پروژه تشخیص سرقت ادبی در برنامه‌نویسی با رویکرد چندسطحی

درس طراحی کامپایلر – نیم‌سال اول ۰۵-۰۴ - دکتر علائیان

گروه ۴

سارینا ناصرمقدسی ۴۰۲۲۲۰۸۳ حنانه احمدی 40247213

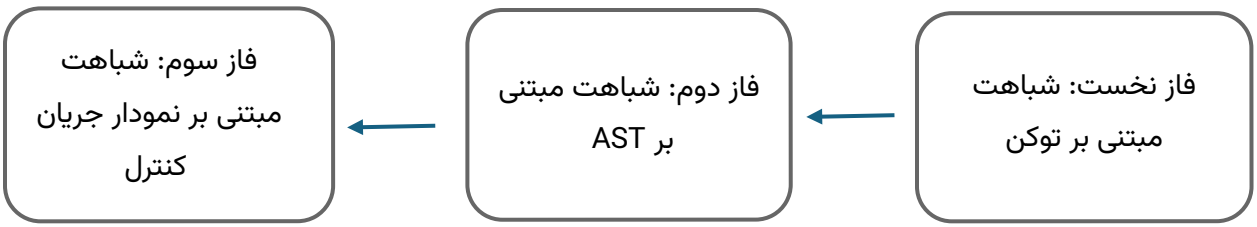
گیت هاب پروژه: https://github.com/srn-nm/compiler-project/

مقدمه

پروژه «تشخیص سرقت ادبی در برنامه‌نویسی» با هدف طراحی و پیاده‌سازی یک سامانه چندسطحی برای سنجش میزان شباهت میان دو قطعه کد تعریف شده است. این سامانه با تحلیل کدها در سه سطح توکن، **درخت نحوی انتزاعی (AST)** و **گراف جریان کنترل (CFG)**، قادر به شناسایی انواع مختلف کپی‌برداری از تغییرات سطحی تا بازنویسی‌های عمیق ساختاری و رفتاری می‌باشد. در این مستند، تمامی اجزای طراحی شده شامل تحلیلگر لغوی، نحوی، گرامرها، الگوریتم‌های تشخیص شباهت و رابط کاربری به تفصیل شرح داده شده است.

معماری کلی سیستم

سامانه از سه فاز اصلی تشکیل شده که به صورت زنجیره‌ای و مکمل عمل می‌کنند. خروجی هر فاز به عنوان ورودی فاز بعدی مورد استفاده قرار می‌گیرد. شکل زیر نمای کلی معماری سیستم را نشان می‌دهد.



- فاز اول: دریافت کد خام، انجام تحلیل لغوی، استخراج توکن‌ها و محاسبه شباهت مبتنی بر توکن.
- فاز دوم: دریافت کد، ساخت درخت نحوی انتزاعی، نرمال‌سازی و مقایسه ساختار درخت‌ها.
- فاز سوم: دریافت AST، ساخت گراف جریان کنترل، اعمال الگوریتم‌های تطبیق گراف و محاسبه شباهت رفتاری.

طراحی Lexer تحلیلگر لغوی

تحلیلگر لغوی وظیفه تبدیل رشته کد ورودی به جریانی از توکن‌های معنادار را بر عهده دارد. برای پشتیبانی از زبان‌های مختلف، از **Strategy Pattern** استفاده شده و هر زبان دارای یک هندلر اختصاصی است.

ساختار هندلرهای زبان

کلاس پایه BaseLanguageHandler در فایل base_handler.py تعریف شده و شامل متدهای انتزاعی get_tokens و get_normalized_tokens می‌باشد.

همچنین توابع مشترکی مانند preprocess_code و normalize_whitespace در این کلاس پیاده‌سازی شده‌اند.

پیاده‌سازی برای پایتون

Python Handler از دو روش برای توکن‌سازی استفاده می‌کند:

۱. **ANTLR** : با بهره‌گیری از گرامر Python3Lexer.g4 و کلاس تولید شده، توکن‌ها با دقت بالا استخراج می‌شوند.

۲. **روش جایگزین (Fallback):** در صورت عدم دسترسی به فایل‌های ANTLR، از عبارت‌های با قاعده برای تشخیص توکن‌ها استفاده می‌شود.

متد _normalize_identifiers نام شناسه‌ها را به IDENTIFIER تبدیل کرده و _
normalize_literals اعداد و رشته‌ها را با برچسب‌های NUMBER و STRING جایگزین می‌کند.

پیاده‌سازی برای جاوا و C++

هندلرهای JavaHandler و CppHandler نیز به طور مشابه طراحی شده‌اند. برای C++ به دلیل پیچیدگی بیشتر، الگوهای regex تخصصی‌تری برای شناسایی هدرها، عملگرهای ترکیبی و کامنت‌های چندخطی تعبیه شده است.

پیش‌پردازش و نرمال‌سازی

تمامی هندلرها کد ورودی را پیش از توکن‌سازی، مطابق تنظیمات پیکربندی (config.json) پردازش می‌کنند:

- **حذف کامنت‌ها** با توجه به نحو هر زبان
- **نرمال‌سازی فاصله‌ها** تبدیل چند فاصله به یک فاصله، حذف خطوط خالی
- **یکسان‌سازی حروف** در صورت غیر حساس بودن به بزرگی یا کوچکی حروف
- **نرمال‌سازی شناسه‌ها و مقادیر ثابت** برای افزایش دقت در تشخیص شباهت

طراحی Parser و AST

فاز دوم سامانه، تحلیل ساختاری کدها را بر عهده دارد. هسته این بخش، ساخت درخت نحوی انتزاعی و مقایسه آن‌هاست.

کلاس ASTNode

کلاس ASTNode در analyzer.py نمایش یک گره درخت را بر عهده دارد. هر گره شامل موارد زیر است:

- node_type نوع گره (مانند FunctionDef, If, Name)

- value مقدار (در صورت وجود، مانند نام تابع یا مقدار ثابت)
- children لیست گره‌های فرزند
- line و col موقعیت در کد منبع
- hash هش محاسبه شده از نوع گره و هش فرزندان (برای مقاومت در برابر تغییر نام)
- structural_hash هش تنها بر اساس ساختار و بدون در نظر گرفتن نوع گره

(متد to_dict ساختار درخت را به صورت دیکشنری بازمی‌گرداند که برای انتقال به فاز سوم (CFG) و ذخیره‌سازی در JSON استفاده می‌شود).

تبدیل کد به AST

در ASTSimilarityAnalyzer.parse_code با توجه به زبان ورودی، متد مناسب فراخوانی می‌شود. برای پایتون، از ماژول استاندارد ast استفاده شده و با پیمایش بازگشتی، شیء ASTNode ساخته می‌شود.

```
def parse_python_ast(self, code: str) -> ASTNode:
    tree = ast.parse(code)
    return self.convert_ast_node(tree)
```

همان طور که در بالا مشاهده می شود، متد convert_ast_node گره‌های ast را به ASTNode تبدیل کرده و در صورت فعال بودن گزینه نرمال‌سازی، نام‌ها و مقادیر ثابت را تغییر می‌دهد.

نرمال‌سازی گره‌ها

- **نرمال‌سازی شناسه‌ها:** تمامی نام متغیرها، توابع و کلاس‌ها به VAR, FUNC, CLASS تبدیل می‌شوند مگر آنکه در لیست استثناها (مانند cout, print) باشند.

- **نرمال‌سازی مقادیر ثابت:** اعداد به NUMBER، رشته‌ها به STRING، بولین‌ها به BOOLEAN و ... تبدیل می‌گردند.

هش ساختاری

دو نوع هش برای مقاومت در برابر تغییرات سطحی محاسبه می‌شود:

- hash: ترکیب node_type و هش فرزندان → تغییر نوع گره را منعکس می‌کند.
- structural_hash: تنها هش فرزندان (بدون در نظر گرفتن نوع گره) → برای مقایسه ساختار صرف نظر از برچسب‌ها.

گرامرها

برای زبان‌های پایتون، جاوا و C++، از گرامرهای رسمی ANTLR استفاده شده است. این گرامرها از مخازن رسمی ANTLR برگرفته و در پوشه grammars/generated قرار گرفته‌اند.

فایل‌های گرامر ANTLR

- Python3Lexer.g4 / Python3Parser.g4
- JavaLexer.g4 / JavaParser.g4
- CPP14Lexer.g4 / CPP14Parser.g4

قوانین Lexer

در هر گرامر، توکن‌های کلیدی، عملگرها، جداکننده‌ها و لیترال‌ها تعریف شده‌اند. به عنوان نمونه، بخشی از گرامر پایتون:

تولید کد از گرامر

با استفاده از ابزار ANTLR، فایل‌های Lexer و Parser به زبان پایتون تولید شده‌اند. دستور نمونه:

```
antlr4 -Dlanguage=Python3 -visitor -o generated/python Python3Lexer.g4
Python3Parser.g4
```

این فایل‌های تولید شده در مسیر grammars/generated/python/ قرار گرفته و توسط هندلرهای مربوطه فراخوانی می‌شوند.

الگوریتم‌های تشخیص شباهت

فاز اول: شباهت مبتنی بر توکن

هدف: تشخیص کپی‌برداری مستقیم و تغییرات جزئی در سطح کد.

مراحل:

۱. توکن‌سازی و نرمال‌سازی کدها توسط هندلر زبان.

۲. محاسبه معیارهای شباهت:

- **Jaccard Similarity** بر روی مجموعه توکن‌ها.
- **Cosine Similarity** بردار فراوانی توکن‌ها.
- **Levenshtein Similarity** فاصله ویرایش رشته توکن‌ها.
- **Sequence Similarity** بر اساس طولانی‌ترین زیررشته مشترک.

۳. ترکیب وزن‌دار معیارها (وزن‌های پیش‌فرض: 0.2، 0.3، 0.3، 0.2).

۴. استخراج بخش‌های مشابه با الگوریتم تطبیق پنجره‌ای.

۵. ذخیره‌سازی نتایج شامل درصد شباهت، آمار توکن‌ها و بخش‌های مشابه در قالب JSON.

کلاس اصلی: TokenSimilarityAnalyzer در phase1/token_similarity_analyzer.py.

خروجی فاز اول پس از اجرای دستور مقابل:

python phase1/src/main.py phase1/tests/test_python/code1.py phase1/tests/test_python/code2.py --visual –verbose

خروجی:

Detailed Metrics:

Jaccard Similarity : 94.12%

Cosine Similarity : 99.78%

Levenshtein Similarity : 76.13%

Sequence Similarity : 76.63%

Normalized Jaccard : 94.12%

Token Statistics:

File 1: 472 tokens (33 unique types)

File 2: 599 tokens (33 unique types)

Common token types: 32

Most Common Tokens:

File 1: NAME(112), NEWLINE(56), NUMBER(38), OPEN_PAREN(30), CLOSE_PAREN(30)

File 2: NAME(148), NEWLINE(77), NUMBER(42), OPEN_PAREN(38), CLOSE_PAREN(38)

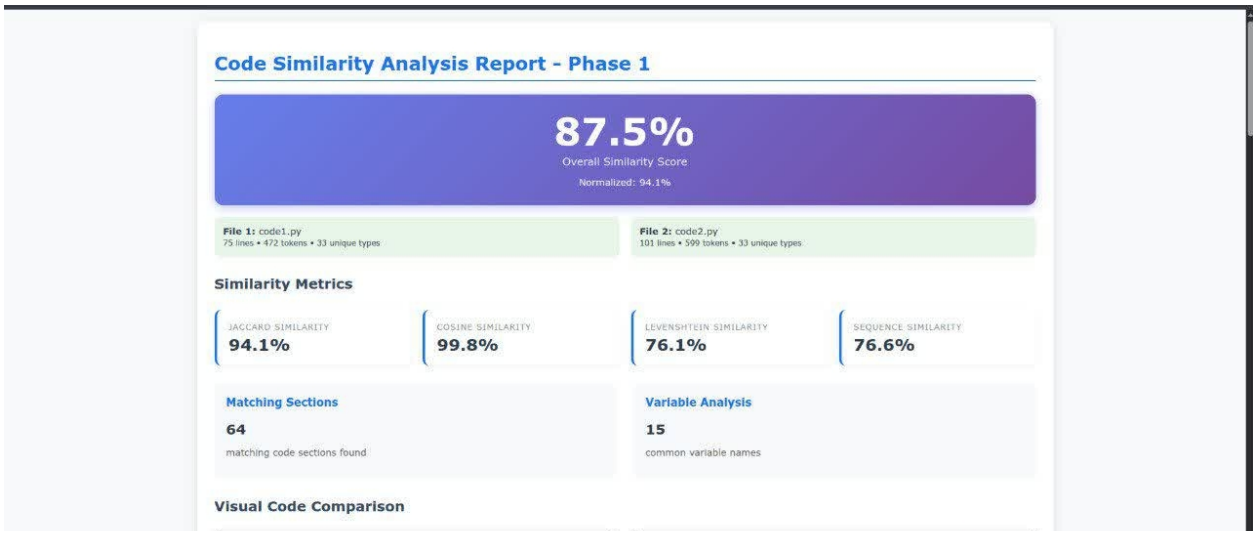
نتایج در فایل‌های زیر ذخیره خواهد شد:

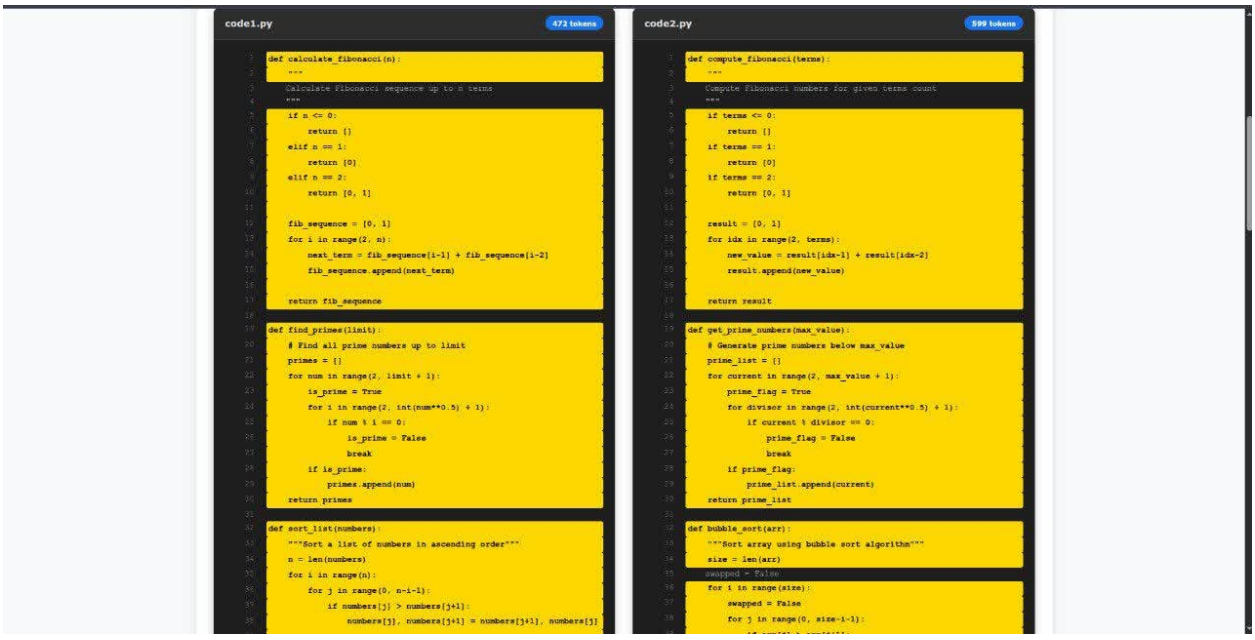
JSON: phase1/results/similarity_report_Phase1.json

HTML: phase1/results/similarity_report_Phase1.html

Text: phase1/results/similarity_report_Phase1.txt

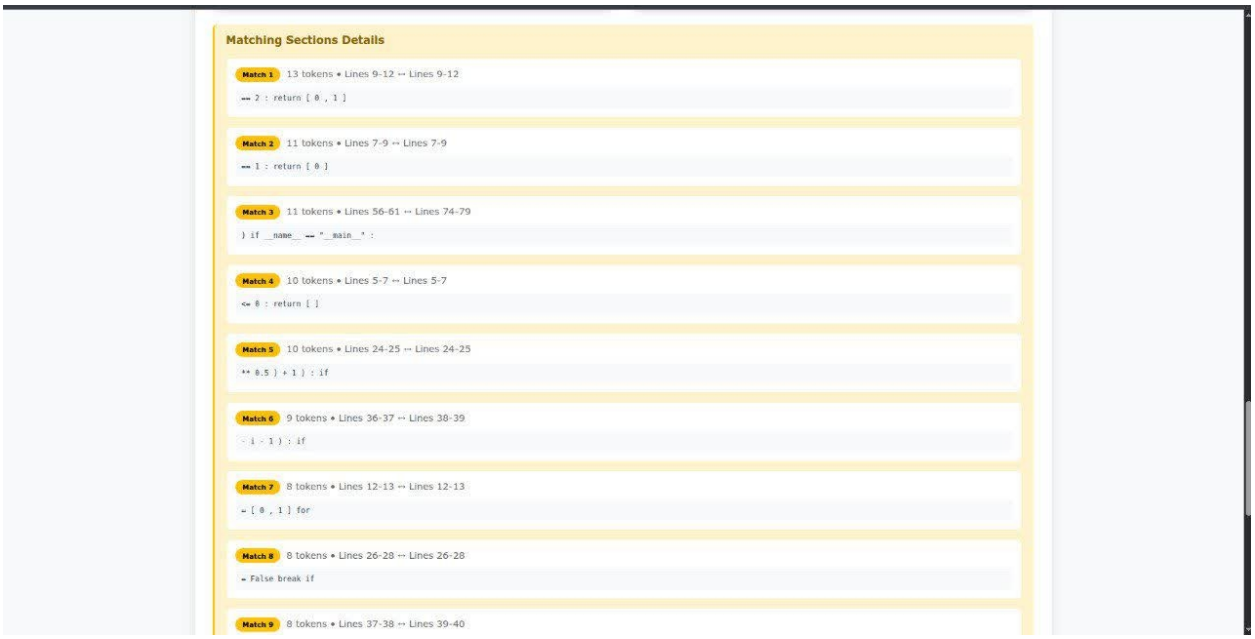
Overall similarity: 87.5%





```
    # Test Fibonacci
    fib_result = calculate_fibonacci(10)
    print(f"First 10 Fibonacci numbers: {fib_result}")

    # Test primes
    prime_result = find_primes(50)
    print(f"Primes up to 50: {prime_result}")

    # Test sorting
    numbers_to_sort = [64, 34, 25, 12, 22, 11, 90]
    sorted_numbers = sort_list(numbers_to_sort.copy())
    print(f"Sorted list: {sorted_numbers}")

    # Test student class
    student1 = Student("Alice", 20, [85, 90, 78, 92, 88])
    student1.display_info()
```

```
    def compute_average_score(self):
        if len(self.scores) == 0:
            return 0.0
        total = 0.0
        for score in self.scores:
            total += score
        return total / len(self.scores)

    def show_details(self):
        average = self.compute_average_score()
        print(f"Name: {self.full_name}")
        print(f"Age: {self.years_old}")
        print(f"Average Score: {average:.1f}")

# Driver code
if __name__ == "__main__":
    # Fibonacci calculation
    fib_output = compute_fibonacci(8)
    print(f"Fibonacci sequence (8 terms): {fib_output}")

    # Prime numbers
    primes = get_prime_numbers(30)
    print(f"Prime numbers up to 30: {primes}")

    # Sorting example
    data = [45, 23, 67, 12, 89, 34, 56]
    sorted_data = bubble_sort(data.copy())
    print(f"Original: {data}")
    print(f"Sorted: {sorted_data}")

    # Student information
    grades = [88, 92, 76, 95, 84]
    student = UniversityStudent("Bob", 22, grades)
    student.show_details()

    # Additional test
    x = 10
    y = 20
    sum_result = x + y
    print(f"Sum of {x} and {y} is {sum_result}")
```



Matching Sections Details

Match 1 13 tokens • Lines 9-12 ↔ Lines 9-12

```
== 2 : return [ 0 , 1 ]
```

Match 2 11 tokens • Lines 7-9 ↔ Lines 7-9

```
== 1 : return [ 0 ]
```

Match 3 11 tokens • Lines 56-61 ↔ Lines 74-79

```
) if __name__ == "__main__" :
```

Match 4 10 tokens • Lines 5-7 ↔ Lines 5-7

```
<= 0 : return [ ]
```

Match 5 10 tokens • Lines 24-25 ↔ Lines 24-25

```
** 0.5 ) + 1 ] : if
```

Match 6 9 tokens • Lines 36-37 ↔ Lines 38-39

```
- i - 1 ) : if
```

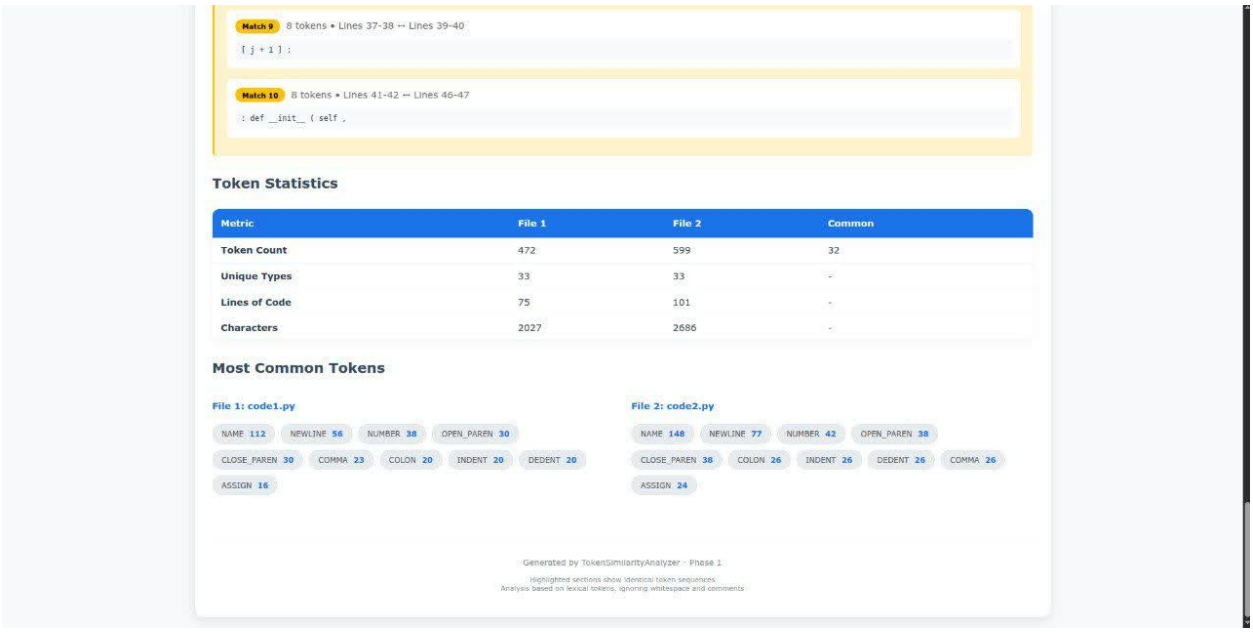
Match 7 8 tokens • Lines 12-13 ↔ Lines 12-13

```
= [ 0 , 1 ] for
```

Match 8 8 tokens • Lines 26-28 ↔ Lines 26-28

```
= False break if
```

Match 9 8 tokens • Lines 37-38 ↔ Lines 39-40



فاز دوم: شباهت مبتنی بر AST

هدف: شناسایی شباهت‌های ساختاری با وجود تغییر نام متغیرها، بازآرایی توابع و تغییرات سطحی.

الگوریتم‌ها:

- **Structural Similarity:** مقایسه مجموعه هش‌های ساختاری همه گره‌ها (Jaccard).
- **Node Type Distribution Similarity:** شباهت کسینوسی توزیع انواع گره.
- **Subtree Matching Similarity:** اشتراک هش زیردرخت‌ها با عمق محدود.
- **Tree Depth Similarity:** شباهت بر اساس اختلاف عمق درخت.

نمره نهایی AST: ترکیب وزن‌دار سه معیار اصلی (ساختاری، توزیع نوع، تطبیق زیردرخت).

ذخیره‌سازی AST: خروجی فاز ۲ شامل دیکشنری ast1_dict و ast2_dict است که توسط متد to_dict از شیء ASTNode تولید می‌شود. این دیکشنری‌ها در فاز ۳ برای ساخت CFG استفاده می‌گردند.

کلاس اصلی: ASTSimilarityAnalyzer و Phase2ASTSimilarity در phase2/src/analyzer.py.

خروجی با اجرای دستور زیر :

Command line:

python -m phase2.src.main --phase1-results phase1/results/similarity_report_Phase1.json -f1 phase1/tests/test_python/code1.py -f2 phase1/tests/test_python/code2.py

Loading Phase 1 results...

Phase 1 results loaded: 87.5% similarity

Phase 1 loaded: 87.5% similarity

- Matching sections: 64
- Common variables: 15

Combined Score: 80.4%

- Token: 87.5%
- AST: 77.4%

AST Metrics:

- Structural: 54.9%
- Node Type: 99.8%
- Subtree: 85.0%

• Depth: 100.0%

Decision:

• Threshold: 65%

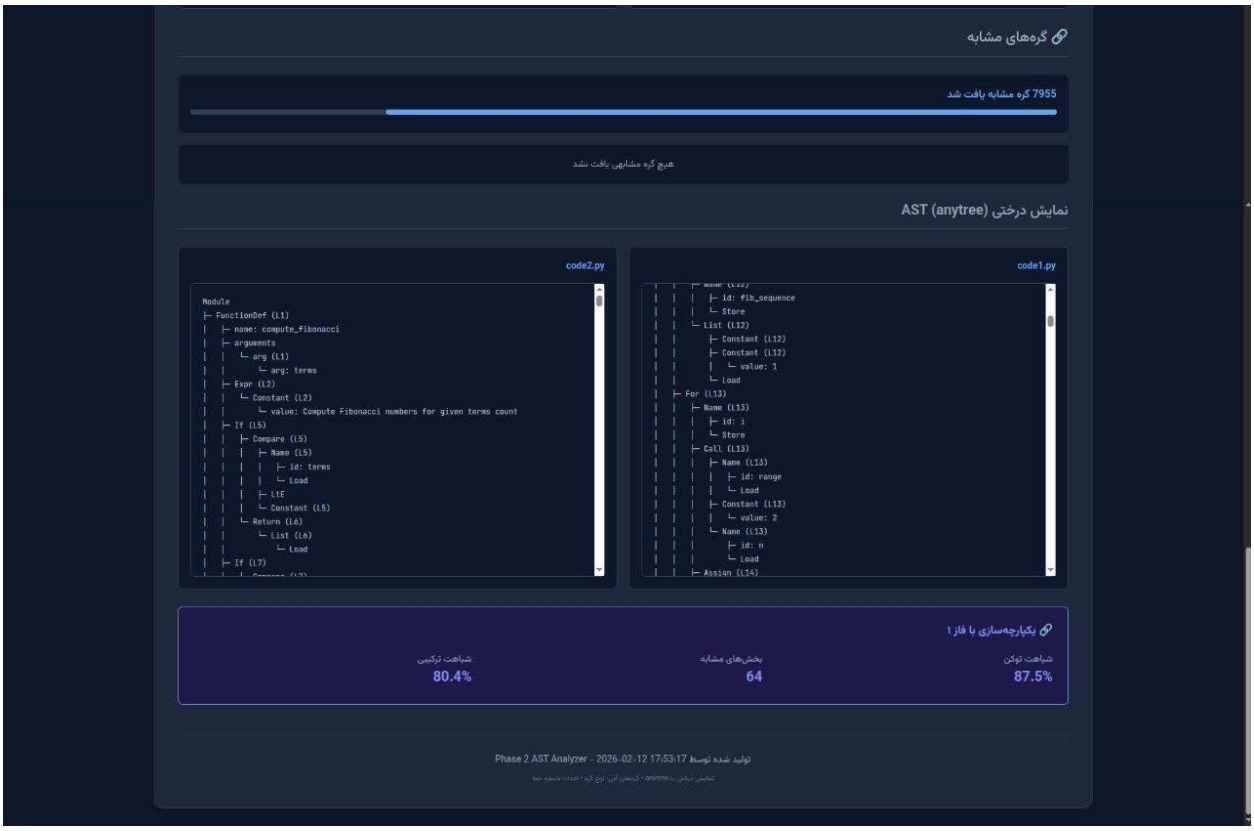
• Result: SIMILAR (Possible Plagiarism)

نهایتاً اطلاعات به دست آمده در آدرس‌های زیر ذخیره می شوند:

JSON report: phase2/results/phase2_report.json

HTML report saved: phase2/results/phase2_report_integrated.html

Final Similarity Score: 80.4%



فاز سوم: شباهت مبتنی بر CFG

هدف: تحلیل رفتار اجرایی برنامه و تشخیص سرقت‌های پنهان در سطح منطق.

ساخت CFG از AST:
کلاس CFGBuilder با پیمایش دیکشنری AST، گره‌های CFG را ایجاد می‌کند. هر گره دارای نوع (ENTRY, EXIT, BASIC_BLOCK, DECISION, LOOP, ...) و لیست یال‌های ورودی/خروجی است. ساختارهای کنترلی مانند while، for، if به گراف تبدیل می‌شوند.

الگوریتم‌های شباهت گراف:

- **Structural Similarity** مقایسه ویژگی‌های سطح گراف (تعداد گره/یال، درجه، توزیع نوع، پیچیدگی سیکلوماتیک).

- **Graph Edit Distance (GED)** یافتن بهترین تطبیق بین گره‌ها با ماتریس شباهت و محاسبه هزینه ویرایش.
- **Subgraph Matching** استخراج زیرگراف‌های مهم (حول گره‌های تصمیم و حلقه) و مقایسه آن‌ها.
- **Execution Path Similarity** شباهت مسیرهای اجرایی مبتنی بر LCS.

یکپارچه‌سازی:

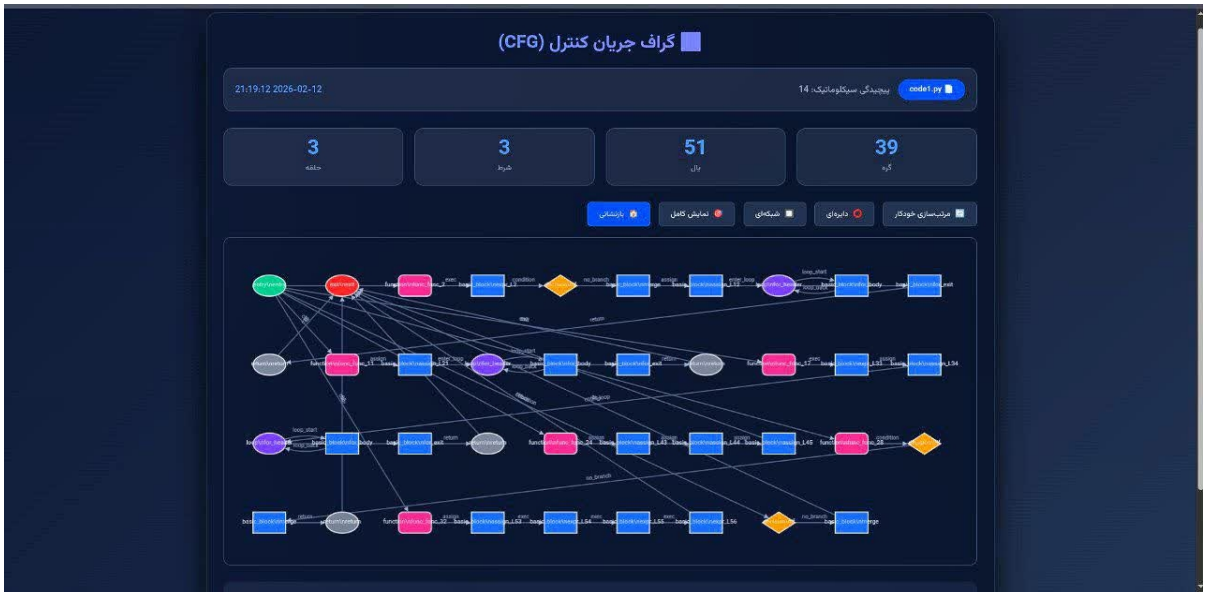
کلاس Phase3CFGSimilarity نتایج فازهای ۱ و ۲ را دریافت کرده، AST واقعی را از phase2_results استخراج و به CFGAnalyzer ارسال می‌کند. سپس نمره CFG را با نمرات قبلی ترکیب می‌کند.

کلاس‌های اصلی:

CFGAnalyzer, Phase3CFGSimilarity در phase3/analyzer/cfg_analyzer.py

GraphSimilarity در phase3/analyzer/graph_similarity.py

خروجی :



نتیجه‌گیری

در این پروژه یک سامانه تشخیص سرقت ادبی سه‌سطحی با قابلیت تحلیل توکن، ساختار و رفتار برنامه پیاده‌سازی شد. استفاده از ANTLR برای تحلیل دقیق زبان‌های مختلف و به‌کارگیری الگوریتم‌های پیشرفته تطبیق گراف، دقت تشخیص را نسبت به روش‌های سنتی بهبود می‌بخشد.