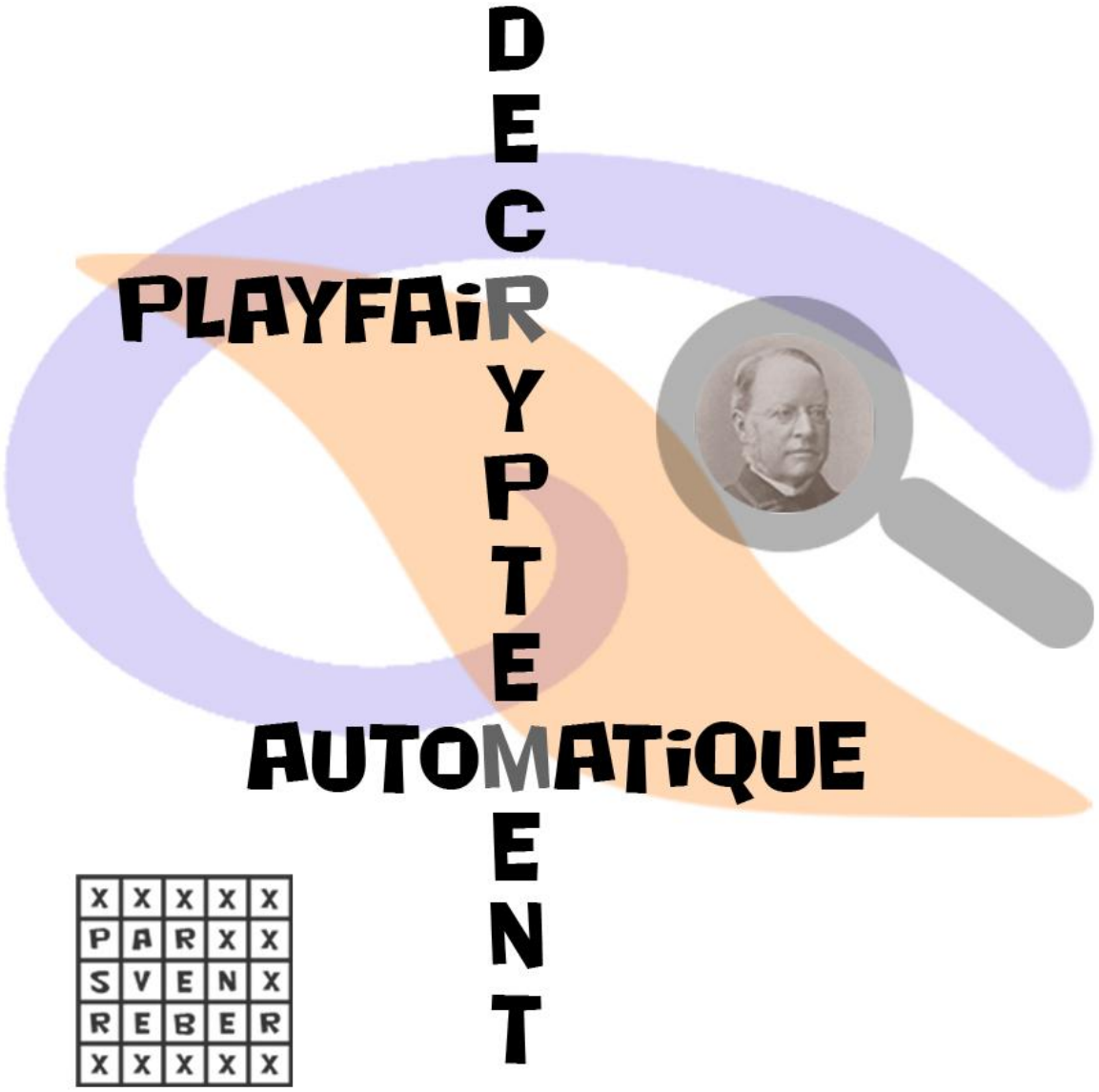
Travail de maturité 2009-2010
Mathématiques et Informatique
Lycée cantonal de Porrentruy

Tentative de décryptement automatique du chiffre de Playfair



Auteur : Sven Reber
Maître responsable : Didier Müller
Expert : Damien Dobler

Tentative de décryptement automatique du chiffre de Playfair

TABLE DES MATIÈRES

1. INTRODUCTION

Après avoir vu et étudié différentes méthodes de cryptage en cours d'application des mathématiques, il m'a paru intéressant de concevoir un programme basé sur le hasard permettant de décrypter automatiquement le chiffre de Playfair. Ceci sera donc l'objet de ce travail de maturité. Le programme sera écrit en langage Python, ce dernier ayant été vu et appris en cours d'option complémentaire informatique.

L'objectif de ce travail est donc de fournir un programme simple de recherche permettant de décrypter le chiffre de Playfair, sans qu'il ne soit forcément d'une haute efficacité. 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 le hasard.

On procédera en plusieurs étapes, en commençant par rechercher une méthode simple permettant *a priori* de décrypter un code Playfair, puis en programmant cet algorithme grâce au langage Python. On analysera ensuite les résultats obtenus, afin d'apporter une éventuelle modification du concept de base dans le cas où cette modification resterait primitive, puis de formuler une conclusion à ce travail.

Il convient cependant d'expliquer en premier lieu ce qu'est le chiffre de Playfair, et quel est son fonctionnement, c'est donc par-là que nous débuterons le travail.

2. Le chiffre de Playfair

2.1 Historique

Comme son nom ne l'indique pas, le chiffre de Playfair (ou chiffre Playfair) a été inventé par Sir Charles Wheatstone (photo ci-contre, à gauche). C'est en effet le 26 mars 1854 que l'on retrouve la première description de cette méthode de chiffrement dans un document écrit par Wheatstone, l'un des pionniers du télégraphe électrique. Mais le British Foreign Office le refuse, à cause de sa trop grande complexité.

Lorsque Wheatstone a proposé de prouver qu'il pouvait le faire comprendre aux enfants de l'école voisine en moins d'un quart d'heure, on lui répond qu'il y parviendra peut-être, mais qu'il ne pourra pas le faire comprendre à des spécialistes.

Son nom lui viendra plus tard de Lyon Playfair (photo ci-contre, à droite), premier Baron Playfair de St. Andrews, qui va le populariser.

Il est déjà utilisé lors de la Guerre des Boers et lors de la Première Guerre mondiale par les forces britanniques, et il le sera également par les Australiens durant la Seconde Guerre mondiale.

Pourtant, il faut attendre 1914 pour que la première solution soit décrite dans un essai de 19 pages écrit par le Lieutenant Joseph O. Mauborgne.

2.2 Fonctionnement

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 pour crypter avec le chiffre de Playfair ; ce mot de passe doit contenir des lettres. 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 les autres lettres de l'alphabet qui n'ont pas encore été utilisées dans leur ordre alphabétique. Par exemple, si le mot de passe est *"Jean-Paul Sartre"*, la grille de chiffrement sera :

J	E	A	N	P	On insère les lettres composant le mot de passe au début de la grille.
U	L	S	R	T	
B	C	D	F	G	Et on remplit le reste de la grille avec les lettres restantes dans leur ordre alphabétique.
H	I	K	M	O	
Q	V	X	Y	Z	

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 décryptement, il suffit d'appliquer ces trois règles à l'envers (voir § 3.2, page 16 de ce document).

Exemple

Chiffrons le texte :
"Playfair code des bigrammes.".

… À l'aide du mot de passe :
"Georges".

… Qui nous donne la grille de chiffrement :

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

Commençons par traiter le texte en éliminant tout caractère qui n'est pas une lettre. On a :
"PLAYFAIRCODEDESBIGRAMMES".

On peut maintenant grouper les lettres par bigrammes, en insérant une nulle ("X") pour séparer deux lettres identiques (ici "MM"), et une nulle à la fin de la chaîne pour pouvoir crypter la dernière lettre (ici "S") :
"PL AY FA IR CO DE DE SB IG RA M**X** ME S**X**".

Il suffit alors de coder chaque bigramme à l'aide de la règle qui convient à la situation dans laquelle se trouve chacun des bigrammes :

- *PL* forment les deux coins opposés d'un rectangle, on prend donc les deux autres coins → *TJ*.
- *AY* → *DU*.
- *FA* sont sur la même ligne, donc on prend les deux lettres directement à leur droite → *AB*.
- *IR* → *KE*.
- *CO* se situent dans la même colonne, on prend donc les deux lettres juste en-dessous → *JC*.
- *DE* → *BR* ; *DE* → *BR* ; *SB* → *EF* ; *IG* → *HE* ; *RA* → *GD* ; *MX* → *PU* ; *ME* → *NG* ; *SX* → *OZ*.

Finalement, cela nous donne le cryptogramme suivant :
"TJDUABKEJCBRBREFHEGDPUNGOZ".

2.3 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 compliquées). Une analyse des fréquences des lettres est donc peu efficace ici, puisque chaque lettre apparaît dans 25 bigrammes, et chaque bigramme est codé différemment, même lorsqu'il contient la même lettre. Par exemple, en prenant la grille suivante :

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

Dans cette grille, le bigramme "LP" sera chiffré "AO", alors que le bigramme "LQ" sera chiffré "BO". Si l'on se concentre sur les lettres, la lettre "L" sera donc chiffrée la première fois "A", et la seconde fois "B", ce qui démontre qu'une analyse des fréquences des lettres est peu efficace (pas complètement inutile cependant, car il faut noter que la lettre "L" sera chiffrée de la même manière avec toutes les lettres d'une même colonne, en enlevant celle qui se trouve sur la même ligne que cette lettre (ici, avec la lettre "L", par exemple "R", "J", "Q" et "Y"), et également avec les lettres se trouvant sur la même colonne qu'elle (ici "U", "A", "B" et "C" pour la lettre "L") ; cela n'est cependant pas assez fiable pour se contenter de ce type d'analyse).

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. DÉCRYPTEMENT DU CHIFFRE DE PLAYFAIR

3.1 Idée générale

Pour parvenir à décrypter automatiquement le chiffre de Playfair, on peut 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 déjà été fait ici : http://www.apprendre-en-ligne.net/crypto/stat/francais.html avec un texte de 99'964 lettres), en enlevant bien sûr la lettre "W", car celle-ci n'apparaît pas lorsque l'on crypte avec Playfair (voir §2.2) :

	_A	_B	_C	_D	_E	_F	_G	_H	_I	_J	_K	_L	_M	_N	_O	_P	_Q	_R	_S	_T	_U	_V	_X	_Y	_Z
A_	31	242	392	208	48	135	232	37	1255	32	7	663	350	1378	17	412	44	905	409	613	599	301	6	69	12
B_	158	2	1	2	130	1	2	0	132	4	10	181	1	1	146	1	3	187	29	16	44	3	0	4	0
C_	312	0	73	19	765	2	2	411	209	3	5	124	5	1	677	11	7	100	14	142	132	2	0	11	0
D_	427	1	8	24	2409	2	5	25	378	3	0	14	21	5	231	4	6	134	64	3	406	4	0	5	0
E_	616	176	917	998	782	258	209	67	179	96	8	1382	1056	2121	136	699	190	1514	3318	1307	761	258	125	15	60
F_	181	1	1	8	180	118	1	1	190	0	0	43	1	1	213	1	2	106	12	1	61	0	0	1	0
G_	135	1	10	9	408	4	63	3	69	6	4	74	10	103	47	5	1	197	12	23	81	1	0	2	0
H_	267	5	4	1	285	0	0	0	149	3	0	3	4	17	107	0	3	18	5	0	42	0	0	7	0
I_	176	85	203	172	1030	114	115	6	49	14	0	798	181	797	524	75	215	400	897	1243	11	190	40	0	4
J_	76	0	0	0	100	0	0	0	2	0	0	0	0	0	91	0	0	0	0	0	42	0	0	2	0
K_	8	0	0	0	6	0	3	0	6	0	0	0	10	3	9	0	0	5	1	0	0	0	0	3	0
L_	1270	14	22	58	2366	25	14	39	512	4	1	647	18	41	281	69	47	16	126	42	369	14	0	15	1
M_	510	152	11	11	1099	0	1	1	302	0	0	7	243	4	334	201	2	10	10	8	52	1	0	3	0
N_	405	30	438	785	985	124	222	24	316	17	7	89	68	249	303	130	82	55	846	1694	114	109	1	19	20
O_	6	83	88	101	46	32	115	7	452	14	3	184	391	1646	8	175	19	491	126	109	1086	28	4	62	4
P_	671	1	3	21	441	5	1	136	119	0	0	377	2	4	505	125	1	363	31	65	140	1	0	1	0
Q_	2	0	3	0	1	0	0	1	0	0	0	1	3	0	0	1	0	1	0	0	975	0	0	0	0
R_	896	53	168	302	1885	46	96	5	583	11	3	292	181	88	520	82	51	176	386	445	183	77	1	21	5
S_	809	85	306	735	1377	151	73	83	565	36	0	453	192	107	521	496	191	137	702	578	343	92	6	30	10
T_	881	25	166	515	1484	52	19	64	984	28	3	331	70	40	363	268	96	668	404	269	270	41	6	18	3
U_	168	87	165	162	781	40	83	4	534	41	3	302	128	516	19	184	15	980	591	469	14	177	264	8	4
V_	277	0	1	0	502	0	0	0	288	0	0	1	0	0	167	0	0	81	0	0	11	0	0	0	0
X_	35	14	37	36	68	8	7	5	57	0	0	21	15	3	7	56	11	3	15	35	2	18	4	0	0
Y_	63	0	7	7	59	3	4	0	0	0	0	13	8	5	15	14	0	10	75	9	2	4	0	0	0
Z_	8	0	2	6	49	3	1	0	1	1	0	11	4	2	15	4	1	0	3	1	0	7	0	0	2

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

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 :

⇒ On doit décrypter un cryptogramme, nommons le *A*.
0. On génère 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.
3. On calcule les fréquences des bigrammes mesurées.
4. On donne un score à la grille (on verra plus bas comment définir ce score, qui est 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.

De façon plus schématique :

0. Composer une grille aléatoire.
1. Permuter deux lettres au hasard.
2. Décrypter le cryptogramme à l'aide de la grille actuelle.
3. Calculer les fréquences des bigrammes mesurées.
4. Donner un score à la solution.
5. Si score > score_min :
   Reprendre ancienne grille.
6. Sinon :
   score_min = score.

Recommencer jusqu'à ce que score_min ⟶ 0

Le pseudo-code serait donc le suivant :

```
score_min := 100      (* Nombre élevé *)
TANT QUE score < valeur_minimale FAIRE :
    permuter()
    decrypter()
    calculer_frequences_bigrammes_mesurees()
    calculer_score()
    SI score > score_min ALORS :
        retablir_grille()
    SINON :
        score_min := score
    FIN SI
FIN TANT QUE
```

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 simplement un 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* de chaque bigramme, et la *fréquence mesurée* 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.
Ceci est appliqué par la formule :

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

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

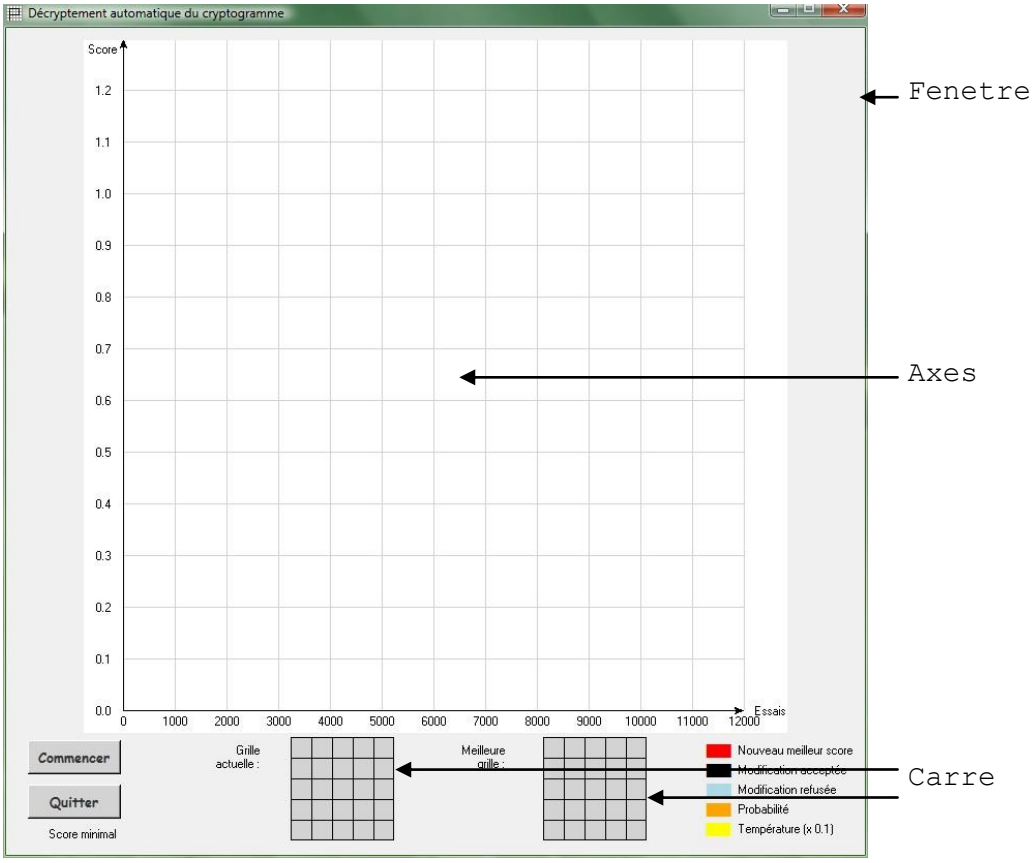
3.2 Première tentative de conception du programme

Nous nous intéresserons essentiellement au programme de recherche de la grille de chiffrement en lui-même, et peu à l'affichage du programme qui se fait à l'aide du module `Tkinter`, car cela ne nous intéresse pas ici et sortirait du cadre de ce travail, puisque cela reviendrait à fournir de simples informations de programmation Python, et qu'il existe des ouvrages spécialisés pour cet aspect.

Le programme contiendra quatre classes, et quelques fonctions supplémentaires, que l'on appellera "fonctions satellites" dans la mesure où elles n'appartiennent à aucune classe, mais sont invoquées par celles-ci. Les différentes classes sont les suivantes :

- `Fenetre` : C'est dans cet objet que seront affichés tous les composants de l'application.
- `Axes` : C'est le canevas principal, dans lequel s'afficheront les courbes lors de la tentative de décryptement.
- `Carre` : Ceci est simplement l'affichage en tant réel de la grille actuelle.
- `Grille` : Cet élément n'est pas visible directement ; c'est la grille de chiffrement qui contiendra plusieurs méthodes s'y appliquant (notamment la fonction de permutation de deux lettres).

Aperçu de la `Fenetre` principale du programme, qui comporte notamment les `Axes` et le `Carre` :



Pour commencer notre programme, la première chose à faire est d'importer les modules `math` et `random` (et `Tkinter` pour l'affichage):

```
from math import *
from random import *
import Tkinter
```

Nous allons en premier lieu présenter compendieusement les trois premières classes du programme, qui s'occupent de l'affichage ; on laissera ensuite celui-ci de côté pour s'intéresser au décryptement proprement dit.

La classe `Fenetre` est l'affichage principal du programme :

```
class Fenetre(Tkinter.Tk):
    def __init__(self, parent = None):
        Tkinter.Tk.__init__(self)
        self.parent = parent
        self.can = Axes(larg = 680, haut = 670)
        self.can.grid(row = 0, column = 0, columnspan = 8)
        self.commencer = Tkinter.Button(master = self, text = 'Commencer', command =
recherche, width = 10, height = 1, bg = 'light grey', fg = '#333333', font = ('Comic Sans
MS', '10', 'bold'))
```

```
        self.commencer.grid(row = 1, column = 0, padx = 5, pady = 5)
        self.quitter = Tkinter.Button(master = self, text = 'Quitter', command = quitter,
width = 10, height = 1, bg = 'light grey', fg = '#333333', font = ('Comic Sans MS', '10',
'bold'))
        self.quitter.grid(row = 2, column = 0, padx = 5, pady = 5)
        self.affichage = Tkinter.Label(master = self, text = 'Score minimal')
        self.affichage.grid(row = 4, column = 0, columnspan = 2)
        Tkinter.Label(master = self, text = 'Grille actuelle :', wraplength = 50, width =
13, anchor = Tkinter.E, justify = Tkinter.RIGHT).grid(row = 1, column = 2)
        self.carre = Carre()
        self.carre.grid(row = 1, rowspan = 4, column = 3)
        self.carre.afficher(5*[5*['']])
        Tkinter.Label(master = self, text = 'Meilleure grille :', wraplength = 50, width =
13, anchor = Tkinter.E, justify = Tkinter.RIGHT).grid(row = 1, column = 4)
        self.carre_best = Carre()
        self.carre_best.grid(row = 1, rowspan = 4, column = 5)
        self.carre_best.afficher(5*[5*['']])
        legende = Tkinter.Frame(master = self)
        legende.grid(row = 1, column = 6, rowspan = 4)
        Tkinter.Label(master = legende, text = '   ', fg = 'red', anchor = Tkinter.E, width
= 10).grid(row = 0, column = 0)
        Tkinter.Label(master = legende, text = '   ', fg = 'black', anchor = Tkinter.E,
width = 10).grid(row = 1, column = 0)
        Tkinter.Label(master = legende, text = '   ', fg = 'light blue', anchor =
Tkinter.E, width = 10).grid(row = 2, column = 0)
        Tkinter.Label(master = legende, text = '   ', fg = 'orange', anchor = Tkinter.E,
width = 10).grid(row = 3, column = 0)
        Tkinter.Label(master = legende, text = '   ', fg = 'yellow', anchor = Tkinter.E,
width = 10).grid(row = 4, column = 0)
        Tkinter.Label(master = legende, text = 'Nouveau meilleur score', anchor =
Tkinter.W, width = 20).grid(row = 0, column = 1)
        Tkinter.Label(master = legende, text = 'Modification acceptée', anchor = Tkinter.W,
width = 20).grid(row = 1, column = 1)
        Tkinter.Label(master = legende, text = 'Modification refusée', anchor = Tkinter.W,
width = 20).grid(row = 2, column = 1)
        Tkinter.Label(master = legende, text = 'Probabilité', anchor = Tkinter.W, width =
20).grid(row = 3, column = 1)
        Tkinter.Label(master = legende, text = 'Température (x 0.1)', anchor = Tkinter.W,
width = 20).grid(row = 4, column = 1)
        self.grid_columnconfigure(0, weight = 2)
        self.grid_columnconfigure(1, weight = 1)
        self.grid_columnconfigure(2, weight = 2)
        self.grid_columnconfigure(3, weight = 1)
        self.grid_columnconfigure(4, weight = 2)
        self.grid_columnconfigure(5, weight = 2)

    def etape_1(self):
        "Modifie l'affichage des boutons pour la première étape de la recherche."
        self.afficher = Tkinter.Button(master = self, text = 'Afficher', command =
afficher, width = 10, height = 1, bg = 'light grey', fg = '#333333', font = ('Comic Sans
MS', '10', 'bold'))
        self.afficher.grid(row = 1, column = 1, padx = 5, pady = 5)
        self.termine = Tkinter.Button(master = self, text = 'Terminé', command = arreter,
width = 10, height = 1, bg = 'light grey', fg = '#333333', font = ('Comic Sans MS', '10',
'bold'))
        self.termine.grid(row = 2, column = 1, padx = 5, pady = 5)
        self.commencer.destroy()

    def etape_2(self):
        "Modifie l'affichage des boutons pour la seconde étape de la recherche."
        self.afficher.destroy()
        self.termine.destroy()
```

La classe `Axes` correspond au canevas central de la fenêtre, dans lequel s'affichent des points représentant les scores des essais de grilles (voir le mode d'emploi (annexe 2) pour plus de précisions):

```
class Axes(Tkinter.Canvas):
    def __init__(self, parent = None, larg = 840, haut = 720):
        Tkinter.Canvas.__init__(self, width = larg, height = haut)
        self.parent = parent
        self.larg = larg
        self.haut = haut
        self.graduations()
        self.axes()

    def axes(self):
        "Affiche les deux axes principaux."
        self.create_line(40, self.haut-20, self.larg-40, 700, fill = 'black', arrow =
Tkinter.LAST)
        self.create_text(self.larg-15, self.haut-20, text = 'Essais')
        self.create_line(40, self.haut-20, 40, 0, fill = 'black', arrow = Tkinter.LAST)
        self.create_text(20, 10, text = 'Score')

    def graduations(self):
        "Affiche les graduations des axes (textes et lignes secondaires)."
        for i in range(0, self.haut-20, 50):
            self.create_line(40, self.haut-20-i, self.larg-40, self.haut-20-i, fill =
'light grey')
            self.create_text(20, self.haut-20-i, text = i/500.)
        for i in range(0, self.larg-40, 50):
            self.create_line(40+i, 0, 40+i, self.haut-20, fill = 'light grey')
            self.create_text(40+i, self.haut-10, text = 20*i)
```

La classe `Carre` est l'affichage en temps réel de la grille :

```
class Carre(Tkinter.Canvas):
    def __init__(self, parent = None, cote = 100):
        Tkinter.Canvas.__init__(self, master = parent, width = cote, height = cote, bg =
'light grey')
        self.parent = parent
        self.cote = cote

    def quadrillage(self):
        "Dessine le quadrillage de la grille."
        for i in range(2, self.cote, self.cote/5):
            self.create_line(2, i, self.cote+2, i, fill = 'black')
            self.create_line(i, 2, i, self.cote+2, fill = 'black')
        self.create_line(2, self.cote+1, self.cote+2, self.cote+1, fill = 'black')
        self.create_line(self.cote+1, 2, self.cote+1, self.cote+2, fill = 'black')

    def afficher(self, grille):
        "Affiche les caractères d'une [grille] dans le tableau."
        self.delete(Tkinter.ALL)
        self.quadrillage()
        for i in range(5):
            for j in range(5):
                self.create_text((j+0.6)*self.cote/5., (i+0.6)*self.cote/5., text =
grille[i][j])
```

Et enfin, la classe `Grille` qui enregistrera par exemple la grille de chiffrement et qui effectuera sur celle-ci les opérations élémentaires telles que le chiffrement, le déchiffrement et les permutations, est la suivante :

```
class Grille:
    def __init__(self, parent = None):
        self.parent = parent
        self.composer_grille_aleatoire()
        self.convertir_lettres()

    def composer_grille_aleatoire(self):
        "Composition d'une grille chiffrante (de nombres) aléatoire."
        self.grille = []
        for i in range(5):
            ligne = []
            for j in range(5):
                ok = 'non'
                while(ok == 'non'):
                    ok = 'oui'
                    nb = randrange(25)
                    for ligneS in self.grille :
                        for nombres in ligneS :
                            if(nombres == nb):
                                ok = 'non'
                    for nombres in ligne :
                        if(nombres == nb):
                            ok = 'non'
                ligne.append(nb)
            self.grille.append(ligne)

    def convertir_lettres(self):
        "Convertir une [grille] de nombres en grille de lettres."
        for i in range(5):
            for j in range(5):
                self.grille[i][j] = alphabet[self.grille[i][j]]

    def permuter(self):
        "Permute deux lettres aléatoirement dans la [grille]."
        ok = 'non'
        while(ok == 'non'):
            ok = 'oui'
            self.ligne_1 = randrange(5)
            self.colonne_1 = randrange(5)
            self.ligne_2 = randrange(5)
            self.colonne_2 = randrange(5)
            if(self.ligne_1 == self.ligne_2 and self.colonne_1 == self.colonne_2):
                ok = 'non'
        self.grille[self.ligne_1][self.colonne_1],
self.grille[self.ligne_2][self.colonne_2] = self.grille[self.ligne_2][self.colonne_2],
self.grille[self.ligne_1][self.colonne_1]

    def decrypter_bigrammes(self, texte):
        "Décryptement des bigrammes du [texte] chiffré avec la [grille] de chiffrement."
        coordonnees = self.remplacer_coordonnees(texte)
        texte_clair = ''
        l = 0
        while(l < len(texte)):
            if(coordonnees[l][0] == coordonnees[l+1][0]):
                texte_clair += self.grille[coordonnees[l][0]][(coordonnees[l][1]-1)%5]
                texte_clair += self.grille[coordonnees[l+1][0]][(coordonnees[l+1][1]-1)%5]
            elif(coordonnees[l][1] == coordonnees[l+1][1]):
                texte_clair += self.grille[(coordonnees[l][0]-1)%5][coordonnees[l][1]]
                texte_clair += self.grille[(coordonnees[l+1][0]-1)%5][coordonnees[l+1][1]]
            else:
                texte_clair += self.grille[coordonnees[l][0]][coordonnees[l+1][1]]
```

```
                texte_clair += self.grille[coordonnees[l+1][0]][coordonnees[l][1]]
            l += 2
        return texte_clair

    def remplacer_coordonnees(self, texte):
        "Remplacement de chaque lettre du message codé par ses coordonnées dans la grille :
[[ligne_lettre0, colonne_lettre0] , [ligne_lettre1, colonne_lettre1], ...]."
        coordonnees = []
        for l in range(len(texte)):
            groupe = []
            for i in range(5):
# Recherche des "coordonnées" de la l-ième lettre du texte.
                for j in range(5):
                    if(texte[l] == self.grille[i][j]):
                        groupe.append(i)
                        groupe.append(j)
            coordonnees.append(groupe)
        return coordonnees

    def retablir(self):
        "Rétablit la [grille] en annulant la dernière permutation."
        self.grille[self.ligne_1][self.colonne_1],
self.grille[self.ligne_2][self.colonne_2] = self.grille[self.ligne_2][self.colonne_2],
self.grille[self.ligne_1][self.colonne_1]
```

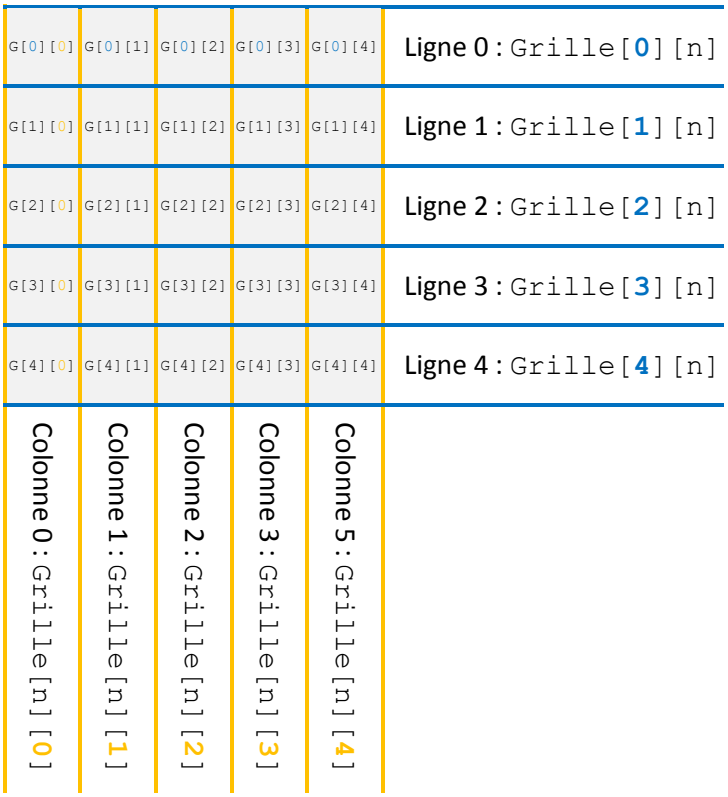
Nous allons maintenant construire petit à petit le reste du programme, et en particulier la classe `Grille` vue ci-dessus. On peut commencer par créer une fonction que l'on intégrera à la classe `Grille`, et qui se chargera de composer aléatoirement la grille initiale :

```
def composer_grille_aleatoire(self):
    "Composition d'une grille chiffrante (de nombres) aléatoire"
    self.grille = []                        # Création d'une nouvelle grille.
    for i in range(5):                      # Ajout de 5 lignes.
        ligne = []                          # Création d'une nouvelle ligne (vide).
        for j in range(5):                  # Ajout de 5 nombres dans la ligne.
            ok = 'non'                      # "Capteur" pour savoir si le nombre est "faux".
            while(ok == 'non'):
                ok = 'oui'
                nb = randrange(25)
                for ligneS in self.grille :
                    for nombres in ligneS :
                        if(nombres == nb):
                            ok = 'non'
                for nombres in ligne :
                    if(nombres == nb):
                        ok = 'non'
            ligne.append(nb)
        self.grille.append(ligne)
```

Explication du script :

La grille sera une liste à deux dimensions : une liste principale qui contiendra elle-même d'autres listes. La liste principale contiendra cinq listes, qui formeront les cinq colonnes de la grille. Chacune de ces cinq listes contiendra cinq éléments, qui seront les cinq cases de chaque ligne.

Si l'on représente schématiquement cette liste à deux dimensions, on obtient une représentation plus "réelle" de la grille de chiffrement, ce qui aide à la compréhension :



Ainsi, la grille de chiffrement :

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

Deviendra la liste :

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

On crée donc une fonction qui génère une grille aléatoire. La grille sera composée de nombres de 0 à 24 (on remplacera ensuite chaque nombre par la lettre se trouvant au rang, dans l'alphabet, désigné par ce nombre) :

1. On crée d'abord une liste vide, c'est celle qui contiendra toutes les autres listes :
```
grille = []
```
2. On crée encore une liste vide, c'est une ligne :
```
ligne = []
```
3. On y insère un nombre, en vérifiant qu'il ne soit déjà présent ni dans la grille, ni dans la ligne elle-même :
```
ok = 'non'
while(ok == 'non'):
    ok = 'oui'
    nb = randrange(25)
    for ligneS in grille :
        for nombres in ligneS :
            if(nombres == nb):
                ok = 'non'
    for nombres in ligne :
        if(nombres == nb):
            ok = 'non'
ligne.append(nb)
```
On répète l'opération 3 cinq fois (pour ajouter cinq nombres, donc cinq colonnes).

On répète l'opération 2 cinq fois (pour ajouter cinq lignes).

4. La liste `grille` a été générée.

On a maintenant besoin de faire une fonction pour convertir les nombres de la grille en lettres, et on intègre ensuite cette fonction à la classe `Grille` :

```
def convertir_lettres(self):
    "Convertir une [grille] de nombres en grille de lettres."
    for i in range(5):
        for j in range(5):
            self.grille[i][j] = alphabet[self.grille[i][j]]
```

Explication du script :

Cette fonction prend chaque nombre par la lettre qui lui correspond dans l'alphabet. Pour ce faire, chaque nombre n de la liste à deux dimensions `grille`, est remplacé par la lettre se trouvant à la $n^{ème}$ position dans la liste `alphabet` (qui est une variable définie en-dehors des classes, contenant simplement les 25 lettres de l'alphabet, sans le "W") :

```
self.grille[i][j] = alphabet[self.grille[i][j]]
```

La valeur de la liste `grille` ne comporte cette fois non plus des nombres, mais des lettres.

Il faut maintenant que l'on puisse permuter deux lettres au hasard. On crée pour cela une nouvelle fonction, qui viendra également s'ajouter à la classe `Grille` :

```
def permuter(self):
    "Permute deux lettres aléatoirement dans la [grille]"
    ok = 'non'
    while(ok == 'non'):
        ok = 'oui'
        self.ligne_1 = randrange(5)
        self.colonne_1 = randrange(5)
        self.ligne_2 = randrange(5)
        self.colonne_2 = randrange(5)
        if(self.ligne_1 == self.ligne_2 and self.colonne_1 == self.colonne_2):
            ok = 'non'
    self.grille[self.ligne_1][self.colonne_1], self.grille[self.ligne_2][self.colonne_2] =
self.grille[self.ligne_2][self.colonne_2], self.grille[self.ligne_1][self.colonne_1]
```

Explication du script :

Il faut tout d'abord définir quelles lettres seront permutées : pour ce faire, on va "choisir" deux lettres, en les désignant par leurs coordonnées dans la grille. On crée donc quatre variables : `self.ligne_1`, `self.colonne_1`, `self.ligne_2` et `self.colonne_2`. Ces variables contiendront un nombre de 0 à 4. Ainsi, la première lettre à échanger aura pour coordonnées (`self.ligne_1` ; `self.colonne_1`), la seconde (`self.ligne_2` ; `self.colonne_2`) ; ou encore `self.grille[self.ligne_1][self.colonne_1]` pour la première, et `self.grille[self.ligne_2][self.colonne_2]` pour la seconde :

```
self.ligne_1 = randrange(5)
self.colonne_1 = randrange(5)
self.ligne_2 = randrange(5)
self.colonne_2 = randrange(5)
```

Mais on ne peut pas choisir tout à fait n'importe quelles lettres à permuter : il faut vérifier que l'on n'ait pas choisi deux fois la même lettre. Si c'est le cas, on génère à nouveaux les coordonnées des lettres jusqu'à ce que l'on obtienne deux lettres différentes l'une de l'autre :

```
if(self.ligne_1 == self.ligne_2 and self.colonne_1 == self.colonne_2):
    ok = 'non'
```

Lorsque les coordonnées choisies ne posent plus de problème, on permute les deux lettres, simplement en intervertissant leurs lignes entre-elles, et leurs colonnes entre-elles :

```
self.grille[self.ligne_1][self.colonne_1], self.grille[self.ligne_2][self.colonne_2] =
self.grille[self.ligne_2][self.colonne_2], self.grille[self.ligne_1][self.colonne_1]
```

Voilà, deux lettres ont été permutées dans la grille.

Il faut alors déchiffrer le cryptogramme avec la `grille` actuelle. Vous l'aurez deviné : on crée une nouvelle fonction, que l'on ajoute, à l'instar des autres, dans la classe `Grille`, afin de l'utiliser plus tard :

```
def decrypter_bigrammes(self, texte):
    "Décryptement des bigrammes du [texte] chiffré avec la [grille] de chiffrement"
    coordonnees = self.remplacer_coordonnees(texte, grille)
    texte_clair = ''
    l = 0
    while(l < len(texte)):
        if(coordonnees[l][0] == coordonnees[l+1][0]):       # Si sur même ligne :
            texte_clair += self.grille[coordonnees[l][0]][(coordonnees[l][1]-1)%5]
            texte_clair += self.grille[coordonnees[l+1][0]][(coordonnees[l+1][1]-1)%5]
        elif(coordonnees[l][1] == coordonnees[l+1][1]):     # Si dans même colonne :
            texte_clair += self.grille[(coordonnees[l][0]-1)%5][coordonnees[l][1]]
            texte_clair += self.grille[(coordonnees[l+1][0]-1)%5][coordonnees[l+1][1]]
        else:                                               # Sinon :
            texte_clair += self.grille[coordonnees[l][0]][coordonnees[l+1][1]]
            texte_clair += self.grille[coordonnees[l+1][0]][coordonnees[l][1]]
        l = l+2
    return texte_clair
```

Explication du script :

Pour déchiffrer un cryptogramme avec la méthode de Playfair, il faut effectuer les opérations inverses à celles que l'on utilise pour chiffrer ! Nous avons vu comment chiffrer un texte avec la grille ; voyons maintenant comment le déchiffrer en possédant la grille.

Les trois règles de chiffrement que l'on a vues précédemment (voir § 2.2) seront donc simplement utilisées à l'envers :

1. Si les deux lettres chiffrées se trouvent sur les deux coins opposés d'un rectangle, alors les lettres claires sont sur les deux autres coins : "SH" → "UN".

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 chiffrées sont sur la même ligne, alors les lettres claires sont celles qui les précèdent directement à leur gauche : "OE" → "CV".

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 chiffrées sont sur la même colonne, alors les lettres claires sont celles qui les précèdent directement en-dessus : "UV" → "GK".

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

Le code qui effectue ceci est très simple, mais il faut préalablement effectuer une étape supplémentaire, en remplaçant chaque lettre du cryptogramme par ses coordonnées dans la grille, afin de pouvoir décoder le texte plus facilement à l'aide de la grille actuelle, comme nous le verrons ci-dessous :

On crée donc une fonction, dans la classe `Grille`, qui sera appelée dans notre fonction de décryptement :

```
def remplacer_coordonnees(self, texte):
    "Remplacement de chaque lettre du message codé par ses coordonnées dans la grille : 
[[ligne_lettre0, colonne_lettre0] , [ligne_lettre1, colonne_lettre1], ...]"
    coordonnees = []
    for l in range(len(texte)):
        groupe = []
        for i in range(5):   # Recherche des "coordonnées" de la l-ième lettre du texte.
            for j in range(5):
                if(texte[l] == self.grille[i][j]):
                    groupe.append(i)
                    groupe.append(j)
        coordonnees.append(groupe)
    return coordonnees
```

Cette fonction est en fait très simple : On parcourt le texte du cryptogramme lettre par lettre, et on cherche les coordonnées de chacune des lettres dans la grille :

```
for i in range(5):
    for j in range(5):
        if(texte[l] == self.grille[i][j]):
            groupe.append(i)
            groupe.append(j)
```

Ensuite, on met ces deux coordonnées (ligne ; colonne) ensemble dans une liste `groupe` que l'on ajoute elle-même dans la liste `coordonnees` (qui est par conséquent à deux dimensions), puis on retourne la valeur de cette liste `coordonnees`.

Une fois ceci fait, le décryptement du texte est très simple : il suffit de prendre les lettres du cryptogramme deux par deux, et de chercher auquel des trois cas possibles on a affaire :

1. Soit les deux lettres sont sur la même ligne, et on soustrait 1 à la coordonnée de la colonne (on décale d'un cran vers la gauche) :

```
if(coordonnees[l][0] == coordonnees[l+1][0]):
    texte_clair += self.grille[coordonnees[l][0]][(coordonnees[l][1]-1)%5]
    texte_clair += self.grille[coordonnees[l+1][0]][(coordonnees[l+1][1]-1)%5]
```

2. Soit elles sont sur la même colonne, et on soustrait 1 à la coordonnée de la ligne (on décale d'un cran vers le haut) :

```
elif(coordonnees[l][1] == coordonnees[l+1][1]):
    texte_clair += self.grille[(coordonnees[l][0]-1)%5][coordonnees[l][1]]
    texte_clair += self.grille[(coordonnees[l+1][0]-1)%5][coordonnees[l+1][1]]
```

3. Sinon, cela signifie qu'elles forment un rectangle : la première lettre claire sera sur la même ligne que la première lettre chiffrée, et sur la même colonne que la seconde ; la seconde lettre claire sera sur la même ligne que la seconde lettre chiffrée, et sur la même colonne que la première lettre chiffrée :

```
else:
    texte_clair += self.grille[coordonnees[l][0]][coordonnees[l+1][1]]
    texte_clair += self.grille[coordonnees[l+1][0]][coordonnees[l][1]]
```

Remarque : Les calculs doivent être effectués modulo 5, car si la lettre chiffrée est dans la première ligne et/ou colonne, il faudra prendre, comme lettre claire, celle qui se situe de l'autre côté de la grille (voir § 2.2).

Il faut maintenant transformer le tableau des fréquences théoriques des bigrammes en une liste, qui nous servira pour calculer le score.

Le but étant de convertir un tableau en une liste, on reprend simplement le concept de la liste à deux dimensions que l'on a déjà utilisée pour enregistrer la grille. Prenons une partie du tableau pour mieux comprendre :

	_A	_B	_C	_D	_E	
A_	31	242	392	208	48	→ frequences[0][n]
B_	158	2	1	2	130	→ frequences[1][n]
C_	312	0	73	19	765	→ frequences[2][n]
	↓	↓	↓	↓	↓	
	frequences[n][0]	frequences[n][1]	frequences[n][2]	frequences[n][3]	frequences[n][4]	

Il suffit dès lors de procéder d'une façon similaire à celle employée pour la grille, c'est-à-dire en créant une liste principale qui contiendra le tableau en entier, dans laquelle on insérera vingt-cinq listes qui contiendront elles-mêmes vingt-cinq éléments. Chacune de ces sous-listes représente l'ensemble des bigrammes commençant par la lettre de l'alphabet qui se trouve à la même position dans l'alphabet : par exemple, la quatrième sous-liste (qui porte l'indice 3) contiendra toutes les fréquences des bigrammes commençant par la quatrième lettre de l'alphabet, donc par la lettre "D".

On aura donc la liste issue du tableau vu précédemment (voir § 3.1) :

```
frequences_bigrammes_theoriques = [[31, 242, 392, 208, 48, 135, 232, 37, 1255, 32, 7, 663,
350, 1378, 17, 412, 44, 905, 409, 613, 599, 301, 6, 69, 12] ,
[158, 2, 1, 2, 130, 1, 2, 0, 132, 4, 10, 181, 1, 1, 146, 1, 3, 187, 29, 16, 44, 3, 0, 4, 0]
,
[312, 0, 73, 19, 765, 2, 2, 411, 209, 3, 5, 124, 5, 1, 677, 11, 7, 100, 14, 142, 132, 2, 0,
11, 0] ,
[427, 1, 8, 24, 2409, 2, 5, 25, 378, 3, 0, 14, 21, 5, 231, 4, 6, 134, 64, 3, 406, 4, 0, 5,
0] ,
[616, 176, 917, 998, 782, 258, 209, 67, 179, 96, 8, 1382, 1056, 2121, 136, 699, 190, 1514,
3318, 1307, 761, 258, 125, 15, 60] ,
[181, 1, 1, 8, 180, 118, 1, 1, 190, 0, 0, 43, 1, 1, 213, 1, 2, 106, 12, 1, 61, 0, 0, 1, 0]
,
[135, 1, 10, 9, 408, 4, 63, 3, 69, 6, 4, 74, 10, 103, 47, 5, 1, 197, 12, 23, 81, 1, 0, 2,
0] ,
[267, 5, 4, 1, 285, 0, 0, 0, 149, 3, 0, 3, 4, 17, 107, 0, 3, 18, 5, 0, 42, 0, 0, 7, 0] ,
[176, 85, 203, 172, 1030, 114, 115, 6, 49, 14, 0, 798, 181, 797, 524, 75, 215, 400, 897,
1243, 11, 190, 40, 0, 4] ,
[76, 0, 0, 0, 100, 0, 0, 0, 2, 0, 0, 0, 0, 0, 91, 0, 0, 0, 0, 0, 42, 0, 0, 2, 0] ,
[8, 0, 0, 0, 6, 0, 3, 0, 6, 0, 0, 0, 10, 3, 9, 0, 0, 5, 1, 0, 0, 0, 0, 3, 0] ,
[1270, 14, 22, 58, 2366, 25, 14, 39, 512, 4, 1, 647, 18, 41, 281, 69, 47, 16, 126, 42, 369,
14, 0, 15, 1] ,
[510, 152, 11, 11, 1099, 0, 1, 1, 302, 0, 0, 7, 243, 4, 334, 201, 2, 10, 10, 8, 52, 1, 0,
3, 0] ,
[405, 30, 438, 785, 985, 124, 222, 24, 316, 17, 7, 89, 68, 249, 303, 130, 82, 55, 846,
1694, 114, 109, 1, 19, 20] ,
[6, 83, 88, 101, 46, 32, 115, 7, 452, 14, 3, 184, 391, 1646, 8, 175, 19, 491, 126, 109,
1086, 28, 4, 62, 4] ,
[671, 1, 3, 21, 441, 5, 1, 136, 119, 0, 0, 377, 2, 4, 505, 125, 1, 363, 31, 65, 140, 1, 0,
1, 0] ,
[2, 0, 3, 0, 1, 0, 0, 1, 0, 0, 0, 1, 3, 0, 0, 1, 0, 1, 0, 0, 975, 0, 0, 0, 0] ,
[896, 53, 168, 302, 1885, 46, 96, 5, 583, 11, 3, 292, 181, 88, 520, 82, 51, 176, 386, 445,
183, 77, 1, 21, 5] ,
[809, 85, 306, 735, 1377, 151, 73, 83, 565, 36, 0, 453, 192, 107, 521, 496, 191, 137, 702,
578, 343, 92, 6, 30, 10] ,
[881, 25, 166, 515, 1484, 52, 19, 64, 984, 28, 3, 331, 70, 40, 363, 268, 96, 668, 404, 269,
270, 41, 6, 18, 3] ,
[168, 87, 165, 162, 781, 40, 83, 4, 534, 41, 3, 302, 128, 516, 19, 184, 15, 980, 591, 469,
14, 177, 264, 8, 4] ,
[277, 0, 1, 0, 502, 0, 0, 0, 288, 0, 0, 1, 0, 0, 167, 0, 0, 81, 0, 0, 11, 0, 0, 0, 0] ,
[35, 14, 37, 36, 68, 8, 7, 5, 57, 0, 0, 21, 15, 3, 7, 56, 11, 3, 15, 35, 2, 18, 4, 0, 0] ,
[63, 0, 7, 7, 59, 3, 4, 0, 0, 0, 0, 13, 8, 5, 15, 14, 0, 10, 75, 9, 2, 4, 0, 0, 0] ,
[8, 0, 2, 6, 49, 3, 1, 0, 1, 1, 0, 11, 4, 2, 15, 4, 1, 0, 3, 1, 0, 7, 0, 0, 2]]
```

Comme les nombres de cette liste représentent le nombre de fois qu'apparaît chaque bigramme dans un texte de 99'964 lettres (100'000 lettres auxquelles on a retiré les "W"), il faut convertir chaque nombre en un pourcentage, grâce à une fonction créée spécialement pour cela (cette fonction est l'une des fonctions satellites qui n'appartiennent à aucune classe) :

```
def pourcentages(liste, nb_lettres):
    "Transforme les nombres de la [liste à deux dimensions] en pourcentages grâce au
[nombre de lettres]."
    for i in range(len(liste)):
        for j in range(len(liste[i])):
            liste[i][j] = float(liste[i][j])/nb_lettres
    return liste
```

Explication du script :

Cette fonction transforme simplement chaque nombre en un pourcentage en le divisant par le nombre total de lettres (que l'on fournit en argument). Donc dans le cas présent par 99'964.

```
liste[i][j] = float(liste[i][j])/nb_lettres
```

Maintenant que le programme peut générer une grille, permuter deux lettres au hasard, et décrypter le texte à l'aide de la nouvelle grille, il faut à présent s'occuper d'assigner un score à la grille. Pour y parvenir, il faut tout d'abord mesurer les fréquences des bigrammes de lettres apparaissant dans le cryptogramme (c'est une deuxième fonction satellite) :

```
def mesurer_frequences_bigrammes(texte):
    "Mesure les fréquences des bigrammes d'un [texte] quelconque."
    alphabet = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'X', 'Y', 'Z']
    frequences = []             # Création de la liste (vide) des fréquences des bigrammes.
    for i in range(25):
        tt = []
        for j in range(25):
            tt.append(0)
        frequences.append(tt)
    l = 0
    while(l < len(texte)-1):    # On va jusqu'à l'avant-dernier et non jusqu'au dernier.
        frequences[alphabet.index(texte[l])][alphabet.index(texte[l+1])] += 1
        l = l+1
    frequences = pourcentages(frequences, len(texte)-1)
    return frequences
```

Explication du script :

Le résultat que l'on veut obtenir est simple : on veut avoir une liste des *fréquences mesurées* des bigrammes, construite de la même manière que la liste des fréquences théoriques :

Prenons par exemple le texte : "Ce texte comporte des bigrammes" ; on veut mesurer à la main les fréquences des bigrammes apparaissant. Pour ce faire, on commence par le début, et on compte ! On a donc d'abord une fois le bigramme "CE", puis le bigramme "ET", puis "TE, "EX", "XT", "TE" : c'est le second bigramme "TE" que l'on rencontre, on ajoute donc évidemment 1 au nombre de bigrammes "TE". On continue ensuite de la même façon jusqu'à la fin : "EC", "CO", "OM", …, "ME", "ES" :

Sur le schéma, on compte les bigrammes en alternant les couleurs : d'abord orange, ensuite rouge, puis bleu, vert, et on recommence à l'orange, rouge, et ainsi de suite jusqu'à la fin du texte.

Le programme agira comme nous avons procédé dans cet exemple : on crée tout d'abord une liste vide, `frequences` (ayant la même structure que la liste contenant les fréquences théoriques des bigrammes, vue précédemment), qui contiendra le nombre de fois qu'un certain bigramme est présent ; tous les nombres vaudront initialement zéro :

```
frequences = []
for i in range(25):
    tt = []
    for j in range(25):
        tt.append(0)
    frequences.append(tt)
```

La fonction prendra ensuite les lettres deux à deux (attention, le mot "CHAT" ne sera pas décomposé en "CH" et "AT", mais bien en "CH", "HA" et "AT" !), puis incrémentera la valeur représentant le nombre de fois qu'elle a rencontré ce bigramme dans la liste `frequences` (qui est, rappelons le pour la dernière fois, construite de la même manière que celle des fréquences théoriques qui a été expliquée précédemment). Il faut faire attention à bien s'arrêter à l'avant-dernière lettre, car la dernière lettre formera un seul bigramme avec l'avant-dernière, contrairement à une lettre du milieu du texte qui en forme deux : un avec la lettre qui la précède et un avec la lettre qui la suit :

```
l = 0
while(l < len(texte)-1):
    frequences[alphabet.index(texte[l])][alphabet.index(texte[l+1])] += 1
    l = l+1
```

La dernière étape consiste à convertir les nombres de bigrammes en pourcentages, pour avoir réellement des fréquences, grâce à la fonction que nous avons vue précédemment :

```
frequences = pourcentages(frequences, len(texte)-1)
```

On retourne enfin la valeur de la liste des fréquences des bigrammes.

Nous pouvons maintenant concevoir une fonction qui calculera le score de la grille (c'est notre troisième fonction satellite) :

```
def calculer_score(frequences_bigrammes_theoriques, frequences_bigrammes_mesurees):
    "Calcule le score actuel de la grille."
    score = 0
    for i in range(25):
        for j in range(25):
            score += abs(frequences_bigrammes_theoriques[i][j]-
frequences_bigrammes_mesurees[i][j])
    return score
```

Explication du script :

Cette fonction est très simple dans le sens où elle n'est rien d'autre que l'application de la formule vue précédemment (voir §3.1) permettant de calculer le score d'une grille, et qui est, pour mémoire :

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

Comme nous avons déjà deux listes comportant pour l'une les fréquences théoriques des bigrammes, et pour l'autre les fréquences mesurées des bigrammes, il suffit de prendre en parallèle un à un les éléments des listes, de calculer leur différence (en valeur absolue, puisque nous voulons l'écart entre les fréquences) et de l'additionner au `score` :

```
for i in range(25):
    for j in range(25):
        score += abs(frequences_bigrammes_theoriques[i][j]-
frequences_bigrammes_mesurees[i][j])
```

On retourne finalement la valeur du `score`.

Une des dernières fonctions de ce programme est celle qui s'occupera de rétablir la grille précédente dans le cas où le score obtenu est supérieur au score minimal :

```
def retablir(grille):
    "Rétablit la [grille] en annulant la dernière permutation"
    self.grille[self.ligne_1][self.colonne_1], self.grille[self.ligne_2][self.colonne_2] =
self.grille[self.ligne_2][ self.colonne_2], self.grille[self.ligne_1][self.colonne_1]
```

Explication du script :

Cette fonction est on ne peut plus simple : comme les variables `ligne_1`, `colonne_1`, `ligne_2` et `colonne_2` sont des attributs de la classe `Grille`, le programme a encore en mémoire les lettres qui ont été permutées. Il suffit donc de les permuter à nouveau, et elles retrouveront leur place originelle :

```
self.grille[self.ligne_1][self.colonne_1], self.grille[self.ligne_2][self.colonne_2] =
self.grille[self.ligne_2][self.colonne_2], self.grille[self.ligne_1][self.colonne_1]
```

Enfin, nous aurons ultérieurement besoin d'une fonction capable de copier la grille actuelle (afin notamment de la sauvegarder si elle est la meilleure grille obtenue) :

```
def copier_grille(grille):
    "Renvoie une copie de la [grille]."
    copie = Grille()
    for i in range(0, 5):
        for j in range(0, 5):
            copie.grille[i][j] = grille.grille[i][j]
    return copie
```

Explication du script :

Il suffit de créer un nouvel objet `Grille` et de modifier son attribut `grille` en lui donnant le contenu de celui de la grille que l'on souhaite copier.

Maintenant que l'on a défini toutes les classes, on peut facilement écrire une dernière fonction satellite contenant le corps principal du programme, qui appelle les fonctions créées précédemment les unes après les autres :

Remarque préliminaire : Dans ce script et les suivants, les lignes de code ayant attrait à l'affichage graphique ont été supprimées, ce pour augmenter la lisibilité, car nous avons dit que nous ne nous intéressions pas à cet aspect du programme.

```
def recherche():
    "Programme principal de la recherche."

    #####--------------------<  Variables initiales  >--------------------#####

    frequences_bigrammes_theoriques = [VOIR PRÉCÉDEMMENT]
    frequences_bigrammes_theoriques = pourcentages(frequences_bigrammes_theoriques, 99963)

    global grille, message

    score_but = 0.4
    k_max = 15000

    grille = Grille()
    grille_best = copier_grille(grille)

    score = calculer_score(frequences_bigrammes_theoriques,
mesurer_frequences_bigrammes(grille.decrypter_bigrammes(message)))
    score_best = score

    k = 0
```

Explication du script :

Cette première partie de la fonction finale ne fait que créer et affecter des valeurs aux variables que l'on va utiliser par la suite. Voyons brièvement quelles sont ces variables :

- `frequences_bigrammes_theoriques` : Nous en avons parlé précédemment, et le nom est explicite. À noter tout de même que l'on transforme la liste vue en pourcentages.
- `message` : C'est simplement le cryptogramme que l'on cherche à casser.
- `grille` : On crée un objet grille (qui se chargera de construire la grille aléatoire).
- `grille_best` : Variable permettant de stocker la meilleure grille trouvée.
- `score` : On calcule le score de la `grille`.
- `score_best` : C'est le meilleur score (donc le plus bas) actuel.
- `score_but` : C'est le score qu'il faut atteindre, donc le score au-dessous duquel il faudra tenter de faire descendre notre `score`.
- `k` : C'est simplement un compteur. Il s'incrémentera à chaque permutation.
- `k_max` : C'est le nombre maximum de permutations que l'on autorise le programme à faire, au cas où il se bloquerait.

Nous avons enfin la deuxième partie de la fonction, qui est en fait le programme principal.

```
    #####-------------------<  Programme principal  >--------------------#####

    ## On tente de trouver la bonne grille par essais successifs :
    while(score > score_max and k < k_max):
        # On permute deux lettres au hasard dans la grille.
        grille.permuter()
        # On décrypte le message initial avec la grille actuelle.
        texte = grille.decrypter_bigrammes(message)
        # On calcule les fréquences des bigrammes dans le nouveau texte.
        frequences_bigrammes_mesurees = mesurer_frequences_bigrammes(texte)
        # On calcule le score actuel.
        score = calculer_score(frequences_bigrammes_theoriques,
frequences_bigrammes_mesurees)
        if(score > score_best):
            grille.retablir()
        else:
            score_best = score
            grille_best = copier_grille(grille)
        k += 1
```

Explication du script :

C'est simplement la programmation du pseudo-code vu lors de la recherche de l'idée générale (voir § 3.1). Réexpliquer en détail ce morceau de code serait donc superflu.

On notera l'utilisation de la fonction `copier_grille`, celle-ci permettant de… copier une grille dans une nouvelle variable (car effectuer une affectation entre deux variables dont l'une contient une liste ne crée qu'une nouvelle référence vers cette même liste en Python). On copie donc chaque élément successivement dans une nouvelle grille que l'on retourne à la fin :

```
def copier_grille(grille):
    "Renvoie une copie de la [grille]."
    copie = Grille()
    for i in range(0, 5):
        for j in range(0, 5):
            copie.grille[i][j] = grille.grille[i][j]
    return copie
```

Notez tout de même que la recherche est très simple : on permute deux lettres, puis on rétablit la grille précédente si le score de la grille obtenu est moins bon (donc plus élevé), ou on la garde s'il est meilleur (donc plus bas), et ce jusqu'à ce que le score obtenu soit satisfaisant, ou que l'on ait fait un nombre jugé maximal d'essais.

On peut maintenant tester le programme en entrant un (long) cryptogramme, et voir si l'on parvient à obtenir un résultat.

3.3 Premiers résultats

On constate dans un premier temps que le script est correct, puisque le programme effectue la recherche comme nous l'avions prévu, sans afficher aucune erreur.

Par contre, on constate également qu'il y a un problème, car la recherche se bloque, ou plutôt se met à "tourner en rond" au bout d'un certain temps, ne parvenant plus à trouver de meilleure solution, alors que la grille obtenue n'est manifestement pas la bonne.

Il va donc falloir modifier une partie du programme pour éviter qu'il ne se bloque ; mais voyons d'abord quelle en est la cause.

3.4 Problèmes rencontrés et recherche de solutions

Le problème rencontré vient du fait que le programme se bloque au bout d'un certain temps de recherche ; en effet, il tombe dans des optima locaux et ne parvient pas à en sortir.

Une solution possible serait de relancer plusieurs fois la recherche, jusqu'à ce qu'un résultat soit obtenu. Cela fonctionnerait certainement, mais le temps de recherche, déjà suffisamment long, deviendrait démesuré, car la probabilité de tomber dans un minimum local est grande, et il faudrait relancer la recherche un grand nombre de fois, ce qui donne un temps de recherche démultiplié ; il faut essayer d'éviter cela, bien que le temps de recherche ne soit pas la priorité.

La modification que l'on va apporter au programme va permettre d'éviter ces optima locaux. En effet, jusqu'à présent, nous avons appliqué une méthode de *descente de plus grande pente* ; nous n'acceptions une solution que si elle était meilleure que toutes celles trouvées précédemment, ce qui nous enfermait dans des minima locaux. Pour remédier à ce problème, il suffirait donc d'accepter certaines solutions qui font augmenter le score de la grille, et qui par conséquent dégradent la solution, au lieu de toutes les annuler. Pour ce faire, il existe un algorithme efficace (bien qu'il ne donne pas dans tous les cas la solution) : c'est le *recuit simulé*.

Recuit simulé *(Simulated Annealing)* :

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.

En informatique, partant d'une solution donnée, on en obtient une seconde en la modifiant. Soit celle-ci améliore le critère que l'on cherche à optimiser, on dit alors qu'on a fait baisser l'énergie du système, soit celle-ci le dégrade. Si on accepte une solution améliorant le critère, on tend ainsi à chercher *l'optimum dans le voisinage de la solution de départ*, et c'est justement ce que l'on cherche à éviter. Or, l'acceptation d'une *mauvaise* solution permet d'explorer une plus grande partie de l'espace des solutions et tend à éviter de s'enfermer trop vite dans la recherche d'un optimum local.

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.

Etat 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 constante.

Pseudo-code

On peut maintenant construire le pseudo-code de cet algorithme, qui est le suivant :

```
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 : température 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
```

3.5 Amélioration du programme

Nous allons à présent appliquer les modifications décrites ci-dessus à notre programme.

Il n'y a en fait que la dernière fonction que l'on a vue, la fonction `recherche`, qu'il faut refondre pour intégrer le recuit simulé ; tout le reste du programme fonctionnera de la même manière.

Tout d'abord, les variables initiales existantes restent les mêmes, et seules quelques-unes sont ajoutées :

```
def recherche():
    "Programme principal de la recherche."


    #####-------------------<  Variables initiales  >--------------------#####

    frequences_bigrammes_theoriques = [VOIR PRÉCÉDEMMENT]
    frequences_bigrammes_theoriques = pourcentages(frequences_bigrammes_theoriques, 99964)

    global grille, message, continuer

    score_but = 0.3                     # Score à atteindre (sous lequel il faut passer).
    k_max = 12000                       # Nombre maximal d'essais.

    t0 = 10                             # Température initiale.
    coef = 0.95                         # Coefficient de décroissance.
    palier = 100                        # Palier de température.

    grille = Grille()
    grille_best = copier_grille(grille)

    # Score au départ (énergie)
    score1 = calculer_score(frequences_bigrammes_theoriques,
mesurer_frequences_bigrammes(grille.decrypter_bigrammes(message)))
    score2 = score1
    score_best = score2

    k = 0
    t = t0
```

Explication du script :

Décrivons rapidement les nouvelles variables :

- `t0` : C'est la température initiale que l'on va utiliser.
- `coef` : Ceci est le coefficient de décroissance de la température, qui doit être inférieur à 1 ; la température subira une décroissance de la forme :
  $$t_{i+1} = t_i \cdot coef \text{ ou } t_n = t_0 \cdot coef^n.$$
- `palier` : La température décroît après un certain nombre de permutations ; ce nombre est le palier de décroissance.
- `score1` ; `score2` : On doit cette fois garder le score précédent en mémoire pour pouvoir calculer la différence d'énergie ΔE.
- `t` : C'est la température actuelle.

Le programme principal, lui, change fondamentalement :

```
    #####-------------------<  Programme principal  >--------------------#####

    ## Etape 1
    while(score2 > score_but and k < k_max):
        # On permute deux lettres au hasard dans la grille.
        grille.permuter()
        # On décrypte le message initial avec la grille actuelle.
        texte = grille.decrypter_bigrammes(message)
        # On calcule les fréquences des bigrammes dans le nouveau texte.
        frequences_bigrammes_mesurees = mesurer_frequences_bigrammes(texte)
        # On calcule le score actuel (énergie).
        score2 = calculer_score(frequences_bigrammes_theoriques,
frequences_bigrammes_mesurees)
        deltaE = score2-score1
        if(deltaE <= 0):
            # Si le score actuel est plus faible, on garde la grille.
            if(score2 < score_best):
                # On actualise le score minimal.
                score_best = score2
                # On mémorise la meilleure grille actuelle.
                grille_best = copier_grille(grille)
            score1 = score2
        elif(random() > exp(-deltaE/t) or score2 > score_best*1.04):
            # On reprend l'ancienne solution.
            grille.retablir()
        k += 1
        if(k%palier == 0):
            t = t*coef                    # On diminue la température.
```

Explication du script :

Rappelons encore une fois que tout ce qui concerne la partie graphique du programme a été supprimée du code ci-dessus.

Cette partie du code est la pure retranscription du pseudo-code du recuit simulé vu plus haut, adaptée au cas de notre recherche. Il faut toutefois noter la modification suivante :

```
        elif(random() > exp(-deltaE/t) or score2 > score_best*1.04):
            # On reprend l'ancienne solution.
            grille.retablir()
```

Dans le cas de notre problème, chaque permutation de deux lettres peut avoir des conséquences drastiques sur le score de la grille. Le recuit simulé utilisé seul n'est donc pas la solution la plus efficace pour converger vers l'optimum global, car une seule mauvaise modification acceptée peut démanteler toute la convergence établie. Une solution consiste donc à renforcer le recuit simulé, et ce en lui fixant simplement une limite : les solutions supérieures à un certain *seuil limite d'acceptation* (fixé par rapport au score minimum obtenu jusque là) seront systématiquement refusées. C'est ce à quoi correspond la seconde partie de la condition ci-dessus. En outre, le seuil est fixé proportionnellement au score minimum, donc plus le score est bas, plus le seuil le sera également, permettant ainsi de réduire l'espace des solutions exploré et donc le risque de régression inutile en fonction de la convergence vers l'optimum global.

Il faut maintenant compter une seconde 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 finale : 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. Par exemple, les deux grilles suivantes sont semblables au niveau de leur qualité à crypter et décrypter un message, mais sont vues comme différentes par le programme :

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

=

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

=

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

En revanche, ces deux grilles sont différentes, et ne fourniront pas le même message lors de leur utilisation :

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

≠

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

Il faut donc tester toutes les permutations de la grille qui ne sont pas un simple décalage de celle-ci dans un sens ou un autre. Il faut par ailleurs prêter attention à ne pas tester deux permutations qui sont elles-mêmes circulairement identique. Il faut en outre. En fait, il faut essayer toutes les permutations circulairement différentes de la grille elle-même et toutes circulairement différentes entre elles.

Il faut tout de même remarquer qu'une rotation de la grille à 90° modifie celle-ci, car la règle qui veut que l'on prenne les deux coins opposés du rectangle sur lequel se trouvent les deux lettres

ne s'appliquera pas de la même manière, car on prend les lettres cryptées au même niveau *horizontalement* que les lettres claires. Par exemple :

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

En effet, le bigramme "GR" sera codé ici "HQ" dans la première grille, et "QH" dans la seconde.

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 24 permutations différentes :

01234	02134	03124	04123
01243	02143	03142	04132
01324	02314	03214	04213
01342	02341	03241	04231
01423	02413	03412	04312
01432	02431	03421	04321

C'est en fait les permutations des objets 1, 2, 3 et 4, car le 0 est fixé afin de supprimer les permutations circulaires :

$$P_4 = 4! = 24.$$

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 120^2 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 !

Le code de cette seconde étape est donné ci-dessous :

```
    ## Etape 2
    permutations = ['01234', '01243', '01324', '01342', '01423', '01432', '02134', '02143',
'02314', '02341', '02413', '02431', '03124', '03142', '03214', '03241', '03412', '03421',
'04123', '04132', '04213', '04231', '04312', '04321']
    score_min = 10000
    grille = copier_grille(grille_best)
    for i in range(len(permutations)):
        grille_t = copier_grille(grille)
        for n in range(5):
            grille_t.grille[n] = grille.grille[int(permutations[i][n])]
        for j in range(len(permutations)):
            grille_tt = copier_grille(grille_t)
            for k in range(5):
                for n in range(5):
                    grille_tt.grille[k][n] = grille_t.grille[k][int(permutations[j][n])]
            score = calculer_score(frequences_bigrammes_theoriques,
mesurer_frequences_bigrammes(grille_tt.decrypter_bigrammes(message)))
            if(score < score_min):
                score_min = score
                grille_min = copier_grille(grille_tt)
```

Explication du script :

On commence par définir les 24 permutations vues plus haut, et on procède exactement comme dit ci-dessus, c'est-à-dire qu'on essaie les 24 permutations des lignes pour chacune des 24 permutations des colonnes. On copie les grilles intermédiaires avec la fonction décrite plus haut, et on garde la grille "permutée" si son score est inférieur au score de la meilleure grille trouvée jusqu'alors.

3.6 Quelques mots sur la partie graphique

L'explication du programme a été concentrée uniquement sur la partie algorithmique du problème. La partie graphique, et notamment l'affichage, a été omise volontairement, et ce pour deux raisons principales : d'une part, ce dossier aurait pris une ampleur démesurée, car la partie constituant l'affichage est au moins aussi longue à expliquer que celle traitée ici. D'autre part, son explication est moins intéressante, car elle consisterait principalement en une explication syntaxique du module `Tkinter` utilisé, contrairement à la partie algorithmique, qui a fait intervenir plusieurs notions intéressantes, autant du point de vue mathématique que logique et informatique. On peut néanmoins décrire ici brièvement le fonctionnement graphique du programme, sans expliquer la syntaxe pythonienne utilisée lors de la programmation.

Le programme principal est constitué de trois fenêtres qui s'ouvrent en boucle l'une après l'autre tant que l'exécution n'est pas interrompue. Si l'on choisit l'option *"Recommencer"* en fin de recherche, la boucle recommence, et rien n'est donc réalisé, sauf la fermeture des fenêtres actuellement ouvertes (c'est la fonction `recommencer` qui s'en charge), alors que si l'on choisit l'option *"Quitter"* à n'importe quel moment, la fonction `quitter` affecte la valeur nulle à la variable globale `stop`. Or, celle-ci étant la condition à la répétition de la boucle (`stop` doit valoir 1), cette dernière s'interrompra. On notera que, lors de la fermeture d'une fenêtre par la croix en

haut à droite, l'action est interceptée, et la fonction `quitter` est appelée, permettant une fermeture *propre* du programme.

Les trois fenêtres s'ouvrent dans l'ordre, et donc, dès qu'une fenêtre est fermée, la suivante s'ouvre automatiquement :

1. Première fenêtre : Écran-titre.
2. Deuxième fenêtre : Sélection du cryptogramme.
3. Troisième fenêtre : Décryptement (fenêtre de la recherche principale).

La première fenêtre est simplement un écran-titre, et le bouton *"Commencer"* provoque la destruction de celle-ci, ouvrant par la même occasion la deuxième fenêtre.

Dans cette deuxième fenêtre, il est demandé d'insérer le message que l'on souhaite décrypter. On peut donc insérer un message soi-même, ou en générer un aléatoirement. Les deux boutons de génération de cryptogramme (court ou long) appellent tous deux la même fonction, `generer_texte`, en lui fournissant une valeur différente pour l'argument longueur, permettant de différencier les cryptogrammes courts et longs. On choisit ensuite aléatoirement un texte parmi ceux disponibles, que l'on affichera dans la boîte de dialogue. Lors de la validation, une autre fonction, `valider_texte`, est appelée afin de récupérer le texte et de tester sa validité (le texte est valide s'il ne comporte que des lettres sans le "W", si il comporte un nombre pair de lettres, et si il a au moins une longueur de 10 caractères, longueur fixée arbitrairement). Dans le cas où le texte n'est pas valide, un message d'erreur est affiché, sinon, la deuxième fenêtre est fermée, provocant ainsi l'ouverture de la troisième fenêtre.

C'est dans la troisième fenêtre que la recherche, la partie fondamentale de l'application, s'effectue. Différents boutons sont utilisables, effectuant les actions décrites dans le *mode d'emploi* du programme (Annexe 2) ; on notera que c'est le bouton *"Commencer"* qui appelle la fonction satellite `recherche`, qui n'est rien d'autre que la recherche elle-même, comme nous l'avons vu ci-dessus. Le bouton *"Afficher"* appelle la fonction du même nom, celle-ci provocant l'ouverture d'une fenêtre dans laquelle se trouve la grille actuelle, et le message décrypté avec celle-ci ; cette fonction affichera différents boutons d'action en fonction de l'étape à laquelle elle est appelée (l'étape est fournie en argument), et son ouverture ne trouble pas la recherche, car aucune boucle principale (`mainloop`) n'est créée pour l'affichage (sauf dans le cas où l'on affiche le résultat obtenu avant de passer à l'étape 2). Enfin, le graphique présent sur cette fenêtre est commenté dans le mode d'emploi. Finalement, on affiche le résultat et les boutons *"Recommencer"* et *"Quitter"* lorsque la recherche est terminée, puis une nouvelle itération de la boucle est réalisée si l'on souhaite recommencer.

3.7 Résultats finaux et conclusion

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 le hasard fait qu'un résultat différent sera obtenu à chaque nouvelle tentative. Il faut donc répéter le même essai plusieurs fois pour obtenir des moyennes.

Voici ci-dessous les résultats de la première batterie de tests. La grille utilisée pour chiffrer le texte était simplement la suivante :

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

Voici les grilles obtenues à la fin de chaque tentative, ainsi que les temps moyens de recherche et le nombre d'essais requis pour obtenir la bonne grille dans le cas où celle-ci a été trouvée :

Échec partiel

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

3'000 essais ; 7 min

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

4'750 essais ; 11 min

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

12'000 essais ; 29 min

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

Échec partiel

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

Échec total

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

Échec total

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

8'000 essais ; 20 min

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

3'250 essais ; 7 min

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

Échec total

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

2'250 essais ; 5 min

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

Échec partiel

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

Les résultats de la seconde batterie de tests sont fournis en annexe (Annexe 3), mais les conclusions sont similaires, comme on le constate ci-dessous.

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

Taux de réussite :

- 12 essais : 6/12 = **50 %**.
- 40 essais : 19/40 = **47.5 %**.

Temps moyen :

- 12 essais : Sans échecs : **13.2 min** en moyenne.
  Avec échecs : **21.6 min** en moyenne.
- 40 essais : Sans échecs : **12.1 min** en moyenne.
  Avec échecs : **21.5 min** en moyenne.

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. 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 ou 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 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.

De nombreuses modifications des paramètres fournis à l'algorithme ont été effectuées, et la meilleure configuration obtenue est celle citée dans le script ci-dessus. 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).
- Coefficient 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 vite, et mieux vaut relancer la recherche en cas d'échec plutôt que de laisser chercher plusieurs heures.

La solution adoptée, qui est par ailleurs la dernière modification que l'on effectuera dans notre programme, 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 ; la variable `der` sert ainsi à compter le nombre de permutations effectuées depuis que la dernière meilleure grille a été trouvée.

Ensuite, l'affichage graphique de la recherche ralentit indubitablement la recherche : 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. Mais le but de ce travail n'est pas de fournir un outil performant et "efficace" de décryptement ; son but était, pour mémoire, de décrypter le chiffre de Playfair, en se basant sur une méthode simple utilisant le hasard. Les graphismes ont donc été laissés, pour fournir une vision globale et schématique de l'avancement de la recherche afin de mieux la cerner.

4. PROLONGEMENT DU PROBLÈME

4.1 Longueur du texte

Il faut savoir que plus le texte fourni est long, plus la recherche a de chances d'être fructueuse. En effet, plus il y a de bigrammes, plus le score sera précis. Imaginez un cryptogramme comportant deux lettres ; il est dans ce cas simplement impossible de retrouver le message original.

Une question intéressante est donc de savoir à partir de quel nombre de lettres un texte a de grandes chances d'être décrypté avec le programme créé, et, à l'inverse, combien de lettres au minimum faut-il pour obtenir un résultat, même avec un temps de recherche très élevé. Par ailleurs, bien que cela sorte du cadre de ce travail, il serait intéressant de calculer le nombre théorique minimum de lettres que le texte devrait comporter.

On peut obtenir des réponses approximatives à ces questions en essayant d'entrer différentes longueurs de textes dans le programme et de les décrypter. C'est d'ailleurs pour cette raison que vous avez la possibilité de générer un cryptogramme long ou court lors du choix du texte : bien que plus longue, la recherche fournit un résultat plus souvent lorsque le cryptogramme est long que lorsqu'il est court (il est d'ailleurs assez rare qu'un résultat soit obtenu pour un message court, la recherche se bloquant après un certain nombre de permutations) ; on constate donc déjà une différence à cette échelle.

4.2 Autres méthodes de décryptement

Plusieurs autres méthodes auraient été envisageables afin de décrypter ce code. Le choix a été porté sur la moins compliquées d'entre-elles, conformément à l'objectif du travail. Cependant, il est possible d'en décrire ici brièvement quelques autres.

Une première approche différente aurait été de ne pas se concentrer uniquement sur les bigrammes de lettres, mais également sur les lettres elles-mêmes, voire sur les trigrammes, ou encore sur un groupe plus grand de lettres ; ce afin d'augmenter la précision du score.

Par ailleurs, il aurait été possible d'exclure le hasard, ou du moins réduire considérablement son impact sur les résultats, par exemple en choisissant les lettres que l'on permute dans la grille en fonction de leur mauvaise aptitude à engendrer le bon texte clair, en œuvrant ainsi en une sorte de *sélection naturelle* des lettres à permuter, les moins adaptées ayant plus de chance de l'être que les autres.

Ensuite, nous aurions pu, au lieu de croiser deux lettres à la fois dans la grille, en permuter trois, quatre, cinq, voire plus. Il aurait également été possible de faire diminuer le nombre de lettres permutées, avec le temps ou la progression par exemple.

Enfin, le recuit simulé aurait pu être remplacé par un grand nombre d'autres métaheuristiques connues permettant également d'optimiser la solution, comme par exemple la recherche par tabous, ou alors un algorithme génétique.

5. CONCLUSION GÉNÉRALE

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 au programme pour l'empêcher de s'enfermer trop vite dans des minima 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.

Ce travail a été très intéressant à réaliser, et il m'a permis de programmer un algorithme, puis de l'améliorer en tenant compte des problèmes rencontrés (en y incorporant le principe de recuit simulé). Il m'a par ailleurs fait découvrir différentes notions intéressantes en rapport avec le langage de programmation Python. En conclusion, le résultat obtenu n'est pas parfait dans l'absolu, mais correspond à l'objectif fixé au commencement.

6. RÉFÉRENCES ET BIBLIOGRAPHIE

Livres :

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

Sites internet :

- 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, *Wkipédia : 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
- http://www.siteduzero.com/tutoriel-3-14171-creer-une-installation.html, *Site du zéro : Créer une installation*, 30 janvier 201

Sources des images (liens de pages internet) :

- http://foto.hut.fi/opetus/260/luennot/historia/wheatstone.gif, 23 août 2009
- http://upload.wikimedia.org/wikipedia/commons/9/96/Lyon_Playfair.jpg, 23 août 2009
- http://thepeerage.com/240637_001.jpg, 7 janvier 2010

Programmes utilisés :

- Adobe Photoshop CS4 (réalisation des images et icônes).
- Inno Setup (programmation de la création de l'exécutable de l'installation du programme).
- Microsoft Office Word 2007 (rédaction de ce dossier et de l'annexe 2).
- Notepad++ (création de fichiers divers, notamment *"autorun.inf"*).
- Py2exe (conversion du fichier Python .py en exécutable .exe).
- Python 2.6 et IDLE (programmation du programme principal de ce travail).

7. ANNEXES

- **CD-ROM** (Annexe 1).
- **Mode d'emploi** du programme (Annexe 2).
- **Batteries de tests** (Annexe 3).

8. DÉCLARATION

Je déclare par la présente que j'ai réalisé ce travail de manière autonome et que je n'ai utilisé aucun autre moyen que ceux indiqués dans le texte. Tous les passages inspirés ou cités d'autres auteur-es sont dûment mentionnés comme tels. Je suis conscient que de fausses déclarations peuvent conduire le Lycée cantonal à déclarer le travail non recevable et m'exclure de ce fait à la session d'examens à laquelle je suis inscrit.

Ce travail reflète mes opinions et n'engage que moi-même, non pas le professeur responsable de mon travail ou l'expert qui m'a accompagné dans cette recherche.

Lieu et date :..... Signature :.....

Mode d'emploi (Annexe 2)

TABLE DES MATIÈRES

1. CONTENU DU CD-ROM

Sur le CD-Rom fourni avec ce dossier (Annexe 1), vous trouverez quatre dossiers :

- Le premier de ces dossiers, intitulé *"Playfair"*, comporte deux sous-dossiers dont le premier, *"Programme"* contient les scripts bruts, au format *.py* (Python), ainsi que les quelques images utilisées par l'application, et le second, *"Partie théorique"* contient ce présent dossier (aux formats *.pdf* et *.docx*), les annexes, et les images qu'ils contiennent.
- Le deuxième dossier, *"Playfair executable"*, comporte dans le sous-dossier *"Executable"* le script compilé grâce au programme Py2exe. Il doit donc marcher sur n'importe quel ordinateur, même non-équipé de Python. Dans le sous-dossier *"Py2exe"*, vous trouverez le programme d'installation de Py2exe, *"py2exe-0.6.9.win32-py2.6.exe"* pour la version 2.6 de Python (en cas de besoin, vous trouverez les programmes d'installation de Py2exe pour les autres versions de Python en suivant ce lien : http://sourceforge.net/projects/py2exe/files/), ainsi que le fichier *"setup.py"* ayant servi à la compilation (consulter la documentation officielle pour plus d'informations).
- Le troisième dossier, *"Playfair installation"*, contient un programme d'installation de l'application, *"playfair_setup.exe"*, ainsi que le script qui a permis de le créer, *"playfair.iss"*. Ce script a été écrit avec le programme *"Inno Setup"*, disponible gratuitement et dont le fichier d'installation est disponible dans ce même dossier sous le nom de *"isetup-5.3.7"*.
- Le quatrième et dernier dossier, *"Python"*, contient le fichier d'installation de Python ainsi que de l'IDLE, *"python-2.6.4.msi"*.
- Pour finir, le fichier *"autorun.inf"* situé à la racine du CD-Rom sert à l'exécution automatique de celui-ci (il lance l'exécution du programme d'installation après l'insertion du CD-Rom).

Pour lire le script contenu dans le premier dossier, il faut préalablement avoir installé Python 2.6, fourni dans le dossier *"Python"* (voir premier dossier ci-dessus), le programme d'installation ayant

pour nom, comme vu ci-dessus, *"python-2.6.4.msi"*, ou alors le télécharger gratuitement à l'adresse suivante : http://www.python.org/ftp/python/2.6.4/python-2.6.4.msi.

Remarque importante : Il faut impérativement que vous possédiez le système d'exploitation Windows (plus répandu que les autres systèmes d'exploitation tels Linux ou Mac OS), et la version 2 de Python (Python 2.*n*), et non Python 3, celui-ci ne pouvant faire fonctionner la plupart des applications écrites avec une version antérieure !

2. INSTALLATION

Après avoir inséré le CD-Rom dans le lecteur, l'installation du programme devrait débuter automatiquement. Si ce n'est pas le cas faites un clic droit sur le lecteur CD et sélectionnez *"Installer ou exécuter un programme"* ; et si cette option n'apparaît pas, choisissez *"Ouvrir"* après le clic droit, puis allez dans le dossier *"Playfair installation"*, et lancez le fichier *"playfair_setup.exe"*.

Une fois le programme d'installation lancé, suivez les instructions à l'écran jusqu'à ce que le programme soit installé ; il vous faut au moins 15 Mo d'espace libre pour l'installation de cette application.

L'installation étant terminée, vous pouvez lancer le programme.

3. FONCTIONNEMENT DU PROGRAMME

3.1 Sélection du cryptogramme

Il vous est en premier lieu demandé d'insérer un message ; c'est le cryptogramme que vous souhaitez déchiffrer que vous devez entrer ici. Il est à noter que votre message doit être en majuscules, et ne doit comporter aucun autre caractère que des lettres de "A" à "Z" (sans "W"), car il n'aurait dans le cas contraire pas été crypté à l'aide de la méthode de Playfair.

Si votre message n'est pas de la forme ci-dessus, il vous est possible de prétraiter le texte très simplement à l'aide de l'applet disponible à l'adresse suivante : http://apprendre-en-ligne.net/crypto/crypto/pretraiter.html.

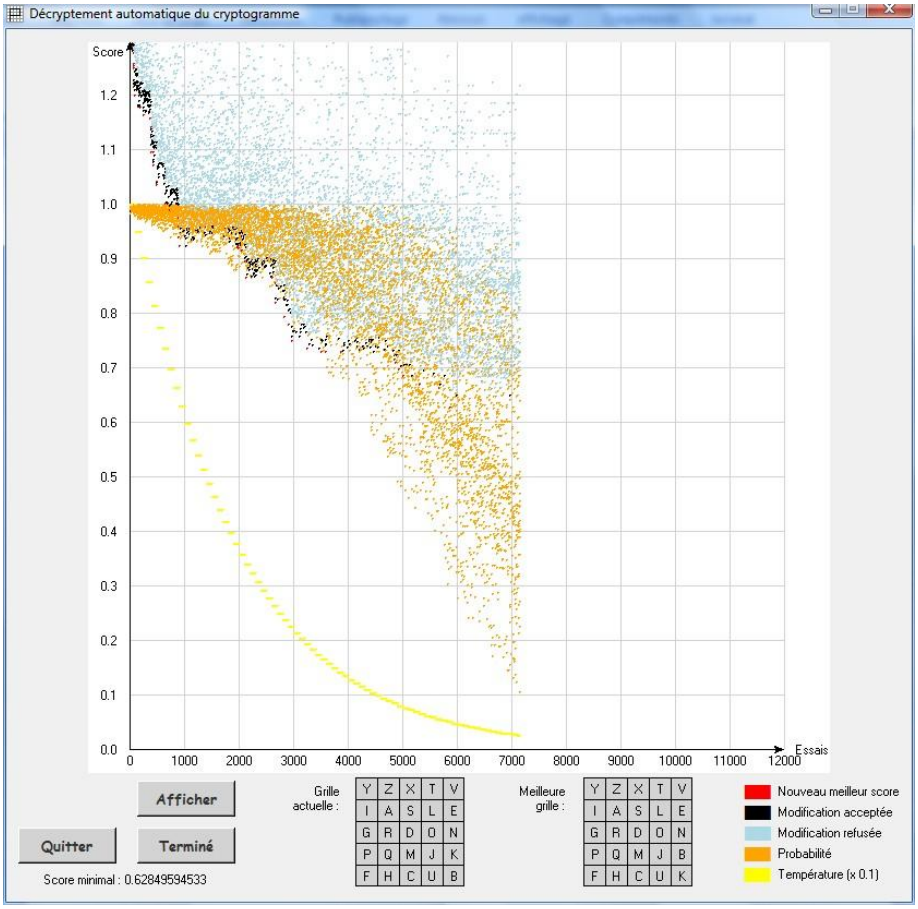
Dans le cas où vous n'avez pas de cryptogramme précis à décrypter, il est possible d'en générer un aléatoirement : le programme en choisira un parmi cinq courts ou cinq longs en fonction de l'option choisie.

3.2 Décryptement

Après avoir entré un texte, la fenêtre de décryptement s'ouvre. Celle-ci se compose d'un graphique, occupant la majeure partie de la place, ainsi que des boutons de contrôle en bas à gauche du graphique, puis de deux grilles à droite des boutons, puis d'une légende dans le coin inférieur droit de la fenêtre. Cet affichage graphique permet de visualiser l'avancement de la recherche afin qu'elle soit plus intuitive, et cela permet de la recommencer dans le cas où elle n'avancerait plus.

Pour commencer la recherche, appuyez sur le bouton "*Commencer*" en bas à gauche, ou sur le bouton "*Quitter*" au-dessous si vous souhaitez quitter l'application.

Une fois la recherche débutée, des points se dessinent dans la zone du graphique. Pour y voir plus clair, voyons ce que représente en réalité ce graphique :



Trois couleurs de points représentent le score obtenu par la grille actuelle en fonction du nombre d'essais : plus le score est élevé, plus la grille est mauvaise ; au contraire, plus le score se rapproche de zéro, plus la grille correspond à la grille utilisée pour chiffrer votre message (consulter le dossier d'explication du programme pour plus de précision sur le calcul du score). On constate que le score varie entre 0 et 1.4. En outre, la grille a de fortes chances d'être la bonne lorsque le score est inférieur à 0.5 en moyenne.

Un point apparaît donc après chaque permutation de la grille. On distingue trois couleurs de points pour le score de la grille :

- Un point s'affiche en **rouge** lorsque la grille obtient un nouveau meilleur score, c'est-à-dire lorsque son score est inférieur à tous ceux de toutes les grilles testées auparavant.
- Si le point apparaît en **noir**, cela signifie que la permutation effectuée dans la grille est acceptée, mais qu'une meilleure grille avait déjà été trouvée précédemment (la présence de ces points est due à l'acceptation d'une partie des mauvaises solutions afin de ne pas converger trop vite vers un optimum local).
- Enfin, si le point est **bleu**, c'est que la modification apportée est au contraire rejetée ; on annule donc la permutation effectuée afin de reprendre la grille précédente.

L'objectif est donc qu'à terme, les points noirs et rouges convergent vers une valeur faible.

Sur ce même graphique sont également représentés des points **orange** : ils représentent la probabilité d'accepter la modification (pour autant que celle-ci soit sous le seuil fixé), qui varie évidemment entre 0 et 1.

Enfin, les points **jaunes** (qui forment une suite de courtes lignes) représentent la température actuelle. Pour pouvoir prendre place sur le graphique, celle-ci a été réduite à un dixième de sa valeur d'origine, c'est-à-dire qu'elle varie en réalité entre 0 et 10, et non entre 0 et 1 comme sur le graphique.

En outre, vous pouvez observer au bas de la fenêtre deux grilles. Celle de gauche représente la grille qui vient d'être acceptée (il n'y a donc pas de changement si la modification est rejetée) ; celle de droite est la meilleure grille trouvée actuellement.

Une fois la recherche démarrée, les boutons de contrôle se modifient : il existe désormais un bouton *"Afficher"* et un bouton *"Terminer"*. Le premier de ces boutons ouvre une nouvelle fenêtre (qui ne perturbe en rien la recherche), dans laquelle est affiché le message décrypté avec la meilleure grille trouvée jusqu'alors. Vous pouvez ensuite fermer simplement cette nouvelle fenêtre en appuyant sur le bouton de gauche, ou quitter l'application avec celui de droite.

Vous pouvez cliquer sur le bouton *"Terminer"* lorsque vous pensez que la grille trouvée correspond relativement bien avec la grille recherchée, donc lorsque le score s'est minimisé de façon satisfaisante (la recherche s'interrompt après 12'000 tentatives si vous n'avez pas demandé à l'arrêter plus tôt).

Vous passez ensuite à la seconde étape en appuyant sur le bouton correspondant sur la nouvelle fenêtre. Cette étape teste les permutations de la grille afin de l'améliorer. Lorsque cette étape est terminée, la recherche prend fin, les résultats sont affichés, et vous êtes invité à recommencer une recherche si besoin, ou à quitter l'application.

Batteries de tests (Annexe 3)

TABLE DES MATIÈRES

1. PREMIÈRE SÉQUENCE DE TESTS

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

3'000 essais ; 7 min				
A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	X	Y	Z

4'750 essais ; 11 min				
A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	X	Y	Z

12'000 essais ; 29 min				
A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	X	Y	Z

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

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

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

8'000 essais ; 20 min				
A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	X	Y	Z

3'250 essais ; 7 min				
A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	X	Y	Z

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

2'250 essais ; 5 min				
A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	X	Y	Z

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

2. SECONDE SÉQUENCE DE TESTS

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

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

4'500 essais ; 10 min				
A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	X	Y	Z

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

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

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

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

9'000 essais ; 21 min				
A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	X	Y	Z

4500 essais ; 11 min				
A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	X	Y	Z

5'500 essais ; 13 min				
A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	X	Y	Z

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

3'750 essais ; 9 min				
A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	X	Y	Z

7'000 essais ; 17 min				
A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	X	Y	Z

2'750 essais ; 6 min				
A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	X	Y	Z

7'500 essais ; 17 min				
A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	X	Y	Z

3'250 essais ; 8 min				
A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	X	Y	Z

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

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

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

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

4'000 essais ; 9 min				
A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	X	Y	Z

3'500 essais ; 8 min				
A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	X	Y	Z

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

11'000 essais ; 27 min				
A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	X	Y	Z

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

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

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

4'750 essais ; 11 min				
A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	X	Y	Z

3'250 essais ; 7 min				
A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	X	Y	Z

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

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

2'000 essais ; 5 min				
A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	X	Y	Z

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

6'000 essais ; 14 min				
A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	X	Y	Z

8'000 essais ; 19 min				
A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	X	Y	Z

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

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

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

3'000 essais ; 7 min				
A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	X	Y	Z

4'500 essais ; 11 min				
A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	X	Y	Z