Codage et Cryptographie
Algorithme du Solitaire

Leestmans Richard

31 Mars 2023

Le programme de cryptage du Solitaire a été réalisé en C++.

1 Introduction

Le but de ce rapport est de présenter l'algorithme de cryptographie dit du Solitaire ainsi que l'application mise en place permettant de crypter et de décrypter des messages en utilisant ce système.

Le système de cryptographie du Solitaire est un système de codage imaginé par Bruce Schneier et permettant, à partir d'un paquet de 54 cartes, de générer une clé permettant de crypter ou décrypter un message secret selon que nous sommes le créateur ou le destinataire du message. Pour que le système fonctionne, il faut que le créateur et le destinataire des messages possèdent initialement la même version du paquet mélangé. De plus, pour ne pas casser le système de cryptage, l'émetteur des messages doit veiller à ne pas utiliser deux fois la même clé afin de garder le système de cryptage inviolable. En effet, si le créateur des messages vient à utiliser la même clé, le système de cryptage des messages peut être craqué en soustrayant deux messages cryptés afin de reconstituer le message original. Pour éviter cela, un algorithme précis permettant de mélanger les cartes du paquet, afin de générer une clé unique, est mis en place et son fonctionnement est présenté ci-après.

2 Algorithme de mélange des cartes du paquet permettant de générer des clés uniques

Afin d'illustrer l'implémentation des différentes parties de l'algorithme dans notre programme, chaque étape sera accompagnée du code C++ correspondant.

2.1 Génération du paquet

Le paquet servant à générer un flux de clé est un jeu de cartes classique composé de 52 cartes comportant 4 couleurs (Carreau, Trèfle, Cœur et Pique) et ayant des valeurs allant d'As à Roi. À ces 52 cartes, il faut rajouter 2 jokers (un joker rouge et un joker noir). Ces deux dernières cartes seront essentielles pour le bon fonctionnement de notre algorithme. Chaque carte possède un numéro qui lui est propre, à l'exception des deux jokers qui possèdent le même numéro (53). Les numéros des cartes restantes sont déterminés selon l'ordre du Bridge: *Trèfle - Carreau - Cœur - Pique* et par ordre croissant de valeur des cartes de même couleur. Ainsi, l'as de Trèfle aura le numéro *1* et le roi de Pique aura le numéro *52*.

Une fois les 54 cartes générées, il faut mélanger le paquet aléatoirement. Dans notre programme, nous avons utilisé la fonction *shuffle* permettant de mélanger le contenu d'un tableau à partir d'une *seed* générée aléatoirement. Pour la suite de la démonstration de l'algorithme, nous utiliserons le paquet suivant:

35 29 39 7 53 34 5 38 21 36 15 23 3 32 37 20 31 41 4 11 26 8 33 50 40 25 51 13
22 43 30 6 24 1 27 48 2 10 16 14 19 49 9 47 28 52 17 53 12 46 45 44 42 18

Afin de visualiser plus facilement le contenu du paquet, nous le représenterons par la suite par les noms de cartes simplifiés suivants:

9H 3H RH 7T BlackJoker 8H 5T DH 8C 10H 2C 10C 3T 6H VH 7C 5H 2P 4T VT RC 8T 7H VP AP DC DP RT 9C 4P 4H 6T VC AT AH 9P 2T 10T 3C AC 6C 10P 9T 8P 2H RP 4C RedJoker DT 7P 6P 5P 3P 5C

Les noms simplifiés des cartes correspondent, à l'exception des deux jokers, à la valeur de la carte suivie de sa couleur (P → Pique, C → Carreau, H → Cœur, T → Trèfle). En tenant compte de ces informations, la première carte de notre paquet, par exemple, est donc le *9 de Cœur* et la dernière le *5 de Carreau*. À partir de ce point, le créateur doit transmettre le contenu de ce paquet au destinataire afin de permettre à ce dernier de décoder les messages dans le futur. Les étapes de l'algorithme permettant de générer un flux de clés permettant de crypter un message seront réalisées sur le paquet de l'émetteur des messages. Les étapes permettant de générer un flux de clés pour décrypter un message sont exactement les mêmes mais seront réalisées sur le paquet de cartes du destinataire.

2.2 Première étape de l'algorithme de mélange du paquet de cartes

La première étape de l'algorithme consiste à faire reculer le joker noir d'une place dans le paquet de cartes. Si le joker noir est la dernière carte du paquet, alors nous le faisons passer à la seconde place dans le paquet.

En reprenant notre exemple, on recherche la position du joker noir dans le paquet de carte. Comme il n'est pas au fond de notre paquet (il est en 5^{eme} position), nous le faisons donc reculer d'une place dans le paquet. Cela revient à échanger sa position avec le *8 de Cœur* situé juste après.

Après cette première étape, nous obtenons le paquet de carte suivant:

9H 3H RH 7T 8H BlackJoker 5T DH 8C 10H 2C 10C 3T 6H VH 7C 5H 2P 4T VT RC 8T 7H VP AP DC DP RT 9C 4P 4H 6T VC AT AH 9P 2T 10T 3C AC 6C 10P 9T 8P 2H RP 4C RedJoker DT 7P 6P 5P 3P 5C

```
1  void Deck::firstStep()
2  {
3      //On récupère la position du joker noir
4      std::list<Card>::iterator blackJokerPos = this->findCardByName("BlackJoker");
5      //On récupère la dernière carte du deck
6      Card lastCard = this->m_cardsDeck->back();
7
8      //Si le joker noir est la dernière carte du deck
9      if((*blackJokerPos).getId() == lastCard.getId())
10     {
11         //On stocke le joker noir dans une variable temporaire
12         Card tempJoker = *blackJokerPos;
13         //On supprime le joker noir de sa position actuelle
14         this->m_cardsDeck->erase(blackJokerPos);
15         //et on l'insère à la position 2 dans le deck
16         std::list<Card>::iterator newPos = this->m_cardsDeck->begin();
17         std::advance(newPos, 1);
18         this->m_cardsDeck->insert(newPos, tempJoker);
19     }
20     //Sinon le joker noir n'est pas à la dernière position dans le deck
21     else
22     {
```

```
        //On stocke le joker noir dans une variable temporaire
        Card tempJoker = *blackJokerPos;
        //On supprime le joker noir de sa position actuelle
        std::list<Card>::iterator pos = this->m_cardsDeck->erase(blackJokerPos);
        //et on l'insère après la carte qui le suivait dans le deck
        //cela revient à échanger la position du joker noir avec la carte qui le suit directement
        std::advance(pos, 1);
        this->m_cardsDeck->insert(pos, tempJoker);
    }
}
```

Code 1: *Implémentation en C++ de la première étape de mélange du paquet de cartes de l'algorithme du Solitaire.*

2.3 Seconde étape de l'algorithme de mélange du paquet de cartes

Dans la seconde étape de l'algorithme, nous devons faire reculer le joker rouge de deux places dans le paquet de cartes. Si ce dernier est en avant-dernière position, nous le faisons passer en seconde position dans le paquet. Si le joker rouge est en dernière position dans le paquet de cartes, nous le faisons passer en troisième position dans le paquet.

En reprenant notre exemple, nous cherchons la position du joker rouge dans le paquet de cartes. Comme il n'est ni en avant-dernière ni en dernière position (il est à la 48^{eme} place), nous le faisons donc reculer de deux places, ce qui correspond à le placer après le *7 de Pique* dans le paquet de cartes. Après cette étape, nous avons le paquet suivant:

9H 3H RH 7T 8H BlackJoker 5T DH 8C 10H 2C 10C 3T 6H VH 7C 5H 2P 4T VT RC 8T 7H VP AP DC DP RT 9C 4P 4H 6T VC AT AH 9P 2T 10T 3C AC 6C 10P 9T 8P 2H RP 4C DT 7P RedJoker 6P 5P 3P 5C

```
void Deck::secondStep()
{
    //On récupère la position du joker rouge
    std::list<Card>::iterator redJokerPos = this->findCardByName("RedJoker");
    //On récupère la dernière carte du deck
    Card lastCard = this->m_cardsDeck->back();
    //Ainsi que l'avant dernière carte du deck
    Card penultimateCard = *(std::prev(std::prev(this->m_cardsDeck->end())));

    //Si le joker rouge est la dernière carte du deck
    if((*redJokerPos).getId() == lastCard.getId())
    {
        //On stocke le joker rouge dans une variable temporaire
        Card tempJoker = *redJokerPos;
        //On supprime le joker rouge de sa position actuelle
        this->m_cardsDeck->erase(redJokerPos);
        //et on l'insère à la position 3 dans le deck
        std::list<Card>::iterator newPos = this->m_cardsDeck->begin();
        std::advance(newPos, 2);
        this->m_cardsDeck->insert(newPos, tempJoker);
    }
    //Sinon si le joker rouge est l'avant dernière carte du deck
    else if((*redJokerPos).getId() == penultimateCard.getId())
    {
        //On stocke le joker rouge dans une variable temporaire
        Card tempJoker = *redJokerPos;
        //On supprime le joker rouge de sa position actuelle
```

```
28          this->m_cardsDeck->erase(redJokerPos);
29          //et on l'insère à la position 2 dans le deck
30          std::list<Card>::iterator newPos = this->m_cardsDeck->begin();
31          std::advance(newPos, 1);
32          this->m_cardsDeck->insert(newPos, tempJoker);
33      }
34      //Sinon le joker rouge n'est ni à l'avant dernière ni à la dernière position dans
        le deck
35      else
36      {
37          //On stocke le joker rouge dans une variable temporaire
38          Card tempJoker = *redJokerPos;
39          //On supprime le joker rouge de sa position actuelle
40          std::list<Card>::iterator pos = this->m_cardsDeck->erase(redJokerPos);
41          //et on l'insère deux places plus loin dans le deck
42          std::advance(pos, 2);
43          this->m_cardsDeck->insert(pos, tempJoker);
44      }
45  }
```

Code 2: *Implémentation en C++ de la seconde étape de mélange du paquet de cartes de l'algorithme du Solitaire.*

2.4 Troisième étape de l'algorithme de mélange du paquet de cartes

Dans cette troisième étape, nous devons effectuer une coupe du paquet de cartes en fonction de la position des deux jokers. En d'autres termes, cela revient à intervertir l'ensemble des cartes situées avant le premier joker avec l'ensemble des cartes situées après le second joker.

Dans notre exemple, nous repairons tout d'abord la position des deux jokers. Puis nous allons extraire la sous-séquence de cartes situées avant le premier joker (ici avant le joker noir *en orange*) et celle située après le second joker (ici le joker rouge *en vert*):

9H 3H RH 7T 8H BlackJoker 5T DH 8C 10H 2C 10C 3T 6H VH 7C 5H 2P 4T VT RC 8T 7H VP AP DC DP RT 9C 4P 4H 6T VC AT AH 9P 2T 10T 3C AC 6C 10P 9T 8P 2H RP 4C DT 7P RedJoker 6P 5P 3P 5C

Nous intervertissons les deux sous-séquences dans le paquet de cartes en mettant la sous-séquence verte au début du paquet et la sous-séquence orange à la fin. Après la troisième étape de l'algorithme, nous obtenons la configuration de paquet suivante:

6P 5P 3P 5C BlackJoker 5T DH 8C 10H 2C 10C 3T 6H VH 7C 5H 2P 4T VT RC 8T 7H VP AP DC DP RT 9C 4P 4H 6T VC AT AH 9P 2T 10T 3C AC 6C 10P 9T 8P 2H RP 4C DT 7P RedJoker 9H 3H RH 7T 8H

```
1   void Deck::thirdStep()
2   {
3       //On récupère la position des deux jokers
4       std::list<Card>::iterator blackJokerPos = this->findCardByName("BlackJoker");
5       std::list<Card>::iterator redJokerPos = this->findCardByName("RedJoker");
6
```

```
7       //On calcule la distance relative entre les deux jokers pour savoir où sont
    placer les deux jokers
8       //l'un par rapport à l'autre et donc savoir comment couper dans le paquet de
    cartes
9       int distance = std::distance(this->m_cardsDeck->begin(), blackJokerPos) - std::
    distance(this->m_cardsDeck->begin(), redJokerPos);
10      //Si la distance est positive, le joker rouge est situé avant le joker noir
11      if(distance > 0)
12      {
13          //On récupère le second fragment situé entre le joker noir non compris et la
    fin du deck
14          std::advance(blackJokerPos, 1);
15          std::list<Card> secondFragment;
16          secondFragment.splice(secondFragment.begin(), *this->m_cardsDeck,
    blackJokerPos, this->m_cardsDeck->end());
17          //On récupère le premier fragment situé entre le début du deck et le joker
    rouge non compris
18          std::list<Card> firstFragment;
19          firstFragment.splice(firstFragment.begin(), *this->m_cardsDeck, this->
    m_cardsDeck->begin(), redJokerPos);
20          //Puis on insère le second fragment à la place du premier et le premier à la
    place du second dans le deck
21          this->m_cardsDeck->insert(this->m_cardsDeck->begin(), secondFragment.begin(),
     secondFragment.end());
22          this->m_cardsDeck->insert(this->m_cardsDeck->end(), firstFragment.begin(),
    firstFragment.end());
23      }
24      //Sinon c'est l'inverse (le joker noir est situé avant le joker rouge dans le
    deck)
25      else
26      {
27          //On récupère le second fragment situé entre le joker rouge non compris et la
     fin du deck
28          std::advance(redJokerPos, 1);
29          std::list<Card> secondFragment;
30          secondFragment.splice(secondFragment.begin(), *this->m_cardsDeck, redJokerPos
    , this->m_cardsDeck->end());
31          //On récupère le premier fragment situé entre le début du deck et le joker
    noir non compris
32          std::list<Card> firstFragment;
33          firstFragment.splice(firstFragment.begin(), *this->m_cardsDeck, this->
    m_cardsDeck->begin(), blackJokerPos);
34          //Puis on insère le second fragment à la place du premier et le premier à la
    place du second dans le deck
35          this->m_cardsDeck->insert(this->m_cardsDeck->begin(), secondFragment.begin(),
     secondFragment.end());
36          this->m_cardsDeck->insert(this->m_cardsDeck->end(), firstFragment.begin(),
    firstFragment.end());
37      }
38  }
```

Code 3: *Implémentation en C++ de la troisième étape de mélange du paquet de cartes de l'algorithme du Solitaire.*

2.5 Quatrième étape de l'algorithme de mélange du paquet de cartes

La quatrième étape de l'algorithme permet d'effectuer une coupe simple en fonction du numéro de la dernière carte du paquet de cartes.

Dans notre exemple, nous regardons donc la dernière carte du paquet qui est ici le *8 de Cœur* et nous récupérons son numéro (*34*). Nous allons extraire du paquet la sous-séquence composée des 34 premières cartes du paquet (*en vert*), et les placer au fond du paquet juste avant la dernière carte, ce qui revient à les insérer entre le *7 de Trèfle* (*en orange*) et le *8 de Cœur* (*en rouge*):

6P 5P 3P 5C BlackJoker 5T DH 8C 10H 2C 10C 3T 6H VH 7C 5H 2P 4T VT RC 8T 7H VP AP DC DP RT 9C 4P 4H 6T VC AT AH 9P 2T 10T 3C AC 6C 10P 9T 8P 2H RP 4C DT 7P RedJoker 9H 3H RH 7T 8H

Après la quatrième étape de l'algorithme, le paquet de cartes mélangé est le suivant:

9P 2T 10T 3C AC 6C 10P 9T 8P 2H RP 4C DT 7P RedJoker 9H 3H RH 7T 6P 5P 3P 5C BlackJoker 5T DH 8C 10H 2C 10C 3T 6H VH 7C 5H 2P 4T VT RC 8T 7H VP AP DC DP RT 9C 4P 4H 6T VC AT AH 8H

```cpp
void Deck::fourthStep()
{
    //On récupère le numéro de la dernière carte
    int n = this->m_cardsDeck->back().getId();
    //On regarde si la dernière carte n'est pas un joker (numéro de carte: 53).
    //Si c'est le cas, la coupe n'est pas nécessaire car cela ne change pas la configuration du deck
    if(n != 53)
    {
        //On récupère les n premières cartes du deck
        //On crée un itérateur qui va pointer sur la carte à l'indice n dans la liste
        std::list<Card>::iterator it = this->m_cardsDeck->begin();
        std::advance(it, n);
        std::list<Card> fragment;
        fragment.splice(fragment.begin(), *this->m_cardsDeck, this->m_cardsDeck->begin(), it);
        //Et on insère ces cartes juste avant la dernière carte du deck
        this->m_cardsDeck->insert(std::prev(this->m_cardsDeck->end()), fragment.begin(), fragment.end());
    }
}
```

Code 4: *Implémentation en C++ de la quatrième étape de mélange du paquet de cartes de l'algorithme du Solitaire.*

2.6 Cinquième étape de l'algorithme de mélange du paquet de cartes

Lors de la cinquième et dernière étape de l'algorithme, nous allons commencer par récupérer le numéro de la première carte du paquet que nous noterons n. Puis nous allons avancer de n cartes dans le paquet afin de récupérer le numéro de la $n+1^{ème}$ carte que nous noterons m. Si la $n+1^{ème}$ carte est un joker, c'est-à-dire que son numéro est égal à *53*, alors nous recommençons une opération complète de mélange depuis l'étape 1 de l'algorithme. Sinon, nous gardons le numéro de cette carte. Si ce dernier est supérieur strictement à 26, alors nous lui soustrayons 26. Enfin, nous transformons ce nombre en lettre, en lui faisant correspondre la lettre de l'alphabet en fonction du résultat obtenu (A = 1, B = 2,..., Z = 26).

Dans notre exemple, la première carte du paquet est le *9 de Pique* (*en orange*). Le numéro associé à cette carte est le *48*. Nous avançons donc dans le paquet de 48 cartes et nous regardons le numéro de la 49ème. Il s'agit ici du *4 de Cœur* (*en rouge*).

9P 2T 10T 3C AC 6C 10P 9T 8P 2H RP 4C DT 7P RedJoker 9H 3H RH 7T 6P 5P 3P 5C BlackJoker 5T DH 8C 10H 2C 10C 3T 6H VH 7C 5H 2P 4T VT RC 8T 7H VP AP DC DP RT 9C 4P 4H 6T VC AT AH 8H

Comme ce n'est pas un joker, nous notons son numéro (*30*). Ce dernier étant supérieur strictement à 26, nous y soustrayons 26 et nous obtenons *4*. Nous y associons la lettre située à la 4ème place dans l'alphabet, c'est-à-dire la lettre *D*. Enfin, nous ajoutons cette lettre au flux de clés permettant de crypter un message.

L'exemple que nous venons de réaliser avec le mélange du paquet de cartes nous permet de crypter un message de *taille 1*, c'est-à-dire contenant un caractère car la méthode de cryptage du Solitaire a besoin d'une clé de même taille que le message pour pouvoir le crypter. Pour générer une lettre à ajouter dans le flux de clés, nous devons faire au minimum une opération entière de mélange du paquet de cartes. Ainsi, pour pouvoir crypter un message de n caractères, nous devrions effectuer au minimum n opérations de mélange. L'avantage du flux de clés est que nous pouvons moduler la taille de la clé à générer à celle du message à crypter: il suffit uniquement de faire varier le nombre d'opérations de mélange du paquet de cartes.

Enfin, lorsque nous faisons une opération de mélange numéro p, nous partons du paquet de cartes obtenu à l'opération numéro p-*1*. Nous ne repartons jamais du paquet généré initialement et cela aura son importance, notamment lors du décryptage des messages.

```
1  std::string Deck::createKeyStream(int size)
2  {
3      //On crée une chaine de caractères qui contiendra la clé servant à coder le
       message
4      std::string keyStream = "";
5      //ainsi qu'un chaine contenant l'alphabet
6      std::string alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
7      //On crée un compteur sur la taille du mot passé en paramètre
8      int count = 0;
9      do{
10         //On crée un booléen pour savoir si la carte obtenue à la cinquième étape est
        ou non un joker
11         bool isAJoker;
12         do{
13             isAJoker = false;
14             //On effectue les quatre premières opérations pour obtenir le flux de
       clefs
15             this->firstStep();
16             this->secondStep();
17             this->thirdStep();
18             this->fourthStep();
19             //Cinquième étape -> lecture d'une lettre pseudo-aléatoire
20             //On récupère le numéro de la première carte du deck
21             int n = this->m_cardsDeck->front().getId();
22             //On crée un itérateur que l'on fait pointer sur la n+1-ième carte du
       deck
23             std::list<Card>::iterator it = this->m_cardsDeck->begin();
24             std::advance(it, n);
25             //On récupère le numéro de la n+1-ième carte du deck
26             int m = (*it).getId();
27             //Si c'est un joker (numéro de carte = 53), on recommence une opération
       complète de mélange (on repart à la première étape)
28             if(m == 53)
29                 isAJoker = true;
30             else
31             {
32                 //Si le numéro de carte dépasse 26, on le soustrait par 26
```

```
                if(m > 26)
                    m -= 26;
                //et on l'ajoute à la liste de flux de clés
                keyStream += alphabet[m-1];
            }
        }while(isAJoker);
        //On incrémente le compteur
        count++;
    }while(count < size);
    return keyStream;
}
```

Code 5: *Implémentation en C++ de la génération du flux de clés en fonction de la taille du message à crypter ou à décrypter. Cette fonction comprend les cinq étapes de mélange du paquet de cartes incluses dans une boucle tant que l'on tombe sur un joker à la cinquième étape de l'algorithme.*

3 Cryptage et décryptage des messages

Pour montrer le cryptage et le décryptage, la démonstration sera faite à l'aide du message: "UN SIMPLE MESSAGE".

3.1 Cryptage

Pour crypter notre message, nous avons besoin d'une clé de cryptage de même taille que notre message. Nous avons vu précédemment que cette clé était générée en effectuant un certain nombre d'opérations de mélange du paquet de cartes. Comme notre message fait 17 caractères, nous devons générer une clé de taille 17. De plus, les espaces, sont pris en compte dans le cryptage et le décryptage des messages. Afin de les considérer dans le message crypté, nous les remplaçons par la suite de caractères *"SZ"* non présente dans la langue française, ce qui porte la longueur du message à crypter à 19 caractères. Après avoir effectué au minimum 19 fois les cinq étapes de mélange du paquet de cartes présenté précédemment, nous obtenons la clé suivante:

DCJKNREALQZQHWWATOS

Afin de crypter le message à l'aide de cette clé, nous devons passer nos deux chaines de caractères (le message à crypter et la clé de cryptage) sous forme numérique suivant l'ordre des lettres dans l'alphabet (A = 1, B = 2, ..., Z = 26). Nous obtenons ainsi les messages, sous forme numérique, suivants:

U	N	␣	S	I	M	P	L	E	␣	M	E	S	S	A	G	E

⇓

Transcription en un message compréhensible par le programme

U	N	S	Z	S	I	M	P	L	E	S	Z	M	E	S	S	A	G	E
21	14	19	26	19	9	13	16	12	5	19	26	13	5	19	19	1	7	5

D	C	J	K	N	R	E	A	L	Q	Z	Q	H	W	W	A	T	O	S
4	3	10	11	14	18	5	1	12	17	26	17	8	23	23	1	20	15	19

Puis nous effectuons la somme terme à terme des valeurs des deux chaines de nombres.

21	14	19	26	19	9	13	16	12	5	19	26	13	5	19	19	1	7	5
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
4	3	10	11	14	18	5	1	12	17	26	17	8	23	23	1	20	15	19

⇓

25	17	29	37	33	27	18	17	24	22	45	43	21	28	42	20	21	22	24

⇓

Si la somme dépasse 26, nous les soustrayons par 26 (*en rouge*), afin d'obtenir des valeurs comprises entre 1 et 26.

25	17	3	11	7	1	18	17	24	22	19	17	21	2	16	20	21	22	24

⇓

Retranscription de la chaine de nombres en chaine de caractères

Y	Q	C	K	G	A	R	Q	X	V	S	Q	U	B	P	T	U	V	X

Nous retranscrivons la chaine de nombres résultante en chaine de caractères pour obtenir le message crypté.

3.2 Décryptage

Afin de décrypter le message obtenu ci-dessus, nous devons tout d'abord générer une clé de décryptage. Comme pour le cryptage, la clé de décryptage doit être de même taille que le message. De plus, nous avions défini dans l'introduction que pour le système de codage fonctionne, l'émetteur des messages et le destinataire devaient avoir le même paquet de cartes initial. Connaissant cela, comme les paquets de cartes du créateur et du destinataire des messages sont identiques et que les clés générées doivent être de même taille que le message à crypter ou à décrypter, par conséquent, la clé de décryptage générée par le flux de clés suite au mélange du paquet de cartes du destinataire sera identique à celle ayant permis de crypter le message initialement soit *DCJKNREALQZQHWWATOS*.

Le décryptage revient à effectuer les étapes de cryptage dans l'autre sens:

Y	Q	C	K	G	A	R	Q	X	V	S	Q	U	B	P	T	U	V	X
25	17	3	11	7	1	18	17	24	22	19	17	21	2	16	20	21	22	24

D	C	J	K	N	R	E	A	L	Q	Z	Q	H	W	W	A	T	O	S
4	3	10	11	14	18	5	1	12	17	26	17	8	23	23	1	20	15	19

Mais cette fois-ci, nous connaissons le message crypté et la clé de décryptage et nous recherchons le message initial. Par conséquent, au lieu de faire l'addition terme à terme, nous allons soustraire les valeurs de la chaine de nombres du message crypté par celle de la clé. Si le résultat obtenu est inférieur strictement à *1*, alors nous l'additionnons avec 26 de sorte d'avoir des valeurs comprises entre *1* et *26*:

25	17	3	11	7	1	18	17	24	22	19	17	21	2	16	20	21	22	24
−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−
4	3	10	11	14	18	5	1	12	17	26	17	8	23	23	1	20	15	19

⇓

21	14	-7	0	-7	-17	13	16	12	5	-7	0	13	-21	-7	19	1	7	5

⇓

On ajoute 26 aux valeurs qui sont strictement inférieures à 1 (*en rouge*).

21	14	19	26	19	9	13	16	12	5	19	26	13	5	19	19	1	7	5

⇓

Retranscription de la chaine de nombres en chaine de caractères

U	N	S	Z	S	I	M	P	L	E	S	Z	M	E	S	S	A	G	E

⇓

Traduction de la chaine de caractères en langage naturel compréhensible par l'Homme (voir *Code 6*).

U	N	␣	S	I	M	P	L	E	␣	M	E	S	S	A	G	E

Nous avons donc obtenu le message qui nous avions crypté initialement.

```
std::string& EncryptAndDecrypt::addDotsAndSpaces(std::string& message)
{
    size_t found = 0;
    //On remplace toutes les suites de caractères "SZ" par un espace
    do{
        found = message.find("SZ", found);
        if(found != std::string::npos)
        {
            message.replace(found, 2, " ");
            found++;
        }
    }while(found != std::string::npos);

    //On fait pareil pour les points
    found = 0;
    do{
        found = message.find("STOP", found);
        if(found != std::string::npos)
        {
            message.replace(found, 4, ".");
            found += 2;
        }
    }while(found != std::string::npos);

    return message;
}
```

Code 6: *Fonction permettant de transformer un message (sous la forme d'une chaine de caractères) en entrée en message compréhensible par l'Homme en remplaçant les suites de caractères "SZ" par des espaces et les suites de caractères "STOP" par des points.*

3.3 Génération du flux de clés et importance de l'ordre de décryptage des messages

Nous avons vu dans l'introduction que, pour que le système reste infaillible, il fallait que les clés servant à crypter les messages soient uniques. De plus, nous avons vu que la clé servant à crypter le message est la même que celle qui sert à décrypter le message.

Nous pouvons en déduire que *"Pour tout message crypté, il existe une seule et unique clé permettant de le décrypter identique à celle ayant crypté le message."*. Et cela vaut pour n'importe quel message, quel que soit sa taille ou son contenu. Dans le cas où le rédacteur des messages en créerait deux de tailles différentes, il est assez évident que si on inverse l'ordre de décryptage des messages par le destinataire, la première clé ayant été générée par le paquet du rédacteur sera nécessairement différente de la première clé générée par le paquet du destinataire car de tailles différentes.

Prenons maintenant le cas de deux messages de même taille et poussons le raisonnement plus loin en prenant le même contenu *"UN SIMPLE MESSAGE"* pour les deux messages. Le premier message

va être crypté en le message suivant: *YQCKGARQXVSQUBPTUVX* en utilisant la clé *DCJKNREALQZQHWWATOS*, et le second va être crypté en le message *WASUDDWLQYIJFOEQQAA* en utilisant la clé *BMZUKUJVETPJSJLXPTV*. Maintenant, intervertissons l'ordre de décryptage et essayons de décrypter le second message, ce qui revient à décrypter le message *WASUDDWLQYIJFOEQQAA* en utilisant la clé *DCJKNREALQZQHWWATOS*:

W	A	S	U	D	D	W	L	Q	Y	I	J	F	O	E	Q	Q	A	A
23	1	19	21	4	4	23	12	17	25	9	10	6	15	5	17	17	1	1

D	C	J	K	N	R	E	A	L	Q	Z	Q	H	W	W	A	T	O	S
4	3	10	11	14	18	5	1	12	17	26	17	8	23	23	1	20	15	19

⇓

23	1	19	21	4	4	23	12	17	25	9	10	6	15	5	17	17	1	1
–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
4	3	10	11	14	18	5	1	12	17	26	17	8	23	23	1	20	15	19

⇓

19	-2	9	10	-10	-14	18	11	5	8	-17	-7	-2	-8	-18	16	-3	-14	-18

⇓

19	24	9	10	16	12	18	11	5	8	9	19	24	18	8	16	23	12	8

⇓

Retranscription de la chaine de nombres en chaine de caractères

S	X	I	J	P	L	R	K	E	H	I	S	X	R	H	P	W	L	H

Le message obtenu n'est pas du tout le même que *"UN SIMPLE MESSAGE"* attendu ce qui confirme l'importance de décrypter les messages dans le même ordre dans lequel ils ont été cryptés.

4 Conclusion

Nous avons donc vu comment crypter et décrypter des messages en utilisant la méthode dit du Solitaire en générant une clé à partir du mélange d'un paquet de cartes. Ce système est inviolable car chaque message crypté l'a été grâce à une clé de cryptage unique. Mais cette technique possède tout de même un point faible: les messages cryptés doivent être décryptés dans le même ordre sous peine de voir le destinataire du message ne pas obtenir le contenu escompté.