AI lab – Ass4

Artyom Hvedjuk 317572204

Yaniv Raizner 300085677

# Part A

## Question 1

We have created a class called CypherText which has a constructor which get a key value and file to encrypt.

***EXAMPLE***:

**Original**: Alice was beginning to get very tired of sitting by her sister on the

**Encrypted**: lzpef mlj xfopaapao gv ofg yfcb gpcfn vd jpggpao xb rfc jpjgfc va grf

**Key**: nyrfcetpxsuawdgikhqjzombvl

***CODE*:**

class CeaserCypher {

string key;

ifstream textFile;

string originalFile;

string outputFile;

std::ofstream ofs;

public:

CeaserCypher(string& key,const string& pathToFile,const string& pathToOutput);

void createCypheredText();

};

CeaserCypher::CeaserCypher(string& key,const string& pathToFile,const string& pathToOutput){

this->key = key;

this->originalFile=pathToFile;

this->outputFile=pathToOutput;

ofs.open(outputFile, std::ofstream::out | std::ofstream::trunc);

ofs.close();

}

void CeaserCypher::createCypheredText(){

textFile.open(originalFile);

ofs.open(outputFile);

string line;

string encryptedLine;

while(getline(textFile, line)){

encryptedLine = line;

for (int i = 0; i < line.length(); i++)

{

char temp = line[i];

if(temp <= 'Z' && temp >='A')

{

temp = temp - ('Z'-'z');

}

if ((temp >= 'a' && temp <= 'z'))

{

encryptedLine[i] = key.find(temp) + 'a';

}

}

this->ofs << encryptedLine << endl;

}

textFile.close();

ofs.close();

}

***Running example:***

string temp = "nyrfcetpxsuawdgikhqjzombvl";

CeaserCypher cc(temp, "c:\\temp\\text\\input.txt", "c:\\temp\\text\\encryptedText.txt");

cc.createCypheredText();

## Question 2

First we have implemented a text parser which goes over the frequency given text file and modeled all the information to the local memory.

We have created a dictionary mapping (Hash Table) for each kind of frequency table (8 – for mono, bigram and trigram) – The modeling is enabling us to compare between the frequencies tables to the results of our algorithm.

The main core of the solution is the function which calculates the fitness. As of all the genetic algorithms we have used until now we have created a population in which every citizen is a permutation of the 26 alphabetic string. (Like: “nyrfcetpxsuawdgikhqjzombvl”). In other words we have created a population of random keys for the given text.

We are using the same methodology we have used till now – creating a random population, calculating fitness for each citizen (which we would describe later), mating and mutating. Though the only change we have added to the generic genetic algorithm is checking rather the new mutation (after mating or mutating) is valid, if not we are fixing it using a new function we have added to the engine.

So regarding the fitness calculation – the basic idea is to use the key we got in hand (the citizen itself) for finding the real letter behind .We have gone through all the encrypted text and counted all the occurrences according to the occurrences we have in the freq.txt. Now we have 2 modeled information – one is the true statistic information and the second is the information we have counted in the encrypted text – Hence now we have a problem comparing those two dictionaries since we do not have same letters, but guess what – we have the key!

We would compare (using the key) the occurrences of each letter (as well as 2 letters and 3 letters) we have found in our encrypted text to the one we have from the source. As more precise as higher fitness.

***CODE*:**

void init\_population(ga\_vector &population,

ga\_vector &buffer )

{

int tsize = GA\_TARGET.size();

for (int i=0; i<GA\_POPSIZE; i++) {

ga\_struct citizen;

citizen.age = 0;

citizen.fitness = 0;

citizen.str.erase();

citizen.str = GA\_TARGET;

for (int j=0; j<tsize; j++)

{

int tempFirst = rand() % tsize;

int tempSecond = rand() % tsize;

char temp = citizen.str[tempFirst];

citizen.str[tempFirst] = citizen.str[tempSecond];

citizen.str[tempSecond] = temp;

}

population.push\_back(citizen);

}

}

int Stats::calculateFitness(string key){

int fitnsess = 0;

for(myMap::iterator i = occurencies.begin(); i != occurencies.end(); i++) {

// iterator->first = key

// iterator->second = value

// Repeat if you also want to iterate through the second map.

string currentEnctypetdLetters = (i->first);

for (int i = 0; i < currentEnctypetdLetters.size();   
i++)

{

if ( (currentEnctypetdLetters[i] >= 'a') && (currentEnctypetdLetters[i] <= 'z'))

{

currentEnctypetdLetters[i] = key[currentEnctypetdLetters[i] - 'a'];

}

}

switch (currentEnctypetdLetters.size())

{

case 1:

if (parser->lettersStatistics.find(currentEnctypetdLetters) != parser->lettersStatistics.end())

{

if (abs( i->second - parser-> lettersStatistics[currentEnctypetdLetters]) < 0.5)

{

fitnsess += 1;

}

if (abs( i->second - parser->lettersStatistics[currentEnctypetdLetters]) < 0.2)

{

fitnsess += 5;

}

}

break;

case 2:

if (parser->bigramsFreIncludingSpaces.find(currentEnctypetdLetters) != parser->bigramsFreIncludingSpaces.end())

{

if (abs( i->second - parser->bigramsFreIncludingSpaces[currentEnctypetdLetters]) < 1)

{

fitnsess += 1;

}

if (abs( i->second - parser->bigramsFreIncludingSpaces[currentEnctypetdLetters]) < 0.5)

{

fitnsess += 2;

}

}

break;

case 3:

if (parser->triagrmsFreIncludingSpaces.find(currentEnctypetdLetters) != parser->triagrmsFreIncludingSpaces.end())

{

if (abs( i->second - parser->triagrmsFreIncludingSpaces[currentEnctypetdLetters]) < 1)

{

fitnsess += 1;

}

if (abs( i->second - parser->triagrmsFreIncludingSpaces[currentEnctypetdLetters]) < 0.5)

{

fitnsess += 2;

}

}

break;

default:

break;

}

}

return fitnsess;

}

Stats::Stats(const string& pathToEncryptedFile, ParseText& parser){

this->parser = &parser;

this->parser->parseTextLunch("C:\\temp\\freq.txt");

this->parser->createStatistics("c:\\temp\\text\\inputToStatistcs.txt"); // the parsing of the real text - artyom BOM!!!

this->counter=0;

ifstream textFile;

string line;

textFile.open(pathToEncryptedFile);

while(getline(textFile, line)){

for(int i=0;i<line.length();i++){

string oneLetter = line.substr(i,1);

char tempChar = oneLetter.c\_str()[0];

if((tempChar >='a' && tempChar <='z') || tempChar == ' ' ){

occurencies[oneLetter]=occurencies[oneLetter]+1;

counter++;

}

string twoLetters ;

if(!(i>line.length()-1)){

char temp1 = ((line.substr(i,1)).c\_str())[0];

char temp2 = ((line.substr(i+1,1)).c\_str())[0];

if (( (temp1 >= 'a' && temp1 <= 'z') || temp1 == ' ' ) && ((temp2 >= 'a' && temp2 <= 'z') || temp2 == ' ' )){

twoLetters= line.substr(i,2);

if ((parser.bigramsFreIncludingSpaces.find(twoLetters) != parser.bigramsFreIncludingSpaces.end()))

{

occurencies[twoLetters] = occurencies[twoLetters]+1;

}

}

}

string threeLetters;

if(!(i>line.length()-2)){

char temp1 = ((line.substr(i,1)).c\_str())[0];

char temp2 = ((line.substr(i+1,1)).c\_str())[0];

char temp3 = ((line.substr(i+2,1)).c\_str())[0];

if (( (temp1 >= 'a' && temp1 <= 'z') || temp1 == ' ' ) && ((temp2 >= 'a' && temp2 <= 'z') || temp2 == ' ' )

&& ((temp3 >= 'a' && temp3 <= 'z') || temp3 == ' ' )){

threeLetters = line.substr(i,3);

if ((parser.triagrmsFreIncludingSpaces.find(threeLetters) != parser.triagrmsFreIncludingSpaces.end())){

occurencies[threeLetters]=occurencies[threeLetters]+1;

}

}

}

occurencies[twoLetters] = occurencies[twoLetters]+1;

}

}

for(map<string,double>::iterator iterator = occurencies.begin(); iterator != occurencies.end(); iterator++) {

if(iterator->first.length()==1){

iterator->second=(iterator->second/(double)counter)\*100;

}

if(iterator->first.length()==2){

iterator->second=(iterator->second/(double)(counter-1))\*100;

}

if(iterator->first.length()==3){

iterator->second=(iterator->second/(double)(counter-2))\*100;

/\*}\*/

}

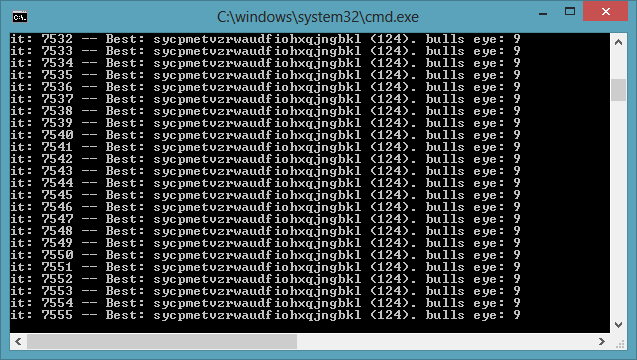
}

***Analysis*:**

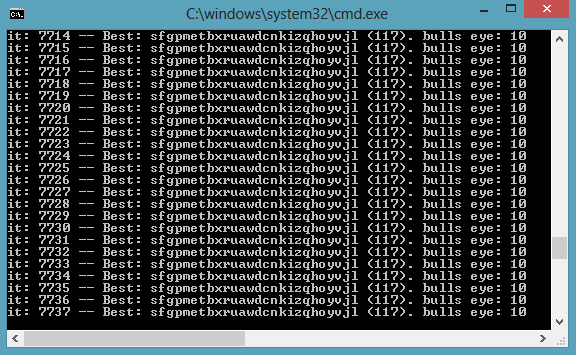
We used basically the Alice In Wonderland story.

**Just to make it clear -**  In all the screenshots bellow the “bulls eye” statitstic are shown only for educational purposes and not taken into account when calculating the fitness.

The first run we have used default inputs (No aging, random selection mating) and population size of 500. The results were poor – After many iterations we didn’t get the real key that we have used:



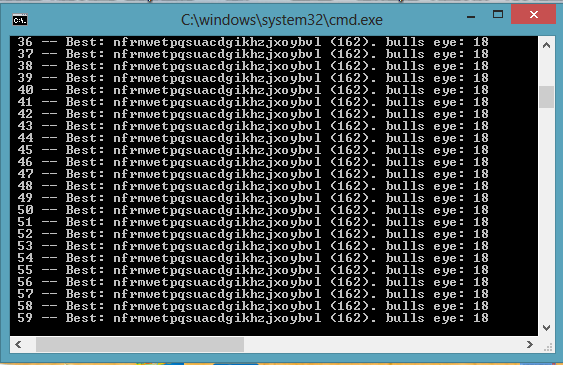
So we started to think what can we do? First thing we wanted to do is checking just monograms occurrences:



Not just it didn’t help – it made it worse in most of the runs. So what can we do next? What’s wrong? So have tried using bigrams only and trigrams only or other combination, though it doesn’t make any sense it’s indeed really didn’t help.

So we have asked ourselves what could went wrong. Maybe our statistics calculation is different from the reference we are using – In other words we are taking or not taking into account some factors in our calculation. If so – we would try to make our calculation instead of the reference – but using the same story for this purpose is kind of manipulating the system though it make a lot of sense to some of us (We kind of arguing about this, but never mind).

We have created our own statistics information’s using the story “War and Peace”. Then we have ran our engine once again, though we couldn’t get into the full solution we could decode 18 out 26 letters in the key which seems enough in order to read and understand the original message.

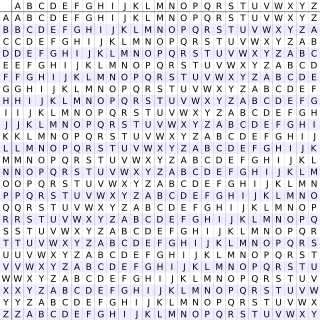


In other words – though we couldn’t get the full solution we are close enough to decrypt the encrypted story.

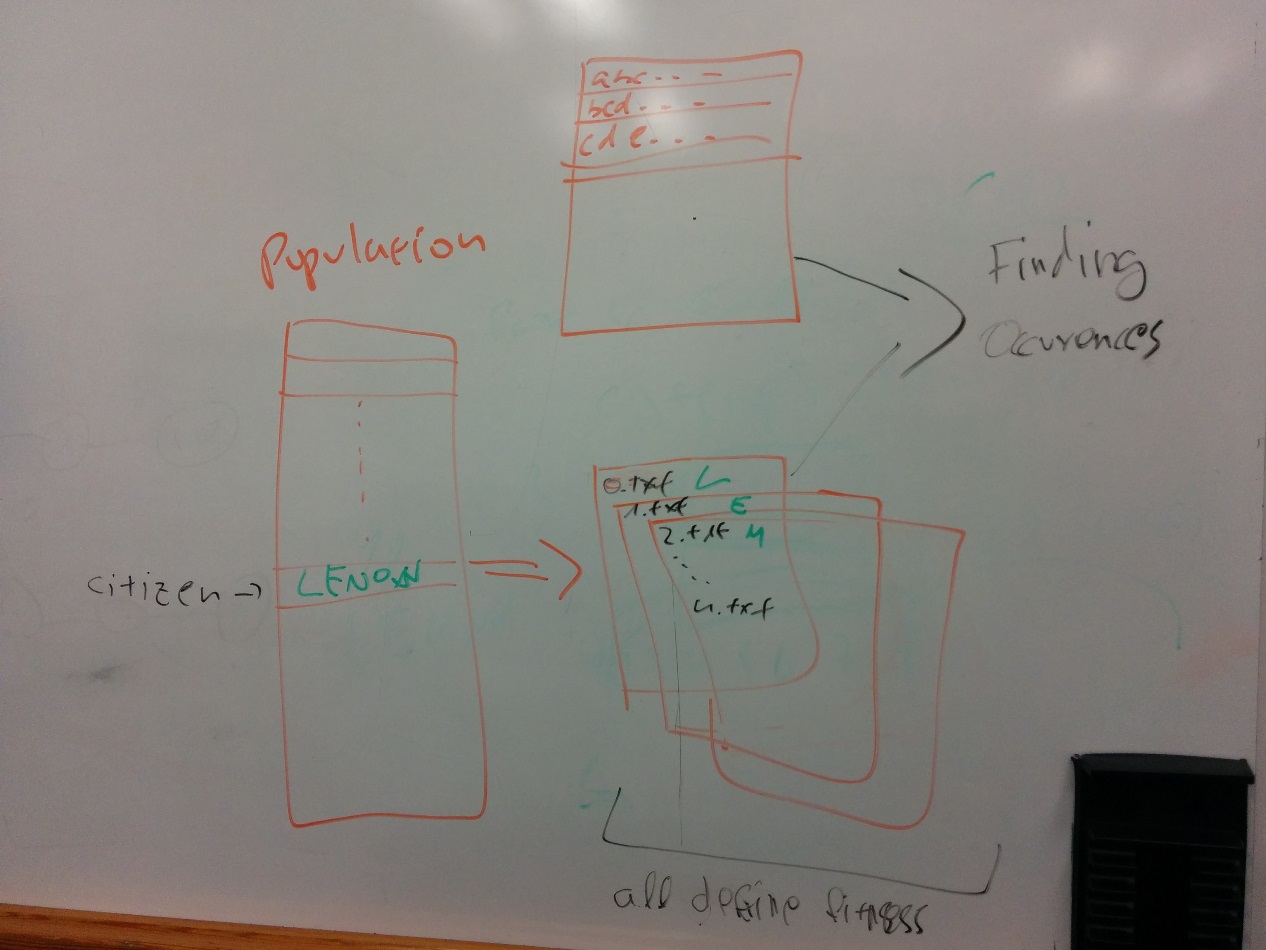
# Part B

## Question

As first approach to this problem we tried to create a population in which each citizen could be any size of random key, for each key we knew the number of letters, means cycle’s size. From the cycle size we could go back to the Ceaser cipher problem we have already solved – but running it cycle-size times. So what we did basically is shuffling a key (size and permutation), then divided it to few problem we already know how to solve (by the number of the cycle size) – so after we have one citizen – we need to ask what its size in order to know to how many files we should divide the original text. Then we are dividing it to many files and on each files we would run our older solution, just this time we do not need to guess our whole key-population since we have the ‘tabula recta’ table:



Then for each key (actually citizen in the previous solution) we are calculating the occurrences according to the reference we have. Then summing all fitness of the solutions and dividing it by the maximum fitness possible multiplying the size of the problem (In other words normalizing the solution to fit any kind of cycle).

And the design:

But after many trials to run it and not reaching any result we were very frustrated. Then we’ve started to think about a new solution , exploring the world wide web and came to an idea of calculating the GCD of index appearances of same substrings. After getting the GCD