LAMP: Large-language-model Augmented Mobile Testing Path Exploration Based on Kea

Group 07

李鹏达 [1], 武泽恺 [2], 张耘彪 [3]

10225101460[1], 10225101429[2], 10225101437[3]

1 背景

移动应用自动化测试是目前正在快速发展的领域，研究热点包括 UI 引导的测试输入生成、基于属性的测试(Property-Based Testing，PBT)、大语言模型(Large Language Model，LLM）的应用以及上下文感知的文本输入生成等技术方向。DroidBot[?] 等 UI 引导的测试工具通过动态构建状态转移模型，生成高效的测试输入，适用于兼容性测试和安全分析；基于属性的测试 [?] 通过定义功能属性并生成输入事件序列，有效检测非崩溃性功能错误；InputBlaster[?], QTypist[?] 和 GPTDroid[?] 等工具利用大语言模型的语义理解和生成能力，在异常输入生成、文本输入生成、测试脚本生成和 Bug 复现等方面表现出色，显著提升了测试覆盖率和错误检测能力；上下文感知的文本输入生成技术 [?] 则通过提取 GUI 上下文信息，生成符合语义的输入，填补了文本输入测试的空白。这些技术在移动应用测试中取得了显著的进展。

其中，KEA[?] 是一个基于属性测试的移动应用测试工具。它将基于属性的测试引入移动应用测试，通过定义功能属性并结合路径探索，发现 Android 应用中的非崩溃性功能错误，提供了更深入的功能验证能力。

KEA 目前提供了三种路径探索策略：随机探索、主路径引导探索、和大语言模型辅助的探索。其中，随机探索策略随机地生成输入事件序列，主路经引导探索策略在用户指定的应用 “主路径” 附近进行随机探索。

大语言模型引导策略（LLM 策略）则是对随机策略的加强，主要负责在应用状态空间中遇到难以探索的 UI 状态时，利用 LLM 生成输入事件以增强功能场景覆盖。具体来讲，该策略在随机探索的基础上，于每次输入事件生成前检测当前的 UI 状态是否处于 UI 陷阱（即 UI tarpit，“焦油坑”[?]）。若处于 UI 陷阱，则使用大语言模型生成输入事件，试图跳出 UI 陷阱。

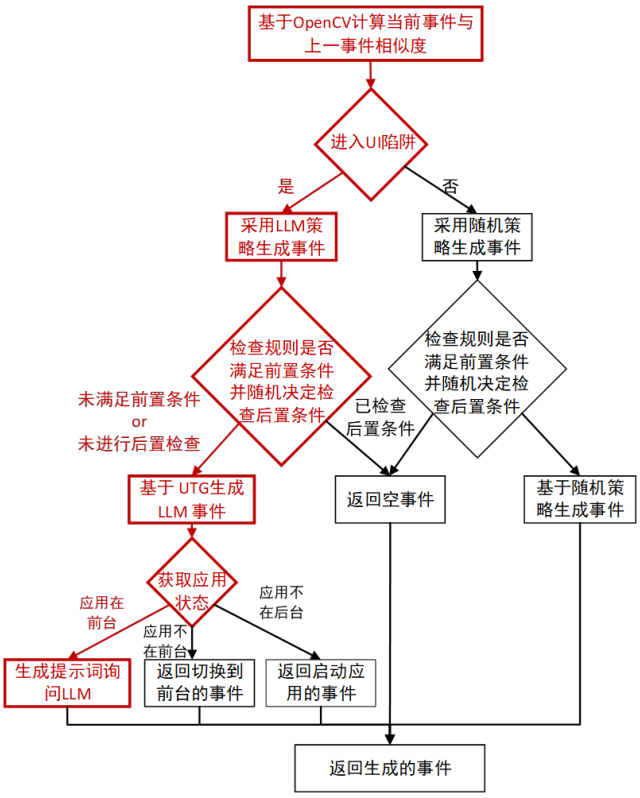

图 1: KEA 现有的大模型引导策略示意图，红色框线图示了触发大模型引导策略的执行过程。

然而，KEA 现有的大语言模型引导策略仍存在一些不足。我们实验观察中发现，在某些情况下，LLM 生成的操作序列仍可能无法有效跳出当前状态，导致陷入死循环或无响应状态。我们将在后续进行分析。

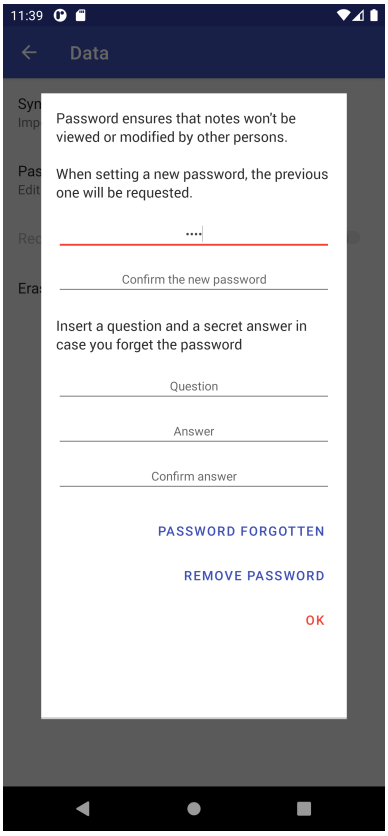

图 2: Omni-Notes 应用的 UI 陷阱示例

2 问题

现有的 LLM 策略主要分为相似度检测和生成事件两个模块。图 **??** 展示了现有的大模型引导策略的基本流程。

进行相似度检测时，KEA 采用了 OpenCV 来对比两个连续的 UI 状态的屏幕截图，从而计算两个状态之间的相似度，当连续多次相似度超过设定的阈值时，KEA 认为当前处于 UI 陷阱。此时，KEA 会使用大语言模型生成输入事件，试图跳出 UI 陷阱。基于现有的 UI 状态信息和可能的输入事件，KEA 会询问大语言模型，由大语言模型给出一个输入事件并执行。

然而，现有 LLM 策略效果不佳。如图 **??** 所示，该界面是 Omni-Notes 的笔记密码设置页面，用户必须在填写表单后才能点击按钮进行跳转。在随机探索的策略下，KEA 会陷入死循环，无法有效地跳出该页面。当我们使用 KEA 现有的 LLM 策略时，发现生成的输入事件仍然无法有效地跳出该页面，导致长期在该页面陷入死循环。这主要是因为，当前 LLM 策略选择的输入事件仅依赖于当前 UI 状态下可执行的输入事件。LLM 并不能理解当前 UI 状态下的界面结构，缺乏对页面上组件信息的深入理解，使其生成的操作不够准确。同时，LLM 策略所生成的输入事件并不依赖于以往的输入事件序列，缺乏上下文感知能力。即，LLM 生成的输入事件 ϕ_i 的生成仅依赖于当前 UI 状态 θ_i，却无法感知 LLM 之前的输入事件所形成的事件序列 $\{\phi_1, \phi_2, \ldots, \phi_{i-1}\}$ 的**真实意图**。这使得 LLM 策略在生成输入事件时缺失了对核心任务的理解，从而导致生成的输入事件不够准确。这些问题都导致了 LLM 策略在跳出 UI 陷阱时的效果不佳。

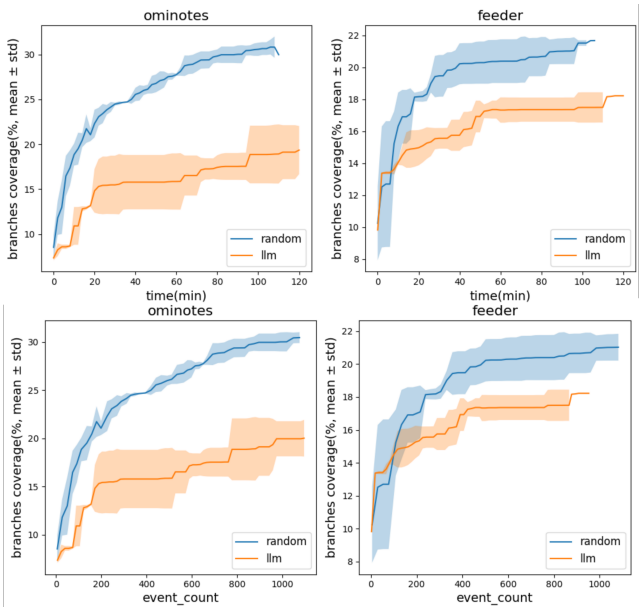

图 3: LLM 策略和随机策略相同时间或相同事件数量时代码分支覆盖率，阴影表示三次实验的标准差。

同时，我们在实验过程中发现，两种情况会导致 LLM 策略产生导致显著的额外开销：1) LLM 策略可能会生成无效操作，生成的输入事件并不能逃离 UI 陷阱；2) 现有的相似度检测方法可能会导致 UI 陷阱误判。在这些情况下，LLM 策略会频繁地调用大语言模型来生成输入事件，消耗了 LLM 资源和通信时间。

图 **??** 展示了运行 LLM 策略和随机策略相同时间或相同事件数量时的代码分支覆盖率，LLM 策略相比随机策略不但没有提升，甚至有所明显的下降。（如，Omni-Notes 中，分支覆盖率平均下降了 39.31%）

总而言之，现有策略存在的三个问题：**1) LLM 策略在解决 UI 陷阱方面效果不佳，其生成的事件不准确，且缺乏上下文感知能力；2) LLM 策略由于需要频繁调用，会产生显著的调用开销；3) 现有的相似度检测方法可能会导致误判**。

Algorithm 1 Detect UI Tarpit

function DETECT(xml_1, xml_2, $threshold$)
  $similarity \leftarrow$ COMPAREXML(xml_1, xml_2)
  if $similarity > 90$ **then**
    $sim\_count \leftarrow sim\_count + 1$
    if $sim\_count \geq threshold$ **then**
      $sim\_count \leftarrow 0$
      return True
    end if
  end if
  return False
end function

因此，我们提出了一种新的大语言模型辅助的路径探索策略，称为 LAMP（Large-language-model Augmented Mobile Testing Path Exploration Based on KEA）。LAMP 通过引入基于语义的 UI 陷阱检测算法和基于迭代事件序列生成的增强型 LLM 探索策略，结合大语言模型的语义理解和生成能力，提升了路径探索的效率和准确性。

3 方法

我们提出了三种方法来优化现有的 LLM 策略。在 §?? 中，我们提出了一种基于语义的 UI 陷阱检测算法，利用 UI 界面的 XML 语义结构来判断 UI 陷阱；在 §?? 中，我们提出了一种基于迭代事件序列生成的增强型 LLM 探索策略，采用循环迭代的方式来逐步引导大语言模型生成操作序列，从而模拟人类操作的思维过程；在 §?? 中，我们提出了一种频率感知的随机探索策略，通过缓存输入事件的触发频率来优化路径探索。其余的贡献在 §?? 中介绍

3.1 基于语义的 UI 陷阱检测算法

我们在 KEA 的 LLM 策略原有的相似度计算方案的基础上，重新提出了一种新的相似度计算方案，希望通过关注 UI 界面的 XML 语义结构来判断 UI 陷阱。我们发现，XML 树结构能够更好地反映 UI 状态的语义信息和层次关系，因此可以通过对比两个 UI 状态的 XML 树结构，来判断两个 UI 状态是否相似。

为此，我们将 XML 树结构进行简化，将 XML 树抽象成只包含 tag 和 children 的轻量结构，便于对结构相似性进行对比。如算法 ?? 所示，我们得到了两个简化后的 XML 树结构 $tree_1$ 和 $tree_2$，然后通过调用 `CompareTree` 函数递归遍历树结构，计算两个树的相似度分数 $score$ 和总节点数 $total$。最后，我们返回相似度分数。

Algorithm 2 Compare XML

function COMPAREXML(xml_1, xml_2)
  $tree_1, tree_2 \leftarrow$ Simplify xml_1, xml_2 and construct trees
  $score, total \leftarrow$ COMPARETREE($tree_1, tree_2$)
  return 100.0 if $total = 0$ else $(score/total) \times 100$
end function

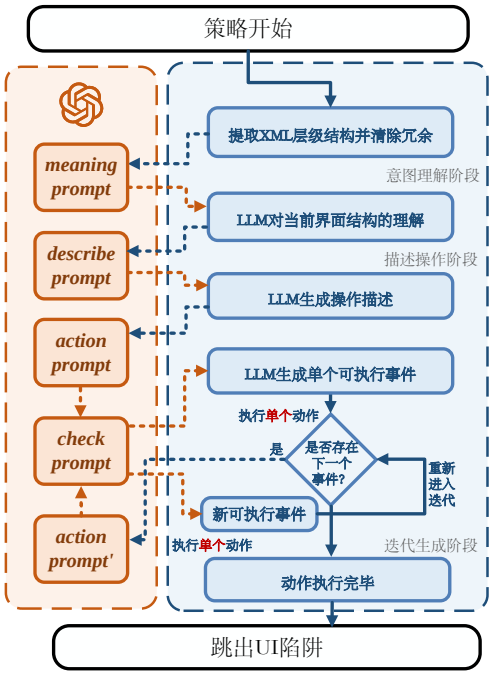

图 4: 基于迭代的提示工程

随后，在算法 ?? 中，我们沿用了 KEA 的 LLM 策略原有的启发式相似度判断方案，使用 `CompareXML` 函数来计算两个 UI 状态的相似度分数 $similarity$ 后，如果相似度分数大于 90，则认为两个 UI 状态相似。我们使用一个计数器 sim_count 来记录连续相似的次数，当连续相似的次数超过设定的阈值时，我们认为当前处于 UI 陷阱。

在后续的测试过程中我们观察到，该方法显著减少了对 UI 陷阱的误判，同时仍能够识别出原有相似度检测方法所发现的 UI 陷阱。经过人工验证，这些

UI 陷阱确实是难以被探索的 UI 状态。这样的方法提高了 LLM 策略的效率，减少了对大语言模型的调用次数。

3.2 基于迭代事件序列生成的增强型 LLM 探索策略

我们将提示工程包括三个阶段：1）**意图理解阶段**（§??）；2）**操作描述阶段**（§??）；3）**迭代生成阶段**（§??）。我们在 §?? 中介绍了整个 LLM 事件生成的算法流程。流程图如图 ?? 所示。

3.2.1 意图理解阶段

在这一阶段，我们将获取到的 XML 结构作为上下文输入到大语言模型中，要求其理解当前界面的意图和功能。但是，由于通过 `uiautomator2` 获取的 XML 结构内容比较复杂，通常含有大量无用和干扰信息。因此，我们首先对 XML 数据进行预处理。首先，我们去除了 XML 中系统状态栏的结构信息，以减少干扰。其次，我们删除了值为空字符串和 `false` 的属性，以精简 XML，减少无用信息。

我们设计了一个提示词 `meaning_prompt()`（见附录 ??），用于引导模型生成对当前界面的描述。通过获取到模型对当前界面结构的理解，我们可以更好地引导模型生成后续的操作序列。

3.2.2 操作描述阶段

在这一阶段，我们根据之前的上下文信息，向大语言模型提出新的要求——要求其生成一个完整任务流程的描述，从而引导模型一步步生成事件。

我们设计了一个提示词 describe_prompt（见附录 ??），用于引导大语言模型基于当前 UI 页面生成一个用户可能执行的新操作。该提示词明确要求模型仅描述一个尚未生成的任务，并补充执行该任务所需的具体步骤，从而在一定程度上避免了重复。此外，提示中加入了当前已生成任务的列表，进一步实现了去重，引导模型从**用户视角**出发进行交互意图推理。

3.2.3 迭代生成阶段

在这个阶段中，我们将大语言模型生成的操作描述作为输入，不断要求模型生成新的操作序列。我们设计了两个提示词 action_prompt 和 action_prompt_prime（见附录 ??），分别用于引导模型生成操作序列的第一步和后续步骤。

Algorithm 3 Main Exploration Loop

```
function START(input_manager)
    count ← 0
    while count < max_event_count do
        Update UI state and snapshots
        Start the APP if essential
        if LLM mode is active then
            event ← GENERATELLMEVENT
        else if DETECTUITARPIT(last_state, current_state) then
            Activate LLM Mode
            event ← GENERATELLMEVENT
        else
            event ← GENERATERANDOMEVENT
        end if
        EXECUTE(event)
        count ← count + 1
    end while
    Clean up and exit
end function
```

在 action_prompt 中，我们引导大语言模型将用户操作步骤以结构化的 JSON 格式进行表达，以便于后续被程序解析与执行。该提示词强调只描述“用户刚刚执行的操作的第一步”，并提供了统一的格式规范和字段说明，如 action、selectors、inputText 和 hasNext。其中，selectors 用于精确定位页面元素，支持多种常见属性组合（如 resourceId 与 text），并要求所有值必须可在对应的 UI XML 中找到，以确保操作的可执行性与准确性。此外，通过布尔值 hasNext，我们可以判断是否还有后续操作需要生成。

在 action_prompt_prime 中，我们引导大语言模型在给定当前页面状态的基础上，继续生成前一个操作的“下一步”操作。该提示词通过动态插入最新的页面 XML 状态，确保模型具备当前 UI 上下文，从而做出合理的操作。

在生成操作序列后，我们还需要对生成的操作进行检查，以确保其符合要求。我们设计了一个提示词 check_prompt（见附录 ??），用于引导大语言模型检查生成的操作是否符合要求。该提示词要求模型检查生成的操作是否符合以下两个条件：1）选择器必须在

Algorithm 4 Generate LLM Event

```
function GenerateLLMEvent
    if Continuing LLM Sequence then
        Build Next Action Prompt
        response ← CallLLM
        response ← ValiditeByLLM
        act ← ParseAction(response)
    else
        Build Meaning Prompt
        r1 ← CallLLM
        Build Task Prompt
        r2 ← CallLLM
        Build First Action Prompt
        r3 ← CallLLM
        response ← ValiditeByLLM
        act ← ParseAction(response)
    end if
    Set LLM Mode to act.hasNext
    return WrapAsU2Event(act)
end function
```

XML 中找到；2）选择器必须唯一标识元素。如果没有问题，则按原样输出；否则，修改相应内容。

3.2.4 事件生成流程

如算法 **??** 所示，系统在主循环中不断执行事件生成与执行流程。每轮循环中，系统会根据当前状态选择事件生成策略：当 LLM 模式处于激活状态时，调用 `GenerateLLMEvent()` 函数生成由大语言模型驱动的操作事件；否则，将调用 `GenerateRandomEvent()` 函数生成随机事件。若系统检测到 UI 陷阱（UI Tarpit），也会触发 LLM 模式并转为基于模型的操作生成。

算法 **??** 展示了 `GenerateLLMEvent()` 函数的内部逻辑。函数首先判断是否正在延续一个 LLM 操作序列：若是，则调用 `BuildNextActionPrompt()` 构造提示，获取 LLM 响应，并依次通过 `ValiditeByLLM()` 验证响应有效性，再调用 `ParseAction()` 解析出具体操作；否则，将按顺序构建页面意图、任务目标和首个动作的提示，分别调用 `CallLLM()` 获取响应，最终进行验证和解析操作。最后，该函数会根据操作是否存在后续步骤（`hasNext`）来更新 LLM 模式，并将解析结果包装为 U2 可执行事件返回。

Algorithm 5 Frequency-Aware Random Strategy

```
function GenerateEvent
    s ← current state
    if s ∉ input_table then
        possible_events ← GetPossibleInputs(s)
        Initialize input_table[s] with an empty events list
        for all event ∈ possible_events do
            Add event to input_table[s].events
            if event ∉ event_table then
                event_table[event] ← 0
            end if
        end for
    end if
    counts ← ∅
    for all event ∈ input_table[s].events do
        counts[event] ← event_table[event].tried
    end for
    weights ← GetWeights(input_table[s].events, counts)
    selected_event ← randomly select one event from the list using weights
    Increment event_table[selected_event].tried
    return selected_event
end function
```

3.3 频率感知的随机探索策略

我们提出了一种频率感知的随机探索策略（见算法 **??**），通过记录和利用输入事件的触发频率，引导探索过程更均匀地覆盖不同事件，避免某些操作被频繁重复触发。

在事件生成过程中，系统首先检查当前状态是否已存在于 `input_table` 中。若不存在，则通过 `GetPossibleInputs` 获取该状态下所有可能的输入事件，并将其记录在 `input_table` 中。对于每个新事件，如果其尚未存在于 `event_table` 中，系统会初始化其尝试次数为零。

随后，系统统计当前状态下各输入事件的触发频次，从 `event_table` 中获取每个事件的尝试次数，并根据这些频率计算采样权重（通过 `GetWeights` 函数）。最终，系统根据权重随机选择一个输入事件，并更新

Algorithm 6 Move App to Foreground

```
function MoveIfNeed
    top ← current foreground app
    if top ≠ target app then
        out_cnt ← out_cnt + 1
        if out_cnt > threshold then
            out_cnt ← 0
            simulate back navigation
            recheck top
            if top ≠ target app then
                force-stop top and recheck top
                if top ≠ target app then
                    force-stop and restart target app
                end if
            end if
        end if
    else
        out_cnt ← 0
    end if
end function
```

其尝试次数，用于执行。

为了避免重复选择已经尝试过的事件，又要防止尝试过的事件完全被忽略，我们设计了一种基于频率的采样方法 `GetWeights` 函数。设 $E = \{e_1, e_2, \ldots, e_n\}$ 为所有可能事件的集合，令 c_i 表示事件 e_i 已被尝试的次数。给定一个衰减因子 $\gamma \in (0, 1)$，选择事件 e_i 的权重 w_i 可以表示为：

$$w_i = w_0 \cdot \gamma^{c_i} \tag{1}$$

因此，事件 e_i 被选择的概率 $P(e_i)$ 可以表示为：

$$P(e_i) = \frac{w_i}{\sum_{j=1}^{n} w_j} = \frac{\gamma^{c_i}}{\sum_{j=1}^{n} \gamma^{c_j}} \tag{2}$$

通过采用指数衰减的方式，我们可以有效地平衡新事件和已尝试事件的选择概率，从而实现更均匀的探索。

3.4 其余贡献

在实验过程中，我们发现 KEA 的唤醒机制问题。我们发现，在某些特定的情况下，测试过程中在跳转到到其他的应用或系统页面时，KEA 所实现的唤醒机制无法有效地将被测应用唤醒至前台，而是通过重启或返回事件才能有概率达到返回应用的目的。我们重新实现了一个唤醒机制，如算法 **??** 所示，我们实现了一个唤醒机制，用于确保目标应用始终处于前台状态。在每次事件执行前，系统会检查当前前台应用是否为目标应用。若检测到应用已被切换至后台，则首先尝试通过模拟返回操作将其唤醒。如果经过多次尝试后仍未恢复前台状态，系统将逐步采取更强制性的措施：首先尝试关闭当前前台应用；若仍无法唤醒目标应用，则最终通过强制停止并重启目标应用的方式将其恢复至前台。同时，我们维护一个计数器 `out_cnt`，用于控制上述操作的触发频率，避免过度干预系统状态。

表 1: 本文使用的应用数据集

App	Primary Feature	Stars	Installations
AnkiDroid	Flashcard Manager	9.5K	10M–50M
Feeder	RSS Feed Reader	2.1K	100K–500K
Newpipe	YouTube Client	33.8K	1M–5M
OmniNotes	Note Manager	2.7K	10M–50M
Wikipedia	Wiki Reader	3K	10M–50M

4 实验

4.1 实验设置

4.1.1 数据集

表 **??** 展示了我们在实验中使用了五个开源应用程序的基本信息，这些应用程序具有不同的功能和复杂性，并具有一定的下载量。我们从 GitHub 上获取了这些应用程序的最新源代码，并使用 JaCoCo[**?**] 工具进行静态插桩，收集代码覆盖度数据。

4.1.2 实验环境

本实验环境基于 Windows 11 家庭中文版（版本 26100.3915），使用 Python 3.11.9 作为脚本执行环境，配合 JaCoCo 0.8.10 进行 Java 代码覆盖率分析，并通过 Android Emulator（Pixel 9，API 34）模拟移动设备运行环境，用于测试和分析 Android 应用的行为和覆盖率表现。

App	Policy	Time (%)			Event (%)		
		Max	Min	Avg	Max	Min	Avg
AnkiDroid	`llm`	6.56	-43.83	-18.69	17.63	-43.83	-10.45
	`new`	51.32	2.72	32.22	55.44	6.91	36.28
Feeder	`llm`	6.88	-22.11	-15.54	10.59	-17.11	-11.76
	`new`	29.81	1.02	15.87	33.25	4.99	22.54
Newpipe	`llm`	121.86	22.63	32.71	144.69	28.24	41.09
	`new`	140.29	29.24	44.10	140.29	42.33	51.83
Omninotes	`llm`	-13.73	-50.08	-39.31	-13.73	-47.34	-35.73
	`new`	16.12	-1.65	3.55	16.12	0.19	9.44
Wikipedia	`llm`	6.68	-39.63	-15.34	34.74	-30.24	-0.58
	`new`	14.57	-14.38	1.97	28.24	-10.54	17.27

表 2: 运行相同时间或相同时间数量时，LLM 策略 (`llm`) 和 LAMP 策略 (`new`) 代码分支覆盖率相比随机策略的相对提升

4.1.3 度量标准

我们使用以下度量标准来评估 LAMP 的有效性：

- **代码覆盖率：** 指令覆盖度、分支覆盖度、圈复杂度、行覆盖度、方法覆盖度、类覆盖度。
- **UI 覆盖率：** UTG 状态数、UTG 边数、Activity 覆盖度。

4.1.4 实验过程

我们对每个应用程序运行 KEA 原有随机策略 (`random`)，LLM 策略 (`llm`) 和我们提出的 LAMP 策略 (`new`) 至少 3 次，每个策略运行 120 分钟或 1200 个事件。

我们使用 JaCoCo 工具收集代码覆盖率数据，并计算平均值和标准差，并统计生成的 UTG 中的状态数、边数和 Activity 覆盖率。

4.2 研究问题

我们设计了以下研究问题来评估 LAMP 策略的有效性：

- **RQ1:** 和 KEA 现有策略相比，我们的策略能够提升多少代码覆盖度?
- **RQ2:** 和 KEA 现有策略相比，我们的策略能够提升多少 UI 覆盖度?
- **RQ3:** 我们的随机探索策略相比于现有的随机探索策略，能够提升多少代码覆盖度?

4.3 实验结果与分析

4.3.1 RQ1

表 **??** 展示了运行相同时间或相同事件数量时，LLM 策略和 LAMP 策略代码分支覆盖率相比随机策略的相对提升百分比，在至少三次运行取平均值后，按时间维度和事件数量维度计算得到的最小值、最大值和平均值。

实验结果显示，LAMP 策略在大多数情况下都能显著提升代码分支覆盖率，尤其是在 AnkiDroid，Feeder 和 NewPipe 应用中，LAMP 策略的平均提升幅度分别达到了 32.22%，15.87% 44.10%。而在 OmniNotes 和 Wikipedia 应用中，LAMP 策略的平均提升幅度也达到了 3.55% 和 1.97%。

然而，无论是时间维度还是事件数量维度，原有的 LLM 策略在大部分应用中表现不佳，尤其是在 AnkiDroid 和 Feeder 应用中，LLM 策略的平均提升幅度分别为 -18.69% 和 -15.54%。而在 NewPipe 应用中，LLM 策略却表现出 32.71% 的较大提升幅度。考

App name	# of UTG States			# of UTG Events			Activity Coverage		
	Random	Ours	Difference	Random	Ours	Difference	Random	Ours	Difference
Omninotes	232.67	261.67	+12.46%	496.00	525.33	+5.91%	0.378	0.422	+11.76%
AnkiDroid	293.67	309.00	+5.22%	517.00	573.33	+10.90%	0.427	0.438	+2.44%
Feeder	169.33	128.67	-24.02%	344.00	393.67	+14.44%	0.185	0.259	+40.00%
Newpipe	367.67	455.67	+23.93%	601.67	625.00	+3.88%	0.238	0.333	+40.00%
Wikipedia	327.33	412.00	+25.87%	605.67	704.67	+16.35%	0.307	0.333	+8.47%

表 3: 不同应用中 LAMP 策略和随机策略 UI 覆盖度的对比

虑到 NewPipe 应用的功能与其他应用差异较大，我们认为这是由于 LLM 策略在不同类型的应用中的适应性差异所导致的。

这表明，无论是相同时间还是相同事件数量，LAMP 策略在不同应用中都能有效提升代码分支覆盖率，而原有的 LLM 策略在某些应用中则表现不佳，甚至出现了负提升幅度。这说明 LAMP 策略在路径探索中具有更好的适应性和有效性。

4.3.2 RQ2

表 ?? 展示了我们提出的 LAMP 策略在不同应用中的 UTG 状态数、UTG 边数和 Activity 覆盖率与随机策略的对比。由于原有 LLM 策略并未提供生成 UTG 功能，我们在此将其排除。

实验数据表明，LAMP 策略在大多数应用中都能显著提升 UTG 状态数和 UTG 边数，尤其是在 Wikipedia 和 NewPipe 应用中，UTG 状态数的提升幅度分别达到了 25.87% 和 23.93%。而在 Activity 覆盖率方面，LAMP 策略在所有应用中都能提升，尤其是在 Feeder 和 NewPipe 应用中，Activity 覆盖率的提升幅度达到了 40.00%。

这说明 LAMP 策略在 UI 覆盖度方面具有显著的优势，能够提升对于应用 UI 页面的探索，达到更深更广的探索路径。

4.3.3 RQ3

图 ?? 展示了增强随机策略与原有随机策略在代码分支覆盖率方面的对比。实验结果表明，增强的随机策略对比原有策略在代码分支覆盖率上明显提升，这是因为我们的策略可以有效地平衡新事件和已尝试事件的选择概率，从而实现更均匀的探索。

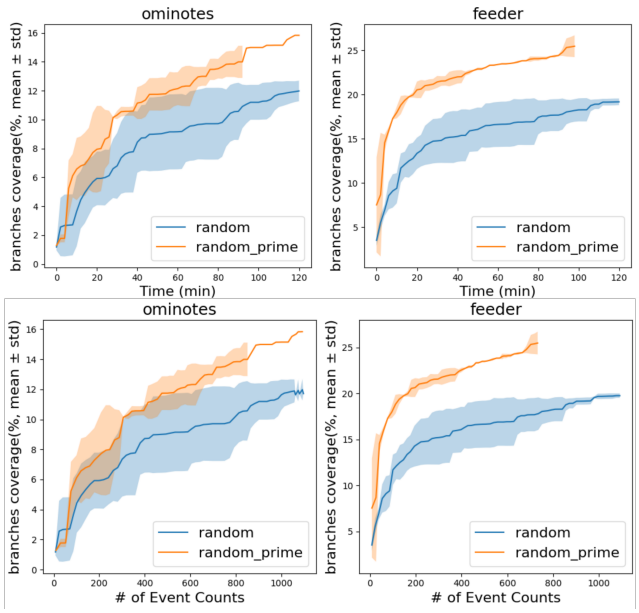

图 5: 增强随机策略与原有随机策略代码分支覆盖率对比

5 结论

本研究基于 KEA 的大语言模型引导策略，提出了三种优化方案。首先，我们通过 XML 标记提高页面语义理解显著提升了对 UI 陷阱识别的准确度；其次，采用逐步引导的模式，使得大语言模型能够产生更为有效的解决方案；最后，我们发现在使用随机探索策略时减少重复探索的可能性提高了探索效率。这三种技术共同构成了一个全面的框架，用于优化大型语言模型在复杂问题解决环境中的表现，

A　提示工程

Listing 1: 意图理解

```
prompt = f"""This is an XML representation of an Android application page:
{get_xml()}
Please describe the purpose of this page in the most concise language
    possible."""
```

Listing 2: 操作描述

```
prompt = f"""If you were the user, what would you do on this page? You can
     only describe one action.
Please try to generate tasks that have not been generated before. Below
    are the tasks that have already been generated:
{list(self._generated_tasks)}
Please list the steps required to complete this action. (This action will
    be named 'The Task')"""
```

Listing 3: 初始事件生成

```
prompt = """Please describe the **first step** of the operation you just
    performed in JSON format, as shown below:
{
    "action": "input_text",
    "selectors": {"resourceId": "com.example:id/input", "text": "password
        "},
    "inputText": "123456",
    "hasNext": true
}
Notes:
- The "action" must be one of: click, long_click, input_text, press_enter
- "selectors" can only include: **text**, **className**, **description**,
    **resourceId**, and must be in camelCase. You can not use other
    selectors.
- The value is the value of the selector, which must be found in the
    previous XML
- "inputText" is the text to input, only present when the action is
    input_text
- "hasNext" is a boolean indicating whether there is a next step. Set it
    to false if there is no next step
Try to combine multiple selectors to uniquely identify the element.
Please return the operation in JSON format only. Do not explain or use
    code blocks.
"""
```

Listing 4: 后续事件生成

```
prompt = f"""Now, the current state of the page is as follows: {get_xml(
    self.device.u2)}
Please describe the **next step** of the operation you just performed in
    JSON format, using the same format as above.
"""
```

Listing 5: 操作检查

```
prompt = """Please check whether the operation or sequence of operations
    you just generated meets the requirements:
- The selectors must be found in the XML.
- The selectors must uniquely identify the element.
If there are no issues, output it as is; otherwise, modify it accordingly.
"""
```