

Cryptanalyse
Université de Rouen
Projet annuel Licence 3 Informatique

Louka Boivin
Alexis Osmont
Manon Guillard
Alan Durand
Sami Babigeon

2020-2021



Sommaire

1	Présentation	3
1.1	But du Projet	3
1.2	Comment accéder au site	4
1.3	Fonctionnement du site	5
1.3.1	Console	7
2	Implantation	8
2.1	Structure de données	8
2.1.1	Dictionnaire	8
2.2	Partie César	11
2.2.1	Fonctionnement du chiffre de César et de son décryptage	11
2.2.2	Implantation de l'attaque par force brute	11
2.2.3	Implantation de l'analyse fréquentielle	13
2.2.4	Implantation de l'attaque par mot probable	14
2.3	Partie Substitution	15
2.3.1	Fonctionnement du Codage par Substitution	15
2.3.2	Analyse de fréquences 'bête'	16
2.3.3	Branch and Bound	18
2.3.4	Recuit simulé et fonction d'évaluation	19
2.3.5	Chaine de Markov	22
3	Système	23
3.1	Raspberry	23
3.1.1	Installation Raspberry	23
3.1.2	Portail captif	23
3.1.3	Redirection des requêtes	26
3.2	Serveur UFR	28
3.2.1	Mise en place	28
4	Problèmes rencontrés	31
4.1	Automatiser la substitution	31
4.2	Expérience utilisateur	31
5	Améliorations possibles	31
6	Conclusion	31

1 Présentation

1.1 But du Projet

Dans ce projet, il nous était demandé d'étudier le système de chiffrement par permutation aussi appelé chiffrement par substitution. Ce système de chiffrement est un chiffrement mono-alphabétique. Autrement dit, chaque lettre du message est remplacée par une autre lettre de l'alphabet.

Nous devons ainsi développer un site web à but pédagogique expliquant et mettant en oeuvre le chiffrement par substitution. Dans le site, l'utilisateur peut essayer de chiffrer et de déchiffrer lui-même un texte. Il peut remplacer une lettre par une autre pour essayer de retrouver le texte en clair ou il peut chiffrer un texte avec la clé de son choix. Il peut alors s'aider de boîtes à outils dans lesquelles il peut calculer l'analyse fréquentielle, autrement dit, le nombre d'occurrences d'une ou plusieurs lettres dans le texte ou alors il peut tester l'appartenance d'un mot au texte. L'utilisateur peut également essayer des outils de chiffrement et de déchiffrement automatiques. Ces derniers permettent, à partir d'un texte crypté, de retrouver la clé automatiquement grâce à des algorithmes que nous avons développés. Ces algorithmes seront détaillés plus loin.

Nous avons donc étudié et présenté dans une première partie le chiffrement par substitution qui consiste à prendre comme clé une permutation de l'alphabet. Dans une seconde partie nous avons étudié le chiffre de César qui est un cas particulier du chiffrement par substitution. En effet, ce chiffrement consiste à décaler les lettres du message d'un nombre de position dans l'alphabet.

Nous avons développé le site web en HTML, CSS et JavaScript. Nous avons également utilisé du Java pour développer un programme qui permet de générer un dictionnaire (voir page 8). Enfin, nous avons utilisé du C pour créer un programme capable d'analyser des textes en français pour obtenir une analyse fréquentielle de référence pour chaque lettre.

1.2 Comment accéder au site

On peut accéder au site par plusieurs moyens :

- par URL : lorsque l'utilisateur est connecté au réseau Wifi de l'Université, il suffit qu'il aille sur l'URL : <https://www.srv-dpi-proj-chiffrement-1.univ-rouen.fr> pour accéder au site. Pour plus d'informations, voir la partie III (page 28).
- par la Rasbery : l'utilisateur peut se connecter au réseau Wifi Cryptanalyse pour pouvoir accéder au site. Pour plus d'informations, voir la partie III (page 23).

Le site est également accessible depuis un ordinateur et un téléphone portable. La version mobile a été pensée pour être simplifiée et plus ergonomique. Toutes les pages ont ainsi été adaptées en fonction de la version : les fichiers CSS et HTML sont différents. C'est une fonction JavaScript qui reconnaît quel type d'appareil l'utilisateur utilise et qui redirige vers les bons fichiers.



Figure 1: Version mobile

1.3 Fonctionnement du site

Le site se présente de la manière suivante : tout d’abord, il y a la page d’accueil. C’est dans cette page que l’utilisateur va pouvoir choisir s’il souhaite un texte chiffré ou clair et dans le cas d’un texte chiffré, choisir le type de son chiffrement (césar ou substitution). Il peut alors entrer un texte de son choix ou en générer un aléatoirement.

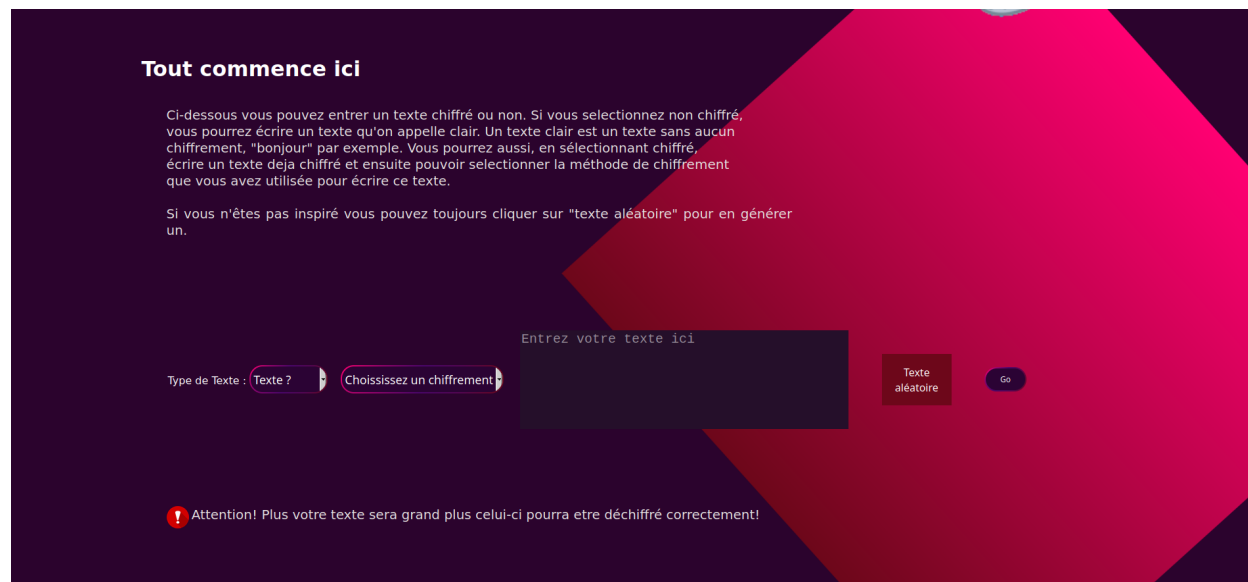


Figure 2: Page d’accueil

En fonction des choix de l’utilisateur, une nouvelle page s’affichera. Il y a au total trois pages différentes qui peuvent s’afficher. A noter qu’en réalité, tout le site repose sur une unique page. Ce sont juste des blocs HTML qui apparaissent et disparaissent en fonction des choix utilisateur. Cela se fait avec du CSS et une fonction JavaScript dédiée.

Revenons donc aux trois pages possibles. Sur chaque page, il y a des explications sur les méthodes de chiffrement et d’attaque. Il y a aussi des outils que l’utilisateur peut tester pour mieux comprendre ces différentes méthodes.

En haut de chaque page, se trouve, dans une barre, le texte (clair ou chiffré) choisi précédemment sur la page d’accueil. Lorsqu’il va tester une action avec l’un des outils, le texte résultant va alors s’afficher sur cette même barre. L’utilisateur va alors pouvoir comparer son texte de base et le texte après chiffrement ou déchiffrement. L’utilisateur peut aussi “Annuler” ou “Refaire” une action avec respectivement les boutons ↶ et ↷. S’il souhaite entrer un nouveau texte, il peut cliquer sur le bouton 🗑️. Pour plus d’explications sur la console voir la partie **1.3.1 Console**.

Présentons plus précisément les trois pages interactives :

- Premièrement, si l’utilisateur choisit un texte clair, il arrive sur la page “Chiffrement”. Il peut alors essayer le chiffrement par césar en donnant un décalage. Il peut également essayer le chiffrement par substitution en entrant une clé pour chiffrer, autrement dit en entrant un alphabet.
- Deuxièmement, si l’utilisateur choisit un texte chiffré par César, il arrive sur la page “Attaque du chiffre de César”. Il peut alors essayer différents types d’attaque : l’attaque par force brute, l’attaque par fréquence ou l’attaque par mot probable.

- Troisièmement, si l'utilisateur choisit un texte crypté par substitution, il arrive sur la page "Attaque par substitution". Il peut alors essayer de décrypter le texte par lui-même ou il peut essayer de le décrypter automatiquement.



Figure 3: Page d'attaque par substitution

1.3.1 Console

La console permet de modifier le texte, clair ou chiffré, pour permettre aux utilisateurs de chiffrer ou d'essayer de décrypter le texte. Elle comprend plusieurs champs et boutons :

- **Remplacer x par y** : permet de remplacer toutes les occurrences de x dans le texte d'origine par y dans l'autre texte. Si l'utilisateur a choisi "Non chiffré" sur la page d'accueil, alors le texte d'origine sera le texte clair (à droite) et les modifications se feront donc sur le chiffré (à gauche). Si l'utilisateur a choisi chiffré ce sera donc l'inverse. Il suffit de remplir les deux champs par une ou plusieurs lettres puis d'appuyer sur ➡. Plus précisément :
 - Si l'utilisateur est sur la page d'attaque du code de César, étant que le décalage est le même pour chaque lettre, toutes les lettres selon remplacés si l'utilisateur en remplace une.
 - Pour la page de chiffrement et celle d'attaque du chiffrement par substitution on peut remplacer directement plusieurs lettres d'un coup. Par exemple certains groupes de lettres peuvent revenir souvent et on peut vouloir remplacer 'bhj' par 'est'. Il faut par contre que les groupes de lettres fassent la même taille.
 - On peut remplacer une lettre déjà remplacée auparavant, et on ne peut pas remplacer deux lettres différentes par une seule lettre.
 - Après avoir lancé l'attaque par substitution, les remplacements se font directement sur le texte clair pour que cela soit plus simple de remplacer les dernières lettres qui ont pu échapper à l'algorithme.
- ↶, ↷ : nous avons implanté un historique des modifications pour pouvoir annuler ou refaire la dernière modification apportée au texte. Cela fonctionne pour les remplacements effectués dans la console mais également pour les différents chiffrements et attaques. Cet historique utilise deux piles qui sauvegardent les textes précédents et suivants.
- 🗑️ : permet de revenir à zéro, d'effacer les textes clairs et chiffrés, l'historique des modifications et les diagrammes des fréquences.

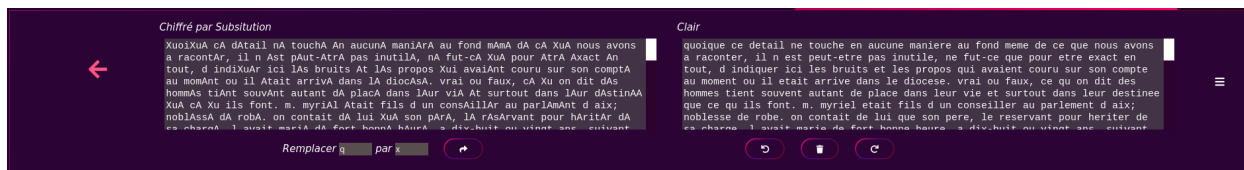


Figure 4: Zoom sur la console

2 Implantation

2.1 Structure de données

2.1.1 Dictionnaire

Les algorithmes principaux que nous avons implémentés pour le site sont les algorithmes d'attaque de texte chiffré par substitution et par la méthode de César. Dans les deux cas, afin de pouvoir vérifier les conditions d'arrêts de ces algorithmes, et de savoir s'ils avaient correctement décrypté, nous avons dû implémenter un dictionnaire.

En effet, étant donné que l'attaque doit être automatique, il faut vérifier si la clé qu'on a trouvée est la bonne. La manière la plus efficace est d'utiliser un dictionnaire et de vérifier dans le texte si chaque mot appartient au dictionnaire ou non, et de calculer le taux de mots qui sont dans le dictionnaire. Ensuite, si l'on a calculé toutes les possibilités, comme dans le cas de l'attaque par force brute du code de César, on prend la clé qui possède le meilleur taux, sinon, on peut déterminer un seuil au-dessus duquel on peut considérer que l'attaque a réussi.

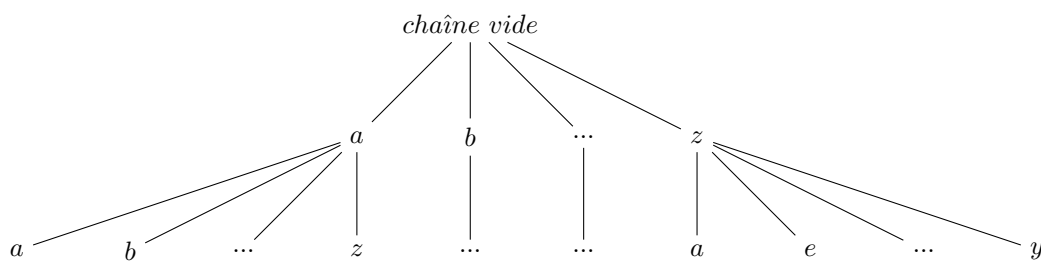
Pour cela nous avons utilisé une structure de données appelés "trie" ou "arbre préfixe".

Il s'agit d'une structure d'arbre souvent utilisée pour stocker des chaînes de caractères. La recherche, l'insertion et la suppression d'un mot dans un arbre préfixe est en complexité linéaire sur la longueur du mot dans le pire des cas, $O(n)$, avec n la longueur du mot. Le principe d'un arbre préfixe est le suivant :

- Contrairement à un arbre binaire de recherche, aucun noeud ne stocke la chaîne à laquelle il est associé, c'est sa position dans l'arbre qui la détermine.
- La racine correspond à la chaîne vide.
- Pour tout noeud, ses descendants ont en commun le même préfixe.

Dans l'arbre que nous avons implanté chaque noeud possède n fils, n étant le nombre de lettre dans l'alphabet utilisé, ici l'alphabet français donc $n = 26$. Cependant parmi ces fils certains sont des noeuds pouvant eux-mêmes avoir des fils, et d'autres sont des feuilles qui ont une valeur nulle. Les fils non nuls sont ceux dont le noeud correspond au préfixe d'un mot existant dans le dictionnaire, et les feuilles sont des noeuds correspondant à des préfixes qui n'existent pas dans la langue.

Par exemple le noeud contenant la lettre z juste au dessous de la racine, possédera moins de fils que celui de même rang contenant la lettre a car il y a moins de mot commençant par z que par a dans la langue française (en se basant sur notre dictionnaire). Voici le schéma simplifié d'un arbre préfixe :



Cela permet de réduire énormément la taille de l'arbre car nous n'avons pas besoin de garder tous les noeuds. De plus pour savoir si un préfixe appartient au dictionnaire il suffit de vérifier l'existence de celui-ci.

Précision : pour éviter d'avoir un alphabet trop grand et également pour pouvoir utiliser ce dictionnaire dans nos fonctions de déchiffrement tous les mots du dictionnaire sont formatés pour contenir uniquement des caractères de l'alphabet de base (26 lettres). Tous les accents sont retirés, et les apostrophes et les traits-d'union sont remplacés par des espaces et donc créent 2 mots.

Pour tester si un mot appartient au dictionnaire il faut également tester si la suite de caractères dont il est composé est dans l'arbre préfixe. Ce test s'effectue donc en $O(n)$, avec n la longueur du mot. Mais pour éviter de dire que n'importe quel préfixe d'un mot quelconque appartient au dictionnaire, nous avons rajouté un attribut à chaque noeud qui permet de savoir si ce noeud correspond à la fin d'un mot.

Pour des raisons d'efficacité la structure est légèrement différente dans les sources mais pour simplifier, la structure d'un noeud est équivalente à la suivante :

```
node {
    end : boolean,
    children : node[]
}
```

Avec cette structure de données la recherche de l'appartenance d'un mot au dictionnaire se fait très simplement et rapidement :

Data :

dict : noeud racine du dictionnaire (on suppose qu'il est initialisé),
alphabet : l'alphabet utilisé, un tableau de caractères,
word : mot à rechercher dans le dictionnaire,
n : longueur du mot à rechercher.

```
1 p ← dict;
2 for i ← 0 to n do
3   | idx ← indexOf(alphabet, word[i]);
4   | if idx = -1 then
5   |   | return false;
6   | end
7   | p ← p.children[idx];
8 end
9 return p.end;
```

Algorithme 1 : Recherche d'un mot dans un arbre préfixe

On suppose ici qu'on possède une fonction **indexOf**() qui prend en paramètre un alphabet et une lettre et qui renvoie la position de cet lettre dans cette alphabet, et -1 si elle n'est pas dedans. On considérera que cette fonction s'exécute en temps constant.

Pour initialiser ce dictionnaire il nous fallait un véritable dictionnaire, un fichier texte contenant le plus de mot possible de la langue française, ainsi que les verbes et leurs conjugaisons. Notre enseignante référente de projet, Mme Bardet, nous a fourni ce dictionnaire et il contenait au départ plus de 300 000 mots (**French.txt**). Cependant plusieurs problèmes se sont posés :

- La structure de données que nous avons utilisée pour implanter le dictionnaire est optimisée pour améliorer la complexité en temps des algorithmes mais très peu pour la complexité en espace. En effet, l'arbre préfixe est un arbre où chaque noeud peut avoir jusqu'à 26 fils (pour notre alphabet) et ou la hauteur de l'arbre est définie par la longueur du plus grand mot contenu dans le dictionnaire. Si on prend le pire des cas pour la langue française usuelle cela correspond à $26^{25} = 2.367.10^{35}$ noeuds. En réalité le nombre de noeuds est majoré par le nombre de mots du dictionnaire donc nous réussirons toujours à l'initialiser, mais avec un dictionnaire de 300 000 mots cela prenait plusieurs secondes à chaque fois qu'on se rendait sur la page du site.
- Un autre problème était que ce dictionnaire était peut-être "trop" complet, il contenait énormément de mots que l'on ne rencontre jamais dans la vie courante, et d'autres suites de caractères qui ne sont pas vraiment des mots. Et donc souvent, en testant nos algorithmes, on arrivait à un résultat faux car le dictionnaire avait considéré qu'un certain mot était correct alors que non.

Pour résoudre ces problèmes, nous avons trouvé et utilisé un dictionnaire beaucoup plus réduit de 22 000 mots (`liste_francais_original.txt`), qui regroupait les mots les plus courants de la langue ainsi que les expressions et les noms propres les plus souvent utilisés. Pour le temps de chargement, le problème était résolu, le dictionnaire se chargeait en moins d’une seconde lorsqu’on accédait à la page. Cependant pour l’efficacité des algorithmes ce dictionnaire n’était pas suffisant car il rencontrait très souvent des mots inconnus.

Nous avons donc écrit un script qui permet de “nourrir” le dictionnaire de nouveaux mots. C’est un programme écrit en Java qui prend en paramètres un nombre quelconques de fichiers (dictionnaires, textes) et qui donne en sortie un dictionnaire, sous la forme d’un fichier texte avec un mot par ligne.

Le fonctionnement est très simple : on prend en entrée n fichiers, puis pour chacun de ces fichiers on va extraire chaque mot, les formater, et les mettre dans une Map où une clé est un mot et la valeur est le nombre de textes différents dans lequel ce mot apparaît. Ensuite on garde les clés dont les valeurs atteignent ou dépassent un seuil fixé, nous avons choisi ici 2. En passant en paramètre le premier dictionnaire (300 000 mots), le second (22 000) et n’importe quel nombre de textes en français, on pouvait ainsi à la fois retirer les mots du premier dictionnaire qu’on ne rencontre jamais réellement dans la langue, et ajouter au dictionnaire plus de mots courants.

Nous avons donc utilisé des textes de littérature française, ancienne et moderne, afin de remplir le dictionnaire avec le vocabulaire le plus varié possible. 11 textes (2 dictionnaires + 9 textes) ont été utilisés pour créer le nouveau dictionnaire, et malgré la grande quantité de mot à traiter, l’utilisation de structure de données efficace en Java (ici une `TreeMap`) permet au script de s’exécuter en quelques secondes. Ce script et les textes utilisés sont situés dans le dossier `annexe/dictionary`.

Au final, nous avons obtenu un dictionnaire plus efficace, comptant environ 36 000 mots (voir le fichier `liste_francais.txt`).

2.2 Partie César

2.2.1 Fonctionnement du chiffre de César et de son décryptage

Le chiffre de César est une manière de chiffrer un message de manière simple : on choisit un nombre (appelé clé de chiffrement) et on décale vers la droite toutes les lettres de notre message du nombre choisi. Par exemple : si je choisis comme clé le nombre 2. Alors la lettre A deviendra C, le B deviendra D ... et le Z deviendra B. On remarquera facilement que pour déchiffrer, il suffit de faire la même chose mais avec l'opposé de la clé (ici -2), soit un décalage vers la gauche.

Le chiffre de César est assez simple à décrypter pour la simple raison qu'il n'y a que 25 décalages possibles car après ça on redémarre l'alphabet. Par exemple, si la clé est de 26 alors le texte sera inchangé, si la clé est de 27, alors le décalage sera d'une seule lettre. Pour cause l'alphabet va de 'a' vers 'z' et une fois à 'z' on revient à 'a' Une clé de 28 revient donc à une clé de 2. On peut donc essayer toutes les clés possibles, c'est ce qu'on appelle la méthode par force brute.

On peut aussi faire une analyse fréquentielle, c'est-à-dire compter le nombre d'occurrence de chaque lettre. En français, statistiquement, la lettre la plus présente est le 'e', donc si votre texte est assez long il y a de fortes chances que la lettre la plus présente dans votre message chiffré soit le 'e' lui aussi. À partir de là, on peut facilement déduire la clé de chiffrement en calculant l'écart entre la lettre chiffrée et le 'e'. Ce sera ici notre deuxième méthode d'attaque. La dernière s'appelle l'attaque par mot probable, elle consiste à supposer la présence d'un mot dans le texte.

Si nous avons décidé d'utiliser le code César dans notre projet, c'est tout d'abord pour sa simplicité pédagogique et sa compréhension. De plus, c'est un cas particulier du codage par substitution et enfin la force brute pouvait être utilisée dessus.

2.2.2 Implantation de l'attaque par force brute

Pour utiliser la force brute, nous récupérons le texte chiffré et nous testons toutes les clés allant de 0 à 25. Puis pour chaque clé "i" nous regardons quel est le pourcentage de mots présents dans le dictionnaire. Nous gardons à chaque fois en mémoire le plus grand pourcentage et à quel "i" il a été trouvé.

Data : *encryptedText* : texte lu sur l'entrée.

maxI : Indice ayant le plus de mots contenus dans le dictionnaire.

maxPourc : Plus grand pourcentage de mots contenus dans le dictionnaire.

```
1 maxI = 0;
2 maxPourc = 0;
3 for i = 0 ; i < 26; i ++ do
4   p = analyseTextCesar(encryptedText avec décalage de i);
5   if p > maxPourc then
6     maxPourc = p;
7     maxI = i;
8   end
9 end
```

```
10 // On affiche ensuite le texte décrypté avec la clé maxI
```

Algorithme 2 : Algorithme d'attaque par force brute

Et le détail de la fonction **analyseTextCesar** qui calcule pour un texte passé en paramètre, son pourcentage de mots appartenant au dictionnaire :

Data : *tab* : tableau contenant le texte découpé selon les espaces
 nbmot : nombre de mots lu
 trouve : nombre de mots présents dans le dictionnaire.

```
1 tab = text.split(' ');
2 nbmot = 0;
3 trouve = 0;
4 for i = 0 ; i < tab.length ; i ++ do
5   | if dico.contains(tab[i]) then
6   |   | trouve ++;
7   | end
8   | nbmot ++;
9 end
10 retourner (trouve * 100) / nbmot;
```

Algorithme 3 : Algorithme d'analyse de texte déchiffré

Pour présenter la force brute de manière pédagogique, le bouton “force brute” est là. Il montre ainsi à intervalle régulier, un nouveau décalage sur le texte chiffré pour ainsi faire comprendre que son rôle est de tester toutes les clés.

La force brute est efficace sur le codage César car il n’y a que 26 possibilités, cependant ce n’est pas le cas de tous les chiffrements comme par exemple substitution qui contient 26! possibilités ($\approx 4 \times 10^{26}$).

2.2.3 Implantation de l'analyse fréquentielle

Pour utiliser l'analyse fréquentielle on analyse tout simplement le texte chiffré, et nous faisons apparaître sous la forme d'un graphique le nombre d'occurrence de chaque lettre. Cela permet de remarquer les lettres les plus présentes. Il faut savoir que statistiquement en français la lettre la plus présente est le "e". Avec le chiffre de césar, le taux d'apparition de chaque lettre reste inchangé, à décalage près, entre le clair et le chiffré. Autrement dit, si il y a 9 'e' dans le texte clair, il y aura forcément 9 'e' dans le texte chiffré mais représentés par une autre lettre.

Il suffit donc de découvrir quelle lettre est en réalité le "e" pour découvrir quelle est la clé, en calculant le décalage entre la lettre chiffré et le "e".

Par exemple : si la lettre la plus présente dans le message chiffré est le "t" :
"t" - "e" = 20 - 5 = 15

Data : *tab* : texte en paramètre découpé selon les espaces
letter : lettre entrée par l'utilisateur (String)
trouve : nombre de mots présents dans le dictionnaire.
codeE : code ascii de 'e'.
codeDec : code ascii de *letter*.
decalage: décalage entre 'e' et *letter*.

```
1 if letter == "" || letter.length > 1 then
2   | message "mauvaise clé";
3 end
4 codeE = "e".charCodeAt(0);
5 codeDec = letter.charCodeAt(0);
6 decalage = codeDec - codeE;
7 if decalage < 0 then
8   | decalage = 26 + decalage;
9 end
10 // On affiche ensuite le texte décrypté de clé decalage;
```

Algorithme 4 : Algorithme d'attaque par analyse fréquentielle

2.2.4 Implantation de l'attaque par mot probable

L'attaque par mot probable consiste à supposer l'existence d'un mot dans le message chiffré. Il est donc possible d'en déduire la clé du message si le mot choisi est bel et bien dans le message.

Par exemple prenons le mot chiffré suivant "OLEUH", nous pouvons essayer "BEIGE" :

On calcule le décalage entre chaque lettre :

- 'O' - 'B' = 13 [26]
- 'L' - 'E' = 7 [26] (Inutile d'aller plus loin, le décalage diffère)

Par contre si nous essayons "LIBRE" :

- 'O' - 'L' = 3 [26]
- 'L' - 'I' = 3 [26]
- 'E' - 'B' = 3 [26]
- 'U' - 'R' = 3 [26]
- 'H' - 'E' = 3 [26]

Le décalage étant le même partout, on peut supposer que "OLEUH" est en réalité "LIBRE".

Attention cependant avec cette méthode des erreurs peuvent avoir lieu comme sur les petits mots comme "un" ou "le" par exemple. Prenons le mot "rk", ce mot peut représenter "le" avec un décalage de 6 ou alors "un" avec un décalage de 23.

Data : *text* : texte chiffré

tab : texte en paramètre découper selon les espaces

mot : mot entrée par l'utilisateur

oldKey : décalage trouvé à précédente position *key* : décalage trouvé à la position actuelle

```
1 tab = text.split(' ');
2 for i = 0 ; i < tab.length; i ++ do
3   if tab[i].length == mot.length then
4     oldKey = tab[i].charCodeAt(0) - mot.charCodeAt(0);
5     if oldKey < 0 then
6       | oldKey = 26 + oldKey;
7     end
8     for u = 1 ; i < tab[i].length; u ++ do
9       | key = tab[i].charCodeAt(u) - mot.charCodeAt(u);
10      | if key < 0 then
11        | key = 26 + key;
12      | end
13    end
14    if key == oldKey then
15      | on affiche le texte déchiffré avec le decalage de key
16    end
17  end
18 end
19 message "le mot probable n'est pas contenu dans le texte"
```

Algorithme 5 : Algorithme de mots probables

2.3 Partie Substitution

2.3.1 Fonctionnement du Codage par Substitution

Le Codage par substitution est une généralisation du codage de César (ou plutôt ce dernier en est un cas particulier). Le principe du codage par substitution est simple : on part d'un espace de départ (l'alphabet clair) et on permute certaines lettres pour créer un nouvel alphabet que l'on va utiliser pour chiffer notre texte. Contrairement au codage de César, le codage par substitution est insensible à une attaque par force brute puisque le nombre de combinaisons possibles est $|A|!$ ou $|A|$ est la taille de l'alphabet utiliser.

Le nombre de combinaison possible est $26! \cdot 2^{88}$ pour l'alphabet classique, le standard de sécurité aujourd'hui étant 2^{128} . A titre d'exemple si votre ordinateur peut générer 10 milliards de combinaisons différentes à la seconde, il lui faudrait plusieurs siècles avant de trouver la bonne combinaison, il faut donc attaquer le problème autrement ...

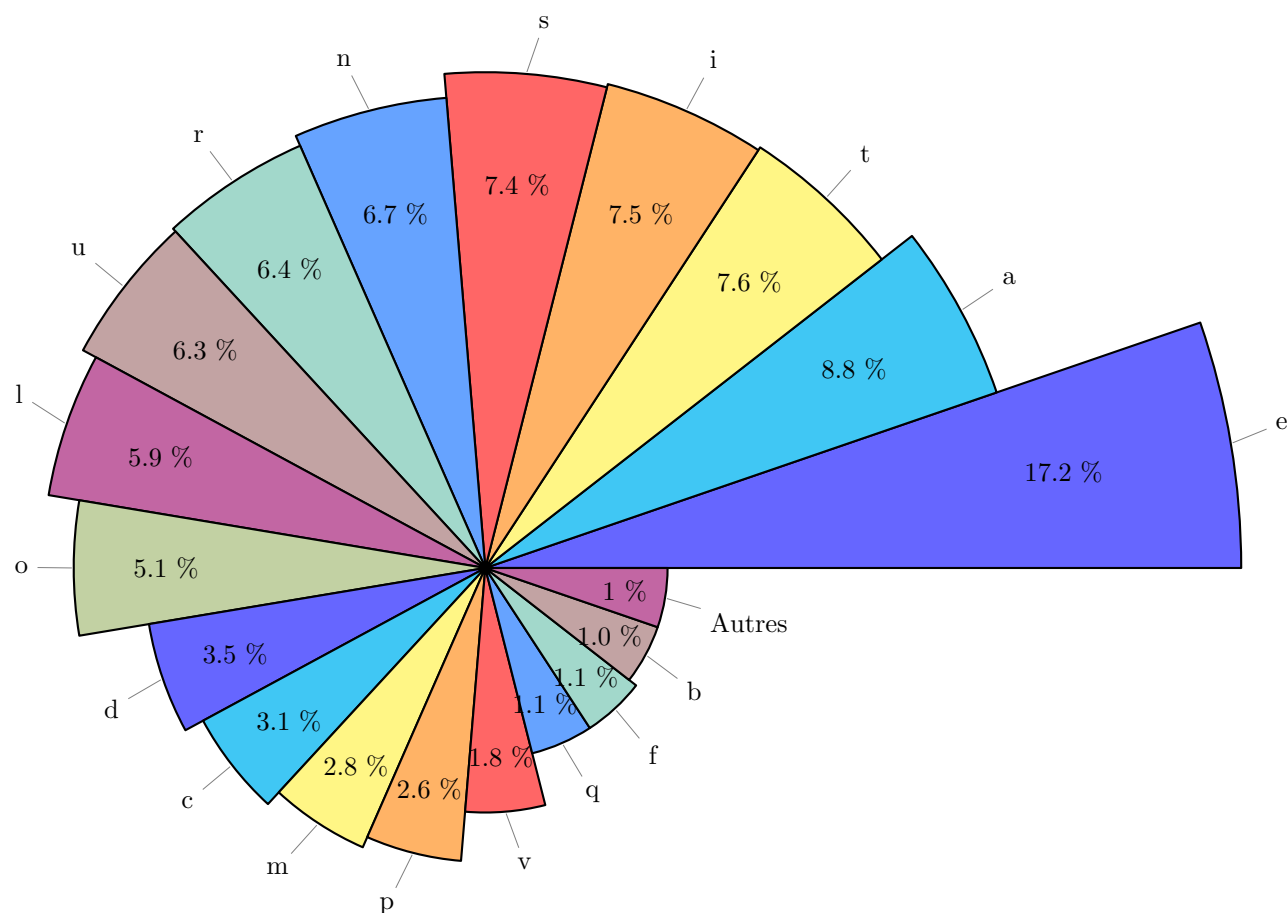
L'analyse de fréquence peut permettre de trouver plusieurs lettres mais nécessite un texte relativement grand et après avoir déterminé les lettres les plus fréquentes on se retrouve face au même problème que pour la force brute.

Note : Dans la suite de cette partie on utilise les notations suivantes :

A désignera un l'alphabet quelconque

$|A|$ est la taille de A

Id_A est l'alphabet identité sur A



Dans la suite, les solutions partielles ou totales évoquées sont des alphabets implémenté par des tableaux de caractères, l'alphabet Id_A correspond $[a_0, a_1, \dots, a_n]$ avec $n = |A|$, on peut également le représenter avec la notation des cycles algébriques comme ceci :

$$\begin{pmatrix} a_0 & a_1 & \dots & a_n \\ a_0 & a_1 & \dots & a_n \end{pmatrix}$$

ce qui signifie que l'élément a_0 est envoyé sur a_0 , a_1 sur a_1 , etc ... Pour représenter une permutation d'un alphabet, par exemple si on permute a_0 et a_2 Id_A on obtiendra :

$$\text{perm}(Id_A, a_0, a_2) = [a_2, a_1, a_0, \dots, a_n]$$

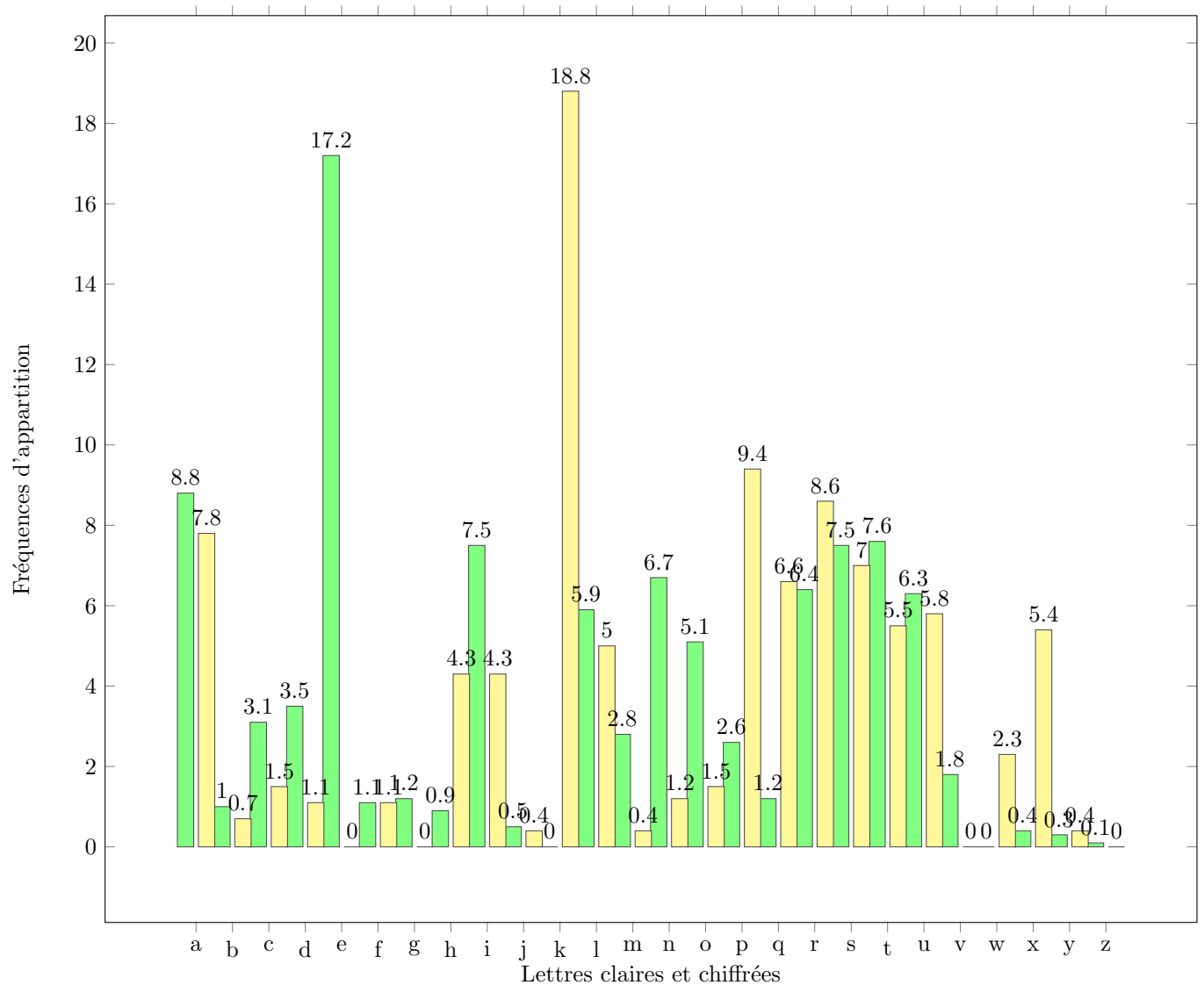
ou alors avec la notation algébrique :

$$\begin{pmatrix} a_0 & a_1 & a_2 & \dots & a_n \\ a_2 & a_1 & a_0 & \dots & a_n \end{pmatrix}$$

On peut représenter un alphabet par un cycle, or un cycle est un produit de permutations, on peut donc passer de n'importe quel alphabet à un autre en permutant des lettres un certain nombre de fois, y compris passer de l'alphabet qui chiffre le texte chiffré à l'alphabet identité (celui du texte clair), c'est cette idée qui a permis d'arriver à la résolution du problème.

2.3.2 Analyse de fréquences 'bête'

Dans cette première approche nous avons utilisé la fréquence d'apparition des lettres puis en règle générale des NGrammes (une suite de N lettres) pour déduire des associations, en comparant entre le texte chiffré et des statistiques de références dans la langue supposée du texte clair (grâce à l'outil *occur* écrit en C).



On remarque que dans le graphique ci-dessus certaines lettres ressortent plus que d'autres, par exemple on peut aisément supposer que le 'k' dans le texte chiffré représente un 'e'.

En moyenne la plupart des textes suivent la même distribution, plus le texte est grand plus il tend vers la distribution de référence. Comme expliqué précédemment cette méthode permet de déduire dans le meilleur des cas une petite partie de l'alphabet automatiquement et le reste des déductions devait être fait manuellement, en plus d'être peu robuste, très lourde (en terme de code) cette méthode était difficile à maintenir.

A force de chercher différentes solutions nous avons penser à utiliser la théorie des graphes et nous nous sommes intéresser aux algorithmes d'optimisations pour résoudre ce problème, comme la méthode suivante : le Branch and Bound.

2.3.3 Branch and Bound

Le Branch and Bound est une méthode générique de résolution de problèmes d'optimisation combinatoire qui consiste à trouver un point minimisant une fonction (coût) dans un ensemble dénombrable. L'idée de l'algorithme est de parcourir une partie du graphe des solutions possibles sans faire un parcours exhaustif puisque cela reviendrait au final à étudier toutes les possibilités c'est à dire $fact(|A|!)$ combinaisons.

L'idée de l'algorithme est de séparer récursivement l'espace de recherche ("Branching"), à lui seul cela revient à une méthode force brute. C'est pour améliorer la performance de l'algorithme que l'on utilise des bornes afin d'éliminer des combinaisons et ainsi ne pas parcourir tout l'espace de recherche. En arrivant sur un noeud (une solution partielle) en regardant la borne Sup (meilleure solution actuelle) et Inf (meilleure solution possible à partir de ce noeud) on peut donc choisir de continuer à explorer ou non le reste de l'arbre.

Data : B : Borne Supérieure

P : pile (LIFO) représente la liste des sommets à explorer

f : fonction d'évaluation

$bound$: fonction qui donne la borne inf

$bound(n)$ renvoie le min de f sur les fils de n

N : représente un noeud de l'arbre

```

1   $B =$  meilleure solution  $(+\infty)$ ;
    $P = [Id_A]$ ;
   while  $P \neq []$  do
2  |    $N = P.pop()$ ;
   |   if  $f(N) < B$  then
3  | |    $B = f(N)$ ;
4  | else
5  | |   for  $c$  in  $N.childs$  do
6  | | |   if  $bound(c) > B$  then
7  | | | |    $P.push(c)$ ;
8  | | | end
9  | | end
10 | end
11 end
```

Algorithme 6 : Algorithme de Branch and Bound

Note : le fait d'utiliser une pile permet de réaliser un parcours en profondeur, d'autres structures de données comme les files peuvent être utilisées pour produire différents types de recherches (par exemple en utilisant une file on obtient une recherche en largeur). Dans notre cas un parcours en profondeur semblait plus intéressant puisque c'est celui qui génère le moins de solutions à la fois (moins coûteux en mémoire).

On parcourt ainsi le graphe des possibilités en ajoutant dans notre pile les solutions que l'on juge intéressantes. Cependant pour que l'algorithme soit efficace, la fonction d'évaluation f doit être optimale et garantir que si $f(N1) < f(N2)$ avec $N1, N2$ des noeuds, alors $f(N1_i) < f(N2_j)$ pour tous i, j les fils respectifs de $N1, N2$. Ce qui est impossible puisque cette dernière s'appuie sur l'analyse de fréquences des bigrammes et trigrammes d'une **partie** de la langue puisqu'il n'est pas possible d'avoir la liste de tous les bigrammes et trigrammes de la langue cible, et même si cela l'était la fonction aurait une complexité beaucoup trop élevée pour être utilisable. Dans la mesure où $bound$ nécessite la définition de f elle en est par conséquent inutilisable.

En rencontrant ce problème nous avons compris que le choix d'une bonne fonction d'évaluation était primordial, cette fonction doit pouvoir être capable de donner un "score de vraisemblance" à une solution en ayant pour possible référence une partie de l'ensemble sur lequel on travaille (en clair : les statistiques sur les fréquences d'apparition des lettres les plus présentes).

2.3.4 Recuit simulé et fonction d'évaluation

Lors des recherches pour trouver une bonne fonction d'évaluation, nous nous sommes intéressés à une façon efficace de représenter l'information que l'on récupère des N -Grammes (pour optimiser les temps de calculs). Nous avons pensé à la notation matricielle, en effet en construisant un tableau (comme celui ci-dessous) on peut facilement voir quel bigrame est le plus présent mais aussi par quoi est suivie telle ou telle lettre. D'après plusieurs références ces tables caractérisent complètement une langue, le tableau du français n'est pas le même que celui de l'anglais) on peut attribuer une signature à chaque langue. De plus si un texte dans un alphabet A_0 est chiffré dans un autre alphabet A_1 , chiffré revient à mélanger la table de A_0 pour obtenir A_1 et déchiffrer à remettre dans le même ordre en s'appuyant sur les fréquences.

	α_0	α_1	α_i	α_{n-1}	α_n
α_0	$P_{\alpha_0\alpha_0}$	$P_{\alpha_0\alpha_1}$	$P_{\alpha_0\alpha_i}$	$P_{\alpha_0\alpha_{n-1}}$	$P_{\alpha_0\alpha_n}$
α_1	$P_{\alpha_1\alpha_0}$	$P_{\alpha_1\alpha_1}$	$P_{\alpha_1\alpha_i}$	$P_{\alpha_1\alpha_{n-1}}$	$P_{\alpha_1\alpha_n}$
α_i	$P_{\alpha_i\alpha_0}$	$P_{\alpha_i\alpha_1}$	$P_{\alpha_i\alpha_i}$	$P_{\alpha_i\alpha_{n-1}}$	$P_{\alpha_i\alpha_n}$
α_{n-1}	$P_{\alpha_{n-1}\alpha_0}$	$P_{\alpha_{n-1}\alpha_1}$	$P_{\alpha_{n-1}\alpha_i}$	$P_{\alpha_{n-1}\alpha_{n-1}}$	$P_{\alpha_{n-1}\alpha_n}$
α_n	$P_{\alpha_n\alpha_0}$	$P_{\alpha_n\alpha_1}$	$P_{\alpha_n\alpha_i}$	$P_{\alpha_n\alpha_{n-1}}$	$P_{\alpha_n\alpha_n}$

Dans cet exemple on peut voir que le bigrame le plus présent (en vert clair) est $\alpha_0\alpha_n$, on peut en déduire que la lettre qui statistiquement suit le plus souvent un α_0 est un α_n , en mélangeant cette table pour avoir un nouvel alphabet on pourra facilement déduire quelle lettre code α_0 et α_n en cherchant la case en vert clair. C'est cette technique que code la fonction d'évaluation.

Data : *alpha* : alphabet qui decode *text*
text : le texte chiffré
sum : variable intermediaire
Bigrams : tableau de fréquences d'apparition des bigrames (contient le bigrame et sa fréquence)
Count : $Count(text, alpha, w)$ la fonction qui renvoie le nombre de fois où le bigrame *w* codé par l'alphabet *alpha* apparaît dans le texte chiffré *text*.

```

1 sum = 0;
  for e in Bigrams do
2   | sum = sum + (e.occurences/100)Count(text,alpha,e.bigram)
3 end
4 renvoyer sum / Bigrams.length;

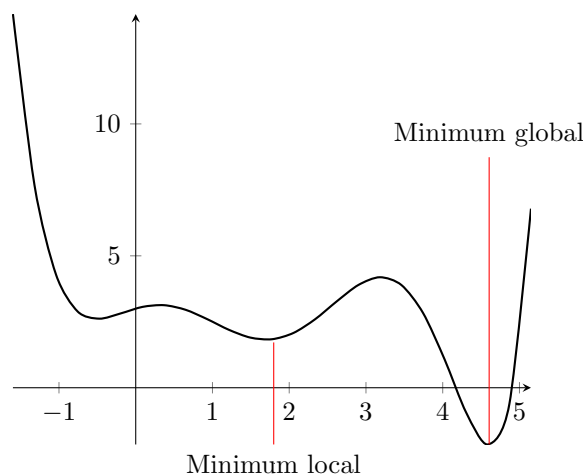
```

Algorithme 7 : Pseudo code de la fonction d'évaluation

Ainsi un bigrame très fréquent dans les statistiques de référence qui serait correctement decodé par *alpha*, aura une valeur très faible, en prenant la moyenne on s'assure d'avoir le score global de l'alphabet pour tous les bigrames, un alphabet ou la moitié des lettres sont bonnes sera aussi voir plus intéressant qu'un alphabet avec une ou deux très bonnes lettres. Le but est maintenant de minimiser cette fonction, c'est pour cela que l'on utilise l'algorithme de recuit simulé (puisque l'on est dans un espace discret et pas continu).

L'algorithme de recuit simulé (inspiré d'un processus utilisé en métallurgie) est un algorithme d'optimisation pour trouver les extremas d'une fonction, il s'appuie sur le critère de Metropolis. Le but de ces algorithmes est en général de trouver le minimum d'une fonction, pour cela on se place sur un point de la fonction et on regarde les voisins de notre point, si l'un de nos voisins est plus petit que nous on se déplace sur sa position et on recommence le processus ... Le problème en faisant ainsi est que l'on peut se retrouver dans un "puits" c'est a dire un minimum local de la fonction (comme illustré ci-dessous). Pour assurer la convergence vers le minimum global on s'autorise de temps en temps (de manière aléatoire) à se déplacer vers un voisin plus haut que le point actuel (pour remonter la pente).

Le recuit simulé utilise le même principe en ajoutant une variation : la température. Au début de l'algorithme la température est élevée, on s'autorise donc plus facilement à explorer des possibilités moins intéressantes (voir le calcul du critère dans l'algorithme plus bas), à chaque itérations on décroît la température pour se concentrer sur les bonnes solutions.



Cette méthode a plusieurs avantages :

- Elle donne un résultat en fonction d'un nombre d'itérations donné en paramètre ce qui permet d'être sûr que la fonction finira par se terminer, on peut donc adapter les paramètres au système qui l'utilise (PC, mobile, tablette).
- Elle est très simple à implanter et nécessite peu de ressources.

En revanche le choix des hyper-paramètres (température, nombre d'itérations, ...) demande plusieurs ajustements avant de donner des résultats.

Data : *alpha* : alphabet qui decode *text*
text : le texte chiffré
temp : température initiale
iter : le nombre d'itérations

```
1 current = alpha;  
  best = alpha;  
  for i = 0; i < iter ++ i do  
2    candidat = Voisin(current);  
    if score(candidat) < score(best) then  
3      | best = candidat;  
4    end  
5    delta = score(candidat) - score(current);  
    critera =  $\exp^{\text{delta}/\text{temp}}$ ;  
    if delta <= 0 || Random() > critera then  
6      | current = candidat;  
7    end  
8    temp = temp * 0.99;  
9 end
```

Algorithme 8 : Pseudo code du Recuit Simulé

La fonction *Voisin*(*A*) renvoie une solution *A'* proche de *A*, dans notre cas on fait une permutation aléatoire de *A*. L'idée de l'algorithme est de chercher une meilleur solution au *voisinage* de la solution actuelle, a chaque étape on remplace la solution actuelle par un de ses voisins si ce voisin représente une meilleure solution (par rapport à la fonction de score) ou si un nombre que l'on tire aléatoirement est inférieur à notre *critère* de selection qui augmente au fur et à mesure que la température diminue, en procédant ainsi on assure le fait de parcourir tout l'espace de recherche comme pour l'algorithme de Metropolis.

2.3.5 Chaîne de Markov

En cherchant la fonction d'évaluation j'ai découvert le concept des Chaines de Markov, ce paragraphe y est dédié même si cette version de la fonction d'évaluation n'est pas utilisée dans le projet car trop coûteuse en temps.

Définition :

"En mathématiques, une chaîne de Markov est un processus de Markov à temps discret, ou à temps continu et à espace d'états discret. Un processus de Markov est un processus stochastique possédant la propriété de Markov : l'information utile pour la prédiction du futur est entièrement contenue dans l'état présent du processus et n'est pas dépendante des états antérieurs (le système n'a pas de "mémoire"). "

On peut donc modéliser un texte comme une chaîne de markov où la probabilité de chaque caractère dépend du précédent.

La probabilité que le mot $w = a_0a_1...a_n$ soit codé par A est :

$$Markov(w, A) = (1/(n+1)) * \prod_{i=0}^{n-1} P(A[a_i]A[a_{i+1}])$$

Avec $P(\alpha)$ la fréquence d'apparition du bigramme α dans la langue cible et $A[\beta]$ le caractère clair associé à β dans A .

La probabilité de la chaîne "salut" se calcule comme ceci :

$$s \xrightarrow{F_{sa}} a \xrightarrow{F_{al}} l \xrightarrow{F_{lu}} u \xrightarrow{F_{ut}} t = 0.9777247860417999$$

On obtiendra une valeur plus faible pour "salut" (puisque le but est de minimiser la fonction d'évaluation) que pour "zdsqv" car les bigrammes qui le compose sont moins fréquents dans la langue française.

$$z \xrightarrow{F_{zd}} d \xrightarrow{F_{ds}} s \xrightarrow{F_{sq}} q \xrightarrow{F_{qv}} v = 0.99872038169631$$

Cette version de la fonction d'évaluation donne des résultats similaire mais nécessite plus de temps de calculs $O(m^n)$ contre $O(m*n)$ avec m le nombre de bigrammes de référence et n le nombre de bigrammes du texte évalué.

Conclusion de la partie substitution

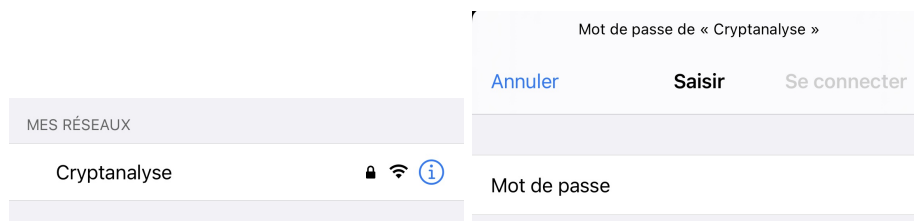
Pour conclure cette partie, nous avons exploré différentes manières d'aborder le problème, parmi toutes les méthodes utilisées c'est le recuit simulé qui a donné les meilleurs résultats, avec plus de temps on pourrait attaquer ce problème avec un autre genre d'algorithme : les algorithmes génétiques, au lieu de choisir un voisin de notre solution à chaque itération on reproduit plusieurs bonnes solutions entre elles pour obtenir de meilleures solutions.

3 Système

3.1 Raspberry

Objet : Utiliser un Raspberry PI 3 pour diffuser notre projet annuel.

Le but de cette partie était donc de créer un hotspot wifi local dissocié d'internet sur lequel nous devions nous connecter pour pouvoir accéder à notre projet annuel. La diffusion de notre projet peut se décomposer en deux parties. La première est le portail captif. Cette méthode permet une fois connecté au hotspot wifi du Raspberry d'afficher une page contenant notre projet.



La seconde méthode et la redirection des requêtes internet vers notre projet, en effet une fois connecté par wifi au hotspot généré par le Raspberry vous pouvez vous rendre sur votre navigateur internet et faire une recherche en format `http://LeSite.fr`. Votre requête est alors envoyé au Raspberry et celui-ci vous redirigera vers notre projet.

Pour mettre en place ces méthodes de diffusion il nous a fallu suivre différentes étapes :

3.1.1 Installation Raspberry

Avant toutes choses il nous a fallu mettre en place la raspberry et la rendre fonctionnelle.

Le Raspberry pi 3 que nous avons choisi comprend une carte wifi et nous a été fourni avec une micro carte sd, un clavier avec trackpad et son câble d'alimentation. Sur cette Raspberry, nous avons donc choisi d'installer le système d'exploitation "Raspbian" pour pouvoir utiliser notre raspberry. Nous avons sélectionné celui-ci car après quelques recherches, il nous a semblé être le plus adapté et le plus simple d'utilisation pour débiter dans ce domaine.

Pour télécharger puis installer Raspbian, je me permets de vous rediriger vers son site dédié via ce lien : <https://www.raspberrypi-france.fr/guide/installer-raspbian-raspberry-pi/>

3.1.2 Portail captif

RaspAP

Il nous faut télécharger RaspAP qui est un logiciel permettant aux Raspberry de devenir des Hotspots wifi en installant ou initialisant divers sous-répertoires.

- `/etc/lighttpd` : Permet de créer un serveur web pour pouvoir accueillir notre site.
- `/etc/dnsmasq.conf` : Permet de gérer les requêtes reçues par le Raspberry PI.

- /etc/dhcp/dhcpd.conf : Permet de configurer le serveur dhcp comme par exemple, réserver une adresse IP a un site en particulier.

- /etc/hostapd/hostapd.conf : Permet la création d'un hotspot wifi.

- /etc/network/interfaces : Permet de gérer la configuration réseau.

Entrez la commande suivante pour pouvoir installer RaspAp sur le Raspberry:

```
curl https://install.raspap.com
```

(curl est une commande permettant de transférer des données venant d'un serveur)

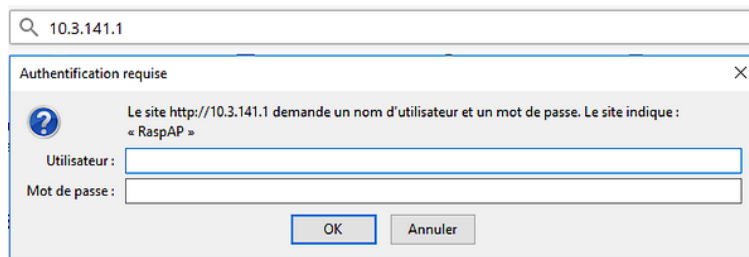
Puis la commande pour reboot le Raspberry pour appliquer ces changements :

```
sudo reboot now
```

(sudo est une commande permettant d'exécuter une commande en tant qu'administrateur)

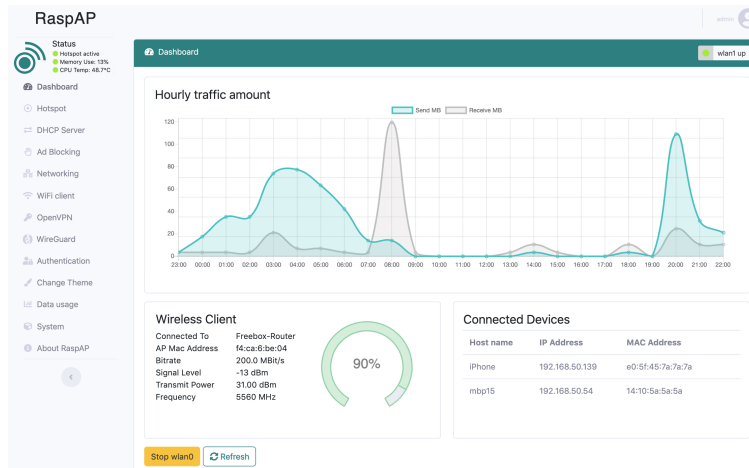
(reboot sert a redémarrer la machine, associé au paramètre now elle redémarre directement après l'exécution)

Une fois que RaspAP a modifié ou créé ces répertoires donc a fini de se télécharger, il nous suffit de nous rendre sur l'interface de RaspAP. Il vous faut lancer votre navigateur et ajouter dans la barre de recherche : localhost ou 10.3.141.1.



Les identifiants demandés sont les suivants :

admin
secret



Dans cette interface, il est possible de modifier simplement le nom ou le mot de passe du hotspot simplement ainsi que beaucoup d'autres choses, nous vous laissons aller voir le manuel d'utilisation de RaspAP si vous souhaitez plus d'informations. (<https://docs.raspap.com/manual/>)

Votre Hotspot wifi est opérationnel, avec comme adresse IP 10.3.141.1, comme nom raspbi-webgui et comme mot de passe ChangeMe.

Cependant celui-ci connecté ou non essaiera de vous rediriger vers vos requêtes internet.

Nodogsplash

Maintenant il nous faut mettre en place le portail captif :

Nous allons avoir besoin d'un compilateur pour Nodogsplash. L'un des plus utilisé dans ce cas précis est Libmicrohttpd.

Entrez la commande suivante pour le télécharger (pour Ubuntu/Debian) :

```
sudo apt install git libmicrohttpd-dev
```

('apt' est le gestionnaire de paquets des distributions Ubuntu, Debian et dérivés, 'install' permet de récupérer et d'installer les paquets donnés en argument.)

Puis vous pouvez récupérer nodogsplash sur git en clonant le répertoire :

```
git clone https://github.com/nodogsplash/nodogsplash.git
```

('git clone' permet de copier un répertoire git présent à l'adresse donnée en argument)

Rendez-vous dans celui-ci compilez le et installez le :

```
cd /nodogsplash (la commande 'cd' permet de nous rendre dans un dossier en donnant son chemin en argument)
```

```
make (la commande permet l'exécution du fichier makefile présent dans le dossier)
```

```
sudo make install (permet de lancer l'exécutable qui installera les fichiers nécessaires)
```

Il nous faut maintenant le configurer pour le faire marcher :

Entrez la commande pour accéder et modifier au fichier de configuration du portail captif :

```
sudo nano /etc/nodogsplash/nodogsplash.
```

(la commande nano permet l'écriture dans un fichier, si celui donné en argument n'existe pas il est alors créé)

Dans le fichier, ajoutez, modifiez ou simplement dé-commentez les lignes suivantes dans le fichier. Attention, certaines sont déjà présentes.

- GatewayInterface wlan0 (Pour définir le lieu d'action du portail captif)
- GatewayAddress 10.3.141.1 (Pour définir l'adresse de la page affichage du portail captif)
- MaxClients 250 (Pour le nombre de connexion en simultané possible)
- AuthIdleTimeout 480 (Pour le temps d'affichage du portail captif)

Faites CTRL + X puis CTRL + O ou Y pour sauvegarder les modifications. Uniquement si vous avez suivi la commande ci-dessus et utilisé la commande 'nano'

NodogSplash est configuré donc nous pouvons le lancer avec la commande suivante :

```
sudo nodogsplash
```

(permet d'exécuter l'application que vous venez d'installer et de configurer).

Pour que nodogsplash s'exécute à chaque allumage du Raspberry entrez la commande suivante :

```
sudo nano /etc/rc.local
```

Et ajoutez : "nodogsplash" une ligne avant le "exit 0" à la fin du fichier.

Le portail captif ainsi que le hotspot sont pleinement opérationnel maintenant mais nécessitent une connexion internet pour fonctionner.

3.1.3 Redirection des requêtes

Placez vous dans le répertoire de nodogsplash votre page grâce à la commande suivante :

```
sudo cd /etc/nodogsplash/htdocs/ ( le chemin des fichiers sources de vote site web )
```

Nous allons nous rendre dans le fichier de configuration de nodogsplash pour y modifier notre page du portail captif et y mettre notre page (index.html) : `sudo nano /etc/nodogsplash/nodogsplash.conf`
Trouvez et modifiez les lignes suivantes :

GatewayName Cryptanalyse (pour modifier le nom du portail captif)

StatusPage index.html (pour diriger en cas d'erreur le portail sur notre page)

SplashPage index.html (pour diriger le portail sur notre page)

Notre portail captif nous redirige maintenant vers notre site.

Pour rediriger les requêtes et par conséquent faire fonctionner le hotspot wifi et le portail captif sans internet il nous faut nous rendre dans `dnsmasq.conf` puis forcer les redirections des requêtes vers l'adresse IP de notre site.

Il vous suffit d'entrer la commande suivante pour pouvoir modifier le contenu du fichier `dnsmasq.conf` et modifier les paramètres de dns :

```
sudo nano /etc/dnsmasq.conf
```

Et d'ajouter la ligne suivante au fichier :

```
Address=//10.3.141.1
```

Ceci permet donc de dériver chaque requête sur notre serveur apache et plus précisément sur notre page web. Le Raspberry PI redirige maintenant toutes les requêtes http et propose un portail captif à tout ceux qui se connectent à son hotspot.

3.2 Serveur UFR

Objet : Diffusion de notre projet annuel sur le réseau de l'UFR

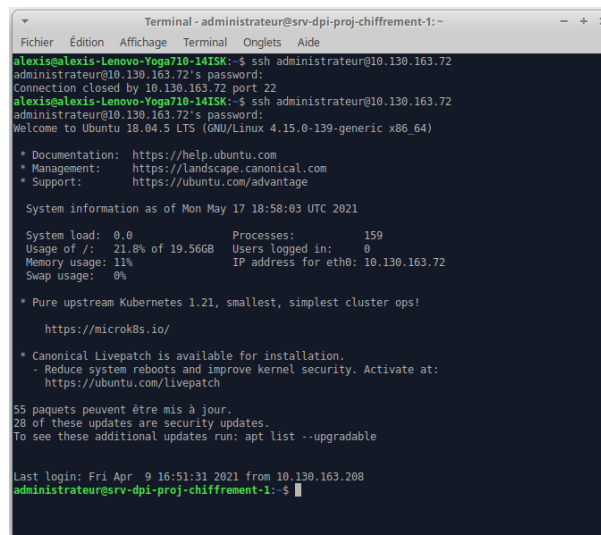
Le but de cette partie était donc de diffuser notre projet via le wifi de l'UFR ou via une connexion passant par le VPN de l'UFR. Pour diffuser notre projet à l'UFR nous avons eu besoin de l'aide Mr. Bruno Macadré qui nous a mis à disposition une machine virtuelle.

3.2.1 Mise en place

Pour la mise en place il nous a suffi de nous connecter en ssh sur notre machine virtuelle via la commande suivante :

```
ssh administrateur@10.130.163.72
```

(la commande ssh permet d'établir une connexion en entrant en argument un login et un ip qui sont ici respectivement, administrateur et 10.130.163.72 séparés par '@')
une fois cette commande exécutée il vous sera demandé un mot de passe qui vous permettra de finaliser la connexion.



```
Terminal - administrateur@srv-dpi-proj-chiffrement-1:~
Fichier  Edition  Affichage  Terminal  Onglets  Aide
alexis@alexis-Lenovo-Yoga710-14ISK:~$ ssh administrateur@10.130.163.72
administrateur@10.130.163.72's password:
Connection closed by 10.130.163.72 port 22
alexis@alexis-Lenovo-Yoga710-14ISK:~$ ssh administrateur@10.130.163.72
administrateur@10.130.163.72's password:
Welcome to Ubuntu 18.04.5 LTS (GNU/Linux 4.15.0-139-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

System information as of Mon May 17 18:58:03 UTC 2021

System load:  0.0          Processes:    159
Usage of /:   21.8% of 19.56GB   Users logged in:  0
Memory usage: 11%          IP address for eth0: 10.130.163.72
Swap usage:   0%

 * Pure upstream Kubernetes 1.21, smallest, simplest cluster ops!
   https://microk8s.io/

 * Canonical Livepatch is available for installation.
   - Reduce system reboots and improve kernel security. Activate at:
     https://ubuntu.com/livepatch

55 paquets peuvent être mis à jour.
28 of these updates are security updates.
To see these additional updates run: apt list --upgradable

Last login: Fri Apr  9 16:51:31 2021 from 10.130.163.208
administrateur@srv-dpi-proj-chiffrement-1:~$
```

Une fois connecté à la machine virtuelle, il nous a fallu mettre en place un serveur pour nous permettre d'héberger notre site web. Pour ce faire, nous avons utilisé le serveur apache en le mettant en place via la commande suivante pour l'installer (sur Ubuntu/Debian) :

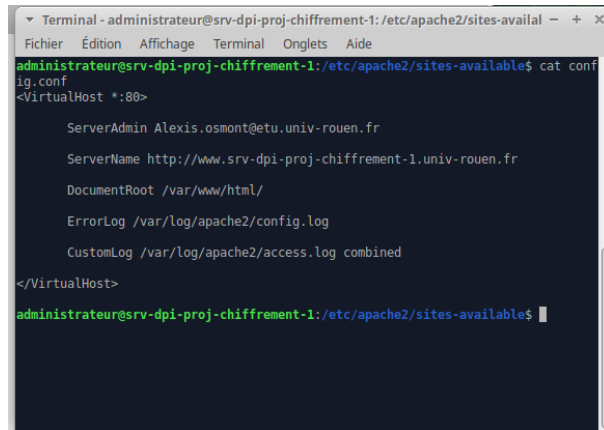
```
sudo apt install apache2
```

Le serveur est maintenant installé il faut donc le configurer.

Entrez la commande :

```
sudo touch /etc/apache2/sites-available/config.conf
```

Cette commande nous permet de créer le fichier de configuration de notre serveur apache qui s'appellera config.conf. Dans notre cas nous avons configuré ce fichier comme ci-dessous.



```
Terminal - administrateur@srv-dpi-proj-chiffrement-1:/etc/apache2/sites-available - + x
Fichier  Édition  Affichage  Terminal  Onglets  Aide
administrateur@srv-dpi-proj-chiffrement-1:/etc/apache2/sites-available$ cat config.conf
<VirtualHost *:80>

    ServerAdmin Alexis.osmont@etu.univ-rouen.fr

    ServerName http://www.srv-dpi-proj-chiffrement-1.univ-rouen.fr

    DocumentRoot /var/www/html/

    ErrorLog /var/log/apache2/config.log

    CustomLog /var/log/apache2/access.log combined

</VirtualHost>
administrateur@srv-dpi-proj-chiffrement-1:/etc/apache2/sites-available$
```

- VirtualHost *:80 : Le serveur accepte les connexions venant de n'importe quelle adresse IP "*" sur le port 80. (Port d'écoute du protocole http)
- ServerAdmin : Courrier électronique de l'administrateur
- ServerName : Le serveur sera appelé par l'adresse https://www.srv-dpi-proj-chiffrement-1.univ-rouen.fr
- DocumentRoot : Les fichiers du site seront dans le répertoire /var/www/html/

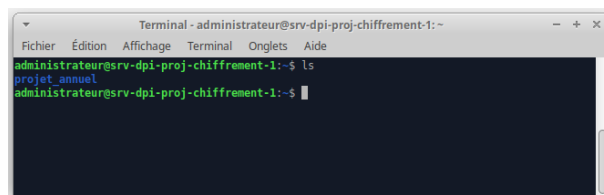
Tout est configuré il ne nous reste plus qu'à activer le serveur via la commande :

```
sudo a2ensite /etc/apache2/sites-available/config.conf
```

Le serveur est maintenant fonctionnel il faut donc y déposer nos fichiers. Pour ce faire, il faut utiliser la commande :

```
sudo scp Documents/projet_annuel.zip administrateur@10.130.163.72:
```

Cette commande permet de copier un fichier venant de la machine source (celle sur laquelle vous êtes) vers la machine virtuelle au niveau source (au même niveau après avoir exécuter la commande "cd").



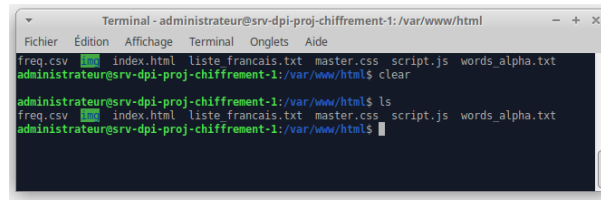
```
Terminal - administrateur@srv-dpi-proj-chiffrement-1:~ - + x
Fichier  Édition  Affichage  Terminal  Onglets  Aide
administrateur@srv-dpi-proj-chiffrement-1:~$ ls
projet_annuel
administrateur@srv-dpi-proj-chiffrement-1:~$
```

Après ça nous avons Dé-zippé le fichier contenant notre projet pour le mettre dans les dossiers choisis précédemment. Pour ce faire, il vous suffit d'entrer la commande suivante en fonction de votre méthode de compression:

```
unzip vote_fichier.zip
```

`tar -xzvf votre_fichier.tar.gz`

`unzip` pour décompresser un fichier `.zip` ou `tar -xzvf` pour un fichier `.tar.gz`



```
Terminal - administrateur@srv-dpi-proj-chiffrement-1: /var/www/html
Fichier  Édition  Affichage  Terminal  Onglets  Aide
freq.csv  index.html  liste_francais.txt  master.css  script.js  words_alpha.txt
administrateur@srv-dpi-proj-chiffrement-1: /var/www/html$ clear

administrateur@srv-dpi-proj-chiffrement-1: /var/www/html$ ls
freq.csv  index.html  liste_francais.txt  master.css  script.js  words_alpha.txt
administrateur@srv-dpi-proj-chiffrement-1: /var/www/html$
```

Voilà, le serveur diffuse bien notre site. Pour le vérifier il vous suffit de taper dans votre barre de recherche `”https://www.srv-dpi-proj-chiffrement-1.univ-rouen.fr”` pour pouvoir y accéder.

4 Problèmes rencontrés

4.1 Automatiser la substitution

Lors de ce projet, nous avons bloqué sur un gros problème : l'automatisation du déchiffrement de la partie substitution. C'est un domaine que nous ne connaissions pas. Après de multiples essais, nous sommes arrivés à un prototype fonctionnel. De plus, lors du développement, nous avons eu des soucis concernant les dictionnaires. En effet, dans les dictionnaires que nous utilisions au départ, il n'y avait pas de verbes conjugués ou de noms propres par exemple. C'est pourquoi, nous avons créé notre propre dictionnaire.

4.2 Expérience utilisateur

Lors de la conception générale et du design du site web, nous avons cherché à avoir un site fonctionnel et harmonieux. Nous avons donc beaucoup travaillé le CSS pour avoir une clarté (mettant le texte lisible) via les couleurs et une ergonomie par la disposition du site.

Également, il a fallu gérer toutes les interactions que l'utilisateur peut avoir avec le site, les boutons, aires de textes etc... À chaque fois que l'utilisateur entre une information, texte ou autre, il faut vérifier si celle-ci est correcte avant de traiter la demande.

Au final, la partie graphique et les interactions avec les utilisateurs ont pris au moins autant de temps que le développement des algorithmes d'attaques et tout ce qu'il y a autour.

5 Améliorations possibles

Nous aurions pu enlever les espaces aux textes chiffrés. Cela aurait rendu le déchiffrement par substitution et par César plus complexe. En effet, notre algorithme d'attaque pour César utilise la vérification des mots dans le dictionnaire pour valider le déchiffrement. Ainsi, sans les espaces, il serait plus difficile de valider si une clé est la bonne ou non.

6 Conclusion

Ce projet nous a permis d'apprendre à travailler à plusieurs en équipe. Nous nous réunissions régulièrement le jeudi matin. C'était l'occasion de voir où on en était, de réunir ce que chacun avait fait et de travailler ensemble. C'était également l'occasion de faire un point avec notre enseignante référente Magali Bardet.

Il nous a permis aussi de découvrir ou d'en apprendre plus sur la cryptographie, les différents chiffrements et méthodes d'attaque. On a pu voir à quel point un chiffrement aussi simple que la substitution (simple par la manière de chiffrer), peut être très difficile à attaquer. Cette première approche nous resservira sûrement étant donné que nous pensons, pour la plupart, poursuivre vers un master SSI.