Tentative de décryptement automatique du chiffre de Playfair

1 Introduction

De toutes les méthodes de cryptage ayant été inventées au cours des siècles, il en est une qui porte le nom de **chiffre de Playfair**. Comme son nom ne l'indique pas, c'est Sir Charles Wheatstone qui en est le précurseur. C'est en effet le 26 mars 1854 que l'on retrouve la première description de cette méthode de chiffrement dans un document signé Wheatstone. Cependant, le British Foreign Office le refuse en raison de sa trop grande complexité. Ce n'est que plus tard que Lyon Playfair donnera son nom à cette méthode en la popularisant. Le chiffre de Playfair est notamment utilisé pendant la Guerre des Boers, ou par les forces britanniques lors de la Première Guerre mondiale, ou encore par l'Australie durant la Seconde Guerre mondiale.

L'objectif de ce travail est de concevoir un programme informatique simple, écrit en Python, permettant de décrypter automatiquement le chiffre de Playfair. Le mot *simple* signifie ici que l'algorithme, l'idée de base, doit être simple. Le but est ainsi de démontrer que l'on peut casser ce code en apparence difficile à décoder en s'inspirant d'un concept primitif, et ce en exploitant la puissance du hasard.

2 Fonctionnement

2.1 On code avec une grille

Le chiffre de Playfair est un chiffre polygrammique permettant de coder un texte à l'aide d'une grille de chiffrement. Cette grille compte 5 lignes et 5 colonnes, ce qui lui permet de contenir 25 lettres : on y dispose toutes les lettres de l'alphabet (à l'exception du "W", qui n'est que très peu fréquent dans la langue française). On peut choisir un mot de passe formé de lettres pour créer la grille de chiffrement. Il suffit pour cela d'insérer dans la grille les lettres du mot de passe dans l'ordre, sans mettre deux fois la même lettre. On ajoute ensuite dans l'ordre alphabétique toutes les lettres de l'alphabet qui n'ont pas encoreété utilisées. Par exemple, si le mot de passe est *Jean-Paul Sartre*, la grille de chiffrement sera :

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

2.2 Chiffrement

Pour chiffrer un texte avec la méthode de Playfair, il faut préalablement avoir traité le texte, en supprimant tous les caractères qui ne sont pas des lettres, et en remplaçant chaque "W" par un "V" (cette dernière étape est valable pour les textes français ; on remplace plus souvent la lettre "I" par la lettre "J" ou l'inverse en anglais, et une autre variante consiste à omettre chaque occurrence de la lettre "Q"). Ceci étant fait, il faut grouper les lettres par paires de deux lettres différentes. Si un bigramme se compose de deux fois la même lettre, on insère une nulle (généralement la lettre "X" en français) entre ces deux lettres. S'il reste une lettre orpheline en fin de chaîne, on insère également une nulle après cette lettre, afin de pouvoir former un bigramme avec la dernière lettre pour pouvoir chiffrer celle-ci. On peut maintenant crypter chaque bigramme l'un après l'autre, en appliquant l'une des trois règles suivantes, suivant la situation dans laquelle on se trouve :

(1) Si les deux lettres claires se trouvent sur les deux coins opposés d'un rectangle, alors les lettres chiffrées sont sur les deux autres coins (en suivant la même ligne) : "UN" → "SH".

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

(2) Si les deux lettres claires sont sur la même ligne, alors les lettres chiffrées sont celles qui les suivent directement à leur droite : "CV" → "OE".

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

(3) Si les deux lettres claires sont sur la même colonne, alors les lettres chiffrées sont celles qui les suivent directement en-dessous : "GK" → "UV".

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

Remarque. Pour la règle (2), si une lettre se trouve dans la dernière colonne de la grille, il faut prendre comme lettre chiffrée la lettre correspondante qui se trouve dans la première colonne ; cela est également valable pour la règle 3 : si la lettre se trouve dans la dernière ligne de la grille, alors la lettre chiffrée sera la lettre correspondante, dans la première ligne de la grille.

Pour le déchiffrement, il suffit d'appliquer ces trois règles à l'envers.

3 Décryptement

3.1 Particularité

La particularité principale de ce chiffre réside dans sa façon de coder un texte par bigrammes de lettres au lieu de substitutions monoalphabétiques plus courantes (c'est-à-dire au lieu de coder chaque lettre l'une après l'autre toujours par le même symbole en effectuant des modifications plus ou moins complexes). Une analyse des fréquences des lettres est donc peu efficace ici, puisque chaque lettre apparaît dans 25 bigrammes, et deux bigrammes contenant une même lettre sont donc codés différemment.

On peut par contre attaquer un texte chiffré avec Playfair en prenant les bigrammes comme caractères distincts, à la place des lettres de l'alphabet (comme on le ferait dans le cas du chiffre de César par exemple). Ceci est bien plus compliqué que l'analyse des fréquences des lettres, car au lieu de 26 caractères, le nombre de bigrammes possibles vaut :

$$A_{25}^2 = 600 \text{ bigrammes.}$$

Il faut veiller à omettre les bigrammes composés de deux fois la même lettre, car Playfair insère une nulle entre deux lettres identiques. On doit ensuite tenter de reconstituer la grille de chiffrement.

3.2 Idée générale

Pour parvenir à décrypter automatiquement le chiffre de Playfair, on peut donc se baser sur les fréquences d'apparition des bigrammes de lettres dans la langue française. Par exemple, dans "Ananas", on a les bigrammes suivants : 2 fois le bigramme "AN", 2 fois "NA" et 1 fois "AS".

On peut alors théoriquement calculer les *fréquences d'apparition* de tous les bigrammes existants avec les 26 lettres de l'alphabet, et ce simplement en calculant les fréquences d'apparition des bigrammes d'un (très) long texte, et en les relevant ensuite dans un tableau (ce qui a été fait par Didier Müller [1] à cette adresse : `http://www.apprendre-en-ligne.net/crypto/stat/francais.html` avec un texte de 99'964 lettres), en omettant la lettre "W", celle-ci n'apparaissant pas avec ce chiffrement.

Bien sûr, le décryptement ne fonctionnera alors qu'avec un texte en langue française, car les fréquences varient d'une langue à l'autre.

1. Professeur de mathématiques et d'informatique au Lycée cantonal de Porrentruy (Jura, Suisse).

Ceci étant fait, on peut commencer par générer une grille de chiffrement aléatoire, puis permuter deux lettres au hasard dans cette grille. On vérifiera ensuite la "qualité" de la permutation, en comparant les *fréquences théoriques* des bigrammes (données par le tableau ci-dessus) et les *fréquences mesurées* des bigrammes à partir du cryptogramme décrypté à l'aide de la grille actuelle.

Voyons ceci plus précisément, étape par étape :
But. On veut décrypter un cryptogramme A.

(0) On génère d'abord aléatoirement une grille de (dé)chiffrement initiale.

(1) On permute deux lettres aléatoirement dans la grille actuelle.

(2) On décrypte le cryptogramme A à l'aide de la grille actuelle en un texte "clair" B.

(3) On mesure les fréquences des bigrammes dans le texte B, ce qui nous donne les fréquences mesurées des bigrammes.

(4) On donne un *score* à la grille (on verra plus bas comment définir ce score, qui est en fait un nombre ; retenez pour l'instant simplement que plus ce score est élevé, plus la grille correspondante est mauvaise).

(5) Si le score obtenu est supérieur au score minimal que l'on a déjà obtenu jusque là, on reprend l'ancienne grille.

(6) Sinon, on définit le score minimal comme étant égal au score actuel (puisque celui-ci est inférieur au score minimal). On recommence ensuite au point (1).

Ou de façon plus schématique :

(0) Composer une grille aléatoirement.

(1) Permuter deux lettres au hasard.

(2) Décrypter le cryptogramme à l'aide de la grille actuelle.

(3) Mesurer les fréquences des bigrammes.

(4) Donner un score à la solution.

(5) Si score > scoreMin :
Reprendre ancienne grille.

(6) Sinon :
scoreMin = score.

↪ Recommencer à (1) jusqu'à ce que scoreMin→0.

Nous obtenons ainsi le pseudo-code suivant :

```
scoreMin := 100        (* Nombre élevé *)
TANT QUE score < valeurMinimale FAIRE :
    permuter()
    decrypter()
    calculerFrequencesBigrammesMesurees()
    calculerScore()
    SI score > scoreMin ALORS :
        retablirGrille()
    SINON ALORS :
        scoreMin := score
    FIN SI
FIN TANT QUE
```

3.3 Calcul du score

On a vu précédemment que l'on attribuait un score à la grille, mais qu'est-ce au juste que ce score ? C'est en fait un simple nombre, qui nous renseigne sur la ressemblance de la grille que l'on a obtenue avec la grille qui a réellement été utilisée pour crypter le texte.

Pour calculer ce score, il suffit de calculer la différence qu'il y a entre la *fréquence théorique* d'apparition de chaque bigramme, et la *fréquence mesurée* d'apparition de chaque bigramme. Il faut donc calculer la différence entre la fréquence théorique et la fréquence mesurée d'apparition du bigramme AA, la même chose pour le bigramme AB, AC, AD, ..., le bigramme CV, CX, CY, CZ, DA, DB, ..., et ainsi de suite jusqu'au bigramme ZZ.

On obtient la formule suivante :

$$score = \sum_{i=1}^{25} \sum_{j=1}^{25} |t_{ij} - f_{ij}|$$

Avec : t : Les 625 ($= 25^2$) fréquences théoriques d'apparition des bigrammes.
$t_{i,j}$: La fréquence théorique d'apparition du bigramme formé des i-ième et j-ième lettres de l'alphabet.
f : Les 625 ($= 25^2$) fréquences mesurées d'apparition des bigrammes.
$f_{i,j}$: La fréquence mesurée d'apparition du bigramme formé des i-ième et j-ième lettres de l'alphabet.

4 Problème rencontré

Une fois le programme conçu selon le modèle énoncé ci-dessus [2], on constate que la recherche se bloque au bout d'un certain nombre d'itérations de la boucle principale. La fonction de recherche tombe en effet relativement rapidement dans un optimum local, car la méthode appliquée est du type de la *descente de plus grande pente* : seule une grille meilleure, c'est-à-dire une solution améliorant le caractère à optimiser, est acceptée.

La solution à ce problème consiste en l'exploration d'un espace de solutions plus étendu, en acceptant une partie des solutions dégradant le score de façon à ne pas s'enfermer dans un optimum local. La métaheuristique choisie ici est le *recuit simulé*.

4.1 Recuit simulé

Le recuit simulé est une métaheuristique inspirée d'un processus que l'on utilise en métallurgie. Ce processus a pour particularité d'alterner des cycles de refroidissement lent et de réchauffage (que l'on appelle recuit) qui tendent à minimiser l'énergie du matériau. On applique ce principe en informatique en vue de diminuer l'énergie de la caractéristique que l'on cherche à optimiser.

Par conséquent, lorsque l'on permute deux lettres, nous avons deux cas qui se présentent : soit le score de la grille ainsi obtenue est plus petit que le score de la grille précédente, auquel cas on acceptera toujours la permutation ; mais la différence avec la première conception du programme réside dans le cas où le score est plus élevé. En effet, jusqu'à présent, on rétablissait constamment la grille précédente ; or, on a remarqué que cette façon de faire nous enferme dans un optimum local. On va donc accepter, comme dit ci-dessus, une partie des *mauvaises* solutions.

État initial de l'algorithme. Dans le cas qui nous occupe, la solution initiale est la grille initiale, composée aléatoirement. L'énergie associée à chaque solution est ce que nous avons jusqu'à maintenant appelé (et nous continuerons d'utiliser ce terme) le *score de la grille* (qui correspond à la somme des différences des fréquences théoriques et mesurées des bigrammes).

Itérations de l'algorithme. À chaque itération de l'algorithme, une modification élémentaire de la solution est effectuée (c'est une permutation de deux lettres dans la grille). Cette modification entraîne une variation ΔE de l'énergie du système (toujours calculée à partir du critère que l'on cherche à optimiser). Si cette variation est négative (c'est-à-dire qu'elle fait baisser l'énergie du système), elle est appliquée à la solution courante. Sinon, elle est acceptée avec une probabilité valant

$$P = e^{-\frac{\Delta E}{T}}.$$

On itère ensuite selon ce procédé en gardant la température T constante.

On peut maintenant construire le pseudo-code de cet algorithme :

```
s := s0        (* état ; s0 : état initial *)
sbest := s     (* meilleure solution *)
e := E(s)      (* énergie *)
ebest := e     (* énergie la plus basse *)
k := 0         (* étape *)
tk := t0       (* température ; t0 : temp. initiale *)
TANT QUE k < kmax ET e > emax :
    permuter(s)
    ek := E(s)
    SI ek < ebest ALORS :
        sbest := s
        ebest := ek
    SINON SI aléatoire() < exp(-(ek-e) / tk)
    ALORS :
        e := ek
    SINON ALORS :
        retablir(s)
    k := k + 1
    diminuer(tk)
    FIN SI
FIN TANT QUE
RETOURNER sbest
```

2. Seuls les aspects fondamentaux de ce travail sont traités ici ; cette conception du programme en Python n'est donc pas détaillée dans ce rapport, mais est disponible en conultation libre sur Internet.

4.2 Permutations circulaires

Il enfin compter une dernière étape. En effet, en supposant que la grille puisse être reconstituée, il y a de très fortes chances qu'elle le soit "dans le désordre", c'est-à-dire que la base de la grille sera présente, mais pas sa forme final : plusieurs colonnes ou lignes peuvent être inversées. Pour remédier à ce problème, il suffit de tester les différentes permutations des lignes et des colonnes de la grille.

La grille ayant cinq lignes et colonnes, le nombre de permutations des lignes ou des colonnes séparément est égal à $P_5 = 5! = 120$, soit 120 permutations possibles des lignes.

Cependant, ce nombre peut être grandement réduit si l'on remarque que seules les permutations usuelles et rectilignes, et non les permutations circulaires entre-elles, qui nous intéressent ici. En effet, une grille A qui peut être obtenue depuis une grille B en décalant simplement toutes ses lignes et/ou colonnes d'un certain cran est semblable à cette grille B.

Pour simplifier les choses, on peut "aplatir" ce problème et transformer la grille en une suite de symboles, par exemples les chiffres de 0 à 4. Pour fixer les idées, on peut poser que chaque chiffre correspond à l'indice de la colonne correspondante. Dès lors, il suffit de rechercher toutes les permutations de ces chiffres en supprimant toutes les permutations circulaires entre-elles. Nous n'avons donc au total plus que le nombre de permutations des objets 1, 2, 3 et 4, le 0 étant fixé afin d'éviter les permutations circulaires ; ce qui fait $P_4 = 4! = 24$ permutations.

Des 120 permutations initiales, il n'en reste que 24. Cependant, ce nombre ne représente que le nombre de permutations des lignes seules ou des colonnes seules. Or, nous souhaitons, pour chaque ordre différent des lignes, modifier l'ordre des colonnes. Ainsi, pour la première permutation des lignes, on veut avoir la première permutation comme ordre des colonnes, puis la seconde, puis la troisième, et ainsi de suite. Donc, pour chacun des 24 ordres différents des lignes correspond 24 ordres pour les colonnes : à la première permutation des lignes en correspond 24 des colonnes, à la deuxième en correspond encore 24, etc. Il y a donc, finalement et au total 24^2 soit 576 permutations à tester, ce qui est 25 fois moins que les 1202 donc 14'400 permutations que l'on aurait pu tester sans prêter attention. Le temps d'exécution de cette étape aurait donc été 25 fois plus grand ; s'il faut 1 minute, il en aurait fallu environ 25 !

5 Résultats

5.1 Résultats finaux

Deux batteries de tests ont été effectuées, chaque batterie ayant été effectuée dans les mêmes conditions (même cryptogramme, même ordinateur, etc.). La première batterie effectuée comporte 12 essais, et la seconde en comporte 40. Ces tests ont été effectués dans les mêmes conditions, car, conséquence du hasard utilisé, un résultat différent sera obtenu à chaque nouvelle tentative. Il faut par conséquent répéter le même essai plusieurs fois afin d'obtenir des moyennes.

Voici une synthèse des résultats obtenus pour ces deux séries de tentatives de décryptement du message :

Taux de réussite :

(1) 12 essais : $6/12 = 50\%$.

(2) 40 essais : $19/40 = 47.5\%$.

Temps moyen :

(1) 12 essais : Sans échecs : 13.2 min en moy.
Avec échecs : 21.6 min en moy.

(2) 40 essais : Sans échecs : 12.1 min en moy.
Avec échecs : 21.5 min en moy.

Le taux de réussite est donc légèrement inférieur à **50%**.

En ce qui concerne le temps moyen, il est intéressant de distinguer le cas où la grille a été retrouvée et le cas contraire. En effet, il faut en moyenne 12 à 13 minutes pour trouver la grille, *en supposant* qu'elle soit bel et bien retrouvée à la fin. Si l'on omet cette hypothèse, il faudra alors en moyenne 21 à 22 minutes pour retrouver la bonne grille. Cela montre que lorsqu'un résultat est obtenu, c'est généralement rapidement, ce qui est une conséquence directe du hasard : en effet, si l'on a de la chance, le score va rapidement converger, et la grille sera

retrouvée ; si l'on n'en a pas, il y a une forte probabilité que le score stagne à une valeur relativement élevée, et que la convergence soit ralentie voire interrompue. Pourtant, grâce au recuit simulé, nous devrions dans presque tous les cas obtenir la bonne grille en laissant chercher indéfiniment, mais comme dit plus haut, mieux vaut relancer la recherche après un certain nombre d'essais si la grille n'a pas été retrouvée plutôt que d'attendre des heures que la bonne grille soit obtenue.

En conclusion, après plusieurs tests sur différents textes, le programme a trouvé la bonne grille de chiffrement un peu moins d'une fois sur deux, ce qui est concluant au vu de l'objectif fixé. Il faut évidemment que le texte fourni soit assez long. Par ailleurs, le temps de recherche est en moyenne de 20 minutes (que la bonne grille ait été trouvée ou non) avec un texte d'une longueur d'un peu plus de 3'000 caractères.

5.2 Paramètres idéaux

De nombreuses modifications des paramètres fournis à l'algorithme ont été effectuées, et la meilleure configuration obtenue est la suivante :

```
scoreBut = 0.3     (* Score à atteindre *)
kMax = 12000       (* Nombre maximal d'essais *)
t0 = 10            (* Température initiale *)
coef = 0.95        (* Cœfficient de décroissance *)
palier = 100       (* Palier de température *)
```

Ci-dessous se trouvent les paramètres qui ont été modifiés pour effectuer des tests (avec entre parenthèses les valeurs testées) :

- Température initiale (0.1 ; 1 ; 5 ; 10 ; 20 ; 100 ; 1000).
- Cœfficient de décroissance de la température (0.3 ; 0.6 ; 0.85 ; 0.9 ; 0.99 ; 0.995 ; 0.999).
- Paliers de température (1 ; 10 ; 50 ; 100 ; 1000).
- Nombre de lettres échangées lors d'une permutation (2 ; 3 ; 4 ; 10 ; 15).
- Seuil limite d'acceptation (différentes méthodes de fixations du seuil, en fonction : du score ; du temps ; à la fois du temps et du score ; en prenant le score minimal ou le score précédent).

Une solution afin d'augmenter la probabilité de réussite serait de faire baisser la température plus lentement, mais le temps de recherche augmente très rapidement, et mieux vaut relancer la recherche en cas d'échec plutôt que de laisser rechercher plusieurs heures. Une dernière modification apportée au programme afin d'optimiser la recherche est de réchauffer le système si celui-ci se fige trop rapidement. Ainsi, si plus aucune meilleure grille n'a été trouvée depuis longtemps et que le score reste élevé, on élève à nouveau la température.

Ensuite, l'affichage graphique de la recherche [3] ralentit indubitablement le processus : en effet, le temps de recherche est en moyenne réduit de moitié sans graphismes, soit 8 minutes au lieu de 15 pour un texte de 3'000 caractères.

6 Conclusion

L'objectif de ce travail était de programmer une méthode simple de décryptement automatique d'un cryptogramme chiffré avec la méthode de Playfair, cette méthode devant être basée sur le hasard. La priorité n'était pas le temps que prend la recherche, mais la simplicité de la méthode. Or, l'idée originelle est on ne peut plus simple : on permute aléatoirement deux lettres dans la grille, et on vérifie la ressemblance entre les fréquences des bigrammes de lettres du message déchiffré et les bigrammes des lettres dans la langue française. Il a par la suite fallu intégrer le principe du recuit simulé afin d'empêcher le processus de s'enfermer dans des optima locaux.

Le temps de recherche est long, mais comme précisé ci-dessus, la priorité n'était pas à la minoration du temps. Par ailleurs, rien ne garantit d'obtenir à tous les coups une grille de déchiffrement optimale, c'est-à-dire *la* grille de chiffrement utilisée. Cela vient du fait que l'algorithme de recherche est basé sur le hasard, il faut donc avoir une part de chance pour trouver la bonne grille rapidement. Mais l'objectif de ce travail, c'est-à-dire parvenir à décoder au moins une fois un message à l'aide de cette application, a été atteint.

3. Encore une fois, se rapporter à une documentation plus détaillée pour plus de précisions sur le programme en lui-même.

7 Bibliographie

7.1 Ouvrages consultés

- Müller Didier, *Les codes secrets décryptés*, City Editions, février 2007.
- Swimmen Gérard, *Apprendre à programmer avec Python*, O'Reilly, mai 2005.

7.2 Sites Internet consultés

- `http://www.apprendre-en-ligne.net/crypto/subst/playfair.html`,
  Apprendre-en-ligne : Le chiffre Playfair,
  23 août 2009.
- `http://lwh.free.fr/pages/algo/crypto/playfair.htm`,
  Algorithmes de cryptage : "Playfair",
  23 août 2009.
- `http://fr.wikipedia.org/wiki/Chiffre_de_Playfair`,
  Wikipédia : Chiffre de Playfair,
  9 janvier 2010.
- `http://www.jw-stumpel.nl/playfair.html`,
  Classical Cryptography,
  9 janvier 2010.
- `http://www.apprendre-en-ligne.net/OCinfo/algo/recuit.html`,
  Apprendre-en-ligne : Le recuit simulé,
  5 décembre 2009.
- `http://fr.wikipedia.org/wiki/Recuit_simulé`,
  Wikipédia : Recuit simulé,
  5 décembre 2009.
- `http://python.developpez.com/`,
  Python - Club des décideurs et professionnels en informatique,
  3 février 2010.