GALWAY-MAYO INSTITUTE OF TECHNOLOGY

Department of Computer Science & Applied Physics

B.Sc. Software Development – Artificial Intelligence (2018)
ASSIGNMENT DESCRIPTION & SCHEDULE

Using Simulated Annealing to Break a Playfair Cipher

T	H	E	Q	U
I	C	K	B	R
O	W	N	F	X
M	P	D	V	L
A	Z	Y	G	S

Note: This assignment will constitute 50% of the total marks for this module.

The field of *cryptanalysis* is concerned with the study of ciphers, having as its objective the identification of weaknesses within a cryptographic system that may be exploited to convert encrypted data (cipher-text) into unencrypted data (plain-text). Whether using symmetric or asymmetric techniques, cryptanalysis assumes no knowledge of the correct cryptographic key or even the cryptographic algorithm being used.

Assuming that the cryptographic algorithm is known, a common approach for breaking a cipher is to generate a large number of keys, decrypt a cipher-text with each key and then examine the resultant plain-text. If the text looks similar to English, then the chances are that the key is a good one. The similarity of a given piece of text to English can be computed by breaking the text into fixed-length substrings, called *n*-grams, and then comparing each substring to an existing map of *n*-grams and their frequency. This process does not guarantee that the outputted answer will be the correct plain-text, but can give a good approximation that may well be the right answer.

You are required to use the ***simulated annealing algorithm to break a Playfair Cipher***. Your application should have the following minimal set of features:

- A **menu-driven command line UI** that enables a cipher-text source to be specified (a file or URL) and an output destination file for decrypted plain-text.
- Decrypt cipher-text with a **simulated annealing** algorithm that uses a log-probability and *n*-gram statistics as a **heuristic evaluation** function.

A full description of the Playfair Cipher, a simulated annealing algorithm for breaking ciphers and *n*-gram statistics are provided below as supplementary material. Note that extra marks will only be given for features that directly relate to the content covered in this module. Do not waste your time developing a complex GUI application.

Deployment and Submission

- ***The project must be submitted by midnight on Sunday 8th April 2018*** as a Zip archive ***(not a rar or WinRar file)*** using the Moodle upload utility. You can find the area to upload the project under the *"Using Simulated Annealing to Break a Playfair Cipher - (50%) Assignment Upload"* heading in the *"Main Assignment"* section of Moodle.
- The name of the Zip archive should be *{id}*.zip where *{id}* is your student number.
- You must use the package name **ie.gmit.sw.ai** for the assignment.
- Do not include the file ***4grams.txt*** with your submission. You can assume that it is in the current directory, i.e. accessible as "**./4grams.txt**".
- The Zip archive should have the following structure (do NOT submit the assignment as an Eclipse project):

Name	Description
src	A directory that contains the packaged source code for your application. You must package your application with the namespace **ie.gmit.sw.ai**. Your code should be well commented and explain the space and time complexity of its classes.
README.txt	A text file outlining the main **features** of your application. Marks will only be given for extra features that are described in the README.
playfair.jar	A Java Application Archive containing the classes in your application. Use the following syntax to create a JAR from a command prompt: **jar –cf playfair.jar ie/gmit/sw/ai/*.class** Your application must be able to be launched using the following syntax from a command prompt: **java –cp ./playfair.jar ie.gmit.sw.ai.CipherBreaker** This requires that the *main()* method for your application is in a class called *CipherBreaker*.

Marking Rubric

Marks for the project will be applied using the following criteria:

Marks	Category
(60%)	Robustness. The application executes correctly.
(15%)	Packaging & Distribution
(25%)	Documented (and relevant) extras.

You should treat this assignment as a project specification. Any deviation from the requirements will result in a loss of marks. Each of the categories above will be scored using the following criteria:

- 0–30% Not delivering on basic expectations
- 40-50% Meeting basic expectations
- 60–70% Tending to exceed expectations
- 80-90% Exceeding expectations
- 90-100% Exemplary

The Playfair Cipher

The Playfair Cipher is a symmetric ***polygram*** substitution cipher invented by the Victorian scientist Sir Charles Wheatstone in 1854 (of Wheatstone Bridge fame). The cipher is named after his colleague Lord Playfair, who popularised and promoted the encryption system. Due to its simplicity, the Playfair Cipher was used at a tactical level by both the British and US forces during WWII and is also notable for its role in the rescue of the crew of PT-109 in the Pacific in 1943.

Polygram substitution is a classical system of encryption in which a group of *n* plain-text letters is replaced as a unit by *n* cipher-text letters. In the simplest case, where $n = 2$, the system is called digraphic and each letter pair is replaced by a cipher digraph. The Playfair Cipher uses digraphs to encrypt and decrypt from a 5x5 matrix constructed from a sequence ***key of 25 unique letters***. Note that the letter J is not included.

For the purposes of this assignment, it is only necessary to implement the decrypt functionality of the Playfair Cipher. It should be noted however, that the 5x5 matrix described below can be augmented with auxiliary data structures to reduce access times to ***O***(1). For example, the letter 'A' has a Unicode decimal value of 65. Thus, an *int* array called *rowIndices* could store the matrix row of a *char val* at *rowIndices*[val – 65]. The same principle can be used for columns… The encryption / decryption process works on diagraphs as follows:

Step 1: Prime the Plain-Text

Convert the plaintext into either upper or lower-case and strip out any characters that are not present in the matrix. A regular expression can be used for this purpose as follows:

```
String plainText = input.toUpperCase().replaceAll("[^A-Za-z0-9 ]", "");
```

Numbers and punctuation marks can be spelled out if necessary. Parse any double letters, e.g. LE**TT**ERKE**NN**Y and replace the second occurrence with the letter X, i.e. LE**TX**ERKE**NX**Y. If the plaintext has an odd number of characters, append the letter X to make it even.

Step 2: Break the Plain-Text in Diagraphs

Decompose the plaintext into a sequence of diagraphs and encrypt each pair of letters. The key used below, *THEQUICKBROWNFOXJUMPEDOVERTHELAZYDOGS*, is written into the 5x5 matrix as *THEQUICKBROWNFXMPDVLAZYGS*, as only the first 25 unique letters are used to form a key. The plain-text sequence of characters, "ARTIFICIALINTELLIGENCE", is broken into the diagraph set {AR, TI, FI, CI, AL, IN, TE, LL, IG, EN, CE} and is encrypted into the cipher-text "SIIOOBKCSMKOHQSLBAKDKH" using the following rules:

Rule 1: *Diagraph Letters in Different Rows and Columns*

Create a "box" inside the matrix with each diagraph letter as a corner and read off the letter at the ***opposite corner*** of the same row, e.g. AR→SI. This can also be expressed as cipher(B, P)={matrix[row(B)][col(P)], matrix[row(P)][col(B)]}. Reverse the process to decrypt a cypher-text diagraph.

T	H	E	Q	U
I	**C**	K	**B**	R
O	W	N	F	X
M	**P**	D	**V**	L
A	Z	Y	G	S

- BP→CV
- PB→VC

T	H	E	Q	U
I	C	K	B	R
O	W	N	F	**X**
M	P	D	V	**L**
A	Z	Y	G	S

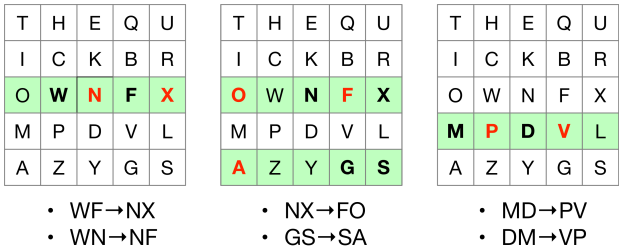
- OL→XM
- LO→MX

T	H	E	Q	U
I	C	**K**	**B**	R
O	W	N	F	X
M	P	D	V	L
A	Z	**Y**	**G**	S

- KG→BY
- GK→YB

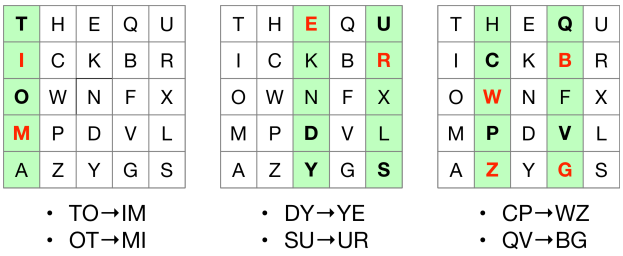
Rule 2: *Diagraph Letters in Same Row*

Replace any letters that appear on the same row with the letters to their ***immediate right***, wrapping around the matrix if necessary. Decrypt by replacing cipher-text letters the with letters on their immediate left.



Rule 3: *Diagraph Letters in Same Column*

Replace any letters that appear on the same column with the letters ***immediately below***, wrapping back around the top of the column if necessary. Decrypt by replacing cipher-text letters the with letters immediately above.



The Playfair Cipher suffers from the following three basic weaknesses that can be exploited to break the cipher, even with a pen and paper:

1. **Repeated** plain-text digrams will create repeated cipher-text digrams.
2. Digram **frequency counts** can reveal the most frequently occurring English digrams.
3. The most frequently occurring cipher-text letters are **likely to be near** the most frequent English letters, i.e. E, T, A and O in the 5x5 square. This helps to reconstruct the 5x5 square.

We will be exploiting weakness (2) in this assignment. Note that these weaknesses rely on repetition and frequency counts and, in the absence of cribs, require enough cipher-text to reveal patterns. In practice, this implies that at least 200 characters of cipher-text are available.

The Simulated Annealing Algorithm

Simulated annealing (SA) is an excellent approach for breaking a cipher using a randomly generated key. Unlike conventional Hill Climbing algorithms, that are easily side-tracked by local optima, SA uses randomization to avoid heuristic plateaux and attempt to find a global maxima solution. The following pseudocode shows how simulated annealing can be used break a Playfair Cipher. Note that the initial value of the variables *temp* and *transitions* can have a major impact on the success of the SA algorithm. Both variables control the ***cooling schedule*** of SA and should be experimented with for best results (see slide 20 of the lecture notes on *Heuristic Search*).

```
1.  Generate a random 25 letter key called parent.
2.  Decrypt the cipher-text with the key
3.  Score the fitness of the key as logProbability(parent)
4.  For temp = 10  To 0 Step -1
5.    For transitions = 50000  To 0 Step -1
6.      Set child = shuffleKey(parent) //Make a small change to the key
7.      Score the fitness of the key as logProbability(child)
8.      Set delta = logProbability(child) - logProbability(parent)
9.      If delta > 0 Then  //New key better
10.       Set parent = child
11.     Else If delta < 0 Then  //New key worse
12.       Set parent = child with probability e^(-delta/temp)
13.   Next
14. Next
```

Simulated Annealing Algorithm for Keys

The generation of a random 25-letter key on line 1 only requires that we shuffle a 25 letter alphabet. A simple algorithm for achieving this was published in 1938 by Fisher and Yates. The ***Fisher–Yates Shuffle*** generates a random permutation of a finite sequence, i.e. it randomly shuffles an array key of *n* elements (indices 0..*n*-1) as follows:

Pseudocode

```
for i from n−1 to 1
  j ← random integer such that 0 ≤ j ≤ i
  swap key[j] and key[i]
```

Java (Version 1)

```
private void shuffle(char[] key){
   int index, temp;
   Random random = new Random();
   for (int i = key.length - 1; i > 0; i--){
      index = random.nextInt(i + 1);
      temp = key[index];
      key[index] = key[i];
      key[i] = temp;
   }
}
```

Java (Version 2 Quicker)

```
public void shuffle(char[] key) {
   int index;
   Random random = ThreadLocalRandom.current();
   for (int i = key.length - 1; i > 0; i--) {
      index = random.nextInt(i + 1);
      if (index != i) {
         key[index] ^= key[i];
         key[i] ^= key[index];
         key[index] ^= key[i];
      }
   }
}
```

The method shuffleKey() on line 6 should make the following changes to the key with the frequency given (you can approximate this using Math.random() * 100):

- Swap single letters (90%)
- Swap random rows (2%)
- Swap columns (2%)
- Flip all rows (2%)
- Flip all columns (2%)
- Reverse the whole key (2%)

Note that the semantic network or state space is implicit and that each small change to the 25-character key is logically the same as traversing an edge between two adjacent nodes.

Using *n*-Gram Statistics as a Heuristic Function

An *n*-gram (gram = word or letter) is a substring of a word(s) of length *n* and can be used to measure how similar some decrypted text is to English. For example, the quadgrams (*4*-grams) of the word "HAPPYDAYS" are "HAPP", "APPY", "PPYD", "PYDA", "YDAY" and "DAYS". A fitness measure or heuristic score can be computed from the frequency of occurrence of a *4*-gram, ***q***, as follows: $\mathbf{P}(q) = \text{count}(q) / n$, where *n* is the total number of *4*-grams from a large sample source. An overall probability of the phrase "HAPPYDAYS" can be accomplished by multiplying the probability of each of its *4*-grams:

$$\mathbf{P}(\text{HAPPYDAYS}) = \mathbf{P}(\text{HAPP}) \times \mathbf{P}(\text{APPY}) \times \mathbf{P}(\text{PPYD}) \times \mathbf{P}(\text{PYDA}) \times \mathbf{P}(\text{YDAY})$$

One side effect of multiplying probabilities with very small floating point values is that underflow can occur[1] if the exponent becomes too low to be represented. For example, a Java ***float*** is a 32-bit IEEE 754 type with a 1-bit sign, an 8-bit exponent and a 23-bit mantissa. The 64-bit IEEE 754 ***double*** has a 1-bit sign, a 11-bit exponent and a 52-bit mantissa. A simple way of avoiding this is to get the *log* (usually base 10) of the probability and use the identity *log*(a × b) = *log*(a) + *log*(b). Thus, the score, ***h(n)***, for "HAPPYDAYS" can be computed as a ***log probability***:

$$\log_{10}(\mathbf{P}(\text{HAPPYDAYS})) = \log_{10}(\mathbf{P}(\text{HAPP})) + \log_{10}(\mathbf{P}(\text{APPY})) + \log_{10}(\mathbf{P}(\text{PPYD})) + \log_{10}(\mathbf{P}(\text{PYDA})) + \log_{10}(\mathbf{P}(\text{YDAY})$$

The resource ***quadgrams.txt*** is a text file containing a total of 389,373 *4*-grams, from a maximum possible number of 26^4=456,976. The *4*-grams and the count of their occurrence were computed by sampling a set of text documents containing an aggregate total of 4,224,127,912 *4*-grams. The top 10 *4*-grams and their occurrence is tabulated below:

q	Count(*q*)	*4*-gram, *q*	Count(*q*)
TION	13168375	FTHE	8100836
NTHE	11234972	THES	7717675
THER	10218035	WITH	7627991
THAT	8980536	INTH	7261789
OFTH	8132597	ATIO	7104943

The *4*-grams of "HAPPYDAYS", their count, probability and log value are tabulated below.

q	Count(*q*)	Probability(*q*)	$\text{Log}_{10}(\text{P}(q))$
HAPP	462336	0.000109451	-3.960779349
APPY	116946	0.000027685	-4.557751689
PPYD	4580	0.000001084	-5.964871583
PYDA	1439	0.000000340	-6.467676267
YDAY	108338	0.000025647	-4.590956247
DAYS	635317	0.000150401	-3.822746584

The final score, ***h(n)***, for "HAPPYDAYS" is just the sum of the log probabilities, i.e. -3.960779349 + -4.557751689 + -5.964871583 + -6.467676267 + -4.590956247 + -3.822746584 = **-29.36478172**. A decrypted message with a larger score than this is more "English" than this text and therefore must have been decrypted with a "better" key.

[1] An excellent analysis of underflow is given by *Goldberg, D., 1991. What every computer scientist should know about floating-point arithmetic. ACM Computing Surveys (CSUR), 23(1), pp.5-48.*