UNIVERSITY OF ONTARIO INSTITUTE OF TECHNOLOGY

SOFE 3770U: Design and Analysis of Algorithms
Project Report

Comparison of Brute Force and Particle Swarm Optimization in Cryptanalysis of Vigenere Cipher

25th November 2018

Allan Santosh - 100557518
Thanushan Rameswaran - 100462146
Dhanushga Lionel - 100616831
Kaamran Minhas - 100593277
Kelvin Leoncius - 100592582

Table of Contents

Introduction

Encryption is the process of converting a body of text or symbols into one where the original information or message is not recognizable. Encryption is commonly used to encode messages or bodies of text. In the process of encrypting a message, a key is used that can later be used to decrypt the encrypted message to reveal the original text.

Vigenere Cipher

One of the simplest and widely known encryption techniques is the Caesar cipher. The encryption technique is to shift each letter in the plain text a fixed number of places to the left or right. For example if the key is a right shift of 4 places, A would become E and X would become B.

In the following example of a Caesar cypher encryption, the message “welcometothejungle” will be encrypted with a right shift of 4. It is then decrypted by shifting the same number of places in the opposite direction.

Plain text	W	E	L	C	O	M	E	T	O	T	H	E	J	U	N	G	L	E
Shifted 4 right	A	I	P	G	S	Q	I	X	S	X	L	I	N	Y	R	K	P	I
Shifted 4 left	W	E	L	C	O	M	E	T	O	T	H	E	J	U	N	G	L	E

The Caesar cypher is easily broken as the encryption method is not very complex.

The vigenere cipher is a more complex cipher that uses a key as part of the encryption process. To encrypt a message using a vigenere cipher. The steps to encrypting a message using the vigenere cipher are as follows:

First you form a plain text key. This key is typically a word or a series of words. The key is then repeated until it matches the length of the plain text to be encrypted. The alphabet is the set of characters used in the message and key, typically the letters of the English alphabet, and is numbered starting at 0. So A-Z will have the corresponding numbers from 0-25. To encrypt a letter with the key, you would take the letter to encrypt and the corresponding key letter. You then add the position in the alphabet corresponding to both letters and take mod 26 of the result. The resulting number will be the encrypted letter position in the alphabet. To decrypt a letter with the key, you would subtract the integer index of the key and subtract it from the encrypted letter's integer index and find mod 26. The result will be the integer index of the decrypted letter.

Example: Plain text: LOCKUPTHECLOCKS KEY: STAR

Plain text	L	O	C	K	U	P	T	H	E	C	L	O	C	K	S
Index	11	14	2	10	20	15	19	7	4	2	11	14	2	10	18
KEY	S	T	A	R	S	T	A	R	S	T	A	R	S	T	A
Key index	18	19	0	17	18	19	0	17	18	19	0	17	18	19	0
Plain text + key index mod 26 = encrypted letter index	11+18 mod26 = 3	14+19 mod 26 = 7	2+0 mod 26 = 2	10+17 mod26 = 1	20 +18 mod26 =12	15+19 mod 26 = 8	19	24	22	21	11	5	20	3	18
Encrypted message	D	H	C	B	M	I	T	Y	W	V	L	F	U	D	S
(Encrypted index - key index) mod 26	(3-18) mod 26 = 11	(7-19) mod 26 = 14	2	10	20	15	19	7	4	2	11	14	2	10	18
Decrypted message	L	O	C	K	U	P	T	H	E	C	L	O	C	K	S

.The strengths in the Vigenere cipher vs other ciphers such as the Ceaser cipher is that the plaintext frequency is hidden. This is because every letter in the plain text is encrypted by a different key character depending on its position in the text. For instance in the above example, the letter L is first encrypted by the key letter S and later is encrypted by the key letter A. This makes it invulnerable to the use of a frequency analysis which could simply deduce the plaintext from the frequency of the characters of the cipher text.

In this project, we will be exploring two methods, The brute force method and Particle Swarm Optimization method and comparing the efficiencies towards each other in relation to decoding a vigenere cypher problem.

Brute force Method:

In our cryptanalysis of the Vigenere cipher we first started with a brute force method. In the brute force method we had to first find the length of the key used. To do this, we first analyzed the ciphertext for instances of repeating substrings starting at a length of 2 up to a length of 9. Repeating substrings in the ciphertext is significant because they could indicate where the same letters of plain text was encrypted with the same letters of the subkey. That is, since we are encrypting a body of plain text which contains actual words, there is a chance that the same word is encrypted by the same subkey more than once.

For example:

V	I	G	E	N	E	R	E	W	A	S	A	T	R	U	E	V	I	G	E	N	E	R	Y
A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
V	J	I	H	N	F	T	H	W	B	U	D	T	S	W	H	V	J	I	H	N	F	T	B

We see that in the plaintext the substring vigener appears twice and that the same key letters happens to be used to encrypt ‘VIGENERE’.

The distance between the repeating substring is 16. Assuming that the repeating text represents the same plaintext segments, the key will be a factor of 16 in length. That is 1, 2, 4, 8, or 16.

We note here that the longer the encrypted message is, the more accurate this test is because there is a higher probability of occurrences of repeating text.

Once all of the instances of repeating substrings were found, we found the distance between them. The list of distances was expanded into a list of factors and the frequency of each factor was found. The factors with the highest count are the most likely lengths of the key.

With the discovered key length, we used brute force to find every possible permutation of the key length with the english alphabet. Each of these keys was used to decrypt the message which was referenced against a dictionary to detect if the text contained english words or just gibberish. Once we found a decrypted message that contained actual english words, we know that the key used to decrypt it, is the actual key.

Particle Swarm Optimization (PSO) Introduction:

Particle Swarm Optimization which we will refer to as PSO for the rest of this text, is a population based metaheuristic stochastic algorithm. PSO was developed in 1995 by Dr. Eberhart and Dr. Kennedy, who were inspired by flocks of birds, and schools of fish.

An explanation of PSO can best explained by the analogy of birds. A group of birds will work together to find hidden food. Whichever bird is closest lets the other birds of the group know where they are. If another circling bird is closer, it lets out a louder chirp, and all the other birds go there. This keeps happening till all the birds are at the location where the food is.

PSO is a very useful algorithm. It is fast algorithm, and is cheaper because it does not use as much data compared to other algorithm.

System requirement

The project was tested on the computer having the following specifications:

Operating System: Windows 10 Pro 64-bit
Ram: 32 GB, 2133 MHz
CPU: Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz
Hard Drive: Samsung SSD 960 EVO 250GB
Compiler: Eclipse Java EE IDE for Web Developers. Version: 2018-09 (4.9.0) Build id: 20180917-1800, Microsoft Visual Studio Community Edition
Programming Language: Java, C++

Methodology

PSO Methodology:

PSO works by initializing a population of random solutions, and searches for the optimal solution by iteratively improving a candidate solution. During a number of iterations, a group of variables have their values adjusted closer to the member whose value is closest to the target at any given moment.

Each particle keeps track of position with the best solution (fitness) it has achieved so far. This fitness value is stored. This value is called pBest. Another "best" value that is tracked by the particle swarm optimizer is the best value, obtained so far by any particle in the neighbors of the particle. This location is called lBest (local best). when a particle takes all the population as its topological neighbors, the best value is a global best and is called gBest.

At each iteration the PSO changes the velocity of the individual particles towards the pBest the lBest. The acceleration of the particle has a random weight term, with separate random numbers being generated for acceleration toward pBest and lBest locations.

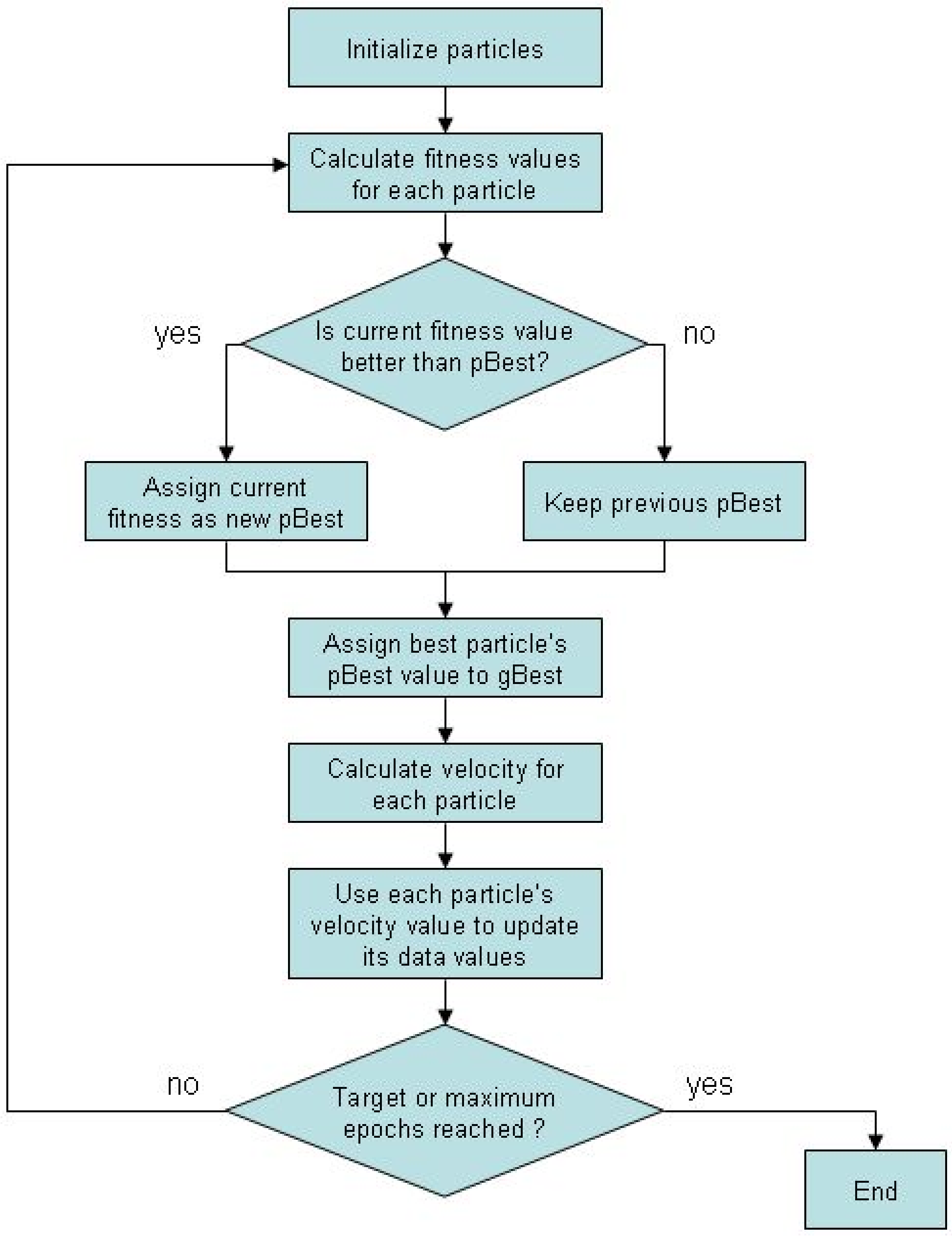

Initialize particles
Calculate fitness values for each particle
Is current fitness value better than pBest?
yes
no
Assign current fitness as new pBest
Keep previous pBest
Assign best particle's pBest value to gBest
Calculate velocity for each particle
Use each particle's velocity value to update its data values
Target or maximum epochs reached ?
no
yes
End

Figure 1: Flow Chart for PSO

Brute Force Methodology:

In our cryptanalysis of the Vigenere cipher we first started with a brute force method. In the brute force method we had to first find the length of the key used. To do this, we first analyzed the ciphertext for instances of repeating substrings starting at a length of 2 up to a length of 9. Repeating substrings in the ciphertext is significant because they could indicate where the same letters of plain text was encrypted with the same letters of the subkey. That is, since we are encrypting a body of plain text which contains actual words, there is a chance that the same word is encrypted by the same subkey more than once.

For example:

V	I	G	E	N	E	R	E	W	A	S	A	T	R	U	E	V	I	G	E	N	E	R	Y
A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
V	J	I	H	N	F	T	H	W	B	U	D	T	S	W	H	V	J	I	H	N	F	T	B

We see that in the plaintext the substring vigener appears twice and that the same key letters happens to be used to encrypt ‘VIGENERE’.

The distance between the repeating substring is 16. Assuming that the repeating text represents the same plaintext segments, the key will be a factor of 16 in length. That is 1, 2, 4, 8, or 16.

We note here that the longer the encrypted message is, the more accurate this test is because there is a higher probability of occurrences of repeating text.

Once all of the instances of repeating substrings were found, we found the distance between them. The list of distances was expanded into a list of factors and the frequency of each factor was found. The factors with the highest count are the most likely lengths of the key.

Pseudocode:

Brute Force:

Input : Encrypted plain text
Output :De crypted plain text

```
Arr spacingl
Int count
Int spacing
For (i = 0 , i to input.size; i++)
        Sub = input.substring(i,0)
        For (int j = i; i to input.size; j ++)
                Sub2 = input.substring(i,)
                If (sub == sub2)
                Count ++
            Spacing. Add (sub.get(indexi) - sub2.get(indexj))

Keylength =  spacing.findMax()
Count = 0
Keyfound = false

For (i = 0 , i to input.size; i++)
      For (int j = 0; i to input.size; j ++) {
      Sub = input.substring(i,j)
      If (Dictionary.contains(sub) {
            Count ++
If (count > average. count) {
      Keyfound =-true
```

PSO:

```
For each particle
{
  Initialize particle
}

Do until maximum iterations or minimum error criteria
{
  For each particle
  {
    Calculate Data fitness value
    If the fitness value is better than pBest
    {
      Set pBest = current fitness value
    }
    If pBest is better than gBest
    {
      Set gBest = pBest
    }
  }

  For each particle
  {
    Calculate particle Velocity
    Use gBest and Velocity to update particle Data
  }
```

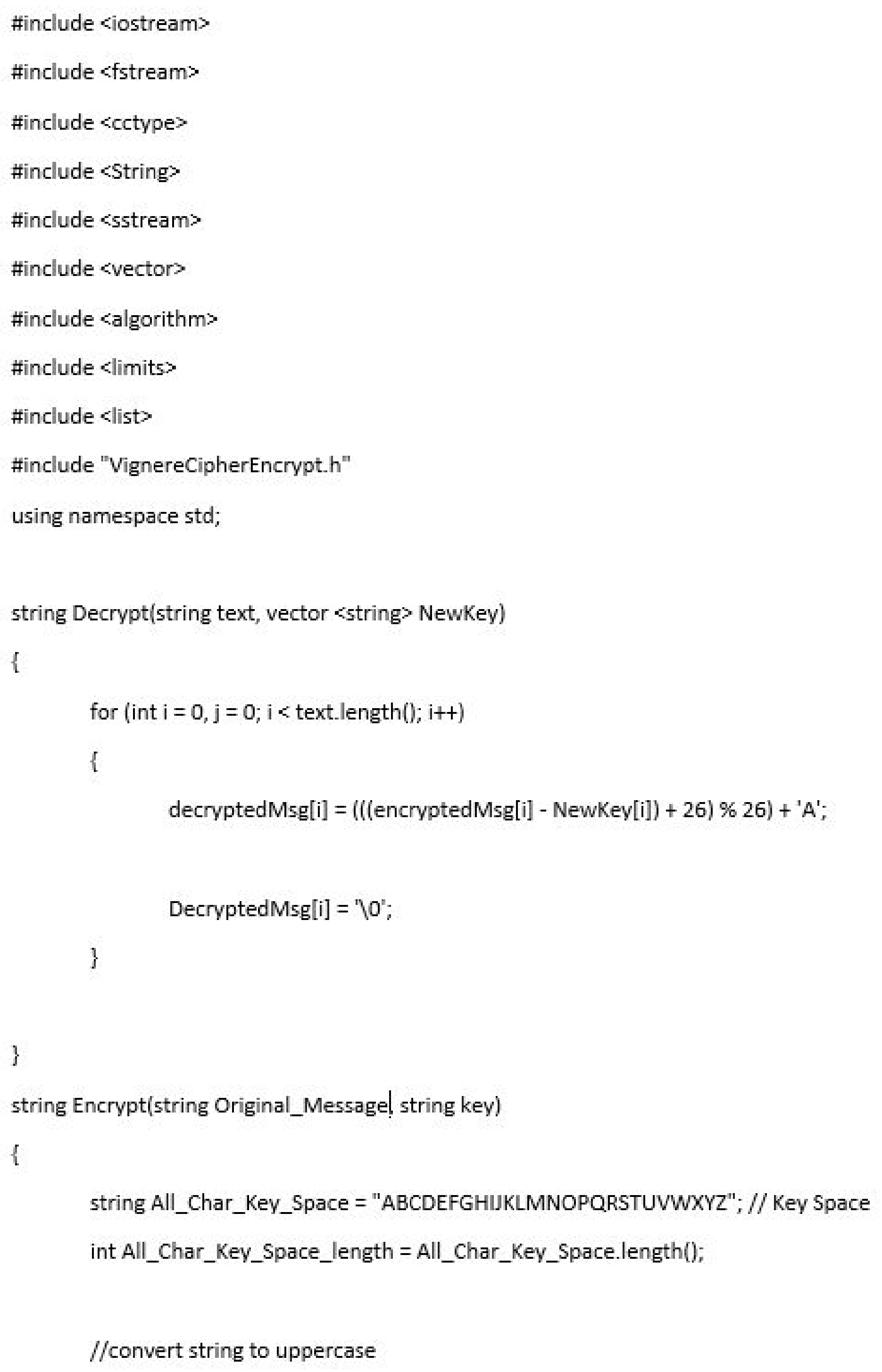
Detailed Code

Brute Force :

```
package BruteForce;

import java.io.*;
import java.util.*;
import java.util.Map.Entry;

    // This is the code to to brute force a Vigenere Cipher.

public class BruteForce
{

    // Initialize some variables needed.

    public static ArrayList<String> dictionary = new ArrayList <String>();
    public static String plain_text, encrypted_text = "";
    public static int key_length;
    public static boolean key_found = false;

    // First we generate plain text from a text file.

    public static void generate_plain_text(String filename, int size) throws IOException {

        System.out.println("Reading file " + filename + " and generating plaintext of size " +
size + ".\n");

        FileReader filereader = new FileReader(filename);
        BufferedReader bufferedreader = new BufferedReader(filereader);
        StringBuilder stringbuilder = new StringBuilder();
       String line = bufferedreader.readLine();

    // Add all the contents in the file and put it in a String.

              while (line != null) {
               stringbuilder.append(line);
              line = bufferedreader.readLine();
              }

              plain_text =  stringbuilder.toString();
              plain_text = plain_text.replaceAll("[^a-zA-Z]", "");
              plain_text = plain_text.toLowerCase();
              plain_text = plain_text.substring(0, size);

              bufferedreader.close();
              System.out.println("This is the plain text:\n\n"+ plain_text + "\n");

    // Filter out everything except characters and store it as plain text.

    }

    public static void populate_dictionary (String filename) throws FileNotFoundException {
        // A dictionary is imported in order to check if the decrypted text is in fact
        // in English Language.


        System.out.println("Populating the dictionary ...\n");
        File f = new File (filename);
        Scanner s = new Scanner (f);

        while (s.hasNextLine()) {
                dictionary.add(s.nextLine());
        }

        System.out.println("Populated the dictionary.\n");

    }

    // After the text is encrypted, the plain text needs to be decrypted. First step
    // is to find the length of the key used.

    public static void find_key_length (String text) {

        System.out.println("Cracking the code .....");

        System.out.println("Figuring out the key length .....\n");

        ArrayList <Integer> spacings = new ArrayList<Integer> ();
        ArrayList <String> substring_text = new ArrayList<String> ();

    // A for loop is used to iterate through substrings of the encrypted text.
    // Starting with sizes 2 to 9 (taken as average length of words in English Language),
    // each substring is checked to see if there is any repetitions. If repetitions are found,
    // then mark down the spaces between repetitions.

        for (int sub_string_length = 2; sub_string_length < 9; sub_string_length ++) {

        for (int i = 0; i < text.length()- sub_string_length -1 ; i++) {

                String sub1 = text.substring(i, i+ sub_string_length);
                int start, finish;
                start = i;

                for (int j = i; j < text.length()- sub_string_length -1; j++) {

                String sub2 = text.substring(j+1 , j+ sub_string_length +1);

                        if (sub1.equalsIgnoreCase(sub2)) {
                                substring_text.add(sub1);
                                finish = j+1;
                                spacings.add(finish - start);
                                break;
                        }
                }
        }
    }


        // For easier view a grid is drawn.
        String grid [][] = new String [substring_text.size()][7];


        for (String[] row: grid) {
              Arrays.fill(row, "-");}


        // The spacings between repetitions are checked to see whether or not they are the
        // factors of the key size.


        for (int i = 0 ; i < substring_text.size(); i++) {
                      grid [i][0] = substring_text.get(i);
                      grid [i][1] = Integer.toString(spacings.get(i));

                      if (spacings.get(i)%3 == 0) {
                             grid[i][2] = "X";
                      }
                      if (spacings.get(i)%4 == 0) {
                             grid[i][3] = "X";
                      }
                      if (spacings.get(i)%5 == 0) {
                             grid[i][4] = "X";
                      }
                      if (spacings.get(i)%7 == 0) {
                             grid[i][5] = "X";
                      }
                      if (spacings.get(i)%11 == 0) {
                             grid[i][6] = "X";
                      }

        }


System.out.println("--------------------------------------------------------------------------
----");
        System.out.println(" SubString     Spacing        3      4      5      7      11");

System.out.println("--------------------------------------------------------------------------
----");

        for (int i = 0 ; i < substring_text.size() ; i++) {
               for (int j = 0 ; j < 7 ; j++) {
                      System.out.printf("%10s",grid[i][j]);
                      System.out.print(" ");
               }
               System.out.println();
        }

        System.out.println();

        // Count number of X's in each column to find the greatest. The column with the
        // greatest number of X's found is the size of the key.

        int count_3 = 0, count_4 = 0, count_5 = 0, count_7 = 0, count_11 = 0;

        for (int i = 0; i < substring_text.size(); i++ ) {
               if (grid[i][2].equals("X")) {
                      count_3++;
```

```
                }
                if (grid[i][3].equals("X")) {
                       count_4++;
                }
                if (grid[i][4].equals("X")) {
                       count_5++;
                }
                if (grid[i][5].equals("X")) {
                       count_7++;
                }
                if (grid[i][6].equals("X")) {
                       count_11++;
                }
        }

        // Checking which count is the greatest and placing the key length into a variable for
        // brute forcing.

        HashMap<Integer, Integer> count_list = new HashMap<>();
        count_list.put(3,count_3);
        count_list.put(4,count_4);
        count_list.put(5,count_5);
        count_list.put(7,count_7);
        count_list.put(11,count_11);

       for (Entry<Integer, Integer> entry : count_list.entrySet()) {  // Itrate through
hashmap
               if (entry.getValue()==(Collections.max(count_list.values()))) {
               key_length = entry.getKey();
               }}

        System.out.println("Possible Key Length of the encrypted text is " + key_length +
".\n");

    }

    // In this method brute_force is used. Using permutations of the english
    // alphabets, every single possible key is checked and matched against a dictionary
    // to check whether the decrypted text is the actual key.


    public static void brute_force (int key_size, String txt) {

        char[] alphabets = {'a','b','c','d','e','f','g','h','i','j','k','l','m',
                       'n','o','p','q','r','s','t','u','v','w','x','y','z'};

       if (key_size == 0 && key_found == false)  {

                dictionary_check(txt , plain_text);
               return;
       }


       // Recursion is used to call the brute force function again in order to crack the key.


       for (int i = 0; i < alphabets.length; ++i)
       {
               if (key_found == false) {
                String new_txt = txt + alphabets[i];
                brute_force(key_size - 1, new_txt);
                }
                else {
                       break;
                }
        }

    }

    // This method decrypts the message given the encrypted text and a key.
    // In order to decrypt a text, the value of an encrypted text letter at
    // index x is subtracted with the value of the key at index x and modded with 26
    // in order to get the plain text back.

    public static String decrypt (String text, String key) {

        String decrypted = "";

        for (int i = 0, j = 0; i < text.length(); i++)
        {
               decrypted = decrypted + (char) ((text.charAt(i) - key.charAt(j) + 26) % 26 +
97);
               j = ++j % key.length();
        }

        return decrypted;
    }

    // This method encrypts the message given the plain text and a key.
    // In order to encrypt a text, the value of an plain text letter at
    // index x is added with the value of the key at index x and modded with 26
    // in order to get the encrypted text.

    public static void encrypt (String text, String key) {

         System.out.println("Enrcypting the plaintext. \n");

         for (int i = 0, j = 0; i < text.length(); i++)
        {

                encrypted_text = encrypted_text + (char) (((text.charAt(i) - 97 ) +
(key.charAt(j)) - 97) % 26 + 97);
               j = ++j % key.length();
        }

        System.out.println("This is the encrypted plain text: \n\n" + encrypted_text + "\n") ;

    }

    // Check to see if any word is found in the dictionary. If a word is contained in the
dictionary,
    // it is most likely to be the correct key used in the decryption.

    public static void dictionary_check (String key, String txt) {


                String decrypted =  decrypt(encrypted_text, key);
                if (decrypted.equals(txt) //|| dictionary.contains(decrypted)
                         ){
                       System.out.println("The key has been found: " + key + "\n");
                       System.out.println("Decrypted succesfully!\n \n" + decrypted + "\n");

                       key_found = true;
                }

                // A method to convert time taken into hours, mins, seconds, ms.
    }

    public static void time_taken(long start , long finish){

        long total = finish - start;

        long days = total / 86400000;
        total = total % 86400000;

        long hours = total / 3600000;
        total = total % 3600000;

        long mins = total / 60000;
        total = total % 60000;

        long sec = total / 1000;
        total = total % 1000;

        long ms = total;

        System.out.printf(
              " Total time taken to run the program is %d days, %d hours, %d minutes, %d
seconds, %d ms. %n",
              days, hours, mins, sec, ms);

    }



       public static void main(String[] args) throws IOException

       {

        String file_name = "random.txt";
        int character_size = 500;
        String key_to_encrypt = "alibaba";
        String dictionary_file_name = "dictionary.txt";
        long start, finish;

        generate_plain_text(file_name,character_size);
        populate_dictionary(dictionary_file_name);
        encrypt(plain_text,key_to_encrypt);
        find_key_length(encrypted_text);

        System.out.println("Using Brute Force to decrypt ..... \n");

        start = System.currentTimeMillis();
        brute_force(key_length,"");
        finish = System.currentTimeMillis();
        time_taken(start,finish);
        }
}
```

PSO :



```
#include <iostream>
#include <fstream>
#include <cctype>
#include <String>
#include <sstream>
#include <vector>
#include <algorithm>
#include <limits>
#include <list>
#include "VignereCipherEncrypt.h"
using namespace std;

string Decrypt(string text, vector <string> NewKey)
{
        for (int i = 0, j = 0; i < text.length(); i++)
        {
                decryptedMsg[i] = (((encryptedMsg[i] - NewKey[i]) + 26) % 26) + 'A';

                DecryptedMsg[i] = '\0';
        }

}
string Encrypt(string Original_Message, string key)
{
        string All_Char_Key_Space = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"; // Key Space
        int All_Char_Key_Space_length = All_Char_Key_Space.length();

        //convert string to uppercase
        transform(Original_Message.begin(), Original_Message.end(), Original_Message.begin(),
::toupper);


        //remove all everything that is not letters

        Original_Message.erase(remove_if(Original_Message.begin(), Original_Message.end(),

                [](char c) { return (!isalpha(c) && c != '\n' && c != '\r'); }), Original_Message.end());


        //convert key to uppercase

        transform(key.begin(), key.end(), key.begin(), ::toupper);


        cout << Original_Message << endl;

        cout << key << endl;

        int index = 0;

        string Encrypt_Message = "";


        //iterate over original message by each character and encrypt and add to new string

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

        {

                char letter = Original_Message[i];

                if (isalpha(letter))

                {

                        int position = (All_Char_Key_Space.find(letter) +
All_Char_Key_Space.find(key[index])) % 26;

                        char Encrypt_Letter = All_Char_Key_Space[position];

                        Encrypt_Message += Encrypt_Letter;

                        index++;

                        if (index >= key.length())

                        {

                                index %= key.length();
```

```
                    }
                    else
                    {
                              Encrypt_Message += letter;
                              index = 0;
                    }
          }

     }
     cout << Encrypt_Message << endl;
     return Encrypt_Message;
}
void PSO(int NumParticles, int KeyLength, int InputSize)
{
     double C1 = 2.05;
     double C2 = 2.05;
     double Inertia_Weight = 0.9;
     double VMin = 0.0;
     double VMax = 1.0;
     double r1 = 0.0;
     double r2 = 0.0;
     double GlobalBestKeyValue = numeric_limits<double>::infinity();
     string GlobalBestKey = "";
     int gBest = 0;
     int gBestTest = 0;
     int epoch = 0;
     bool done = false;          auto s = to_string(KeyLength);
     double RandomNum = ((double)rand() / (RAND_MAX)) + 1;//value between 0-1
     int num = rand() % 26;
```

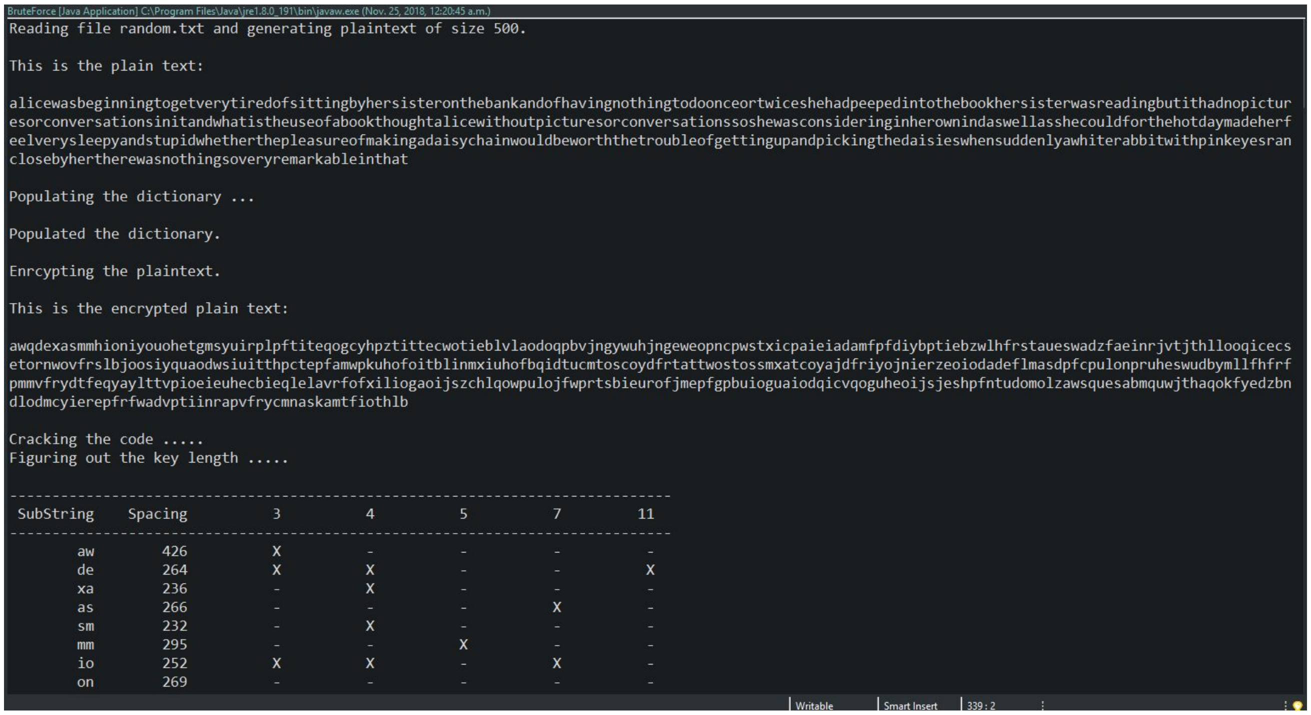
```
vector<string>LocalBestKey;
vector<string>CurrentKey;
vector<double>PFit;//pKey
vector<double>LocalBestFit;
vector<double>KeyVelocity;
vector<double>FitVelocity;

for (int i = 0; i < NumParticles; i++)//initalize particles
{
        string NewKey = "";
        for (int j = 0; j < KeyLength; j++)
        {
                NewKey += static_cast<char>('A' + num);
        }
        LocalBestKey.push_back("");
        CurrentKey.push_back(NewKey); //initalize key
        PFit.push_back(0.0);
        LocalBestFit.push_back(std::numeric_limits<double>::infinity());//initalize fitness
        KeyVelocity.push_back(VMin + (VMax - VMin)*RandomNum);
        FitVelocity.push_back(VMin + (VMax - VMin)*RandomNum);
}
string text = "";
int iteration = 100;
while (iteration != 1 || GlobalBestKeyValue < 0.05)
{
        for (int i = 0; i < NumParticles; i++)
        {
                string DecryptFile = Decrypt(text, CurrentKey); //decrypt with current key
        }
        }
        intialize();
}
void initialize()
{
        int total;

        for (int i = 0; i <= MAX_PARTICLES - 1; i++)
        {
                total = 0;
                for (int j = 0; j <= MAX_INPUTS - 1; j++)
                {
                        particles[i].setData(j, getRandomNumber(START_RANGE_MIN,
START_RANGE_MAX));
                        total += particles[i].getData(j);
                } // j
                particles[i].setpBest(total);
        } // i

        return;
}
void getVelocity(int gBestIndex)
{
        /* from Kennedy & Eberhart(1995).
                vx[][] = vx[][] + 2 * rand() * (pbestx[][] - presentx[][]) +
                                                        2 * rand() * (pbestx[][gbest] - presentx[][])
        */
        int testResults, bestResults;
        float vValue;
        bestResults = testProblem(gBestIndex);

        for (int i = 0; i <= MAX_PARTICLES - 1; i++)
        {
                testResults = testProblem(i);
                vValue = particles[i].getVelocity() +
                        2 * gRand() * (particles[i].getpBest() - testResults) + 2 * gRand() *
                        (bestResults - testResults);

                if (vValue > V_MAX) {
                        particles[i].setVelocity(V_MAX);
                }
                else if (vValue < -V_MAX) {
                        particles[i].setVelocity(-V_MAX);
                }
                else {
                        particles[i].setVelocity(vValue);
                }
        } // i
}

void updateParticles(int gBestIndex)
{
        int total, tempData;

        for (int i = 0; i <= MAX_PARTICLES - 1; i++)
        {
                for (int j = 0; j <= MAX_INPUTS - 1; j++)
                {
                        if (particles[i].getData(j) != particles[gBestIndex].getData(j))
                        {
                                tempData = particles[i].getData(j);
                                particles[i].setData(j, tempData +
static_cast<int>(particles[i].getVelocity()));
                        }
                } // j

                //Check pBest value.
                total = testProblem(i);
                if (abs(TARGET - total) < particles[i].getpBest())
                {
                        particles[i].setpBest(total);
                }

        } // i

}

int testProblem(int index)
{
        int total = 0;

        for (int i = 0; i <= MAX_INPUTS - 1; i++)
        {
                total += particles[index].getData(i);
        } // i
        return total;
}

float gRand()
{
        // Returns a pseudo-random float between 0.0 and 1.0
        return float(rand() / (RAND_MAX + 1.0));
}

int getRandomNumber(int low, int high)
{
        // Returns a pseudo-random integer between low and high.
        return low + int(((high - low) + 1) * rand() / (RAND_MAX + 1.0));
}

int minimum()
{
        //Returns an array index.
        int winner = 0;
        bool foundNewWinner;
        bool done = false;

        do
        {
                foundNewWinner = false;
                for (int i = 0; i <= MAX_PARTICLES - 1; i++)
                {
                        if (i != winner) {          //Avoid self-comparison.
                                //The minimum has to be in relation to the Target.
                        if (abs(TARGET - testProblem(i)) < abs(TARGET - testProblem(winner)))
                        {
                            winner = i;
                            foundNewWinner = true;
                        }
                    }
                } // i

                if (foundNewWinner == false)
                {
                    done = true;
                }

            } while (!done);

            return winner;
}
void Encrypt_File(string key)
{
            ifstream input("Input.txt");
            ofstream output;

            string data;

            try
            {
                stringstream ss;
                ss << input.rdbuf();
                data = ss.str();
```

```
		cout << "File Read Successfully" << endl;
	}


	catch (exception e)
	{
		cout << e.what();
	}
	input.close();
	string Encrypted_Data = Encrypt(data, key);
	cout << "File Data Encrypted Successfully" << endl;
	try
	{
		output.open("Output.txt");
		output << Encrypted_Data;
		output.close();
		cout << "Encrypted Data stored on File" << endl;
	}
	catch (exception ee)
	{
		cout << ee.what();
	}



}
int main()
{
	//read a file char by char
	cout << "Program for Vignere Cipher" << endl;
	string key = "OMGI";
```

```
            Encrypt_File(key);
    //change the input size
            PSO(100,4,100);//number of particles, key length, and inputsize
            system("pause");
            return 0;


    }
```

Screenshots

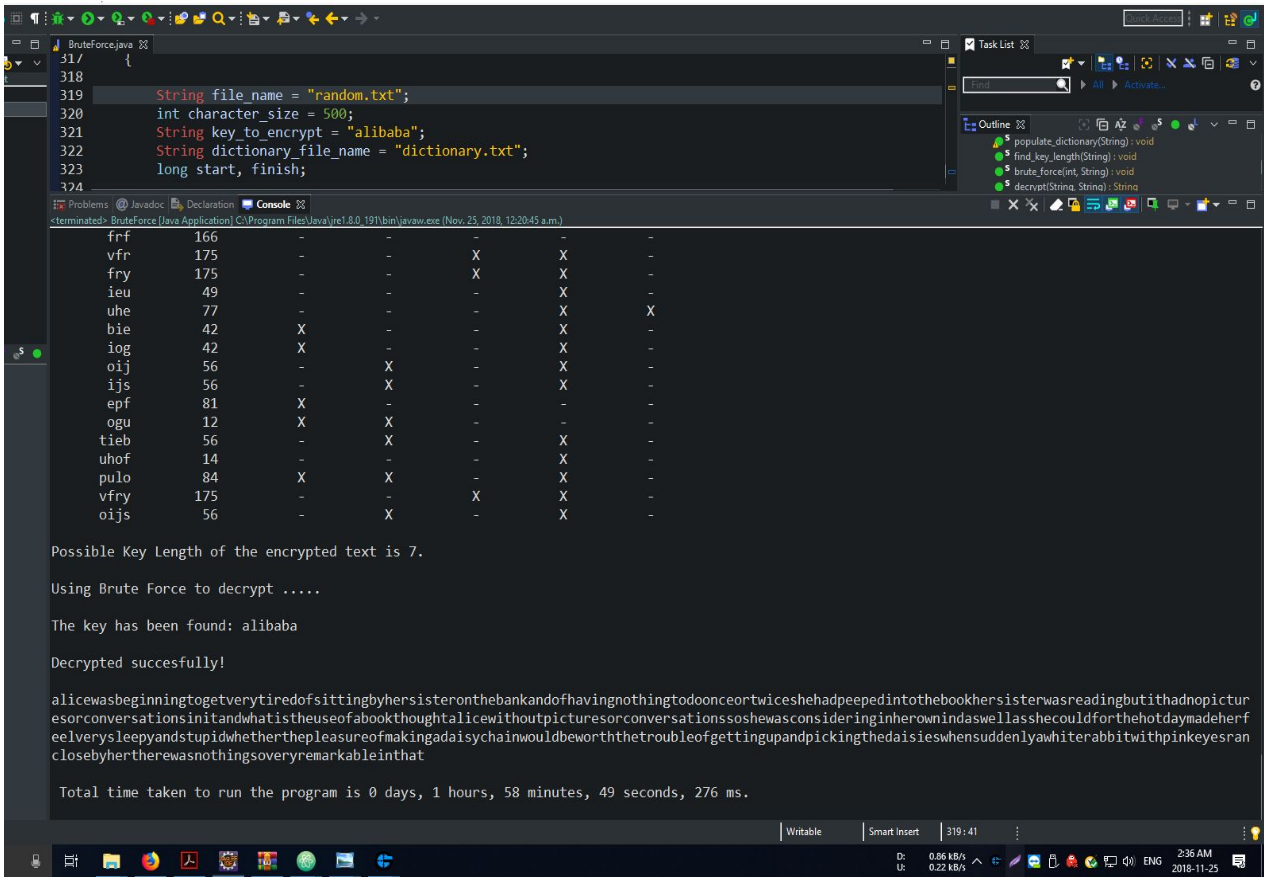
Brute Force Sample: size of 500, key length of 7



BruteForce [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe (Nov. 25, 2018, 12:20:45 a.m.)

```
Reading file random.txt and generating plaintext of size 500.

This is the plain text:

alicewasbeginningtogetverytiredofsittingbyhersisteronthebankandofhavingnothingtodoonceortwiceshehadpeepedintothebookhersisterwasreadingbutithadnopictur
esorconversationsinitandwhatistheuseofabookthoughtalicewithoutpicturesorconversationssoshewasconsideringinherownindaswellasshecouldforthehotdaymadeherf
eelverysleepyandstupidwhetherthepleasureofmakingadaisychainwouldbeworththetroubleofgettingupandpickingthedaisieswhensuddenlyawhiterabbitwithpinkeyesran
closebyhertherewasnothingsoveryremarkableinthat

Populating the dictionary ...

Populated the dictionary.

Enrcypting the plaintext.

This is the encrypted plain text:

awqdexasmmhioniyouohetgmsyuirplpftiteqogcyhpztittecwotieblvlaodoqpbvjngywuhjngeweopncpwstxicpaieiadamfpfdiybptiebzwlhfrstaueswadzfaeinrjvtjthllooqicecs
etornwovfrslbjoosiyquaodwsiuitthpctepfamwpkuhofoitblinmxiuhofbqidtucmtoscoydfrtattwostossmxatcoyajdfriyojnierzeoiodadeflmasdpfcpulonpruheswudbymllfhfrf
pmmvfrydtfeqyaylttvpioeieuhecbieqlelavrfofxiliogaoijszchlqowpulojfwprtsbieurofjmepfgpbuioguaiodqicvqoguheoijsjeshpfntudomolzawsquesabmquwjthaqokfyedzbn
dlodmcyierepfrfwadvptiinrapvfrycmnaskamtfiothlb

Cracking the code .....
Figuring out the key length .....

------------------------------------------------------------------------------
 SubString    Spacing          3          4          5          7         11
------------------------------------------------------------------------------
        aw       426           X          -          -          -          -
        de       264           X          X          -          -          X
        xa       236           -          X          -          -          -
        as       266           -          -          -          X          -
        sm       232           -          X          -          -          -
        mm       295           -          -          X          -          -
        io       252           X          X          -          X          -
        on       269           -          -          -          -          -
```

Total time taken for key length of 7



```
        {

            String file_name = "random.txt";
            int character_size = 500;
            String key_to_encrypt = "alibaba";
            String dictionary_file_name = "dictionary.txt";
            long start, finish;
```

```
<terminated> BruteForce [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe (Nov. 25, 2018, 12:20:45 a.m.)
       frf        166          -          -          -          -          -
       vfr        175          -          -          X          X          -
       fry        175          -          -          X          X          -
       ieu         49          -          -          -          X          -
       uhe         77          -          -          -          X          X
       bie         42          X          -          -          X          -
       iog         42          X          -          -          X          -
       oij         56          -          X          -          X          -
       ijs         56          -          X          -          X          -
       epf         81          X          -          -          -          -
       ogu         12          X          X          -          -          -
      tieb         56          -          X          -          X          -
      uhof         14          -          -          -          X          -
      pulo         84          X          X          -          X          -
      vfry        175          -          -          X          X          -
      oijs         56          -          X          -          X          -

Possible Key Length of the encrypted text is 7.

Using Brute Force to decrypt .....

The key has been found: alibaba

Decrypted succesfully!

alicewasbeginningtogetverytiredofsittingbyhersisteronthebankandofhavingnothingtodoonceortwiceshehadpeepedintothebookhersisterwasreadingbutithadnopictur
esorconversationsinitandwhatistheuseofabookthoughtalicewithoutpicturesorconversationssoshewasconsideringinherownindaswellasshecouldforthehotdaymadeherf
eelverysleepyandstupidwhetherthepleasureofmakingadaisychainwouldbeworththetroubleofgettingupandpickingthedaisieswhensuddenlyawhiterabbitwithpinkeyesran
closebyhertherewasnothingsoveryremarkableinthat

 Total time taken to run the program is 0 days, 1 hours, 58 minutes, 49 seconds, 276 ms.
```

Experimental Results

	Brute Force (in sec)				PSO (in sec)			
Key size	200 char	300 char	400 char	500 char	200 char	300 char	400 char	500 char
3	1.043	1.042	1.040	1.045	1.00	0.53	0.48	1.02
5	780	820	825	840	301	350	380	400
7	7140	7530	7852	8532	980	1022	1233	1500
11	N.F	N.F	N.F	N.F	N.F	N.F	N.F	N.F
3	1.042	1.045	1.049	1.051	1.02	0.84	0.67	0.99

5	765	785	768	795	350	355	350	450
7	7453	7754	73245	8320	950	890	930	1900
11	N.F	N.F	N.F	N.F	N.F	N.F	N.F	N.F
3	1.055	1.065	1.035	1.040	1	1	1.01	0.95
5	789	810	795	746	400	380	375	480
7	7455	7980	7653	8593	977	984	911	1532
11	N.F	N.F	N.F	N.F	N.F	N.F	N.F	N.F
3	1.045	1.044	1.044	1.043	0.891	0.90	1.06	1.8
5	786	788	793	850	380	400	420	420
7	7103	7553	7832	8912	975	923	986	1432
11	N.F	N.F	N.F	N.F	N.F	N.F	N.F	N.F

Discussion & Big O Analysis

Most of errors noticed during the project was when the key was having a similar character pattern or similar pattern. The program miss understands the key as the factor of the repetitions.

Success / Error * 100

(48 / 64) * 100 = 75 % success rate for the brute analysis.

Brute Force :

In order to find the frequency of repetitions of substrings, two for loops were used to iterate through the entire loop. A 3rd inner for loop was used to store and compare the the text. The Big O for finding the key length was n^3.

Java handles string comparisons real quick. When it comes to comparing pre mutated alphabets of key length > 5, the running time grows exponentially. In order to Brute Force, two for loops were used. One to iterate through each word of the decrypted text and another to compare it with a dictionary to check if the decrypted word is in the english dictionary. The Big O is n^2. Where n could be integer of size > 10 million depending on the key length.

PSO:

PSO uses swarm technique which requires following the its pbest value and swarm best value, so in terms of finding the key, It didn't require any recursions so in terms of that piece of code, it is a big O of (n). We found that in our code, the code fragment that had the greatest big O value that nullified the rest was the actual key search where we found the Big O to be of n^3.

Conclusion

In general, we found PSO to be a lot more efficient compared to Brute force and would recommend it over Brute Force. Brute Forces method simply does not hold up as it requires a drastic amount of trial and error to find the best solution, while PSO has a more efficient and calculated way of going about it. Even when it comes to the

Bibliography:

Vigenere Cipher:

Dcode. (2018). Vigenere cipher. [online] Available at: https://www.dcode.fr/vigenere-cipher
https://pages.mtu.edu/~shene/NSF-4/Tutorial/VIG/Vig-Base.html
http://practicalcryptography.com/cryptanalysis/stochastic-searching/cryptanalysis-vigenere-cipher/ [Accessed 26 Nov. 2018].
http://www.math.ucdenver.edu/~wcherowi/courses/m5410/m5410cc.html
https://www.dcode.fr/vigenere-cipher
https://pages.mtu.edu/~shene/NSF-4/Tutorial/VIG/Vig-Base.html
http://practicalcryptography.com/cryptanalysis/stochastic-searching/cryptanalysis-vigenere-cipher/
http://www.swarmintelligence.org/
http://mnemstudio.org/particle-swarm-introduction.htm