Projet LU2IN013

Encadrante : Valérie Ménissier-Morain

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

---

CHEMALI Maïssa, DAHER Sarah, SOUAIBY Christina

Juin 2023

Table des matières

Introduction

La cryptologie, du grec κρυπτός (*kruptos*, secret) et λόγος (*logos*, science), est une science mathématique traitant des messages secrets. Elle regroupe deux domaines : la cryptographie et la cryptanalyse. La cryptographie est l'étude de l'ensemble des techniques de chiffrement d'un texte, donc de transformation d'un texte clair en un texte inintelligible au premier abord dans le but de le rendre incompréhensible pour toute personne autre que l'auteur ou le destinataire. Déchiffrer un texte c'est passer du texte codé au texte original par le moyen d'une clef de déchiffrement, correspondant à la transformation inverse de celle effectuée pour le chiffrer. La cryptanalyse est l'étude et l'application de l'ensemble des techniques dont le but est de retrouver le texte d'origine à partir du chiffré sans disposer de la clef. [1]

La cryptanalyse est un domaine qui remonte à l'Antiquité. Au fil des siècles, de nombreux systèmes cryptographiques ont été développés pour protéger les informations sensibles, telles que les secrets militaires, les affaires privées et plus récemment les données personnelles. Mais à mesure que la technologie évolue, ces cryptosystèmes évoluent également, ce qui rend leur cryptanalyse plus complexe. D'un autre côté, la cryptanalyse peut à présent s'appuyer sur une puissance de calcul bien plus importante qu'à l'époque où tous les calculs devaient être faits à la main.

En cryptographie, un chiffrement par substitution mono-alphabétique est une méthode de chiffrement où chaque lettre d'un texte est remplacée par une autre selon deux conditions : deux lettres différentes sont codées de manière distincte et la même lettre est toujours codée de la même façon. Dans ce cadre, une solution est une permutation de l'alphabet, et la solution optimale est la clef de déchiffrement. Pour cryptanalyser ce type de chiffrement, de nombreuses méthodes automatisées ont été développées, combinant des algorithmes d'exploration aléatoire de l'espace des solutions avec une fonction d'évaluation de la qualité de la solution.

Parmi ces méthodes figurent notamment le *Hill Climbing*, le Recuit Simulé et les Algorithmes Génétiques. Par la suite, nous développerons certaines de ces méthodes de cryptanalyse afin d'en tirer les points forts et les points faibles de chacune selon différentes implémentations.

Hill Climbing

2.1 Définition

L'algorithme de *Hill Climbing* est une méthode d'optimisation mathématique : son but est de trouver la meilleure solution locale pour un problème donné en partant d'une solution aléatoire.
Le principe de cet algorithme est le suivant ([2], [3]) :

- On part d'une certaine solution initiale arbitraire.
- On modifie légèrement cette solution en solution voisine en effectuant des modifications mineures sur celle-ci.
  - si cette nouvelle solution est meilleure que l'ancienne alors on la garde,
  - sinon on garde l'ancienne.
- On recommence à partir de la solution obtenue tant qu'on n'a pas atteint une condition d'arrêt.

L'algorithme se termine lorsqu'on atteint un nombre prédéfini d'itérations, ou si on obtient la même solution durant un certain nombre d'itérations consécutives. En effet, l'algorithme de *Hill Climbing* cherche à atteindre un maximum global, mais il est susceptible de rencontrer un maximum local : si toutes les solutions voisines sont moins satisfaisantes que la solution actuelle, l'algorithme restera bloqué sur celle-ci et ne pourra plus progresser vers une meilleure solution même si celle-ci existe.

L'algorithme de *Hill Climbing* peut être utilisé pour résoudre une variété de problèmes d'optimisation, y compris la recherche de la meilleure stratégie pour un jeu et l'optimisation de fonctions mathématiques complexes.

Dans le cadre de ce projet, nous utilisons l'algorithme de *Hill Climbing* pour déchiffrer un texte chiffré par une substitution mono-alphabétique. Nous cherchons des solutions voisines en choisissant deux lettres différentes de l'alphabet et en échangeant toutes leurs occurrences dans le message déchiffré par la solution actuelle. La qualité de chaque solution correspond à sa vraisemblance dans la langue française.

2.2 Implémentation

Pour implémenter l'algorithme, nous avons procédé par étapes :

Étape 1 : Nous avons établi des statistiques caractéristiques de la langue nécessaires pour estimer la vraisemblance d'une solution dans la langue étudiée. Ainsi, nous avons d'abord entrepris une analyse d'un texte représentatif de la langue française. Pour cela, nous avons choisi l'œuvre de Victor Hugo intitulée "Les Misérables" comme référence. Après avoir effectué une étape préliminaire de traitement de l'œuvre, comprenant la suppression de la ponctuation, des accents et la mise en majuscules de l'ensemble du texte, nous avons créé des dictionnaires regroupant les nombres de monogrammes, bigrammes, trigrammes et tétragrammes. Ces dictionnaires constituent les références statistiques sur lesquelles nous nous appuierons lors du processus de cryptanalyse ultérieur.

Étape 2 : Nous avons codé des fonctions d'évaluation permettant d'évaluer le texte en lui attribuant un score de conformité aux caractéristiques de la langue qui peut être calculé par différentes méthodes, parmi lesquelles les `n`-grammes et la corrélation de Pearson qui seront détaillées par la suite.

Étape 3 : Enfin, nous avons implémenté l'algorithme de *Hill Climbing* en suivant les étapes décrites plus haut. Notre fonction prend en paramètres le texte à déchiffrer, la fonction d'évaluation choisie avec ses propres paramètres, ainsi que diverses conditions d'arrêt (nombre maximal d'itérations, borne maximale de stagnations, score de satisfaction, etc).

Remarque. La borne maximale de stagnations représente le nombre maximal d'itérations consécutives sans qu'il n'y ait d'amélioration du score. Le score de satisfaction est un seuil qui, une fois atteint, indique que la solution est considérée acceptable et permet de mettre fin à la boucle.

2.3 Analyse

Nous avons procédé à un grand nombre de tests sur des textes différents et de toutes tailles afin de comprendre quels paramètres permettaient d'obtenir de meilleurs résultats, ou au contraire de moins bons. Nous en avons obtenu ces conclusions :

— Exécuter deux fois *Hill Climbing*, en partant la seconde fois de la clef trouvée la première fois, n'améliore pas le rendement : en effet, si on est sorti de *Hill Climbing* c'est qu'on a déjà stagné un grand nombre de fois `k`, il n'est donc pas pertinent de retenter.

— Le résultat est meilleur quand la borne maximale de stagnations `k` (passée en paramètre) est aux alentours de `200`. En effet, si elle est trop petite il est probable qu'une meilleure solution atteignable existe mais ne soit pas atteinte ; d'un autre côté, si elle est trop grande le programme prendra plus de temps sans pour autant atteindre une meilleure solution (il y a seulement $\binom{26}{2}$`=325` permutations possibles). Nous avons trouvé que `k=250` est optimal pour détecter la présence d'un minimum local et donc ne plus continuer inutilement.

— Le résultat est meilleur quand le clair du texte étudié est représentatif de la langue française (par exemple, si le texte en clair possède très peu de `E`, l'algorithme aura plus de mal à trouver la bonne solution).

— Le paramètre de satisfaction est inutile : en effet, il est plus intéressant que l'arrêt de la fonction soit causé par la détection d'un minimum local que par autre chose.

— Pour la même raison que le point précédent, le nombre maximal d'itérations n'est pas un bon cas d'arrêt. Nous le fixerons donc assez haut, à `2000`, pour qu'il ne soit que rarement atteint, et qu'ainsi la sortie de l'algorithme soit davantage provoquée par la détection d'un minimum local.

Recuit Simulé

3.1 Définition et Implémentation

Le recuit simulé est une méthode d'optimisation inspirée du processus de refroidissement d'un matériau, lui permettant d'atteindre un état de plus basse énergie. Pour implémenter cette méthode, on part d'un état initial d'un système et on le modifie pour en atteindre un nouveau. Ce nouvel état peut soit améliorer le score, ce qui nous rapproche de l'optimum en explorant le voisinage de l'état initial, soit le dégrader ce qui nous permet d'explorer une plus grande zone d'états, évitant de nous renfermer rapidement dans la recherche d'un optimum local [4] (contrairement à *Hill Climbing*).

Le principe général de cet algorithme est le suivant [5] :

- On part d'une certaine solution.
- On modifie légèrement cette dernière en une solution voisine.
  - — si cette solution voisine est meilleure que l'ancienne alors on la garde,
  - — sinon, on génère un nombre aléatoire `r` compris entre `0` et `1` qu'on compare avec

$$\exp\left(\frac{\texttt{score\_solution\_courante - score\_solution\_optimale}}{\texttt{T}}\right)$$

    où `T` est la température courante, initialement égale à $\texttt{T}_{\texttt{init}}$ donnée en paramètre.
    Si `r` est inférieur à ce nombre, on garde la nouvelle solution même si elle est moins satisfaisante que la solution précédente. Sinon, on ignore cette solution et on garde l'ancienne.
- Après un nombre de paliers bien défini passé en paramètre, on multiplie `T` par un coefficient de décroissance `c` (compris entre `0` et `1` exclus) transmis en paramètre afin de faire diminuer `T`.
- On répète ces étapes avec la solution obtenue tant que la température courante `T` n'a pas atteint une certaine température minimale $\texttt{T}_{\texttt{min}}$ transmise en paramètre.

Des optimisations de l'algorithme sont possibles. Dans notre implémentation, nous avons fait le choix de garder en mémoire la meilleure solution rencontrée. Ainsi, au moment de retourner la solution, si la solution optimisée sélectionnée par l'algorithme est moins bonne qu'une solution précédemment rencontrée, on renverra à la place cette meilleure solution. Cela permet d'explorer un ensemble de solutions variées, tout en gardant trace de la meilleure rencontrée dans le cas où on se serait égaré dans un mauvais optimum local.

L'implémentation est similaire à celle du *Hill Climbing*. En effet, les deux premières étapes, concernant les statistiques de la langue et les fonctions d'évaluation, sont communes. Ensuite, l'implémentation de l'algorithme suit les étapes listées ci-dessus. Les paramètres de notre fonction sont le texte, la fonction score permettant d'évaluer la vraisemblance des solutions, la température initiale $\texttt{T}_{\texttt{init}}$, la température minimale $\texttt{T}_{\texttt{min}}$ servant de condition d'arrêt, le coefficient de décroissance `c` permettant de diminuer progressivement la température courante, le nombre d'itérations par palier `nb_iter_par_palier` et la clef de chiffrement.

3.2 Analyse

Comme nous l'avons fait pour l'algorithme de *Hill Climbing*, nous avons entrepris une série exhaustive de tests sur des textes variés et de tailles différentes afin d'identifier les paramètres clés qui garantissent les meilleurs résultats et ceux qui produisent des performances moins satisfaisantes. Ce processus s'est avéré fastidieux en raison du nombre considérable de variables à prendre en compte. Nous avons pu en extraire des conclusions significatives :

— Le résultat est meilleur quand la température initiale $\texttt{T}_{\texttt{init}}$ passée en argument est grande. En effet, l'algorithme est alors itéré un nombre important de fois.
Précisément, le nombre d'itérations est compris entre

$$\texttt{nb\_iter\_par\_palier} \times \frac{\ln(\texttt{T}_{\texttt{init}})}{\ln(\texttt{c})} \text{ et } \texttt{nb\_iter\_par\_palier} \times \left(\frac{\ln(\texttt{T}_{\texttt{init}})}{\ln(\texttt{c})} + 1 \right).$$

Cependant, une température initiale trop élevée augmenterait le temps de traitement, il nous faut donc trouver un compromis entre la qualité et l'efficacité de l'algorithme. Pour cela, nous avons opté pour l'initialisation de la température à $\texttt{T}_{\texttt{init}}$ `= 1000`.

— Modifier la température minimale $\texttt{T}_{\texttt{min}}$ n'est pas nécessaire car la maîtrise de la température initiale $\texttt{T}_{\texttt{init}}$ est suffisante : en effet, si on souhaite augmenter la température minimale, cela revient à diminuer la température initiale, et inversement. Ainsi, en fixant la température minimale à $\texttt{T}_{\texttt{min}}$`= 1`, nous garantissons l'obtention de statistiques cohérentes et satisfaisantes.

— Le choix du coefficient de décroissance `c` est optimal s'il est compris entre `0.8` et `0.95`. En effet, il permet dans ce cas une baisse de la température courante `T` qui n'est ni trop rapide (auquel cas on explorerait un ensemble de solutions plus réduit) ni trop lente (sinon on réaliserait une recherche trop coûteuse en temps). Nous fixerons `c` à `0.8` dans ce projet.

— Un palier représente une étape de l'algorithme dans laquelle l'exploration des clefs a lieu à une température bien spécifique (constante). Une augmentation du nombre d'itérations par palier accroît les chances de l'algorithme de trouver une solution plus pertinente. Cependant, un nombre d'itérations par palier trop important ferait également augmenter le temps de calcul nécessaire pour obtenir une solution.
De ce fait, nous avons fixé le nombre d'itérations par palier à 100 afin de générer une solution satisfaisante dans un délai raisonnable.

— Remarquons que plus la température courante `T` est grande, plus la probabilité d'accepter une moins bonne solution est conséquente. Ainsi, au début de l'algorithme on se permet une grande marge d'acceptation qui diminue ensuite progressivement. De même, on observe que moins une solution est bonne, moins elle a de chance d'être gardée.

— Si $\texttt{T}_{\texttt{init}}$ est initialisée à `0`, on se ramène au cas de *Hill Climbing* : seulement les solutions améliorant le score sont acceptées.

Notons néanmoins que d'autres configurations des paramètres initiaux sont possibles et tout autant pertinentes que celles proposées ci-dessus. Par exemple, augmenter le nombre d'itérations par palier peut correspondre à une augmentation de la température initiale $\texttt{T}_{\texttt{init}}$ et du coefficient de décroissance `c`, puisque dans les deux cas le nombre d'itérations augmente. Ainsi, en jouant avec les initialisations possibles des paramètres $\texttt{T}_{\texttt{init}}$, `nb_iter_par_palier`, et `c`, on peut créer une multitude d'initialisations convenables.

Fonctions scores

Une fonction *fitness* est une fonction utilisée dans des algorithmes d'optimisation pour évaluer la qualité d'une solution.
Dans des algorithmes de cryptanalyse comme par exemple le *Hill Climbing* et le recuit simulé, il est primordial de connaître la pertinence d'une solution pour identifier la meilleure direction à suivre. La qualité d'une solution est déterminée par sa similarité à la langue française. Pour évaluer si une solution est meilleure qu'une autre, on attribue à chaque texte un score. Un score plus élevé montre une ressemblance plus importante entre le texte proposé et un texte référence de la langue française. Les fonctions d'évaluation, autrement nommées fonctions scores ou fonction *fitness*, utilisent de différentes manières les statistiques de référence de la langue pour calculer ce score. Dans le cadre de ce projet, nous avons implémenté deux de ces fonctions : les `n`-grammes et la corrélation de Pearson.

4.1 Méthode des `n`-grammes

Un `n`-gramme est une combinaison de `n` lettres de l'alphabet avec `n` un entier naturel non nul. On distingue les bigrammes qui sont des séquences de deux lettres, les trigrammes qui sont des séquences de trois lettres et les tétragrammes qui sont des séquences de quatre lettres.

4.1.1 Principe

La langue française peut être repérée par la répétition régulière de certaines combinaisons de lettres comme `QUE`, `OU`, `MENT`, `ETTE` etc. Cette méthode se base ainsi sur la comparaison des fréquences de ces `n`-grammes dans un texte de référence de la langue française à celles des combinaisons retrouvées dans le texte à évaluer. Le score d'un `n`-gramme est donné par sa fréquence dans la langue française. Concrètement, on peut remarquer que si un trigramme `ABC` est aussi fréquent dans la langue qu'un trigramme `XYZ` dans la solution, il est probable que `XYZ` corresponde au chiffré de `ABC`. On peut définir le score d'un mot par le produit des scores des `n`-grammes qui le forment, à `n` fixé. [6]

Voici comment procéder pour évaluer la phrase `CHUTCESTUNSECRET` avec `n=3` :
`Score(CHUTCESTUNSECRET)` = `Score(CHU)` × `Score(HUT)` × `Score(UTC)` × `Score(TCE)` × `Score(CES)` × `Score(EST)` × `Score(STU)` × `Score(TUN)` × `Score(UNS)` × `Score(NSE)` × `Score(SEC)` × `Score(ECR)` × `Score(CRE)` × `Score(RET)`
En effet le parcours de la chaîne de caractères se fait par glissement, comme suit :

4.1.2 Implémentation

Le nombre de `n`-grammes distincts obtenus dans un texte est donné par la formule : `len(texte)-n+1` (pour `len(CHUTCESTUNSECRET)=16` et `n=3`, on a `16-3+1 = 14` trigrammes).

Ce qui nous mène à la formule générale de calcul du score d'un texte selon les `n`-grammes :

$$\prod_{i=0}^{\texttt{len(texte)-n}} \texttt{fréquences(texte[i:i+n])}$$

Dans le but d'améliorer la précision, d'éviter les problèmes de dépassement et d'assurer une meilleure stabilité numérique lors des calculs sur ordinateur, on préfère utiliser les occurrences plutôt que les fréquences et passer à un ordre logarithmique. Le produit est ainsi transformé en somme, ce qui donne à l'implémentation la formule suivante :

$$\sum_{i=0}^{\texttt{len(texte)-n}} \log(\texttt{occurrences(texte[i:i+n])})$$

4.1.3 Résultats

Après avoir intégré la fonction d'évaluation des `n`-grammes dans nos algorithmes de cryptanalyse et réalisé une série de tests approfondis sur des textes de différentes catégories et tailles, chiffrés avec diverses clés et en variant la taille des `n`-grammes, il est légitime d'affirmer que cette fonction donne globalement d'assez bons résultats.

Nous pouvons manifestement constater que la performance de la fonction est étroitement liée à la taille du texte utilisé. Cette observation découle de plusieurs facteurs significatifs. En effet, les textes plus grands contiennent une plus grande quantité de données linguistiques, ce qui permet une estimation plus précise des probabilités d'apparition des séquences de caractères. Ainsi, les caractéristiques linguistiques globales, telles que la fréquence des mots, les schémas de structure grammaticale et les relations entre les mots, sont mieux reflétés.

Remarque. Dans les tableaux ci-dessous, *clef_utilisee* et *clef_trouvee* représentent respectivement la clef de chiffrement réellement utilisée et la clef de chiffrement supposée par l'algorithme. Les caractères rouges indiquent les positions correctes dans la clé trouvée. (cf **7.1** pour plus de données quantitatives)

taille	clef_utilisee	clef_trouvee	%_caracteres_egaux	ngram
119	NJKRPUAEDZXCIMOGQHTSVYLFBW	NQUHPYFIVWJRKMOGLCTSDEAXBZ	36.36	3
119	NJKRPUAEDZXCIMOGQHTSVYLFBW	NQUKPGFWVIAMCHOEJRDTSYLXBZ	22.73	3
119	NJKRPUAEDZXCIMOGQHTSVYLFBW	SGRKPYQXNWZCEHOUFVTDMIABLJ	18.18	3
119	NJKRPUAEDZXCIMOGQHTSVYLFBW	DWURPYIGNJLCKMOEQHTSVBAFXZ	50	3
119	NJKRPUAEDZXCIMOGQHTSVYLFBW	DYFHVBGITLARWNKOUSCMPQJXEZ	**0**	3
119	NJKRPUAEDZXCIMOGQHTSVYLFBW	VUCHTFWEKZXPOSDIYBRNMQAGJL	4.55	3
119	NJKRPUAEDZXCIMOGQHTSVYLFBW	MYUWTCAJRFZPBNKIXHDVSOLQEG	4.55	3
119	NJKRPUAEDZXCIMOGQHTSVYLFBW	SEHYPBQWNIJCKVOUFTMDRGLAXZ	13.64	3
119	NJKRPUAEDZXCIMOGQHTSVYLFBW	REWYPGBJNILMCKOHQDSTVUXFAZ	22.73	3
119	NJKRPUAEDZXCIMOGQHTSVYLFBW	NWERPXIGDLZCKMOUQHSTVYAFBJ	**59.09**	3
			23.183	
1199	FBJHNUQIACVEOMPKXDLTYSRWZG	FBJHNUCIAQGEOMPKXDTLYSRVZW	69.23	3
1199	FBJHNUQIACVEOMPKXDLTYSRWZG	FBJHNUQIACREOMPKXDLTYSVWZG	92.31	3
1199	FBJHNUQIACVEOMPKXDLTYSRWZG	FBJHNUQIACREOMPKXDLTYSVWZG	92.31	3
1199	FBJHNUQIACVEOMPKXDLTYSRWZG	FIJHNBQUACREOMPKXDLTYSVWZG	80.77	3
1199	FBJHNUQIACVEOMPKXDLTYSRWZG	FBJHNCQIAGREOMPKXDLTYSVWZU	80.77	3
1199	FBJHNUQIACVEOMPKXDLTYSRWZG	FUEHNSBIACRJOMPKXDLTYQVWZG	69.23	3
1199	FBJHNUQIACVEOMPKXDLTYSRWZG	HOKYLBQWMZRFPJDSIENTAUGXCV	**7.69**	3
1199	FBJHNUQIACVEOMPKXDLTYSRWZG	FBJHNUQIACVEKMPOXDLTYSRWZG	88.46	3
1199	FBJHNUQIACVEOMPKXDLTYSRWZG	FBJHNUQIACREOMPKXDLTYSVWZG	**92.31**	3
1199	FBJHNUQIACVEOMPKXDLTYSRWZG	FBEHNUIQACRDOJPKXMLTYSVWZG	69.23	3
			74.231	

Légende
%MIN
%MAX
%MOYEN

Tableau statistique représentant les résultats obtenus par l'algorithme de Hill Climbing appliqué plusieurs fois sur deux textes de tailles différentes, pour un même ***n****-gramme.*

Quant à la taille des n-grammes, les résultats expérimentaux ont démontré de manière concluante que les trigrammes et les tétragrammes ont présenté une performance supérieure dans le contexte de nos techniques de cryptanalyse. Certes, choisir la taille appropriée des n-grammes constitue un compromis délicat entre deux aspects essentiels. D'une part, les n-grammes de taille plus petite sont plus susceptibles d'être sujets à des coïncidences fortuites qui peuvent fausser les résultats. D'autre part, des n-grammes trop grands, tels que des combinaisons de cinq lettres ou plus, peuvent être moins descriptifs, car ces motifs se répètent moins fréquemment dans le langage naturel.

taille	cle_utilisee	cle_trouvee	%_caracteres_egaux	ngram
1199	FBJHNUQIACVEOMPKXDLTYSRWZG	LBKJNSWUFIVYHMPQXDTEAORCZG	34.62	1
1199	FBJHNUQIACVEOMPKXDLTYSRWZG	LSKHNBIQMWGDOEPJXYTFACRVUZ	19.23	1
1199	FBJHNUQIACVEOMPKXDLTYSRWZG	LIJKNZWBMXGEHDPQSYTFAORCUV	**15.38**	1
1199	FBJHNUQIACVEOMPKXDLTYSRWZG	LIJKNCXBFZRAHDPOQETMYSVWUG	26.92	1
1199	FBJHNUQIACVEOMPKXDLTYSRWZG	LUKHNQCSTVRYOMPJIDAFEXGBWZ	23.08	1
1199	FBJHNUQIACVEOMPKXDLTYSRWZG	LIKHNQUCTXGPJAESBDMFYORWVZ	19.23	1
1199	FBJHNUQIACVEOMPKXDLTYSRWZG	LXJKNQWIACVEODPHBMTFYSRZUG	**46.15**	1
1199	FBJHNUQIACVEOMPKXDLTYSRWZG	LXJHNSICAWGEKMPOBDTFYQRZUV	34.62	1
1199	FBJHNUQIACVEOMPKXDLTYSRWZG	FXHKNBIULWRYJTPOQAMEDSGVCZ	**15.38**	1
1199	FBJHNUQIACVEOMPKXDLTYSRWZG	LWKHNXBCTUVYOFPJSDEAMQRIGZ	23.08	1
			25.769	
1199	FBJHNUQIACVEOMPKXDLTYSRWZG	FBJHNUQIACVEOMPKXDLTYSRWZG	**100**	4
1199	FBJHNUQIACVEOMPKXDLTYSRWZG	FBJHNUQIACREOMPKXDLTYSVWZG	92.31	4
1199	FBJHNUQIACVEOMPKXDLTYSRWZG	FBJHNUQIACVOEMPKXDLTYSRWZG	88.46	4
1199	FBJHNUQIACVEOMPKXDLTYSRWZG	FBJHNUQIACREOMPKXDLTYSVWZG	92.31	4
1199	FBJHNUQIACVEOMPKXDLTYSRWZG	FBJHNUQIACVEOMPKXDLTYSRWZG	**100**	4
1199	FBJHNUQIACVEOMPKXDLTYSRWZG	FBJHNUQIACREOMPKXDLTYSVWZG	92.31	4
1199	FBJHNUQIACVEOMPKXDLTYSRWZG	FUJHNDQIACREOMPKXBLTYSVWZG	**80.77**	4
1199	FBJHNUQIACVEOMPKXDLTYSRWZG	FBJHNSQIACGEOMPKXDLTYUVWZR	**80.77**	4
1199	FBJHNUQIACVEOMPKXDLTYSRWZG	FBJHNUQIACVEOMPKXDLTYSGWZR	92.31	4
1199	FBJHNUQIACVEOMPKXDLTYSRWZG	FBJHNUQIACVEOMPKXDLTYSRWZG	**100**	4
			91.924	

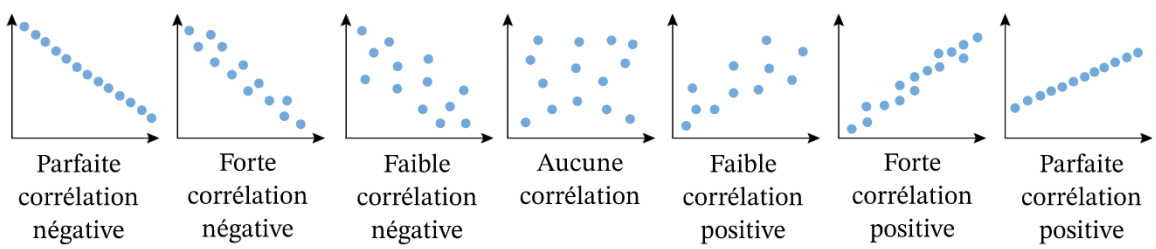
Légende
%MIN
%MAX
%MOYEN

Tableau statistique représentant les résultats obtenus par l'algorithme de Recuit Simulé appliqué plusieurs fois sur un même texte, pour deux n-grammes différents.

4.2 Méthode de la corrélation de Pearson

4.2.1 Principe

Le coefficient de corrélation de Pearson est une mesure statistique utilisée pour évaluer la relation linéaire entre deux variables aléatoires. Sa valeur est comprise entre `-1` et `1`. [7] Dans le contexte de cryptanalyse de textes chiffrés par substitution monoalphabétique, cette méthode est utilisée pour comparer les fréquences d'apparition des lettres dans le texte chiffré avec les fréquences attendues dans la langue. L'idée principale est de trouver une correspondance entre les lettres du texte et les lettres réelles en se basant sur les différences entre les fréquences. Un fort coefficient de corrélation de Pearson (proche de `1` ou `-1`) indique une correspondance probable entre les lettres chiffrées et non chiffrées, ce qui facilite le décryptage. Un coefficient de corrélation proche de `0` indique une mauvaise correspondance et peut rendre la cryptanalyse plus difficile.



4.2.2 Implémentation

Pour calculer le score de corrélation de Pearson, on utilise la formule suivante [7] :

$$\frac{\sum (X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum (X_i - \bar{X})^2 \sum (Y_i - \bar{Y})^2}}$$

où les variables `X` et `Y` représentent respectivement les fréquences des lettres dans le texte chiffré et dans la langue utilisée (en l'occurrence le français). Pour chaque $\texttt{X}_\texttt{i}$ correspondant à la fréquence d'un caractère `c` dans le texte chiffré en cours de traitement, le $\texttt{Y}_\texttt{i}$ associé est la fréquence de `c` dans le texte de référence de la langue utilisée. Par exemple, si `c'` est chiffré en `c`, alors $\texttt{X}_\texttt{i}$ correspond à la fréquence de `c` dans le texte chiffré et $\texttt{Y}_\texttt{i}$ correspond à celle de `c'` dans le texte de référence de la langue française.
Dans l'implémentation, les fréquences des lettres dans le texte chiffré et dans la langue française sont converties en tableaux `numpy` pour faciliter les calculs. Ensuite, les moyennes et les écarts par rapport aux moyennes sont calculés. Finalement, la formule du coefficient de corrélation de Pearson est appliquée pour obtenir le score de corrélation.

4.2.3 Résultats

Nous avons constaté dans notre projet que la méthode de corrélation de Pearson ne donne pas les résultats attendus lorsqu'elle est appliquée aux chiffrements par substitution mono-alphabétique. En fait, il existe certaines limites à la comparaison des distributions de caractères dans ce contexte particulier. L'analyse de la distribution des caractères dans les textes chiffrés indique que plusieurs lettres ont des fréquences voisines en français. Cela signifie que certains textes chiffrés peuvent apparaître aussi fréquemment que d'autres, ce qui peut prêter à confusion. Par exemple, si les lettres `A` et `I` sont rapprochées dans le texte chiffré, il devient difficile de déterminer laquelle correspond à chaque lettre dans le texte original. Cette confusion peut entraîner des erreurs de déchiffrement et le rendre moins fiable.

taille	clef_utilisee	clef_trouvee	%_caracteres_egaux	score_final
743	WKRGODMVBAUIXNCSFYLPJETHZQ	LYCWORGJXKHAMIBPVUZTQFDNES	4.17	0.73859927
743	WKRGODMVBAUIXNCSFYLPJETHZQ	ABLDRFOQGUTCMJHPNESKIVWXYZ	0	0.709749721
743	WKRGODMVBAUIXNCSFYLPJETHZQ	TYCDNAGSBQKLIOWPJHRFUMEXVZ	4.17	0.771041585
743	WKRGODMVBAUIXNCSFYLPJETHZQ	MHYAEIDNLGKOFJBTCZSWRVUQXP	0	0.719364556
743	WKRGODMVBAUIXNCSFYLPJETHZQ	ZAIHBFXMYDKLOWVJNRCPUTQGSE	4.17	0.730500994
743	WKRGODMVBAUIXNCSFYLPJETHZQ	AICBXSKGTPRVLJDZQEFUWMNOYH	0	0.815697535
743	WKRGODMVBAUIXNCSFYLPJETHZQ	LBGHSFNEMJWOIAZRQUVKYPDTCX	0	0.756435704
743	WKRGODMVBAUIXNCSFYLPJETHZQ	WBDNJSKIAYZLQMOVHXUTECGRPF	4.17	0.752553219
743	WKRGODMVBAUIXNCSFYLPJETHZQ	XBCWFTGVZJKIMNOPLRESYHUQDA	12.5	0.748859374
743	WKRGODMVBAUIXNCSFYLPJETHZQ	LATGMOJVQSKICPEZBRWNYFHUXD	12.5	0.807771991
			4.168	

Légende
%MIN
%MAX
%MOYEN

Tableau statistique représentant les résultats obtenus par l'algorithme de Hill Climbing appliqué plusieurs fois sur un même texte de taille moyenne, qui montre de fortes corrélations positives pour une qualité moins satisfaisante.

Conclusion

Durant ce projet, nous avons entrepris une exploration approfondie et une étude attentive de diverses techniques de cryptanalyse spécialement adaptées aux chiffrements par substitution mono-alphabétique. En examinant les différentes métaheuristiques telles que le *Hill Climbing* et le recuit simulé, en conjonction avec l'utilisation de différentes fonctions d'évaluation et une variation judicieuse des paramètres, nous avons abouti à des conclusions pertinentes :

— Dans le cas d'un texte court (moins de `200` caractères), les résultats obtenus ne sont pas concluants quelle que soit la combinaison de métaheuristique et fonction d'évaluation choisie. En revanche, il a été constaté que l'utilisation de l'évaluation basée sur les tétragrammes offre le meilleur arrangement entre le score obtenu et l'efficacité temporelle de l'algorithme.

— En outre, il est valable de soutenir que la fonction d'évaluation de la corrélation de Pearson n'est pas la plus adaptée aux algorithmes implémentés dans notre travail. Effectivement, une forte corrélation n'indique qu'un rapprochement, possiblement accidentel, des fréquences des lettres, ce qui ne garantit en aucun cas leur emplacement correct dans le texte, et créant ainsi de fausses pistes pour des algorithmes qui comparent naïvement les données quantitatives.

— L'algorithme du recuit simulé fournit de meilleurs résultats que l'algorithme de *Hill Climbing* grâce à sa capacité à éviter les maxima locaux et à explorer davantage l'espace de recherche, tout en conservant la meilleure proposition trouvée. Cependant, en ce qui concerne l'efficacité temporelle, l'algorithme de *Hill Climbing* peut être plus rapide en raison de sa convergence rapide vers des maxima locaux contrairement au recuit simulé qui demandera généralement plus d'itérations si nous voulons adopter une solution plus optimale. Le choix de l'algorithme dépendra ainsi des exigences du problème rencontré.

Bibliographie

[1] Ghislaine Labouret. *Introduction à la cryptologie*. 1998. https://www.labouret.net/crypto/

[2] Hein de Haan. *How to Implement the Hill Climbing Algorithm in Python*. 2020. https://towardsdatascience.com/how-to-implement-the-hill-climbing-algorithm-in-python-1c65c29469de

[3] Thomas Kaeding. *Slippery hill-climbing technique for ciphertext-only cryptanalysis of periodic polyalphabetic substitution ciphers*. 2019. Pages 1-4. https://eprint.iacr.org/2020/302.pdf

[4] Bendahmane Amine. *Le recuit simulé, Module : Optimisation avancée*. 2011. https://rfia2012.files.wordpress.com/2011/12/amine_le_recuit_simulc3a9.pdf

[5] Didier Müller. *Les métaheuristiques en cryptanalyse*. 2020. https://www.vsmp.ch/bulletin/suche/Artikel/143/143_M%C3%BCller.pdf

[6] Jeremy Kun. *Cryptanalysis with N-Grams*. 2012. https://jeremykun.com/2012/02/03/cryptanalysis-with-n-grams/

[7] Claude Grasland. *Initiation aux méthodes statistiques en sciences sociales*. Chapitre 6 : La corrélation, 1998. http://grasland.script.univ-paris-diderot.fr/STAT98/stat98_6/stat98_6.htm

Annexes

7.1 Tableaux de statistiques

Les tableaux statistiques ci-dessous résument les performances des deux algorithmes explicités dans ce projet, implémentés selon les différentes fonctions d'évaluation abordées, en prenant en compte un large éventail de paramètres. Il est important de noter que chaque ligne représente un résultat moyen obtenu sur dix tests effectués.

Dans les tableaux ci-dessous, la colonne *nom* indique le nom du fichier contenant le texte chiffré ; la colonne *taille* donne le nombre de caractères du texte correspondant ; la colonne *%caractere_ egaux* représente la moyenne, basée sur dix tests, du pourcentage de similitude entre la clé de chiffrement trouvée par l'algorithme et la clé de chiffrement utilisée pour chiffrer le texte ; la colonne *ngram* indique le `n`-gramme utilisé par la méthode d'évaluation des `n`-grammes ; la colonne *nb_ iter* donne la moyenne, sur dix tests, du nombre d'itérations effectuées par l'algorithme ; la colonne *temps* représente la moyenne du temps nécessaire pour obtenir le résultat ; enfin, la colonne *score_ final* indique la moyenne du score obtenu pour la solution à la fin de l'algorithme.

7.1.1 *Hill Climbing* : Variation des conditions d'arrêt de l'algorithme

L'analyse du tableau ci-dessous montre que l'algorithme requiert un temps important lorsque les nombres maximaux d'itérations (MAX_iter) et de stagnations (MAX_stagn) sont élevés, sans apporter d'améliorations significatives au score. Lorsque le nombre maximal d'itérations est fixé à 2000 et celui de stagnations à 50, les textes obtenus ne ressemblent pas au français ; même conclusion lorsque le nombre maximal d'itérations est égal à 1000. Dans ces deux cas, nous observons fréquemment une sortie de la fonction avec trop peu de stagnations, indiquant qu'on n'a pas atteint un maximum local. En fixant le nombre maximal d'itérations à 2000, nous constatons qu'aucune amélioration significative du score n'est observée, que ce soit pour 200, 250 ou 500 itérations maximales. Par conséquent, il est préférable de sélectionner un nombre maximal de stagnations d'environ 200 pour un nombre maximal d'itérations de 2000, ce qui permet d'optimiser le temps requis par l'algorithme.

taille	%_caract_egaux	ngram	MAX_stagn	Max_iter	nb_iter	temps
570	16	1	4000	10000	4756,9	5,45890234
570	64,89	2	4000	10000	5197,6	6,09864672
570	74	3	4000	10000	4289,8	4,98929478
570	75,6	4	4000	10000	5178,3	5,92357801
570	20	1	2000	7500	2845,89	3,99555737
570	29,3	2	2000	7500	2679,3	3,82021331
570	47,5	3	2000	7500	3489,5	4,15381622
570	62,3	4	2000	7500	3567,9	4,92780871
570	12,7	1	500	5000	1854,3	2,34966075
570	65,3	2	500	5000	1984,2	2,4569312
570	51,4	3	500	5000	2224,3	3,0643892
570	61,5	4	500	5000	2304,3	3,128903
570	16,45	1	50	2000	1342	0,523122205
570	32	2	50	2000	385,45	0,53836725
570	27,2	3	50	2000	386,5	0,615982628
570	40	4	50	2000	411,9	0,461943698
570	18	1	250	2000	883,5	0,995710278
570	71,2	2	250	2000	1398,8	1,93889482
570	66	3	250	2000	1248,5	1,752062154
570	77,2	4	250	2000	1342	2,058465028
570	12	1	200	1000	898,5	0,902775574
570	23,4	2	200	1000	1000	1,2021331
570	22,1	3	200	1000	975,4	1,15381622
570	45,8	4	200	1000	987,2	1,78480914
570	19	1	200	2000	1134,5	0,97755737
570	61,2	2	200	2000	1083,5	1,72021331
570	76	3	200	2000	1288,8	1,821538162
570	79,2	4	200	2000	1308,5	1,96043214
570	20	1	500	2000	1640,889	1,250451
570	63,556	2	500	2000	1846,2	1,927992
570	65	3	500	2000	1895,4	2,437855
570	78,182	4	500	2000	1849,6	2,312863

7.1.2 *Hill Climbing* évalué par la méthode des n-grammes

nom	taille	%_caracteres_egaux	ngram	nb_iter	temps
chiffre_germinal_1_119_1	119	11,365	1	909,8	0,2137007
chiffre_germinal_1_119_1	119	34,998	2	1201,2	0,342000914
chiffre_germinal_1_119_1	119	23,183	3	1124	0,327509022
chiffre_germinal_1_119_1	119	12,272	4	1067,9	0,286961651
chiffre_germinal_2_192_2	192	29,565	1	897,5	0,341596484
chiffre_germinal_2_192_2	192	38,695	2	1100	0,519308972
chiffre_germinal_2_192_2	192	38,697	3	1019,4	0,487338591
chiffre_germinal_2_192_2	192	43,913	4	1078,2	0,417573214
chiffre_germinal_4_247_1	247	13,184	1	965,5	0,459317994
chiffre_germinal_4_247_1	247	46,818	2	1046,2	0,589496469
chiffre_germinal_4_247_1	247	40,453	3	1088,6	0,589876604
chiffre_germinal_4_247_1	247	46,819	4	1286,9	0,677875662
chiffre_germinal_3_318_2	318	30,4	1	955,2	0,661393356
chiffre_germinal_3_318_2	318	52,8	2	1247,4	0,961967802
chiffre_germinal_3_318_2	318	38,8	3	1269,5	0,970536661
chiffre_germinal_3_318_2	318	74,4	4	1609	1,402224135
chiffre_germinal_1_366_1	366	18,332	1	911,9	0,751217437
chiffre_germinal_1_366_1	366	60	2	1300,8	1,293551469
chiffre_germinal_1_366_1	366	71,667	3	1328,3	1,235517931
chiffre_germinal_1_366_1	366	70	4	1155,1	0,979178667
chiffre_germinal_1_446_1	446	22,4	1	890,3	0,904692936
chiffre_germinal_1_446_1	446	56,8	2	1374,1	1,780303597
chiffre_germinal_1_446_1	446	70,4	3	1249,7	1,322660065
chiffre_germinal_1_446_1	446	49,6	4	1188,8	1,250602794
chiffre_germinal_2_570_1	570	18	1	883,5	0,995710278
chiffre_germinal_2_570_1	570	71,2	2	1398,8	1,93889482
chiffre_germinal_2_570_1	570	66	3	1248,5	1,752062154
chiffre_germinal_2_570_1	570	77,2	4	1342	1,758465028
chiffre_germinal_5_743_1	743	28,332	1	972,6	1,469328332
chiffre_germinal_5_743_1	743	56,249	2	1294,8	2,048874044
chiffre_germinal_5_743_1	743	67,499	3	1250	1,927586746
chiffre_germinal_5_743_1	743	71,25	4	1304,9	2,013058019
chiffre_germinal_5_1082_3	1082	42,4	1	899	1,981043696
chiffre_germinal_5_1082_3	1082	74,8	2	1451,5	3,44497757
chiffre_germinal_5_1082_3	1082	73,6	3	1338,5	3,244619608
chiffre_germinal_5_1082_3	1082	81,6	4	1282,3	2,92930038
chiffre_germinal_52_1199_1	1199	23,845	1	1076,5	2,650442195
chiffre_germinal_52_1199_1	1199	63,847	2	1247,2	3,35212307
chiffre_germinal_52_1199_1	1199	74,231	3	1475	4,129423809
chiffre_germinal_52_1199_1	1199	66,923	4	1495,9	3,671046805

7.1.3 Recuit Simulé évalué par la méthode des n-grammes

nom	taille	%_caracteres_egaux	ngram	nb_iter	temps
chiffre_germinal_1_119_1	119	11,819	1	3100	1,86130145
chiffre_germinal_1_119_1	119	29,547	2	3100	2,02649758
chiffre_germinal_1_119_1	119	49,546	3	3100	1,46481431
chiffre_germinal_1_119_1	119	19,092	4	3100	0,89086084
chiffre_germinal_2_192_2	192	16,087	1	3100	2,98836854
chiffre_germinal_2_192_2	192	35,654	2	3100	3,00166946
chiffre_germinal_2_192_2	192	37,826	3	3100	2,2575825
chiffre_germinal_2_192_2	192	61,303	4	3100	1,45152862
chiffre_germinal_4_247_1	247	14,093	1	3100	3,92254119
chiffre_germinal_4_247_1	247	49,092	2	3100	4,01027792
chiffre_germinal_4_247_1	247	76,817	3	3100	2,95788968
chiffre_germinal_4_247_1	247	57,273	4	3100	1,81967421
chiffre_germinal_3_318_2	318	17,2	1	3100	5,13764417
chiffre_germinal_3_318_2	318	53,6	2	3100	5,06461334
chiffre_germinal_3_318_2	318	78	3	3100	3,85677824
chiffre_germinal_3_318_2	318	66	4	3100	2,48815444
chiffre_germinal_1_366_1	366	16,249	1	3100	5,61365058
chiffre_germinal_1_366_1	366	59,583	2	3100	5,42708633
chiffre_germinal_1_366_1	366	80,416	3	3100	4,37971194
chiffre_germinal_1_366_1	366	82,5	4	3100	3,1684319
chiffre_germinal_1_446_1	446	18,4	1	3100	7,07035766
chiffre_germinal_1_446_1	446	52,4	2	3100	6,94251397
chiffre_germinal_1_446_1	446	74,8	3	3100	5,57327449
chiffre_germinal_1_446_1	446	75,6	4	3100	3,71519997
chiffre_germinal_2_570_1	570	22,8	1	3100	8,8847178
chiffre_germinal_2_570_1	570	64	2	3100	8,64754226
chiffre_germinal_2_570_1	570	84,8	3	3100	7,20182934
chiffre_germinal_2_570_1	570	82,8	4	3100	4,89917605
chiffre_germinal_5_743_1	743	20,833	1	3100	11,0296566
chiffre_germinal_5_743_1	743	80,834	2	3100	10,8589158
chiffre_germinal_5_743_1	743	88,75	3	3100	9,63187339
chiffre_germinal_5_743_1	743	82,502	4	3100	6,38678541
chiffre_germinal_5_1082_3	1082	26,4	1	3100	17,0066279
chiffre_germinal_5_1082_3	1082	78,8	2	3100	17,0001615
chiffre_germinal_5_1082_3	1082	81,2	3	3100	13,4525157
chiffre_germinal_5_1082_3	1082	87,6	4	3100	9,71585882
chiffre_germinal_52_1199_1	1199	25,769	1	3100	19,5852663
chiffre_germinal_52_1199_1	1199	83,461	2	3100	19,130015
chiffre_germinal_52_1199_1	1199	88,846	3	3100	16,0712713
chiffre_germinal_52_1199_1	1199	90,769	4	3100	11,7597788

7.1.4 *Hill Climbing* évalué par la méthode de la corrélation de Pearson

nom	taille	%_caracteres_egaux	nb_iter	score_final	temps
chiffre_germinal_1_119_1	119	4,094	6000	0,712956157	2,261422825
chiffre_germinal_2_192_2	192	1,305	6000	0,739471676	2,82611351
chiffre_germinal_4_247_1	247	4,548	6000	0,724334199	3,319811535
chiffre_germinal_3_318_2	318	8	6000	0,750158194	3,86924448
chiffre_germinal_1_366_1	366	3,333	6000	0,742760164	5,52164166
chiffre_germinal_1_446_1	446	6	6000	0,734019289	4,748970652
chiffre_germinal_2_570_1	570	2,8	6000	0,718872739	5,859348035
chiffre_germinal_5_743_1	743	4,168	6000	0,755057395	6,800807548
chiffre_germinal_5_1082_3	1082	2,8	6000	0,68166033	9,229564738
chiffre_germinal_52_1199_1	1199	4,231	6000	0,686976496	9,979178548

7.1.5 Recuit Simulé évalué par la méthode de la corrélation de Pearson

nom	taille	%_caracteres_egaux	nb_iter	score_final	temps
chiffre_germinal_1_119_1	119	4,092	3100	0,663388155	1,503901625
chiffre_germinal_1_366_1	366	6,668	3100	0,528726209	3,717935944
chiffre_germinal_1_446_1	446	4	3100	0,474407951	3,087809277
chiffre_germinal_3_318_1	318	3,6	3100	0,361028942	2,053674984
chiffre_germinal_2_570_1	570	2,8	3100	0,313906215	2,932762289
chiffre_germinal_2_192_2	192	3,48	3100	0,463268799	1,482723975
chiffre_germinal_5_1082_3	1082	6	3100	0,529706609	4,693678308
chiffre_germinal_5_743_1	743	6,251	3100	0,44227867	3,32326777
chiffre_germinal_4_247_1	247	3,184	3100	0,391218473	1,560636401
chiffre_germinal_52_1199_1	1199	3,463	3100	0,578446501	4,95868423

7.2 Code

7.2.1 Fichier auxiliaires.py

```
#imported libraries
import string
import random
import matplotlib.pyplot as plt
from collections import Counter

#ensemble des caracteres spéciaux : voyelles
ens_e = {"è", 'é', 'ê', 'ë', 'ē', 'ė', 'ę', "È", 'É', 'Ê', 'Ë', 'Ē', 'Ė', 'Ę'}
ens_i = {'ï', 'í', 'ī', 'į', 'ì', 'ì', 'Ï', 'Í', 'Ī', 'Į', 'Ì', 'Ì'}
ens_o = {'ö', 'ò', 'ó', 'ø', 'ō', 'õ', 'Ö', 'Ò', 'Ó', 'Ø', 'Ō', 'Õ' }
ens_a = {'à', 'á', 'â', 'ä', 'ã', 'ā', 'å', 'À', 'Á', 'Â', 'Ä', 'Ã', 'Ā', 'Å'}
ens_u = {'û', 'ù', 'ú', 'ū', 'ü', 'Û', 'Ù', 'Ú', 'Ū', 'Ü'}
ens_y = {'ÿ', 'Ÿ'}

#ensemble des caracteres spéciaux : consonnes
ens_c = {'ç', 'Ç'}

#retourne si le caractere est une lettre acceptee par nos algorithmes de
↪ dechiffrement
```

```
19  def est_lettre(lettre):
20      return ord('a') <= ord(lettre) <=ord('z') or ord('A') <= ord(lettre)
        ↪  <=ord('Z')
21
22  def min_en_maj(lettre):
23      if ord('a') <= ord(lettre) <=ord('z'):
24          return lettre.upper()
25      else:
26          return lettre
27
28  #passage en majuscule, suppression de la ponctuation et remplacement des
    ↪  caractères spéciaux par leur lettre équivalente
29  def nettoyage(nom_fichier):
30      res = ""
31      with open(nom_fichier, 'r', encoding="utf8") as fichier:
32          texte = fichier.read()
33      for i in range(len(texte)):
34              if texte[i] in ens_e:
35                  res += 'E'
36              elif texte[i] in ens_i:
37                  res += 'I'
38              elif texte[i] in ens_o:
39                  res += 'O'
40              elif texte[i] in ens_a:
41                  res += 'A'
42                  elif texte[i] in ens_c:
43                  res += 'C'
44              elif texte[i] in ens_u:
45                  res += 'U'
46              elif texte[i] in ens_y:
47                  res += 'Y'
48              elif texte[i] == 'œ' or texte[i] == 'Œ':
49                  res += 'OE'
50              elif texte[i] == 'æ' or texte[i] == 'Æ':
51                  res += "AE"
52              elif est_lettre(texte[i]):
53                  res += min_en_maj(texte[i])
54      return res
55
56
57  # permuter tous les A d'un texte en B
58  def permutation(A, B, texte):
59    texte_res = ""
60    for i in range(len(texte)):
61      if texte[i] == A:
62        texte_res += B
63      elif texte[i] == B:
64        texte_res += A
65      else:
66        texte_res += texte[i]
67    return texte_res
68
```

```
69
70  # a utiliser dans chiffrement aleatoire :  renvoie un dictionnaire contenant une
    ↪  cle de chiffrement aleatoire
71  def substitution_aleatoire():
72    alphabet = string.ascii_uppercase
73    alphabet_list = list(alphabet)
74    # Melanger la liste des lettres aleatoirement
75    random.shuffle(alphabet_list)
76    # Creer un dictionnaire de substitution aléatoire
77    substitution_dict = {}
78    for i in range(len(alphabet)):
79      substitution_dict[alphabet[i]] = alphabet_list[i]
80    return substitution_dict
81
82
83  # chiffre un texte donne pour pouvoir appliquer nos algorithmes
84  def chiffrement_aleatoire(text):
85    substitution = substitution_aleatoire()
86    texte_chiffre = ''
87    for c in text:
88      texte_chiffre += substitution[c]
89    return texte_chiffre
90
91  # renvoie un dictionnaire (ngram : occurrences) (utilise dans score et dans les
    ↪  stats)
92  def get_ngram_frequencies(text, n):
93    alphabet ="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
94    ngrams = []
95    for i in range(len(text) - n + 1):
96      ngrams.append(text[i:i + n])
97    for s in alphabet:
98      if s not in ngrams:
99        ngrams.append(s)
100   ngram_counts = Counter(ngrams)
101   return dict(ngram_counts)
102
103
104 #calcule le % de similtitude entre 2 clefs
105 def evaluation_lettres_en_commun(text, clef_text, clef_original):
106   cpt = 0
107   lettres = set(text)
108   n = len(set(text))
109
110   for i in range(26):
111     if clef_text[i] == clef_original[i] and clef_text[i] in lettres:
112       cpt += 1
113   return round(100 * cpt / n, 2)
114
115
116  #prend une clef de dechiffrement en majuscules et renvoie la clef de chiffrement
     ↪  correspondante
117 def convertir_clef(clef):
```

```
    alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    res = ""
    for lettre in alphabet:
      #trouver l'index de la lettre correspondante dans la clef de déchiff
      index = clef.find(lettre)
      #ajouter la lettre correspondante dans l'alphabet a la clef de chiff
      res += alphabet[index]
    return res

#trace un histogramme ordonne par ordre d'occurences croissant sur les dictionnaires de ngrams
def histogramme_ordonne(dico):
    dico_ordonne=dict(sorted(dico.items(), key=lambda item: item[1]))
    lettres=[cle for cle in dico_ordonne]
    frequences=[dico[cle] for cle in dico_ordonne]
    plt.figure(figsize=[26,13])
    plt.bar(lettres,frequences)
    plt.title('nombres d occurences des lettres par ordre croissant')
    plt.savefig("Histogramme_ordonne.png",dpi=300);
    plt.show();

```

7.2.2 Fichier stats.py

Notons que les fichiers *n-grammes.json* sont stockés dans un répertoire *stats* et ont été créés à partir des statistiques du texte *Les Misérables* de Victor Hugo.

```
#imported libraries
import string
import json

# initialisation des dictionaires de statistiques (ngrams : occurrences) faites sur le texte "Les miserables" qu'on a nettoye avant

with open('statistiques_monogrammes.json', 'r') as f:
  monograms_stats = json.load(f)

with open('statistiques_bigrammes.json', 'r') as f:
  bigrams_stats = json.load(f)

with open('statistiques_trigrammes.json', 'r') as f:
  trigrams_stats = json.load(f)

with open('statistiques_4grammes.json', 'r') as f:
  quatregrams_stats = json.load(f)

#global variable
caracteristiques = ("", 0, string.ascii_uppercase, string.ascii_uppercase, 100, 1, 200, 10, 1000, 0.0)
```

7.2.3 Fichier scores.py

```
#imported libraries
import collections
import numpy as np
import string
import math

#imported files
import auxiliaires as aux
import stats

def score_ngram(text, key, n, ngrams_stats):
  score = 0
  ngram = ""
  for i in range(len(text) - n + 1):
    ngram = text[i:i + n]
    if ngram in ngrams_stats:
      score += np.log(ngrams_stats[ngram])
  return round(score, 2)


def score_pearson(text, cle):

  alphabet = string.ascii_uppercase

  dict_txt = aux.get_ngram_frequencies(text, 1)
  if '\n' in dict_txt:
    del dict_txt['\n']
  dict_fr = stats.monograms_stats

  tot_langue = sum(dict_fr.values())
  freq_fr = [dict_fr[alphabet[i]]/tot_langue for i in range(26)] #frequences des lettres dans la langue francaise
  freq_txt = [dict_txt[cle[i]]/len(text) for i in range(26)] #frequences de leur chiffrement correspondant dans le texte

  freq_fr = np.array(freq_fr)
  freq_txt = np.array(freq_txt)

  moy_fr = np.mean(freq_fr)
  moy_txt = np.mean(freq_txt)

  ecart_fr = freq_fr - moy_fr
  ecart_txt = freq_txt - moy_txt

  return abs( np.sum(ecart_fr * ecart_txt) / np.sqrt(np.sum(ecart_fr**2) * np.sum(ecart_txt**2)) )

def score_1gram(text, key):
  return score_ngram(text, key, 1, stats.monograms_stats)

```

```
48  def score_2gram(text, key):
49    return score_ngram(text, key, 2, stats.bigrams_stats)
50
51  def score_3gram(text, key):
52    return score_ngram(text, key, 3, stats.trigrams_stats)
53
54  def score_4gram(text, key):
55    return score_ngram(text, key,  4, stats.quatregrams_stats)
56
```

7.2.4 Fichier hillclimbing.py

```
1   #imported libraries
2   import random
3   import string
4
5   #imported files
6   import stats
7   import auxiliaires as aux
8
9   def hill_climbing(text, score, MAXiter, MAXsurplace, key=string.ascii_uppercase,
    ↪  *args):
10
11    nom, taille, cle_u, cle_t, lettres_com, ngram_u, nbmax_it, nb_it, score_f, temps
      ↪  = stats.caracteristiques
12
13    current_text = text
14    optimized_text = text
15    optimized_score = score(optimized_text,aux.convertir_clef(key), *args)
16    clef =key
17
18    i = 0
19    surplace = 0
20    while  i < MAXiter and surplace < MAXsurplace:
21      indice1 = random.randint(0, 25)
22      indice2 = random.randint(0, 25)
23      while (indice1 == indice2):
24        indice2 = random.randint(0, 25)
25      if(indice1>indice2) :
26        indice1, indice2 = indice2, indice1
27
28      current_text = aux.permutation(clef[indice1], clef[indice2], current_text)
29      clef_temp
        ↪  =clef[:indice1]+clef[indice2]+clef[indice1+1:indice2]+clef[indice1]+clef[indice2+1:]
30      current_score = score(current_text, aux.convertir_clef(clef_temp), *args)
31
32      if current_score >= optimized_score:
33        optimized_text = current_text
34        optimized_score = current_score
35        surplace = 0
```

```
        clef = clef_temp
      else:
        current_text = optimized_text
        surplace += 1
      i += 1

  print("Le score final apres ", i, " iterations est ", optimized_score)
  print("On a stagné ", surplace, " fois")

  score_f = optimized_score
  cle_t = aux.convertir_clef(clef)
  nbmax_it = MAXiter
  nb_it = i
  stats.caracteristiques = (nom, taille, cle_u, cle_t, lettres_com, ngram_u,
  ↪  nbmax_it, nb_it, score_f, temps)

  return optimized_text

```

7.2.5 Fichier recuitsimule.py

```
#imported libraries
import numpy as np
import random
import string

#imported files
import stats
import auxiliaires as aux


def recuit_simule_paliers(text,
↪  score,T,nbiter_dans_palier,c,Tmin,key=string.ascii_uppercase, *args):
  nom, taille, cle_u, cle_t, lettres_com, ngram_u, nb_it, score_f, temps, other =
  ↪  stats.caracteristiques
  best_text = text
  best_score = score(best_text, aux.convertir_clef(key), *args)
  current_text = text
  optimized_text = text
  optimized_score = best_score
  clef = key
  r = 1

  nbiter = 0
  while T > Tmin:  #on peut ajouter des conditions de sortie : MAXiter..
    indice1 = random.randint(0, 25)
    indice2 = random.randint(0, 25)
    while (indice1 == indice2):
      indice2 = random.randint(0, 25)
    if (indice1 > indice2):
```

```
28          indice1, indice2 = indice2, indice1
29
30      current_text = aux.permutation(clef[indice1], clef[indice2], current_text)
31      clef_temp = clef[:indice1] + clef[indice2] + clef[indice1 + 1:indice2] +
        ↪  clef[indice1] + clef[indice2 + 1:]
32      current_score = score(current_text, aux.convertir_clef(clef_temp), *args)
33
34      diff = current_score - optimized_score
35      if diff < 0:
36          r = random.random()
37      if diff >= 0 or r < np.exp(diff / T):
38          optimized_text = current_text
39          optimized_score = current_score
40          clef = clef_temp
41          if best_score < optimized_score:
42              best_text = optimized_text
43              best_score = optimized_score
44      else:
45          current_text = optimized_text
46
47      nbiter += 1
48      if nbiter % nbiter_dans_palier == 0:
49          T = c * T
50
51    score_f = best_score
52    cle_t = aux.convertir_clef(clef)
53    nb_it = nbiter
54    stats.caracteristiques = (nom, taille, cle_u, cle_t, lettres_com, ngram_u,
      ↪  nb_it, score_f, temps, other)
55
56    return best_text
57
```

7.2.6 Fichier tableaux.py

```
1   #imported libraries
2   import pandas as pd
3   import string
4   import time
5   import os
6
7   #imported files
8   import scores as scores
9   import stats as stats
10  import auxiliaires as aux
11  import hillclimbing as hc
12  import recuitsimule as rs
13
14  #global variable
15  caracteristiques = ("", 0, string.ascii_uppercase, string.ascii_uppercase, 100, 1,
    ↪  200, 10, 1000, 0.0)
```

```
16
17  data = []
18  repertoire_entree = "repertoire"
19
20  for nom_fichier in os.listdir(repertoire_entree):
21    chemin_entree = os.path.join(repertoire_entree, nom_fichier)
22    if nom_fichier[1] == 'h':
23      with open(chemin_entree, "r") as f:
24        contenu = f.read()
25        # diviser le nom de fichier en trois parties
26        parties = nom_fichier.split("_")
27        nom_cle = "repertoire/clef" + nom_fichier[7:]
28        with open(nom_cle, "r") as file:
29          cle = file.read()
30
31        #cas fonction d'evaluation score_ngram
32        for i in range(1,5):
33          if i==1:
34            ngram_stats = stats.monograms_stats
35            score = scores.score_1gram
36          elif i==2:
37            ngram_stats = stats.bigrams_stats
38            score = scores.score_2gram
39          elif i==3:
40            ngram_stats = stats.trigrams_stats
41            score = scores.score_3gram
42          else :
43            ngram_stats = stats.quatregrams_stats
44            score = scores.score_4gram
45
46        """ #cas fonction d'evaluation score_pearson
47        score = scores.score_pearson
48        """
49          #indentation : si fonction d'évaluation score_ngram : dans la boucle for i
            ↪  in ... ; sinon non
50        for j in range(10):
51          start = time.time()
52
53          """ #cas methode de dechiffrement hill_climbing
54          dechiffrement = hc.hill_climbing(contenu, score, 6000, 7000)
55          """
56
57          dechiffrement = rs.recuit_simule_paliers(contenu, score, 1000, 100, 0.8,
            ↪  1)
58
59          end = time.time()
60          nom, taille, cle_u, cle_t, lettres_com, ngram_u, nb_it, nb_max_it,
            ↪  score_f, temps= stats.caracteristiques
61          temps = end - start
62          nom = nom_fichier
63
64          #cas fonction d'evaluation score_pearson
```

```
65          ngram_u = 1
66
67          """ #cas fonction d'evaluation score_ngram
68            ngram_u = i
69          """
70          taille = int(parties[3])
71          cle_u = cle
72          lettres_com = aux.evaluation_lettres_en_commun(contenu, cle_t, cle_u)
73          print("texte dechiffre ",j ," :", dechiffrement)
74          data.append([nom, taille, cle_u, cle_t, lettres_com, ngram_u, nb_it,
            ↪  nb_max_it, score_f, temps])
75
76  df = pd.DataFrame(data, columns=['nom', 'taille', 'cle_utilisee', 'cle_trouvee',
    ↪  '%_caracteres_egaux', 'ngram_utilise', 'nb_iterations', 'nbmax_iter',
    ↪  'score_final', 'temps'])
77  df.to_csv('tableau.csv', index=False)
```

7.2.7 Fichier main.py

```
1
2   #imported files
3   import scores as scores
4   import hillclimbing as hc
5   import recuitsimule as rs
6
7   algo = input("Choisissez une  metaheuristique (recuitsimule , hillclimbing ou
    ↪  recuitsimule_et_hillclimbing) : ")
8   score_txt = input("Choisissez une fonction score (ngram ou pearson) : ")
9   texte = input("Entrez le texte : ")
10
11  if score_txt == "ngram":
12      choix = int(input("Choisissez un n (entre 1 et 4) : "))
13      if choix == 1:
14          score = scores.score_1gram
15      elif choix == 2:
16          score = scores.score_2gram
17      elif choix == 3:
18          score = scores.score_3gram
19      elif choix == 4:
20          score = scores.score_4gram
21
22  elif score_txt == "pearson":
23      score = scores.score_pearson
24  else:
25    print("La fonction score", score_txt, "n'est pas definie")
26    exit(1)
27
28
29  if algo == "recuitsimule":
30    choix_param = int(input("Vouliez- vous choisir les paramètres? Tapez 1 si oui, 2
      ↪  si vous voulez les paramètres optimaux: "))
```

```
31    if choix_param == 1:
32      T = float(input("Entrez la valeur de T initiale : "))
33      nbiter_dans_palier = int(input("Entrez le nombre d'itérations dans un palier :
        ↪  "))
34      c = float(input("Entrez la valeur de c (entre 0 et 1 exclus) : "))
35      Tmin = float(input("Entrez la température minimale : "))
36      print("Texte déchiffré avec recuit simulé: ", rs.recuit_simule_paliers(texte,
        ↪  score, T, nbiter_dans_palier, c, Tmin))
37    elif choix_param == 2:
38      print("Texte déchiffré avec recuit simulé: ", rs.recuit_simule_paliers(texte,
        ↪  score, 1000, 100, 0.8, 1))
39
40
41  elif algo == "hillclimbing":
42    choix_param = int(input("Vouliez- vous choisir les paramètres? Tapez 1 si oui, 2
      ↪  si vous voulez les paramètres optimaux: "))
43    if choix_param == 1:
44      MAXiter = int(input("Entrez le nombre maximum d'itérations : "))
45      MAXsurplace = int(input("Entrez le nombre maximum de stagnations : "))
46      print("Texte déchiffré avec hill climbing: ", hc.hill_climbing(texte, score,
        ↪  MAXiter, MAXsurplace))
47    if choix_param == 2:
48       print("Texte déchiffré avec hill climbing: ", hc.hill_climbing(texte,score,
         ↪  2000, 250))
49
50  elif algo =="recuitsimule_et_hillclimbing":
51    choix_param = int(input("Vouliez- vous choisir les paramètres? Tapez 1 si oui, 2
      ↪  si vous voulez les paramètres optimaux: "))
52    if choix_param == 1:
53      T = float(input("Entrez la valeur de T initiale : "))
54      nbiter_dans_palier = int(input("Entrez le nombre d'itérations dans un palier :
        ↪  "))
55      c = float(input("Entrez la valeur de c : "))
56      Tmin = float(input("Entrez la température minimale : "))
57      MAXiter = int(input("Entrez le nombre maximum d'itérations : "))
58      MAXsurplace = int(input("Entrez le nombre maximum de stagnations : "))
59      print("Texte déchiffré avec recuit simulé: ",rs.recuit_simule_paliers(texte,
        ↪  score, T, nbiter_dans_palier, c, Tmin))
60      print("Texte déchiffré avec hill climbing: ", hc.hill_climbing(texte, score,
        ↪  MAXiter, MAXsurplace))
61    if choix_param == 2:
62      print("Texte déchiffré avec hill climbing: ", hc.hill_climbing(texte,score,
        ↪  2000, 250))
63      print("Texte déchiffré avec recuit simulé: ",rs.recuit_simule_paliers(texte,
        ↪  score, 1000, 100, 0.8, 1))
64
65  else:
66    print("La metaheuristique", algo, "n'est pas definie")
67    exit(1)
```