MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

Improvements of the Randomness Testing Toolkit

Bachelor's Thesis

TOMÁŠ MAREK

Brno, Fall 2023

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Improvements of the Randomness Testing Toolkit

Bachelor's Thesis

TOMÁŠ MAREK

Advisor: Ing. Milan Brož, Ph.D.

Department of Computer Systems and Communications

Brno, Fall 2023


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Tomáš Marek

Advisor: Ing. Milan Brož, Ph.D.

Acknowledgements

These are the acknowledgements for my thesis, which can span multiple paragraphs.

Abstract

This is the abstract of my thesis, which can
span multiple paragraphs.

Keywords

keyword1, keyword2, ...

Contents

Introduction

To be done, now it is only a collection of ideas.

The desired properties of random sequence are *uniformity* (for each bit the probability for both zero and one are exactly $1/2$), *independence* (none of the bits is influenced by any other bit) and *unpredictability* (it is impossible to predict next bit by obtaining any number of previous bits). [1, p. 1-1]

1 Randomness testing

Goal of this chapter is to provide overview of randomness testing process and to explain all used terms. Explanations of both one and two level tests are accompanied by example applications.

1.1 Overview / Introduction

During a randomness test a *random sequence* is tested. In this document, a random sequence is a finite sequence of zero and one bits, which was generated by a tested random number generator. [1, p. 1-1]

Randomness test is a form of *empirical statistical test*, where we test our assumption about the tested data - the *null hypothesis* (H_0). During the randomness test it states that the sequence is *random*. Associated with the null-hypothesis is the *alternative hypothesis* (H_1), which states that the sequence is *non-random*. Goal of the test is to search for evidence against the null-hypothesis. [2, p. 2]

The result of the test is either that we *accept* the null hypothesis (the sequence is considered random), or that we *reject* the null-hypothesis (and accept the alternative hypothesis - the sequence is considered non-random). We reject the null hypothesis when the evidence found against the null-hypothesis is strong enough, otherwise we accept it. Based on the true situation of null hypothesis, four situations depicted in Table 1.1 may occur. [3, p. 417]

Table 1.1: Possible outcomes when assessing the result of statistical test.

TRUE SITUATION	TEST CONCLUSION	
	Accept H_0	Reject H_0
H_0 is True	No error	Type I error
H_0 is False	Type II error	No error

1.2 Single-level testing

A single-level test examines the random sequence directly (compare with 1.3). Before the test user must choose a *significance level*, which determines how strong the found evidence has to be to reject the null-hypothesis. The test yields a *p-value*, which is used to make the accept or reject the null-hypothesis.

The *significance level* (α) is crucial to assessing the test result and must be set before the test. The α is equal to probability of Type I Error. Usual values are $\alpha = 0.05$ or $\alpha = 0.01$ [3, p. 390], for use in testing of cryptographic random number generators lower values may be chosen. [1, p. 1-4] The lower α is set, the stronger the found evidence has to be to reject the null hypothesis.

The randomness test is defined by a *test statistic* Y, which is a function of a finite bit sequence. Distribution of its values under the null hypothesis must be known (or at least approximated). The value of the test statistic (y) is computed for the tested random sequence. Each test statistic searches for presence or absence of some "pattern" in the sequence, which would show the non-randomness of the sequence. There is infinite number of possible test statistics. [4, p. 4]

The *p-value* is the probability of the test statistic Y taking value at least as extreme as the observed y, assuming that the null hypothesis is true. In randomness testing it is equal to the probability that *perfect random number generator* would generate less random sequence. The smaller is the p-value, the stronger is the found evidence against the null-hypothesis. [3, p. 386] The p-value is calculated based on the observed y.

1.2.1 Result interpretation

Decision about the test result is based on the computed *p-value*. If the p-value is lower than the α, we *reject the null hypothesis* (and accept the alternative hypothesis), because strong enough evidence against null hypothesis was found. If the p-value is greater than or equal to the α, we *accept the null hypothesis*, because the evidence against the randomness was too weak. [3, p. 390] It is sometimes recommended to report the *p-value* as well instead of accept/reject only, as it yields more information. [2, p. 90]

The p-values close to α can be considered *suspicious*, because they do not clearly indicate rejection. Further testing of the random number generator on *other* random sequences is then in place to search for further evidence. [4, p. 5] The reason is that *randomness* is a probabilistic property, therefore even the perfect random number generator may generate a nonrandom sequence with low p-value (although it is very unlikely). The further evidence is used to differentiate between the bad generator generating a non-random sequence systematicaly and the good generator generating non-random sequence 'by chance'. [2, p. 90]

1.2.2 Example

To demonstrate how a single randomness test is made, the Frequency (Monobit) Test from NIST STS battery was chosen. [1, p. 2-2] This test is based on testing the fraction of zeroes and ones within the sequence. For a random sequence with length n the count of ones (and zeroes) is expected to be around $n/2$ (the most probable values are close to $n/2$).

For the Monobit test it is recommended that the tested sequence has at least 100 bits. The test statistic S_{obs} of the Monobit test is defined as

$$S_{obs} = \frac{|\#_1 - \#_0|}{\sqrt{n}}$$

where $\#_1$ is count of ones in the tested sequence (similarly for zeroes) and n is length of of the tested sequence. Under the null hypothesis, the reference distribution of S_{obs} is half normal (for large n). The p-vaue is computed as

$$p = erfc(\frac{S_{obs}}{\sqrt{2}})$$

where $erfc$ is the *complementary error function* used to calculate probabilities in normal distribution.

Let

$$\epsilon = 1001100101001000001000100101100110110000110100011110101001010010010011100111001100110010010100111011$$

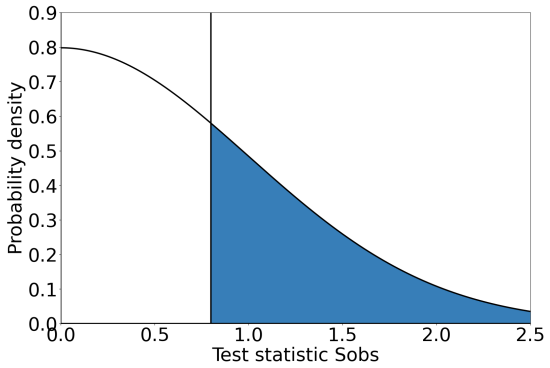


Figure 1.1: Visualization of example p-value for test statistic value $y = 0.8$

be the tested sequence. The test statistic for this sequence is

$$S_{obs} = \frac{|46 - 54|}{\sqrt{100}} = \frac{|-8|}{10} = 0.8$$

and the p-value (visualised at Figure 1.1) is

$$p = erfc\left(\frac{0.8}{\sqrt{2}}\right) \approx 0.423$$

To interpret the test, we compare the computed *p-value* to the chosen α. The *p-value* ≈ 0.423 is greater than both usual $\alpha = 0.05$ and $\alpha = 0.01$, therefore we accept the null hypothesis for both *significance levels* and the sequence ϵ is considered random.

1.3 Two-level testing

The *two-level test* is done by repeating the single-level test n times. The important part is comparing the distribution of produced p-values to the expected distribution. The two-level test allows the random sequence to be examined both locally and globally, while the single-level test examines the sequence only on the global level. This may lead to discovering local patterns, which cancel out on the global level. [4, p. 7]

To apply the two-level test the tested sequence is split into n equal-length disjoint subsequences. The same single-level test is applied to each of the subsequences (as described in Section 1.2) and its *p-values* are collected, resulting in set of n *p-values*[1]. The tests are called *first-level tests* and the p-values are called *first-level p-values*. Under the null-hypothesis, the first-level p-values of a given test statistic are uniformly distributed over the interval $(0, 1]$. [5, p. 14]

The crucial part of two-level test is examining the distribution of *observed first-level p-values*. Usually, the *goodness-of-fit* (GOF) tests are applied as the *second-level test*. [4, p. 6] GOF tests are a family of methods used for examining how well a data sample fits given distribution. [6, p. 1] The most used GOF tests in randomness testing are the χ^2 (chi-squared) and Kolmogorov-Smirnov test, another notable tests are the Anderson-Darling and Cramér-von-Mises test. [5, p. 14]

The second-level test is defined by a test statistic Y, which is a function of the first-level p-values. Test statistic value (y) is calculated from the observed *first-level p-values* and then the *second-level p-value* is calculated from y. At last, the second-level p-value is interpreted as in one-level test (as described in Subsection 1.2.1).

Alternatively, a *proportion of subsequences passing the first-level test* is used to examine the fist-level p-values uniformity. Under the null-hypothesis, it is expected for $n \cdot \alpha$ subsequences to *be rejected* (i.e. to have p-value $< \alpha$) by the first-level test (be a subject to Type I Error). The ratio of sequences passing the first-level test is expected to be around $1 - \alpha$, different ratio indicates non-uniformity of observed first-level p-values. [1, p. 4-2] No p-value is reported in this case, only the ratio.

1.3.1 Kolmogorov-Smirnov test

The one-sample Kolmogorov-Smirnov (KS) test is used in randomness testing to compare the observed first-level p-values to the uniform distribution. The Kolmogorov-Smirnov test is built on comparing the cumulative distribution function (CDF)[2] of the expected distribution

1. Note that the p-values are not subject to accept/reject decision.
2. For a given distribution and value x, the CDF returns the probability of drawing a value less than or equal to x.

and the empirical cumulative distribution function (eCDF)[3] of the observed samples.

In first variant, two test statistics are calculated. The test statistic D^+ (D^-) is the maximal vertical distance between CDF and eCDF above (under) the CDF. In second variant, only the test statistic D (maximal vertical distance between CDF and eCDF) is measured. Formally, the test statistics are defined as

$$\begin{aligned} D^+ &= sup_x\{F_n(x) - F(x)\} \\ D^- &= sup_x\{F(x) - F_n(x)\} \\ D &= sup_x\{|F_n(x) - F(x)|\} = max(D^+, D^-) \end{aligned}$$

where $F(x)$ is the CDF and $F_n(x)$ is the eCDF [6, p. 100].

1.3.2 Chi-squared test

The Pearson's χ^2 test is used to find statistically significant difference between frequencies of categories in two sets of categorical data. The first-level p-values are split into k equal-width bins (categories) and their respective frequencies are counted. The counted frequencies are compared to the expected frequencies.

For data with k categories the test statistic χ^2 is defined as

$$\chi^2 = \sum_{i=1}^{k} \frac{(x_i - m_i)^2}{m_i}$$

where x_i is the observed frequency in i-th category and m_i is the expected frequency in i-th category. For first-level p-values, the expected frequency is equal in each interval. For a correct test the expected frequency in each category must be at least five. [7, p. 171]

1.3.3 Example

In the two-level test example, I will test one sequence using both one and two-level tests to demonstrate the difference between them. First, the sequence is tested using the one-level Frequency (Monobit) test

3. For a set of observed data and value x, the eCDF returns the probability of drawing a value less than or equal to x.

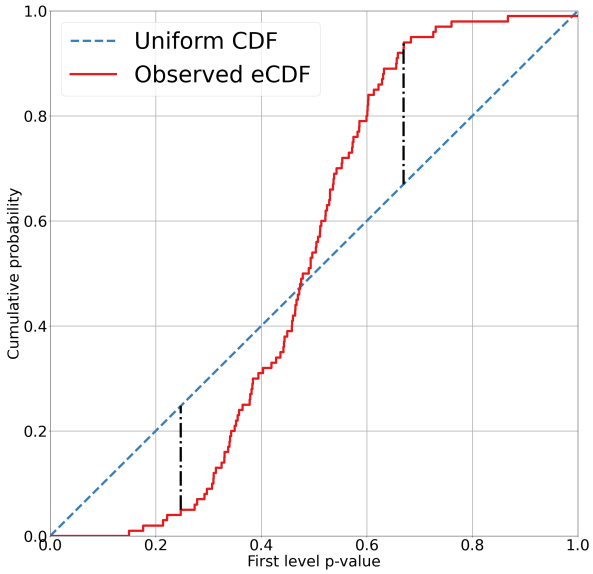


Figure 1.2: Visualization of Kolmogorov-Smirnov test statistics for expected uniform distribution and observed data sample.

Table 1.2: First-level p-values produced by Monobit test

p-value	occurrences
$1.52 \cdot 10^{-23}$	30
0.31	5
1.0	15

from NIST STS battery.[1, p. 2-2] Then the same sequence is assessed by the two-level test using the Frequency test as the first-level test and KS and χ^2 tests as second-level test. Let

$$\begin{aligned}\epsilon = \; & 15 * (100\, \textit{consecutitive zeroes}) + \\ & 15 * (100\, \textit{alternating ones and zeroes}) + \\ & 5 * (55\, \textit{zeroes and}\, 45\, \textit{ones}) + \\ & 15 * (100\, \textit{consecutive ones})\end{aligned}$$

be the tested sequence.

Result of the one-level Frequency test for the sequence ϵ is p-value ≈ 0.479. The null hypothesis is accepted for both $\alpha = 0.01$ and $\alpha = 0.05$ and the sequence ϵ is considered random. This sequence however clearly contains a pattern, therefore the probability of it being generated by a perfect random number generator is very low.

For the two-level test, the sequence ϵ is split into $n = 50$ disjoint 100 bit long subseqeunces. The Monobit test is applied on each subsequence resulting in set of first-level p-values shown in Table 1.2.

Last step is to apply the goodness-of-fit tests. The first applied test is the Pearson's χ^2 test with $k = 10$ (number of categories), the expected frequency of p-values in each category is five. The statistic of the test is

$$\chi^2 = \sum_{i=1}^{10} \frac{(x_i - 5)^2}{5} = 180$$

and the p-value of this test is $p \approx 5.06 \cdot 10^{-34}$. The null hypothesis is rejected for both $\alpha = 0.01$ and $\alpha = 0.05$.

Next, the Kolmogorov-Smirnov test is applied. The eCDF is calculated and then the D statistic is computed. The statistic is $D = 0.6$ and results in p-value $\approx 9.63 \cdot 10^{-18}$. Again, the null-hypothesis is re-

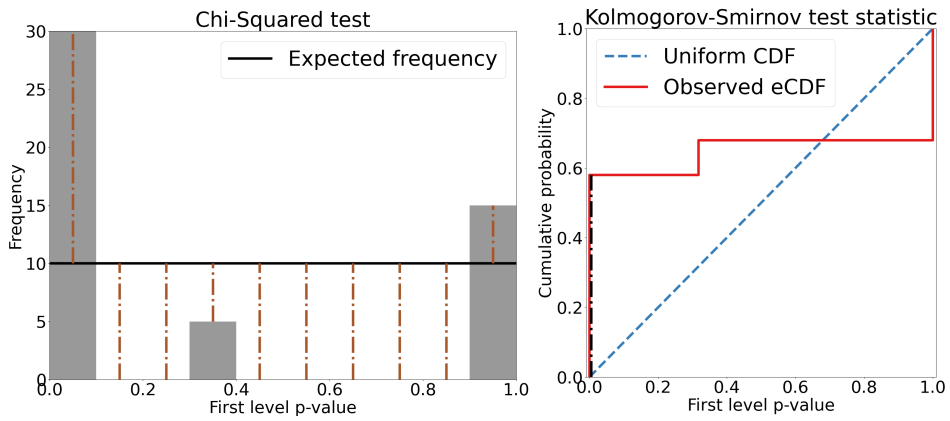


Figure 1.3: Visualisations for χ^2 and KS tests for two-level test example.

jected for both $\alpha = 0.01$ and $\alpha = 0.05$ and the sequence is considered non-random.

2 Available solutions

This chapter serves to describe various works and programs this thesis connects to.

2.1 Statistical testing batteries

More or less deep description of each battery. Should contain information about test parameters/settings.

command on how to run battery
Mention overflow detection
If there are any problems with the battery (e.g. tests which read different amount of data from DieHarder). - Discuss colision with the paper
how batteries interpret results (first/second level, more statistics)
strong and weak things
describe output / image

General description of battery - set of tests, choose test by ID, set tested file
individual test - smallest executable part, possibly more test statistics, setting of the test(ntup, bitw...), data consumption (fixed vs set by user), number of first levels, return-first level pvalues, second-level p-value,

Randomness testing battery is a set of randomness tests

Individual test is the smallest executable unit and applies one two-level test on the tested sequence. Some individual tests apply more than one two-level test. For example, the Diehard Craps Test plays 200 000 games of craps, the test statistics are based on number of wins and number of throws needed to end the game TODO: cite dieharder.

Usually, the following settings are required for each *individual test*.

- number of first-level tests
- test statistic settings
- size of first-level sequence (if applicable)

2.1.1 Dieharder

Explain p-samples, name tests with irregular read bytes

2.1.2 NIST STS

Explain stream-size and stream-count
Results - in experiments directory

2.1.3 Test U01

List all batteries. Explain repetitions, –bit -nb -w -r -s, mention repeating tests with different parameters

2.2 Testing toolkits

JUST COPIED FROM WORK TO ACADEMIC WRITING COURSE, TO BE USED AND CHANGED LATER.

TODO: MENTION INSTALATION

In the previous chapter different randomness testing batteries were described. The typical user, however, uses more than one battery, which means installing and running each testing battery individually. Also it is strongly recommended (sometimes even needed) to set up parameters for each test from the battery individually based on tested file and to run this test manually.

Since this approach is not convenient, Ľubomír Obrátil from Center for Research on Cryptography and Security (CRoCS) at FI MU created the Randomness Testing Toolkit (*RTT*). This toolkit allows users to run and configure three test batteries by a single command.

This work was followed by Patrik Vaverčák from Faculty of Electrical Engineering and Information Technology at Slovak University of Technology. He created newer variant of *RTT* called Randomness Testing Toolkit in Python (*rtt-py*). Compared to *RTT*, it contains two additional test batteries.

2.3 Randomness Testing Toolkit

RTT was created in 2017 and its main idea was to combine *Dieharder*, *NIST STS*[1] and *Test U01* statistical test batteries into one program. It was written in C++.

The concept of *RTT* is that it acts only as a unified interface of the batteries. Each test battery is executed by *RTT* as a separate program. The *RTT* then collects the output and processes in into a unified format. [8, p. 8]

However some problems in the processing of the output were found; these are addressed in chapter **??**.

2.3.1 Settings

TODO: MORE RIGID DESCRIPTION The *RTT* needs to be set up by the user before running. The first part of user settings contains general settings made for the *RTT*, the second part contains configuration for individual test batteries. Each of these parts is stored in its own JSON[2] file. The original setup description is from

The general settings are stored in *rtt-settings.json* file, which has to be located in the working directory of the *RTT* [8, p. 10] . These settings are usually not changed between runs. The most important setting from the general part are paths to the executable binaries of individual statistical test batteries. This is the only setting that has to be manually filled by the user.

The storage database can also be filled in by the user, but this functionality is often unused. The following general settings have implicit values and do no need to be changed unless the user wishes to. They are paths to storage directories for results and logs of individual runs and execution options (test timeout and maximal number of parallel executions of tests).

The battery configurations are dependant on the size of the tested file, therefore the file with the battery configuration is specified for each run of the *RTT*. These configurations are different for each battery (see sections **??**, **??** and **??**), but settings for all of the batteries can be

1. National Institute of Standards and Technology - Statistical Test Suite
2. JavaScript Object Notation

stored together in a single file. [8, p. 11] The *RTT* contains several prepared battery configurations for various sizes of tested file.

2.3.2 Output

The output of *RTT* is in a plain text format. The most important part of the output is the direct report, which is saved in the results directory. At the beginning of the report are general information – the name of the tested file, the name of the used battery, ratio of passed and failed tests and battery errors and warnings in case there were any.

After the general information is a list of results of individual test runs in a unified format. The fist part of the single test report contains the name of the test and user settings (*e.g. P-sample count in Dieharder battery or Stream size and count in NIST STS battery*). The second part of the single test report are the resulting second-level P-values alongside the names of statistic used (usually Kolmogorov-Smirnov statistic or Chi-Square test). At the end of the single test report is a list of first-level P-values produced by the test. Example of the output can be seen in Figure 2.1.

2.3.3 Disadvantages

The problems/weak points we want to improve with this thesis. Namely at least non machine-machine readable format, running only one battery at time, maybe re-calculation of results

There are two most notable disadvantages of the *RTT*. The first one is that each battery has to be run individually by the user. This lowers the convenience of usage for the user. The second one is the output format. While it is easy to read for human users, machine reading requires complicated parsing.

2.4 Randomness Testing Toolking in Python

The Randomness Testing Toolkit in Python (*rtt-py*) was created by Patrik Vaverčák. It is supposed to be a better version of *RTT* [9, p. 24] and it was written in Python. However there are still some functional differences between *RTT* and *rtt-py*. The most notable difference is in the output format and supported batteries.

```
------------------------------------------------------------
Diehard Birthdays Test test results:
    Result: Passed
    Test partial alpha: 0.01

    User settings:
        P-sample count: 65
    ************

    Kolmogorov-Smirnov statistic p-value: 0.46269520       Passed
    p-values:
        0.01554128 0.01704044 0.07338199 0.08890865 0.13047059
        0.14641850 0.14648858 0.14985241 0.15741014 0.17234854
        0.17570707 0.18313806 0.19708195 0.21929163 0.23582928
        0.23875056 0.24659048 0.24810255 0.26921690 0.29350665
        0.29444024 0.29618689 0.30017915 0.30767530 0.32816499
        0.33671597 0.33723518 0.33723518 0.35046577 0.36986762
        0.38616538 0.40739822 0.42316216 0.42606175 0.42712489
        0.46376818 0.47710967 0.51301110 0.55638736 0.58615965
        0.58816320 0.62212002 0.63106447 0.65794861 0.66078115
        0.66209483 0.67060673 0.69336319 0.70343506 0.72259414
        0.74451995 0.79749441 0.81986290 0.85442793 0.88851953
        0.88897431 0.89604503 0.92240447 0.93852901 0.93852901
        0.95468456 0.96540827 0.96785289 0.96922576 0.99790555
    ============
------------------------------------------------------------
```

Figure 2.1: The example of single test report from the *RTT*

2.4.1 Settings

The settings of *rtt-py* are very similar to the original *RTT*. According to Vaverčák, the general settings from the *RTT* should be compatible with *rtt-py*, but in reality there is problem with settings for the NIST STS's experiments directory. Also, no database connection is implemented in *rtt-py*, therefore the *mysql-db* attribute is ignored. [**rtt-py-github**]

The second part of user settings are tests configurations. They use exactly the same format as those used in *RTT* (as mentioned in 2.3.1) and are interchangeable. [9, p. 25] The user has to keep in mind that the *rtt-py* uses FIPS[3] and BSI[4] batteries, which are not used in *RTT*.

2.4.2 Output

There is a significant difference in the output format between *RTT* and *rtt-py*. The *rtt-py* creates output in two formats – *CSV*[5] and *HTML*[6]. Both of these report formats contain overview table. Each row from the table represents results of one particular test. The first column contains the name of the test and the name of the battery it belongs to.

The second column contains *failure rate* - ratio representing how many instances of this particular test failed compared to number of executed instances on *all* files with data.

Each of the following columns is named after one tested file. The record contains either P-value reported by the test, or number of failed runs – this depends on the battery. Example of this table can be seen at figure 2.2.

The output in the HTML format contains more information compared to the output in the CSV format. For each battery and for each tested file an HTML file with reports is generated.

In each report file there is a list of reports for each executed test from the given battery. The single test report contains the result of the test (either reported P-value, or number of failed runs) and it may contain additional information such as settings of the test or other information connected to the result. The contained information

3. Federal Information Processing Standards
4. Bundesamt für Sicherheit in der Informationstechnik
5. Comma-separated values
6. Hypertext Markup Language

Results overview

	Failure rate	../input2/1000MB.dat	../input2/1000MB_2.dat
rgb_minimum_distance_0 (DIEHARDER)	0.0	0.754103	0.407390
rgb_permutations_0 (DIEHARDER)	0.0	0.931074	0.184047
diehard_operm5_0 (DIEHARDER)	0.0	0.228052	0.680182
sts_monobit_0 (DIEHARDER)	0.0	0.279259	0.096535
sts_serial_0 (DIEHARDER)	0.0	0.279259	0.096535
sts_serial_1 (DIEHARDER)	0.0	0.972893	0.052121
sts_serial_2 (DIEHARDER)	0.0	0.621721	0.618407

Figure 2.2: The example of the overview table from the *rtt-py*

depends on the battery and on the executed test. Example of the report can be seen in figure 2.3

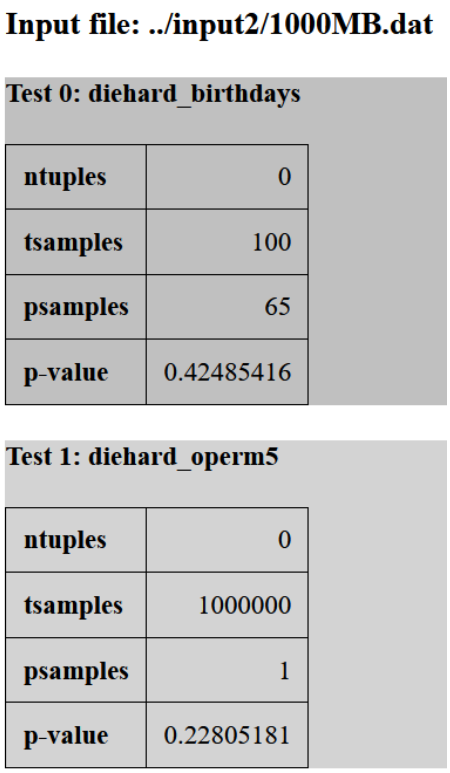
2.4.3 Disdvantages

One of the problems that need to be addressed is that the *rtt-py* ignores errors and warnings from tests. The most notable example why this is a problem is when the tested file does not contain enough data for current battery configuration.

In this case, the test will read some parts of the data more than once and inform the user about this situation on the error output. The test will still produce result, which will, however, be biased by repeated parts of the tested file.

This may lead to incorrect interpretation of the results and to false acceptance or false rejection of the tested data. Since the *rtt-py* ignores this, there is no way for the user to be informed about this situation.

Compared to the *RTT* the reports created by *rtt-py* contain less information. Namely the first-level P-values are ignored, even though they can be useful for deeper examination of the results and the generator.



Input file: ../input2/1000MB.dat

Test 0: diehard_birthdays

ntuples	0
tsamples	100
psamples	65
p-value	0.42485416

Test 1: diehard_operm5

ntuples	0
tsamples	1000000
psamples	1
p-value	0.22805181

Figure 2.3: The example of HTML Dieharder report from the *rtt-py*

3 Tests Analysis

We can choose from various test statistics. Most of the test statistics in widely used test batteries work with data of fixed length. TODO: REF ANALYSIS CHAPTER However, in some tests data with varying length are tested. These statistics further split into two categories. In the first category, the length of tested data is preset by user. These can be further viewed as fixed-length tests. In the second category, the length of tested data is determined during the testing process.

3.1 Data Consumption

several big tables, mention exact parameters the tests were run with

3.2 Time Consumption

again some big tables, choose one test as a reference and the rest will be relative. mention exact parameters, maybe add throughput?

3.3 Configuration Calculator

goal of the config calc, description, usage etc...

3.4 P-Values

Various problems with test p-values distributions, will probably be split into more sections

4 Implementations Comparison

4.1 Output

Mentioned differences
for both RTT and rtt-py - subset or whole?

4.2 Missing Features of *rtt-py*

4.3 Proposed improvements

included things: adding first-level p-values,

5 Conclusion

A An appendix

Here you can insert the appendices of your thesis.

Bibliography

1. BASSHAM III, Lawrence E; RUKHIN, Andrew L; SOTO, Juan; NECHVATAL, James R; SMID, Miles E; BARKER, Elaine B; LEIGH, Stefan D; LEVENSON, Mark; VANGEL, Mark; BANKS, David L, et al. *SP 800-22 rev. 1a. a statistical test suite for random and pseudorandom number generators for cryptographic applications*. National Institute of Standards & Technology, 2010. Available also from: `https://csrc.nist.gov/Projects/random-bit-generation/Documentation-and-Software`.
2. L'ECUYER, Pierre; SIMARD, Richard. *TestU01 - A Software Library in ANSI C for Empirical Testing of Random Number Generators - User's guide, compact version*. 2002. Available also from: `http://simul.iro.umontreal.ca/testu01/guideshorttestu01.pdf`.
3. MOORE, David S.; NOTZ, William I. *The Basic Practice of Statistics*. Macmillan Learning, 2021. ISBN 1-319-38368-8.
4. L'ECUYER, Pierre; SIMARD, Richard. TestU01: A C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software (TOMS)*. 2007, vol. 33, no. 4, pp. 1–40.
5. SÝS, Marek; OBRÁTIL, Lubomír; MATYÁŠ, Vashek; KLINEC, Dušan. A Bad Day to Die Hard: Correcting the Dieharder Battery. *Journal of Cryptology*. 2022, vol. 35, pp. 1–20.
6. D'AGOSTINO, Ralph B.; STEPHENS, Michael A. *Goodness-of-Fit-Techniques*. Routledge, 1986. ISBN 0-8247-8705-6.
7. SHESKIN, David J. Parametric and nonparametric statistical procedures. *Boca Raton: CRC*. 2000.
8. OBRÁTIL, Ľubomír. *The automated testing of randomness with multiple statistical batteries*. Brno, 2017. Available also from: `https://is.muni.cz/th/uepbs/`. Master's thesis. Masaryk University, Faculty of Informatics. Supervised by Petr ŠVENDA.

9. VAVERČÁK, Patrik. *Aplikácia na štatistické testovanie pseudonáhodných postupností*. Bratislava, 2022. Available also from: `https://opac.crzp.sk/?fn=detailBiblioFormChildEBUS6&sid=D9F0BB0A8DA9926980A890D31B33&seo=CRZP-detail-kniha`. Master's thesis. Slovak University of Technology in Bratislava, Faculty of Electrical Engineering. Supervised by Matúš JÓKAY.