v_c, v_m), 并将其加入归属关系边集 E_B. 通过以上方法, 可以获得程序完整的静态结构骨架, 包含了所有可能的类、method 以及它们之间明确的成员关系.

● 基于轻量化插桩的动态执行轨迹捕获. 为了确定在特定测试用例失败场景下哪些 method 被激活以及它们之间的实际调用关系, 采用轻量化程序插桩 (program instrumentation)[26]技术, 在每个 method 的入口和出口插入“探针”, 将已知的失败测试用例作为输入, 执行插桩后的程序. 探针将记录下一条时间有序的 method 级执行轨迹 $T_{\text{dynamic}} = \langle event_1, event_2, \ldots, event_n \rangle$, 该轨迹是后续构建调用关系边 E_C 和筛选被覆盖节点的直接依据. 值得注意的是, 传统的基于程序频谱的缺陷定位方法 (SBFL) 通常需要通过运行大量的测试用例来收集程序覆盖信息 (例如在 Defects4J 数据集的 Closure 项目中, 每个缺陷代码版本平均需要运行 6 560 个测试用例), 而本文方法仅需要对极少量失败的测试用例 (每个缺陷代码版本平均不到 3 个) 进行轻量化插桩, 其所需要运行的测试用例数量远少于 SBFL, 从而大幅缩减了运行开销, 适用于测试用例稀缺的使用场景.

● 异构图的动静融合与合成. 将前两个阶段的结果进行匹配与融合, 进行节点标记和调用关系边构建, 生成最终的程序动态调用图 $G = (V_C \cup V_M, E_B \cup E_C)$. 具体而言, 节点标记是指通过分析遍历动态执行轨迹 T_{dynamic}, 将所有在轨迹中出现过的类节点和 method 节点标记为“被失败测试用例所覆盖”. 对于调用关系边构建, 当 method $v_{m,i}$ 的执行包含了方法 $v_{m,j}$ 的完整执行时, 视为存在一个动态调用, 创建一条调用关系边 $e_c = (v_{m,i}, v_{m,j})$, 并加入边集 E_C. 如图 2 中的样例所示, 识别出调用链 PlotChangeEvent<64-66>→ ChartChangeEvent<69-70>→ setType<131-132>, 这些都将成为 E_C 中的边.

通过以上步骤, 可以构建出用于表征缺陷软件的程序动态调用图, 该图不仅通过调用关系边精确描绘了测试失败场景下的运行时行为, 还通过归属关系边保留了程序的面向对象结构信息. 需要强调的是, 与一般的程序调用图相比, 本文定义的程序动态调用图在图结构和上下文针对性上均展现出显著的差异: 在图结构上, 一般的程序调用图通常是同构的, 即节点仅表示 method, 边仅表示调用关系. 而本文的图是一种异构图, 其引入了两类节点 (类节点 V_C 和方法节点 V_M) 与两类边 (归属关系边 E_B 和动态调用关系边 E_D), 能够承载更丰富多元的程序信息; 在上下文针对性上, 本文提出的程序动态调用图具有高度的目标导向性. 与那些基于所有测试用例或任意单次执行所构建的通用动态调用图不同, 本文提出的调用图天然聚焦于程序出错时的特定执行路径和交互行为, 为后续的缺陷定位过程提供了高度相关的上下文.

(2) 缺陷位置并行探索

现有的基于 LLM 的缺陷定位方法在定位过程中只探索单条决策路径, 失去了探索其他决策路径的机会, 这可能会降低 LLM 寻找到真实缺陷位置的成功率. 为解决这一问题, 本文提出一种基于 LLM 的缺陷位置并行探索算法. 具体而言, 在获得程序动态调用图 G 之后, 本文将缺陷定位任务建模为 LLM 在图 G 上的并行探索问题. 在某个推理步骤, LLM 可以基于其多种决策分裂出不同的搜索路径 $p \in P$, 每个搜索路径并行地执行定位过程, 以提高成功识别真实缺陷位置的成功率.

每个独立的搜索路径由一个基于 LLM 的调试智能体 (debugging agent) 所驱动, 该智能体的核心是一个具备函数调用 (有时也被称作工具调用) 能力的 LLM. 智能体的行为不局限于生成静态的文本回复, 而是能够主动与外部环境 (即程序动态调用图 G) 交互, 以获取决策所需的关键信息. 如图 2 所示, 一个调试智能体的运行遵循一个迭代式的“观测-行动”周期, 在观测阶段, 智能体根据当前状态 $s_t = (C, F_{\text{fail}}, H_t)$ 推理得出下一步需要采取的动作 a_t. 其中 $H_t = (a_0, o_1, a_1, o_2, \ldots, a_{t-1}, o_t)$ 表示 LLM 与环境交互的历史记录, 即一个“观测-动作”序列, 会随着定位过程的进行而不断得到扩充. 在行动阶段, 智能体可能会终止搜索路径或继续调用预定义函数执行动作 a_t, 函数的返回结果将作为新的观测 o_{t+1} 用于更新智能体的状态, 即 $s_{t+1} = (C, F_{\text{fail}}, H_t \cup \{a_t, o_{t+1}\})$.

● 预定义函数: 智能体可以主动调用预定义的函数接口来查询程序动态调用图G. 这些函数是 LLM 感知和探索代码库的“传感器”. 我们为智能体提供了一个预定义函数集$\mathcal{F}$, 包含 3 个函数.

(1) 获取被覆盖类 (get_covered_classes): 无需指定参数, 该函数会返回程序动态调用图G中所有被覆盖的类节点$V'_C \subseteq V_C$的名称, 从而大大压缩了 LLM 的搜索空间. 由于该函数对定位过程的基础性贡献, 其默认在搜索路径的初始阶段被调用.

(2) 获取类中被覆盖的 method (get_covered_method_ids_for_class): 给定某个类的名称 (类的唯一标识, 如 com.example.MyClass), 该函数会返回这个类中被失败测试用例所覆盖的所有 method 的 ID.

(3) 获取 method 源码 (get_method_code_for_id): 给定某个 method 的 ID (method 的唯一标识, 如 com.example.MyClass.myMethod<5-15>), 该函数会返回这个 method 的完整源代码.

● 动作种类: 如图 2 所示, 在观测阶段, LLM 有可能通过生成文本内容的方式产生两类动作决策.

(1) 调用函数以获取上下文: 基于当前的观测, LLM 生成一段结构化的思考文本, 用于总结已有线索、形成假设, 并分析下一步需要获取的上下文 (如图 2 中的 LLM 思考样例“I will analyze the code of this method to understand how the legend items are obtained and if there could be potential source of test failure”). 在此基础上, LLM 将决策下一步需要采取的行动, 返回 JSON 格式的函数调用信息.

(2) 停止调用函数并总结缺陷根因: 当 LLM 智能体根据一系列的观测和推理, 认为其已经定位到缺陷根因时, 它将生成一个包含缺陷原因分析的最终结论, 并停止继续发起新的函数调用, 从而终止当前搜索路径. 如图 2 中的 LLM 思考过程所示, 一个智能体可能在某步迭代中得出结论“So, this misleading condition in getLegendItems early return is likely the root cause of failing assertion”, 标志着其所在搜索路径的成功终止. 特别地, 在每条搜索路径终止后, PRIME 会引入一个额外的总结步骤, 使用 LLM 对整个定位过程进行回顾, 并枚举所有可能存在缺陷的 method ID.

● 回复采样: 在软件调试的实践中, 开发人员面对同一缺陷, 往往会基于自身的经验和直觉提出不同的诊断假设, 并沿着不同的路径展开调查, 这种思维方式的多样性是高效解决复杂问题的关键[27]. 为了模拟这种并行的探索性思维, PRIME 通过调控 LLM API 的两个核心参数温度 (temperature, T) 和回复采样次数 N_r 来实现对回复结果的多次采样, 从而让 LLM 在面对缺陷信息时, 能够生成一组多样化的、高质量的探索策略.

(1) 温度参数 T 用于控制模型生成文本时的随机性程度. 当 T 趋近于 0 时, 模型变得高度确定性, 几乎总是提供相同的回复, 输出趋向单一、保守. 反之, 当设定一个较高的温度 (例如, $T \in [0.7, 1.2]$), 那些原本概率较低的词元有更大的机会被选中, LLM 的回复将更加多样. 这在缺陷定位的场景下至关重要, 因为它可以鼓励 LLM 跳出最显而易见的标准思路, 转而探索更具创造性的思考路径和行动决策. 回复采样次数参数 N 则直接指定了对于同一次输入请求, 需要 LLM 独立生成多少个不同的回复.

(2) 在 PRIME 框架中, 每当 LLM 需要根据当前状态推理出下一步的行动决策时, 将 API 的 N_r 参数设置为一个大于 1 的整数 (例如, $N_r = 5$), 同时将温度参数 T 调至一个较高的水平. LLM 将返回一个包含 N_r 个独立且各不相同的回复集合 $\mathcal{R} = \{r_1, r_2, \ldots, r_{N_r}\}$. 此回复集合构成了多样性驱动的动作选择机制的输入源. 通过对 LLM 的回复进行多次采样, 可以将单一的搜索路径扩展成为多条不同的路径, 从而显著拓宽缺陷定位的搜索广度, 避免单一路径的局限性, 极大地提升找到真正缺陷位置的可能性.

● 多样性驱动的动作选择: 回复采样可以为 LLM 提供多样性的行为决策, 但仍无法避免生成相似的决策结果. 为了尽可能防止多个智能体走上雷同的搜索路径, 从而最大化并行探索的效益, PRIME 引入一种多样性驱动的动作选择机制 (diversity-driven action selection, DDAS), 旨在提升搜索过程的广度与效率. 该机制的核心思想是, 在 LLM 进行路径扩展时, 优先选择与已探索方向差异性最大的动作, 从而避免在相似的路径上进行冗余探索. 为此, DDAS 构建一个共享的动作决策池, 并利用语义表征与相似度计算技术对候选动作进行动态排序.

为了记录和利用整个搜索过程中的决策历史, DDAS 引入了一个共享的动作决策池 P, 用于存储所有已被选择用于扩展搜索路径的动作的语义表征. 初始状态下, 动作池 P 为空集. 每当一个新动作被选中后, 其对应的向量化表示将被添加入池中, 从而动态地丰富和更新已探索的决策空间. 形式化地, 动作池 P 可以表示为 $P =$

$\{e_1, e_2, \ldots, e_n\}$, 其中 e_i 是第 i 个被选入池中的动作的嵌入向量 (embedding vector), n 是池中当前动作的总数.

在搜索的每一个决策节点, 首先利用 LLM 采样生成一个包含 k 个候选动作的集合, 记作 $A_{\text{cand}} = \{a_1, a_2, \ldots, a_k\}$. 为了量化这些动作的语义信息, 采用预训练的嵌入模型 (embedding model)[28], 将每个候选动作 a_j 映射为一个高维向量 v_j. 对于一个候选动作 a_j, 其与决策池 P 的相似度得分 $S(a_j, P)$ 定义为该动作的嵌入向量 v_j 与池中所有动作向量 e_i 的最大余弦相似度:

$$S(a_j, P) = \max_{e_i \in P} CosineSimilarity(v_j, e_i) \tag{10}$$

在获得所有候选动作的相似度得分后, 对动作集 A_{cand} 进行重排序, 将相似度更低的动作排在前面, 从而确保与历史决策差异性最大、最具新颖性的动作获得最高的探索优先级. 最后, 选择排名最高的动作 a^* 作为本次扩展的最优动作, 用于扩展当前搜索路径. 为了确保未来的决策能够考虑到本次选择, 使用最优动作的嵌入向量 $v^* = \mathcal{F}(a^*)$ 对动作决策池进行更新:

$$P \leftarrow P \cup \{v^*\} \tag{11}$$

通过以上步骤的循环执行, DDAS 能够持续地引导搜索过程向未被充分探索的方向进行, 从而提升搜索的覆盖范围和发现隐蔽缺陷位置的可能性.

3.3 使用节点重要性评估方法对候选缺陷位置进行排序

在通过缺陷位置并行探索获取候选缺陷位置节点集后, 一个核心挑战随之出现: 并行探索策略可能导致候选缺陷位置众多, 从而使得用户难以判断真实的缺陷位置. 为了提升缺陷定位的准确性, 需要设计一种有效的评估机制, 对这些候选位置进行重要性量化与排序, 从而引导开发者优先审查最可疑的程序单元. 本节提出一种融合 LLM 自一致性与图论中心性的混合式节点重要性评估方法, 其核心在于从语义和结构两个维度综合考量每个候选节点在缺陷场景中的关键程度. 评估方法的流程主要包括基于最短路径的子图重塑、节点重要性计算以及线性加权融合与最终排序这 3 个阶段.

● 基于最短路径的子图重塑. 如前文所述, 完整的程序动态调用图 $G = (V, E)$ 虽然精确地描绘了失败测试用例的执行全貌, 但对于分析多个候选缺陷位置之间的内在关联而言, 其信息冗余度依然较高. 此外, 候选缺陷方法节点可能在图中并不直接相连, 需要对这些节点之间的空间结构关系进行恢复. 为了在分析中聚焦于最关键的交互路径, PRIME 首先对图进行重塑, 构建一个高度相关的聚焦子图 G_{sub}.

该重塑过程以初始获得的候选缺陷方法节点集 $V_{\text{cand}} \subseteq V_M$ 为基础. 本文的主要假设是, 如果多个程序单元共同导致一个缺陷, 那么它们在动态调用图上的交互路径往往最短、最直接. 因此, 连接这些候选节点的最短路径集合, 能够以最高效的方式勾勒出它们之间的潜在协作或依赖关系. 具体而言, 对于候选节点集 V_{cand} 中的任意两个不同节点 v_i、$v_j \in V_{\text{cand}}$, 在原图 G 中计算它们之间的所有最短路径. 令 $SP(v_i, v_j)$ 代表连接 v_i 和 v_j 的所有最短路径上的节点集合. 聚焦子图的节点集 V_{sub} 是候选节点集和中间节点集的并集 $V_{\text{sub}} = V_{\text{cand}} \cup V_{\text{cen}}$, 其中中间节点集 V_{cen} 是所有候选节点对之间最短路径节点的并集 $V_{\text{cen}} = \bigcup_{v_i, v_j \in V_{\text{cand}}, i \neq j} SP(v_i, v_j)$, 子图的边集 E_{sub} 包含原图 G 中连接 V_{sub} 内任意两个节点的所有边. 经过重塑, 生成的聚焦子图极大地压缩了分析空间, 剔除了大量与候选节点核心交互无关的旁路信息, 为后续节点重要性的精确计算奠定了坚实基础.

● 节点重要性计算. 在获得聚焦子图 G_{sub} 后, PRIME 采用两种互补的策略计算图中各节点的重要性, 此设计旨在结合 LLM 的语义洞察力与图算法的结构分析能力, 以期获得更鲁棒、更全面的评估结果.

基于 LLM 自一致性的节点重要性计算. 在缺陷位置并行探索过程中, LLM 可能会在多个推理路径中反复指向某些特定的程序单元. 这种现象可以被视为一种“自一致性 (self-consistency)”[24]的体现. 一个节点被 LLM 提及的频率越高, 在某种程度上表明模型认为该节点与缺陷相关的可能性越强. 因此, 可以将这种出现频率转化为节点的初始重要性权重. 形式化地, 该计算过程定义如下.

设 $\mathcal{T}$ 为失败测试用例的集合, 其总数为 $N_{\mathcal{T}} = |\mathcal{T}|$. 对于每一个失败测试用例 $t \in \mathcal{T}$, LLM 通过分析可生成一个或多个独立的搜索路径. 设 $\mathcal{P}_t$ 为由测试用例 t 生成的搜索路径集合, 其路径数量为 $M_t = |\mathcal{P}_t|$. 对于每一条搜索路径 $p \in \mathcal{P}_t$, 该路径会最终指向一个候选缺陷节点的集合. 令 $V_{t,p}$ 为该路径所产生的候选节点集, 候选节点数量为

$K_{t,p}=\left|V_{t,p}\right|$. 基于此 3 层结构, 当一个具体的节点 v 在测试用例 t 的搜索路径 p 中被识别为候选缺陷 (即 $v\in V_{t,p}$) 时, 该次枚举所贡献的权重被定义为:

$$w(v,t,p)=\frac{1}{N\cdot M_t\cdot K_{t,p}} \tag{12}$$

公式 (12) 体现出重要性的逐级稀释. 在顶层, 总重要性被所有 $N_{\mathcal{T}}$ 个测试用例均分; 在中间层, 每个测试用例 t 的重要性, 被其产生的所有 M_t 条搜索路径均分; 在底层, 每条搜索路径 p 的重要性被其识别出的所有 $K_{t,p}$ 个候选节点均分. 在此基础上, 一个节点的最终重要性, 是其在所有测试用例的所有搜索路径中出现时, 所获权重之和. 因此, 对于聚焦子图 G_{sub} 中的任意一个候选缺陷节点 v, 其基于 LLM 自一致性的最终重要性得分 $S_{\text{LLM}}(v)$ 计算为:

$$S_{\text{LLM}}(v)=\sum_{t\in\mathcal{T}}\sum_{p\in\mathcal{P}_{\mathcal{T}}}\mathbb{I}(v\in V_{t,p})\cdot w(v,t,p)=\sum_{t\in\mathcal{T}}\sum_{p\in\mathcal{P}_{\mathcal{T}},\,\text{s.t. }v\in V_{t,p}}\frac{1}{N\cdot M_t\cdot K_{t,p}} \tag{13}$$

其中, $\mathbb{I}(\cdot)$ 是指示函数, 用于确保只有当节点 v 确实出现在路径 p 的候选集 $V_{t,p}$ 中时, 才会累加其对应的权重. 通过这种方式, 该方法不仅考虑了节点出现的频率, 更重要的是, 它奖励了那些在更“确定”的分析路径 (即 M_t 和 $K_{t,p}$ 较小) 中被识别出的节点, 从而为重要性评估提供了更精确、更具分辨力的语义层面依据.

● 基于节点中心性算法的重要性计算. 与 LLM 的语义分析不同, 图中心性算法从网络拓扑结构出发, 客观地评估节点在图中的结构性重要程度. 在程序调用图中, 一个结构上重要的节点往往是信息流动的枢纽或控制路径的关键. 若此类节点发生故障, 其影响范围更广, 更容易导致可观测的系统级失效. 基于此假设, 本文选用介数中心性 (betweenness centrality) 算法来量化节点的结构重要性. 介数中心性衡量一个节点出现在图中其他节点对之间最短路径上的频率. 一个节点的介数中心性越高, 说明它在程序动态调用路径中扮演“桥梁”的角色越关键. 对于 G_{sub} 中的一个节点 v_i, 其介数中心性 $S_{\text{centra}}(v_i)$ 计算公式如下:

$$S_{\text{centra}}(v_i)=\sum_{s\neq v_i\neq t,\ s,t\in V_{\text{sub}}}\frac{\sigma_{st}(v_i)}{\sigma_{st}} \tag{14}$$

其中, σ_{st} 是从节点 s 到节点 t 的最短路径总数, 而 $\sigma_{st}(v_i)$ 是这些路径中经过节点 v_i 的数量. 该指标能够独立于 LLM 的判断, 纯粹从程序执行流的结构特征中识别出关键控制节点, 为重要性评估提供了结构层面的坚实证据.

● 线性加权融合与最终排序. 上述两种方法各有侧重: S_{LLM} 侧重于语义, 而 S_{centra} 侧重于拓扑结构. 单独使用任何一种方法都可能存在偏差. 因此, 将二者进行有效融合, 是实现优势互补、提升评估准确性的关键. 本文采用线性加权融合策略, 生成最终的重要性分数. 在融合前, 为消除量纲差异, 首先对两类得分进行归一化, 将其映射到 $[0,1]$ 区间. 令归一化后的分数为 $\widehat{S_{\text{LLM}}}$ 和 $\widehat{S_{\text{centra}}}$, 对于子图中的任一节点 v_i, 其最终重要性分数 $S_{\text{Final}}(v_i)$ 的计算公式为:

$$S_{\text{Final}}(v_i)=\alpha\cdot\widehat{S_{\text{LLM}}}(v_i)+(1-\alpha)\cdot\widehat{S_{\text{centra}}}(v_i) \tag{15}$$

其中, $\alpha\in[0,1]$ 是一个超参数, 用于调节两种重要性来源的权重.

最终, 根据所有候选节点的 S_{Final} 分数进行降序排列, 生成候选缺陷位置的优先级排序列表. 此列表反映了各程序元素为真正缺陷位置的可能性大小, 能够有效地指导开发人员进行后续的审查和修复工作.

4 实验分析

4.1 实验数据集

为了全面评估 PRIME 的效果, 本文在公开缺陷数据集 Defects4J[29]上进行实验, 表 1 给出了数据集所对应的详细项目和缺陷数量信息. Defects4J 是一个广受欢迎的基准数据集, 专为推动软件工程领域的自动化调试、缺陷定位和程序修复等研究而设计. 该数据集收集了数百个源自真实世界 Java 项目中的缺陷及开发者修复版本, 并提供了一整套完善的基础设施, 极大地便利了相关研究的开展. 由于本文主要关注 method 级缺陷定位, 我们将开发者修改的 method 作为真实的缺陷位置, 并剔除了部分无法被定位的样本, 包括未修改 method、新增 method、缺陷 method 未被失败测试用例所覆盖等. 最终, 本实验选取的数据集包含来自 16 个 Java 项目的 691 个真实软件缺陷.

表 1 实验数据集

项目	项目全称	样本数	选取样本数	项目代码行数 (k)
Chart	jfreechart	26	25	207
Closure	closure-compiler	176	166	123
Lang	commons-lang	65	61	56
Math	commons-math	106	99	164
Mockito	mockito	38	35	20
Time	joda-time	27	25	61
Cli	commons-cli	40	37	4
Codec	commons-codec	18	15	5
Collections	commons-collections	28	3	61
Compress	commons-compress	47	44	30
Csv	commons-csv	16	15	2
Gson	gson	18	14	10
JacksonCore	jackson-core	26	19	24
JacksonDatabind	jackson-databind	112	39	65
JacksonXml	jackson-dataformat-xml	6	6	6
Jsoup	jsoup	93	88	4
总计		842	691	965

4.2 评价指标

本文主要关注方法级粒度的缺陷定位任务, 采用以下 3 种评价指标[11]来评估各个缺陷定位方法的性能.

(1) Top-*N* 指标. Top-*N* 是一种直观的评估方法, 如果缺陷定位工具给出的前 *N* 个候选 method 中包含了真实的缺陷 method, 就认为本次定位是成功的. 已有研究表明, 超过 80% 的开发者在使用自动化缺陷定位工具时, 通常只愿意检查工具推荐的前 5 个可疑程序位置[30]. 因此, Top-*N* 准确率直接衡量了方法是否能在开发者耐心阈值内提供有效帮助, 是衡量其实用价值的关键. 基于以上考量, 本文主要采用 $N \in \{1,3,5\}$ 用于评估不同方法的缺陷定位能力, 在第 4.6 节 (参数影响分析) 中, 本文额外展示了 PRIME 的 Top-10 准确率, 以更全面地分析不同参数设置对其缺陷定位能力的影响.

(2) 平均倒数排名 (mean reciprocal rank, *MRR*). *MRR* 着重于评估排序结果中第 1 个正确答案的定位效率. 对于每一个缺陷定位任务提供的缺陷定位结果, *MRR* 首先确定第 1 个缺陷 method 的排名, 并取其倒数作为该任务的倒数排名, 最终的 *MRR* 值是所有缺陷定位任务的倒数排名的平均值. 该指标的优势在于其简洁性和对首位命中情况的敏感性, 它能有效地衡量方法将缺陷位置快速推至排序列表顶部的能力, 尤其适用于那些只需找到一个缺陷位置即可开始修复的场景. *MRR* 的计算公式为:

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i} \tag{16}$$

其中, $|Q|$ 表示缺陷总数, $rank_i$ 表示第 i 个缺陷定位结果中第 1 个正确缺陷 method 的排名.

(3) 平均准确率均值 (mean average precision, *MAP*). *MAP* 是一种更为全面的排序评估指标, 它综合考虑了所有正确答案的排名. 对于每个缺陷定位任务, *MAP* 首先得到该任务的平均准确率 (average precision, *AP*). *AP* 会计算每一个被正确识别的缺陷 method 在当前排名列表中的准确率, 并将所有这些准确率求平均. 最终的 *MAP* 值是所有缺陷的 *AP* 值的平均值. 相较于 *MRR*, *MAP* 能够更全面地反映方法的整体性能, 高 *MAP* 值表明该方法能够有效地将所有相关的缺陷 method 都排在靠前的位置. *MAP* 的计算公式为:

$$MAP = \frac{1}{|Q|}\sum_{i=1}^{|Q|}AP_i,\ AP = \frac{\sum_{k=1}^{n}(P@k\times rel(k))}{\text{定位到的缺陷 method 总数}} \tag{17}$$

其中, $P@k$ 表示前 k 个结果中的准确率, 如定位结果列表中找到的前 3 个结果中有 2 个为缺陷 method, 则 $P@3 =$

2/3. $rel(k)$ 是一个指示函数, 如果第 k 个结果是缺陷 method, 则 $rel(k) = 1$, 否则为 0. n 是排名列表的长度.

4.3 实验设置

为了控制缺陷定位过程的整体开销, 本文设置了多个参数来控制 PRIME 的行为. 默认参数设置包括: 在较高的温度下, LLM 更可能产生不同的决策, 从而提升搜索路径的多样性, 本文默认设置 LLM 温度参数 $T = 1.2$; 由于更高的回复采样次数会导致更多的 LLM 词元开销, 因此本文默认设置 LLM 回复采样次数 $N_r = 5$; 理论上, 更高的搜索路径数量可以获得更好的缺陷定位性能, 为将开销控制在合理范围内, 本文默认设置最大搜索路径数量 $N_p = 6$; 最大分支因子 F_b 是 LLM 在单次决策中最大可以采纳的决策数量, 即每次节点扩展时最大可以产生的新搜索路径数量, 可用于控制搜索树的宽度, 本文默认设置 $F_b = 2$; 线性融合参数 α 主要用于控制节点重要性评估中来自基于 LLM 自一致性的节点重要性和来自基于中心性的节点重要性的比重, 为避免过拟合问题, 本文未对此参数进行调优, 默认设置为 $\alpha = 0.5$.

本文选择 jina-embeddings-v4[31]模型作为 PRIME 默认的向量模型. 为了对不同缺陷定位方法进行公平的对比, 同时考虑到计算成本, 本文统一使用 GPT-4o mini[32]作为所有缺陷定位方法的 LLM. 特别地, 由于 FlexFL[13]专门为开源 LLM 设计, 因此本文保留该方法的默认设置, 采用其在 Llama-3-8B[33]模型下的实验结果. 在方法实现上, 本文使用原有基线方法提供的源代码进行实验, 由于 Agentless[11]的初始版本仅支持对 Python 项目进行缺陷定位, 因此本文在不改变其原有实现逻辑的前提下对其进行了适配, 使其支持 Defects4J 数据集中的缺陷定位.

4.4 对比方法与实验结果

为了评估基于并行探索的大模型缺陷定位增强方法 PRIME 的有效性, 本文选取了 3 种最先进的基于 LLM 的缺陷定位技术 (Agentless[12]、AutoFL[13]和 FlexFL[11]) 作为对比方法. Agentless 是一种面向固定工作流的程序修复技术, 其中包含由 3 个阶段组成的固定缺陷定位流程: 首先, 构建软件仓库中所有代码文件与文件夹的目录结构, 让 LLM 通过初始缺陷信息定位至可能存在问题的文件; 其次, 将代码文件转化为简洁的“代码骨架”, 只保留类头 (header)、域 (field) 定义、方法签名 (signature) 等, 令 LLM 指出可能存在问题的程序元素; 最后, 使用 LLM 对定位结果进行精化, 确认需要被修复的具体编辑位置. AutoFL 与本文提出的技术同源, 是一种面向 LLM 自主决策的定位方法, 给定初始缺陷信息, 此方法允许 LLM 通过自主调用函数从软件仓库中获取上下文信息, 最终推理得出缺陷根因位置. 需要强调的是, 与 AutoFL 不同, 本文提出的技术克服了此类方法搜索路径单一的局限性, 旨在进一步提高缺陷定位的准确率. FlexFL 的定位流程包含两个阶段, 其首先利用基于程序频谱的缺陷定位技术缩小缺陷代码的搜索空间, 生成一个可疑方法的候选列表. 随后, 该方法利用 LLM 深入检查第 1 阶段建议的方法代码片段, 以完善缺陷定位结果. 该方法灵活地利用了多种缺陷相关信息, 其特色在于通过构建基于开源 LLM 的智能体来定位缺陷.

对比方法与本文方法在 Defects4J 数据集上的缺陷定位结果如表 2 所示. 与其他 3 种基于 LLM 的缺陷定位方法相比, 本文提出的方法 PRIME 有着显著优势, 在大部分缺陷项目上都获得了最好的效果. 总的来看, 本文方法 PRIME 的表现最好, 其可以将 78.4% (542/691) 的缺陷排在前 5 名, 其综合 *MRR* 值和 *MAP* 值分别为 0.600 5 和 0.596 6; FlexFL 紧随其后, 可以找到 71.9% (497/691) 的缺陷; AutoFL 可以将 59.5% (411/691) 的缺陷定位至 Top-5; Agentless 表现最差, 只能定位到 31.5% (218/691) 的缺陷. 相较于 FlexFL、AutoFL 和 Agentless, PRIME 在 Top-1 指标上的性能分别提升了 18.6%、21.2% 和 141.6%, 在 Top-5 指标上的提升幅度分别达到了 9.1%、31.9% 和 148.6%. 观察单个缺陷项目中的定位结果, 可以看出除了在少部分项目 (如 Math 和 JacksonDatabind) 上 PRIME 的表现不及 FlexFL 之外, 本文方法在绝大多数项目中都表现出了更好的定位性能. 经过仔细分析, 我们发现 FlexFL 首先会通过结合基于信息检索的缺陷定位 (IRFL)、基于程序频谱的缺陷定位 (SBFL) 等多种其他缺陷定位方法来对缺陷位置进行初步的筛选, 然后才会使用 LLM 对候选的缺陷位置实施进一步的分析. 相比之下, PRIME 更关注于直接利用 LLM 本身的能力来进行缺陷定位, 这有可能是 PRIME 在某些项目上的效果不及 FlexFL 的主要原因. 此外, 我们还发现 PRIME 在某些缺陷项目中的效果提升非常显著, 例如在 Closure 项目上, PRIME 在 Top-1、Top-3、Top-5 指标上比 AutoFL 定位到的缺陷数量分别增加了 15、34 和 40 个; 然而, 在另一些缺陷项目中 PRIME 的效

果提升幅度较小, 例如在 Math 项目上, PRIME 在 3 个指标上分别只比 AutoFL 多定位了 3、7 和 9 个缺陷. 通过对这些项目进行分析, 本文发现不同项目在缺陷复杂程度上的差异是造成此现象的主要原因. 相较于 Math 项目, Closure 等项目中的缺陷往往更为复杂, 涉及更多软件组件之间的相互协作, 因而缺陷位置也更为隐蔽, 在这种情况下, 本文提出的并行探索方法可以有效地增大 LLM 的搜索范围, 从而显著地提高了定位的准确率.

表 2 PRIME 与其他缺陷定位方法性能比较

缺陷项目	缺陷数量	Top-1				Top-3				Top-5				*MRR*				*MAP*			
		Au	Ag	Fl	PR	Au	Ag	Fl	PR	Au	Ag	Fl	PR	Au	Ag	Fl	PR	Au	Ag	Fl	PR
Chart	25	14	7	14	**16**	16	10	**20**	18	17	11	**22**	18	0.6033	0.3433	0.6780	**0.6800**	0.6033	0.3278	0.6113	**0.6800**
Closure	166	46	16	40	**61**	62	24	59	**96**	63	25	70	**103**	0.3208	0.1210	0.3042	**0.4805**	0.3213	0.0916	0.2890	**0.4736**
Lang	61	44	19	41	**46**	51	19	51	**53**	52	22	**55**	53	0.7958	0.3238	0.7587	**0.8194**	0.7958	0.2951	0.7026	**0.8194**
Math	99	56	35	**63**	59	70	46	**80**	77	73	47	**86**	82	0.6076	0.3914	**0.6934**	0.6585	0.6069	0.3738	0.6474	**0.6540**
Mockito	35	21	15	15	**23**	22	16	20	**23**	22	16	23	**27**	0.5766	0.4429	0.4784	**0.6473**	0.5766	0.2418	0.4593	**0.6358**
Time	25	13	3	13	**16**	**17**	4	16	**17**	17	5	**18**	**18**	0.6171	0.1594	0.6250	**0.7186**	0.6171	0.1385	0.5421	**0.7186**
Cli	37	18	12	19	**26**	22	16	21	**28**	23	16	21	**30**	0.6286	0.4362	0.6250	**0.8586**	0.6286	0.3832	0.5265	**0.8430**
Codec	15	8	4	8	**9**	9	5	**11**	10	9	6	**11**	10	0.6538	0.3487	0.7051	**0.7308**	0.6538	0.2897	0.6423	**0.7308**
Collections	3	0	**1**	**1**	0	0	**1**	**1**	**1**	0	1	**2**	1	0	**0.3333**	0.3000	0.1250	0	**0.3333**	0.3000	0.1250
Compress	44	19	12	20	**23**	27	16	34	**35**	27	17	36	**39**	0.5232	0.3353	0.6140	**0.6836**	0.5252	0.2956	0.5820	**0.6836**
Csv	15	12	5	5	**13**	13	7	10	**14**	13	8	13	**14**	0.8929	0.4464	0.5857	**0.9643**	0.8929	0.375	0.5643	**0.9643**
Gson	14	8	3	**10**	9	8	4	**11**	10	8	4	**12**	11	0.4706	0.2452	**0.6294**	0.5706	0.4706	0.2417	**0.5905**	0.5706
JacksonCore	19	12	7	14	**15**	17	7	20	**22**	18	8	22	**23**	0.5697	0.3816	0.6436	**0.7128**	0.5714	0.3243	0.5568	**0.7038**
JacksonDatabind	39	12	4	**28**	22	15	4	**42**	35	18	4	**49**	43	0.1387	0.1026	**0.3534**	0.2943	0.1387	0.021	**0.3342**	0.2943
JacksonXml	6	1	1	**3**	2	1	1	**4**	2	1	1	**4**	2	0.1667	0.1667	**0.5833**	0.3611	0.1667	0.1667	**0.4833**	0.3611
Jsoup	88	37	17	34	**49**	49	24	48	**65**	50	27	53	**68**	0.5187	0.239	0.5053	**0.6952**	0.5198	0.1810	0.4685	**0.6931**
总计	691	321	161	328	**389**	399	204	448	**506**	411	218	497	**542**	0.4761	0.2701	0.4798	**0.6005**	0.4765	0.2249	0.5184	**0.5966**

注: Au表示AutoFL, Ag表示Agentless, Fl表示FlexFL, PR表示PRIME, 加粗数据用于标记在该指标上表现最好的缺陷定位方法

为进一步理解不同缺陷定位方法的能力差异, 本文对不同方法的缺陷定位 Top-5 结果重叠关系进行了分析, 结果如后文图 3 所示. PRIME 的缺陷定位能力最强, 可以定位 69 个其他方法无法找到的缺陷. 相比之下, Agentless 仅能独自将 7 个缺陷定位至 Top-5, 而 AutoFL 只有 2 个. 需要注意的是, 虽然 Agentless 的整体性能不如 AutoFL (见表 2), 但其可以独自定位到的缺陷数量比 AutoFL 多 5 个. 这一现象可能与不同方法在定位流程上的差异有关. PRIME 与 AutoFL 均属于面向 LLM 自主决策的定位方法, 而 Agentless 则可被分类为面向固定工作流的定位方法, 因此 PRIME 与 AutoFL 的重叠程度更高. 这说明根据专家知识人为设定的工作流程在某些情况下可能会优于 LLM 自主规划的缺陷定位流程. 特别地, 本文注意到 FlexFL 可以独立定位 68 个缺陷, 展现出与 PRIME 在定位能力上的互补性. 通过对比 FlexFL 和 PRIME 的缺陷定位结果, 本文发现导致这一现象的主要原因在于 FlexFL 额外融合了来自其他缺陷定位方法的中间结果, 从而可以定位一部分大模型难以发现的缺陷. 例如, FlexFL 利用了开发者提供的缺陷报告信息, 通过基于信息检索的缺陷定位方法 BoostN[34]来预先缩小大模型的搜索范围. 由于部分缺陷报告中明确指出了有错误的 method, 因此 FlexFL 会更容易定位此类缺陷. 总体而言, 图 3 中 4 种缺陷定位方法一共可以将 93.8% (691 个缺陷中的 648 个) 的缺陷排在前 5 位. 这说明通过结合其他缺陷定位方法或缺陷信息, 有望进一步提高本文方法的效果.

4.5 消融实验

为了充分探讨本文所提出的设计决策对 PRIME 性能的影响状况, 本文构造了以下 5 种方法变体.

- 不使用并行探索: 敲除第 3.2 节介绍的缺陷位置并行探索方法 (即将最大搜索路径数量 N_p 设置为 1), 不改变第 3.3 节的缺陷位置重排序流程. 这一操作将使得 PRIME 回退为一种普通的面向 LLM 自主决策的定位方法.
- 不使用动作选择: 敲除第 3.2 节中介绍的多样性驱动的动作选择机制, 不改变第 3.3 节的缺陷位置重排序流程. 这一操作将降低 LLM 在进行下一步行为决策时选择的多样性.
- 不使用缺陷位置排序: 完全敲除第 3.3 节中介绍的缺陷位置重排序方法, 只对 LLM 产生的多个候选缺陷位

置进行去重操作.

● 不使用基于 LLM 自一致性的节点重要性计算: 保留第 3.3 节中介绍的缺陷位置重排序方法, 但敲除排序方法中基于 LLM 自一致性的节点重要性计算部分.

● 不使用基于节点中心性算法的重要性计算: 保留第 3.3 节中介绍的缺陷位置重排序方法, 但敲除排序方法中基于节点中心性算法的重要性计算部分.

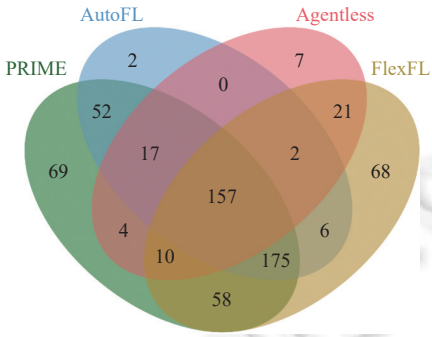


图 3 缺陷定位 Top-5 结果重叠关系韦恩图

使用默认参数设置在同一数据集上评估每种方法变体的缺陷定位效果, 结果如表 3 所示. 在敲除并行探索机制后, PRIME 在 Top-1、Top-3 和 Top-5 指标上的性能分别下降了 27、47 和 62, 在 *MRR* 和 *MAP* 指标上的性能分别下降了 0.052、0.049, 这充分证实了本文假设的正确性, 即通过让 LLM 搜索不同的决策路径可以有效地提高 LLM 找到缺陷位置的成功率. 实验结果显示, 动作选择机制虽然对 PRIME 性能有着积极贡献, 但贡献程度较小, 这一现象有可能通过设置更大的回复采样次数 N_r 和最大搜索路径数量 N_p 得到改善. 缺陷位置排序是影响 PRIME 效果的一个关键环节, 敲除该部分将使得定位方法在 Top-1、Top-3 和 Top-5 指标上的性能分别下降 171、84 和 49. 特别地, 缺陷位置排序中使用的两种节点重要性计算方法都产生了积极贡献, 而基于 LLM 自一致性的节点重要性计算发挥的作用大于基于节点中心性算法的重要性计算.

表 3 消融实验结果

方法名称	Top-1		Top-3		Top-5		*MRR*		*MAP*	
不使用并行探索	362	↓27	459	↓47	480	↓62	0.5485	↓0.052	0.5473	↓0.049
不使用动作选择	383	↓6	492	↓14	533	↓9	0.5881	↓0.012	0.5846	↓0.012
不使用缺陷位置排序	218	↓171	422	↓84	493	↓49	0.4355	↓0.165	0.4300	↓0.167
不使用基于LLM自一致性的节点重要性计算	230	↓159	431	↓75	499	↓43	0.4483	↓0.140	0.4445	↓0.152
不使用基于节点中心性算法的重要性计算	369	↓20	493	↓13	525	↓17	0.5758	↓0.025	0.5700	↓0.027

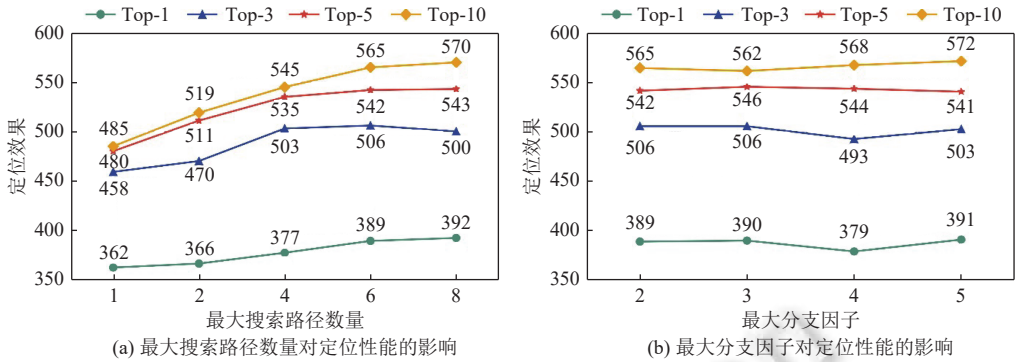
注: 使用↓标注的数据为该变体方法相对于原始方法的性能下降幅度

总体而言, 所有方法变体相对于本文提出的完整方法 PRIME 都有着不同程度的性能下降, 说明本文设计的每个组件都有助于增强 LLM 的缺陷定位能力.

4.6 参数影响分析

为了充分探讨本文方法 PRIME 受不同参数设置的影响状况, 我们展示了设置不同最大搜索路径数量 N_p 和最大分支因子 N_b 的情况下, PRIME 性能的变化趋势. 实验在相同数据集上开展, 结果如图 4 所示.

图 4(a) 展示了定位效果随最大搜索路径数量变化的趋势. 从图中可以清晰地看出, 在所有 Top-*N* 指标下, 定位效果基本上都随着最大搜索路径数量的增加而提升. 当路径数量从 1 增加至 6 时, 各项指标的性能提升尤为显著, Top-1 指标从 362 提升至 389, Top-5 指标从 480 提升至 542. 然而, 当路径数量从 6 继续增加至 8 时, Top-5 和 Top-10 指标的性能趋于饱和, 增益非常有限, Top-3 指标的性能甚至出现了轻微的下降 (从 506 降至 500). 这表明更多的搜索路径虽然可能覆盖更广的搜索空间, 但也可能引入冗余或噪声信息, 导致边际效益递减.



(a) 最大搜索路径数量对定位性能的影响 (b) 最大分支因子对定位性能的影响

图 4 PRIME 在不同搜索参数设置下的缺陷定位性能变化

图 4(b) 展示了最大分支因子对定位效果的影响. 从实验结果可以得出, 此参数的变化对定位方法的整体性能影响不大. 在 2–5 的参数范围, 所有 Top-N 指标的得分都保持在一个相对稳定的区间内. 具体而言, Top-5 指标得分始终在 541–546 之间浮动, 变化范围极小. 同样, Top-10 指标的得分也稳定在 562–572 的区间内. 这种平稳的性能表现说明 PRIME 对于最大分支因子参数不敏感, 具有较好的鲁棒性.

以上实验结果表明, 在相对合理的参数配置下, 可以保证 PRIME 方法维持良好的缺陷定位性能.

4.7 计算成本分析

PRIME 方法通过并行探索机制来扩展 LLM 的搜索范围, 从而提高缺陷定位的准确性. 相应的, 这种策略会带来额外的计算开销. 为了全面评估 PRIME 方法的效率, 本文统计了不同基于 LLM 的缺陷定位方法定位每个缺陷所消耗的平均计算成本. 本文通过 LLM 消耗的词元 (token) 数量来衡量计算成本, 将 PRIME 方法在不同搜索路径数量 (N_p = 1, 2, 4, 6, 8) 下的变体与 AutoFL、Agentless 和 FlexFL 等基线方法进行了比较.

表 4 展示了各方法定位每个缺陷的平均输入和输出词元开销. 从表中数据可以看出, AutoFL、Agentless 和 FlexFL 的计算成本相对较低, 平均需要的 LLM 输入 (输出) 词元数皆不超过 30 000 (3 000) 个, 而 PRIME 方法的计算成本相对较高, 且随着最大搜索路径数量的增加而上升. 当最大搜索路径数量从 1 增加至 8 时, PRIME-p8 的输入词元开销达到了 PRIME-p1 的约 8 倍, 而输出词元开销则增加了约 32 倍. PRIME 通过更高的计算开销以换取更好的缺陷定位效果, 本文认为, 随着 LLM 技术的不断进步, 这种开销是可以接受的. 一方面, 随着计算效率的提高, 现代大语言模型服务的价格不断下降. 即便是目前最先进的 DeepSeek-V3[35]模型, 其百万词元输入 (输出) 也仅需要 4 (12) 元人民币, 这意味着使用 PRIME-p6 定位每个缺陷平均只需要约 4 元. 另一方面, 提示词缓存 (prompt caching)[36]技术的出现使得 LLM 无需每次从头进行推理, 可以进一步降低本文方法的计算成本. 对于复杂的软件缺陷, 开发人员通常需要花费数小时甚至数天的时间进行手动调试. PRIME 方法通过可控的额外计算成本, 显著提高了缺陷定位精度, 有助于为软件开发团队节省大量人力资源和时间成本.

表 4 缺陷定位方法定位每个缺陷的平均计算成本统计

输入/输出	AutoFL	Agentless	FlexFL	PRIME-p1	PRIME-p2	PRIME-p4	PRIME-p6	PRIME-p8
输入	27k±43k	21k±14k	17k±10k	92k±188k	191k±382k	380k±791k	576k±1199k	766k±1524k
输出	2161±2890	2161±2890	491±103	5805±9260	47k±76k	94k±157k	142k±239k	189k±309k

注: pN (N = 1,2,4,6,8)表示设置不同最大搜索路径数量 N_p 时的PRIME变体; ±符号后的数值为标准差

4.8 实验结果的局限性

模型泄露问题: Defects4J 数据集可能已经包含在本文使用的 LLM 的训练语料库内, 这种预训练知识可能会对 LLM 的定位性能产生积极影响, 从而导致实验结果存在偏差. 为了尽可能减轻这一影响, 本文在实验中采用了严格的变量控制, 确保所有对比方法 (除 FlexFL 之外) 均使用相同的 LLM. 通过这种方式, 本文的实验结果确保了 PRIME 方法的性能提升源于并行探索和节点重要性评估机制, 而非 LLM 本身对特定数据集的先验知识.

结果泛化性问题: 受限于 LLM API 的调用成本, 本研究仅使用了一个闭源 LLM GPT-4o mini 进行实验. 尽管 GPT-4o mini 在性能上表现出色, 但无法排除其他 LLM (如更大型或不同架构的模型) 在相同任务上的表现差异. 为了促进后续研究和分析, 本文提供了所有实验的完整源码, 包括方法实现、LLM 提示词、参数设置以及数据处理脚本. 这使得其他研究者可以轻松地复现本文的实验, 并在不同的模型和数据集上进行更深入的探索.

5 讨 论

为了深入剖析 PRIME 并行探索机制在方法粒度缺陷定位任务中的实际优势, 本节以 Closure-81 缺陷作为典型案例, 将 PRIME 与现有的基于 LLM 的缺陷定位方法 AutoFL 的工作流程进行对比分析. Closure-81 缺陷的根本原因在于, 当解析器遇到一个未命名函数语句 (unnamed function statement) 时, 本应抛出一个解析错误 (parsing error), 但程序中缺失了相应的处理逻辑. 如图 5 所示, 失败测试用例的堆栈跟踪信息显示, 测试用例 testUnnamedFunctionStatement 在 ParserTest.java 文件中的第 796 行调用 parseError 时触发了测试断言失败. 对于该缺陷, 仅凭初始的缺陷信息难以准确定位缺陷位置, 堆栈跟踪中提供了初始线索, 但其仅指向了断言失败的位置, 而并不包含与缺陷的根因位置 (即 IRFactory 类中的 processFunctionNode() 方法) 相关的任何信息. 因此, 为了找到缺陷根因, 大模型需要从代码仓库中收集更多与缺陷相关的上下文信息, 如失败测试用例覆盖的类和相关 method 的源代码等.

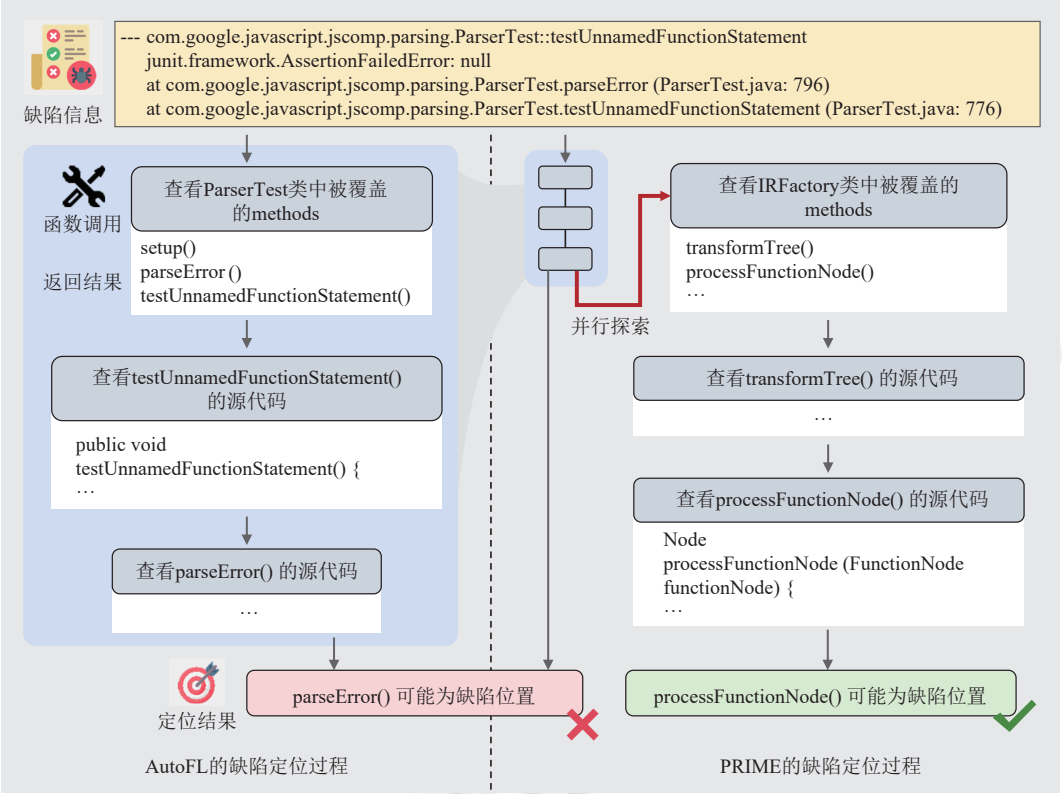


图 5 案例分析

如图 5 左侧所示, AutoFL 的缺陷定位过程为一条单一的推理路径. 其首先从堆栈跟踪信息出发, 通过函数调用获取了测试类 ParserTest 中被覆盖的方法列表, 包括 setup()、parseError() 和 testUnnamedFunctionStatement(). 随后, AutoFL 分别检查了 testUnnamedFunctionStatement() 和 parseError() 的源代码. 由于 parseError() 是堆栈跟踪中最终导致断言失败的 method, 因此 AutoFL 错误地将其判定为缺陷位置. 此案例暴露了 AutoFL 等现有缺陷定位方法的局限性, 此类方法过度依赖堆栈跟踪等最直接的表面线索, 这种线性推理方式很容易被误导至错误的路径, 并

因缺乏替代路径而无法纠正, 最终导致定位失败.

如图 5 右侧所示, PRIME 在处理同一缺陷时采用了截然不同的策略, 虽然其同样从堆栈跟踪信息开始分析, 但 PRIME 并不局限于单一线索, 而是可以生成并维护多条推理路径. 其中, 一条路径可能与 AutoFL 的推理相似, 同样探索了 ParserTest 类并导向了 parseError(). 然而, 如图 5 中红色箭头所示, PRIME 通过对 LLM 的决策进行多次采样, 开辟了一条全新的探索路径. 该路径正确地识别出 IRFactory 类可能与抽象语法树的构建和解析逻辑相关. 沿着这一路径, PRIME 进一步检查了 IRFactory 类中的 method, 并准确定位到 processFunctionNode() 为真正的缺陷位置. 这一案例表明, 通过并行维护多条推理路径, PRIME 避免了过早收敛到由表面线索所导致的局部最优解, 从而显著增大了大模型找到并定位真实缺陷位置的可能性.

6 总 结

基于大语言模型的缺陷定位方法近年来在缺陷定位任务中展现出较好效果. 本文提出了一种基于并行探索的大模型缺陷定位增强方法. 针对现有基于大语言模型的缺陷定位方法仅支持单一决策路径, 在大规模软件系统中定位效果受限等问题, 本文通过设计缺陷位置的并行探索机制扩展大语言模型的搜索范围, 并通过进一步引入基于节点重要性评估方法的候选缺陷位置排序机制对缺陷定位结果进行优化. 通过与其他缺陷定位方法在包含 691 个真实缺陷的数据集上进行比较, 验证了本文所提出的缺陷定位方法可以有效增强大语言模型的缺陷定位性能.

References

[1] Liu NF, Lin K, Hewitt J, Paranjape A, Bevilacqua M, Petroni F, Liang P. Lost in the middle: How language models use long contexts. In: Trans. of the Association for Computational Linguistics, Vol. 12. Cambridge: MIT Press, 2024. 157–173. [doi: 10.1162/tacl_a_00638]

[2] Zhang FJ, Chen B, Zhang Y, Keung J, Liu J, Zan DG, Mao Y, Lou JG, Chen WZ. RepoCoder: Repository-level code completion through iterative retrieval and generation. In: Proc. of the 2023 Conf. on Empirical Methods in Natural Language Processing. Singapore: ACL, 2023. 2471–2484. [doi: 10.18653/v1/2023.emnlp-main.151]

[3] Wu D, Ahmad WU, Zhang DJ, Ramanathan MK, Ma XF. REPOFORMER: Selective retrieval for repository-level code completion. In: Proc. of the 41st Int'l Conf. on Machine Learning. Vienna: PMLR, 2024. 53270–53290.

[4] Schäfer M, Nadi S, Eghbali A, Tip F. An empirical evaluation of using large language models for automated unit test generation. IEEE Trans. on Software Engineering, 2024, 50(1): 85–105. [doi: 10.1109/TSE.2023.3334955]

[5] Kang S, Yoon J, Yoo S. Large language models are few-shot testers: Exploring LLM-based general bug reproduction. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering. Melbourne: IEEE, 2023. 2312–2323. [doi: 10.1109/ICSE48619.2023.00194]

[6] Gu SQ, Fang CR, Zhang QJ, Tian FY, Chen ZY. TestART: Improving LLM-based unit test via co-evolution of automated generation and repair iteration. arXiv:2408.03095v1, 2024.

[7] Xiang JH, Xu XY, Kong FC, Peng P, Zhang Z, Zhang YQ. Survey on application and development of large language models in software defect detection and repair. Ruan Jian Xue Bao/Journal of Software, 2025, 36(4): 1489–1529 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/7268.htm [doi: 10.13328/j.cnki.jos.007268]

[8] Xia CS, Zhang LM. Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. In: Proc. of the 33rd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Vienna: ACM, 2024. 819–831. [doi: 10.1145/3650212.3680323]

[9] Yang J, Jimenez CE, Wettig A, Lieret K, Yao SY, Narasimhan K, Press O. SWE-agent: Agent-computer interfaces enable automated software engineering. In: Proc. of the 38th Int'l Conf. on Neural Information Processing Systems. Vancouver: Curran Associates Inc., 2024. 1601. [doi: 10.5555/3737916.3739517]

[10] Qin YH, Wang SW, Lou YL, Dong JH, Wang KX, Li XL, Mao XG. SoapFL: A standard operating procedure for LLM-based method-level fault localization. IEEE Trans. on Software Engineering, 2025, 51(4): 1173–1187. [doi: 10.1109/TSE.2025.3543187]

[11] Xia CS, Deng YL, Dunn S, Zhang LM. Agentless: Demystifying LLM-based software engineering agents. arXiv:2407.01489, 2024.

[12] Kang SNM, An GB, Yoo S. A quantitative and qualitative evaluation of LLM-based explainable fault localization. Proc. of the ACM on Software Engineering, 2024, 1(FSE): 64. [doi: 10.1145/3660771]

[13] Xu CY, Liu ZX, Ren XX, Zhang GH, Liang M, Lo D. FlexFL: Flexible and effective fault localization with open-source large language models. IEEE Trans. on Software Engineering, 2025, 51(5): 1455–1471. [doi: 10.1109/TSE.2025.3553363]

[14] Jiang ZH, Ren XX, Yan M, Jiang W, Li Y, Liu ZX. CoSIL: Software issue localization via LLM-driven code repository graph searching. arXiv:2503.22424v1, 2025.

[15] Hart PE, Nilsson NJ, Raphael B. A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. on Systems Science and Cybernetics, 1968, 4(2): 100–107. [doi: 10.1109/TSSC.1968.300136]

[16] Newell A, Shaw JC, Simon HA. Report on a general problem-solving program. In: Proc. of the 1st Int'l Conf. on Information Processing. Paris: UNESCO, 1960.
[17] Tash J, Russell S. Control strategies for a stochastic planner. In: Proc. of the 12th National Conf. on Artificial Intelligence. Seattle: AAAI, 1994. 1079–1085.
[18] Campbell M, Hoane Jr AJ, Hsu FH. Deep blue. Artificial Intelligence, 2002, 134(1–2): 57–83. [doi: 10.1016/S0004-3702(01)00129-1]
[19] Brown N, Sandholm T. Superhuman AI for multiplayer poker. Science, 2019, 365(6456): 885–890. [doi: 10.1126/science.aay2400]
[20] Silver D, Huang A, Maddison CJ, Guez A, Sifre L, van den Driessche G, Schrittwieser J, Antonoglou I, Panneershelvam V, Lanctot M, Dieleman S, Grewe D, Nham J, Kalchbrenner N, Sutskever I, Lillicrap T, Leach M, Kavukcuoglu K, Graepel T, Hassabis D. Mastering the game of Go with deep neural networks and tree search. Nature, 2016, 529(7587): 484–489. [doi: 10.1038/nature16961]
[21] Gray J, Lerer A, Bakhtin A, Brown N. Human-level performance in no-press diplomacy via equilibrium search. In: Proc. of the 9th Int'l Conf. on Learning Representations. OpenReview.net, 2021.
[22] Yao SY, Yu D, Zhao J, Shafran I, Griffiths T, Cao Y, Narasimhan K. Tree of thoughts: Deliberate problem solving with large language models. In: Proc. of the 37th Conf. on Neural Information Processing Systems. New Orleans, 2023.
[23] Besta M, Blach N, Kubicek A, Gerstenberger R, Podstawski M, Gianinazzi L, Gajda J, Lehmann T, Niewiadomski H, Nyczyk P, Hoefler T. Graph of thoughts: Solving elaborate problems with large language models. In: Proc. of the 38th AAAI Conf. on Artificial Intelligence. Vancouver: AAAI, 2024. 17682–17690. [doi: 10.1609/aaai.v38i16.29720]
[24] Ahmed T, Devanbu P. Better patching using LLM prompting, via self-consistency. In: Proc. of the 38th IEEE/ACM Int'l Conf. on Automated Software Engineering. Echternach: IEEE, 2023. 1742–1746. [doi: 10.1109/ASE56229.2023.00065]
[25] Aho AV, Ullman JD, Sethi R, Lam MS. Compilers: Principles, Techniques, & Tools. 2nd ed., Boston: Pearson, 2007.
[26] Java. Interface instrumentation. 2025. https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/Instrumentation.html
[27] Chattopadhyay S, Nelson N, Gonzalez YR, Leon AA, Pandita R, Sarma A. Latent patterns in activities: A field study of how developers manage context. In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Montreal: IEEE, 2019. 373–383. [doi: 10.1109/ICSE.2019.00051]
[28] Asudani DS, Nagwani NK, Singh P. Impact of word embedding models on text analytics in deep learning environment: A review. Artificial Intelligence Review, 2023, 56(9): 10345–10425. [doi: 10.1007/s10462-023-10419-1]
[29] Just R, Jalali D, Ernst MD. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In: Proc. of the 2014 Int'l Symp. on Software Testing and Analysis. San Jose: ACM, 2014. 437–440. [doi: 10.1145/2610384.2628055]
[30] Kochhar PS, Xia X, Lo D, Li SP. Practitioners' expectations on automated fault localization. In: Proc. of the 25th Int'l Symp. on Software Testing and Analysis. Saarbrücken: ACM, 2016. 165–176. [doi: 10.1145/2931037.2931051]
[31] Hugging Face. Jina embeddings v4: Universal embeddings for multimodal multilingual retrieval. 2025. https://huggingface.co/jinaai/jina-embeddings-v4
[32] OpenAI. GPT-4o mini: Advancing cost-efficient intelligence. 2024. https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence
[33] Hugging Face. Meta Llama-3-8B. 2024. https://huggingface.co/meta-llama/Meta-Llama-3-8B
[34] Razzaq A, Buckley J, Patten JV, Chochlov M, Sai AR. BoostNSift: A query boosting and code sifting technique for method level bug localization. In: Proc. of the 21st IEEE Int'l Working Conf. on Source Code Analysis and Manipulation (SCAM). Luxembourg: IEEE, 2021. 81–91. [doi: 10.1109/SCAM52516.2021.00019]
[35] DeepSeek. DeepSeek-V3. 2025. https://github.com/deepseek-ai/DeepSeek-V3
[36] Gim I, Chen GJ, Lee SS, Sarda N, Khandelwal A, Zhong L. Prompt cache: Modular attention reuse for low-latency inference. In: Proc. of the 11th Annual Conf. on Machine Learning and Systems. Santa Clara: mlsys.org, 2024. 325–338.

附中文参考文献

[7] 香佳宏, 徐霄阳, 孔繁初, 彭湃, 张钊, 张煜群. 大模型在软件缺陷检测与修复的应用发展综述. 软件学报, 2025, 36(4): 1489–1529. http://www.jos.org.cn/1000-9825/7268.htm [doi: 10.13328/j.cnki.jos.007268]

作者简介

秦意浩, 博士生, 主要研究领域为智能化软件工程, 软件维护与演化.
王尚文, 博士, 助理研究员, CCF 专业会员, 主要研究领域为智能化软件工程, 软件维护与演化.
林博, 博士生, CCF 学生会员, 主要研究领域为智能化软件工程, 软件维护与演化.
陈立前, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为程序分析与验证, 抽象解释.
刘万伟, 博士, 教授, CCF 高级会员, 主要研究领域为自动机理论, 形式化方法, 人工智能形式化验证.
毛晓光, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为高可信软件技术.