
# 算法设计与分析

实验报告（二）查重

| | |
|---|---|
| 姓　　名 | 熊恪峥 |
| 学　　号 | 22920202204622 |
| 日　　期 | 2022年3月23日 |
| 学　　院 | 信息学院 |
| 课程名称 | 算法设计与分析 |

# 实验报告（二）查重

## 目录

# 1 问题描述

# 2 实现思路

要求给定两个程序，判断它们的相似性。显然，*程序的相似性和代码字符串的相似性无关，而与实际执行逻辑的相似性有关*，例如1中有两段代码本身不尽相同的代码，但是执行逻辑完全一致。那么最准确的方式是进行DFA(Data Flow Analysis)和CFA(Control Flow Analysis)，对于相似的程序它们应当能相当准确地反映出相似度。这正是现代IDE对重复代码给出修改建议的方式。但这种方式实现相当复杂，本程序通过对问题进行简化有效地实现了**基于语义的**代码相似性判断。

图 1: 逻辑相同但代码本身差异较大的代码

```
 int main()
{
         bool a = true;
        if (a)
        {
                printf("helloworld");
        }
        else
        {
                printf("worldhello");
        }
        return 0;
}
```

```
 int main()
{
         int the_flag = 1;
        if (the_flag)
        {
                puts("helloworld");
        }
        else
        {
                puts("worldhello");
        }
        return 0;
}
```

## 2.1 两点假设

为了简化问题，首先进行以下两个假设

- 程序的抽象语法树(AST, Abstract Syntax Tree)和实际执行逻辑高度相关
- 抽象语法树中的语句节点和表达式节点是所有节点中和实际执行逻辑最相关的两类节点

根据这两点假设，通过从程序编译时的抽象语法树的语句(Statement)节点和表达式(Expression)节点序列中寻找最长公共子序列可以有效地衡量程序的逻辑相似性。定义逻辑相似度$s$，其中$AST_j$是程序代码$j$的抽象语法树的语句节点和表达式节点序列

$$s = \frac{|LCS_{AST_1,AST_2}|}{\max\{|AST_1|,|AST_2|\}}$$

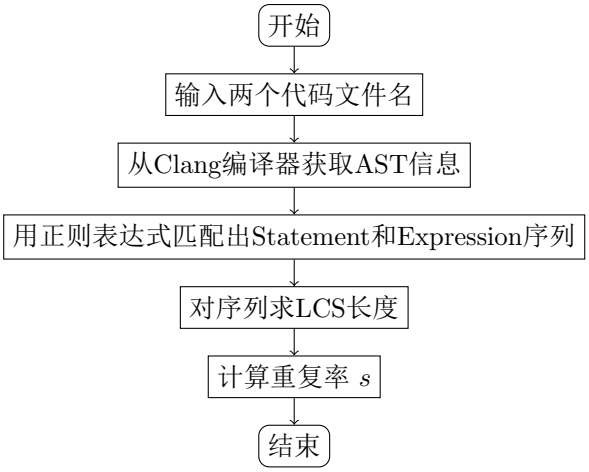
## 2.2 算法

根据以上分析，首先需要获得AST中Statement和Expression组成的序列。这里借助Clang编译器。通过观察可以发现输出结果中Statement和Expression都符合正则表达式(1)。

$$([a-zA-Z\_][0-9a-zA-Z\_]*Expr)|([a-zA-Z\_][0-9a-zA-Z\_]*Stmt) \tag{1}$$

因此可以用正则表达式匹配输出来获得上述序列。

图 2: 查重流程图设计




根据以上分析可以得到算法流程图 2，然后可以实现核心部分如代码 1，完整代码见附录：代码实现节中的代码 3。该实现依赖Clang编译器。

代码 1: 核心部分代码

```python
def lcs(s1, s2):
    f = [[0] *  len(s1) * 2] *  len(s2) * 2

    for i in  range(1,  len(s1) + 1):
        for j in  range(1,  len(s2) + 1):
            if s1[i - 1] == s2[j - 1]:
                f[i][j] = 1 + f[i - 1][j - 1]
            else:
                f[i][j] =  max(f[i - 1][j], f[i][j - 1])

    return f[ len(s1)][ len(s2)]


def duplication_check(file1:  str, file2:  str):
    ast1 =  str(shell(['clang -cc1 -ast-dump {}'. format(file1)]))
    ast2 =  str(shell(['clang -cc1 -ast-dump {}'. format(file2)]))

    elems1 =  list([i[0] if i[0] != '' else i[1] for i in
                re.findall(r'([a-zA-Z\_][0-9a-zA-Z\_]*Expr)|([a-zA-Z\_][0-9a-zA-Z\_]*Stmt)', ast1
                    )])
    elems2 =  list([i[0] if i[0] != '' else i[1] for i in
                re.findall(r'([a-zA-Z\_][0-9a-zA-Z\_]*Expr)|([a-zA-Z\_][0-9a-zA-Z\_]*Stmt)', ast2
                    )])

    print("Repeat Rate: {}%". format((lcs(elems1, elems2) /  max( len(elems1),
        len(elems2))) * 100.0))
```

使用以上程序对图 1中的代码进行查重，输出的重复率是81.25%，可以看出比起直接比较代码文面，该算法有效地反映出了底层逻辑的相似性，排除了修改变量名等传统降重方法造成的干扰。

## 2.3 结果分析

根据上述测试结果可以得出结论，通过AST得节点序列可以有效地刻画程序的相似度。然而这种方法的

思路依然是使用更高层级的“形式相似性”近似“逻辑相似性”，它考虑了一定程度的语义信息。因此，可以使用更为高级的降重技巧规避，例如

- 改变语句顺序
- 将递归结构改为非递归结构
- 将一部分代码移动到一个子过程中

因此这种方式相比于正规的程序静态分析手段而言还是有不足的。然而这种方法实现计算简单，计算量较少，在实际使用中有一定的优势。

# A 附录：代码实现

## A.1 矩阵连乘

代码 2: 矩阵连乘

```
p =  list([5, 10, 3, 12, 5, 50, 6])
N = 6

m =  list([ list([0x7fffffff for i in  range(0, N + 1)]) for j in  range(0, N + 1)])
s =  list([ list([0x7fffffff for i in  range(0, N + 1)]) for j in  range(0, N + 1)])


def pretty_print(i, j):
    if i == j:
        print('A{}'. format(i), end=' ')
    else:
        print("(", end=' ')
        pretty_print(i, s[i][j])
        pretty_print(s[i][j] + 1, j)
        print(")", end=' ')


for i in  range(1, N + 1):
    m[i][i] = 0

for i in  range(N, 0, -1):
    for j in  range(i, N + 1):
        if i == j:
            m[i][j] = 0
        else:
            for k in  range(i, j):
                val = m[i][k] + m[k + 1][j] + p[i - 1] * p[j] * p[k]
                if m[i][j] > val:
                    m[i][j] = val
                    s[i][j] = k

for i in  range(1, N + 1):
    for j in  range(1, N + 1):
        print("inf" if m[i][j] == 0x7fffffff else m[i][j], end=' ' if j != N else '\n')

for i in  range(1, N + 1):
    for j in  range(1, N + 1):
        print("None" if s[i][j] == 0x7fffffff else s[i][j], end=' ' if j != N else '\n')

pretty_print(1, N)
```

## A.2 查重程序

注意： **该实现依赖*Clang*编译器**

代码 3: 查重程序

```
import subprocess
from typing import Final
import re
import argparse


def shell(command):
    try:
        return subprocess.check_output(command, shell=True, stderr=subprocess.STDOUT).stdout
    except subprocess.CalledProcessError as exc:
        return exc.output


def lcs(s1, s2):
    f = [[0] *  len(s1) * 2] *  len(s2) * 2

    for i in  range(1,  len(s1) + 1):
        for j in  range(1,  len(s2) + 1):
            if s1[i - 1] == s2[j - 1]:
                f[i][j] = 1 + f[i - 1][j - 1]
            else:
                f[i][j] =  max(f[i - 1][j], f[i][j - 1])

    return f[ len(s1)][ len(s2)]


def duplication_check(file1:  str, file2:  str):
    ast1 =  str(shell(['clang -cc1 -ast-dump {}'. format(file1)]))
    ast2 =  str(shell(['clang -cc1 -ast-dump {}'. format(file2)]))

    elems1 =  list([i[0] if i[0] != '' else i[1] for i in
                    re.findall(r'([a-zA-Z\_][0-9a-zA-Z\_]*Expr)|([a-zA-Z\_][0-9a-zA-Z\_]*Stmt)', ast1
                        )])
    elems2 =  list([i[0] if i[0] != '' else i[1] for i in
                    re.findall(r'([a-zA-Z\_][0-9a-zA-Z\_]*Expr)|([a-zA-Z\_][0-9a-zA-Z\_]*Stmt)', ast2
                        )])

    print("Repeat Rate: {}%". format((lcs(elems1, elems2) /  max( len(elems1),
        len(elems2))) * 100.0))


if __name__ == "__main__":
    parser = argparse.ArgumentParser(prog="dupcheck",
                                     usage='%(prog)s [options] file1 file2',
                                     description="Duplication checker")
    parser.add_argument("file1")
    parser.add_argument("file2")

    args = parser.parse_args()

    duplication_check( str(args.file1),  str(args.file2))
```