Politechnika Śląska

Wydział Automatyki, Elektroniki i Informatyki

Vigenere cipher decryption, analysis tool

Author	Wiktor Utracki
Instructor	Dr. inż. Anna Gorawska
Year	2019/2020
Lab group	Thursday, 10:00 – 11:30
Deadline	16.01.2020

1. Project's topic

The main goal of the project is to help in analysis of a ciphertext ciphered using the Vigenere cipher. Program performs analysis of the given text, finding most probable key length and then most probable key letters for the given key length.

2. Analysis of the task

2.1. Data structures

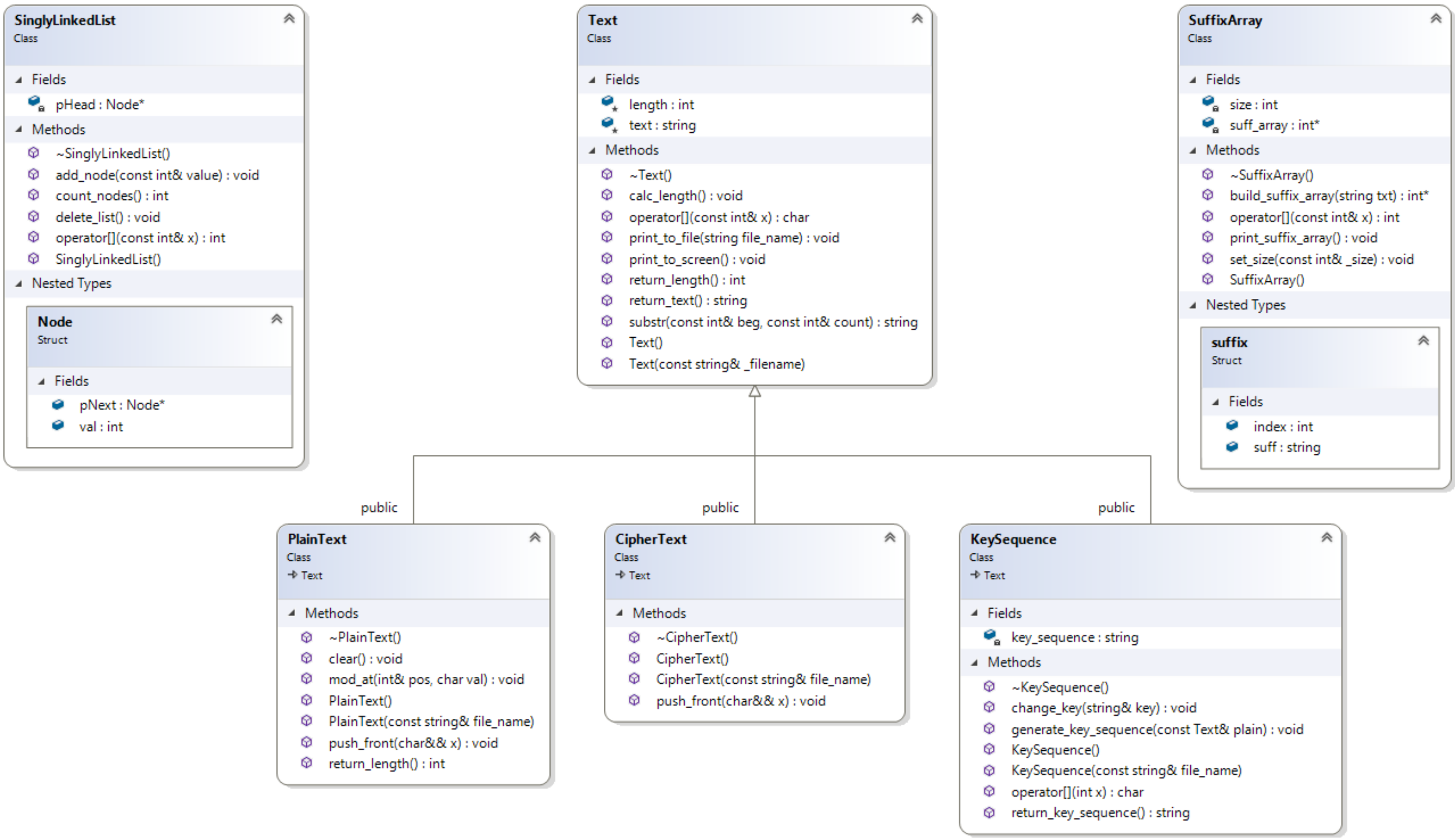

Figure 1 Class diagram.

Program uses 3 classes that are primarily used to store analyzed text: PlainText, CipherText and KeySequence. Names of those classes are self-explanatory. They all inherit from main class Text for they all represent text so they have a lot in common. The other two are SinglyLinkedList and SuffixArray. The last one is quite interesting. It stores suffix array built from ciphertext. Program utilizes also a UniqueList, which is a singly linked list but it stores only a single node of given value, and counts values that were put into the list.

2.2. Algorithms

2.2.1. Suffix array preparation

The suffix array created in SuffixArray class is sorted but contains all suffixes of the cipher and we can only use the patterns in text.

`prepare_suffix_array_for_pattern_search(SuffixArray *suffix_array, const Text &text)`

This function reduces the suffix array to patterns only and groups found patterns alphabetically. This function is quite straightforward: it takes first element as group symbol and second as current line, compares them and if they are a part of the same pattern it adds current one to the new array and moves the current line to the next element. It repeats this for as long as current is part of the same group pattern. If not it moves group symbol to the current symbol and current symbol to the group+1 and the process is repeated until whole array is sorted.
There is a problem if ciphertext has some clustered letters for example: “AGPEKKKKM” because the suffix array would normally have them recognized as patterns (“KK(…)”, “KKK(…)”, “KKKK(…)”) and would have take them into account during Kasiski’s test. Letter grouped in ciphertext like that can mean that some cluster of the same plaintext letters has been encrypted using the same letter of the key (if plaintext would be “OOOO(…)” and encryption key “EEEELA” the ciphertext would look like “SSSS(…)” this can happen but is poorly probable. In most cases such clusters are just a coincidence since Vigenere cipher is polyalphabetic cipher so we would like to exclude them from the suffix array and my function makes that.

```
Suffix array:
 84: "AKEJMHUWRKZS"
 39: "AKEFVLDNYXRQENWHUTYISDGLXPVIJYPUSKFLQTZHUHDZGAKEJMHUWRKZS"
 30: "DTVEPHRFGAKEFVLDNYXRQENWHUTYISDGLXPVIJYPUSKFLQTZHUHDZGAKEJMHUWRKZS"
  0: "DTKALYEIRIHGZGULNXMOHRVPHVOEEFDTVEPHRFGAKEFVLDNYXRQENWHUTYISDGLXPVIJYPUSKFLQTZHUHDZGAKEJMHUWRKZS"
 41: "EFVLDNYXRQENWHUTYISDGLXPVIJYPUSKFLQTZHUHDZGAKEJMHUWRKZS"
 28: "EFDTVEPHRFGAKEFVLDNYXRQENWHUTYISDGLXPVIJYPUSKFLQTZHUHDZGAKEJMHUWRKZS"
 83: "GAKEJMHUWRKZS"
 38: "GAKEFVLDNYXRQENWHUTYISDGLXPVIJYPUSKFLQTZHUHDZGAKEJMHUWRKZS"
 20: "HRVPHVOEEFDTVEPHRFGAKEFVLDNYXRQENWHUTYISDGLXPVIJYPUSKFLQTZHUHDZGAKEJMHUWRKZS"
 35: "HRFGAKEFVLDNYXRQENWHUTYISDGLXPVIJYPUSKFLQTZHUHDZGAKEJMHUWRKZS"
 54: "HUTYISDGLXPVIJYPUSKFLQTZHUHDZGAKEJMHUWRKZS"
 89: "HUWRKZS"
 78: "HUHDZGAKEJMHUWRKZS"
 85: "KEJMHUWRKZS"
 40: "KEFVLDNYXRQENWHUTYISDGLXPVIJYPUSKFLQTZHUHDZGAKEJMHUWRKZS"
 23: "PHVOEEFDTVEPHRFGAKEFVLDNYXRQENWHUTYISDGLXPVIJYPUSKFLQTZHUHDZGAKEJMHUWRKZS"
 34: "PHRFGAKEFVLDNYXRQENWHUTYISDGLXPVIJYPUSKFLQTZHUHDZGAKEJMHUWRKZS"
 12: "ZGULNXMOHRVPHVOEEFDTVEPHRFGAKEFVLDNYXRQENWHUTYISDGLXPVIJYPUSKFLQTZHUHDZGAKEJMHUWRKZS"
 82: "ZGAKEJMHUWRKZS"
```

Figure 2 Example of a suffix array with patterns filtered out and grouped. Numbers on the left are positions in ciphertext.

Program will print the sorted array if its length will be 35 or less.

2.2.2. Kasiski’s test

This is a way of guessing the key length. It analyses distances between patterns in previously prepared suffix array and finds their divisors. Then produces a table which contains counted divisors. Divisor with highest number is the most probable key length.

```
Kasiski method key length search results:
Divisor:    3 counted:    7
Divisor:    5 counted:    7
Divisor:    9 counted:    3
Divisor:   15 counted:    5
Divisor:    2 counted:    3
Divisor:    6 counted:    2
Divisor:   10 counted:    2
Divisor:   13 counted:    1
Divisor:    7 counted:    2
Divisor:    4 counted:    1
Divisor:    8 counted:    1
Divisor:   12 counted:    1
Divisor:   11 counted:    1
Divisor:   14 counted:    1
```

Figure 3 Kasiski method output. Since 3 and 5 are most probable we have to choose one that we think is the correct one. Analyzing other divisors I would say in this case 5 is slightly more probable.

Worth to mention, 2 is very often the most frequent divisor but we can discard it in most cases. Two-lettered key is very easy to break and is foolish to use so we can ignore it since chance that the actual key is of the length 2 is very small.
I have also limited the key length searched with this method to 25 since longer key is also normally not very probable and if there is no boundary value, this table for very long text (>1000 characters) can become unnecessarily long.

2.2.3. χ^2 method

It is a method of finding the most probable key of given length.
Since there is some frequency in which letters in English appears on average, we can determine how much the given text is close to being an average English text and chi squared method does exactly that. Vigenere cipher is polyalphabetic cipher so average frequency of each letter is going to be closer to 1/26 the better the key used will be. But we know that every *n-th* letter (*n* is the key length) is encrypted with the same letter and is just monoalphabetic cipher. We divide therefore the ciphertext in *n* cosets and then shift each coset left 26 times (trying every letter) each time calculating the χ^2 value. The shift witch the smallest value is most probably unencrypted (meaning the letter by which we shifted is the key letter).

Letter	Each collumn is one coset of the ciphertext				
A	13.87	1.0434	34.628	7.6525	4.9406
B	10.166	13.292	15.428	10.061	17.964
C	8.0852	5.5796	14.764	3.027	18.323
D	1.07	48.475	3.2894	8.4382	6.9113
E	63.68	11.197	5.4307	1.1331	13.065
F	14.51	46.949	14.993	16.011	7.6864
G	24.082	6.8834	11.985	8.1883	17.931
H	6.983	36.59	5.7849	42.124	0.9747
I	47.46	9.3881	19.209	9.9923	28.895
J	5.7122	19.182	29.558	16.717	10.054
K	23.363	23.993	13.637	5.996	27.397
L	24.368	3.7891	14.718	7.9506	8.4347
M	12.299	2.8164	14.902	3.1168	27.863
N	46.477	7.0446	7.5183	15.388	3.1461
O	5.1928	57.193	11.421	17.66	14.754
P	4.773	5.7664	21.366	10.593	7.6024
Q	1.7525	18.445	15.204	28.665	30.312
R	51.678	4.2846	0.54757	6.7767	33.595
S	3.9861	19.151	6.8383	8.1171	19.385
T	14.213	5.3702	13.394	0.79403	5.2756
U	31.144	56.746	14.812	6.884	4.8274
V	29.198	25.975	6.1629	7.0188	39.387
W	12.599	24.829	17.721	18.935	3.8549
X	18.462	21.894	5.0258	16.165	9.2752
Y	27.491	9.9941	10.733	14.766	16.589
Z	5.6044	6.0739	14.911	9.5364	28.953

Figure 4 χ^2 method output. We can see that the most probable key is DARTH (which in this case was true)

2.3. External specification

Program takes following switches:

- -c <file_name> is the input ciphertext-containing file
- -p <file_name> is the input plain text
- -k <file_name> is the input key

- -o <file_name> is the output file for decryption
- -q <file_name> is the output file for encryption

Combinations of plaintext, key [, output for encryption] or cipher [, output for decryption] are accepted.

2.4. Project's requirements

1) At least 4 classes *fulfilled*
2) Project split into separate source and header files *fulfilled*
3) Including the following topics from the laboratories: inheritance AND overloaded operators AND (plymorphism OR multiple inheritance) *fulfilled with polymorphism used in main function to write class name into output file*
4) Resource management in destructors *fulfilled in class SuffixArray*
5) No memory leaks proof generated by an appropriate tool *fulfilled – screenshot in Conclusions*

3. Conclusions

Program runs smoothly, performs all required functions and has no memory leaks:

```
==========================================================================
FINAL SUMMARY:

DUPLICATE ERROR COUNTS:

SUPPRESSIONS USED:

ERRORS FOUND:
      0 unique,     0 total unaddressable access(es)
      0 unique,     0 total uninitialized access(es)
      0 unique,     0 total invalid heap argument(s)
      0 unique,     0 total GDI usage error(s)
      0 unique,     0 total handle leak(s)
      0 unique,     0 total warning(s)
      0 unique,     0 total,      0 byte(s) of leak(s)
      0 unique,     0 total,      0 byte(s) of possible leak(s)
Details: D:\Programy\DrMemory\drmemory\logs\DrMemory-VigenereProject.exe.22460.000\results.txt
```

Figure 5 Report from Dr.Memory leaks detector.