demo笔记

参考文献

1. 特点：针对JAVA
2. 概念解析
    a. Call Graph（调用图）

    用来展示调用关系

    - 节点——函数
    - 边——函数调用，调用方 指向 被调用方

    b. 两跳调用上下文

    目标函数前后两层的调用关系

    - 1 跳被调用方：A直接调用的函数（比如A→B、A→C）；
    - 2 跳被调用方：B和C再调用的函数（比如B→D、C→E）；
    - 1 跳调用方：直接调用A的函数（比如F→A、G→A）；
    - 2 跳调用方：F和G被谁调用（比如H→F、I→G）；

    最终返回这些函数的 “函数体代码”，就是 SCLogger 需要的调用上下文
3. SCLogger的方法论：
    a. 用 Java 的静态分析工具（Soot、Eclipse JDT）解析源码，生成调用图
    b. 输入目标方法，在调用图里找 “调用它的方法”（逆向找）和 “它调用的方法”（正向找）
    c. 限制 “两跳”（避免找太多导致信息冗余）
    d. 提取这些方法的函数体，作为上下文喂给 LLM

Go Static: Contextualized Logging Statement Generation

YICHEN LI, The Chinese University of Hong Kong, China
YINTONG HUO*, The Chinese University of Hong Kong, China
RENYI ZHONG, The Chinese University of Hong Kong, China

ZHIHAN JIANG, The Chinese University of Hong Kong, China
JINYANG LIU, The Chinese University of Hong Kong, China
JUNJIE HUANG, The Chinese University of Hong Kong, China
JIAZHEN GU, The Chinese University of Hong Kong, China
PINJIA HE, The Chinese University of Hong Kong, China
MICHAEL R. LYU, The Chinese University of Hong Kong, China

Logging practices have been extensively investigated to assist developers in writing appropriate logging statements for documenting software behaviors. Although numerous automatic logging approaches have been proposed, their performance remains unsatisfactory due to the constraint of the single-method input, without informative programming context outside the method. Specifically, we identify three inherent limitations with single-method context: limited static scope of logging statements, inconsistent logging styles, and missing type information of logging variables.

To tackle these limitations, we propose SCLogger, the first contextualized logging statement generation approach with inter-method static contexts. First, SCLogger extracts inter-method contexts with static analysis to construct the *contextualized prompt* for language models to generate a tentative logging statement. The contextualized prompt consists of an extended static scope and sampled similar methods, ordered by the chain-of-thought (COT) strategy. Second, SCLogger refines the access of logging variables by formulating a new *refinement prompt* for language models, which incorporates detailed type information of variables in the tentative logging statement.

The evaluation results show that SCLogger surpasses the state-of-the-art approach by 8.7% in logging position accuracy, 32.1% in level accuracy, 19.6% in variable precision, and 138.4% in text BLEU-4 score. Furthermore, SCLogger consistently boosts the performance of logging statement generation across a range of large language models, thereby showcasing the generalizability of this approach.

CCS Concepts: • **Software and its engineering → Maintaining software**.

Additional Key Words and Phrases: code generation, software maintenance, large language models

*Yintong Huo is the corresponding author.

Authors' addresses: Yichen Li, The Chinese University of Hong Kong, Hong Kong, China, ycli21@cse.cuhk.edu.hk; Yintong Huo, The Chinese University of Hong Kong, Hong Kong, China, ythuo@cse.cuhk.edu.hk; Renyi Zhong, The Chinese University of Hong Kong, Hong Kong, China, ryzhong22@cse.cuhk.edu.hk; Zhihan Jiang, The Chinese University of Hong Kong, Hong Kong, China, zhjiang22@cse.cuhk.edu.hk; Jinyang Liu, The Chinese University of Hong Kong, Hong Kong, China, jyliu@cse.cuhk.edu.hk; Junjie Huang, The Chinese University of Hong Kong, Hong Kong, China, junjayhuang@outlook.com; Jiazhen Gu, The Chinese University of Hong Kong, Hong Kong, China, jiazhengu@cuhk.edu.hk; Pinjia He, The Chinese University of Hong Kong, Shenzhen, China, hepinjia@cuhk.edu.cn; Michael R. Lyu, The Chinese University of Hong Kong, Hong Kong, China, lyu@cse.cuhk.edu.hk.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2024 Copyright held by the owner/author(s).
ACM 2994-970X/2024/7-ART28
https://doi.org/10.1145/3643754

callgraph3643754.pdf

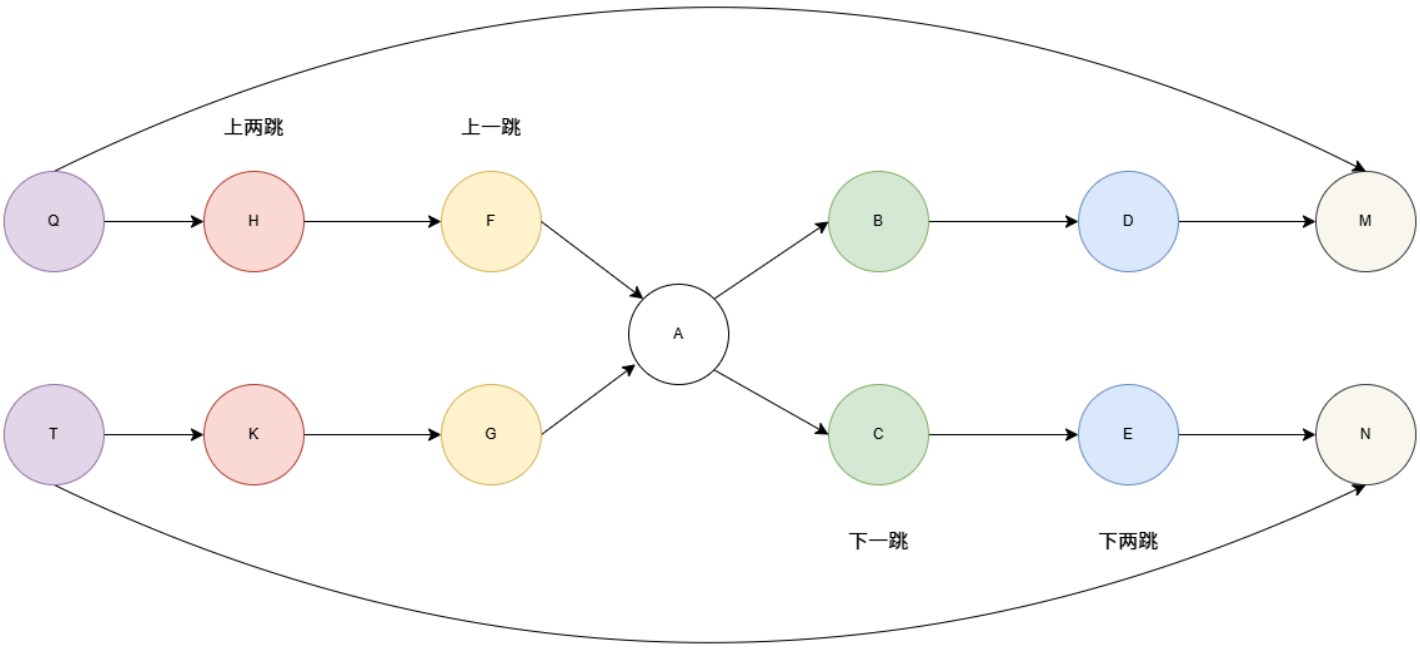
1. demo实现

实验环境：Ubuntu

待测程序：

test.cpp

手动绘制callgraph



1.1 安装LLVM

1. 更新软件源

代码块

```
sudo apt update && sudo apt upgrade -y
```

2. 安装LLVM

```
代码块
sudo apt install -y llvm-14 clang-14 lld-14
```

3. 验证安装情况

```
代码块
llc-14 --version
opt-14 --version
```

1.2 把 C/C++ 源码转换成 LLVM 中间代码（IR）

原因：LLVM 不能直接分析源码，需要先把.c/.cpp文件转换成 “中间代码（.ll 文件）”

1. 进入待分析.cpp文件目录，打开终端
2. 将cpp文件转换为中间代码.ll文件

```
代码块
clang-14 -emit-llvm -S -O0 -g test.cpp -o test.ll
```

1.3 用 LLVM 构建 Call Graph

1. 生成callgraph

若callgraph.txt没有内容，则手动将终端输出的内容复制到该文件中即可

```
代码块
opt-14 -print-callgraph test.ll -o /dev/null > callgraph.txt
```

2. 还原函数名

比如：_Z1Av 会还原成 A()，便于理解

```
代码块
cat callgraph.txt | c++filt > callgraph_clean.txt
```

1.4 解析调用图，找两跳上下文

使用python脚本自动提取

extract_context.py

1. 将代码中的 target_function 改为目标函数的名字（包括括号）

```
if __name__ == "__main__":
    # ------------------------- 配置参数 -------------------------
    # 1. 选择调用图文件路径（二选一，或修改为实际路径）
    # callgraph_path = "callgraph.txt"        # 未清洗文件（修饰名：_Z1Av）
    callgraph_path = "callgraph_clean.txt"  # 清洗文件（原始名：A()）

    # 2. 选择查询模式（二选一）
    target_function = "A()"  # 模式1：查询指定函数（需匹配文件格式）
    # target_function = None  # 模式2：批量查询所有函数

    # ------------------------- 执行分析 -------------------------
    analysis_result = analyze_function_hops(callgraph_path, target_function)
    print(analysis_result)

    # 保存结果到文件
    output_file = "function_multi_hop_result.txt"
    with open(output_file, "w", encoding="utf-8") as f:
        f.write(analysis_result)
    print(f"\n分析结果已保存到：{output_file}")
```

2. 在终端执行脚本

```
代码块
1   python3 extract_context.py
```

3. 结果输出示例

```
lulu@oslab:~/CallGraph_DEMO$ python3 extract_context.py
=== 函数调用多跳关系分析（基于清洗后的callgraph）===
说明：函数名格式为原始函数名（如A()、main()）
============================================================

目标函数：A()
上一跳（直接调用该函数的函数）：F(), G(), main
上两跳（间接调用该函数的函数）：H(), K(), main
下一跳（该函数直接调用的函数）：B(), C()
下两跳（该函数间接调用的函数）：D(), E()

分析结果已保存到：function_multi_hop_result.txt
```