Automatisation de la cryptanalyse des cryptosystèmes classiques à l'aide d'algorithmes modernes

LU2IN013

Jules Panon Desbassayns de Richemont
Léa Cohen-Solal
Zein Sakkour
dirigé par Valérie Ménissier-Morain

15 juillet 2022

Table des matières

Chapitre 1

Introduction

1.1 Éléments historiques

La **cryptologie** - étymologiquement *science du secret* - regroupe à la fois la **cryptographie** : l'écriture secrète, et la **cryptanalyse** : l'analyse de cette dernière. La cryptographie est à distinguer de la stéganographie *écriture étanche* qui consiste à dissimuler discrètement de l'information, dans le sens où la cryptographie ne cherche pas à dissimuler son contenu, mais bien à le rendre intelligible uniquement par qui-de-droit.

Les origines de l'utilisation de cryptographie pour protéger des données confidentielles remontent à l'Antiquité, et le plus vieux document chiffré actuellement connu date XVIème siècle avant notre ère [1]. Parmi les techniques cryptographiques notables qui nous sont parvenues nous pouvons par exemple citer la *scytale* chez les Spartiates, un bâton d'un certain diamètre autour duquel on enroulait une lanière de cuir pour y écrire un message, lequel ne serait déchiffrable qu'à l'aide d'un autre bâton de diamètre équivalent. Cette technique relève du chiffrement par **transposition**.
Vers le Vème siècle avant notre ère les Hébreux utilisent dans leurs textes religieux de nouvelles méthodes de chiffrement dits par **substitution**. La plus connue d'entre elles est sans nul doute *Atbash*, qui est une méthode de substitution inversée : la première lettre devient la dernière, la deuxième l'avant-dernière, et ainsi de suite.

D'autres méthodes de chiffrement, majoritairement par substitution, ont vu le jour à partir du IIème siècle avant J-C. Comme par exemple le *code de César*, qui est une méthode de chiffrement par **substitution mono-alphabétique** avec un décalage dans la clé, ou le *carré de Polybe*, premier procédé de chiffrement par substitution mono-alphabétique homophonique. La solution à de telles méthodes de chiffrement par substitution mono-alphabétique parallèlement étudiée en cryptanalyse - puisque les deux disciplines sont les deux revers de la même pièce - est l'analyse des fréquences des lettres du texte chiffré. Nous reviendrons dessus en détails prochainement.

Faisons un bond dans le temps pour citer de manière non exhaustive d'autres méthodes de chiffrement. Au XVIème siècle, voient le jour des techniques de chiffrement par **substitution polyalphabétiques** comme le *Chiffre de Vigenère* [2]où une lettre de l'alphabet dans le texte en clair peut être chiffrée de plusieurs manières . Au XIXème siècle est inventé le chiffrement de Playfair, qui est basée sur une méthode de substitution diagrammatique consistant à remplacer un couple de lettres adjacentes par un autre couple choisi dans une grille qui constitue la clé.

Entre 1968 et 1970, un sérial killer surnommé le *Tueur du Zodiaque* a assassiné au moins 5 personnes dans la région de San Francisco et est probablement responsable de nombreux autres non-confirmés. Sa particularité était de narguer les policiers en envoyant de lettres aux journaux locaux, lettres qui contenaient des messages codés. Plusieurs de ces messages n'ont encore jamais été décodés et l'affaire et toujours en cours plus de 50 ans après.

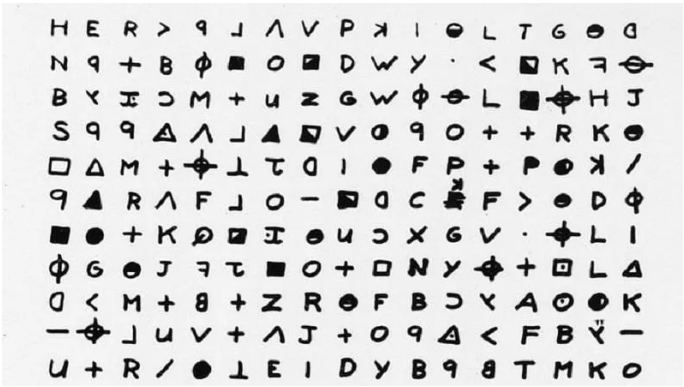

Figure 1 : Exemple de code du tueur du Zodiaque

En 2001, dans une prison de Californie un psychiatre intercepte sur un des prisonniers un message chiffré à l'aide de symboles particuliers, qui lui fait penser au tueur du Zodiaque. Il a alors l'idée de le soumettre au mathématicien Perci Diaconis, qui travaillait comme professeur de Statistiques à l'université de Stanford, juste à côté.
À l'aide de ses étudiants, Diaconis a mit au point une approche mathématique qui a permis de déchiffrer le message de la prison. Cette méthode s'appuie sur les chaînes de Markov. C'est dans la lignée de cette méthode, mais avec un angle d'approche informatique que s'inscrit notre travail de recherche.

1.2 Objectifs

Depuis l'apparition de la cryptographie, les méthodes de cryptanalyse se sont beaucoup diversifiées et ces différentes techniques donnent des résultats plus ou moins concluant. Il est alors intéressant d'étudier ces différentes méthodes, d'en donner leur algorithme, leurs points forts ainsi que leurs points faibles.
Ce projet a pour but dans un premier temps de présenter la méthode de hill-climbing, puis d'étudier l'efficacité de ses variantes sur différents cryptosystèmes, avant enfin de proposer une implémentation qui permet l'automatisation du processus.

Chapitre 2

Développement de méta heuristiques contre les chiffrements par substitution

Le chiffrement par substitution mono-alphabétique [3] est la méthode la plus classique pour chiffrer/déchiffrer un message. Il suffit simplement de changer un symbole du texte clair par un autre. L'échange d'un signe à l'ordre donne naissance à une clé. Cette clé correspond à chaque permutation de symbole de l'alphabet utilisé.Donnons l'exemple le plus connu, appelé *Code de César*[4]. Lors de ses batailles, l'empereur Jules César chiffrait ses messages en utilisant une clé bien particulière. il décalait chaque lettre de 3 rangs vers la droite. Le déchiffrement du message "erqmrxu" est "bonjour".

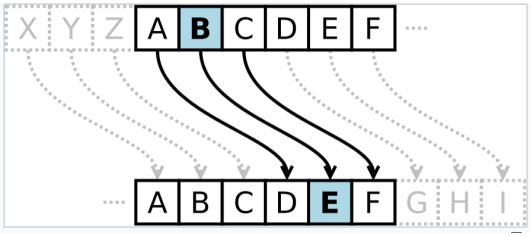

Figure 2 : Représentation du code de César

Plus généralement, le code de César est une méthode de chiffrement par substitution mono-alphabétique qui fonctionne par décalage de la première lettre de l'alphabet dans la clé de chiffrement. Ce code est cependant peu robuste car il n'offre que 25 possibilités de clé.

Les **méta heuristiques** ne sont pas des algorithmes propres à la cryptanalyse. Ce sont des algorithmes itératifs permettant de résoudre un problème que l'on ne peut pas résoudre de manière directe. Une méta heuristique permet en effet de rechercher la solution d'un problème en s'en approchant le plus possible. Nous considérerons que le problème a été résolu lorsqu'une approximation de la meilleure solution aura été trouvée. Il ne s'agit donc pas de trouver une seule solution exacte mais bien une approximation celle-ci. C'est ce principe que nous allons utiliser dans notre algorithme de hill-climbing.

2.1 Algorithme de hill-climbing

2.1.1 Idée de la démarche

Tout d'abord, rappelons que si on s'autorise à remplacer aléatoirement chaque lettre par une autre, le nombre de combinaisons possibles explose : 26 ! = 4.032915.10**26. Cela est impossible à tester efficacement avec nos ressources informatiques, et il nous est donc obligatoire de ruser un peu pour trouver une bonne solution assez rapidement. Pour attaquer ce problème, l'approche que nous étudierons repose sur les statistiques.

```
ERQMRXUOHFWHXUVDFKHTXHMXOHVWHVRXK
DLWHXQHSDQRXLVVHPQWLPPHQVGDQVWR
XWFHTXHWXHQWUHSUHQGUDVGDQVWDYLH
```

Figure 3 : Texte chiffré par substitution

Par exemple, dans le texte chiffré ci-dessus il nous paraît instinctif de se dire que le caractère qui chiffre le E, et ceux qui chiffrent le A, le I et le S devraient être les plus fréquents. Cette idée est une première piste à l'algorithme de hill-climbing, mais il est trop simple et ne fournit que des résultats médiocres, même avec des très longs textes, qui ressemblent à la langue originale (dans l'exemple le français) mais ne veulent rien dire.

Pour obtenir une bonne cryptanalyse, nous nous intéressons à la ressemblance du texte à la langue d'origine. Pour cela, nous nous concentrons sur les enchaînements de lettres. Certains sont très probables, d'autres beaucoup moins, et cela nous permet d'évaluer si la suite semble correspondre à sa langue, ici à du français.

C'est l'idée du hill-climbing : associer à toute suite un score de plausibilité pour vérifier si elle ressemble bien à sa langue supposée.

2.1.2 Définition de l'algorithme

La méthode **hill-climbing**[5] (dite méthode d'escalade) est une méthode itérative locale qui vise à rechercher un optimum local. Elle prend trois objets : une configuration initiale, une fonction permettant de générer des configurations proches de la configuration initiale, et une dernière fonction **fonction fitness** capable d'évaluer la qualité de chaque configuration. La méthode de hill-climbing consiste ensuite à choisir la meilleure configuration voisine et répéter la solution jusqu'à ce qu'aucune ne soit (significativement) meilleure. On dit alors que nous avons trouvé un optimum local.
Concrètement, lorsque l'on s'intéresse à la cryptanalyse d'un texte chiffré par substitutions mono-alphabétiques, l'enjeu réside dans la découverte de la **clé** (configuration des lettres de l'alphabet) qui a été utilisée pour le chiffrement. Cette clé est assimilable à un vecteur à 26 coordonnées, et dont on va échanger au moins 2 coordonnées (lettres) à chaque itération de l'algorithme de hill-climbing, générant ainsi une clé potentielle. Si l'évaluation de cette clé potentielle est moins bonne que celle de la clé initiale, on procède à une autre substitution en partant de la clé initiale. Dans le cas contraire la clé potentielle devient la clé initiale et on recommence l'algorithme.

Algorithm 1: SOLVER($puzzle, num\_trials, num\_swaps, scoringFunction$)

input : substitution cipher $puzzle$, parameters num_trials and num_swaps controlling the amount of computation, and scoring function $scoringFunction$

output : best decryption key found $best\_key$ and its corresponding score $best\_score$, locally maximizing the scoring function

```
best_score ← −∞
for i ← 1 to num_trials do
    key ← random permutation of the alphabet
    best_trial_score ← −∞
    for j ← 1 to num_swaps do
        new_key ← key with two of its letters swapped randomly
        score ← score puzzle using scoringFunction after decrypting it with new_key
        if score > best_trial_score then
            key ← new_key
            best_trial_score ← score
        endif
    end
    if best_trial_score > best_score then
        best_key ← key
        best_score ← best_trial_score
    endif
end
return {best_key, best_score}
```

Figure 4 : Pseudo-code de l'algorithme de hill climbing

On voit déjà se dessiner plusieurs problématiques : Quand doit-on s'arrêter ? Est-on sûrs d'avoir trouvé la bonne solution quand l'algorithme s'arrête ? Comment mesurer au mieux la qualité d'une configuration de clé ?

Avant de détailler le principe des fonction fitness, il est important de donner les différents outils que nous avons utilisés.

Les différentes fonctions que nous avons implémenté sont centrées sur un certain paramètre que l'on appellera n. Ce n est un chiffre pouvant varier entre 1 et 5 et qui représente le nombre de caractères, le **n-gramme**, utilisé comme référence pour notre évaluation. Utiliser une fonction fitness avec un n égal à 3 revient à évaluer un déchiffrement selon les fréquences des trigramme (enchaînements successifs de 3 lettres).

Pour mener à bien notre étude, nous avons majoritairement manipulé des statistiques de la distribution des n-gramme en anglais et en français, mais le programme peut tout-à-fait convenir pour toute autre langue.

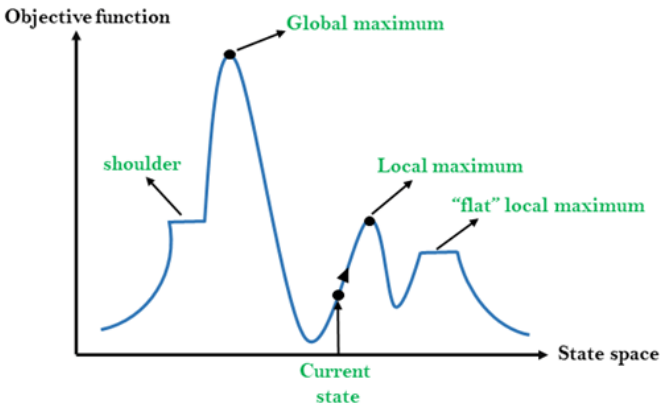

Figure 4 : Illustration de l'algorithme de hill climbing

2.2 Fonctions fitness

2.2.1 Fitness function : N-gramme

Les **fitness function** (que nous traduirons en français *fonction fitness*) nous permettent, dans le cadre de la cryptanalyse, d'évaluer le bon déchiffrement d'un texte[6]. Nous avons implémenté des fonctions fitness qui nous permettront d'évaluer la qualité d'une tentative de déchiffrement du texte chiffré par une clé précise, et ainsi d'évaluer implicitement la "pertinence" de cet algorithme de déchiffrement. Ces différentes fonctions fitness nous donneront un score, représentant le "score du déchiffrement" qui est environ proportionnel à la lisibilité du texte déchiffré. Cependant, nous verrons dans la suite que le score fitness n'est pas un indicateur suffisant de lisibilité.

Dans le cas de la fonction fitness n-gramme, nous allons tout d'abord fixer n égal à 2,3,4 ou 5. Si par exemple n=2, nous allons évaluer le déchiffrement du texte par rapport à des fréquences de bi-gramme. Une fréquence de bi-gramme correspond à la distribution d'un enchaînement donné de 2 lettres dans la langue. Par exemple les bi-gramme ER, IN, ON ont une fréquence bien plus importante en français que les bi-gramme PL, ZD ou TM.

Une fois ce n fixé, nous allons parcourir le texte déchiffré n lettres par n lettres et additionner les fréquences des différentes combinaisons de n lettres. Pour obtenir un score fitness intelligible et n'explosant pas sur des longs textes nous avons choisi de le calculer de manière logarithmique. Nous ajoutons la somme des probabilités des n-gramme du texte en logarithme 2.

$$Score\ fitness = \sum_i log_2(p_i)$$

Figure 5 : formule du score de fitness n-gramme

Avant de juger le score obtenu, il est important de comparer celui-ci au score du texte clair, c'est à dire le score que l'on voudrait idéalement atteindre. C'est en comparant la différence entre les deux scores que l'on peut commencer à juger l'efficacité de l'algorithme de déchiffrement. Mais cela n'est pas suffisant et il est aussi important de compter le nombre de lettres correctes dans la clé retournée par l'algorithme.

```
dico = cipher.ngram(4,path_stats)
text=fichier.NETTOYER_lire_fichier("./text/textCLAIRE.txt")
textchiffre = cipher.encipher(text,cle_de_cryptage)

score1 = cipher.fitness1(text,dico)
score2 = cipher.fitness1(textchiffre,dico)
print("le score du texte clair est "+ str(score1))
print("le score du texte chiffré est "+ str(score2))
```

```
le score du texte clair est 16555.698525018357
le score du texte chiffré est 9578.804937977618
```

Figure 6 : Différence entre la fonction fitness appliquée au texte chiffré et au texte déchiffré

2.2.2 Fonction fitness : Relations de Pearson

L'étude des corrélations de Pearson[7] permet de modéliser la relation linéaire entre deux variables aléatoires continues X et Y. Cette corrélation est caractérisée par un coefficient de corrélation appelé **r de Pearson**. Plus ce coefficient se rapproche de 1, plus X et Y varient ensemble dans le même sens. Une corrélation parfaite des variables X et Y produit une densité jointe $\rho_{X,Y} = 1$.

Nous allons ici utiliser cette corrélation, où X représente la fréquence de chaque caractère dans le texte chiffré en cours de transformation, $\overline{X}$ la moyenne de la variable aléatoire X, Y la fréquence des caractères dans la langue de référence et $\overline{Y}$ la moyenne de la variable aléatoire Y. On peut modéliser cette corrélation de Pearson par la formule suivante :

$$r = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \bar{X})^2}\sqrt{\sum_{i=1}^{n}(Y_i - \bar{Y})^2}}$$

Figure 7 : formule du coefficient de corrélation de Pearson

Notre deuxième fonction fitness nous permet donc de calculer le score fitness d'un déchiffrement grâce à l'étude du r retourné.

2.3 Automatisation

Nous avons choisi d'automatiser la cryptanalyse en Python3.

2.3.1 Applications au hill-climbing et résultats

Dictionnaires de fréquences

Pour pouvoir manipuler toutes ces fréquences de n-gramme, que ce soit pour fitness n-gramme ou pour fitness Pearson (n=1), nous avons choisi d'implémenter des dictionnaires, car cette structure de données est plus efficace pour les opérations auxquelles nous aurons recours. Le dictionnaire des fréquences se présente ainsi :

- Le n, égal à 1,2,3,4 ou 5, est la valeur de la clé '-n' du dictionnaire.
- Chaque combinaison de n lettres représente une clé et sa fréquence représente sa valeur. Si n=1 il n'y aura aucune confusion entre le '-n' représentant le nombre de lettres choisi et la clé 'n' du dictionnaire. Ce dictionnaire sera au centre de l'évaluation faite dans les différentes fonctions fitness.

Pour générer un dictionnaire de fréquences d'un texte nous avons implémenté la fonction

```
def lire_ngrame(nomfichier):
```

qui ouvre le fichier *nomfichier*, lit le premier caractère de la première ligne du fichier qui représentera le n et crée les fréquences de n-gramme en parcourant tout le texte.

Les statistiques desdits n-gramme ont été stockés dans un fichier *.txt* grâce à un script python *Creer-stats.py* qui va permettre d'organiser les fréquences pour avoir plus de facilités à y accéder. Ce sont ces statistiques qui nous permettent ensuite de créer les dictionnaires qui servent de référence statistique aux fonctions fitness. En particulier le cas n=1 correspond au dictionnaire utilisé par fitness Pearson, n=2 correspond à fitness bi-gramme, n=3 à fitness tri-gramme etc...

Nous nous retrouvons donc avec des fichiers de la forme suivante :

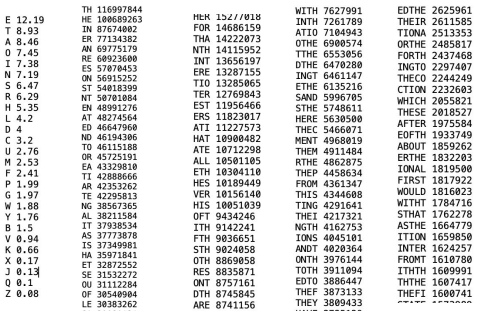

Figure 8 : Échantillons des bibliothèques de n-gramme que nous avons utilisé

Lors d'un déchiffrement, nous avons décidé de ne pas tenir compte des majuscules, minuscules, accents, signes de ponctuations et espaces. En effet ceux-ci augmenteraient encore le nombre de possibilités et ralentiraient ainsi le programme. De plus nous verrons que l'algorithme de hill climbing est assez efficace pour se contenter des 26 caractères de l'alphabet latin. Nous avons donc implémenté la fonction

```
def Nettoyer_lire_fichier(nomfichier)
```

qui supprime les ponctuations, espaces, accents, et met les caractères en majuscule (de 'A' à 'Z').

Instinctivement nous avons choisis de faire ces statistiques par rapport à la langue française. Le texte intégral de *Germinal*, qui nous a été fourni, a été dans un premier temps le support des statistiques utilisés dans notre étude. Au fil de nos tests nous nous sommes aperçus que le déchiffrement avec les tetra-gramme et penta-gramme donnait des résultats contre-intuitifs. Cette différence d'efficacité entre n=2 ou 3 et n=4 ou 5 s'expliquait par le fait que le texte *Germinal* n'est pas assez long pour être représentatif de la langue française en terme de fréquences des n-gramme quand n≤4.

Nous nous sommes donc tournés vers la langue anglaise pour la suite de notre étude, et nous sommes procurés en ligne des statistiques établies sur un corpus de textes plus conséquent[8].

Hill-climbing

La fonction de Hill-climbing a pour but de tester en boucle un grand nombre de clés potentielles et d'évaluer à chaque tour le score de la clé potentielle à l'aide d'une certaine fonction fitness. Si celui-ci est meilleur que celui de la clé de référence, la boucle suivante prendra la clé essayée comme nouvelle clé de référence.

```
def hillClimbing(fitness,texte,dictionnaire,
                 NBITERGLOB,NBITERSTATIC,cle)
```

Paramètres : Nous passons en paramètre une fonction fitness, le texte chiffré, le dictionnaire associé à la fonction fitness, et une clé qui correspond à une combinaison des 26 lettres de l'alphabet. Cela nous laisse la possibilité, si besoin, de partir d'une clé particulière. Si la clé passée en paramètre est vide on génère aléatoirement une clé initiale.

Initialisation : Nous initialisons les compteurs d'itérations statiques *cmp* et globales i à 0, et nous les incrémenterons de 1 à chaque tour de boucle s'ils restent inférieurs respectivement à *NBITERSTATIC* et *NBITERGLOB*. Nous initialisons aussi deux variables pour stocker les scores associés aux clés potentielles des tours *n-1* et *n*. Le scoreParent est initialisé au score de la clé passée en paramètres, et correspond ensuite à celui de la clé *n-1*. Le scoreEnfant est initialisé à 0 et correspond au score de la clé *n*.

Boucle : Tant que $cmp<NBITERSTATIC$ et $i<NBITERGLOB$:
- Si le scoreEnfant est supérieur au scoreParent, scoreParent devient scoreEnfant, la clé Parent est remplacée par la clé Enfant. On comparera ensuite le score de la clé Enfant sur laquelle 2 lettres ont été substituées avec scoreEnfant.
- Sinon on incrémente les compteurs d'itérations globales et locales de 1.

Fin : La sortie de l'algorithme s'effectue si :
- Trop de tours ont été effectués sans changement significatif du score fitness à la hausse ; c'est le cas d'atteinte de maximum d'itérations locales.
- L'algorithme fait trop de tours et atteint le maximum d'itérations globales

Compte-tenu de l'aspect heuristique de la démarche de hill-climbing, il était capital de déterminer quels paramètres fournissent les résultats les plus performants. Ainsi, la prochaine partie du projet est consacrée à la variation de variables qui représentent le nombre d'itérations, ainsi qu'à la comparaison des différentes fonctions fitness.

2.3.2 Optimisation des paramètres et statistiques

Après avoir lancé l'algorithme quelques fois pour nous familiariser, nous commençons à remarquer des comportements différents selon les paramètres utilisés. Nous avons alors émis l'hypothèse que l'efficacité du déchiffrement varie selon la fonction fitness choisie et la longueur du texte. Pour confirmer cette hypothèse, nous avons implémenté un script python qui nous permet de générer des batteries de test sur la fonction hill-climbing. Nous avons fait tourner en continu ces tests pendant plusieurs jours sur les ordinateurs de la PPTI. C'est la commande Unix *nohup* qui nous a permis de maintenir le processus lancé ininterrompu.

La démarche propre à ces tests est empirique. Nous rappelons ainsi que cette analyse a pour but de fournir rapidement des résultats acceptables, mais que ces résultats peuvent ne pas être optimaux.
Dans un premier temps nous faisons varier le maximum d'itérations globales autorisé *NBITERGLOB* entre 1000 et 10000 avec un pas de 500, puis pour chaque valeur de *NBITERGLOB* nous faisons varier le maximum d'itérations

locales *NBITERSTATIC* de 500 à 4000 avec un pas de 250 et en s'assurant que *NBITERSTATIC<NBITERGLOB*.
Pour chaque couple *(NBITERGLOB , NBITERSTATIC)* nous avons effectué 20 tests, de sorte à avoir des statistiques moyennes qui soient suffisamment représentatives. Pour chacun de ces tests nous avons relevé le nombre de caractères corrects de la clé par rapport à la clé de chiffrement que nous connaissions, ainsi que le score fitness et le temps d'exécution de l'algorithme de hill climbing. À partir de ces statistiques nous avons calculé des moyennes pour chaque couple *(NBITERGLOB , NBITERSTATIC)*. Ce sont les colonnes *Clé* , *Score*, *Temps* du tableau en annexe. Enfin nous avons appliqué cette logique de test à chaque fonction fitness : Pearson, bi-gramme, ..., penta-gramme.
Les statistiques obtenues sont résumées dans le tableau donné en Annexe I.

Textes courts

Nos tests sur des textes courts (200 à 500 caractères) nous ont permis de tirer les conclusions suivantes :
- On remarque que le déchiffrement avec fitness tri-gramme donne les meilleurs résultats.
- Les scores commencent à devenir bons lorsque le nombre d'itérations statiques dépasse les 1250. En revanche lorsque *NBITERSTATIC* dépasse 2000 les résultats n'augmentent plus et le temps d'exécution devient trop long.
- Il est idéal de garder *NBITERGLOB* entre 2000 et 4000, car en deçà de 2000 itérations globales les résultats moyens sont insatisfaisants. Augmenter à plus de 5000 le nombre d'itérations globales n'a pas d'intérêt non plus car la sortie de l'algorithme sera provoquée par l'atteinte de *NBITERSTATIC*.

En regardant de plus près l'exécution de la fonction hill-climbing nous remarquons que le score augmente de manière logarithmique par rapport au temps :

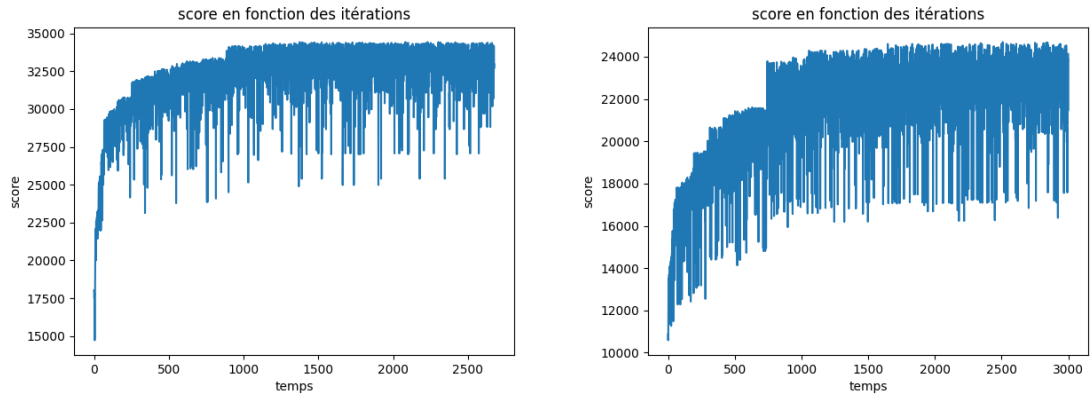

Figure 9 : Évolution du score des bi-gramme et tri-gramme (respectivement) en fonction des itérations

A partir d'environ 1500 itérations globales pour les bi-gramme et 1000 pour les tri-gramme, le score augmente très lentement. Cela signifie que dans un plan topologique, l'algorithme de hill-climbing nous fait rapidement rentrer sur un sommet. À ce moment là, le score de déchiffrement augmente très vite, mais "l'escalade du sommet" (littéralement *hill-climbing*) est de plus en plus lente. Cela vient appuyer la conclusion faite qu'il ne sert à rien d'augmenter *NBITERGLOB* passé un certain point car cela ne fait plus augmenter significativement le score. Ce sera quasi-systématiquement l'atteinte de *NBITERSTATIC* qui provoquera la fin du programme. Le choix du nombre d'itérations statiques est donc plus "important" que celui du nombre d'itérations globales.

Textes longs

Pour des textes plus longs (environ 1500 caractères) nous tirons les conclusions suivantes :
- On remarque que le déchiffrement avec les tétra-gramme donne les meilleurs résultats.
- Les scores sont optimaux lorsque *NBITERSTATIC* est compris entre 2000 et 3000.
- Là aussi c'est quasi-systématiquement l'atteinte du maximum d'itérations locales qui provoque la fin de l'algorithme, donc *NBITERGLOB* peut être fixé entre 3500 et 6000.

2.3.3 N-gramme optimales

Grâce aux jeux de tests accumulés sur différents textes, il nous est possible de décrire le comportement typique des différentes fonctions fitness. Il faut tout de même garder en tête que le hill climbing est une marche aléatoire, et que cela implique par définition que les résultats varient d'un test à l'autre même avec des conditions identiques.

- La fonction fitness n-gramme avec n = 2 est très rapide et donne des résultats satisfaisants pour des textes courts. Elle manque cependant de précision sur des textes de plus de 500 caractères et produit parfois des déchiffrements peu lisibles.
- La fonction fitness n-gramme avec n = 3 est rapide et très performante, que ce soit sur des textes courts ou longs et réussit généralement à déchiffrer l'ensemble du texte de façon lisible.
- La fonction fitness n-gramme avec n = 4 est assez efficace pour des textes courts, et très efficace sur des longs textes. Au dessus de 1000 caractères on peut s'attendre à au moins 24 lettres correctes à chaque lancer, mais en raison de la taille du dictionnaire le temps d'exécution de fitness tetra-gramme est assez important (environ 3s pour 300 caractères et environ 12s pour 1500).
- La fonction fitness n-gramme avec n = 5 n'est pas vraiment efficace sur des textes courts, où elle trouve rarement plus de 20 lettres de la clé de chiffrement. Pour les textes plus longs elle fournit des résultats convenables mais avec un temps d'exécution beaucoup trop long (jusqu'à +20s). La complexité temporelle de penta-gramme fait qu'on lui préférera tetra-gramme.
- La fonction fitness Pearson nous a fourni des résultats décevants. Elle ne fait que mesurer le coefficient de corrélation entre la fréquence de caractères du texte chiffré en cours de transformation et celle de la langue. Nous nous retrouvons ainsi parfois avec des textes qui "ressemblent" à de l'anglais (ou du français) mais qui sont totalement illisibles. Cela se produit quand plusieurs lettres de fréquences similaires sont interchangées par la marche aléatoire. Pearson peine à les remettre à la bonne place faute d'outil de lien entre les caractères. En conclusion, la fonction fitness Pearson n'est pas bien adaptée au hill-climbing, mais elle permet de donner un indice de "clarté" de la langue intéressant pour mesurer la lisibilité d'un déchiffrement. Plus ce déchiffrement ressemblera à la langue d'origine, plus le coefficient de Pearson sera proche de 1.

TABLE 2.1 – PEARSON

Ces quelques résultats nous montrent que le r de Pearson est proportionnel à la justesse de la clé dans une certaine mesure, mais que des textes complètement faux peuvent aussi avoir un r très élevé. Un score de fitness Pearson aussi proche de 1 devrait pourtant correspondre à la langue d'origine.

clé	score obtenu
5/26	*0.2918653*
8/26	*0.395423*
9/26	*0.35289*
11/26	*0.625217*
14/26	*0.7918653*
2/26	*0.995289*

2.3.4 Combinaisons de n-gramme

La fonction hill-climbing permet d'effectuer un grand nombre d'itérations jusqu'à ce que le score atteigne un extremum. Cependant cet extremum n'est pas forcément l'extremum global, qui correspond au texte clair. La tentative de déchiffrement qui se bloque dans un extremum local peut d'ailleurs ne même pas être lisible.

Notre algorithme de hill climbing est une marche aléatoire qui va effectuer un brassage de clé à chaque tour de boucle en échangeant deux caractères. Pour essayer de sortir des extrema locaux dans lesquels aurait pu s'aventurer notre fonction, il serait nécessaire de renforcer ce brassage aléatoire. Forts de ce constat, nous nous sommes mit en tête de combiner des fonctions fitness dans le même appel à notre fonction de cryptanalyse. Notre hypothèse était que cela permettrait de débloquer la fonction de hill climbing bloquée dans un mauvais extremum grâce à une deuxième fonction fitness. Ainsi nous obtiendrons un meilleur score final, tout en gardant un temps d'exécution raisonnable car chaque appel à hill climbing se ferait en moins d'itérations.

Nous avons donc implémenté un script en python permettant de combiner 2 fonction fitness à la suite. En théorie, l'idée était d'appeler dans un premier temps l'algorithme de hill climbing avec une fitness function rapide (n = 2 ou 3) qui ferait un travail de gros sur le texte chiffré, et retournerait une clé intermédiaire. Cette clé serait ensuite passée en paramètre du second appel de la fonction de hill climbing, avec qu'une autre fitness function plus précise (n = 1, 4 ou 5) qui permettrait d'affiner la clé davantage pour obtenir un texte déchiffré le plus proche possible du texte clair.

Une fois le script créé nous avons lancé des batteries de test sur les différentes combinaisons de fitness. Nous avons essayé toutes les combinaisons *(n,m)* telles que $n, m \in [[2,5]]$ et $n \leq m$. Pour chacune de ces combinaisons, nous avons effectué 12 tests de sorte à obtenir des résultats moyens représentatifs.

Le tableau ci-dessous (voir Table 2.2) modélise l'ensemble des combinaisons que nous avons testées. Par exemple pour la combinaison (2,3), nous avons cherché le nombre d'itérations globales et statiques optimal pour les bi-gramme et tri-gramme. On applique la fonction de hill-climbing à fitness bi-gramme, on stocke les scores et la clé obtenue à la fin de ce déchiffrement, puis on applique une nouvelle fois la fonction de hill-climbing à fitness tri-gramme, avec comme clé de départ la clé obtenue en fin d'appel de bi-gramme. Généralement, combiner 2 fonction fitness avec le même n-gramme n'entraîne pas d'augmentation du score final. Une combinaison de la forme *(n, n+k)* avec $k > 1$ permet parfois d'augmenter le score et le nombre de lettres trouvées, mais cela n'est pas fiable comme en témoigne la ligne associée à la combinaison (4,5) du tableau, où le score régresse.

Nous tirons de cette analyse que, bien qu'elle soit intéressante, la démarche de combiner des fonctions fitness par des appels successifs de hill climbing n'est pas concluante. En effet, le brassage aléatoire de la clé à chaque appel de hill climbing est trop important, et faire un premier travail d'affinage en gros de la clé ne sert à rien puisque cette clé sera ensuite décomposée. Les quelques succès dont témoigne le tableau sont imputables à l'ordre d'appel des fitness functions. Par exemple lors du test de la combinaison (2,5), le hill climbing avec bi-gramme nous donne en moyenne 18,8 lettres correctes, et le hill climbing avec penta-gramme en donne 25,2 car il est plus précis.

Des lignes du tableau ci-dessous (voir TABLE 2.2) figurent en double pour illustrer le manque de fiabilité de la méthode des combinaisons.

Cependant nous avons remarqué qu'un appel à fitness Pearson sur un texte déchiffré au préalable est assez intéressant, puisque le r retourné est un indicateur supplémentaire de la qualité du déchiffrement. En particulier nous avons remarqué empiriquement que le r renseignait assez bien sur la lisibilité du texte obtenu (voir TABLE 2.3).

2.4 Conclusion

Au cours de ce projet nous avons pu faire le tour de certaines techniques de cryptanalyse applicables sur des chiffrements par substitution mono-alphabétique. En particulier nous nous sommes attardé sur l'analyse de fréquence au travers de l'algorithme en définitive très efficace de hill climbing.
En pratique, face à ce genre de technique, les résultats obtenus et détaillés dans ce rapport nous invitent à distinguer les cas suivants :

- Si le texte est très court, moins de 200 caractères, la cryptanalyse par analyse de fréquence donne des résultats assez moyens. Il faudrait alors lui préférer une autre approche.

- Si le texte est court, entre 200 et 500 caractères, les meilleurs résultats ont été obtenus en faisant appel pour le hill climbing à la fitness function tri-gramme. Pour obtenir le meilleur rapport entre la pertinence du déchiffrement et le temps d'exécution, nous préconisons de fixer le nombre maximal d'itération locales à 1500, et le nombre maximal d'itérations globales à 3500. Cette opération devrait prendre moins de 4 secondes.

- Si le texte est long, plus de 1500 caractères, nous recommandons d'utiliser pour l'algorithme de hill climbing la fonction fitness tetra-gramme ou penta-gramme. Dans un soucis d'efficacité, en tenant compte de la qualité de la cryptanalyse et de sa rapidité, nous recommandons de fixer le nombre maximal d'itération locales à 2500, et le nombre maximal d'itérations globales à 6000. Si le texte ne dépasse pas les 2000 caractères, le temps d'exécution devrait être inférieur à 20 secondes.
- Si le texte chiffré a une longueur comprise entre les deux derniers cas énoncés, l'utilisateur peut aussi bien utiliser la fonction fitness avec n=3 ou n=4. Les maximums devrait alors être de 2000 itérations locales et de 4500.

Chapitre 3

Annexe : tableaux de statistiques

3.1 Statistiques pour un texte de 300 caractères

TABLE 3.1 – BI-GRAMME 300

Iterations Globales	Iterations Locales	Clé	Score	Temps
2000	*750*	*22.25/26*	*39988.600*	*2.037*
2500	*1000*	*22.2/26*	*39992.033*	*2.417*
2500	*1250*	*22.9/26*	*40036.755*	*2.512*
3000	*2500*	*22.2/26*	*39999.609*	*3.082*
3500	*750*	*22.65/26*	*39979.500*	*2.364*
3500	*1000*	*22.6/26*	*39993.850*	*2.608*
3500	*1750*	*22.5/26*	*39996.806*	*3.392*
3500	*3000*	*23.1/26*	*40043.142*	*3.593*
4000	*1750*	*23.05/26*	*40029.701*	*3.460*
4000	*2250*	*22.3/26*	*39972.386*	*3.903*
4000	*2750*	*22.7/26*	*40008.763*	*4.056*
4000	*3000*	*22.6/26*	*39999.047*	*4.083*
4000	*3250*	*24.15/26*	*40112.429*	*4.099*
4500	*2750*	*23.05/26*	*40043.188*	*4.283*
4500	*3000*	*24.55/26*	*40118.422*	*4.504*
4500	*3500*	*24.05/26*	*40048.389*	*4.588*
4500	*4000*	*22.2/26*	*39947.538*	*4.609*
5000	*1750*	*22.0/26*	*39968.217*	*3.704*
5000	*3750*	*22.8/26*	*39975.115*	*5.058*
5000	*4000*	*22.9/26*	*39985.767*	*5.123*
5500	*750*	*24.2/26*	*40118.422*	*2.482*
5500	*1750*	*23.75/26*	*40116.268*	*3.528*
5500	*2000*	*24.45/26*	*40120.036*	*3.921*
5500	*2500*	*22.65/26*	*39993.034*	*4.698*
5500	*2750*	*23.15/26*	*40041.241*	*4.727*
5500	*3000*	*22.7/26*	*39972.830*	*5.027*
5500	*3250*	*23.15/26*	*40044.702*	*5.093*
5500	*3500*	*22.05/26*	*39972.692*	*5.132*
6000	*2250*	*23.15/26*	*40042.456*	*4.156*
6000	*3500*	*22.3/26*	*39968.496*	*5.415*
6000	*3750*	*22.8/26*	*40043.822*	*5.487*
6000	*4000*	*23.8/26*	*40080.056*	*5.786*

TABLE 3.2 – TRI-GRAMME 300

Iterations Globales	Iterations Locales	Clé	Score	Temps
2500	*1750*	*22.7/26*	*34245.244*	*2.681*
2500	*2000*	*22.2/26*	*34218.207*	*2.678*
3000	*1250*	*23.9/26*	*34498.324*	*3.004*
3000	*2750*	*22.8/26*	*34277.278*	*3.214*
3500	*1000*	*23.55/26*	*34420.275*	*2.709*
3500	*1250*	*23.5/26*	*34486.696*	*3.094*
3500	*2250*	*22.75/26*	*34274.977*	*3.664*
3500	*3250*	*22.8/26*	*34258.847*	*3.749*
4000	*1500*	*22.7/26*	*34255.901*	*3.344*
4000	*3500*	*22.05/26*	*34031.815*	*4.279*
4500	*2250*	*23.2/26*	*34312.671*	*4.113*
4500	*3250*	*22.45/26*	*34247.870*	*4.789*
5000	*1000*	*22.05/26*	*34097.879*	*2.858*
5000	*1500*	*22.4/26*	*34290.460*	*3.365*
5000	*3250*	*23.8/26*	*34499.712*	*5.128*
5500	*750*	*24.45/26*	*34649.691*	*2.395*
5500	*1250*	*22.75/26*	*34278.783*	*3.221*
5500	*1750*	*22.8/26*	*34213.551*	*3.865*
5500	*2000*	*22.55/26*	*34275.536*	*4.183*
5500	*2250*	*22.95/26*	*34261.244*	*4.421*
5500	*4000*	*23.6/26*	*34495.810*	*5.758*
6000	*1250*	*23.5/26*	*34489.919*	*3.252*
6000	*2250*	*22.4/26*	*34272.127*	*4.306*

TABLE 3.3 – TETRA-GRAMME 300

Iterations Globales	Iterations Locales	Clé	Score	Temps
2000	*750*	*22.5/26*	*28934.638*	*2.083*
2000	*1000*	*23.3/26*	*29284.407*	*2.203*
2000	*1250*	*22.45/26*	*29046.843*	*2.213*
2500	*1250*	*23.0/26*	*29202.284*	*2.698*
2500	*2250*	*21.7/26*	*28705.755*	*2.758*
3000	*1500*	*22.95/26*	*29332.017*	*3.174*
3000	*2500*	*21.55/26*	*28819.780*	*3.308*
3000	*2750*	*22.8/26*	*29234.546*	*3.315*
3500	*750*	*21.55/26*	*28672.206*	*2.589*
3500	*1250*	*23.05/26*	*29231.535*	*2.998*
4000	*3500*	*21.55/26*	*28799.692*	*4.404*
4500	*1000*	*21.8/26*	*28780.991*	*2.801*
4500	*1500*	*21.3/26*	*28764.088*	*3.330*
5000	*2000*	*23.85/26*	*29682.348*	*3.929*
5000	*2250*	*22.5/26*	*29192.210*	*4.323*
5500	*1750*	*23.85/26*	*29684.477*	*3.688*
6000	*3250*	*22.65/26*	*29257.430*	*5.403*
6000	*4000*	*22.1/26*	*28883.534*	*6.242*

TABLE 3.4 – PENTA-GRAMME 300

Iterations Globales	Iterations Locales	Clé	Score	Temps
2500	*2000*	*19.1/26*	*22210.145*	*2.965*
3000	*2000*	*19.35/26*	*22808.969*	*3.527*
3000	*2250*	*19.3/26*	*22838.095*	*3.507*
3500	*1500*	*19.5/26*	*22904.669*	*3.638*
3500	*2250*	*19.1/26*	*22801.731*	*4.035*
4000	*500*	*19.25/26*	*22784.414*	*2.054*
4000	*1500*	*19.45/26*	*22873.637*	*3.589*
4000	*1750*	*20.95/26*	*23605.558*	*4.080*
4000	*3250*	*19.2/26*	*22862.520*	*4.677*
4500	*1250*	*20.2/26*	*23460.553*	*3.448*
4500	*1500*	*19.0/26*	*22891.732*	*3.906*
4500	*2000*	*20.6/26*	*23627.060*	*4.337*
4500	*2250*	*19.35/26*	*22893.935*	*4.401*
4500	*2500*	*19.0/26*	*22927.946*	*4.571*
4500	*3250*	*19.45/26*	*22898.931*	*5.236*
4500	*3500*	*21.45/26*	*24243.010*	*5.262*
5000	*1000*	*20.15/26*	*23584.914*	*3.410*
5000	*1250*	*19.15/26*	*22857.979*	*3.308*
5000	*1500*	*20.05/26*	*23516.637*	*3.881*
5000	*2000*	*21.15/26*	*24287.161*	*4.509*
5500	*750*	*21.4/26*	*24055.284*	*2.857*
6000	*1250*	*20.0/26*	*23545.868*	*3.581*
6000	*1750*	*20.3/26*	*23379.277*	*3.963*
6000	*2250*	*19.45/26*	*22919.881*	*4.482*
6000	*3500*	*19.25/26*	*22858.833*	*5.909*
6000	*3750*	*19.0/26*	*22814.971*	*6.313*

3.2 Statistiques pour un texte de 1500 caractères

TABLE 3.5 – BI-GRAMME 1500

Iteration GLOBALE	Iteration LOCAL	Cle	SCORE	TIME
3000	*2250*	*24.65/26*	*133266.747*	*10.060*
3500	*1250*	*25.1/26*	*133300.840*	*9.828*
3500	*2000*	*25.9/26*	*133616.306*	*11.165*
3500	*3250*	*24.05/26*	*132928.374*	*11.732*
4000	*1250*	*24.2/26*	*133208.932*	*9.215*
4000	*2500*	*24.7/26*	*133167.713*	*12.979*
4000	*3750*	*24.05/26*	*133059.642*	*13.400*
4500	*1000*	*24.5/26*	*133120.530*	*8.422*
4500	*1500*	*24.6/26*	*133279.965*	*10.705*
4500	*1750*	*24.5/26*	*133048.501*	*11.836*
4500	*2500*	*24.35/26*	*133162.651*	*14.107*
5000	*2250*	*24.5/26*	*133103.568*	*13.293*
5000	*3000*	*26.0/26*	*133639.957*	*15.388*
5000	*3750*	*24.9/26*	*133355.754*	*16.541*
5500	*1750*	*24.6/26*	*133133.653*	*11.857*
5500	*2750*	*24.3/26*	*132975.793*	*15.081*
5500	*3000*	*25.5/26*	*133459.833*	*16.222*
5500	*4000*	*24.75/26*	*133220.516*	*18.126*
6000	*1000*	*24.75/26*	*133227.736*	*9.433*
6000	*1250*	*24.55/26*	*133180.663*	*10.377*
6000	*1500*	*25.4/26*	*133392.808*	*10.188*
6000	*1750*	*24.3/26*	*133159.829*	*11.636*
6000	*2500*	*25.65/26*	*133482.930*	*13.475*
6000	*3750*	*25.2/26*	*133324.888*	*18.083*
6500	*2500*	*24.35/26*	*133144.493*	*14.110*
6500	*3000*	*24.45/26*	*133068.335*	*16.312*
6500	*3500*	*24.15/26*	*132977.012*	*17.193*
7000	*1000*	*24.9/26*	*133283.122*	*8.722*
7000	*2750*	*24.15/26*	*133093.183*	*15.329*
7000	*3000*	*25.15/26*	*133302.686*	*16.248*
7500	*1000*	*24.05/26*	*133081.095*	*9.493*
7500	*1250*	*24.75/26*	*133166.217*	*9.594*
7500	*2000*	*24.2/26*	*133150.237*	*12.862*
7500	*2250*	*24.0/26*	*133051.330*	*12.655*

TABLE 3.6 – TRI-GRAMME 1500

Iterations Globales	Iterations Locales	Clé	Score	Temps
2000	*750*	*24.7/26*	*114374.509*	*6.855*
2000	*1000*	*24.7/26*	*114361.450*	*6.992*
2000	*1250*	*24.85/26*	*114776.080*	*7.016*
2000	*1750*	*24.0/26*	*113874.046*	*7.035*
3000	*2500*	*25.65/26*	*115274.071*	*10.528*
3500	*750*	*24.7/26*	*114824.535*	*7.532*
3500	*3000*	*24.9/26*	*114845.925*	*12.281*
4000	*1250*	*24.25/26*	*114336.447*	*9.384*
4000	*3000*	*24.85/26*	*114835.706*	*14.016*
4000	*3500*	*24.8/26*	*114895.254*	*14.016*
4500	*500*	*24.65/26*	*114261.730*	*8.018*
4500	*1000*	*24.8/26*	*114902.984*	*8.851*
4500	*2750*	*24.9/26*	*114883.472*	*14.558*
4500	*3000*	*24.25/26*	*114329.080*	*15.270*
5000	*750*	*25.25/26*	*114882.292*	*9.039*
5000	*1250*	*24.7/26*	*114726.016*	*10.027*
5000	*2250*	*24.9/26*	*114866.893*	*12.885*
5000	*4000*	*24.0/26*	*114171.043*	*17.534*
5500	*3000*	*25.45/26*	*115125.927*	*15.965*
6000	*750*	*24.3/26*	*114300.160*	*8.224*
6000	*3000*	*24.0/26*	*114099.985*	*16.967*
6500	*500*	*25.7/26*	*115547.727*	*6.747*
6500	*1500*	*24.8/26*	*114923.019*	*10.708*
6500	*2750*	*24.85/26*	*114923.441*	*15.275*
6500	*3750*	*24.75/26*	*114410.211*	*19.444*
7000	*1500*	*24.8/26*	*114904.337*	*10.579*
7000	*2250*	*24.8/26*	*114880.958*	*13.877*
7000	*2500*	*25.9/26*	*115607.459*	*14.349*
7500	*500*	*24.05/26*	*114320.316*	*6.711*
7500	*750*	*24.3/26*	*114262.740*	*8.379*
7500	*2000*	*24.65/26*	*114809.369*	*12.530*
8000	*750*	*24.55/26*	*114421.175*	*7.901*
8000	*1250*	*24.9/26*	*114859.097*	*10.076*
8000	*2750*	*26.0/26*	*115724.936*	*16.058*

Iterations Globales	Iterations Locales	Clé	Score	Temps
8000	*3000*	*24.8/26*	*114929.687*	*16.560*
8000	*3500*	*24.5/26*	*114351.949*	*17.710*
8000	*3750*	*26.0/26*	*115724.936*	*18.356*
8500	*1750*	*25.05/26*	*114909.573*	*11.438*
8500	*2750*	*25.4/26*	*115129.601*	*15.922*
8500	*3000*	*24.75/26*	*114873.000*	*16.385*
9000	*750*	*24.45/26*	*114350.001*	*7.471*
9000	*1250*	*24.6/26*	*114386.287*	*10.019*
9000	*1750*	*24.85/26*	*114924.414*	*11.936*
9500	*1000*	*24.85/26*	*114923.441*	*8.917*
9500	*3500*	*25.5/26*	*115144.665*	*17.931*
9500	*4000*	*24.9/26*	*114926.432*	*19.728*
10000	*500*	*25.9/26*	*115539.106*	*7.052*
10000	*1750*	*24.75/26*	*114638.554*	*12.600*
10000	*2000*	*24.9/26*	*114901.210*	*12.667*
10000	*2750*	*25.05/26*	*114948.405*	*15.896*
10000	*3500*	*24.8/26*	*114886.000*	*18.142*

Table 3.7 – TETRA-GRAMME 1500

Iterations Globales	Iterations Locales	Clé	Score	Temps
2000	*750*	*24.05/26*	*96430.937*	*6.870*
2000	*1750*	*23.2/26*	*95652.827*	*7.201*
3500	*500*	*23.25/26*	*95955.489*	*6.565*
3500	*750*	*24.3/26*	*97170.754*	*7.374*
3500	*1500*	*23.8/26*	*96853.294*	*10.663*
3500	*2000*	*23.65/26*	*95907.934*	*11.972*
3500	*2250*	*24.65/26*	*97953.650*	*12.395*
4000	*1000*	*23.45/26*	*96134.212*	*8.980*
4000	*1250*	*23.25/26*	*95467.288*	*9.822*
4000	*3000*	*23.65/26*	*96875.083*	*14.317*
4000	*3250*	*24.85/26*	*98445.002*	*14.349*
4000	*3500*	*23.85/26*	*95833.176*	*14.367*
4500	*1000*	*23.6/26*	*96807.215*	*9.330*
4500	*2250*	*23.95/26*	*96942.420*	*13.989*
4500	*2500*	*23.8/26*	*96890.629*	*14.440*
5000	*500*	*25.5/26*	*98408.296*	*6.757*
5000	*1500*	*24.95/26*	*98406.938*	*13.401*
5000	*2000*	*23.6/26*	*96902.603*	*12.948*
5000	*3500*	*26.0/26*	*99965.502*	*17.141*
5000	*3750*	*23.7/26*	*96815.774*	*17.561*
5000	*4000*	*26.0/26*	*99965.502*	*17.891*
5500	*1750*	*23.0/26*	*95660.905*	*13.259*
5500	*2750*	*23.8/26*	*96858.062*	*15.499*
6000	*1250*	*23.65/26*	*96822.269*	*10.558*
6000	*1500*	*23.9/26*	*97054.516*	*11.101*
6000	*2500*	*23.8/26*	*96931.351*	*15.155*
6000	*2750*	*24.8/26*	*98372.891*	*16.156*
6000	*3250*	*24.8/26*	*98449.887*	*16.885*
7000	*750*	*24.95/26*	*98223.349*	*9.035*
7000	*1000*	*24.8/26*	*98312.759*	*9.620*
7000	*2500*	*23.6/26*	*96817.806*	*15.253*
7000	*2750*	*23.7/26*	*96825.390*	*16.523*
7000	*3250*	*23.0/26*	*95552.853*	*17.691*
7000	*3750*	*23.75/26*	*96809.175*	*19.955*

Iteration GLOBALE	Iteration LOCAL	Cle	SCORE	TIME
7500	*1250*	*23.7/26*	*96743.753*	*9.367*
7500	*2000*	*23.9/26*	*96834.850*	*14.199*
7500	*2250*	*23.9/26*	*96856.811*	*13.524*
7500	*2500*	*23.7/26*	*96862.701*	*14.795*
7500	*2750*	*23.85/26*	*96938.281*	*16.322*
7500	*4000*	*23.85/26*	*96897.056*	*20.623*
8000	*2000*	*23.9/26*	*96868.920*	*12.376*
8000	*2250*	*24.9/26*	*98434.445*	*13.764*
8000	*2500*	*26.0/26*	*99965.502*	*15.143*
8000	*3500*	*23.6/26*	*96943.944*	*18.688*
8000	*3750*	*24.95/26*	*98421.758*	*19.375*
8000	*4000*	*23.8/26*	*96898.536*	*20.417*
8500	*500*	*23.2/26*	*95884.209*	*7.468*
8500	*1250*	*23.55/26*	*96818.229*	*9.603*
8500	*1500*	*23.55/26*	*95791.719*	*11.108*
8500	*1750*	*24.25/26*	*97130.058*	*12.293*
9000	*500*	*23.35/26*	*96275.730*	*6.855*
9000	*750*	*24.7/26*	*98121.565*	*8.061*
9000	*1250*	*24.85/26*	*98370.873*	*10.494*
9000	*2750*	*23.6/26*	*96892.203*	*15.355*
9000	*3000*	*23.3/26*	*95750.774*	*16.800*
9000	*3250*	*23.95/26*	*96838.814*	*16.649*
9000	*3750*	*23.6/26*	*96917.096*	*18.629*
9000	*4000*	*24.9/26*	*98396.636*	*19.389*
10000	*1000*	*23.95/26*	*96853.554*	*10.237*
10000	*1250*	*23.05/26*	*95683.559*	*10.631*
10000	*1500*	*23.15/26*	*95621.730*	*11.264*
10000	*1750*	*24.55/26*	*97263.092*	*12.579*
10000	*3500*	*23.85/26*	*96852.550*	*18.256*

Table 3.8 – PENTA-GRAMME 1500

Iteration GLOBALE	Iteration LOCAL	Cle	SCORE	TIME
3000	*1250*	*23.45/26*	*81105.214*	*10.574*
3500	*1250*	*23.6/26*	*81388.341*	*11.025*
4000	*3000*	*24.8/26*	*83660.521*	*15.682*
4500	*2500*	*24.6/26*	*82713.218*	*16.528*
4500	*3250*	*24.75/26*	*83648.834*	*17.131*
5000	*750*	*23.45/26*	*81251.824*	*8.817*
5500	*2750*	*23.8/26*	*81392.992*	*17.462*
5500	*3750*	*24.7/26*	*83623.298*	*20.430*
6500	*3500*	*24.8/26*	*83653.384*	*20.823*
7000	*750*	*23.75/26*	*80862.902*	*8.802*
7500	*2250*	*23.6/26*	*81300.276*	*16.444*
7500	*3750*	*23.7/26*	*81330.953*	*22.243*
8000	*2250*	*24.9/26*	*83645.426*	*15.623*
8000	*2750*	*23.9/26*	*81272.716*	*17.462*
8500	*750*	*23.35/26*	*80424.116*	*9.444*
9500	*1250*	*23.65/26*	*81332.174*	*11.854*
9500	*2000*	*23.6/26*	*81191.380*	*14.141*
10000	*3750*	*23.7/26*	*81283.441*	*22.304*
10000	*4000*	*25.0/26*	*83662.785*	*21.764*

3.3 Tableau de résultats des tests de combinaisons de fitness

Table 3.9 – COMBINAISONS

combinaison	itérations globales	itérations statiques	moyenne lettres 1	moyenne lettres 2	moyenne score 1	moyenne score 2	Temps
(2,2)	1500	(750,500)	22.83/26	24.0/26	40115	40115	2.406
(2,3)	1500	(750,500)	20.3/26	25.1/26	40112	34685	2.688
(2,4)	1500	(1000,1000)	22.25/26	25.17/26	40121	30128	3.951
(2,5)	1000	(750,750)	18.8/26	25.2/26	40105	25941	2.603
(3,3)	1500	(1000,750)	23.8/26	25.2/26	34714	34714	3.134
(3,3)	1000	(1000,1000)	9.8/26	12.0/26	31963	31963	2.573
(3,4)	1500	(1250,750)	20.25/26	23.33/26	34355	29380	3.538
(3,5)	1500	(1250,750)	23.2/26	24.4/26	34714	34714	3.771
(4,4)	1500	(1000,1000)	23.8/26	24.8/26	30128	30128	3.507
(4,5)	3500	(1500,1500)	26.0/26	25.6/26	30128	26373	5.434
(4,5)	3500	(1500,1500)	26.0/26	25.6/26	30128	26373	5.434
(5,5)	2000	(1750,1750)	24.0/26	25.2/26	26373	26373	5.591
(5,5)	2000	(1500,1500)	15.8/26	15.2/26	20776	20776	5.520

3.4 Statistiques tri-gramme puis contrôle avec Pearson

TABLE 3.10 – Combinaison de TRI-GRAMME et Pearson

Iteration GLOBALE	Iteration LOCAL	Cle	SCORE	TIME	Person
2000	*1000*	*18.8/26*	*33465.164*	*2.116*	*0.96868*
2000	*1250*	*19.85/26*	*33721.920*	*2.150*	*0.96868*
2000	*1500*	*22.75/26*	*34131.228*	*2.134*	*0.96868*
2000	*1750*	*21.6/26*	*33884.048*	*2.148*	*0.96868*
2500	*1000*	*22.45/26*	*34225.686*	*2.554*	*0.96868*
2500	*1250*	*21.5/26*	*34026.762*	*2.647*	*0.96868*
2500	*1500*	*20.2/26*	*33754.986*	*2.666*	*0.75072*
2500	*1750*	*21.55/26*	*34013.159*	*2.685*	*0.96868*
2500	*2000*	*19.6/26*	*33570.603*	*2.672*	*0.96868*
2500	*2250*	*21.75/26*	*34046.778*	*2.695*	*0.96868*
3000	*1000*	*18.05/26*	*33268.642*	*2.686*	*0.76986*
3000	*1250*	*15.85/26*	*32918.388*	*3.018*	*0.96868*
3000	*1500*	*19.3/26*	*33617.169*	*3.145*	*0.96868*
3000	*1750*	*16.65/26*	*32921.674*	*3.132*	*0.96868*
3000	*2000*	*19.6/26*	*33509.081*	*3.180*	*0.96868*
3000	*2250*	*20.2/26*	*33767.630*	*3.201*	*0.96590*
3000	*2500*	*20.65/26*	*33826.663*	*3.226*	*0.72097*
3500	*1000*	*19.2/26*	*33556.938*	*2.643*	*0.96868*
3500	*1250*	*21.3/26*	*34026.240*	*3.163*	*0.96868*
3500	*1500*	*20.85/26*	*33792.278*	*3.212*	*0.96868*
3500	*1750*	*17.45/26*	*33153.854*	*3.313*	*0.96868*
3500	*2000*	*21.45/26*	*34038.924*	*3.578*	*0.96868*
3500	*2250*	*22.65/26*	*34231.483*	*3.698*	*0.96868*
3500	*2500*	*19.9/26*	*33589.459*	*3.730*	*0.76696*
4000	*1000*	*20.6/26*	*33799.794*	*2.974*	*0.96868*
4000	*1250*	*21.45/26*	*34048.651*	*3.075*	*0.96868*
4000	*1500*	*18.4/26*	*33372.827*	*3.481*	*0.96868*
4000	*1750*	*18.15/26*	*33368.381*	*3.727*	*0.77374*
4000	*2000*	*22.25/26*	*34112.004*	*3.873*	*0.96868*
4000	*2250*	*20.1/26*	*33827.398*	*4.032*	*0.96868*
4000	*2500*	*21.15/26*	*34058.471*	*4.174*	*0.96868*

Bibliographie

[1] Histoire de la cryptologie https://fr.wikipedia.org/wiki/Histoire_de_la_cryptologie

[2] Chiffre de Vigenère https://fr.wikipedia.org/wiki/Chiffre_de_Vigenère

[3] Chiffrement par substitution mono-alphabétique https://www.bibmath.net/crypto/index.php?action=affiche&quoi=substi/defsub

[4] Chiffrement de César https://fr.wikipedia.org/wiki/Chiffrement_par_décalage

[5] Understanding Hill Climbing Algorithm https://www.apprendre-en-ligne.net/crypto/bibliotheque/index3.html

[6] David KAHN, The Codebreakers : A Comprehensive History of Secret Communication from Ancient Times to the Internet, Revised and Updated, New York, Scribner, 1996

[7] Calculate the Pearson Correlation Coefficient in Python https://datagy.io/python-pearson-correlation/

[8] Stats for letters frequency http://practicalcryptography.com/cryptanalysis/letter-frequencies-various-languages/