Rapport de projet annuel
L3 Informatique

Etude, implantation et cryptanalyse de systèmes de chiffrement alphabétiques

Mars 2018

Auteurs: Tom Chambaretaud, Salomé Degré-Quédec, Corentin Trouvay
Enseignant référent: Magali Bardet

Table des matières

1 Introduction

Le chiffrement d'informations s'effectue à l'aide d'une clé dont les seuls possesseurs doivent être les personnes autorisées. Ainsi, l'envoyeur chiffre ses informations à l'aide de la clé puis effectue l'envoi. Une fois le message reçu, le destinataire n'a plus qu'à déchiffrer le message à l'aide de la clé pour accéder à l'information. Nous traiterons de deux chiffrements qui ont, de leurs temps, permis de conserver les conversations secrètes de leurs utilisateurs : « Le chiffrement de Vigenère » ainsi que « le chiffrement de César ». Puis nous vous décrirons notre implantation informatique de ces deux chiffrements.

Le but du chiffrement est de rendre opaque l'information à l'intérieur du tube de communications : l'intérêt se situe dans le fait que si le message est intercepté par une personne non autorisée, donc ne possédant pas la clé, elle ne pourra pas accéder à l'information. Le seul moyen pour cette personne d'accéder à l'information cachée serait de parvenir à déchiffrer le message sans l'aide de la clé, c'est ce que l'on nomme la cryptanalyse, que nous traiterons dans la deuxième partie.

2 Enjeux et historique de la Cryptographie

2.1 Les enjeux du chiffrement

La confidentialité des communications a été un enjeu majeur. Ainsi, on a pu retrouvé la trace d'une recette secrète de poterie chiffrée qui date du XVIe siècle av. J.-C. en Irak.

Dans tous contextes économiques et sociaux, l'action de dissimuler à des personnes tierces sonne comme une nécessité. Dans un contexte de guerre, n'est-il pas essentiel de garder masqué les interactions entre soldats ?

Deux disciplines artistiques puis scientifiques se sont formées au cours des siècles : La cryptographie et la cryptanalyse, pierres angulaires de la cryptologie, l'art de dissimuler.

La cryptographie peut-être comparée à une course d'endurance : la recherche d'un chiffrement robuste ne permettant qu'aux personnes autorisées d'accéder aux informations confidentielles.

La cryptanalyse peut être comparée à une course de rapidité : l'analyse du chiffrement doit permettre d'accéder rapidement à des informations sensibles sans en avoir été au préalable autorisé, sans avoir eu accès à la clé.

2.2 Histoire des chiffrements de Vigenère et de César

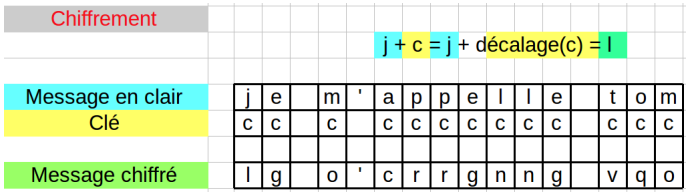
Le chiffrement de César, aussi connu sous le nom de chiffrement par décalage, est une méthode de chiffrement très simple utilisée avant J.C., notamment par Jules César. Il s'agit d'un chiffrement par substitution mono-alphabétique : une même lettre est toujours chiffrée à l'aide du même symbole de substitution. Assez rudimentaire dans son ensemble, il peut être facilement mis en place et permet une utilisation très rapide. Malheureusement, la simplicité de ce chiffrement permet très peu de clés différentes : nous verrons dans la deuxième partie que des attaques peuvent être très facilement mises en place.

Le chiffrement de Vigenère est fait par substitution polyalphabétque : à l'inverse de César, une même lettre ne sera pas chiffrée par la même partie de clé, ainsi sa version chiffrée ne sera pas forcément le même symbole de substitution. Ce chiffrement est décrit dans le « Traité des chiffres »de Blaise de Vigenère en 1586. Mais on retrouve des traces d'une méthode de chiffrement similaire dans un traité de Giovan Battista Bellaso en 1533. Le chiffrement de Vigenère a été percé par le prussien Friedrich Kasiski en 1863, mettant fin à quasiment trois siècles d'invulnérabilité. Même s'il est possible que Charles Babbage soit le premier en 1854, bien qu'il n'ait pas publié ses résultats.

3 Cryptographie du chiffrement de César et Vigenère

3.1 Fonctionnement des méthodes de chiffrements

Le chiffrement de César consiste en un décalage par la droite de chaque caractère présent dans l'alphabet, où décalage('a') = 0, décalage('b') = 1 ... décalage('z') = 25 pour un exemple avec l'alphabet latin par défaut.



Le chiffrement de Vigenère consiste aussi un décalage par la droite de chaque caractère présent dans l'alphabet, mais cette fois la clé n'est pas toujours la même : la clé est représentée par une chaîne de caractères, on effectue une rotation dans cette clé ainsi :

Algorithm 1 chiffrer_vigenere(message$_{clair}$: **chaîne**, clé : **chaîne**) :(**chaîne**)

```
message_chiffre = chaîne
for i = 0 : i < longueur(message) do
    message_chiffre(i) = message_clair(i) + clé(i mod longueur(clé)))
end for
return (message_chiffre)
```

x	a	b	c	d	e	f	g	h	...	z
décalage(x)	0	1	2	3	4	5	6	7	...	25

Chiffrement

j + c = j + décalage(c) = l

Message en clair	j	e		m	'	a	p	p	e	l	l	e		t	o	m
Clé	c	l		e		c	l	e	c	l	e	c		l	e	c
Message chiffré	l	p		q	'	c	a	t	g	w	p	g		e	s	o

3.2 Cas particulier : le chiffrement de César

En étudiant le chiffrement de Vigenère, on peut en déduire que le chiffrement de César est un cas particulier de Vigenère : effectivement, le décalage effectué par le chiffrement de César peut être vu comme un chiffrement de Vigenère à l'aide d'une clé d'une lettre correspond au décalage. Ainsi, notre implantation du chiffrement César n'est que l'utilisation de notre méthode de Vigenère à l'aide d'un mot d'une lettre comme clé.

3.3 Implantation de Vigenère

Notre chiffrement prend en paramètre le message à chiffrer, la clé mais aussi l'alphabet que l'on souhaite utiliser, les caractères que nous ne souhaitons pas garder et les caractères que nous voulons transformer. Ainsi par défaut, nous utilisons l'alphabet latin en minuscule, nous supprimons les caractère de mise en forme et les espaces, et nous changeons les accents et caractères non gérés en ASCII (non-étendu) par leur lettre d'origine.

Nous procédons ainsi : nous changeons et supprimons les caractères indiqués dans les paramètres dans la chaîne de caractères à chiffrer, puis pour chaque caractère indiqué comme autorisé, nous effectuons le décalage dans l'alphabet à l'aide de la clé. Les caractères non autorisés sont réécris sans modifications. Ainsi, un chiffre, dans notre configuration par défaut, sera réécris dans notre sortie. Notre déchiffrement a un comportement quasi identique à la différence près que le décalage est un décalage opposé.

4 Automatisation de la reconnaissance des mots

4.1 La classe Dico

Le but du projet étant d'automatiser ses attaques, il faut trouver un moyen de savoir si le message déchiffré est porteur de sens ou non. En effet, pour la machine, il lui faut analyser les mots pour déterminer s'ils existent ou non dans la langue sélectionnée. Pour cela nous avons besoin de dictionnaires ainsi que de livres pour déterminer les caractères les plus présents dans une langue donnée.

Nous avons donc créé une classe utilitaire manipulant des fichiers contenant des dictionnaires ou des livres. L'intérêt d'associer un livre au dictionnaire vient du fait de pouvoir gérer la même langue entre le dictionnaire et le livre. Cette classe possède deux attributs : une liste correspondant à l'ensemble des mots du dictionnaire triés par leur première lettre et un tableau des caractères les plus présents dans la langue dans l'ordre décroissant. Le constructeur de cette classe prend en paramètre un nom de fichier et un nom de livre. Ces informations permettent d'initialiser la liste et le tableau à l'aide de la manipulation des fichiers.

4.1.1 Recherche rapide dans un dictionnaire

Nous avons voulu accélérer la recherche d'un mot dans un dictionnaire. Pour cela nous parcourrons le dictionnaire et nous le partitionnons dans des sous-listes dont tous les mots commencent par la même lettre. Ainsi la recherche d'un mot se fait uniquement parmi ceux dont la première lettre est la même. Cela permet d'équilibrer la rapidité de la recherche et la taille de la mémoire, c'est un compromis entre une recherche exhaustive lente mais peu coûteuse en mémoire et une table de hachage rapide mais très coûteuse en mémoire.

Algorithm 2 Initialisation de la liste(dictionnaire : **fichier**)

```
liste : liste
for all mot in dictionnaire do
  sous_liste = liste[premiere_lettre(mot)]
  ajouter(mot, sous_liste)
end for
return liste
```

4.1.2 Fréquence des caractères dans un livre

Pour déterminer les caractères les plus présents dans une langue, on calcule, pour chaque lettre d'un livre, son nombre d'occurrence.

Algorithm 3 Initialisation du tableau des fréquences(livre : **fichier**)

```
tableauFrequence : (entier,caractere)[ ]
for all lettre in livre do
    ajouter 1 à l'entier correspondant à lettre
end for
return tri(liste) par ordre croissant d'occurrence
```

4.2 Redécoupage d'une chaîne de caractère sans espace

Le problème avec une chaîne de caractères sans espaces se situe dans le fait qu'elle n'est pas reconnaissable à l'aide d'un dictionnaire puisque le dictionnaire est composé de mots.

Nous devons donc parvenir à redécouper notre chaîne de caractères pour que les mots soient reconnaissables par le dictionnaire. Nous pourrons alors automatiser la reconnaissance d'une chaîne de caractères et ainsi, parvenir à définir si ce que l'on déchiffre est porteur d'informations.

Pour parvenir à redécouper une chaîne de caractères avec des espaces, nous devons tester différentes possibilités, un grand nombre, et définir la meilleure version de découpage. Nous calculons donc, à l'aide d'un système de points, le score de chaque version.

Première règle du système de points : « Plus une version contiendra un grand nombre de mots appartenant au dictionnaire, plus le score de cette version sera élevé. » Cette première règle est essentielle pour déterminer si l'information déchiffrée est porteuse de sens. Mais si nous nous limitons à cette règle, nous risquons d'avoir une version avec beaucoup de mots de petite taille, puisqu'un mot tel que « à » existe dans le dictionnaire sous la forme « a ». Chaque « a » sera donc potentiellement un « a » découpé avec un espace avant et après.

Deuxième règle : « le score de chaque mot sera calculé à l'aide de la formule : 'longueur(mot) – 1' » Cette deuxième règle permet de valoriser les mots de plus de 2 lettres dans le système de score. Ce qui est plus cohérent avec notre vocabulaire courant.

5 Cryptanalyse du chiffrement de César

5.1 Introduction

Nous avons vu que la cryptanalyse du chiffrement de César était possible grâce au faible nombre de clés possibles. Nous pouvons donc effectuer deux types d'attaques qui consistent à tester les différentes clés et à déterminer si le résultat semble porteur de sens.

5.2 Attaque force brute

5.2.1 Explications

L'attaque par force brute consiste à tester toutes les clés possibles dans l'ordre de 0 à (longueur de l'alphabet - 1). Cette attaque est possible avec César puisque le nombre de clés est faible. On peut donc déchiffrer le message puis tester si les mots du message déchiffré sont présents dans le dictionnaire. Chaque mot dans le dictionnaire est alors comptabilisé. On calcule alors le pourcentage de lettres des mots reconnus par rapport au nombre de lettres total du message. Si celui-ci est supérieur au pourcentage passé en paramètre de l'algorithme, on s'arrête : on considère que le message déchiffré est porteur de sens. Sinon on renvoie le message avec le pourcentage de comparaison le plus élevé.

5.2.2 Algorithme de l'attaque par force brute du chiffrement de César

Algorithm 4 attaqueForceBruteCesar(message : **chaîne**, d : **dico**, alphabet : **chaîne**, pourcentageMin : **réel**) :(**chaîne**,**entier**)

```
clé : entier
pourcentageMax : réel
cléMax : entier
chaîneMax : chaîne
clé ← 0
pourcentageMax ← 0.0
cléMax ← 0
chaîneMax ← chaîne vide
on transforme le message, on enlève les espaces et on met tout en minuscules
while i < longueur(alphabet) do
  compteur : entier
  compteur ← 0
  messageDechiffreAvecEspaces : chaîne
  messageDechiffreAvecEspaces ← on déchiffre et on retrouve les espaces
  tableauMots : chaine[ ]
  tableauMots ← decoupeChaine(messageDechiffreAvecEspaces)
  for all mot dans tableauMots do
    if mot ≠ chaine vide  and estDans(mot, d) then
      compteur ← compteur + longueur(mot) - 1
    end if
  end for
  pourcentage ← (compteur / longueur(message)) * 100.0
  if pourcentage ≥ pourcentageMin then
    return  (messageDechiffreAvecEspaces, cle)
  end if
  if pourcentage ≥ pourcentageMax then
    (renvoie le message déchiffre avec le pourcentage de comparaison le plus grand)
    pourcentageMax ← pourcentage
    cléMax ← clé
    chaîneMax ← messageDéchiffréAvecEspaces
  end if
  clé ← clé + 1
end while
return  (chaineMax, cléMax)
```

5.3 Attaque fréquentielle

5.3.1 Explications

Pour l'attaque fréquentielle, on teste toutes les clés possibles également mais l'ordre est prédéfini selon la lettre la plus présente : on peut penser que, statistiquement, la lettre la plus présente du message est égale à la lettre la plus présente de la langue du message - décalage de la clé. Ainsi, on prend cette différence pour clé et tenter de déchiffrer. Si on détecte un échec, on recommence en prenant la prochaine lettre la plus fréquente de la langue. Cette solution permet d'arriver rapidement au résultat puisque d'une manière générale il y a moins de clés à tester. Plus le texte est long, plus l'efficacité de cette attaque est importante à comparer avec l'attaque par force brute.

5.3.2 Algorithme de l'attaque fréquentielle du chiffrement de César

Algorithm 5 attaqueFrequentielleCesar(message :**chaîne**, d :**dico**,alphabet :**chaîne**, pourcentageMin : **réel**)(**chaîne**,**entier**)

```
i : entier
i ← 0
pourcentageMax : réel
cléMax : entier
chaîneMax : chaîne
pourcentageMax ← 0.0
cléMax ← 0
chaîneMax ← chaîne vide
index : entier
index ← on renvoie l'indice de la lettre la plus présente (0 pour 'a' ... 25 pour 'z')
on transforme le message, on enlève les espaces et on met tout en minuscules
while i < longueur(caractèresAutorisés) do
  compteur : réel
  compteur ← 0
  ordre : entier
  ordre ← décalage entre la i^eme lettre la plus fréquente dans la langue et le 'a'
  clé : entier
  clé ← index - ordre
  messageDéchiffréAvecEspaces : chaîne
  messageDéchiffréAvecEspaces ← on déchiffre et on retrouve les espaces
  tableauMots : chaine[ ]
  tableauMots ← decoupeChaine(messageDéchiffréAvecEspaces)
  for all mot dans tableauMots do
    if mot ≠ chaîne vide and estDans(mot, d) then
      compteur ← compteur + longueur(mot) − 1
    end if
  end for
  pourcentage ← (compteur / longueur(message)) ∗ 100.0
  if pourcentage ≥ pourcentageMin then
    return (messageDechiffreAvecEspaces, clé)
  end if
  if pourcentage ≥ pourcentageMax then
    (renvoie le message déchiffré avec le pourcentage de comparaison le plus grand)
    pourcentageMax ← pourcentage
    cléMax ← clé
    chaineMax ← messageDéchiffréAvecEspaces
  end if
  i ← i + 1
end while
return (chaineMax, cléMax)
```

6 Cryptanalyse du chiffre de Vigenère

6.1 Introduction

Contrairement au chiffrement de César, le nombre de clés possibles pour Vigenère est fort, puisque la clé n'a de limite que par la longueur du texte à chiffrer. On ne peut donc pas effectué d'attaque par force brute de type César. Le chiffre de Vigenère étant un chiffrement poly-alphabétique, contrairement à celui de César qui est mono-alphabétique : il n'est donc pas non plus possible d'effectuer une attaque fréquentielle comme celle de César vue précédemment.

Sa cryptanalyse se déroule donc en deux temps :

Il faut en premier lieu trouver la longueur de la clé utilisée. Cette longueur l permet de découper le message chiffré en l sous textes. Ce découpage se fait en ajoutant un caractère tous les l caractères à un certain sous texte. Le premier sous texte commence le découpage à partir du premier, le deuxième à partir du deuxième et ainsi de suite. En découpant le message chiffré selon cette longueur, on se place dans un contexte de cryptanalyse du chiffre de César pour chaque sous message. En effet, chaque sous message est chiffré avec une seule lettre de la clé : il s'agit d'un chiffrement de César. Une fois cette longueur clé trouvée, il faut retrouver, dans un second temps, la clé en elle-même. Nous allons donc voir qu'il existe plusieurs méthodes pour ces deux parties de l'attaque de Vigenère.

6.2 Déroulement de l'attaque : première partie

Il est possible de retrouver la longueur de la clé grâce à deux attaques différentes. La première que nous allons étudier est celle de Kasiski, la seconde est basée sur ce que l'on appelle l'indice de coïncidence.

6.2.1 Kasiski

L'attaque de Kasiski a pour but de retrouver des segments de trois lettres qui se répètent dans le message chiffré. Ces répétitions peuvent indiquer deux choses : soit ce sont des lettres totalement différentes qui, une fois chiffrées, ont donné la même suite de lettres ; mais cela est peu probable, on peut donc envisager une autre raison ; soit cela signifie que ces séquences, de lettres identiques dans le message en clair, ont été chiffrées par la même partie de la clé. À partir de là, on mesure la distance, c'est-à-dire le nombre de lettres, entre ces répétitions. Kasiski et Babbage ont identifié cette distance comme étant un multiple de la longueur de la clé. Il ne reste plus qu'à récupérer les distances entre toutes les répétitions, puis de trouver le diviseur de ces distances qui apparaît le plus souvent.

Pour mieux se rendre compte des étapes de la méthode, prenons l'exemple du texte suivant :

v'cevmfmqsslnikoypop'erbownvyqkrasilctpykpkqko
wropcfmqowyeqmxhc,irvinvyqsqnyvrkrrmslmssbwk
ewgmejtekkmqyvekrgci.cslcyamiqkhczeqcijowdbsld
mcbiqnyayrrsrcxxcevmzicxirspysrqzmpohcxsklvce
wccesdvccgmwtcdmrsslcqscmakpcc.

Ici, certains segments trouvés ont été mis en évidence. Nous allons maintenant voir ce que donne l'analyse des distances entre ces derniers :

		diviseurs					
segment	distance	2	3	5	7		67
cev	150	X	X	X			
fmq	45		X	X		...	
ssl	189		X		X		
nvy	42	X	X		X		
qny	67						X

On remarque que le diviseur le plus commun est le 3. En effet il apparaît quatre fois, et les autres deux et une fois. On peut donc facilement supposer que la longueur de la clé utilisée pour chiffrer ce message est 3. Pour calculer la longueur on pourrait être tenté de faire un simple calcul de PGCD sur toutes les distances. Malheureusement le fait que certains segments puissent être dus au hasard nous en empêche. En effet, comme le montre l'exemple, la répétition des lettres 'qny' ne doit pas être prise en compte. Sa distance de répétition étant 67, le calcul du PGCD serait erroné. L'algorithme utilisé est donc plus complexe avec l'utilisation d'une nouvelle fonction : $most_plausible(args)$, pour ce projet-ci, qui retourne la longueur de clé la plus probable.

6.2.2 Indice de coïncidence

Maintenant que nous avons vu la méthode de Kasiski, passons à celle-ci. Cette méthode a été inventée par William F. Friedman en 1920. L'indice de coïncidence correspond à la probabilité de tomber deux fois sur la même lettre lorsque l'on choisi deux lettres au hasard dans un texte. Plus généralement cet indice permet de savoir s'il s'agit d'un texte chiffré avec un chiffre mono-alphabétique (avec César), ou poly-alphabétique (avec Vigenère). Le chiffrement de Vigenère utilise plus d'alphabets que César, la probabilité est donc réduite par le nombre d'alphabets (26, un alphabet par lettre). Donc pour le cas du poly-alphabétique, l'indice du message est égal à $\frac{1}{26} \simeq 0,0385$. Sinon l'indice du message chiffré mono-alphabétiquement est le même que celui du message clair (*0,065* pour un texte en français) : l'alphabet reste le même, il est juste décalé. La formule de l'indice prend en compte le nombre total de lettre dans le texte, n, ainsi que le nombre de chaque lettre de l'alphabet une à une, n_i avec i représentant a si i=0, b si i=1, etc. :

$$IC = \sum_{q=1}^{q=Z} \frac{n_q*(n_q-1)}{n*(n-1)}$$

Pour trouver la longueur de la clé, à chaque tour de boucle i entre 2 (1 étant un chiffrement de César) et un limite l, fixée arbitrairement, on partage le message en i sous message. Ensuite on teste si, pour chaque sous message, l'indice est supérieur à 0,065. Si c'est le cas, cela signifie que i est la longueur de la clé. Sinon l'indice sera aux alentours de 0.0385, et on continue jusqu'à la limite l. Si aucune longueur n'a été trouvée, cela signifie qu'il y a une erreur quelque part. Il est possible que le texte soit trop : toutes les lettres de l'alphabet ne sont pas représentées.

Algorithm 6 indiceDeCoïncidence(message :**chaine**, b :**float**)(**entier**)

```
limite, i , acc : entier
limite ← 10
i ← 2
acc ← 0
while (i < limite) do
  listeSsTxt : list(chaine)
  listeSsTxt ← liste des i sous texte de message
  for all ssText in listeSsTxt do
    indice : float
    indice ← indice de coïncidence de ssText
    if indice ≥ b then
      acc ← acc + 1
    end if
  end for
  if acc ≥ (i / 2) then
    return i
  end if
  i ← i + 1
end while
return ERREUR
```

Si nous reprenons l'exemple précédent nous obtenons, pour i=3, les indices suivants :*0.0682*, *0.1159* et *0.0801*. Ils sont bien tous les trois supérieurs à 0.065, donc la longueur de la clé est 3.

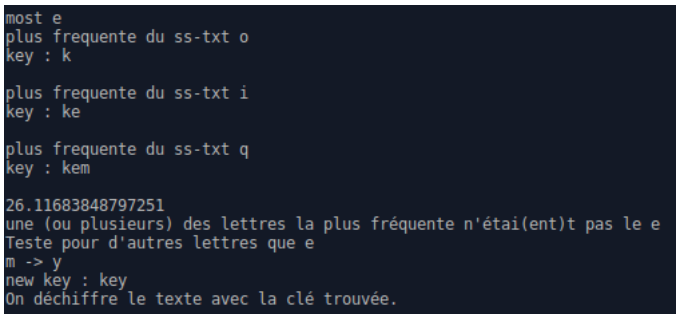
6.3 Déroulement de l'attaque : seconde partie

Une fois la longueur trouvée, il faut encore retrouver la clé, plus précisément les lettres qui la composent. Là aussi, il existe deux méthodes différentes. La première que nous allons étudier est l'analyse fréquentielle, la seconde est la méthode qui utilise l'indice de coïncidence mutuel.

6.3.1 Analyse fréquentielle

L'analyse fréquentielle du chiffre de Vigenère est similaire à celle de César. En effet, dans une version simplifiée de l'algorithme, il suffit de reprendre les sous textes de l'indice de coïncidence. Puis pour chaque sous textes, il faut chercher la lettre la plus fréquente et la comparer à la plus fréquente du livre de référence (généralement le « e »). Une fois le décalage trouvé entre la plus fréquente du sous texte et celle du livre, on possède toutes les lettres de la clé. Pour finir, il suffit de vérifier que le message déchiffré avec cette clé appartient bien au langage du message clair de base. Si ce n'est pas le cas, on refait un test, mais en prenant la deuxième lettre la plus fréquente. Et ainsi de suite, jusqu'à trouver la bonne clé.

Pour donner une meilleur idée de l'algorithme, observons la sortie sur le terminal :



Sur la photo ci-dessus, nous pouvons voir que *most* représente la lettre la plus fréquente du livre rentré en paramètre (« Vingt mille lieues sous les mers », Jules Verne). Ensuite la lettre la plus présente dans chaque sous textes est affichée. La comparaison est faite avec *most* pour donner la lettre de la clé. Malheureusement il y a une lettre qui ne convient. On le voit à l'œil nu, et cela est corroboré par le *pourcentage* affiché. Il n'est pas assez élevé, il devrait être supérieur à *50* pour appartenir au bon langage. On cherche donc les autres lettres les plus fréquentes, pour obtenir un meilleur résultat. On finit donc par changer la lettre « m » par « y ». Et on obtient la bonne clé : *key*.

Même si on finit par trouver la bonne clé, ce n'est pas la meilleure méthode. En effet, le fait d'être obligé de revérifier toutes les lettres de la clé prend beaucoup de temps. C'est pourquoi la prochaine méthode est plus automatisable, car il y a moins de test en interne à effectuer.

6.3.2 Indice de coïncidence mutuel

L'attaque par indice de coïncidence mutuel est basée, comme son nom l'indique, sur l'indice de coïncidence. Cette fois-ci l'indice révèle la probabilité de tomber deux fois sur la même lettre, en tirant au hasard dans deux textes différents, à la même position. Le calcul de l'indice permet, en retrouvant les décalages des lettres, de passer d'un chiffrement poly-alphabétique à un mono-alphabétique.

La formule de l'indice de coïncidence mutuel est le suivant :

$$MIc(x,y) = \sum_{i=0}^{25} p_i(x) * p_{(i+dcalage)(\mod 26)}(y),$$

avec x et y deux textes différents, et $p_i = \frac{n_k}{n}$, $(0 \leq k \leq 25)$ qui représente l'occurrence de chaque lettre sur le nombre total de lettres dans le texte.

L'attaque consiste à utiliser ce calcul sur les sous textes, fait grâce à la longueur de la clé trouvée auparavant. Pour chaque couple de sous texte, on en fixe un qui prendra la place de x dans l'équation. Un couple est formé grâce à la liste des sous textes et la longueur de la clé. Soit (l_i, l_{i+1}) un couple tel que $0 \leq i \leq longueurCle$. Ensuite, on chiffre le deuxième sous texte, par la i^{me} lettre de l'alphabet à chaque tour i. La somme est faite, chaque tour de i sur toutes les lettres de l'alphabet. Pour retrouver le décalage de la clé, il suffit de regarder pour quel i, la somme est la plus élevée (i.e ≥ 0.065).

On obtient donc un décalage dans l'alphabet pour chaque couple. Pour retrouver la clé entière, il suffit de prendre les décalages des couples, dont x était le premier sous texte de la liste. Comme x était fixe, on le prend comme référence et donc le premier décalage d_1 est 0, soit $k_1 =' a'$. A partir de ça, on cherche la deuxième lettre grâce au couple (l_1, l_2), et ainsi de suite. Le calcul de la deuxième lettre par rapport au premier se fait ainsi :

$$k_2 = (k_1 - d_2) \mod 26 \; k_3 = (k_1 - d_3) \mod 26 \; ,$$

jusqu'à avoir toutes les lettres de la clé.

On obtient une clé de décalage relatif, mais ce n'est pas forcément la bonne. Il faut donc déchiffrer le message avec cette clé de décalage relatif. Puis faire une cryptanalyse de César dessus, pour retrouver le bon décalage de la première lettre et ainsi toutes les autres. Voici l'algorithme pseudo-code de l'attaque d'indice mutuel :

Algorithm 7 indiceDeCoïncidenceMutuel(message :**chaîne**, lgClé :**entier**)(**chaîne**, **chaîne**)

```
listTexts : list(chaîne)
listTexts ← listeSousTtexte(message, lgClé)
dictDécalages : dict(couple, entier)
dictDécalages ← calculIndMutuel(listTexts, lgClé)
if dictDécalages ≤ vide then
   (clé pas trouvée, mauvaise longueur de clé)
   return vide , message
end if
clé_decalageR : chaîne
clé_decalageR ← trouverCléDécalageRelatif(dictDécalages, lgClé)
msg_dechiffre : chaîne
msg_dechiffre ← déchiffreVigenere(message, clé_decalageR,..)
msg_attaque : chaîne
clé_Cesar : entier
(msg_attaque, clé_Cesar) ← cryptanalyseCesar(msg_dechiffre, ..)
return chiffreCésar(clé_decalageR, clé_Cesar), msg_attaque
```

On parvient donc à déchiffrer le message, qui devient, après application de l'algorithme de redéfinition des espaces :

> l'eurovision demeure l'un des plus anciens programmes televises au monde, e t le plus important concours musical jamais organise. s on succes a depasse les frontieres du continent europeen et i la inspire de nombreuses autres competitions musicales.

7 Bilan du projet

7.1 Choix des outils

Python a été notre choix de langage de programmation car c'est un langage polyvalent qui contient déjà beaucoup de structures utilisables dans notre projet. Nous pensions pouvoir utiliser la structure "dict" pour nos recherches dans nos dictionnaires mais elle s'est révélée inutile, nous avons donc dû créer notre structure nous-même. Notre choix avait été motivé par le fait que Python est un langage très utilisé dans le monde professionnel pour du scripting et que sa maîtrise sera un plus pour notre futur.

De même, l'utilisation de Git pour la gestion de projet a été motivée par le monde professionnel.

Nous avons aussi décidé de fournir une interface graphique de notre projet ainsi que des applications utilisables en ligne de commandes pour pouvoir, dans une éventuelle amélioration du projet, les utiliser au sein d'un site internet à travers une manipulation de fichiers côté serveur avec un outil tel que PHP.

7.2 Apports du projet

Ce projet nous a permis de produire un projet conséquent en groupe. Nous obligeant à s'organiser à plusieurs au moyen d'outils numériques. Pour nous trois, c'est le premier projet de cet ampleur, il nous a donc dû gérer notre temps et le stress.

Au niveau des technologies, ce projet nous a permis d'exploiter tout le potentiel de Python a travers les différentes bibliothèques standards ainsi que la bibliothèque graphique « tkinter ».

Pour Git, bien que les premières utilisations se sont révélées ardues, la prise en main à tout de même réussi à se faire.

Voici une capture d'écran de notre application graphique créée grâce à à tkinter. On peut remarquer que nous avons donné un grand choix des paramètres à l'utilisateur.

7.3 Conclusion

Nous sommes fiers d'avoir terminé ce projet dans les temps. Fonctionnel, il pourrait, pourquoi pas, supporter différentes améliorations d'implantations graphiques tel qu'un site web. Ce projet nous a permis de conforter notre projet d'orientation au sein du master « Sécurité des Systèmes d'Informations » de l'Université de Rouen. Découvrir la cryptographie a pu nous faire connaître un nouvel aspect de la sécurité que nous n'avions pas encore eu la chance de voir. Nous tenons à remercier Mme M. Bardet, pour sa patience et son aide précieuse durant toutes les étapes de notre projet.

7.4 Bibliographie

Wikipédia :
Cryptologie : Page consultée en mars 2018.
url : https://fr.wikipedia.org/wiki/Cryptologie
Cryptographie : Page consultée en mars 2018.
url : https://fr.wikipedia.org/wiki/Cryptographie
Cryptanalyse : Page consultée en mars 2018.
url : https://fr.wikipedia.org/wiki/Cryptanalyse
César :. Page consultée en mars 2018.
url : https://fr.wikipedia.org/wiki/Chiffrement_par_d%C3%A9calage
Vigenère : Page consultée en février 2018.
url : https://fr.wikipedia.org/wiki/Chiffre_de_Vigen%C3%A8re
Kasiski : Page consultée en février 2018.
url : https://fr.wikipedia.org/wiki/Cryptanalyse_du_chiffre_de_Vigen%C3%A8re
Analyse Fréquentielle : Page consultée en février 2018.
url : https://fr.wikipedia.org/wiki/Analyse_fr%C3%A9quentielle
Indice de Coïncidence : Page consultée en février 2018.
url : http://bibmath.net/crypto/index.php?action=affiche&quoi=complements%2Findice_coincidence

Simon Singh pour les exemples : Page consultée en mars 2018.
url : https://simonsingh.net/cryptography/cryptograms/
url : https://simonsingh.net/cryptography/cipher-challenge/the-ciphertexts/