Dirt Spot Sweeping Random Strategy

Mian Asbat Ahmad
Department of Computer Science
The University of York
York, United Kingdom
mian.ahmad@york.ac.uk

Manuel Oriol
Department of Computer Science
The University of York
York, United Kingdom
manuel.oriol@york.ac.uk

ABSTRACT

This paper presents an enhanced and improved form of automated random testing, called the Dirt Spot Sweeping Random (DSSR) strategy. DSSR is a new strategy that takes the assumption that a number of failure domains are contiguous. DSSR starts as a regular random+ testing session — a random testing session with some preference for boundary values. When a failure is found, it increases the chances of using neighbouring values in subsequent tests, thus slowly sweeping around the failure found in hope of finding failures from a different kind in its vicinity.

DSSR was implemented within the YETI random testing tool. We evaluate DSSR against random+ and pure random strategies by testing 60 classes with one million (10^5) calls for each session 30 times for each strategy. We found that for 70% of the classes all three strategies find the same unique failures, for remaining 30% DSSR perform better (up to as much as 33%) then random strategy and (up to as much as 16%) then random+ strategy. Overall, DSSR also found 78 more unique failures than random and 11 more unique failures than random+.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Comparison, Verification,

Keywords

software testing, automated random testing, comparative analysis

1. INTRODUCTION

Success of a software testing technique is mainly based on the number of faults it discovers in the Software Under Test (SUT). An efficient testing process discovers the maximum number of faults in a minimum possible amount of time. Exhaustive testing, where software is tested against all possible inputs, is in most cases not feasible because of the size of the input domain, limited resources and strict time constraints. Therefore, strategies in automated software testing tools are developed with the aim to select more fault-finding test input from the input domain for a given SUT. Producing such targeted test input is difficult because each system has its own requirements and functionality.

Chan et al. [3] discovered that there are patterns of failure-causing inputs across the input domain. They divided the patterns into point, block and strip patterns on the basis of their occurrence across the input domain. Chen et al. [7] found that the performance of random testing can be increased by slightly altering the technique of test case selection. In adaptive random testing, they found that the performance of random testing increases by up to 50% when test input is selected evenly across the whole input domain. This was mainly attributed to the better distribution of input which increased the chance of selecting inputs from failure patterns. Similarly Restricted Random Testing [4], Feedback directed Random Test Generation [37], Mirror Adaptive Random Testing [9] and Quasi Random Testing [11] also stress the need for test case selection covering the whole input domain to improve results.

In this paper we take the assumption that for a significant number of classes failure domains are contiguous or are very close by. From this assumption, we devised the Dirt Spot Sweeping[1] Random strategy (DSSR) which starts as a random+ strategy — a random strategy focusing more on boundary values. When it finds a new failure, it then increases the chances of finding more faults using neighbouring values. Note that, similarly to previous studies [32] we approximate faults with unique failures. Of course, since this strategy is also a random testing strategy, it has the potential to find all unique failures in the program, but we expect it to be faster at finding unique failures for classes in which failure domains are contiguous than the pure random (R) and random+ (R+) strategies.

We implemented DSSR as a strategy for the random testing tool YETI[2]. To evaluate our approach, we tested thirty

[1]The name refers to the cleaning robots strategy which insists on places where dirt has been found in large amount.
[2]`http://www.yetitest.org`

times each one of 60 classes from the Qualitas Corpus[3] with each of the three strategies DSSR, R, and R+. We found that for 70% of the classes all three strategies find the same unique failures, for remaining 30% DSSR perform better (up to as much as 33%) then random strategy and (up to as much as 16%) then random+ strategy. Overall, DSSR also found 78 more unique failures than random and 11 more unique failures than random+.

The rest of this paper is organised as follows: Section 2 describes the DSSR strategy. Section 3 presents our implementation of the strategy. Section 4 explains our experimental setup. Section 5 shows the results of our experiments. Section 6 discusses the results. Section 7 presents related work and we conclude in Section 8.

2. DIRT SPOT SWEEPING RANDOM STRATEGY

The Dirt Spot Sweeping Random strategy (DSSR) is a new strategy which combines the random+ strategy with a dirt spot sweeping functionality. The strategy is based on two intuitions. First, boundaries have interesting values and using these values in isolation can provide high impact on test results. Second, faults and unique failures reside in contiguous blocks and stripes. If this is the case, DSSR increases the performance of the test strategy. Each strategy is briefly explained as follows.

2.1 Random Strategy (R)

The pure random strategy is a black-box testing technique in which the SUT is executed using randomly selected test data. Test results obtained are compared to the defined oracle, using SUT specifications in the form of contracts or assertions. In the absence of contracts and assertions the exceptions defined by the programming language are used as test oracles.

Because of its black-box testing nature, this strategy is particularly effective in testing softwares where the developers want to keep the source code secret [10]. The generation of random test data is comparatively cheap and does not require too much intellectual and computation efforts [16, 14]. It is mainly for this reason that various researchers have recommended this strategy for automatic testing tools [13]. YETI [33, 34], AutoTest [25, 12], QuickCheck [17], Randoop [36], Jartage [30] are some of the most common automated testing tools based on random strategy.

Efficiency of random testing was made suspicious with the intuitive statement of Myers [28] who termed random testing as one of the poorest methods for software testing. However, experiments performed by various researchers, [12, 20, 21, 24, 29] have experimentally proven that random testing is simple to implement, cost effective, highly efficient and free from human bias as compared to its rival techniques.

Because programs tested at random typically fail a large number of times (there are a large number of calls), it is necessary to cluster failures that likely represent the same fault. The traditional way of doing it is to compare the full stack traces and error types and use this as an equivalence class [12, 32] called a unique failure. This way of grouping failures is also used for random+ and DSSR.

[3] `http://www.qualitascorpus.com`

2.2 Random Plus Strategy (R+)

The random+ strategy [25] is an extension of the pure random strategy. It uses some special pre-defined values which can be simple boundary values or values that have high tendency of finding faults in the SUT. Boundary values [2] are the values on the start and end of a particular type. For instance, such values for `int` could be `MAX_INT`, `MAX_INT-1`, `MIN_INT`, `MIN_INT+1`, `-1`, `0`, `1` etc.

Similarly, the tester might also add some other special values that he considers effective in finding faults for the current SUT. For example, if a program under test has a loop from -50 to 50 then the tester can add -55 to -45, -5 to 5 and 45 to 55 to the pre-defined list of special values in order to be selected for a test. This static list of interesting values is manually updated before the start of the test and has slightly high priority than selection of random values because of more relevance and high chances of finding faults for the given SUT. These special values have high impact on the results particularly detecting problems in specifications [14].

2.3 Dirt Spot Sweeping

Chan et al. [3] found that there are patterns of failure-causing inputs across the input domain. Figure 1 shows these patterns for two dimensional input domain. They divided these patterns into three types called points, block and strip patterns. The black area (Points, block and strip) inside the box show the input which causes the system to fail while white area inside the box represent the genuine input. Boundary of the box (black solid line) surrounds the complete input domain and also represents the boundary values. They also argue that a strategy has more chances of hitting these fault patterns if test cases far away from each other are selected. Other researchers [4, 9, 11], also tried to generate test cases further away from one another targeting these patterns and achieved higher performance. High performance of these strategies confirm that faults more often occur contiguous across the input domain. In Dirt Spot Sweeping we propose that if test case reveal a fault then for the next test case it should not look farthest away from itself because it is in the fault region instead it should pick the closest test case to find another fault from the same region.

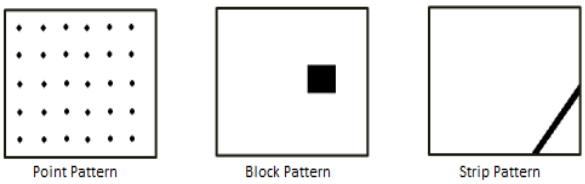

Figure 1: Failure patterns across input domain [7]

Dirt spot sweeping is the part of DSSR strategy that comes into action when a failure is found in the system. On finding a failure, it immediately adds the value causing the failure and its neighbouring values to the already existing list of interesting values. For example in a program if the `int` type value 50 causes a failure in the system then spot sweeping

will add values from 47 to 53 to the list of interesting values. If the failure lies in the block or strip pattern, then adding its neighbours will explore other failures present in that block or strip. As against random plus where the list of interesting values remain static, the list of interesting values is dynamic and changes during the test execution of each program in the DSSR strategy.

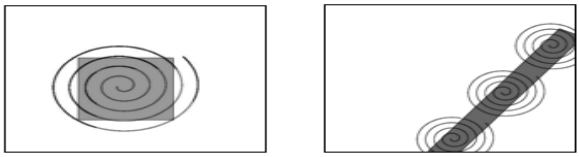

Figure 2: DSSR covering block and strip pattern

Figure 2 shows how dirt spot sweeping explores the failures residing in the block and strip patterns of a program. The failure coverage from the pattern is shown in spiral form because first failure will lead to second, second to third and so on till the end. In case the failure is positioned on the point pattern then the added values will not be very effective because point pattern is only an arbitrary failure point in the whole input domain.

2.4 Structure of the Dirt Spot Sweeping Random Strategy

The DSSR strategy is explained with the help of a flowchart in Figure 3. In this process, the strategy continuously tracks the number of failures during the execution of the test session. To keep the system fast this tracking is done in a very effective way with zero or minimum overhead [26]. The execution of test is performed normally until a failure is found in the SUT. Then the program does not only copy the values that lead to the failure, but also copies its surrounding values to the variable list of interesting values. As presented in the flowchart, if the failure finding value is of primitive type then the DSSR finds the type of the value and add values only of that particular type to the interesting values. Addition of these values increases the size of the list of interesting values that provide relevant test data for the remaining test session and the new generated test cases are more targeted towards finding new failures in the given SUT around pre-existing failures.

Boundary values and other special values that have a high tendency of finding faults in the SUT are added to the list by random plus strategy prior to the start of test session where as to sweep the failure pattern, the fault-finding value and its surrounding values are added at runtime after a failure is found. Table 1 presents the values that are added to the list of interesting values when a failure is found. In the table the test value is represented by X where X can be int, double, float, long, byte, short, char and String. All values are converted to their respective types before adding to the list of interesting values and vice versa.

2.5 Explanation of DSSR on a concrete example

The DSSR strategy is explained through a simple program seeded with at least three faults. The first fault is a division

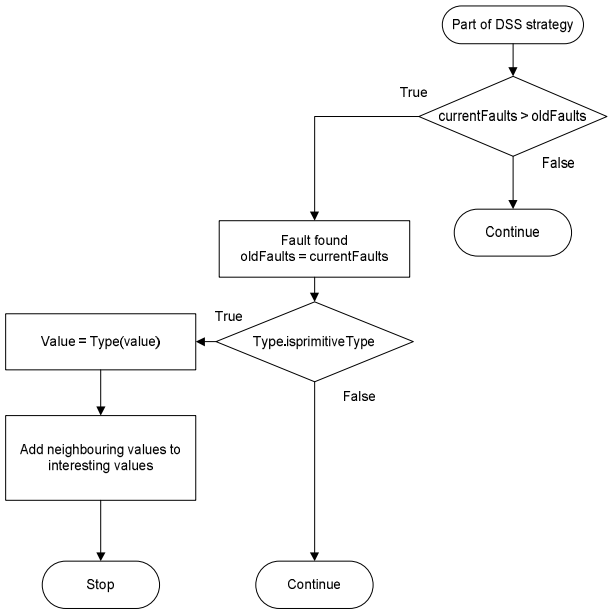

Figure 3: Working mechanism of DSSR Strategy

Type	Values to be added
X is int, double, float, long, byte, short & char	X, X+1, X+2, X-1, X-2
X is String	X X + " " " " + X X.toUpperCase() X.toLowerCase() X.trim() X.substring(2) X.substring(1, X.length()-1)

Table 1: Neighbouring values for primitive types and String

by zero exception denoted 1 while the second and third are failing assertion statements denoted 2 and 3 in the following program. Below we describe how the DSSR strategy perform execution when the following class is expose to testing.

```
/**
* Calculate square of given number
* and verify results.
* The code contain 3 faults.
* @author (Mian and Manuel)
*/
public class Math1 {
 public void calc (int num1) {
  // Square num1 and store result.
  int result1 = num1 * num1;
  int result2 = result1 / num1; // 1
  assert Math.sqrt(result1) == num1; // 2
  assert result1 >= num1; // 3
 }
}
```

In the above code, one primitive variable of type `int` is used, therefore, the input domain for DSSR strategy is from `-2,147,483,648 to 2,147,483,647`. The strategy further selects some values (`0, Integer.MIN\_VALUE` and `Integer.MAX\_VALUE`) as interesting values which are prioritised for selection as inputs. As the test starts, three faults are quickly discovered by DSSR strategy in the following order.

Fault 1: The DSSR strategy might select value `0` for variable `num1` in the first test case because `0` is available in the list of interesting values and therefore its priority is higher than other values. This will cause Java to generate division by zero exception.

Fault 2: After catching the first fault, the strategy adds it and its surrounding values to the list of interesting values which includes `0, 1, 2, 3 and -1, -2, -3` in this case. In the second test case DSSR strategy may pick `-3` as a test value and lead to the second fault where assertion (2) fails because the square root of `9` will be `3` instead of the input value -3.

Fault 3: After few tests DSSR strategy may select `Integer.MAX\_VALUE` for variable `num1` from the list of interesting values which will lead to the 3rd fault because `result1` will not be able to store the square of `Integer.MAX\_VALUE`. Instead of the actual square value Java assigns a negative value (Java language rule) to variable result1 that will lead to the violation of the next assertion (3).

The process above explains that the pre-defined values including border values, fault-finding values and the surrounding values lead to the available faults quickly and in small number of tests as compared to random and random+ strategy. Random and random+ takes longer to discover the second and third fault because they start again searching for new unique failures randomly although the remaining faults are very close to the first one.

3. IMPLEMENTATION OF THE DSSR STRATEGY

As mentioned previously, the implementation of the DSSR strategy is made in the YETI open-source automated random testing tool. YETI is developed in Java and capable of testing systems developed in procedural, functional and object-oriented languages. Its language-agnostic meta model enables it to test programs written in multiple languages including Java, C#, JML and .Net. The core features of YETI include easy extensibility for future growth, speed of up to one million calls per minute on java code, real time logging, real time GUI support, ability to test programs using multiple strategies, and auto generation of test report at the end of the testing sessions. For large-scale testing there is a cloud-enabled version of YETI that is capable of executing parallel test sessions in Cloud [34]. A number of hitherto faults have successfully been found by YETI in various production softwares [32, 31].

YETI can be divided into three decoupled main parts: the core infrastructure, language-specific bindings and strategies. The core infrastructure contains representation for routines, a group of types and a pool of specific type objects. The language specific bindings contain the code to make the call and process the results. The strategies section defines the procedure of how to select the modules (classes) from the project, how to select routines (methods) from these modules and how to generate values for the instances involved inside these routines. The most common strategies are random and random+.

By default, YETI uses the random+ strategy if no particular strategy is defined during test initialisation. It also enables the user to control the probability of using null values and the percentage of newly created objects for each test session. YETI provides an interactive Graphical User Interface (GUI) in which users can see the progress of the current test in real time. In addition to the GUI, YETI also provides extensive logs of the test session for more in-depth analysis.

The DSSR strategy has then been added as an extension of YetiRandomStrategy, which in itself is an extension of an abstract class YetiStrategy. The class hierarchy is shown in Figure 4.

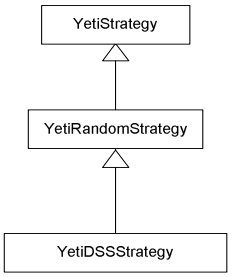

Figure 4: Class Hierarchy of DSSR in YETI

4. EVALUATION

To evaluate the DSSR strategy, we compare its performances to the performances of both pure random testing (R) and the random+ (R+) [25] strategy. General factors such as system software and hardware as well as the YETI specific factors like percentage of null values, percentage of newly created objects and interesting value injection probability have the same values for the experiments.

4.1 Research questions

To evaluate the DSSR strategy and its usefulness, we set out to answer the following research questions:

1. Is any of the three strategies R, R+ and DSSR provide better results than the other two?
2. Is there a subset of the classes for which R, R+, or DSSR provide better results than the other two?
3. If such categories exist, what are their sizes and how do they compare and can we pick a default strategy according to this criterion?

4.2 Experiments

To evaluate the performances of DSSR we performed extensive testing of programs from the Qualitas Corpus [40]. The

Qualitas Corpus is a curated collection of open source java projects built with the aim of helping empirical research on software engineering. These projects are collected in an organised form containing both the source and binary forms. The present evaluation uses version 20101126 which contains 106 open source java projects. We picked 32 projects at random and picked 80 classes at random that produced at least 1 failure and did not timeout with a testing session of maximum 10 minutes. We tested each of the 80 classes thirty times with each strategy. Total number of code lines in selected 80 classes are 40,213. Individual lines of code per class under test is given in column 3 of table 3. Names and versions of the projects to which these classes belong are given in table 2.

Each class is evaluated through 10^5 calls in each testing session.[4] Because of the absence of the contracts and assertions in the code under test, similarly to previous approaches [32], we use undeclared exceptions to compute unique failures found.

ant-1.8.1
antlr-3.2
aoi-2.8.1
argouml-0.30.2
artofillusion281
aspectj-1.6.9
axion-1.0-M2,
azureus
castor-1.3.1
cayenne-3.0.1
cobertura-1.9.4.1
colt-1.2.0
emma-2.0.5312
freecs-1.3.20100406
hibernate-3.6.0
hsqldb-2.0.0
itext-5.0.3
jasml-0.10
jmoney-0.4.4
jruby-1.5.2
jsXe-04_beta
quartz1.8.3
sandmark3.4
squirrel-sql-3.1.2
tapestry-5.1.0.5
tomcat-7.0.2
trove-2.1.0
velocity-1.6.4
weka-3.7.2
xalan-2.7.1
xerces-2.10.0
xmojo-5.0.0

Table 2: Name and versions of 32 Projects randomly selected from the Qualitas Corpus for the experiments

All tests are performed using a 64-bit Mac OS X Lion Version 10.7.4 running on 2 x 2.66 GHz 6-Core Intel Xeon with 6.00 GB (1333 MHz DDR3) of RAM. YETI runs on top of the Java™SE Runtime Environment [version 1.6.0_35]. The machine took approximately 100 hours to process the experimental data.

[4]The total number of tests is thus $80 \times 30 \times 3 \times 10^5 = 720 \times 10^6$ *tests*.

4.3 Performance measurement criteria

Various measures including the E-measure, P-measure and F-measure have been used by researchers to find the effectiveness of the random test strategy. The E-measure (expected number of failures detected) and P-measure (probability of detecting at least one failure) were heavily criticized [7] and are not considered effective techniques for measuring efficiency of test strategy. The F-measure (number of test cases used to find the first fault) has been often used by researchers [6, 8]. In our initial experiments the F-measure was used to evaluate the efficiency. Soon after a few experiments, it was realised that this was not the right choice because in some experiments the first strategy found the first fault quickly than the second strategy but on the completion of test session the first strategy found lower number of total faults than the second strategy. The preference to a strategy only because it found the first fault better without giving due consideration to the total number of faults was not fair [27].

The literature review revealed that the F-measure is used where testing stops after identification of the first fault and the system is given back to the developers to remove the fault found. In such cases it make sense to use F-measure but nowadays automated testing tools test the whole system and print all of the faults found in one go therefore F-measure is not the favourable choice. Therefore in our experiments, performance of the strategy was measured by the maximum number of faults in a particular number of test calls [37], [12], [15]. This measurement was found effective because it clearly measured the performance of the strategy when all the other factors were kept constant.

5. RESULTS

5.1 Is there an absolute best for DSSR, R+ and R?

Figure 5 presents the results of 80 randomly selected classes evaluated by the R, R+ and DSSR strategies in an intuitive normalised stackbar representation where projects are ranked according to the relative number of unique failures found by DSSR. As a first visual interpretation, it seems that, except in rare cases, all strategies find significantly the same number of uniques failures.

Table 3 contains more detailed information: name of the classes, mean value, maximum number of unique failures, minimum number of unique failures and relative standard deviation for each of the 80 classes tested using R, R+ and DSSR strategy. The total value (table 3 last row) shows DSSR detects slightly more unique failures (1192.55), on average, than R (1165.53) and R+ (1188.73). This represents 2.3% on average than R and .3% more than R+. It also shows that DSSR found a higher number of maximum unique failures (1234) and minimum unique failures (1126) than R (1181), (1055) and R+ (1224), (1127) respectively. This represents: 4.5% improvements over R and .8% over R+ for the maximum and 6.7% improvement over R and .1% decrease over R+ for the minimum. Eventually, the

S. No	Class Name	LOC	R				R+				DSSR			
			mean	max	min	rel std dev	mean	max	min	rel std dev	mean	max	min	rel std dev
1	ActionTranslator	709	96	96	96	0	96	96	96	0	96	96	96	0
2	AjTypeImpl	1180	80	83	79	0.03	80	83	79	0.03	80	83	79	0.03
3	Apriori	292	3	4	3	0.23	3	4	3	0.23	3	4	3	0.23
4	BitSet	575	9	9	9	0	9	9	9	0	9	9	9	0
5	CatalogManager	538	7	7	7	0	7	7	7	0	7	7	7	0
6	CheckAssociator	351	7	8	2	0.60	6	9	2	0.82	7	9	2	0.70
7	Debug	836	4	6	4	0.35	5	6	4	0.28	5	8	4	0.56
8	DirectoryScanner	1714	33	39	0	0.83	35	38	31	0.14	36	39	32	0.13
9	DiskIO	220	4	4	4	0	4	4	4	0	4	4	4	0
10	DOMParser	92	7	7	3	0.70	7	7	6	0.10	7	7	7	0
11	Entities	328	3	3	3	0	3	3	3	0	3	3	3	0
12	EntryDecoder	675	8	9	7	0.17	8	9	7	0.17	8	9	7	0.17
13	EntryComparator	163	13	13	13	0	13	13	13	0	13	13	13	0
14	Entry	37	6	6	6	0	6	6	6	0	6	6	6	0
15	Facade	3301	3	3	3	0	3	3	3	0	3	3	3	0
16	FileUtil	83	1	1	1	0	1	1	1	0	1	1	1	0
17	Font	184	12	12	11	0.05	12	12	11	0.05	12	12	11	0.05
18	FPGrowth	435	5	5	5	0	5	5	5	0	5	5	5	0
19	Generator	218	17	17	17	0	17	17	17	0	17	17	17	0
20	Group	88	11	11	11	0	10	4	11	0.49	11	11	11	0
21	HttpAuth	221	2	2	2	0	2	2	2	0	2	2	2	0
22	Image	2146	14	18	7	0.55	12	14	4	0.58	14	15	11	0.20
23	InstrumentTask	71	2	2	1	0.35	2	2	1	0.35	2	2	2	0
24	IntStack	313	4	4	4	0	4	4	4	0	4	4	4	0
25	ItemSet	234	4	4	4	0	4	4	4	0	4	4	4	0
26	JavaWrapper	513	3	2	2	0	4	4	3	0.17	4	4	3	0.17
27	JmxUtilities	645	8	8	6	0.17	8	8	7	0.08	8	8	7	0.08
28	List	1718	5	6	4	0.28	6	6	4	0.23	6	6	5	0.11
29	NameEntry	172	4	4	4	0	4	4	4	0	4	4	4	0
30	NodeSequence	68	38	46	30	0.29	36	45	30	0.29	38	45	30	0.27
31	NodeSet	208	28	29	26	0.07	28	29	26	0.05	28	29	26	0.07
32	PersistentBag	571	68	68	68	0	68	68	68	0	68	68	68	0
33	PersistentList	602	65	65	65	0	65	65	65	0	65	65	65	0
34	PersistentSet	162	36	36	36	0	36	36	36	0	36	36	36	0
35	Project	470	65	71	60	0.11	66	78	62	0.17	69	78	64	0.14
36	Repository	63	31	31	31	0	40	40	40	0	40	40	40	0
37	Routine	1069	7	7	7	0	7	7	7	0	7	7	7	0
38	RubyBigDecimal	1564	4	4	4	0	4	4	4	0	4	4	4	0
39	Scanner	94	3	5	2	0.70	3	5	2	0.70	3	5	2	0.70
40	Scene	1603	26	27	26	0.02	26	27	26	0.02	27	27	26	0.02
41	SelectionManager	431	3	3	3	0	3	3	3	0	3	3	3	0
42	Server	279	15	21	11	0.47	17	21	12	0.37	17	21	12	0.37
43	Sorter	47	2	2	1	0.35	3	3	3	0	3	3	3	0
44	Sorting	762	3	3	3	0	3	3	3	0	3	3	3	0
45	Statistics	491	15	17	12	0.23	23	25	22	0.09	24	25	22	0.08
46	Status	32	53	53	53	0	53	53	53	0	53	53	53	0
47	Stopwords	332	7	8	7	0.10	7	8	6	0.20	8	8	7	0.08
48	StringHelper	178	43	45	41	0.06	44	46	42	0.06	44	45	42	0.04
49	StringUtils	119	19	19	19	0	19	19	19	0	19	19	19	0
50	TouchCollector	222	3	3	3	0	3	3	3	0	3	3	3	0
51	Trie	460	21	22	21	0.03	21	22	21	0.03	21	22	21	0.03
52	URI	3970	5	5	5	0	5	5	5	0	5	5	5	0
53	Itextpdf	245	8	8	8	0	8	8	8	0	8	8	8	0
54	WebMacro	311	5	5	5	0	5	6	5	0.14	5	7	5	0.28
55	XMLAttributesImpl	277	8	8	8	0	8	8	8	0	8	8	8	0
56	XMLChar	1031	13	13	13	0	13	13	13	0	13	13	13	0
57	XMLEntityManger	763	17	18	17	0.04	17	17	16	0.04	17	17	17	0
58	XMLEntityScanner	445	12	12	12	0	12	12	12	0	12	12	12	0
59	XObject	318	19	19	19	0	19	19	19	0	19	19	19	0
60	XString	546	24	24	23	0.02	23	24	23	0.03	24	24	23	0.02
	Total	40,213	1166	1217	1078	8.116	1187	1233	1137	6.877	1201	1244	1152	5.504

Table 3: Complete results for R, R+ and DSSR. Results present mean, max, min and relative standard deviation.

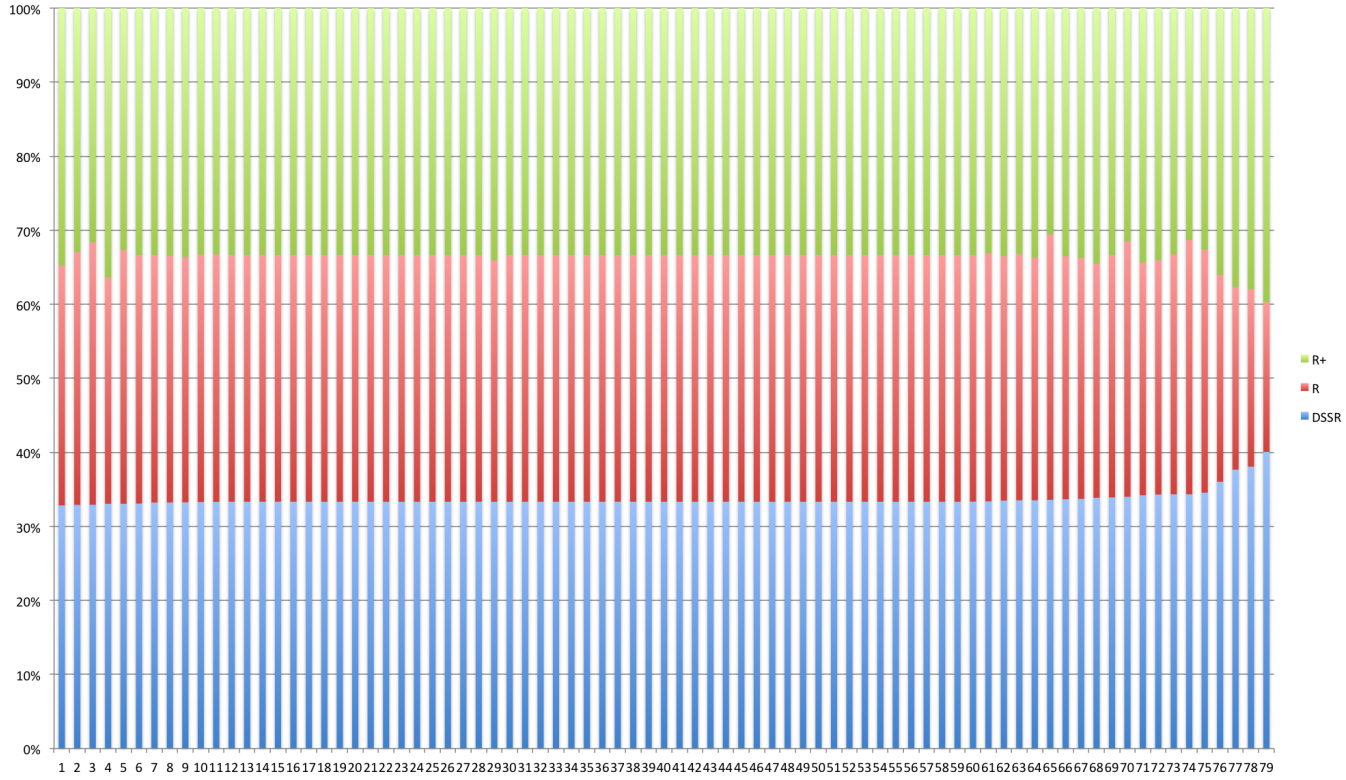

Figure 5: Normalised stacked bar diagram of all tested classes.

standard deviations are all of the order of magnitude of .1% for all strategies.

The answer to this research question is thus that whereas DSSR produces a slightly higher number of unique failures, this is not significantly higher than R+. We can thus say that R+ and DSSR are better choices than R as an absolute strategy, but that neither significantly outperforms the other.

5.2 Are there classes for which either one of the strategies provides better results?

Results can be split into six different categories as shown in figure 6. The first category is the largest where each strategy performed equally well and found the same number of unique failures after 10^5 tests. It contain 50 classes (62% of the experiments).

The second category contains 11 (14%) classes where R performed better than DSSR and R+.

The third category contains 7 classes where (9%) R+ performed better than DSSR and R.

The fourth category contain 7 classes (9%) where DSSR performed better than R and R+.

The fifth category contain only one class (1%) where both DSSR and R found an equal number of unique failures and performed better than R+.

The sixth category contain 4 classes (5%) where DSSR and R+ found an equal number of unique failures and performed better than R.

No class is found for which R performs equal to R+.

The answer to this research question is that most classes (62%) did not exhibit significantly different behaviours independent of the strategy. In 38% of the cases, though, one or two strategies work better than the other(s). In particular, 14% of the classes performed better with R, 9% of the classes performed better with R+ and 9% with DSSR. This shows that the assumptions made when developing R+ — some border values are more bug-prone — and DSSR — failure domains are connected — only verify in a minority of cases and are code-dependent.

5.3 Can we pick the best default strategy between R, R+ and DSSR?

With the data presented in this section, it is not possible to pick a best strategy for all classes. In most cases, results are not different from one strategy to another, but in the other cases, the best strategy is very much dependent on the tested code and none of R, R+ and DSSR are significantly better than the other two on larger classes.

In the next section we also discuss other factors that influence the outcome of such a question such as time taken by each strategy, effect of test duration, number of tests, number of faults and the effect of finding first by random

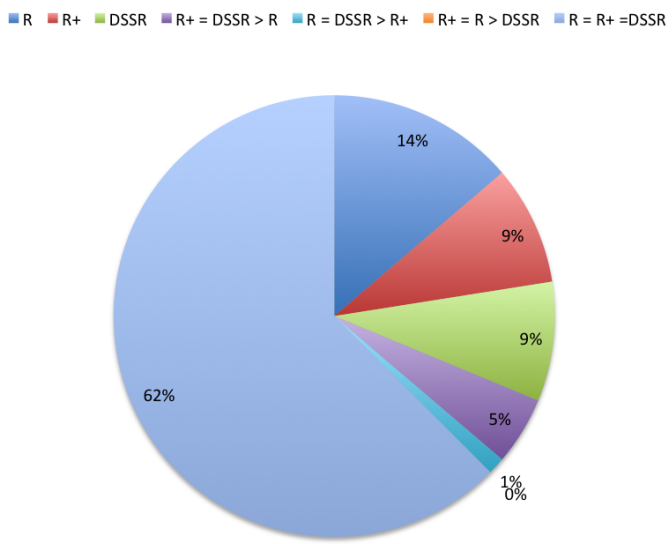

Figure 6: Result Categories.

strategy on DSSR performance.

6. DISCUSSION

Time taken by DSSR strategy, Random strategy and Random plus strategy to execute tests: To execute an equal number of test cases, DSSR takes slightly more time (between 5 and 10% overhead) than both pure random and random plus. This is due to maintaining sets of interesting values. The overhead is dependent on our implementation and could also be reduced if needed.

Effect of test duration and number of tests on the results: All three techniques have the same potential for finding bugs. If testing infinitely, all techniques should find the same number of unique failures. So the results will converge the longer (resp. the more tests) testing sessions contain. We suspect however that some of the unique failures found would be extremely long to find using random or random+ only. Further experiments should confirm this point.

Effect of number of faults on results: We found that the DSSR strategy performs better when the number of faults is higher in the code. The reason seems to be that when there are more faults, their domains are more connected and DSSR then works better. Further studies might use historical data to pick the best strategy.

DSSR strategy dependence on finding the first unique failures early enough: During the experiments we found that if the unique failures is not found quickly enough there is no value added to the list of interesting values and then the test is equivalent random testing. This means that better ways of populating failure-inducing values are needed to leverage DSSR better. As an example, the following piece of code would be unlikely to fail under the current setting:

```
public void test(float value){
 if(value == 34.4445) {
   10/0;
 }
}
```

In this case, we could add constant literals from the SUT to the list of interesting values in a dynamic fashion. These literals can be obtained from the constant pool in the class files of the SUT.

In the example above the value 34.4445 and its surrounding values would be added to the list of interesting values before the test starts and the DSSR strategy would find the unique failure right away.

DSSR strategy and coverage: Random strategies typically achieve high level of coverage [34]. It might also be interesting to compare R, R+ and DSSR with respect to the achieved coverage or even to use a DSSR variant that adds a new interesting value and its neighbors when a new branch is reached.

Threats to validity: As usual with such empirical studies, the present work might suffer from a non-representative selection of classes. The present selection was however made through random selection and objective criteria and it seems unlikely that they would not be representative.

The parameters of the study might also have prompted incorrect results. This is however unlikely due to previous results on random testing [32].

7. RELATED WORK

Random testing is a popular technique with simple algorithm but proven to find subtle faults in complex programs and Java libraries [35, 19, 18]. Its simplicity, ease of implementation and efficiency in generating test cases make it a best choice for test automation [24]. Few of the well

known automated tools based on random strategy includes Jartege [30], Eclat [35], JCrasher [19], AutoTest [12], [13] and YETI [34, 32] which was used to conduct this research study.

In pursuit of better results and lower overhead, many variations of random strategy have been proposed [10, 11, 4, 5, 9]. Adaptive random testing (ART), Quasi-random testing (QRT) and Restricted Random testing (RRT) achieved better results by selecting test inputs random but evenly spread across the input domain. Similarly Mirror ART and ART through dynamic partitioning increased the performances by reducing the overhead of ART. One of the main reason behind the better performance of these strategies is that even spread of test input increases the chance of exploring the fault patterns present in the input domain.

The random+ (R+) strategy [25] is a variation of the random strategy in which interesting values, beside pure random values, are added to the list of test inputs. These interesting values includes border values [2] which have high tendency of finding faults in the given SUT. Results conducted with R+ strategy show significant improvement of pure random strategy. DSSR strategy also rely on R+ strategy in the start until any fault is found where it switches to spot sweeping strategy.

It is interesting that numerous efforts have been made to discover the fault patterns [10, 11, 4, 5, 9], etc. but in our knowledge, none has been published on covering/sweeping all the faults lying in a specific pattern once it has been discovered.

A common practice to evaluate performance of newly created or existing strategies is to compare the results obtained (theoretically and empirically) after applying them to similar programs [22, 21, 23]. Arcuri et al., stress on the use of random testing as a comparison baseline to assess other test strategies [1]. We followed similar procedure and evaluated DSSR against R and R+ under constant conditions.

Qualitas Corpus [39] is a collection of open source java programs maintained for independent empirical research [32, 40, 38]. These projects are carefully selected that spans across the whole set of java applications.

8. CONCLUSIONS

The main goal of the present study was to develop a new random strategy which could find more faults in lower number of test cases that would leverage on the assumption that in a significant number of classes, failure domains are contiguous or are very close by. The result is the dirt spot sweeping strategy, a strategy which adds neighbouring values of failure values to a set of preferred values.

We implemented DSSR as a strategy for the random testing tool YETI and tested thirty times each one of 80 classes from the Qualitas Corpus with each of the three strategies DSSR, R, and R+.We found that for 68% of the classes all three strategies find the same unique failures, for 9% of the classes random+ performs better, for 14% pure random performs better, and for 9% DSSR performs better. Overall, DSSR also found 2.3% more unique failures than random and .3% more unique failures than random+.

Overall DSSR is a strategy that uncovers more unique failures than both random and random+ strategies. It however achieves this with a 5-10% overhead which makes it an unlikely candidate as a default strategy for a random testing tool. It however yields encouraging results and advocates to develop the technique further for settings in which it is significantly better than both R+ and R.

9. ACKNOWLEDGMENTS

The authors thank the Department of Computer Science, University of York for its financial support through the Departmental Overseas Research Scholarship (DORS) award. Authors also thanks Richard Page, for his valuable help and generous support.

10. REFERENCES

[1] A. Arcuri, M. Z. Iqbal, and L. Briand. Random testing: Theoretical results and practical implications. *IEEE Transactions on Software Engineering*, 38:258–277, 2012.

[2] B. Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[3] F. Chan, T. Chen, I. Mak, and Y. Yu. Proportional sampling strategy: guidelines for software testing practitioners. *Information and Software Technology*, 38(12):775 – 782, 1996.

[4] K. P. Chan, T. Y. Chen, and D. Towey. Restricted random testing. In *Proceedings of the 7th International Conference on Software Quality*, ECSQ '02, pages 321–330, London, UK, UK, 2002. Springer-Verlag.

[5] T. Chen, R. Merkel, P. Wong, and G. Eddy. Adaptive random testing through dynamic partitioning. In *Quality Software, 2004. QSIC 2004. Proceedings. Fourth International Conference on*, pages 79 – 86, sept. 2004.

[6] T. Chen and Y. Yu. On the expected number of failures detected by subdomain testing and random testing. *Software Engineering, IEEE Transactions on*, 22(2):109 –119, feb 1996.

[7] T. Y. Chen. Adaptive random testing. *Eighth International Conference on Qualify Software*, 0:443, 2008.

[8] T. Y. Chen, F.-C. Kuo, and R. Merkel. On the statistical properties of the f-measure. In *Quality Software, 2004. QSIC 2004. Proceedings. Fourth International Conference on*, pages 146 – 153, sept. 2004.

[9] T. Y. Chen, F. C. Kuo, R. G. Merkel, and S. P. Ng. Mirror adaptive random testing. In *Proceedings of the Third International Conference on Quality Software*, QSIC '03, page 4, Washington, DC, USA, 2003. IEEE Computer Society.

[10] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. H. Tse. Adaptive random testing: The art of test case diversity. *J. Syst. Softw.*, 83:60–66, January 2010.

[11] T. Y. Chen and R. Merkel. Quasi-random testing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 309–312, New York, NY, USA, 2005. ACM.

[12] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Experimental assessment of random testing for object-oriented software. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 84–94, New York, NY, USA, 2007. ACM.

[13] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Artoo: adaptive random testing for object-oriented software. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 71–80, New York, NY, USA, 2008. ACM.

[14] I. Ciupa, B. Meyer, M. Oriol, and A. Pretschner. Finding faults: Manual testing vs. random+ testing vs. user reports. In *Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*, pages 157–166, Washington, DC, USA, 2008. IEEE Computer Society.

[15] I. Ciupa, A. Pretschner, A. Leitner, M. Oriol, and B. Meyer. On the predictability of random tests for object-oriented software. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pages 72–81, Washington, DC, USA, 2008. IEEE Computer Society.

[16] I. Ciupa, A. Pretschner, M. Oriol, A. Leitner, and B. Meyer. On the number and nature of faults found by random testing. *Software Testing Verification and Reliability*, 9999(9999):1–7, 2009.

[17] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM.

[18] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 35(9):268–279, Sept. 2000.

[19] C. Csallner and Y. Smaragdakis. Jcrasher: An automatic robustness tester for Java. *Software—Practice & Experience*, 34(11):1025–1050, Sept. 2004.

[20] J. W. Duran and S. Ntafos. A report on random testing. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pages 179–183, Piscataway, NJ, USA, 1981. IEEE Press.

[21] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *Software Engineering, IEEE Transactions on*, SE-10(4):438 –444, july 1984.

[22] W. Gutjahr. Partition testing vs. random testing: the influence of uncertainty. *Software Engineering, IEEE Transactions on*, 25(5):661 –674, sep/oct 1999.

[23] D. Hamlet and R. Taylor. Partition testing does not inspire confidence [program testing]. *Software Engineering, IEEE Transactions on*, 16(12):1402 –1411, dec 1990.

[24] R. Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[25] A. Leitner, I. Ciupa, B. Meyer, and M. Howard. Reconciling manual and automated testing: The autotest experience. In *Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, HICSS '07, pages 261a–, Washington, DC, USA, 2007. IEEE Computer Society.

[26] A. Leitner, A. Pretschner, S. Mori, B. Meyer, and M. Oriol. On the effectiveness of test extraction without overhead. In *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, pages 416–425, Washington, DC, USA, 2009. IEEE Computer Society.

[27] H. Liu, F.-C. Kuo, and T. Y. Chen. Comparison of adaptive random testing and random testing under various testing and debugging scenarios. *Software: Practice and Experience*, 42(8):1055–1074, 2012.

[28] G. J. Myers and C. Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[29] S. C. Ntafos. On comparisons of random, partition, and proportional partition testing. *IEEE Trans. Softw. Eng.*, 27:949–960, October 2001.

[30] C. Oriat. Jartege: a tool for random generation of unit tests for java classes. *CoRR*, abs/cs/0412012, 2004.

[31] M. Oriol. York extensible testing infrastructure, 2011.

[32] M. Oriol. Random testing: Evaluation of a law describing the number of faults found. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 201 –210, april 2012.

[33] M. Oriol and S. Tassis. Testing .net code with yeti. In *Proceedings of the 2010 15th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS '10, pages 264–265, Washington, DC, USA, 2010. IEEE Computer Society.

[34] M. Oriol and F. Ullah. Yeti on the cloud. *Software Testing Verification and Validation Workshop, IEEE International Conference on*, 0:434–437, 2010.

[35] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *In 19th European Conference Object-Oriented Programming*, pages 504–527, 2005.

[36] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for Java. In *OOPSLA 2007 Companion, Montreal, Canada*. ACM, Oct. 2007.

[37] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society.

[38] E. Tempero. An empirical study of unused design decisions in open source java software. In *Software Engineering Conference, 2008. APSEC '08. 15th Asia-Pacific*, pages 33 –40, dec. 2008.

[39] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. Qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, Dec. 2010.

[40] E. Tempero, S. Counsell, and J. Noble. An empirical study of overriding in open source java. In *Proceedings of the Thirty-Third Australasian Conferenc on Computer Science - Volume 102*, ACSC '10, pages 3–12, Darlinghurst, Australia, Australia, 2010. Australian Computer Society, Inc.