README

目录结构

- README.md 说明文件
- profile_v2.sh 收集单个进度
- run_all_inst.sh 运行实验2插桩
- run_all_overhead.sh 运行实验2的RQ3
- start.sh 初始化实验环境
- exprtment1.sh 运行实验1单个
- experiment1_all_all.sh 运行实验一全部
- check_results.rb 校验实验进度
- analyze1.sh 分析结果
- outputs/out/ 分析结果的输出
- defects4j/ 数据集
- tools/ 相关工具

实验说明

实验 1

- 分步插桩
- 收集的数据包括 实验2
- 包括四个 RQ
- RQ1 和 RQ2 先收集同一组 profile 数据，分别不同方法模拟和分析
- RQ3 分析运行时间
- RQ4 对 space 项目比较不同方法 对原实验修改
- 数据集换成了 defects4j，需要运行脚本整理一下，以及对大型项目做一些性能优化。
- CBI实验中，怀疑度的计算公式换成了 O

实验结果

分析汇总的文件，运行输出的原始数据

- 实验1的结果 lab/log.txt
- 实验2 RQ1 ..._1_10_v8-v8_m.xlsx
- 实验2 RQ2 ..._1_10_v1-v7_m.xlsx
- 实验2 RQ3 ..._overhead.xlsx
- 实验2 RQ4 space_correlation.xlsx

实验环境

实验1 依赖 mps 实验2 依赖 java7

实验步骤

实验1

- 执行 `experment1_run_all.sh`

- 收集到数据的参数有：项目编号，k=1,5,10
- 收集到数据结果，

实验2

- RQ1
  - 执行 `bash run_all_inst.sh` 收集 profile
  - 执行 `bash analyze1.sh` 进行模拟和分析
  - 得到对应的 Excel 文件
- RQ2
  - 需要先执行 RQ1 以得到 profile
  - 执行 `bash analyze1.sh _return` 进行模拟和分析
  - 执行 `bash analyze1.sh _scalar` 进行模拟和分析
  - 执行 `bash analyze1.sh _branch` 进行模拟和分析
  - 分别的得到对应的 Excel 文件
- RQ3
  - 需要先执行 RQ1 以得到分析结果
  - 然后执行 `bash run_all_overhead.sh` 得到不同插桩方式下测试用例的执行时间
    - 这里每次测试用例执行六遍，且只包含测试用例执行时间
  - 然后执行 `java -cp tools/HI.jar zuo.util.readfile.IterativeTimeReader ~/oopsla artifacts/single/Subjects/Java/ nanoxml ~/oopsla artifacts/single/Console/`
  - 得到汇总的 Excel 文件
- RQ4
  - 执行 `bash run_all_inst.sh` 后
  - 执行 `java -cp HI.jar zuo.processor.functionentry.client.iterative.Client_Ranking ~/oopsla_artifacts/single/Subjects/C/ space ~/oopsla_artifacts/single/Console/`
  - 这里因为修改了 HI 中的 importance 的公式，结果的 Excel 文件会和原先有差异。

代码的修改

JSampler

目的是性能优化，不然对真实大型项目会跑不起来。

1. 运行多个测试，不用反复重启 jvm ，所以添加了初始化（清空）统计信息的代码。提升测试执行时间明显。
2. 原来两轮对代码分析是独立的，字节码和中间代码的转换共进行四次，可以减少到两次。减少soot执行时间。

PredicateCheckerReporter

```
public static synchronized void reset(String name) {
    return_reports.clear();
    branch_reports.clear();
    scalarPair_reports.clear();
    methodEntry_reports.clear();
    output_file_reports = name;
}
```

PCounter，PInst 使用同一组中间代码，可以合并

```
//
```

HI

输出列表

JavaClient

```
printOutMethodsListByMode();//去掉注释，用于输出分析结果
```

修改了怀疑度公式。

```
//zuo.processor.functionentry.client.iterative.java.JavaClient
//104;
return Pattern.matches("subv[0-9]*", name) && new File(dir,
name).listFiles().length >= 1;//把检查文件数去掉
```

```
// package zuo.processor.cbi.client.CBIClient;
// private void increasePartialSamples(Set<Integer> failingSet, Set<Integer>
passingSet, double percent) {
        while (failingSet.size() < (fs > 2 ? fs : 2)) {
//失败测试用例个数最小值去掉，修改 fs，这个限制是原先怀疑度公式包含的。
```

```
//zuo.processor.functionentry.processor.SelectingProcessor;
public int computeCBIBound(double threshold) {
    if (DH(2, this.totalPositive) <= 0.0D){
      throw new RuntimeException("abnormal case 1");//把这行注释掉，检查测试用例数是否
大于等于 2
    }
```

package zuo.processor.cbi.processor.Processor;

原始代码

```
  public static double importance(int neg_t, int pos_t, int neg, int pos, int
totalNeg, int totalPos) {
    assert neg_t <= neg && pos_t <= pos;
    if (neg_t <= 1 || pos_t + neg_t == 0)
      return 0.0D;
    double increase = neg_t / (pos_t + neg_t) - neg / (pos + neg);
    if (increase < 0.0D || Math.abs(increase - 0.0D) < 1.0E-7D)
      return 0.0D;
    return 2.0D / (1.0D / increase + Math.log(totalNeg) / Math.log(neg_t));
  }
```

修改为 Ochiai

```
  public static double importance(int neg_t, int pos_t, int neg, int pos, int
totalNeg, int totalPos) {
    assert neg_t <= neg && pos_t <= pos;
  if (neg == 0 || pos_t + neg_t == 0)
    return 0;
  return neg_t/Math.sqrt(neg*(pos_t + neg_t));
  }
```

```
    public static double importance(int neg_t, int pos_t, int neg, int pos, int
totalNeg, int totalPos) {
        assert neg_t <= neg && pos_t <= pos;
        return neg != 0 && pos + neg != 0 ? (double)neg / Math.sqrt((double)
(totalNeg * (pos + neg))) : 0.0D;
    }
```

修改了终止条件

zuo.processor.functionentry.client.iterative.IterativeFunctionClient#printPruneCase

```
if(value.getC_score()==0)skip=true;
```

因为

- 会涉及大量无关函数，会超时。
- 修改可能会影响 average 指标

mps

修改了输出文件名称，源于samlercc和jsampler的命名约定差异

defects4j

defects4j 自身用 ant 来处理 junit3 和 junit4 的版本问题

- formatter 在执行单元测试时被调用，正好用来收集和处理一些信息。
- ，这个由ant作为任务依赖会判断是否需要再次调用。

formatter.jar

```
  public void startTest(Test test) {
    this.allTests.println(test.toString());
    PredicateCheckerReporter.reset(System.getenv("SAMPLER_FILE") +
test.toString());
  }
```

每个测试用例单独汇总

defects4j/framework/bin/d4j/d4j-test

```
# $project->compile_tests()
```

注释掉，这个ant会自动调用，以控制执行时间。

DRAFT

- outOfRange case 3 会不会影响结果正确性，应该是前怀疑度的校验代码
- SelectPro 里的 是作为对照实验的
- 现在运行时间是输出到 time 文件，注意 RQ1和RQ3 的衡量标准不一样。
- RQ3输出不含profile的时间，目前是重复执行四次算后三次
- 整理分析的输出