COMP3211 Software Engineering

Project Report

Group 24

Zheng Hongyi 13104036D

Chen Yunkun 13103214D

Li Dawei 13103693D

1. Coverage Information

For getting the coverage information of testing programs, we choose Coverage.py package as primal tool. The detail procedures are described as following:

1. Run “coverage run ./testing program input_case” on each bug version of program and test case for generating coverage information.
2. Run “coverage report -m” for print out coverage information with line number of missing lines.
3. For each output of Coverage.py, transfer it into a matrix file describe in papers. Each matrix is corresponding to a bug version of testing programme. Each row represents one test case and each column represents one line in the programming, where 1 represents “covered” and 0 vise versa. The last column represents success (represented by 0) and failure (represented by 1) of the programme.

```
1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
1 1 1 1 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0
```

We wrote a Python script Auto.py for the automation of above steps. The output matrix are saved and further feed into fault localization programs described later.

```
    for line in lines:
        vector = ""
        miss = []
        line = line.replace("\n", "")
        print(line)
        success = call(["coverage", "run", source, line])
        print("finish")
        output = check_output(["coverage", "report", "-m"])
        coverage = output.decode("utf-8").split('\n')[2]
        items = coverage.split()

        total = file_len(source)
```

2. Testing Programs

2.1. Testing Programs

For test programs, we choose two programs for testing:

1. 4-element naïve sorting program

   This program receives 4 integer numbers, and aims to give a sequence of these 4 numbers with descending order. For example, the input is [4,2,3,5], the correct output should be [5,4,3,2]. If there exist numbers with the same value, either sequence between or among them is ok. For example, the input is [1,1,3,1], and the correct output should be [3,1,1,1].

2. Simple encrypt-decrypt program

   This program receives a string as input, and uses a map among letters to encrypt it. After encryption, a decryption will make the string back to the original one. In this program, letters will be encrypted through its position on a QWERTY keyboard. That is, ‘a’=>’q’, ‘b’=>’w’, ‘c’=>’e’, ‘d’=>’r’ and so on. For example, the input is “I love software engineering”, the correct output should also be “I love software engineering”.

As for solving oracle problem, it is easy to determine whether the two programs give correct results or not. For the sorting program, we check whether the numbers are shown in descending order, which is easy to judge by both people and computer. We compared the result with a correct sorting program, and determine whether they have the same output. For the encrypt-decrypt program, it does not matter what encrypt method is used, so we only need to judge whether the output is the same as input.

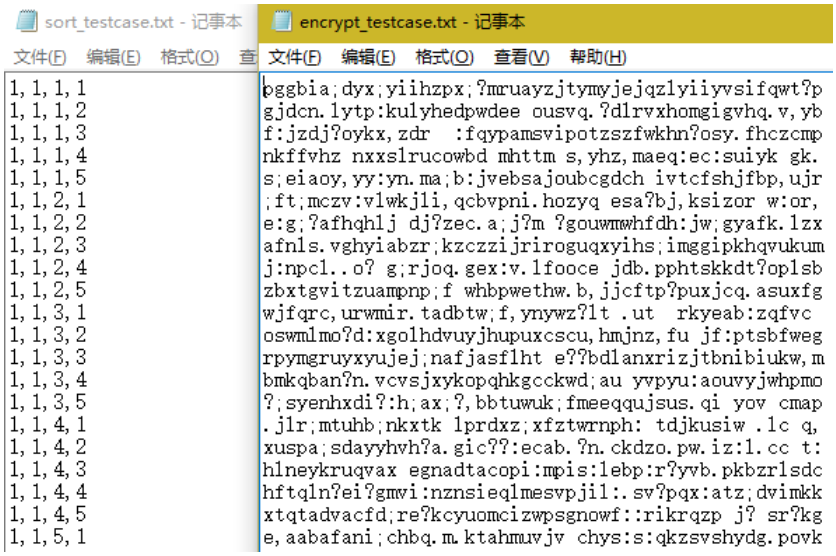
2.2. Bugs

For each program, we have 10 buggy versions of it. Each buggy program is made by changing one line of code of the correct program. The figure below shows the position of bugs and how they are different from the correct one.

```
[5, "if (e >= f):","if (e == f):"],
[10,"return [c, f, d, e]","return [f, c, d, e]"],
[13,"else:","elif (e>=f):"],
[16,"if (c >= e):","if(c<=e):"],
[17,"return [c, e, d, f]","return [c, c, d, f]"],
[24,"if (c >= f):","if (c >= d):"],
[38,"if (e >= f):","if (e >= f) or (d >= f):"],
[43,"return [d, f, c, e]","return [d, f, 2, e]"],
[49,"if (d >= e):","if c==2:"],
[62,"if (d >= f):","if True:"]
```

2.3. Test Cases

We create totally 1125 test cases for these two programs. 625 are for sorting program, and 500 are for encrypt-decrypt program. That is, each buggy sorting program will run 625 times, while each buggy encrypt-decrypt program will run 500 times. Here show the test cases:



We implemented and tested two auto-debugging methods: Tarantula Automatic Fault-Localization Technique and CrossTab-Based Statistical Method.

3. Fault-Localization Techniques

3.1. Tarantula Automatic Fault-Localization Technique:

We use the basic model that described in the paper, the insight of the Tarantula model is that entities in a program that are primarily executed by failed test cases are more likely to be faulty than those that are primarily executed by passed test cases. Thus the suspicious can be calculated by this formula:

$$Suspicious(s) = \frac{fail(s) / totalfail}{fail(s) / totalfail + pass(s) / totalpass}$$

The problem we may face in the actual implementation stage is that the value of fail(s), and pass(s) can be 0.

From the formula, we can know that if the value of fail(s) is 0, the suspicious(s) should be 0, and similarly, if the value of pass(s) is 0, the suspicious(s) should be 1. So we need to ensure that the program return 0 when fail(s) is 0, and return 1 when pass(s) is 1.

The figure below shows our implementation of the Tarantula debugger.

```
def cal_susp(data_list):
    susp_list = []
    for data in data_list:
        total_passed = data[0] + data[1]
        total_failed = data[2] + data[3]
        passed = data[0]
        failed = data[2]
        if failed == 0:
            susp_list.append(0)
            continue
        if passed == 0:
            susp_list.append(1)
            continue
        under = (passed/total_passed) + (failed/total_failed)
        #print(under)
        hue = (passed/total_passed)/ under
        susp = 1 - hue
        susp_list.append(susp)
    return susp_list

l = []
size = 0
for lines in sys.stdin:
    line = lines.split()
    size = len(line) - 1
    l.append(line)
susp_list = cal_susp(cal_datalist(l,size))
rank = sorted(range(len(susp_list)), key=lambda k: susp_list[k])
rank = [x+1 for x in rank]
print(rank[::-1])
```

3.2. CrossTab-Based Statistical Method:

Our implementation of this method mainly follows the instruction of the paper. However, it is possible that the number of cases passed or failed is 0. Such position can cause “divided by 0” problem since a divider should not be 0. To solve this problem, we check whether a divider is 0. If so, it will be replaced by 1. Such modification can make it calculable.

	s1 is covered	s1 is not covered	∑
successful executions	16	11	27
failed executions	0	0	0

∑	16	11	27

The table above gives an example. If a line of a program has no failed cases, it will be (0/0)/(16/27) when calculating φ(ω), which is not calculable. However, with our modification, it will be (0/1)/(16/27)=0, which is calculable.

The figure below shows the modification part of our implementation of CrossTab method.

```
for (i=0;i<size;i++){
    ecf=Math.max((coverFail[i]+notCoverFail[i])*(coverFail[i]+coverSuccess[i])/(total),1/total);
    ecs=Math.max((coverSuccess[i]+notCoverSuccess[i])*(coverFail[i]+coverSuccess[i])/(total),1/total);
    euf=Math.max((coverFail[i]+notCoverFail[i])*(notCoverFail[i]+notCoverSuccess[i])/(total),1/total);
    eus=Math.max((coverSuccess[i]+notCoverSuccess[i])*(notCoverFail[i]+notCoverSuccess[i])/(total),1/total);
    chiSquare[i]=Math.pow((coverFail[i]-ecf),2)/ecf+Math.pow((coverSuccess[i]-ecs),2)/ecs+
            Math.pow((notCoverFail[i]-euf),2)/euf+Math.pow((notCoverSuccess[i]-eus),2)/eus;
    mValue[i]=chiSquare[i]/total;
    if(coverSuccess[i]==0){
        coverSuccess[i]++;
    }
    if(coverFail[i]+notCoverFail[i]==0){
        notCoverFail[i]++;
    }
    phiValue[i]=coverFail[i]*(coverSuccess[i]+notCoverSuccess[i])/coverSuccess[i]/(coverFail[i]+notCoverFail[i]);
    zetaValue[i]=phiValue[i]>=1?mValue[i]:-mValue[i];
}
```

4. Performance of Debuggers

We compared the effectiveness between these two debuggers by comparing the ranking percentage of buggy line in the output of suspiciousness rank. That is, how many percent of innocent lines of a program is needed to check before finding the buggy line. For example, in a program with 92 lines of code, the rank of buggy line is 2, then the percentage is (2-1)/92=1.087%. The table below shows the performance of these two methods.

	Tarantula	CrossTab
0-2.5%	8	11
2.5-5%	4	2
5-20%	1	6
20-40%	1	1
40-65%	5	0
65-100%	1	0

Detailed data are shown below:

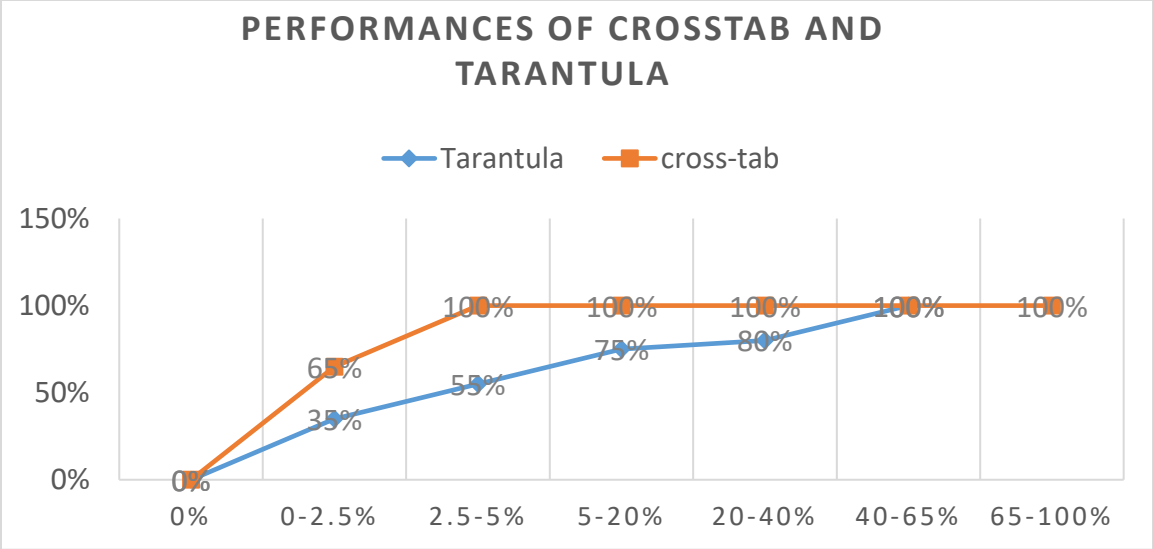
Sorting programs (92 lines)

Tarantula	CrossTab
3.261%	3.261%
1.087%	1.087%
15.217%	15.217%
3.261%	1.087%
1.087%	1.087%
3.261%	1.087%
2.174%	2.174%
1.087%	1.087%
3.261%	1.087%
2.174%	3.261%

Encrypt-decrypt programs (128 lines)

Tarantula	CrossTab
0.78125%	0.78125%
65.625%	2.34375%
1.5625%	0.78125%
0.78125%	0.78125%
61.71875%	9.375%
60.9375%	13.28125%
61.71875%	14.84375%
60.15625%	16.40625%
60.15625%	17.1875%
35.15625%	28.125%

The graph below show the cumulative curve of two methods:



5.Conclusion

CrossTab has a better performance in effectiveness compared with Tarantula. It is relatively reliable and able to find bugs in high rank (top 5 %) statements.

6.Job Division

Zheng Hongyi 13104036D – Coverage information, CrossTab

Chen Yunkun 13103214D – Testing programs, bugs, test cases

Li Dawei 13103693D – Tarantula, Result analysis