单位代码：10293 密　级：_____

南京邮电大学

专 业 学 位 硕 士 论 文

论文题目：基于深度学习的算法编程作业自动评价技术研究

学　　号	1221045708
姓　　名	胡天昊
导　　师	张卫丰
专业学位类别	电子信息硕士
类　　型	全日制
专业（领域）	分析测试
论文提交日期	2024.6

Deep Learning Based Feedback Generation for Online Judge Student Assignments

Thesis Submitted to Nanjing University of Posts and Telecommunications for the Degree of

Master of Electronic Information

By

Tianhao Hu

Supervisor: Prof. Weifeng Zhang

June 2024

南京邮电大学学位论文原创性声明

本人声明所呈交的学位论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得南京邮电大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

本人学位论文及涉及相关资料若有不实，愿意承担一切相关的法律责任。

研究生学号：_____ 研究生签名：_____ 日期：_____

南京邮电大学学位论文使用授权声明

本人承诺所呈交的学位论文不涉及任何国家秘密，本人及导师为本论文的涉密责任并列第一责任人。

本人授权南京邮电大学可以保留并向国家有关部门或机构送交论文的复印件和电子文档；允许论文被查阅和借阅；可以将学位论文的全部或部分内容编入有关数据库进行检索；可以采用影印、缩印或扫描等复制手段保存、汇编本学位论文。本文电子文档的内容和纸质论文的内容相一致。论文的公布（包括刊登）授权南京邮电大学研究生院办理。

非国家秘密类涉密学位论文在解密后适用本授权书。

研究生签名：_____ 导师签名：_____ 日期：_____

摘要

随着互联网时代的不断发展，计算机相关专业的招生人数越来越多，使得高校教师们的教学压力也越来越大。与此同时，越来越多的人选择在学校之外自学编程。代码在线评测系统（Online Judge）在编程学习中扮演着至关重要的角色，因为它能够自动运行参与者提交的代码，并根据预先设定的测试用例来评测代码的正确性和效率。然而，目前的系统通常只能告知使用者代码错误，或者提供出错的测试用例，而不能帮助使用者找到错误的根源并进行修改。现有的反馈信息生成工具通常会寻找正确版本代码和错误版本代码的不同之处来生成反馈，但往往过于依赖上下文信息，导致错误的反馈结果。另外，现有的代码跟踪对齐工具如 APEX 使用符号分析来进行数据对齐，但忽略了语义信息的差异，因此可能产生质量较低的对齐结果，最终生成低质量的反馈。

本文设计了基于深度学习的算法编程作业自动评价技术，主要贡献有 (1) 基于深度学习的代码自动对齐方法。微调 “all-MiniLM-L6-v2” 模型来使其适应本工作，接着使用程序的动态分析和符号分析方法获取正确版本代码和错误版本代码的跟踪信息并进行初始对齐，然后使用输入语句作为基准，辅助以深度学习模型推测结果，对初始对齐结果进行纠正与扩充。最后使用广度遍历算法扩充序列对，完成扩充生成最终对齐结果。(2) 基于深度学习的代码自动反馈生成技术。从不符合预期的输出语句出发，把输出语句实例进行对齐，找到关键的输出语句实例，并在错误版本和正确版本中进行目标实例静态切片和动态切片，获得与直接错误相关的信息。静态切片的结果直接通过源代码级别的语义解释向使用者反馈代码语义区别。并对齐动态切片生成的结果，根据对齐结果生成反馈内容。

经过实验验证，本文提供的方法在对齐真实数据集提供的正确版本代码跟踪实例与错误版本代码跟踪实例的任务中，精确度高达 96%，在召回率上也达到了 96%。还验证了本工作中模型微调、阈值设置和扩充对齐对实验结果的影响。同时，对于错误的反馈生成率的平均值为 89.9%，并在五分制的方法有效性上进行调查，获得了 3.9 分的评分。

关键词： 编程学习；动态分析；深度学习；反馈生成；预训练模型

Abstract

With the continuous development of the Internet era, the enrollment of computer related majors is increasing, which makes the teaching pressure of college teachers become greater and greater. Meanwhile, more and more people are choosing to self-study programming outside of school. The Online Judge system plays a crucial role in programming learning, as it can automatically run code submitted by participants and evaluate the correctness and efficiency of code based on pre-set test cases. However, current systems typically only inform users of code errors or provide test cases for errors, but cannot assist users in identifying the root cause of the error and making modifications. Existing feedback generation tools usually look for the differences between the correct version code and the incorrect version code to generate feedback, but they often rely too much on contextual information, leading to incorrect feedback results. In addition, existing code tracking alignment tools such as APEX use symbol analysis for data alignment, but ignore differences in semantic information, which may result in low-quality alignment results and ultimately generate low-quality feedback.

This article proposes a deep learning based algorithmic programming homework automatic evaluation technique, with the main contributions being (1) a code automatic alignment method based on deep learning. Fine tune the ”all MiniLM-L6-v2” model to adapt to this work, then use dynamic analysis and symbol analysis methods of the program to obtain tracking information of correct and incorrect version codes and perform initial alignment. Then, use input statements as benchmarks to assist in deep learning model inference results, and correct and expand the initial alignment results. Finally, use the breadth traversal algorithm to expand the sequence pairs and generate the final alignment result. (2) Code automatic feedback generation technology based on deep learning. Starting from unexpected output statements, align the output statement instances, find the key output statement instances, and perform static and dynamic slicing of the target instance in the incorrect and correct versions to obtain information related to the direct error. The results of static slicing directly provide feedback to users on code semantic differences through source code level semantic interpretation. Align the results generated by dynamic slicing and generate feedback content based on the alignment results.

After experimental verification, the method provided in this article achieves an accuracy of up to 96% in aligning correct version code tracking instances and incorrect version code tracking instances provided by real datasets, and also achieves a recall rate of 96%. It also verified the impact of model fine-tuning, threshold setting, and expansion alignment on the experimental results in this work. Meanwhile, the average feedback generation rate for errors was 89.9%, and a survey was conducted

on the effectiveness of the five point method, resulting in a score of 3.9.

Keywords: Programming learning；Dynamic analysis；Deep learning；Feedback generation；Pre-trained model

目录

第一章 绪论

1.1 课题研究背景和意义

随着互联网时代的发展，计算机专业招生人数不断扩大[1]，自学编程技术的人员不断增加，教学者的压力激增。对于课堂教育而言，不同学员提交的代码往往是不同的，尤其是不能正确解决问题的代码，往往存在着各式各样的错误。教学者需要对个体的代码进行分析并给出反馈、与学员进行一对一的错误讲解以及给出客观公正的评分。而对于自学者，由于“教师”角色的缺失，自学者往往需要耗费更长的时间与更多的精力去发现自己代码的错误，这大大提高了自学编程的难度，延缓了他们的学习进度。

传统的编程学习平台，最常见的是在线评判系统 (Online Judge)[2–6]，通常采用黑盒[7] 的方式对代码进行测试，也就是说当学生提交自己的代码之后，平台会自动的使用预设的测试用例对学生的代码进行测试，根据通过代码的通过率来给学生反馈信息。这种方法导致了如果学生想要定位到自己的错误，这无疑会花费大量的时间，严重影响了学习的效率，对于某些生疏的问题，即使经过长时间的尝试定位，学生也可能很难找到自己的出错原因。除此之外，有些系统会提供给学生出错的测试用例来辅助学生定位错误位置[8]，这种方法虽然在某种意义上减轻了定位错误位置的负担，但是这也带来了“面向测试用例编程”问题的出现。这种过拟合的问题，与编程学习的目的背道而驰。

许多研究者为了解决这种在编程学习中的难点，提出了多种方法来自动分析学生的代码错误或者自动的来解决代码中的错误[8–21]。绝大多数的方法都是关注于编译级别的错误，而对于逻辑上的错误无法进行处理。即使是可以处理逻辑上错误的方法，大多数也是直接采用大模型对错误代码进行错误分析[22–25] 或者在基于“错误代码和正确的代码仅在出错的地方存在代码形式的不同”的基础上进行错误分析[26]。大模型难以对定制化的算法包装任务完成需求分析，一旦对算法题进行场景包装大模型就很难理解题目意思；而“错误代码和正确的代码仅在出错的地方存在代码形式的不同”这种理想化的假设往往是并不存在的，即使是相同解法的代码，它们的代码也往往有很大差别，一旦遇到这种问题，这些方法往往表现得不尽人意。APEX[26] 使用了动态分析和符号分析的方法来区分不同代码之间的语义区别和错误，然后对序列进行对齐之后再进行错误分析。然而由于不同代码的逻辑区别，在对齐过程中很容易出现错误对齐，导致后续的连锁反应，即连续的错误对齐，最终导致低质量的反馈出现。

本文提出了一种基于预训练模型[27] 的方法来解决在对齐序列过程中出现的各种问题，保证对齐后序列的正确性。在现有工作的基础上，本文结合了程序分析和深度学习模型，利用了程序的静态、动态分析以及符号表达。确定了对齐基准，使用深度学习模型对对齐结果进行纠正和扩充，并根据对齐结果进行静态切片[28] 获取错误信息的相关上下文信息，并在此基

础上使用 LLM[29] 对错误的代码生成更为准确的反馈信息，方便学生发现自己的错误，提高学习效率。

1.2 研究现状

国内外已经存在多种针对此类任务的研究。研究主要有两个方向，分别是自动程序修复技术和自动代码评分技术。

1.2.1 自动程序修复技术

自动程序修复（Automated Program Repair，APR）是指用于自动识别和修复软件程序中的错误或缺陷的技术和工具。APR 系统通常会分析代码，确定错误的根本原因，然后生成修复或补丁来纠正问题。

FixML[30] 提出了一种新的函数语言纠错算法，该算法结合了统计错误定位和类型导向程序合成，并通过分量减少和使用符号执行的搜索空间修剪来增强。CAFE[31] 面向一种函数式编程语言 Ocaml，从方法出发，使用独特的上下文相关的修复算法，提高了代码的修复率。但是 Ocaml 并不是常见的编程语言，其适用性有很大局限。TransplantFix[32] 使用基于图的差异算法从供体方法中提取语义修复操作。J Kim 等人[33] 从人工编写的补丁中收集抽象的 AST 变化及其 AST 上下文，为补丁生成提供丰富的资源。这些收集到的更改仅应用于具有相同上下文的可能修复位置以生成修补程序。CETI[34] 通过方法调用生成修补程序。Apex[26] 基于动态分析和符号分析[35] 来把错误版本和正确版本的动态跟踪信息来对齐，基于对齐结果生成反馈信息。ssFix[36] 从由本地故障程序和外部代码库组成的代码数据库（或代码库）中找到与目标块语法相关的候选代码块，即，在结构上相似且在概念上相关的候选码块。ssFix 假设候选块中包含正确的修复，并利用每个候选块为目标块生成补丁。SCRepair[37] 提出了用于程序修复的相似代码片段的可重用性，通过将程序语法树级别的相似性和差异性相结合，挑选出最合适的可重用候选者。BovInspector[38] 对于检测到漏洞，先通过分析 CFG 控制流图、符号执行分析，生成路径可达条件和漏洞触发条件取交后的约束，再进行约束求解，根据求解结果生成相应的修复代码。

自动程序修复技术在一定程度上解决了代码错误的问题，但是其正确度往往高度依赖于纠正数据库，无法保证其准确度；其次，即使正确修复了代码，仍旧无法实现“协助使用者发现错误原因并纠正”的核心目的。

1.2.2 自动代码评分技术

自动代码评分技术的目的在于降低教学者的负担，增加教学者的工作效率。评分标准主要取决于代码的正确性和判断提交代码是否符合题目要求。AutoGrader[19] 在学生提交的内容和参考实现之间搜索语义不同的执行路径来判断代码的对错。Web-Cat[3] 对学生提交的代码进行静态分析，检查代码的格式、结构和规范性；通过运行学生提交的代码，对其进行动态分析，检测代码的逻辑错误、性能问题和异常情况，根据静态和动态分析的结果，为学生的代码生成评分 Drummond 等人[17] 提出了一系列的用于计算提交代码之间的语义和句法距离的方法，寻找与已评分提交代码相近的其他提交代码，为这些代码赋予同样的分数，已达到减少教学者需评分代码数量的目的。J Clune 等人[39] 使用等效算法来增强手动评分过程，确定学生提交的材料之间的等效性，从而增加人工评分的效率。Paprika[40] 通过静态分析结合动态分析的方式进行自动评分。Tartare[41] 依赖于一种基于模式模板的方法。通过创建具有自动反馈的语句，学生有机会在不同的实例上随意练习。

自动代码评价技术是对使用者代码进行评分的技术。它可以提高教学者的效率，但无法对学习者生成个性化的反馈，学习者难以通过评价技术获得代码能力的提升。

1.3 本文主要研究工作概述

本文的主要研究工作是面向 C 语言初学者在完成算法题时提交的错误代码，结合程序分析技术和深度学习模型以及人工智能大模型，自动生成辅助修改错误的反馈信息。

1. 提出了一种基于深度学习模型的自动对齐算法，并根据算法完成了工具 CDAlign。通过结合动态和静态程序分析技术，获取到错误代码和正确版本代码的跟踪信息，基于符号表达和序列对齐算法将所得到的代码跟踪进行初始对齐并使用有序的输入语句作为纠正基准，之后依赖于深度学习模型对单挑代码语句的相似度进行判断，得到单条语句的相似度，并设定不同的阈值使其兼顾精度与召回率。最后通过扩充得到最终的对齐结果。

2. 从多个算法学习网站上，随机生成单条代码语句的代码对，人工标注数据，并使用数据作为训练集，对 SENTENCE-BERT 模型进行微调，使得 SENTENCE-BERT 模型对本类问题的处理更加精确。

3. 提出了一种基于深度学习、静态切片与大模型的反馈生成算法，并根据算法完成了工具 DPEX。找到错误代码和正确版本的代码的输出实例的错误位置，根据该实例使用静态切片工具，找到与错误信息有关的代码切片，随后使用大语言模型对代码切片进行分析，给出反馈结果。

4. 使用收集的实验数据和人工标注进行实验评价。在对齐方面，与相关的工具 APEX 对比，实验结果显示 DAlign 的精确度对比 APEX 有显著提高，使用了深度学习模型的相似度对

比，对源代码的比较更能提高精确度；在反馈生成方面，与相关工具 CLARA 比较，实验结果显示 DPEX 具有更高质量的反馈结果，与现有方法对比，更能辅助学生完成代码的修改。

1.4 论文内容安排

全文分为六个章节，主要安排如下：

第一章为绪论部分，提出课题研究的背景和意义，接着介绍当前的研究现状，引出本文的研究工作，最后介绍文章结构。

第二章对相关技术进行介绍，主要包括与本文相关的工具和技术以及评价指标。

第三章提出一种基于深度学习的自动对齐技术，并实现了工具 DPAlign。包括研究动机和方法设计。并详细介绍算法设计。

第四章提出一种基于深度学习的反馈生成技术，并实现了工具 DPEX。包括研究动机和方法设计。并详细介绍算法设计。

第五章分别对 DPAlign 和 DPEX 进行评估，对于 DPAlign，评估其精确度与召回率看，并研究其变体；对于 DPEX，评估其反馈生成率与反馈生成质量。

第六章是对全文的总结与展望，对本文基于深度学习的代码自动评价技术研究进行总结，并指出工作的不足与未来的研究方向。

第二章 相关背景知识介绍

2.1 程序分析技术

2.1.1 程序依赖

程序依赖图[42] 是描述程序结构和依赖性的一种图形表示，是计算机程序分析和优化的重要工具。程序依赖图通常表示程序中各个元素的依赖关系，包括变量、函数、模块和库等。它能够可视化程序中各个元素之间的依赖关系，帮助程序员和分析工具理解程序结构和性能瓶颈，从而优化程序代码。

程序依赖图可以分为控制流依赖图 (Control Dependence Graph) 和数据流依赖图 (Data Dependence Graph) 两种类型

控制流依赖图: 控制流依赖图描述了程序中的控制依赖关系，即程序在运行时的控制流动态分支情况，包括 if 语句、循环语句、函数调用等。这种依赖是程序单条指令的执行顺序和抉择关系，通常由有向图表示。控制流依赖图可以用于检测代码的控制流是否存在问题，并且可以帮助程序员识别并优化程序的性能瓶颈。

数据流依赖图: 数据流依赖图描述了程序中的数据依赖关系，即程序中各个元素之间的数据流动态关系，包括变量、常量、函数调用等。这种依赖是程序中指令的输入输出之间的关系，通常由有向图表示。数据流依赖图可以用于检测代码的数据依赖关系是否存在问题，例如重复计算、不必要的操作等，并且可以帮助程序员识别并优化程序的性能瓶颈。

2.1.2 程序跟踪

程序插桩 (instrumentation)[43] 是指向程序代码中注入额外的代码以收集其运行时行为信息的过程。这种技术通常用于代码分析、调试、性能优化等领域。程序插桩的应用场景很多，例如：

1. 代码分析：通过插桩收集程序运行时的行为数据，用于代码分析、代码覆盖率测试等。

2. 调试：通过插桩调试程序出现的问题，例如在程序的关键点处输出日志信息，区分程序是否正确执行。

3. 性能优化：通过插桩收集程序的运行时性能数据，找到程序的瓶颈，并对其进行优化。

本文使用程序插桩获取代码的跟踪信息，分析行为数据，包括控制流数据、依赖数据以及变量值跟踪。

2.1.3 符号执行

符号执行（Symbolic Execution）[35] 是一种基于数学逻辑符号和约束求解器的静态分析技术，通过标识程序代码中的符号变量，并将程序的执行路径和状态表示为符号表达式和符号约束，在进行程序路径探索和分析时，通过符号求解器求解约束，自动产生测试用例，以检测程序中可能存在的漏洞和错误。

符号执行的核心思想是将程序中的数据变量抽象成符号变量，即不考虑数据的具体值，而是考虑其可能具有的取值范围和限制条件。然后，程序的执行过程中，将数据变量的值表示成一组符号表达式，基于这些符号表达式和程序代码，产生一系列条件表达式，并使用符号求解器解决条件表达式，生成合法的输入数据，以探索各种路径和约束条件分支情况。

SAT 是用于检测布尔表达式的可满足性问题的技术，即确定是否存在布尔赋值使得给定的布尔表达式为真。SAT 问题是一个 NP 完全问题，因此通常采用启发式算法和 SAT 求解器进行求解。

SMT 是一种扩展了 SAT 的技术，用于处理包含多种理论的可满足性问题，如整数、数组、位向量等。SMT 问题可以表示为一组约束和一组推理规则，目标是找到满足所有约束的模型。SMT 求解器结合了定理证明和启发式搜索等技术，用于解决包含理论的可满足性问题。

符号执行技术的主要优点在于，可以自动发现程序中的所有执行路径和相应执行区间内的漏洞和错误，并生成测试用例。符号执行技术被广泛应用于软件测试、程序分析、漏洞和安全研究等领域。

2.1.4 抽象语法树

抽象语法树（Abstract Syntax Tree，AST）是编程语言中常用的一种数据结构，用于表示程序的语法结构。抽象语法树是编译器或解释器在对源代码进行语法分析（Parsing）后生成的一种树形结构，每个节点表示源代码中的一个语法结构，如表达式、语句、函数等，而节点之间的连接则表示这些语法结构之间的关系。

抽象语法树和源代码具有一一对应的关系，但是抽象语法树去除了源代码中的具体细节，例如括号、分号等，只保留了语法结构和关系，因此称为“抽象”语法树。通过抽象语法树，程序员可以更方便地理解和操作源代码，也可以进行一些代码分析和转换操作。

抽象语法树在编程语言处理中有着广泛的应用，它可以用于语法分析、语义分析、代码生成等环节。编译器和解释器通常会在语法分析阶段生成抽象语法树，然后通过遍历、分析和转换抽象语法树来完成后续的编译或解释工作。

2.2 神经网络语言模型

2.2.1 孪生网络

孪生网络[44]一般包括两个相同的神经网络[45]，每个网络输入一个样本，通过共享参数和权重进行训练，最终输出两个样本之间的相似度。孪生网络主要用于比较任务，例如比较两个文本的相似度、两张图像的相似度等。

假设输入的两个样本分别为 $\boldsymbol{x}_1$ 和 $\boldsymbol{x}_2$，经过神经网络处理后得到它们的特征向量为 $\boldsymbol{y}_1$ 和 $\boldsymbol{y}_2$，则网络的输出可以表示为：

$$o = \sigma\left(\boldsymbol{w}^T \left|\left|\boldsymbol{y}_1 - \boldsymbol{y}_2\right|\right|\right)$$

其中，$\sigma(\cdot)$ 是激活函数，$\boldsymbol{w}$ 是待学习的权值。$||\cdot||$ 表示 L_2 范数。$|\cdot|$ 表示向量取绝对值。

损失函数可表示为：

$$L(\theta) = \frac{1}{m}\sum_{i=1}^{m} y_i^{(true)} log(p_i) + (1 - y_i^{(true)}) log(1 - p_i)$$

其中，θ 是神经网络中的参数，m 为样本总数，$y_i^{(true)}$ 为样本 i 的真实标签，p_i 表示模型预测样本 i 的标签为 1 的概率。

优化算法常用的是梯度下降法或其变种，基于反向传播算法[46]。

总体来说，孪生网络的主要思想是将两个样本通过相同的网络生成表示，然后计算两个表示之间的差异，最终输出它们的相似度。

2.2.2 sentence-BERT

Sentence-BERT（SBERT）[47]是一种基于孪生网络的文本嵌入方法，旨在提供更好的句子级别表示。它通过进一步训练预训练的 Transformer[27] 模型，以便在不同的句子之间生成具有相似含义的嵌入。SBERT 在自然语言处理中的应用十分广泛，例如在句子相似度判断、垃圾邮件过滤、自然语言推理等任务中都产生了良好的效果。

具体来说，SBERT 的核心思想是在预训练的 Transformer 模型基础上添加一个新的网络结构，通过相似度损失函数进行训练，从而得到句子对的嵌入表示。SBERT 包括三个主要部分：

1. Sentence encoder：使用 Transformer 编码器处理每个输入句子，生成其定长的向量表示。

2. Pooling strategy：对于一个句子对，需要从两个向量中合成一个定长的向量表示。SBERT 使用了不同的 Pooling 策略，例如 Mean Pooling 和 Max Pooling 等，来获取每个句子的最重要的特征。

3. Similarity function：将合成的向量表示输入到相似度函数中，计算出两个句子的相似度得分。

假设需要比较的两个句子分别为 a 和 b，其通过 Transformer 编码器得到的向量分别为 $\mathbf{h}_a$ 和 $\mathbf{h}_b$，使用 Pool 策略 $pool$，得到的向量分别为 $\mathbf{v}_a$ 和 $\mathbf{v}_b$：

$$\mathbf{v}_a = pool(\mathbf{h}_a)$$

$$\mathbf{v}_b = pool(\mathbf{h}_b)$$

SBERT 使用余弦相似度来计算两个向量之间的相似度，并且引入了两个附加参数 W 和 b，用于归一化分数：

$$\text{similarity}(a,b) = \frac{\mathbf{v}_a^T W \mathbf{v}_b + b}{\left\|\mathbf{v}_a^T W\right\| \left\|\mathbf{v}_b^T W\right\|}$$

最终的训练目标是通过 K 个正样本和 K 个负样本构成的 mini-Batch 计算三元组损失，使每个正样本与其对应的负样本之间的距离尽可能小，与非对应的句子之间的距离尽可能大：

$$L = \frac{1}{N}\sum_{i=1}^{N}\left[\left\|v_i^+ - v_i^-\right\|^2 + \alpha - \left\|v_i^+ - v_i\right\|^2\right]_+$$

其中，α 是边界参数，$[x]_+ = max(0,x)$ 是 ReLU 函数。$\left\|v_i^+ - v_i^-\right\|^2$ 代表正样本与负样本之间的距离，$\left\|v_i^+ - v_i\right\|^2$ 代表正样本与非对应的句子之间的距离。

综上所述，SBERT 是一个在预训练模型的基础上添加新模块的方法，能够生成句子对间的嵌入表示，此模型相比于传统的嵌入方法，具备更多优势。

2.3 相关工具

本节介绍在工作中所使用的一些主要工具，主要包括编译器基础架构 LLVM(Low Level Virtual Machine)[29]，SMT 求解器 Z3[48]，代码静态分析工具 joern[1]。

LLVM（Low Level Virtual Machine）是一个编译器基础架构，其目的是提供灵活、可扩展的程序编译器开发框架。LLVM 最初是作为编译器中间表示（IR）的一种表示形式的设计，但现在已经演变成一个完整的编译器开发框架，可以用于构建高性能、优化的编译器。LLVM 的能够支持多种不同的编程语言和目标架构。LLVM 提供了一组模块化的工具和库，其中包括（Frontend）、优化器（Optimizer）和后端（Backend）。不同的编程语言的前端可以将源代码转换成通用的 LLVM IR，然后 LLVM 的优化器可以对 IR 进行各种优化，最后后端可以将优化后的 IR 转换成目标机器的机器码。LLVM 的优化器是其核心组件之一，它提供了许多高

[1]https://joern.io/

级优化技术，如函数内联、循环优化、常量传播等，这些优化技术可以显著提高程序的性能。另外，LLVM 还提供了一种称为 Just-In-Time（JIT）[49] 编译器的技术，这使得程序可以在运行时即时编译成机器码，从而提高程序的执行效率。本文使用 Clang 获取代码跟踪信息。

Z3 是由微软研究院开发的一种自动定理证明器，用于解决数学逻辑和计算机科学领域的各种问题。Z3 求解器基于 SMT（Satisfiability Modulo Theories）理论，可以处理包括布尔逻辑、整数和实数算术、位向量、数组、以及复杂数据结构在内的多种理论。它能够自动推导和证明公式的可满足性或不可满足性，提供了一种高效、快速的方法来解决各种约束满足问题。本文使用 Z3 求解器判断符号表达式是否相等。

Joern 是一款开源安全分析平台，用于分析和可视化大型代码库的漏洞和安全风险。Joern 基于模块化架构，包括解析器、存储库、查询引擎和可视化引擎四个模块。解析器将源代码解析成中间表示；存储库存储解析后的代码和分析结果；查询引擎使用 Joern 查询语言 (JQL) 执行查询；可视化引擎生成交互式可视化来探索分析结果。Joern 可以执行静态分析、动态分析、控制流分析和数据流分析。本文使用 Joern 生成程序控制图，并使用程序控制图生成切片。

2.4 评价指标

为了评估我们工作对齐的性能，需要将对齐结果与人工标注的结果进行比较。采用的评价指标有精确率 (Precision)、召回率 (Recall) 和两者的调和平均 F_1Score。

精确率 (Precision) 衡量的是预测为正例的样本中实际为正例的比例，即对于所有预测为正例的样本，有多少是真正的正例。公式定义如下：

$$Precision = \frac{TP}{TP + FP} \tag{2.1}$$

召回率 (Recall) 衡量的是实际为正例的样本中被正确预测为正例的比例，即对于所有真正的正例样本，有多少被正确地预测为正例。公式定义如下：

$$Recall = \frac{TP}{TP + FN} \tag{2.2}$$

F_1Score（F1 分数）是精确率（Precision）和召回率（Recall）的调和平均数，综合考虑了分类模型的准确性和完整性。F1 Score 的取值范围在 0 和 1 之间，值越接近 1 表示模型的性能越好。公式定义如下：

$$F_1 Score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{2.3}$$

在公式中，TP 表示真正例（True Positives），即算法计算出对齐并且在人工标注中也被对齐的实例对；FP 表示假正例（False Positives），即算法计算出对齐但是在人工标注中没有被对齐；FN 表示假负例（False Negatives），即算法计算出不被对齐但是在人工标注中被对齐。

2.5 本章小结

本章主要介绍了我们与我们工作相关的一些知识背景，包括程序分析技术、神经网络语言模型等；接着介绍了一些我们在工作中所使用到的一些工具，包括 LLVM、Z3 和 joern 等；最后介绍了对我们工作的评价指标精确率、召回率和 F_1Score。

第三章 基于深度学习的代码对齐

本章提出一种基于深度学习的代码自动对齐技术，并完成了工具 CDALIGN。首先通过现有工作的错误引出研究动机，之后简单介绍本方法的实现步骤，最后对方法中的算法进行详细的介绍。

3.1 研究动机

现阶段基于动态分析的对齐方法都过于依赖变量的值与数据依赖。他虽然使用符号分析解决了变量名所带来的无法对齐的缺陷，但是这又引入了新的问题，即赋值语句的错误对齐。这就导致了精确度的验证损失，以至于影响反馈的正确性，导致错误的方向引导。

他将赋值语句与循环语句的初始赋值进行了对齐。这导致了其版本的循环语句并不能与另一版本的循环语句完美对齐，这在后续的对齐与反馈生成中是灾难性的。

错误对齐的原因是过度依赖于符号表达，变量的值在对齐过程中占据了太多的权重。这导致了一旦赋值属性相同，就会把两个语句对齐，这种不同司职的变量被赋予相同的值在编程中是极其常见的，尤其是在涉及到循环语句的初始化问题上。因此我们引入了一种基于深度学习的技术来解决这种错误的对齐。图 3.1展示了来自 $CODECHEF$[1] 中两段解决 $ALEXTASK$[2] 问题的代码，其中代码3.1a是能编译通过并运行但是存在逻辑错误不能完美解决问题的代码，代码3.1b是可以完整解决问题的代码。

问题描述： Alexey 正在为一个非常简单的微处理器开发程序。程序的作用是定时从若干传感器采集数据。但问题是，一定不能在同一时刻从多个传感器采集数据。一旦发生，微处理器会陷入死锁，必须进行重置。

有 N 个需要定时采集数据的传感器。第 i 个传感器每隔 A_i 毫秒需要进行一次采集，而第一次数据采集发生在微处理器启动后 A_i 毫秒的时刻。进行一次数据采集需要恰好 1 毫秒。

Alexey 想要知道，在他启动微处理器后，经过多少毫秒微处理器会陷入死锁。

输入格式： 输入的第一行包含一个整数 T，代表测试数据的组数。接下来是 T 组数据。每组数据的第一行包含一个整数 N，代表传感器的数量。接下来一行包含 N 个整数 $A_1, A_2, ..., A_N$，代表每个处理器的数据采集间隔。详细地说，第 i 个传感器在微处理器启动后的 Ai 毫秒后进行第一次数据采集，然后每隔 Ai 毫秒进行一次数据采集。

[1] https://www.codechef.com

[2] https://www.codechef.com/problems/ALEXTASK

输出格式：对于每组数据，输出一行，包含一个整数，代表微处理器陷入死锁的时刻，以微处理器启动后经过的毫秒数表示。

```
#include <stdio.h>
int gcd(int a,int b) {
    if(a==0)
    return b;
    return gcd(b%a,a);
}
int main(void) {
    int t,n,a[500];
    scanf("%d",&t);
    while(t--) {
        scanf("%d",&n);
        int c,b;
        for(int i=0;i<n;i++) {
            scanf("%d",&a[i]);
        }
        int max=0;
        for(int j=0;j<n;j++) {
            for(int k=j+1;k<n;k++) {
                if( gcd(a[j],a[k])>max) {
                    c=a[j];
                    b=a[k];
                    max=gcd(a[j],a[j+1]);
                }
            }
        }
        int result=(b*c)/max;
        printf("%d\n",result);
    }
return 0;
}
```

(a) 错误代码

```
#include<stdio.h>
#include<limits.h>
int gcd(int a, int b) {
    if(a == 0)
    return b;
    return gcd(b%a, a);
}
int main() {
    int t;
    scanf("%d", &t);
    while(t--) {
        int i, j, n;
        scanf("%d", &n);
        long long a[n];
        for(i=0; i<n; i++)
            scanf("%lld", &a[i]);
        int min = INT_MAX;
        for(i=0; i<n-1; i++)
            for(j=i+1; j<n; j++)
                if((a[i]*a[j])/gcd(a[i], a[j]) <
    min)
                    min = (a[i]*a[j])/gcd(a[i], a[j
    ]);
        printf("%d\n", min);
    }
}
```

(b) 正确代码

图 3.1 ALEXTASK 代码示例

图3.2展示了错误版本代码和错误版本代码的输入、输出和当前状态。步骤 1 初始化了测试用例的数量，表示当前代码只有 1 组测试用例。步骤 2 表示当前用例有多少个传感器，当前的输入表示这组测试用例有 5 个传感器。步骤 3 表示当前 5 个传感器的数据采集间隔分别是什么。步骤 4 表示找出的出现死锁的时间。本算法的任务是求出数组中任意两个元素的最小公倍数。错误代码的错误原因是其逻辑出现错误，认为最小公倍数的求解方式是先求出数组中任意两个元素的最大公约数，在用其乘积除以最大公约数。

BPEX 的错误对齐：APEX 使用符号表达来构建对齐，Bpex 使用基于信念传播的概率推断来解决 APEX 中关于对齐的不确定性。但是在对齐过程中，符号表达的缺点依然没有被避

#	输入	输出	状态
1	1		共有 1 组测试数据
2	5		共有 5 个传感器
3	2 3 6 4 3		每个传感器的数据采集间隔
4		6	在第 6 毫秒产生死锁

(a) 错误执行

#	输入	输出	状态
1	1		共有 1 组测试数据
2	5		共有 5 个传感器
3	2 3 6 4 3		每个传感器的数据采集间隔
4		3	在第 3 毫秒产生死锁

(b) 正确执行

图 3.2 错误执行和正确执行

local	src	value	local	src	value
...	...	...	...	...	...
24_1	int max=0	0	23_1	for(i=0; i<n-1; i++)	0
25_1	for(int j=0;j<n;j++)	0			
...	...	...	...	...	...

图 3.3 BPEX 错误对齐

免。如图3.3所示，BPEX 在对齐过程中，错误的将错误版本代码第 24 行代码的第一个运行实例“int max=0”与正确版本代码第 23 行代码的第一个运行实例“for(i=0; i<n-1; i++)”对齐。导致了错误版本代码的第 25 行代码第一个实例“for(int j=0;j<n;j++)”在正确版本代码中找不到与其对应的实例。这种类型的错误对齐是十分常见的；在两个版本的代码中，无法保证变量初始化与循环语句初始化不出现在相同位置，而这种错误的对齐会导致丢失或者错误的反馈。BPEX 使用了概率推断和信念传播的方法来解决对齐的不确定性，这种对齐无论是从数据上还是信念传播所传播的概率上都是应当被对齐的；但是这种对齐具有最直观的区别就是其源代码的区别，因此我们提出了一种基于深度学习模型的方法增加源码的权重来避免这种错误的对齐。

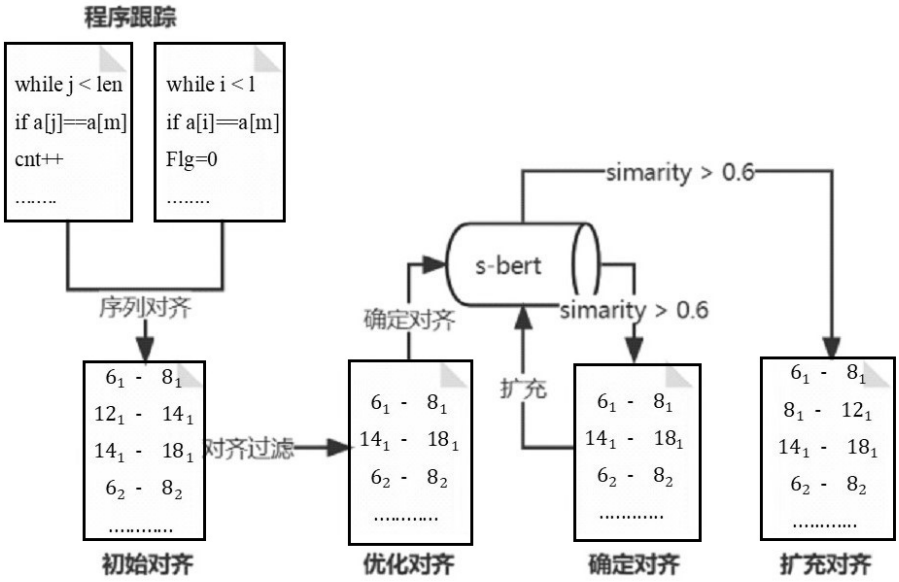


图 3.4 CDALIGN 方法流程图

3.2 方法概述

我们实现的工具 CDALIGN 的总体流程如图3.4所示，对齐过程包括有初始对齐、优化对齐、确定对齐与扩充对齐，SBERT 模型在其中进行相似度判断。优化对齐与确定对齐在下文中统一称之为对齐过滤。

模型微调: Sentence-BERT 作为 BERT 的改进版本，专注于对句子语义进行建模，被证明在多个句子级任务中取得了优异的性能。为了使用 sentence-bert 对单条代码语句进行相似度判断，我们需要对 sentence-bert 进行微调来使其适应此任务，保证判断的正确性。由于代码语句的简单性，我们选择了较为轻量的”all-MiniLM-L6-v2” 模型，并在其基础上进行微调，完成我们的任务。

初始对齐: 使用 LLVM 对正确版本代码与错误版本代码进行插桩，完成程序跟踪，得到代码跟踪结果。然后使用序列对齐算法将错误版本与正确版本的追踪结果进行对齐。请注意，这里的对齐并不考虑到准确性，而是最大可能的生成更多的对齐结果。因此，在此阶段仅仅将符号表达作为关注点。

对齐过滤: 在初始对齐的过程中，由于符号表达占用了极大的权重，因此并不能保证初始对齐的结果都是正确的。在常规问题中，有很多对齐都是错误对齐。在对齐过滤中，我们注意到了输入数据的确定性，以输入的控制依赖作为基准，对初始对齐进行过滤，过滤掉错误的对齐。

最终对齐: 在通过初始对齐与对齐过滤之后，因为初始对齐的目的仅仅是为了获得最大

的序列对，对对齐的准确度并不做要求，因此，经过对齐过滤后生存下来的序列对往往是不完整的。为了解决这个问题，我们使用一种类似广度遍历的算法进行序列对的扩充。

3.3 模型微调

3.3.1 构造数据集

在本工作中，需要使用两两相对的代码对进行微调 SBERT，由于是新提出的工作，所有数据均由实验室内人员人工收集并标注。我们使用爬虫工具从 LEETCODE[3]收集了 10 道使用 C 语言代码实现的算法问题的官方解答，并以行为单位，对其进行随机配对，生成 3000 条随机组合，配对完成的一组中包含有两条代码语句，之后再从中随机挑选出 450 组数据，最后再并人工标注其相似度，相似度分别标注为 0.1、0.3、0.5、0.7、0.9 、1.0，其中，0.1 表示两条代码语句毫无关系，0.3 表示两条代码语句不相似，0.5 表示两条代码语句略微相似，0.7 表示两条代码语句相似，0.9 表示两条代码语句非常相似，1.0 表示两条代码语句完全相同。数据选取采用“蓄水池算法”，他动态地选择一部分数据作为样本，使得每个元素被选中的概率相等。这样可以保证样本的随机性和代表性，从而有效地进行统计分析或其他处理。

3.3.2 选取预训练模型

在本工作中，所需模型的核心功能为：单条语句代码的语义相似度判断。其次，为了工具的效率以及性能限制，所需模型需要具有轻量化的性质。因此我们选择了 sentence-bert 下的”all-MiniLM-L6-v2” 轻量化模型。all-MiniLM-L6-v2 是一个经过训练的语言模型，采用了 MiniLM 架构，具有 6 层的深度，将语句转化为 384 维的向量。该模型旨在生成自然流畅的文本，并对各种主题进行理解和回应。MiniLM 是一种轻量级的语言模型架构，它在保持模型大小相对较小的同时，尽可能地提高了性能。这使得 MiniLM 系列模型在资源受限的环境下也能够提供良好的性能表现。all-MiniLM-L6-v2 模型在各种自然语言处理任务中都表现出色，可以应用于多种应用场景。

3.4 初始对齐

3.4.1 程序跟踪

本文使用 LLVM 分析程序的执行过程，获取程序跟踪信息。跟踪信息由语句实例的集合组成，语句实例为单条语句在单次执行的所有信息，包括但不限于源代码信息、实例的直接

[3]https://leetcode.cn/

控制依赖、实例的直接数据依赖、变量值与操作数组成。在本文中，具体实例由 f_i 和 t_j 表示，分别表示错误代码 f 的第 i 次执行和正确代码 t 的第 j 次运行。

本文用于对齐的语句实例包括赋值语句 (Store Instance,STIns)、分支语句 (Branch Instance,BRIns) 和输出语句 (Print Instance,PRINT)，由输出结果和变量值组成。输出结果就是输出语句中输出的字符串等信息，变量值是在语句执行过程中赋予变量的值。并在变量值基础上应用符号表达来抽象化变量的赋值操作。根据这些信息来获得所需要的程序跟踪信息。

3.4.2 序列对齐

算法3.4.2描述了尼德曼-翁施 (Needleman-Wunsch)[50] 序列对齐算法，算法的目的是寻找两个序列中最大化对齐数量的实例对。该算法利用动态规划方法求得最佳的全局配对方案。算法的基本思想是对两个输入序列进行逐步比对，选定最佳的配对方案，并根据该方案生成输出序列。其匹配算法由得分机制驱动，对矩阵中的每个位置 (i, j)，根据 N-W 算法的规则计算其得分，即匹配得分、插入得分和删除得分。需要注意的是，匹配得分和非匹配得分的大小是根据一组预设的性质而定的。本文场景下，得分机制见 SCORE 函数，当两个实例具有等同的符号表达式时得 1 分，否则得 0 分，也就是说，当符号表达不相同时，其应该对齐的概率无线接近于 0 。

该算法接受两个跟踪序列 $E1$ 和 $E2$ 作为输入，并初始化一个算法分数矩阵 $Align$。在这个矩阵中，i 代表 $E1$ 序列的第 i 个实例，j 代表 $E2$ 序列的第 j 个实例。在第 2 到第 8 行，算法计算矩阵中的每一项 $Align[i_1][i_2]$。如果 $SCORE(E_1[i_1], E_2[i_2])$ 为 1，那么当前项取左上角的值加 1，意为直接连续；否则按照从左上、左、上方向的顺序取最大值，意为找到当前最长可连续序列对。随后，在第 12 到第 22 行，算法对矩阵进行回溯，从右下角的单元格开始。如果 $SCORE(E_1[i_1], E_2[i_2])$ 为 1，则回溯到左上角的单元格，并将 $(E_1[i_1], E_2[i_2])$ 加入对齐序列 $alignments$ 中。否则，按照从左上、左、上方向的顺序回溯到具有最大值的单元格，直到超出矩阵范围。最后，将 $alignments$ 作为序列对齐的结果返回。

算法 1 序列对齐算法

procedure SEQUENCEALIGN(E_1, E_2)
  for $i = 1; i < E_1.length; i++$ **do**
    for $j = 1; j < E_2.length; j++$ **do**
      $Align[i][j] = max($
      $Align[i-1][j-1] + SCORE(E_1[i], E_2[j]),$
      $Align[i-1][j], (Align[i][j-1])$
    end for
  end for
  $alignments = \{\}$
  $i = E_1.length$
  $j = E_2.length$
  while $i \geq 1$ *and* $j \geq 1$ **do**
    if $Align[i][j] == Align[i-1][j-1] + SCORE(E_1[i], E_2[j])$ **then**
      $alignments = alignments \cup (E_1[i], E_2[j])$
      $i = i - 1$
      $j = j - 1$
    else if $Align[i][j] == Align[i-1][j]$ **then**
      $i = i - 1$
    else if $Align[i][j] == Align[i][j-1]$ **then**
      $j = j - 1$
    end if
  end while
  return $alignments$
end procedure
procedure SCORE(t_i, t_j)
  if $symb(t_i) == symb(t_j)$ **then**
    return 1
  else
    return 0
  end if
end procedure

3.5 对齐过滤

3.5.1 对齐约束

初始对齐建立之后，其余对齐都要建立在初始对齐的基础上。在获得进一步对齐的过程中，对齐结果总是由多种约束产生。每一个被对齐的单例对，都需要满足多个约束才会被最终放入对齐结果之中。除此之外，已被对齐的单例对对其依赖背书，即已被对齐的单例对，他们的控制依赖也存在对齐可能。通过不断的迭代来获得对齐结果。图3.5展示了在对齐过程中的一些约束定义和规则。

定义：

τ, ϵ：正确代码语句标签，学生代码语句标签

τ_i, ϵ_j：语句 τ/ϵ 的第 i/j 个实例

I_{τ_i,ϵ_j}：语句 τ_i 和 ϵ_j 已被对齐

L_{τ_i}：包含语句 τ_i 的循环语句

Value(τ_i)：语句 τ_i 中的变量值

Symb(τ_i)：语句 τ_i 中的符号表达式

Igno(τ_i, ϵ_j)：语句 τ_i 和 ϵ_j 不存在对齐可能

cdep(τ_i, n)：语句 τ_i 距离为 n 的控制依赖的集合

ddep(τ_i, n)：语句 τ_i 距离为 n 的数据依赖的集合

约束规则：

1.Value(τ_i) $\neq$ Value(ϵ_j) $\Rightarrow$ Igno(τ_i, ϵ_j)

2.$I_{\tau_i,\epsilon_j} \Rightarrow I_{L_{\tau_i,\epsilon_j}}$

3.Value(τ_i) = Value(ϵ_j) $\cap$ Symb(τ_i) $\neq$ Symb(ϵ_j) $\cap$ $I_{L_{\tau_i,\epsilon_j}} \nRightarrow$ Igno(τ_i, ϵ_j)

4.Symb(τ_i) $\neq$ Symb(ϵ_j) $\Rightarrow$ Igno(τ_i, ϵ_j)

5.$I_{\tau_i,\epsilon_j} \iff \forall \tau_p \in \text{cdep}(\tau_i, n), \exists \epsilon_q \in \text{cdep}(\tau_i, n), I_{\tau_p,\epsilon_q}$

6.$I_{\tau_i,\epsilon_j} \iff \forall \tau_p \in \text{ddep}(\tau_i, n), \exists \epsilon_q \in \text{ddep}(\tau_i, n), I_{\tau_p,\epsilon_q}$

图 3.5 对齐约束

约束规则 1 表示：如果两个实例 τ_i 和 ϵ_j 具有不同的变量，那么他们不存在对齐的可能。

在后续的对齐过程中不考虑计算两个实例的对齐可能性。

约束规则 2 表示：如果两个实例 τ_i 和 ϵ_j 都处在循环语句中，并且他们的循环语句已被对齐，那么他们也是应该被重点关注的对齐实例。该规则是可逆的，即当两个实例已被对齐并都存在于循环语句之中，其包含他们的循环语句也应该被重点关注。该规则基于循环语句的不同表达方式与不同的代码风格产生。

mid = (start+end)/2	mid = (end-start)/2 + start

约束规则 3 表示：如果两个实例 τ_i 和 ϵ_j 都处在循环语句中，并且他们的循环语句已被对齐，但是他们的符号表达不相同，这种情况无法将其判断为无法对齐。该规则关注于对同一结果的不同计算方法。两个相同的结果可能由多种计算方法得出，当其存在于循环语句中时，往往担任及其重要的角色。上图两个代码片段展示了在代码编写中十分常见的两种计算 *start* 和 *end* 变量的平均值的方法，左边代码把两者相加之和除以 2 以获得中间值 *mid*，而右边代码通过先计算 *end* 和 *start* 的差值，再把差的一半加上 *start* 获得。这类语句在循环中往往担任重要角色，需要重点关注其对齐结果。

约束规则 4 表示：在不满足规则 3 的情况下，如果两个实例 τ_i 和 ϵ_j 具有不同的符号表达式，代表其在算法流程中不充当重要角色，不会是直接错误原因，将其暂时忽略。

约束规则 5、6 表示：如果两个实例 τ_i 和 ϵ_j 已经对齐，那么，我们需要重点关注其控制依赖和数据依赖的对齐情况。该约束规则是可逆的，即两个实例的依赖已经被对齐，这两个实例也应当被重点关注。即使实例对已被规则 4 所忽略，在本规则情况下需要将其重新纳入对齐判断中。

3.5.2 优化对齐

在调研过程中，算法题目存在同一种共性，即不管是什么题目，他的输入总是有迹可循的。即不论代码的正确性与否，每个代码都必须保证输入的数量与每个输入的顺序相同。因此得出结论，每个对应输入的控制依赖是必然对齐的。例如，假设变量的输入顺序为 $param_1, param_2, param_3$，对变量 $param_1$，在错误版本的代码中，他的输入语句的直接控制依赖为 $cdep_w$，在正确版本中为 $cdep_c$，那么 $cdep_w$ 与 $cdep_c$ 是必然对齐的。其次，编程语言中的规范化语法与语句决定了其同样功能的代码语句，源代码语句总是高度相似的。影响源代码语句相似度的关键信息主要在于不同学生变量名的创建习惯。APEX 过分夸大了源代码所做出的错误引导，采用了符号表达来完全舍弃源代码的信息，这是有失偏颇的。即符号表达不能独立于源代码之外来决定代码语句的对齐与否。为了获得同样的结果，不同的计算表达式总是导致不同的符号表达式，然而其语义信息却具有高度的相似性；其次，即使是相同的符号表达也无法武断的认

为这两个代码对应该被对齐，不同司职的单条代码也很有可能存在相同的符号表达。上文中约束规则 3 就是问题显著的代表。因此我们引入了深度学习模型 Sentence-BERT（SBERT）来根据源代码语义的相似性来进行确定对齐。

算法 FIRSTALIGN 描述了如何获得确定对齐序列。算法输入语句为 S_1 和 S_2，S_1 来自错误版本，S_2 来自正确版本。$firstAlignment$ 存储了他们所有的控制依赖。为了获取 $firstAlignment$，将两个版本的输入语句 S_1 和 S_2 排序并组对后，依次将其的直接控制依赖存储到 $firstAlignment$ 中。

算法 REFINE 描述了如何优化 $oldAlignment$。获得 $firstAlignment$ 后，将 $firstAlignment$ 和上文做的序列对齐结果 $oldAlignment$ 作为输入，遍历 $oldAlignment$ 里的序列对 (i, j)，如果 $firstAlignment$ 中存在序列对 (i, j) 中的任意一项，且序列对的另外一项并不在 $firstAlignment$ 中，即 $firstAlignment$ 和 $oldAlignment$ 存在对齐冲突，因为 $oldAlignment$ 并没有对对齐的正确性作出精确度要求，而 $firstAlignment$ 中的序列对均为高度确认过的对齐序列，所以直接剔除掉 $oldAlignment$ 中与 $firstAlignment$ 存在冲突的序列对，作为最初步的优化。

算法 DETERALIGN 描述了如何进一步处理经过 $firstAlignment$ 优化过后的 $oldAlignment$。优化后的对齐结果 $alignment$ 中，并不能认为其已经完全不存在错误的对齐结果。编程语言中的规范化语法与语句决定了其同样功能的代码语句，源代码语句总是高度相似的，相同的符号表达并不能精确的确认两条代码的对齐结果。因此我们使用 sentence-bert 模型来针对单条源代码的语义信息进行判断，判断对齐结果 $alignment$ 中的每一对代码语句是否具有相似的语义信息。由 SBERT 可以生成实例对的相似度，我们可以设置可变阈值作为认定是否应该对齐的基准。在本文工作中，我们将阈值设置为 0.6，即相似度超过 0.6 的实例对我们认为他们是应该对齐的。

算法 2 优化对齐算法

```
1: procedure FIRSTALIGN(S_1, S_2)
2:     firstAlignment = {}
3:     for i = 1.j = 1; i < S_1.length, j < S_2.length; i++, j++ do
4:         firstAlignment = firstAlignment ∪ (S_1.cdep, S_2.cdep)
5:     end for
6:     return firstAlignment
7: end procedure
8: procedure REFINE(firstAlignment, oldAlignment)
9:     newAlignment = {}
10:    for (i, j) in oldAlignment do
11:        if (i, j) in oldAlignment or (i not in firstAlignment.0 and j not in firstAlignment.1)
   then
12:            newAlignment = newAlignment ∪ (i, j)
13:        end if
14:    end for
15:    return newAlignment
16: end procedure
17: procedure DETERALIGN(alignment)
18:    deterAlignment = {}
19:    for i = 1; i < alignment.length; i++ do
20:        if SIMARITY(alignment[i].0, alignment[i].1) > 0.6 then
21:            deterAlignment = deterAlignment ∪ (alignment[i].0, alignment[i].1)
22:        end if
23:    end for
24:    return deterAlignment
25: end procedure
```

3.6 扩充对齐

基于上文的确定对齐之后，我们已经可以认定存留下的序列对都是必然对齐的。然而由于存在没有进行任何处理的实例，因此会导致对齐的缺失。为了解决这个问题，我们使用了基于 *worklist* 算法驱动的一种广度遍历的方法进行对齐实例的扩充。首先将上文已对齐的所

有序列对存储到 *worklist* 之中，并初始化最终结果 *result* 列表为空，之后循环弹出 *worklist* 中的单个实例对，并存入最终结果 *result* 列表中。弹出的单个实例对，分别判断其正确版本和错误版本的控制依赖、数据依赖的变量值以及其代码相似度。如果他们的变量值相同并且相似度超过阈值，即认为他们应该被扩充，将控制依赖对与数据依赖对存入 *worklist* 列表中，直到 *worklist* 列表为空，即到达稳态，结束扩充，获得最终结果。

算法 3 对齐扩充算法

```
1: procedure PROPAGATE(alignments)
2:     result = {}
3:     worklist = {alignments}
4:     while worklist is not empty do
5:         (I_wt, I_ct) = worklist.pop
6:         result = result ∪ (I_wt, I_ct)
7:         if I_wt.cdep.value == I_ct.cdep.value and SIMARITY(I_wt.cdep, I_ct.cdep) > 0.6 then
8:             worklist = worklist ∪ (I_wt.cdep, I_ct.cdep)
9:         end if
10:        if I_wt.ddep.value == I_ct.ddep.value and SIMARITY(I_wt.ddep, I_ct.ddep) > 0.6
   then
11:            worklist = worklist ∪ (I_wt.ddep, I_ct.ddep)
12:        end if
13:    end while
14: end procedure
```

3.7 本章小结

对于错误版本代码和正确版本版本代码跟踪的对齐问题，本章提出了一种基于深度学习的代码对齐方法，并根据方法实现了工具 CDALIGN。首先通过微调 “all-MiniLM-L6-v2” 模型来使其适应本工作，接着使用程序的动态分析和符号分析方法获取正确版本代码和错误版本代码的跟踪信息并进行初始对齐，然后使用输入语句作为基准，辅助以深度学习模型推测结果，对初始对齐结果进行纠正与扩充。最后使用广度遍历算法扩充序列对，完成扩充生成最终对齐结果。

第四章 基于深度学习的代码自动反馈生成

由于当前方法的对齐精度不足，对错误代码的修正反馈质量不高这两种情况，为了获得更高质量的反馈，增加学生的学习效率。本章提出了一种基于深度学习的代码自动反馈生成技术，实现了工具 DPEX。

4.1 研究动机

当前绝大多数自动代码分析和错误修复工具仅能在语法或编译出错的情况下做出修改，即使是可以对逻辑错误生成反馈，也只是将错误版本代码和正确版本代码进行匹配，并且基于数据庞大的数据库，根据上下文信息寻找类似的代码作为修改反馈。

现有工作忽视了相同的代码功能在实现上也会存在差异[26]，也可能无法匹配导致错误的反馈。现有工作的反馈的质量高度依赖于对齐结果的正确性，如果出现错误的对齐，也会出现误导性的反馈，严重影响学习效率。除此之外，现有工作并不能在源代码层面给出修改建议，无法辅助学生理解正确版本和错误版本源码的区别。

APEX 通过程序的动态分析和符号表达分析弱化代码实现的不同导致的错误对齐。这种方法在一定程度上缓解了错误，但依然存在很多缺陷，导致低精确度的对齐，进一步导致了不正确的反馈结果。

CDALIGN 将错误版本代码与正确版本代码对齐，使用深度学习模型来引入源代码级的语义对齐，并通过输入语句作为基准纠正可能的错误对齐，提高了对齐的精确度。并使用动态分析与静态分析结合，生成不仅包含错误也包含有源代码辅助理解的反馈。

4.2 方法概述

对齐阶段调用 CDALIGN 方法获取对齐结果，使用 BPALIGN 中生成的跟踪序列。之后经过输出语句对齐、目标实例切片、动态切片对齐三个阶段，最后分别根据跟踪信息和源代码两个角度生成反馈。

目标实例对齐阶段：不符合正确答案预期的输出语句即为出错位置。然而输出语句只是错误出现而不是错误原因。它们提供了错误所在的位置，但并未揭示错误产生的原因。我们认为与输出语句直接相关的语句为直接错误。直接错误可能存在于错误版本代码中，也可能存在于正确版本代码中，也有可能两个版本都存在。为了获得与错误相关的信息，我们从直接错误也就是不符合正确答案的输出语句出发，将错误版本的输出语句与正确版本的输出语句对齐，找到出错的位置。

目标实例切片阶段：输出语句对齐之后，存在无法对齐的输出语句即为目标实例。为了获取到所有有可能出错的信息，需要对目标实例进行切片。我们使用动态切片和静态切片两种切片方式。动态切片考虑到被切片位置的数据依赖和控制依赖。静态切片获取有关错误位置的源代码。

切片对齐阶段：与错误相关的语句实例包含在错误版本代码的动态跟踪信息中，同样的与错误没有直接关系但是无法被对齐的语句也包含在内。

反馈生成阶段：基于动态切片对齐与静态切片，分别生成基于动态和静态的自然语言的反馈文档，供使用者参考。

4.3 目标实例对齐

为了获取到目标实例集合，需要分别对输出实例的输出字符串进行匹配对齐与输出字符串中包含的变量的数据依赖进行匹配对齐。

4.3.1 输出实例对齐

输出语句实例是最能直观体现错误发生的语句实例。为了能获取完整的错误信息，需要把输出语句实例进行对齐。输出语句实例对齐有两部分内容，一是输出语句所输出的字符串信息，还有输出语句种包含的变量信息。这些都是通过动态分析获得。比如输出语句 *printf*(”计算结果是 *%d* ”,*a*)，在某一次执行中 $a = 5$，那么该输出实例的输出字符串为 “计算结果是5”，变量值就是 5。

算法 4 输出实例对齐算法

```
1:  procedure PRINTALIGN(L, T)
2:      result = { }
3:      sort(L)
4:      sort(T)
5:      while !L.isEmpty and !T.isEmpty do
6:          result = result ∪ (L.pop, T.pop)
7:      end while
8:      if !L.isEmpty then
9:          while !L.isEmpty do
10:             result = result ∪ (L_i, ∅)
11:         end while
12:     else if !T.isEmpty then
13:         while !T.isEmpty do
14:             result = result ∪ (∅, T_i)
15:         end while
16:     end if
        return result
17: end procedure
```

对于错误版本代码和正确版本代码的所有的输出实例，首先需要将他们合并并对齐。算法4描述了输出实例的合并与对齐过程。算法中，L 表示错误版本代码跟踪的所有输出实例集合，T 表示正确版本代码跟踪的所有输出实例集合，*result* 表示最终对齐的结果。算法第 3-4 行表示对 L 和 T 按照次序进行排序，保证两者的次序是顺序的。算法 5-7 行表示，当 L 和 T 都存在数据时，按照顺序取出两者的第一个数据，配对之后加入到对齐结果 *result* 中，直到两者至少有一个不包含任何数据，循环停止。算法 8-16 行表示，当循环结束后，如果 L 和 T 其中之一仍存在数据，循环将其俺顺序取出与空配对，并存入 *result* 中，直到其中不含有任何数据。

对于 *result* 集合中的每个实例对，根据先后发生的次序依次比较，寻找到目标实例对 P 。首先对输出语句的输出内容进行比较，内容不相同的输出实例即为找到的目标实例。当错误版本输出实例 L_i 与正确版本输出实例 T_i 两个实例的输出字符串 $S(L_i)$ 和 $S(T_i)$ 不同或者 L_i 或 T_i 其中之一为空时，即第一次不同的输出结果，即视为目标实例出现，当前实例对即目标实例，停止寻找。第一个满足公式4.1的即为目标实例对 P 中。

$$S(L_i) \neq S(T_i) \vee L_i = \varnothing \vee T_i = \varnothing \Rightarrow P = P \cup (L_i, T_i) \tag{4.1}$$

4.3.2 变量对齐

对于一些包含有多个变量的输出语句，例如 $printf$("a 和 b 分别是 $\%d, \%d$ ",a, b)，假如错误版本的输出字符串为 “a 和 b 分别是 1,2”，正确版本的输出字符串为 “a 和 b 分别是 1,5”，这种在输出字符串的比较中被视为目标实例。然而对比变量值，错误版本和正确版本的变量 a 的值是相同的，在进行数据依赖切片工作中，a 的切片数据为冗余的，会造成多余的反馈信息。为了减少冗余出现，我们对变量值进行进一步比较。 ensure**Output:**

算法 5 变量对齐算法

```
1:  procedure VARALIGN(U, V)
2:      result = { }
3:      while !U.isEmpty and !V.isEmpty do
4:          result = result ∪ (U.pop, V.pop)
5:      end while
6:      if !U.isEmpty then
7:          while !U.isEmpty do
8:              result = result ∪ (U_i, ∅)
9:          end while
10:     else if !V.isEmpty then
11:         while !V.isEmpty do
12:             result = result ∪ (∅, V_i)
13:         end while
14:     end if
          return result
15: end procedure
```

对于上文求得的目标实例对 P，它的形式为 (l, t)，如果 l 为空，说明错误版本代码缺少输出；如果 t 为空，说明错误版本代码输出多余。这两种情况没有对变量对齐的需求，否则执行变量对齐算法。

算法5描述了输出语句中变量的对齐方法。算法输入 U 和 V 表示求得的目标实例对 (l, t) 分别含有的变量集合。U 表示 l 中的所有变量组成的集合，V 表示 t 中所有变量组成的集合。$result$ 表示最终对齐的结果。算法 3-5 行表示，当 U 和 V 都不为空集时，都从两者中取出第

一个数据，配对后加入到 *result* 集合中，直到两者存在至少一个为空，停止循环。算法 6-14 行表示，当循环结束后，如果 U 和 V 其中之一仍存在数据，循环将其俺顺序取出与空配对，并存入 *result* 中，直到其中不含有任何数据。完成变量对齐之后，根据公式4.2将对齐结果中所有不同的操作数的直接数据依赖存入目标实例 P 中。

$$l_i \neq t_i \vee l_i = \varnothing \vee t_i = \varnothing \Rightarrow P = P \cup (l.ddep, t.ddep) \tag{4.2}$$

4.4 目标实例切片

对于目标实例，我们对齐分别进行动态切片与静态切片。静态切片我们使用 joern[1]静态分析工具，静态切片无需进行5，只需要对4生成的初始目标实例对的源代码进行切片。对于目标实例集合 P 中的每个实例对，我们通过依赖关系，即控制依赖与数据依赖对实例对中的两个语句进行动态切片。

4.4.1 静态切片

为了获取到与目标实例有关的源代码信息，我们使用静态分析方法，构建程序依赖图，并使用目标实例中的变量作为切片准则，并将切片长度设置为 3，以获得相对完整但不冗余的切片结果。在算法4之后获得的目标实例集合 P 仅含有一对实例对 (l, t)，我们使用 joern 工具对错误版本代码和正确版本代码分别构建程序依赖图，并分别对 $l.src$ 和 $t.src$ 进行静态切片，获得两段代码段 $parag(l.src)$ 和 $parag(t.src)$。

4.4.2 动态切片

我们通过控制依赖与数据依赖对实例对目标实例集合 P 中的每个实例对中的两个语句进行动态切片，分别生成错误版本的动态切片和正确版本的错误切片两个版本。

[1]https://joern.io/

算法 6 动态切片算法

```
1:  procedure DYNSLICES(P)
2:      L = {}
3:      T = {}
4:      for P in P do
5:          L = L ∪ P[0]
6:          T = T ∪ P[1]
7:      end for
8:      L = SLICE(L)
9:      T = SLICE(T)
          return L, T
10: end procedure
11: procedure SLICE(sequences)
12:     for sequence in sequences do
13:         slice = slice ∪ ddeps(sequences)
14:         slice = slice ∪ cdeps(sequences)
15:         slice = sort(slice)
16:     end for
          return slice
17: end procedure
```

算法6描述了动态切片的算法。第 4-6 行将目标实例 P 拆分为错误版本实例和正确版本实例两部分并分别存储到 L 和 T 中，L 表示目标实例中的错误版本实例，T 表示目标实例中的正确版本实例。第 8-9 行表示使用切片算法对错误版本实例 L 和正确版本实例 T 切片获得切片结果 **L, T** 并返回。第 11-17 行描述了如何获得一个实例序列的切片。对于实例序列 *sequences* 中的单个实例 *sequence*，获取其数据依赖和控制依赖，并存储到 *slice* 中，即获取到其切片结果。获得结果之后，对切片结果 *slice* 根据语句被执行的顺序进行排序，获得最终的动态切片结果。

4.4.3 切片对齐

获得切片之后，如果只是简单的将切片反馈给使用者，虽然在一定程度上可以辅助使用者修改自己的代码，但是无法判断错误原因。我们考虑到错误出现的三种情况，即：当前实例的计算出错、当前实例不应该被运行和实例缺失。

定义：

τ, ϵ：正确代码语句标签，学生代码语句标签

τ_i, ϵ_j：语句 τ/ϵ 的第 i/j 个实例

A：所有已被对齐的实例对集合

B：对齐但当前实例出错集合

C：缺少实例集合

D：冗余实例结合

规则：

1.$\tau_i, \epsilon_j \in A \cap \tau_k, \epsilon_l \notin A \Rightarrow \tau_k, \epsilon_l \in B$

2.$\nexists \tau_x, \tau_x, \epsilon_y \in A \Rightarrow \epsilon_j \in D$

3.$\nexists \epsilon_x, \tau_x, \epsilon_y \in A \Rightarrow \tau_i \in C$

图 4.1 对齐规则

图4.1展示了切片之后的对齐规则。其中 A 表示第三章中，CDALIGN 得到的最终对齐结果集合。B 存储的实例对应当被对齐但出错。C 存储正确版本存在的实例但是错误版本没有与之对应的实例。C 存储错误版本中存在但是正确版本没有与之对应的实例。规则 1 表示，τ_i 和 ϵ_j 在 CDALIGN 中已经被对齐，但是当前实例 τ_k 和 ϵ_l 没有被对齐，表示 τ_k 和 ϵ_l 是计算出错的实例。规则 2 表示，对于集合 A，不存在任意的正确版本实例可以和当前错误版本实例的任意执行次序对齐，表示实例 ϵ_j 为无意义的冗余实例。规则 3 表示，对于集合 A，不存在任意的错误版本实例可以和当前正确版本实例的任意执行次序对齐，表示实例 τ_k 为错误版本所缺的实例。我们根据图4.1所约束的对齐规则，使用 CDALIGN 所提供的对齐方法，完成目标实例动态切片的对齐。并使用对齐结果进行反馈的生成。

4.5 反馈生成

在反馈生成阶段，根据静态切片和动态切片，我们使用两种不同的方式对他们分别生成反馈。静态切片仅包含源代码信息，用以辅助使用者了解源代码功能，尽快的了解源码内容。动态切片包含有更多语句实例的信息，用以精确定位错误位置与修改建议。两种反馈结合来引导使用者达到了解代码含义到精准修改代码两个阶段。

4.5.1 静态反馈

在由静态切片生成的反馈中，错误版本代码与正确版本代码会分别生成两个代码段。我们对 GPT3.5 设计 prompt，使其扮演“C 语言老师”，让其对两个代码段分别进行代码含义解释，之后再生成两段代码段在实现与语义上的区别。基于 GPT 给出的结果，生成静态反馈内容。

4.5.2 动态反馈

动态切片对齐过程中，参与对齐的包含三种语句类型，分别是赋值语句实例 (Store Instance,STIns)、分支语句实例 (Branch Instance,BRIns) 和输出语句实例 (Print Instance，PRINT)。对于这三种实例，我们采用不同的反馈方式生成修改反馈。

$$\tau_i \in STIns \wedge \epsilon_j \in STIns \vee \tau_i \in STIns \wedge \epsilon_j = \varnothing \vee \tau_i = \varnothing \wedge \epsilon_j \in STIns \tag{4.3}$$

$$\tau_i \in BRIns \wedge \epsilon_j \in BRIns \vee \tau_i \in BRIns \wedge \epsilon_j = \varnothing \vee \tau_i = \varnothing \wedge \epsilon_j \in BRIns \tag{4.4}$$

$$\tau_i \in PRINT \wedge \epsilon_j \in PRINT \vee \tau_i \in PRINT \wedge \epsilon_j = \varnothing \vee \tau_i = \varnothing \wedge \epsilon_j \in PRINT \tag{4.5}$$

公式4.3、4.4和4.5分别展示了反馈三种语句实例类型的条件。动态实例切片分为两部分，即错误版本切片与正确版本切片，所以一对实例对中，至多存在 2 个实例，至少存在 1 个实例。

在赋值语句的反馈生成中，我们根据实例对的变量信息生成反馈内容。在赋值语句实例的反馈中，需要包含有变量值的区别，用数据驱动使用者对赋值语句实例的理解。实例对信息包含错误版本代码文本 F 和正确版本代码文本 T，错误版本代码变量值 f 和正确版本代码变量值 t。反馈时，如果存在两个实例，反馈内容为“When F, variable should be t rather than f like T”。如果只存在错误版本实例，反馈内容为“When F, you should not have done”。如果只存在正确版本实例，反馈内容为”You should have done T”。

在分支语句的反馈生成中，我们根据实例对的错误版本代码文本 F 和正确版本代码文本 T 进行反馈。相同的，实例存在数量也决定了反馈生成的内容。存在两个实例时，反馈内容为“When F, should take another branch like T”。当仅存在错误版本实例时，反馈内容为“You should take branch like T”。当仅存在正确版本实例时，反馈内容为”You should not take branch F”。

在输出语句的反馈生成中，我们也需要考虑输出字符串文本与引用变量值。实例对信息包含错误版本输出字符串 FS 和正确版本输出字符串 TS，错误版本代码变量值 f 和正确版本代码变量值 t。我们使用与赋值语句类似的反馈生成办法。

4.6 本章小结

对于错误代码修改反馈生成的问题，本章提出了一种基于静态切片和动态切片对齐结合的反馈生成技术，并实现了工具 DPEX。本章通过从不符合预期的输出语句出发，我们首先把输出语句实例进行对齐，之后找到关键的输出语句实例，并在错误版本和正确版本中进行目标实例静态切片和动态切片，获得与直接错误相关的信息。静态切片的结果直接通过源代码级别的语义解释向使用者反馈代码语义区别。动态切片生成的结果我们再进行对齐，并根据对齐结果生成反馈内容。

第五章 实验评估及结果分析

为了研究 CDAlign 和 DPEX 在真实的学生代码反馈意见生成场景中与现有方法在效果和性能上的差异、不同相似度阈值和不同扩充深度的影响以及在实际应用中用户的体验差异，本章进行了一系列实验，并对实验结果进行了案例讨论来验证本工作的有效性，最后分析了本工作的不足之处。

5.1 数据构造

5.1.1 数据收集

我们使用爬虫工具从 LEETCODE[1]收集了 10 道使用 C 语言代码实现的算法问题的官方解答，并以行为单位，对其进行随机配对，生成 3000 条随机组合，配对完成的一组中包含有两条代码语句，之后再从中随机挑选出 450 组数据，最后再并人工标注其相似度，相似度分别标注为 0.1、0.3、0.5、0.7、0.9 、1.0，其中，0.1 表示两条代码语句毫无关系，0.3 表示两条代码语句不相似，0.5 表示两条代码语句略微相似，0.7 表示两条代码语句相似，0.9 表示两条代码语句非常相似，1.0 表示两条代码语句完全相同。表5.1标注了代码对相似度标注的数量信息。

表 5.1 代码相似度标注

相似度	0.1	0.3	0.5	0.7	0.9	1.0	总计
数量	192	108	64	62	15	9	450

我们使用 AUTOGRADE[2]提供的爬虫工具，从 CODECHEF[3]收集了 16 道 C 语言实现的算法题的错误提交作为错误版本数据，并使用 GPT3.5[51] 辅助人工对学生版本进行错误修改，获得正确版本的代码，保证可以通过 CODECHEF 的所有用例。考虑到本方法的目标群体是编程学习的初学者，者 16 道题目均是从难度标签为 easy 的题目库里挑选而来。表5.2描述了 16 道算法题目的信息，其中**算法题**表示算法题的名称；**行数**表示每个算法题的正确版本和错误版本的代码行数；**实例数**表示通过动态分析从正确版本代码和错误版本代码中获取的语句实例的数量。表中描述的实例数量仅包含用于后续实验中的实例类型，即 BRIns、STIns 和 PRINT 三种类型。**标记对齐数**表示使用 CDALIGN 对代码进行对齐之后每对代码生成的对齐数量。本工具合计生成了 1104 个标记对齐对。

[1]https://leetcode.cn/

[2]https://github.com/s3team/AutoGrader

[3]https://codechef.com

表 5.2 算法题信息

算法题	行数		实例数		标记对齐数
	正确版本行数	错误版本行数	正确版本实例数	错误版本实例数	
ALEXTASK	33	45	139	161	126
ANKTRAIN	48	26	138	69	36
CHEFAPAR	39	33	162	149	146
CHRL4	48	25	55	19	11
ENTEXAM	75	70	101	110	98
KOL16B	35	27	54	51	41
TRISQ	31	17	268	25	24
SUNDAY	22	22	56	57	52
RGAME	40	34	27	25	22
REMOVECARDS	27	33	489	431	316
MISSP	36	35	62	148	38
MERGEARRAY	37	30	122	116	107
MAXTIME	31	29	34	29	30
BINARYSEARCH	36	35	23	22	21
AVGPERM	45	30	54	74	32
APPLEORANGE	22	23	46	42	17
平均值	37.8	32.1	114.3	95.5	69

表 5.3 对齐结果

题目	对齐数	时间 (s)	*Precision*	*Recall*	F_1Score
ALEXTASK	124	2.76	0.983	0.968	0.976
ANKTRAIN	38	0.748	0.947	1.0	0.972
CHEFAPAR	143	3.22	0.993	0.986	0.989
CHRL4	12	0.07	0.916	1.0	0.956
ENTEXAM	95	2.875	1.0	0.969	0.984
KOL16B	40	0.778	0.950	0.927	0.938
TRISQ	60	0.126	0.95	0.95	0.95
SUNDAY	51	0.794	1.0	0.980	0.989
RGAME	22	0.450	1.0	1.0	1.0
REMOVECARDS	311	7.904	0.993	0.996	0.995
MISSP	100	1.11	0.98	0.97	0.975
MERGEARRAY	105	2.473	0.980	0.963	0.971
MAXTIME	28	0.549	1.0	0.933	0.965
BINARYSEARCH	21	0.599	1.0	1.0	1.0
AVGPERM	32	0.647	0.968	0.968	0.968
APPLEORANGE	17	0.258	1.0	1.0	1.0

5.2 实验设置

本章实验部分所使用的主要编程语言是 Python3.7，使用的主要工具包括 LLVM[29]、Z3[48]、joern 和 GPT3.5[51]。在数据构造过程中使用处理器为 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz，内存为 8G，操作系统为 Windows11 的 Desktop 完成。在序列对齐与反馈生成中使用的 VMware 下的 Ubuntu16.04 虚拟机完成。代码跟踪的获取及符号表达对比沿用了 APEX[26] 的方法，在代码跟踪中记录了每条语句实例的标签，包含本代码语句的所处行数、本代码语句的当前执行次数、本代码语句的语句类型、符号表达式、变量值、控制依赖和数据依赖等。符号表达式使用 Z3 求解器来求解，对齐和反馈均使用 Python3.7 实现。

为了验证本文方法的可行性，我们提出了以下研究问题：

RQ1: CDALIGN 与现有对齐方法相比，对齐效果如何？

RQ2: 获取输入语句的依赖的效率如何？

RQ3: 不同相似度阈值对 CDALIGN 的对齐结果有何影响？

RQ4: 对 sentence-bert 模型的微调对 CDALIGN 的对齐结果有何影响？

RQ5: CDALIGN 中，独立于对齐流程中的扩充对齐在对齐过程中起到什么样的作用？

RQ6: DPEX 对学生代码的反馈生成率上表现如何？

RQ7: 关于静态切片的反馈质量如何?

RQ8: DPEX 对学生的帮助如何?

为了考察本文提出的方法与现有方法在不同方面的各类表现，需要与现有的代码序列对齐方法和反馈生成方法进行对比。由于 APEX[26] 项目已经不再进行维护，我们复现了一个最直观并与我们工作类似的错误分析工具和 BPEX[]，并复用 BPEX 论文中有关 APEX 的相关数据。

对于 **RQ1**,我们将 CDALIGN 对表5.2中已经标注过的数据集对齐,具有对齐时间、精确度、召回率和 F_1Score 四个指标对结果进行评估；并使用 BPEX 所提供的数据再次进行 CDALIGN 实验以比较 CDALIGN 和 APEX。

对于 **RQ2**，我们对表5.2中的所有代码的输入语句进行直接控制依赖提取，并对控制依赖获取其数据依赖。

对于 **RQ3**，我们通过调整了 sentence-bert 所生成的语句相似度阈值来获得不同阈值所生成的对齐结果。我们尝试了使用 0.5、0.6、0.7 三种阈值来生成对齐结果，对这三种结果分别比较精确度、召回率和 F_1Score 三个指标。

对于 **RQ4** 和 **RQ5**，我们使用了5.1中的数据对 sentence-bert 进行了微调。并使用未微调但扩充对齐、微调但没有扩充对齐、微调并扩充对齐这三种对齐方式，对这三种结果分别比较对齐数、精确度、召回率和 F_1Score 四个指标。

对于 **RQ6**，我们通过取消扩充对齐步骤，对表5.2中的数据进行对齐，从对齐时间、精确度、召回率和 F_1Score 四个指标评估扩充对齐的作用。

对于 **RQ6**，我们通过对5.2中的算法题进行了反馈生成，观察他们的反馈生成率。

对于 **RQ7** 和 **RQ8**，我们随机挑选了 16 条入门级别的 C 语言编程题目，分发给 16 名 C 语言初学者，让他们完成其中一道。除去完全正确的题目，我们共收获到 10 条错误代码。在我们人工纠正代码后，用错误代码和正确代码进行错误反馈，并记录反馈结果是否对学生有帮助。

5.3 实验结果分析

RQ1:CDALIGN 与现有对齐方法相比，对齐效果如何?

为了回答 **RQ1**,我们把 CDALIGN 在时间、精确度 (Precision)、召回率 (Recall) 和 F_1Score($F_1 = 2 * \frac{precision*recall}{precision+recall}$) 计算其性能，对齐结果与评价指标见表5.3。因为 APEX 已经停止维护，无法获得其工具代码，为了与其进行比较，我们使用了 BPEX 中关于 APEX 的数据作为参照。由于机器性能的不同，时间指标不具有参考意义，所以将时间数据从表格中剔除，具体对比信息见表5.4。需要注意的是，BPEX 中每个题目都包含有 2-3 个样本数，对齐结果均为平均值。我们使用 BPEX 所提供的所有数据完成了表5.4，其中指标也均为平均值。

表 5.4 结果比较

题目	对齐数		*Precision*		*Recall*		F_1*Score*	
	APEX	CDALIGN	APEX	CDALIGN	APEX	CDALIGN	APEX	CDALIGN
ALEXTASK	110	124	0.91	0.983	0.96	0.968	0.93	0.976
ANKTRAIN	66	76	0.93	0.947	1.0	1.0	0.96	0.972
CHEFAPAR	120	143	0.87	0.993	0.93	0.986	0.9	0.989
CHRL4	28	24	0.85	0.916	0.95	1.0	0.89	0.956
DISHOWN	45	34	0.61	0.83	0.64	0.84	0.62	0.83
ENTEXAM	85	95	0.91	1.0	0.94	0.969	0.92	0.984
KOL16B	41	40	0.98	0.950	1	0.927	0.99	0.938
、NOTINCOM	54	65	0.86	0.99	0.91	0.99	0.88	0.99
RGAME	18	22	0.93	1.0	0.93	1.0	0.93	1.0
TRISQ	61	60	0.84	0.95	0.92	0.95	0.87	0.95
平均值	63	67	0.87	0.96	0.92	0.96	0.89	0.96

在**对齐数**方面，CDALIGN 与 APEX 的对齐数量有多有少，因为两种方法使用的对齐方式不同，APEX 使用 MAX-SAT 来建模解决对齐问题，该方法目的为尽可能对齐更多的代码实例。这种方法在一定程度上增加了实例对的数量。而 CDALIGN 的对齐是基于深度学习模型对单条代码语句相似度判断的基础上，满足源代码相似度阈值的实例对才被对齐，并且在后续加入了 APEX 没有的扩充对齐机制，保证了未被注意到的本应被对齐的实例对。所以两种方法在不同的题目下的对齐数量没有确定的数量关系。在 **Precision** 方面，APEX 的平均值为 87%，而 CDALIGN 的平均值达到了 96% ，高于 APEX。CDALIGN 不仅平均值要高于 APEX，关注单个题目，CDALIGN 的所有单个题目的 **Precision** 都要高于 APEX，其方差也较 APEX 更好。在 **Recall** 召回率方面，CDALIGN 的 96% 略高于 APEX 的 92%。APEX 虽然使用 MAX-SAT 尽可能获得更多的对齐结果，但是得益于扩充对齐步骤与深度学习模型预测，CDALIGN 在召回率上相较 APEX 仍有略微提升。总体上来说，CDALIGN 在精确度与召回率上都要高于 APEX，其平均得分 F_1**Score** 要比 APEX 高 7.8%。

RQ2: 获取输入语句的依赖的效率如何?

为了回答 **RQ2**，我们对表5.2中所有的算法获取了其输入语句的跟踪实例，并根据前后关系对其进行直接对齐，即错误版本的第 i 个输入语句无条件与正确版本的第 i 个输入语句进行对齐。对齐后根据其先后关系分别获得其控制依赖和控制依赖的数据依赖的对齐序列，表5.5描述了每个算法题目对齐的输入语句跟踪数量、控制依赖数量和控制依赖数的数据依赖数量。并不是所有的输入语句都存在控制依赖，也不是所有的控制依赖都存在数据依赖。根据题目的代码规模，最少的输入语句数量为 4 个，最多的输入语句有 49 个；控制依赖的数量

从 4 个到 23 个不等；数据依赖的数量也包含有 6 个到 40 个。它们的平均值分别为 18.3 个，11.9 个和 13.9 个。

表 5.5 依赖数量

算法题	输入语句跟踪数量	控制依赖数量	数据依赖数量
ALEXTASK	17	7	12
ANKTRAIN	12	23	24
CHEFAPAR	37	21	10
CHRL4	13	5	8
ENTEXAM	13	14	26
KOL16B	6	15	12
TRISQ	12	13	22
SUNDAY	22	10	8
RGAME	8	4	6
REMOVECARDS	49	19	8
MISSP	17	7	12
MERGEARRAY	32	21	40
MAXTIME	29	5	8
BINARYSEARCH	20	7	12
AVGPERM	4	16	8
APPLEORANGE	4	4	6
平均值	18.3	11.9	13.9

RQ3: 不同相似度阈值对 CDALIGN 的对齐结果有何影响?

在对齐过程中，设定一个合适的阈值来判断源代码是否相似是很重要的。太低的相似度阈值会导致过多的错误对齐，而过高的相似度阈值又会错失很多本应对齐的实例对。为了回答 **RQ3**，如图5.1所示，我们使用三种相似度阈值分别为 0.5、0.6 和 0.7 来分析不同相似度阈值对 CDALIGN 的性能的影响。在评估中，0.6 是 CDALIGN 所使用的相似度阈值。

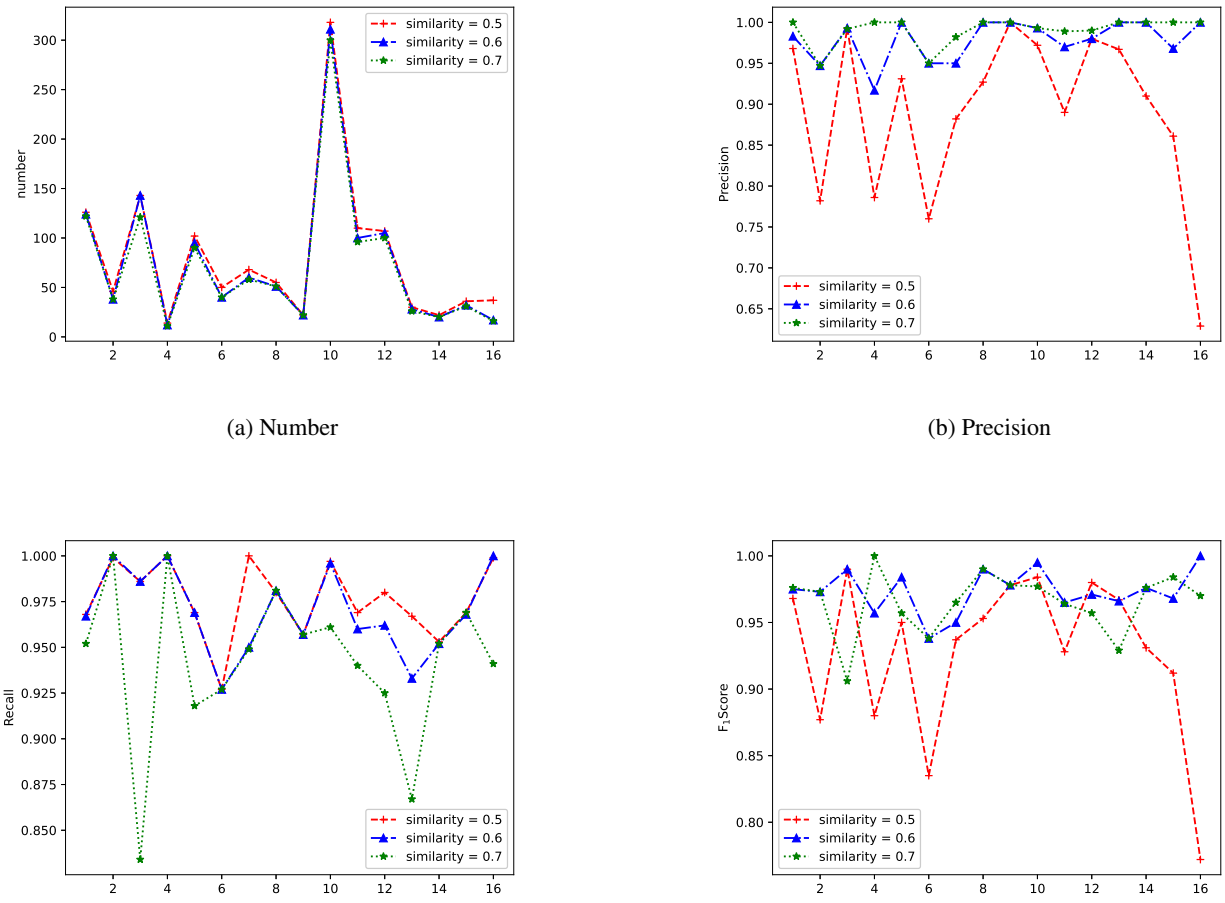
对齐数量是在不同相似度阈值下生成序列的对齐数量。由于对齐数量过多，在对齐数量上面，三种阈值所得到的对其数量差距并不能直观的感受到，但也随着相似度阈值的增加，每个算法题所得到的对齐数量呈现下降趋势。由于每个算法题目在实现上的不同与代码风格的差异，相同的阈值在不同的题目中所带来的影响是不同的，在某些算法题上显现明显，而有些问题上差距并不大。如图所示，有些题目在三种相似度阈值下生成的对齐数量是相等的，而有些题目的数量差距可以到达到 20%。

在精确度方面，当相似度阈值设置为 0.7 时，对齐结果的精确度一直处于一个相对高的位置，即使是精确度最低的算法任务中，其精确度仍旧处在 95% 左右。在高度形式化的单行

代码语言中，0.7 的相似度意味着两行代码已经高度相似，不同语义代码相似度达到 0.7 是很罕见的。当相似度阈值设置为 0.5 时，不同的算法任务其精确度方差相较于 0.6 有巨大变化。因为在不同代码里，两个版本中略微相似的代码数量不同，0.5 的相似度阈值处于模棱两可的位置，两条代码语句有相似的可能就会被认为两条代码相似，所以在不同任务中有较大的区别。总体而言，精确度与相似度阈值之间呈现正相关关系。

在召回率方面，相似度阈值设置为 0.7 时，在所有的任务中均低于阈值 0.5 和 0.6。过高的相似度阈值约束了对齐结果里每对的两个代码行需要有高度的相似性。即使已经弱化了变量名在相似度计算中的权重，但无法消除其影响，所以存在语义相同代码语句无法达到相似度阈值，导致了较低的召回率，但在错误版本代码与正确版本代码风格类似的任务中并没有导致低的召回率。总体而言，召回率与相似度阈值之间呈现负相关关系。

在综合了精确度与召回率的 F_1Score 方面，当相似度阈值设置为 0.5 时，因为过低的精确度。不论是均分还是方差都要低于相似度阈值设置为 0.6 和 0.7。在某些问题上相似度阈值设置为 0.7 具有更好的表现，但平均来看相似度阈值设置为 0.6 是更好的选择。



(a) Number

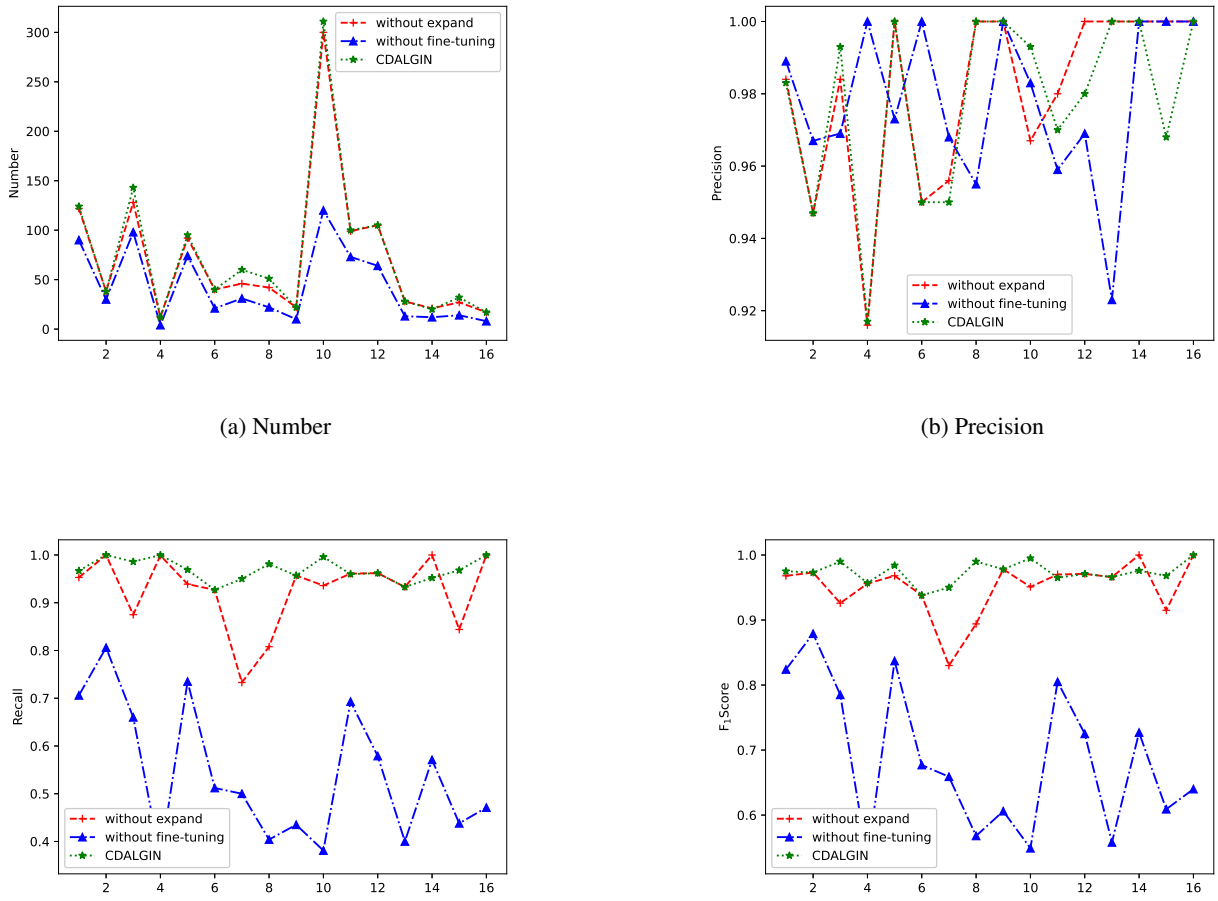
(b) Precision

(c) Recall

(d) F1

图 5.1 不同相似度阈值结果

RQ4: 对 sentence-bert 模型的微调对 CDALIGN 的对齐结果有何影响？**RQ5:** CDALIGN 中，独立于对齐流程中的扩充对齐在对齐过程中起到什么样的作用？



(a) Number

(b) Precision

(c) Recall

(d) F1

图 5.2 微调与扩充对齐

对于 **RQ4** 和 **RQ5**，如5.2所示，我们在同一个实验中将其与 CDALIGN 进行对比，without expand 是未微调但扩充对齐、without fine-tuning 是微调但没有扩充对齐、CDALIGN 是微调并扩充对齐。微调是为了使得模型更加适应本工作的“计算单行代码的相似度”，而扩充对齐是为了防止应该被对齐的实例没有参与对齐而做的扩充工作。

在对齐数量上来看，未经微调的结果明显小于其余两种。差距最大的算法任务种，未经微调的数量要比其他两种低超过 50%；但在某些算法任务中数量相差无几。由于不同算法任务种，两个代码的风格各不相同。当代码风格相似时，其相似度更容易通过自然语言的语义判断方法进行判断，而风格差异较大时，往往导致差异较大的相似度。而扩充对齐的目的是增加未被计算的实例对，所以在数量上，要略大于未进行扩充对齐的变体。

在精确度方面，对于不同的任务，三种方法各有高低。总的来说，未经过扩充对齐的方法要比另外两种方法拥有更高的精确度。因为经过微调之后，模型在单条代码相似度计算任务

上有更好的表现，而扩充对齐虽然解决了被忽略的实例，然而也有可能导致错误的对齐。糟糕的方差也体现了未经微调的方法变体由于代码风格的差异表现出的不同性能。

未经微调的方法变体在召回率方面要远落后于其他两种方法。因为未经微调的模型无法精准判断出单行代码语句的相似度，其相似度往往偏低，所以导致其较低的召回率。增加了扩充对齐在 CDALIGN 多数任务中都要比没有扩充对齐的变体要有更高的召回率，这也达到了扩充对齐的目的。

在综合了精确度与召回率的 F_1Score 方面，由于三种方法的精确度相差并不大，所以召回率在计算 F_1Score 时扮演十分重要的角色，F_1Score 与召回率的走向也具有相同趋势。由于未经微调的变体的召回率要明显低于其他两种，所以其 F_1Score 也明显低于其余两种。同样，由于 CDALIGN 的召回率要略高于未扩充的变体，所以 CDALIGN 的 F_1Score 也在三种方法中整体更高。

RQ6: DPEX 对学生代码的反馈生成率上表现如何?

反馈生成率 (反馈生成率 $=\frac{\text{生成的正确反馈数量}}{\text{错误的数量}}$) 用于评价工具对错误反馈的效率。图5.3展示了图5.2中所有算法题信息的反馈生成率。在 16 个任务中，反馈生成率最高的任务，其反馈生成率为 97.6%，而反馈率生成最低的任务反馈率仅为 78.9%，虽然可以发现多数的错误位置，但并不是优秀的结果。经过对每个任务的检查发现，较低反馈率的原因有两个。第一，极少数的反馈丢失是因为对应位置的跟踪信息没有被对齐，这种原因仅在 KOL16B 与 TRISQ 两个任务中共出现 3 次。第二，并不优秀的测试用例导致了无法覆盖到所有的错误位置，由于错误位置没有被执行到，所以无法找到错误，大多数丢失的反馈都是由于这个原因。

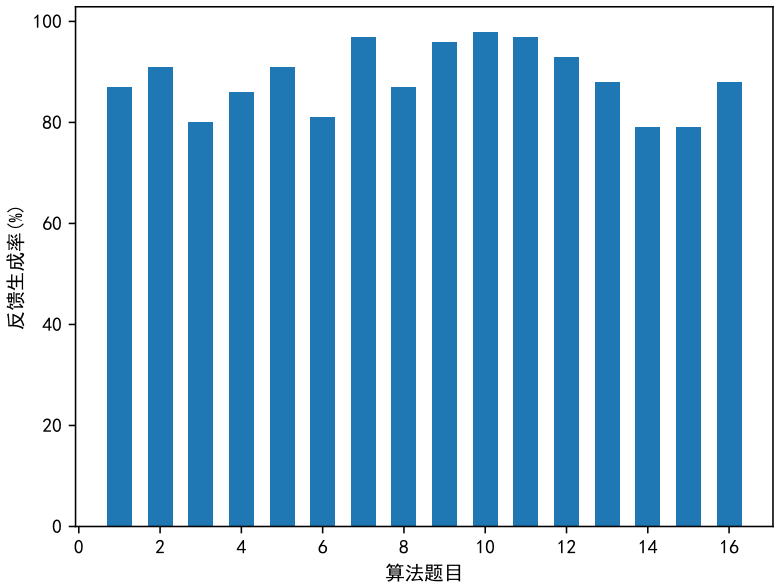


图 5.3 反馈生成率

RQ7: 关于静态切片的反馈质量如何？**RQ8:** DPEX 对学生的帮助如何？

首先，我们随机挑选了 16 条入门级别的 C 语言编程题目，并使用线上与线下两种方式分发给 16 名 C 语言初学者，即每人完成其中一道题目。我们收到来自线上的答案 8 条与来自线下的答案 8 条。除去完全正确的解答，我们共收获道 10 条存在错误的代码，其中有 7 条来自线下，3 条来自线上。在我们人工纠正错误之后，我们使用两个版本代码分别提交给 BPEX 和 DPEX 进行反馈，并使用五点李克特量表让初学者们对生成的两种反馈结果进行打分，有非常同意、同意、不一定、不同意、非常不同意五种回答，分别记为 5、4、3、2、1 分。

题目序号	**BPEX**	**DPEX**
1	5	5
2	3	4
3	4	3
4	4	5
5	3	3
6	5	4
7	4	4
8	3	4
9	4	4
10	3	3
平均值	3.8	3.9

表 5.6 反馈评价表

表5.6展示了学生们对 10 道题目由 BPEX 和 DPEX 的反馈生成结果的评价。由评价结果可以看到，两种反馈方法的质量基本相同，仅存在略微差别，这是因为两种方法的反馈生成方法是类似的。但是在新的对齐方法下，DPEX 仍以 0.1 的均分超过了 BPEX。除了两者共有的反馈结果外，DPEX 还存在有以源代码为内容的静态切片的代码理解反馈。我们再次使用五点李克特量表让初学者们对生成的源码理解准确度进行评价，即是否能正确的解释源码的语义与关键区别。表5.7展示了对两个版本的代码切片的语义理解和差异分析的得分评价表。可以看到得益于大模型优秀的性能，对代码的理解和差异分析在所有的题目上都有很好的结果。

题目序号	1	2	3	4	5	6	7	8	9	10	平均值
得分	5	5	4	5	5	5	5	5	5	4	4.8

表 5.7 源码语义评价表

5.4 案例讨论

我们使用图3.1中展示的任务来作为例子进行分析，来更好的理解我们的方法在对齐和反馈的过程。

5.4.1 对齐问题

在第三章中，我们已经分析过了 APEX 和 BPEX 存在的对齐缺陷，并使用源码级的语义相似度来解决过分依赖符号表达所导致的低质量对齐。由于对齐结果的数量太大，本文以图3.1展示的两个版本代码的核心代码图5.4的一部分对齐结果来进行对齐的分析。

```
1  int max=0;
2  for(int j=0;j<n;j++) {
3      for(int k=j+1;k<n;k++) {
4          if(gcd(a[j],a[k])>max) {
5              c=a[j];
6              b=a[k];
7              max=gcd(a[j],a[j+1]);
8          }
9      }
10 }
11 int result=(b*c)/max;
12 printf("%d\n",result);
```

```
1  int min = INT_MAX;
2  for(i=0; i<n-1; i++) {
3      for(j=i+1; j<n; j++) {
4          if((a[i]*a[j])/gcd(a[i], a[j]) < min) {
5              min = (a[i]*a[j])/gcd(a[i], a[j]);
6          }
7      }
8  }
9  printf("%d\n", min);
```

图 5.4 核心代码

图5.4左侧为错误版本代码，右侧为正确版本代码。代码中函数 gcd() 用以求解两个数字的最大公约数，并且两个版本的 gcd() 函数都是正确的。

图5.5展示了图5.4的第一次迭代的对齐结果。灰色格子表示两个序列被最终对齐。在对齐过程中，int max = 0 与 int min = INT_MAX 虽然两者的相似度已经超过了阈值 0.6，但是由于其赋给变量的值不同，即使两者的功能是相同的，但是无法忽视变量的区别将其对齐。BPEX 的错误对齐是将 int max = 0 与 for(i=0; i<n-1;i++) 对齐，这是因为两者赋给变量的值是相同的"0"。在我们的工作中，我们计算这两条代码的相似度仅为 0.232，其相似度远远低于设置的阈值 0.6，所以拒绝把这两条语句对齐。因此 2_1^{τ} 没有被错误对齐，最终正确的与自己司职相同且相似度高达 0.983 的 2_1^{ϵ} 对齐。对于 4_1^{τ} 与 4_1^{ϵ} 类型的分支语句，其使程序能够根据不同的条件做出不同的响应，虽然其内部变量的值不同，但是其具有高度的语义相似性，也将其正确对齐。

错误版本 ϵ	src	正确版本 τ	src
...	...	...	...
1_1	int max = 0		
		1_1	int min = INT_MAX
2_1	for(int j=0;j<n;j++)	2_1	for(i=0; i<n-1; i++)
3_1	for(int k=j+1;k<n;k++)	3_1	for(j=i+1; j<n; j++)
4_1	if(gcd(a[j],a[k])>max)	4_1	if((a[i]*a[j])/gcd(a[i], a[j]) < min
5_1	c=a[j]		
6_1	b=a[k]		
7_1	max=gcd(a[j],a[j+1])	5_1	min = (a[i]*a[j])/gcd(a[i], a[j])
...	...	...	...
11_1	int result=(b*c)/max		
...	...	...	...
12_1	printf("%d",result)	9_1	printf("%d", min)

图 5.5 案例对齐

5.4.2 反馈生成问题

动态反馈生成需要根据目标实例的切片对齐结果生成。在动态反馈生成过程中，与 BPEX 采用了类似的动态反馈生成方法。以图5.5为例，当前任务的目标实例对为 $(12_1^{\epsilon}, 9_1^{\tau})$。其中 12_1^{ϵ} 的直接数据依赖是 11_1^{ϵ}，而 9_1^{τ} 的直接数据依赖是 9_n^{τ}。需要注意的是，两个版本代码的实现方式不同，错误版本代码在迭代结束求得 max 值之后才计算最终结果，而正确版本代码在迭代过程中动态的计算最终结果。因此错误版本代码的 12_1^{ϵ} 的直接数据依赖 11_1^{ϵ} 并没有实例与其对齐。所以找到 11_1^{ϵ} 的数据依赖 7_n^{ϵ}，由于错误版本分支语句 4^{ϵ} 与正确版本分支语句 4^{τ} 条件不同，7^{ϵ} 与 5^{τ} 并非每个运行实例都被对齐。因此三个语句对 $(4^{\epsilon}, 4^{\tau})$，$(7^{\epsilon}, 5^{\tau})$ 和 $(11^{\epsilon}, \varnothing)$ 即我们方法找到的“对齐但不匹配”与“冗余”结果。基于找到的这三种结果，我们生成的反馈如图5.6所示。

在静态反馈方面，我们分别图3.1所示对两段代码的目标实例进行切片，错误版本代码的目标实例为第 27 行，正确版本代码的目标实例为第 22 行。切片结果如图5.7所示。其中的数字代表获得的切片行号。即错误版本的切片结果是第 16、19、22 和 26 行，正确版本的切片结果是 17、20 和 21 行。将错误版本代码的第 17-27 行与正确版本代码的第 17-22 行提交给 GPT3.5，得到的反馈如图5.8所示

When ”if(gcd(a[j],a[k])>max)”, you should take another branch like ”if((a[i]*a[j])/gcd(a[i], a[j]) < min)”
When ”max=gcd(a[j],a[j+1])”, variable should be 3 rather than 12 like ”min = (a[i]*a[j])/gcd(a[i], a[j])”
You should not have done ”int result=(b*c)/max”
When ”printf(” %d” ,result)”, variable should be 3 rather than 6 like ”printf(” %d” , min)”

图 5.6 反馈结果

16 19 22 26	17 20 21

图 5.7 静态切片结果

5.4.3 效度威胁

本节对本实验的效度威胁进行分析，包括内部效度威胁与外部效度威胁。内部效度指的是研究结果是否可以正确地归因于研究中的因果关系，外部效度指的是研究结果是否可以推广到其他场景或总体的程度。

内部效度威胁：在微调模型阶段，由于所需微调数据的独特性，数据均来自随机收集并标注。所收集的数据集与人工标注的主观性无法保证微调后的模型对于所有的 C 语言代码相似度计算的精确性。

由于算法任务产生的实例众多，无法完成对所有数据集的标注，所以无法评估所有任务的对齐结果。因此我们挑选了有代表性的 16 道题目进行实验，无法保证是否在其他题目也有类似的性能。

在对齐的人工标注阶段，工具的精确度与召回率均由人工标注计算而来。尽管人工标注是由实验室其他与本工作无关的研究员标注，依然无法保证标注数据的完全准确。

外部效度威胁：DPEX 仅能处理 C 语言代码的错误反馈。尽管 C 语言依然是编程初学者的第一选择，但是也有不少编程学习者选择学习 Java 或者 Python 等语言，我们还无法处理其他语言的任务。

DPEX 仅能对能够编译运行的完整代码进行错误反馈。现如今不少的代码学习网站已经使用了核心代码的编程训练，即使用者只需要编写完成任务的核心代码。DPEX 无法处理这种类型的问题。

DPEX 能够生成反馈的前提是所使用的测试用例触发了错误。如果测试用例无法触发错误，DPEX 也无法生成错误反馈。

第一段代码是在一个双重循环中遍历数组中的所有元素对，并计算它们的最小公倍数除以最大公约数，然后更新最小值。这样做的好处是，它能够遍历所有可能的元素对，确保找到最小的结果。
而第二段代码则是在双重循环中，每次只更新一个最大公约数，并更新相应的乘积值，最终得到的乘积并不一定是最小公倍数除以最大公约数，因为它只在找到一个更大的最大公约数时更新，可能会错过更小的乘积值。

图 5.8 静态反馈结果

DPEX 能够生成反馈的前提是代码可以编译运行，如果错误版本代码错误的原因是代码报错而不是逻辑错误，DPEX 也无法生成错误反馈。

5.4.4 本章小结

为了验证 DPALIGN 和 DPEX 的有效性，本章进行了若干实验。我们对从 CODECHEF 和 LEETCODE 上选取的 16 道具有代表性的代码题目的错误版本和正确版本作为数据集，使用两种版本代码进行反馈生成。实验显示我们的方法在对齐和反馈生成上与现有方法均有一定提升。为了探究方法中相似度阈值、微调模型和扩充对齐对结果的影响，我们又对工具的多种变体进行消融实验。实验结果表明，使用微调后的模型，设置 0.6 的相似度阈值并进行扩充对齐能取得最好的结果。最后对实验的效度威胁进行了分析。

第六章 总结与展望

本章将对本工作进行全面的总结，并讨论当前工作的不足之处，对未来进行展望。

6.1 工作总结

随着互联网时代的发展，计算机专业招生人数不断扩大，自学编程技术的人员不断增加，教学者的压力激增。针对错误代码生成反馈对缓解教学者压力、辅助使用者修复错误具有十分重要的意义。本文就无法解决错误的逻辑问题、现有工作的低质量对齐与反馈问题，提出了基于深度学习的编程作业反馈生成技术。本文的具体工作如下：

1. 随机生成了 3000 对单行代码的随机组合，并随机挑选 450 条数据进行人工标注其相似度。并使用此数据集对 SBERT 进行微调使其适应我们的任务。

2. 从 CODECHEF 和 LEETCODE 上人工收集了具有代表性的 16 道算法题目，其中包含 16 对错误版本代码和正确版本代码组成的数据集，并对他们的程序跟踪进行人工标注，共标注了 1056 条实例对。

3. 提出了一种基于深度学习的代码对齐方法，并根据方法实现了工具 CDALIGN。首先通过微调 “all-MiniLM-L6-v2” 模型来使其适应本工作，接着使用动态程序分析和符号分析方法获取正确版本代码和错误版本代码的跟踪信息并进行初始对齐，然后使用输入语句作为基准，辅助以深度学习模型推测结果，对初始对齐结果进行纠正与扩充。最后使用广度遍历算法扩充序列对，完成扩充生成最终对齐结果。

4. 提出了一种基于动态切片对齐和静态切片的反馈生成方法，并实现了工具 DPEX。从不符合预期的输出语句出发，首先把输出语句实例进行对齐，之后通过关键的输出语句实例，并在错误版本和正确版本中进行目标实例静态切片和动态切片，获得与直接错误相关的信息。静态切片的结果直接通过源代码级别的语义解释向使用者反馈代码语义区别。动态切片与 DPALIGN 的对齐结果再次对齐，根据对齐结果生成动态反馈。

6.2 未来展望

本文基于程序分析技术和深度学习技术，以 C 语言代码题目为研究场景，研究代码跟踪的序列对齐和反馈生成技术。本文第 5 章针对研究目的进行了充分的实验和分析，但本工作仍有需要完善的地方。

语言扩展：本文仅能在 C 语言场景下完成研究目标，为了保证方法的有效性和可扩展性，未来需要使本工作可以应用在更多开发语言的场景下，验证本工作的结论。

方法改进：本文方法基于代码跟踪完成对齐与反馈，无法处理语法或编译等原因导致的错误，并且对错误版本代码的反馈高度依赖于高质量的正确版本的代码。未来考虑加强静态分析在工作中的权重，减弱对正确版本代码的依赖性。

在反馈生成方面，动态反馈的内容都是基于正确版本代码提出修改意见，可能会导致使用者出现理解偏差。未来考虑生成图形化的反馈文档，设计更容易理解的反馈信息。

此外，本工作只有在测试用例能够触发错误的情况下才能生效。未来考虑继承测试工具，例如 AFLGO，保证发现错误的完整性[52]。

方法扩展：本文工作的对象是规模小的算法题目，验证了深度学习方法在场景中的可行性。未来考虑将对齐与反馈方法应用在规模更大的工程代码的错误定位与错误反馈上。

参考文献

[1] Soper T. Analysis: The exploding demand for computer science education, and why america needs to keep up [J]. GeekWire. com, June, 2014, 6: 2014.

[2] Hollingsworth J. Automatic graders for programming classes [J]. Communications of the ACM, 1960, 3(10): 528-529.

[3] Edwards S H, Perez-Quinones M A. Web-cat: automatically grading programming assignments [C]// Proceedings of the 13th annual conference on Innovation and technology in computer science education. 2008: 328-328.

[4] Higgins C, Hegazy T, Symeonidis P, et al. The coursemarker cba system: Improvements over ceilidh [J]. Education and Information Technologies, 2003, 8: 287-304.

[5] Wang T, Su X, Ma P, et al. Ability-training-oriented automated assessment in introductory programming course [J]. Computers & Education, 2011, 56(1): 220-226.

[6] Queirós R A P, Leal J P. Petcha: a programming exercises teaching assistant [C]//Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education. 2012: 192-197.

[7] Beizer B. Black-box testing: techniques for functional testing of software and systems [M]. John Wiley & Sons, Inc., 1995.

[8] Tillmann N, De Halleux J, Xie T, et al. Teaching and learning programming and software engineering via interactive gaming [C]//2013 35th International Conference on Software Engineering (ICSE). IEEE, 2013: 1117-1126.

[9] Adam A, Laurent J P. Laura, a system to debug student programs [J]. artificial intelligence, 1980, 15(1-2): 75-122.

[10] Tillmann N, De Halleux J. Pex–white box test generation for. net [C]//International conference on tests and proofs. Springer, 2008: 134-153.

[11] Singh R, Gulwani S, Solar-Lezama A. Automated feedback generation for introductory programming assignments [C]//Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation. 2013: 15-26.

[12] Bhatia S, Kohli P, Singh R. Neuro-symbolic program corrector for introductory programming assignments [C]//Proceedings of the 40th International Conference on Software Engineering. 2018: 60-70.

[13] Wang K, Singh R, Su Z. Search, align, and repair: data-driven feedback generation for introductory programming exercises [C]//Proceedings of the 39th ACM SIGPLAN conference on programming language design and implementation. 2018: 481-495.

[14] Gulwani S, Radiček I, Zuleger F. Automated clustering and program repair for introductory programming assignments [J]. ACM SIGPLAN Notices, 2018, 53(4): 465-480.

[15] Pu Y, Narasimhan K, Solar-Lezama A, et al. sk_p: a neural program corrector for moocs [C]//Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity. 2016: 39-40.

[16] Yi J, Ahmed U Z, Karkare A, et al. A feasibility study of using automated program repair for introductory programming assignments [C]//Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. 2017: 740-751.

[17] Drummond A, Lu Y, Chaudhuri S, et al. Learning to grade student programs in a massive open online course [C]//2014 IEEE International Conference on Data Mining. IEEE, 2014: 785-790.

[18] Gulwani S, Radiček I, Zuleger F. Feedback generation for performance problems in introductory programming assignments [C]//Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering. 2014: 41-51.

[19] Liu X, Wang S, Wang P, et al. Automatic grading of programming assignments: an approach based on formal semantics [C]//2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET). IEEE, 2019: 126-137.

[20] Ahmed U Z, Srivastava N, Sindhgatta R, et al. Characterizing the pedagogical benefits of adaptive feedback for compilation errors by novice programmers [C]//Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering Education and Training. 2020: 139-150.

[21] Messer M, Brown N C, Kölling M, et al. Automated grading and feedback tools for programming education: A systematic review [J]. ACM Transactions on Computing Education, 2024, 24(1): 1-43.

[22] Kiesler N, Lohr D, Keuning H. Exploring the potential of large language models to generate formative programming feedback [C]//2023 IEEE Frontiers in Education Conference (FIE). IEEE, 2023: 1-5.

[23] Phung T, Cambronero J, Gulwani S, et al. Generating high-precision feedback for programming syntax errors using large language models [J]. arXiv preprint arXiv:2302.04662, 2023.
[24] Phung T, Pădurean V A, Singh A, et al. Automating human tutor-style programming feedback: Leveraging gpt-4 tutor model for hint generation and gpt-3.5 student model for hint validation [C]//Proceedings of the 14th Learning Analytics and Knowledge Conference. 2024: 12-23.
[25] Zhang Q, Zhang T, Zhai J, et al. A critical review of large language model on software engineering: An example from chatgpt and automated program repair [J]. arXiv preprint arXiv:2310.08879, 2023.
[26] Kim D, Kwon Y, Liu P, et al. Apex: automatic programming assignment error explanation [J]. ACM SIGPLAN Notices, 2016, 51(10): 311-327.
[27] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need [J]. Advances in neural information processing systems, 2017, 30.
[28] Weiser M. Program slicing [J]. IEEE Transactions on software engineering, 1984(4): 352-357.
[29] Lattner C, Adve V. Llvm: A compilation framework for lifelong program analysis & transformation [C]// International symposium on code generation and optimization, 2004. CGO 2004. IEEE, 2004: 75-86.
[30] Lee J, Song D, So S, et al. Automatic diagnosis and correction of logical errors for functional programming assignments [J]. Proceedings of the ACM on Programming Languages, 2018, 2(OOPSLA): 1-30.
[31] Song D, Lee W, Oh H. Context-aware and data-driven feedback generation for programming assignments [C]// Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2021: 328-340.
[32] Yang D, Mao X, Chen L, et al. Transplantfix: Graph differencing-based code transplantation for automated program repair [C]//Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. 2022: 1-13.
[33] Kim J, Kim S. Automatic patch generation with context-based change application [J]. Empirical Software Engineering, 2019, 24: 4071-4106.
[34] Nguyen T, Weimer W, Kapur D, et al. Connecting program synthesis and reachability: Automatic program repair using test-input generation [C]//Tools and Algorithms for the Construction and Analysis of Systems: 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I 23. Springer, 2017: 301-318.
[35] King J C. Symbolic execution and program testing [J]. Communications of the ACM, 1976, 19(7): 385-394.
[36] Xin Q, Reiss S P. Leveraging syntax-related code for automated program repair [C]//2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2017: 660-670.
[37] Ji T, Chen L, Mao X, et al. Automated program repair by using similar code containing fix ingredients [C]// 2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC): volume 1. IEEE, 2016: 197-202.
[38] Gao F, Wang L, Li X. Bovinspector: automatic inspection and repair of buffer overflow vulnerabilities [C]// Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. 2016: 786-791.
[39] Clune J, Ramamurthy V, Martins R, et al. Program equivalence for assisted grading of functional programs [J]. Proceedings of the ACM on Programming Languages, 2020, 4(OOPSLA): 1-29.
[40] Wang Z, Xu L. Grading programs based on hybrid analysis [C]//Web Information Systems and Applications: 16th International Conference, WISA 2019, Qingdao, China, September 20-22, 2019, Proceedings 16. Springer, 2019: 626-637.
[41] Brieven G, Baum V, Donnet B. Tartare: Automatic generation of c pointer statements and feedback [C]// Proceedings of the 26th Australasian Computing Education Conference. 2024: 192-201.
[42] Ferrante J, Ottenstein K J, Warren J D. The program dependence graph and its use in optimization [J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 1987, 9(3): 319-349.
[43] Huang J. Program instrumentation and software testing [J]. Computer, 1978, 11(4): 25-32.
[44] Bromley J, Guyon I, LeCun Y, et al. Signature verification using a" siamese" time delay neural network [J]. Advances in neural information processing systems, 1993, 6.
[45] Szegedy C, Liu W, Jia Y, et al. Going deeper with convolutions [C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2015: 1-9.
[46] Rumelhart D E, Hinton G E, Williams R J. Learning representations by back-propagating errors [J]. nature, 1986, 323(6088): 533-536.
[47] Reimers N, Gurevych I. Sentence-bert: Sentence embeddings using siamese bert-networks [J]. arXiv preprint arXiv:1908.10084, 2019.

[48] De Moura L, Bjørner N. Z3: An efficient smt solver [C]//International conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2008: 337-340.
[49] Aycock J. A brief history of just-in-time [J]. ACM Computing Surveys (CSUR), 2003, 35(2): 97-113.
[50] Needleman S B, Wunsch C D. A general method applicable to the search for similarities in the amino acid sequence of two proteins [J]. Journal of molecular biology, 1970, 48(3): 443-453.
[51] Ouyang L, Wu J, Jiang X, et al. Training language models to follow instructions with human feedback [J]. Advances in neural information processing systems, 2022, 35: 27730-27744.
[52] Böhme M, Pham V T, Nguyen M D, et al. Directed greybox fuzzing [C]//Proceedings of the 2017 ACM SIGSAC conference on computer and communications security. 2017: 2329-2344.

附录 1 攻读硕士学位期间申请的专利

[1] 胡天昊，姜一鸣，张卫丰. 针对代码仓库基于 transition 转换系统的 commit 查询语言自动生成方法，202310390767.9，2023.4,2023.5

致谢

感谢我的导师张卫丰，他们给予我耐心和指导，使我能够顺利完成毕业论文。在科研中，他对我的启发和建议不仅提高了我的研究水平，也拓展了我的视野；在我撰写论文时耐心地给予我指导，提出宝贵的意见和改进建议，让我的论文更加完善和专业；在前路规划上，给予我宝贵的建议和指引，启发我思考未来的发展方向，让我更加明确自己的职业发展和个人成长方向。

感谢我的室友们，他们给予了我无私的支持和关心。他们不仅在我熬夜写论文时给予了理解和鼓励，还在生活上给予了我无微不至的关怀和帮助。

感谢在华为实习时的同事，他们让我更好的了解行业发展，开拓自己的视野。

最后感谢我的父母、女朋友和家人们。在我的求学之路上，他们不断地支持我鼓励我并提供给我无微不至的关怀。

在此，感谢每一位在我成长道路上相遇的人，让我变得更加坚强和成熟，我会继续拼搏，向人生的新阶段迈进！