CS 562　Final Report

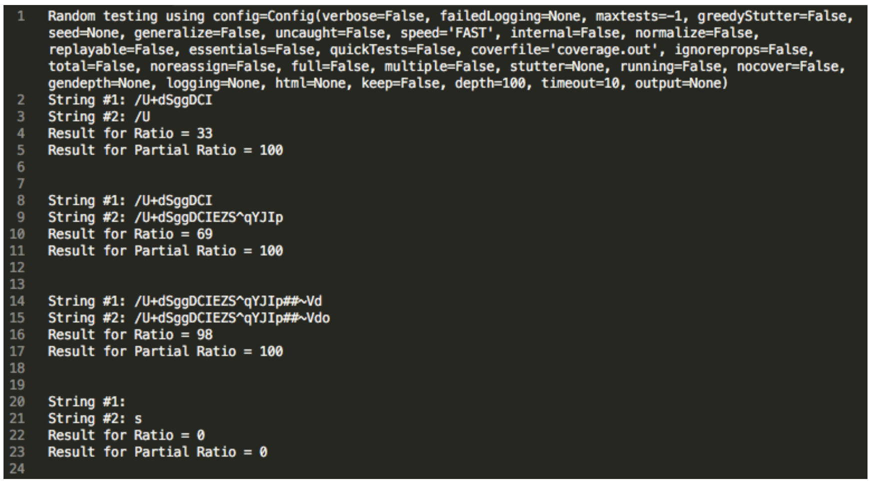
Haoxiang Wang, Student ID: 932359049

For the term testing project, a third party python library named "***fuzzywuzzy***" is fully tested. The library is implemented to accomplish the fuzzy string matching, and it can be found here on the Github: https://github.com/seatgeek/fuzzywuzzy. In this library, several functions are provided in order to handle several different situations. By the end of the term, which is the time this report is written, all of the functions in this library are tested. One bug is found keeping showing in different functions.

1. Accomplishments

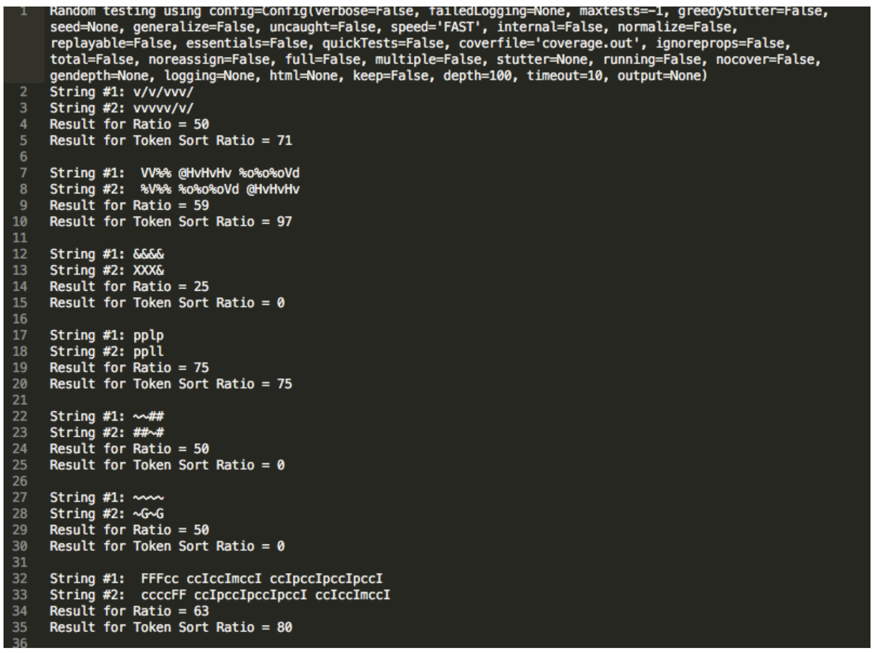
To introduce the accomplishment of the project, all test files I wrote for all functions will be explained. There are 6 functions in the library, and 4 testers are written to test them. The file "ratio_test.tstl" tests the functions named "*fuzz.ratio*" and "*fuzz.partial_ratio*". The "token_sort_test.tstl" tests the function "*fuzz.token_sort_ratio*", while another file named "token_set_test.tstl" tests the function "*fuzz.token_set_ratio*". Finally, the file "process_test.tstl" tests another two functions from different file named "*process.extract*" and "*process.extractOne*"*.* All of these tests and results will be introduced in the following paragraphs.

The first tstl file creates the strings contains lower and upper cases of letters along with special symbols with some common parts and passes them to "*fuzz.ratio*" and "*fuzz.partial_ratio*" functions. The "*fuzz.ratio*" function will test the most common part of two strings and return the similarity percentage. While "*fuzz.ratio*" function takes two entire string into account, "*fuzz.partial_ratio*" function only tests part of the strings. If string2 is part of the string1, "*fuzz.partial_ratio*" will return 100 percent match instead of a lower percentage returned by "*fuzz.ratio*". Therefore, for two strings have a lot common part, the "*fuzz.partial_ratio*" will always return a higher percentage than "*fuzz.ratio*" does. The result turned out that both of these two functions can handle the strings perfectly, and the "*fuzz.partial_ratio*" always returns higher match percentage than "*fuzz.ratio*" does as expected. These two functions worked just fine and no bug is found. Some strings for testing and their results are shown in the following screenshot.



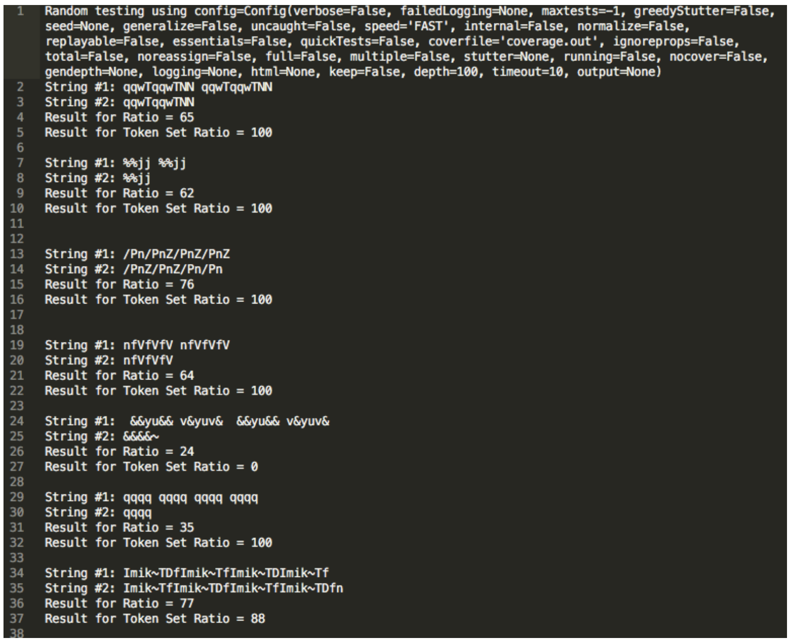
```
Random testing using config=Config(verbose=False, failedLogging=None, maxtests=-1, greedyStutter=False,
seed=None, generalize=False, uncaught=False, speed='FAST', internal=False, normalize=False,
replayable=False, essentials=False, quickTests=False, coverfile='coverage.out', ignoreprops=False,
total=False, noreassign=False, full=False, multiple=False, stutter=None, running=False, nocover=False,
gendepth=None, logging=None, html=None, keep=False, depth=100, timeout=10, output=None)
String #1: /U+dSggDCI
String #2: /U
Result for Ratio = 33
Result for Partial Ratio = 100


String #1: /U+dSggDCI
String #2: /U+dSggDCIEZS^qYJIp
Result for Ratio = 69
Result for Partial Ratio = 100


String #1: /U+dSggDCIEZS^qYJIp##~Vd
String #2: /U+dSggDCIEZS^qYJIp##~Vdo
Result for Ratio = 98
Result for Partial Ratio = 100


String #1:
String #2: s
Result for Ratio = 0
Result for Partial Ratio = 0

```

The second tstl file creates the strings with a pretty similar way as the first one does. This file takes “*fuzz.token_sort_ratio*” and “*fuzz.ratio*” into test. The “*fuzz.token_sort_ratio*” ignores the order of “words” in the string. The “words” are separated by “space” in the strings. Therefore, the only difference lies in this file’s creating strings part is taking “space” into account. Considering the fact that “*fuzz.token_sort_ratio*” will sort the words first and then execute the comparing, when string2 contains similar “words” with different order as the string1 does, “*fuzz.token_sort_ratio*” will return a higher percentage than “*fuzz.ratio*” does. Since TSTL executes the commands randomly, there exists the situation that the strings only contain one “word”. Under this situation, these two functions should return the same percentage with respect to their working logics. However, during the test, when the “*fuzz.token_sort_ratio*” is dealing with the strings contain special symbols, it will return a lower percentage than the “*fuzz.ratio*” does. This situation is really obvious when the strings only have one “word”. This result shows that “*fuzz.token_sort_ratio*” could not deal with the special symbols well, and this is a BUG! The sample testing results shown in the following image, and the bug will be explained in the next section.



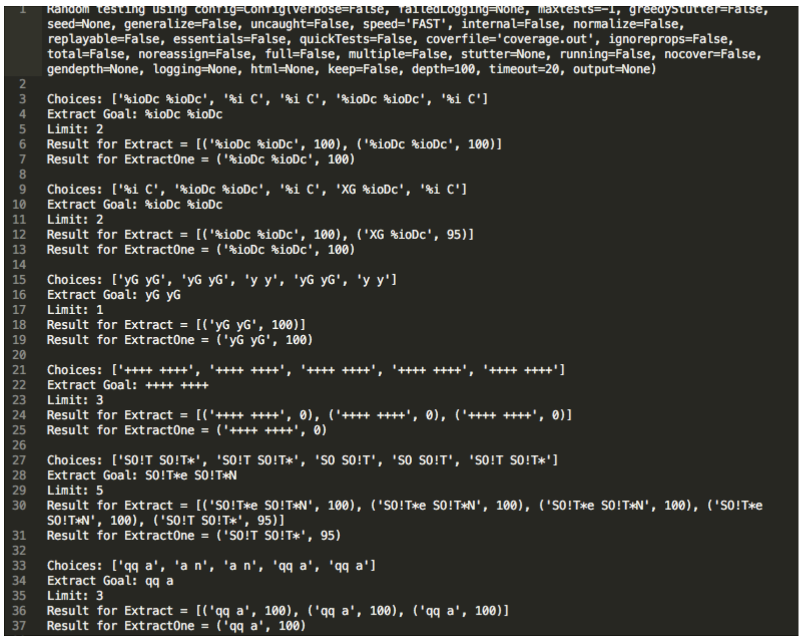
```
Random testing using config=Config(verbose=False, failedLogging=None, maxtests=-1, greedyStutter=False,
seed=None, generalize=False, uncaught=False, speed='FAST', internal=False, normalize=False,
replayable=False, essentials=False, quickTests=False, coverfile='coverage.out', ignoreprops=False,
total=False, noreassign=False, full=False, multiple=False, stutter=None, running=False, nocover=False,
gendepth=None, logging=None, html=None, keep=False, depth=100, timeout=10, output=None)
String #1: v/v/vvv/
String #2: vvvvv/v/
Result for Ratio = 50
Result for Token Sort Ratio = 71

String #1:  VV%% @HvHvHv %o%o%oVd
String #2:  %V%% %o%o%oVd @HvHvHv
Result for Ratio = 59
Result for Token Sort Ratio = 97

String #1: &&&&
String #2: XXX&
Result for Ratio = 25
Result for Token Sort Ratio = 0

String #1: pplp
String #2: ppll
Result for Ratio = 75
Result for Token Sort Ratio = 75

String #1: ~~##
String #2: ##~#
Result for Ratio = 50
Result for Token Sort Ratio = 0

String #1: ~~~~
String #2: ~G~G
Result for Ratio = 50
Result for Token Sort Ratio = 0

String #1:  FFFcc ccIccImccI ccIpccIpccIpccI
String #2:  ccccFF ccIpccIpccIpccI ccIccImccI
Result for Ratio = 63
Result for Token Sort Ratio = 80

```

The testing for the function “*fuzz.token_set_ratio*” is really similar to the function “*fuzz.token_sort_ratio*”. The only difference is that instead ignoring the order of the words, “*fuzz.token_set_ratio*” ignoring the repeating words. Therefore, if string2 contains the same words as string1 does, and it has a few words repeated several times, the “*fuzz.token_set_ratio*” will return a higher matching percentage than the “*fuzz.ratio*” does. Creating the test string is also similar to the test for “*fuzz.token_sort_ratio*”. Instead swapping the substrings’ order, repeating the same string is applied. Just as the prediction made in the previous report, the same bug also exists in this function, the function deals with the special symbols in a wrong way. The sample testing result is shown in the following image.



```
Random testing using config=Config(verbose=False, failedLogging=None, maxtests=-1, greedyStutter=False,
seed=None, generalize=False, uncaught=False, speed='FAST', internal=False, normalize=False,
replayable=False, essentials=False, quickTests=False, coverfile='coverage.out', ignoreprops=False,
total=False, noreassign=False, full=False, multiple=False, stutter=None, running=False, nocover=False,
gendepth=None, logging=None, html=None, keep=False, depth=100, timeout=10, output=None)
String #1: qqwTqqwTNN qqwTqqwTNN
String #2: qqwTqqwTNN
Result for Ratio = 65
Result for Token Set Ratio = 100

String #1: %%jj %%jj
String #2: %%jj
Result for Ratio = 62
Result for Token Set Ratio = 100


String #1: /Pn/PnZ/PnZ/PnZ
String #2: /PnZ/PnZ/Pn/Pn
Result for Ratio = 76
Result for Token Set Ratio = 100


String #1: nfVfVfV nfVfVfV
String #2: nfVfVfV
Result for Ratio = 64
Result for Token Set Ratio = 100

String #1:  &&yu&& v&yuv&  &&yu&& v&yuv&
String #2: &&&&~
Result for Ratio = 24
Result for Token Set Ratio = 0

String #1: qqqq qqqq qqqq qqqq
String #2: qqqq
Result for Ratio = 35
Result for Token Set Ratio = 100

String #1: Imik~TDfImik~TfImik~TDImik~Tf
String #2: Imik~TfImik~TDfImik~TfImik~TDfn
Result for Ratio = 77
Result for Token Set Ratio = 88

```

The "*process.extract*" and "*process.extractOne*" functions come from another file named "process". These two functions basically provide a search-like operation. The function "*process.extractOne*" takes a list of strings as a choice list as input, and finding one inputted target string in the choices. It also returns the matching percentage at the same time. The function "*process.extract*" takes one more input than "*process.extractOne*", which is named as "limit". The limit controls the number of results the function returns, and the function also returns the matching percentage as one of the outputs. These two functions are tested in one file named "process_test.tstl". Depending on the logic these two functions hold, the "*process.extractOne*" should always return the same result as first result "*process.extract*" returns. Unexpectedly, the same bug "*fuzz.token_set_ratio*" and "*fuzz.token_sort_ratio*" have also exists in these two functions. Both of these two functions cannot deal with special symbols correctly. The sample testing result is shown in the following image.



```
Random testing using config=Config(verbose=False, failedLogging=None, maxtests=-1, greedyStutter=False,
seed=None, generalize=False, uncaught=False, speed='FAST', internal=False, normalize=False,
replayable=False, essentials=False, quickTests=False, coverfile='coverage.out', ignoreprops=False,
total=False, noreassign=False, full=False, multiple=False, stutter=None, running=False, nocover=False,
gendepth=None, logging=None, html=None, keep=False, depth=100, timeout=20, output=None)

Choices: ['%ioDc %ioDc', '%i C', '%i C', '%ioDc %ioDc', '%i C']
Extract Goal: %ioDc %ioDc
Limit: 2
Result for Extract = [('%ioDc %ioDc', 100), ('%ioDc %ioDc', 100)]
Result for ExtractOne = ('%ioDc %ioDc', 100)

Choices: ['%i C', '%ioDc %ioDc', '%i C', 'XG %ioDc', '%i C']
Extract Goal: %ioDc %ioDc
Limit: 2
Result for Extract = [('%ioDc %ioDc', 100), ('XG %ioDc', 95)]
Result for ExtractOne = ('%ioDc %ioDc', 100)

Choices: ['yG yG', 'yG yG', 'y y', 'yG yG', 'y y']
Extract Goal: yG yG
Limit: 1
Result for Extract = [('yG yG', 100)]
Result for ExtractOne = ('yG yG', 100)

Choices: ['++++ ++++', '++++ ++++', '++++ ++++', '++++ ++++', '++++ ++++']
Extract Goal: ++++ ++++
Limit: 3
Result for Extract = [('++++ ++++', 0), ('++++ ++++', 0), ('++++ ++++', 0)]
Result for ExtractOne = ('++++ ++++', 0)

Choices: ['SO!T SO!T*', 'SO!T SO!T*', 'SO SO!T', 'SO SO!T', 'SO!T SO!T*']
Extract Goal: SO!T*e SO!T*N
Limit: 5
Result for Extract = [('SO!T*e SO!T*N', 100), ('SO!T*e SO!T*N', 100), ('SO!T*e SO!T*N', 100), ('SO!T*e
SO!T*N', 100), ('SO!T SO!T*', 95)]
Result for ExtractOne = ('SO!T SO!T*', 95)

Choices: ['qq a', 'a n', 'a n', 'qq a', 'qq a']
Extract Goal: qq a
Limit: 3
Result for Extract = [('qq a', 100), ('qq a', 100), ('qq a', 100)]
Result for ExtractOne = ('qq a', 100)
```

2. Bugs

As it mentioned in the previous section, one bug keeps showing in different functions. The "*fuzz.ratio*" and the "*fuzz.partial_ratio*" functions could handle the special symbols perfectly, while the "*fuzz.token_sort_ratio*", "*fuzz.token_set_ratio*" and "*process*" functions could not. The reason that raised as an assumption in the previous report is that the functions mess up the "space" with other special symbols. Every time they meet a special symbol, these functions treat it as space and apply specific operations on it. This will chop one "word" into apart and causes lower matching percentage.

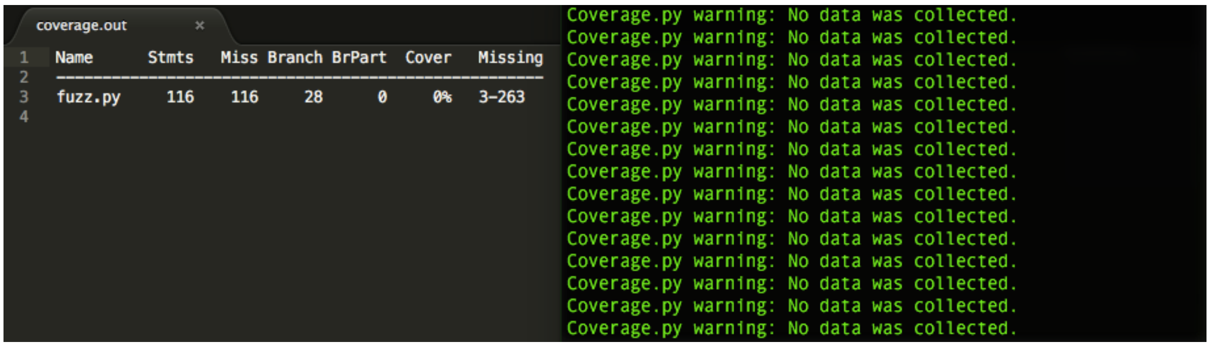
The sample results in the previous section support the assumption for sure. The third sample in the "*fuzz.token_set_ratio*" sample result only has one word for each string. However the function treats the slash (/) as the space, ignores the repeating and returns a 100 percent matching. The fourth sample in the "*process*" sample result has words formed all of special symbol. However the function treats them as space, and returns 0 percent matching for no words.

3. Randomtester

TSTL executes the SUT randomly using randomtester. This feature makes the code generates random infinite strings based on several characters during the test. This is actually a really good thing for the test since the test needs to run humongous kinds of strings in order to find the bug. The tests in this project did get benefits from this feature, otherwise the bug will not be found since only several cases will not show a general bug clearly. However, also because of the fact that randomtester executes the SUT randomly, it's sometimes hard to control the strings in order to get the correct form that is required. Therefore, the randomtester works pretty good in the bug finding part but causes some troubles for creating strings.

4. Coverage

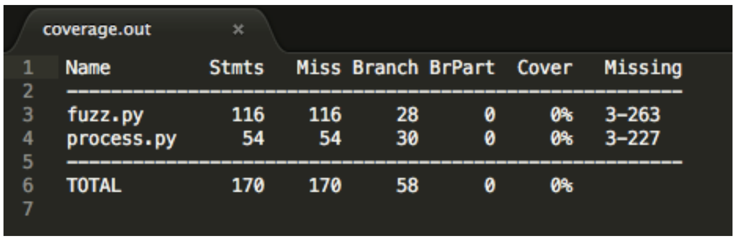
The code coverage is impossible to test under the condition I have. Might due to the version of Coverage.py installed in my Mac, the TSTL keep returning the warning "Coverage.py warning: No data was collected." Also, non randomtester nor coverage.out has the useful information for the code coverage evaluation. An image is listed below to show the result I got during the all of my tests.



```
coverage.out

Name      Stmts   Miss Branch BrPart  Cover   Missing
-----------------------------------------------------
fuzz.py     116    116     28      0     0%   3-263
```

```
Coverage.py warning: No data was collected.
Coverage.py warning: No data was collected.
Coverage.py warning: No data was collected.
Coverage.py warning: No data was collected.
Coverage.py warning: No data was collected.
Coverage.py warning: No data was collected.
Coverage.py warning: No data was collected.
Coverage.py warning: No data was collected.
Coverage.py warning: No data was collected.
Coverage.py warning: No data was collected.
Coverage.py warning: No data was collected.
Coverage.py warning: No data was collected.
Coverage.py warning: No data was collected.
Coverage.py warning: No data was collected.
Coverage.py warning: No data was collected.
```

The situation gets nothing better when the process.py is included. Both fuzz.py and process.py remain 0% coverage in the coverage.out and the warning won't disappear for a second. The screenshot of the situation that has both fuzz.py and process.py is listed below. This makes the coverage examination impossible.



```
Name          Stmts   Miss Branch BrPart  Cover   Missing
---------------------------------------------------------
fuzz.py         116    116     28      0     0%   3-263
process.py       54     54     30      0     0%   3-227
---------------------------------------------------------
TOTAL           170    170     58      0     0%
```

5. TSTL's Pros & Cons

TSTL uses randomtester to generate the testing cases for the functions. Just as it mentioned in the randomtester section, TSTL provides an easy way to run humongous different cases on functions in order to find the bugs. This is the biggest benefit TSTL provides in my opinion. Also, the grammar and logic of TSTL is easy to understand, and this provides an opportunity that everyone could do sort of bug digging using TSTL.

However TSTL does have several drawbacks. The biggest one will be it isn't robust enough to perform fully functional. The coverage problem mentioned in the coverage section makes the coverage examine impossible, which is a great pity in this project.

6. Summary

Testing the library using TSTL is a fun job to do. Coming up a way to create the required strings to test is a challenging but interesting work. I'm looking forward to the future work and willing to keep running tests using TSTL on some other programs in the future study.