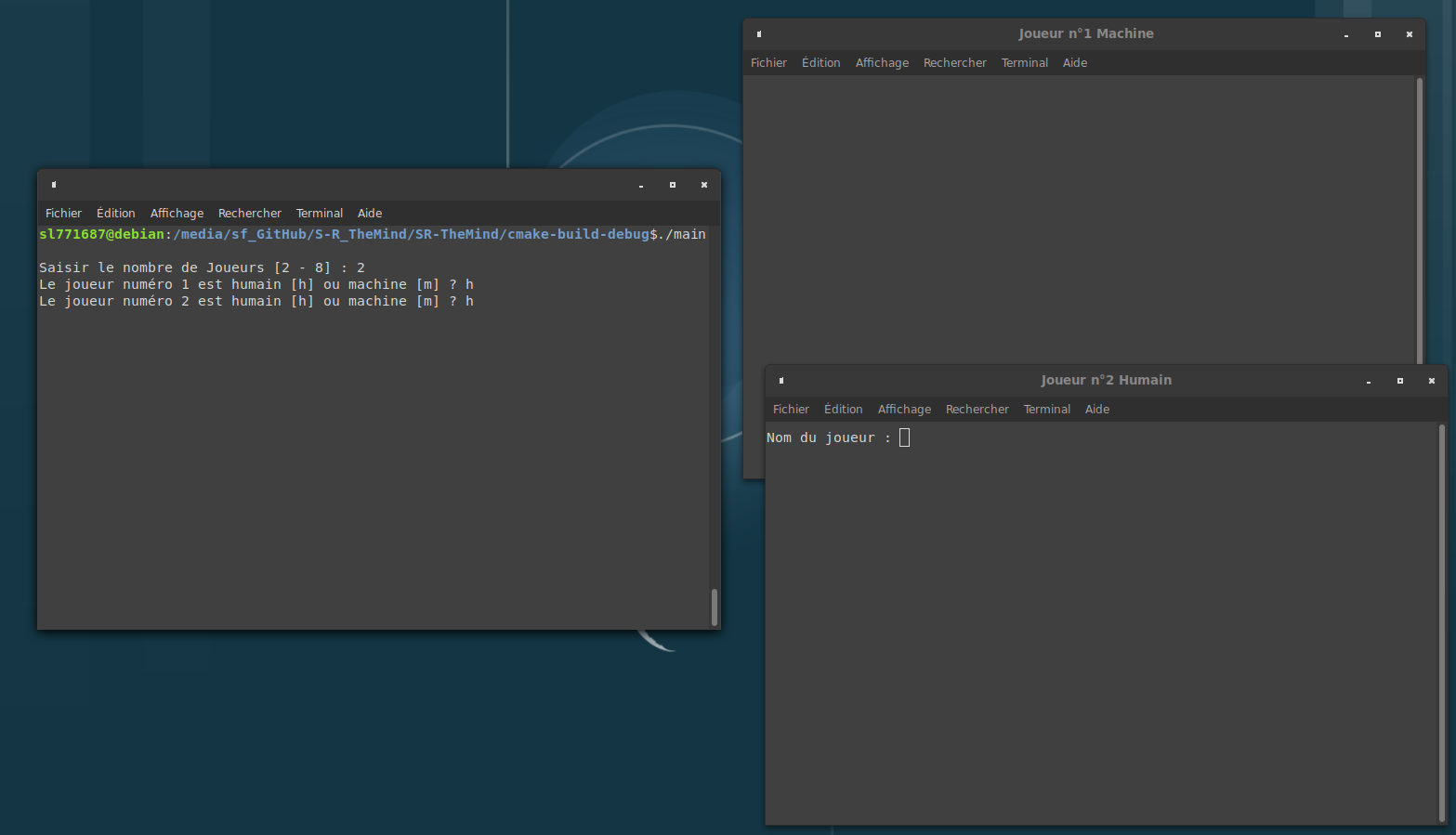


CryptoSharp

LACHAUD Samuel / PAZOLA Loïs – M1 Informatique



Une image contenant texte

Description générée automatiquement[](Installation/CDAA-R_UWP+EXE.mp4)

2/10

SOMMAIRE

Sur la partie de droite, nous retrouvons le décodage du message. L’utilisateur peut saisir ici le message à décoder. Le bouton « Decode » après avoir saisi l’ordre des cartes de la clé pour obtenir le message décodé qui sera visible dans la partie centrale.

Sur la partie centrale, nous retrouvons l’encodage du message. L’utilisateur peut saisir ici le message à encoder après avoir saisi l’ordre des cartes dans la première partie. Enfin il peut cliquer sur le bouton « Encode » afin d’encoder le message. Celui-ci sera alors visible dans la troisième partie.

Sur la partie de gauche, nous retrouvons une liste déroulante avec l’intégralité des 54 cartes du deck. L’utilisateur peut cliquer sur une carte et la déplacer avec une action de « glisser déposer ». Il y a également un bouton « Shuffle » permettant de mélanger aléatoirement le deck.

Il n’existe pour le moment aucune version compilée de CryptoSharp. Le projet a été réalisé sur Visual Studio 2022, il est donc nécessaire d’avoir cet IDE d’installé afin de lancer facilement notre projet. CryptoSharp est réalisé en C# (langage de programmation) + WPF (Framework graphique) et se présente comme une fenêtre de style Windows 10 où l’utilisateur pourra interagir avec le logiciel, et d’un terminal affichant des informations supplémentaires par rapport aux codages et décodages ainsi qu’a la génération de clé. Sur l’interface graphique, nous retrouvons 3 zones de contrôleurs :

Présentation de l’application

CryptoSharp est un projet de codage et cryptographie codé en C# portant sur l’encodage et le décodage d’un message à l’aide d’un paquet de cartes. La méthode utilisée est le solitaire de Bruce Schneier. Celle-ci permet un encodage facile à mettre en œuvre et très complexe à rompre. Le paquet de cartes utilisé est un jeu de 54 cartes (52 cartes + 2 Jokers). Il aurait également possible d’utiliser un jeu de Tarot (incluant les cavaliers) mais nous sortons du cadre conventionnel. Le projet est open source et trouvable à l’adresse suivante : <https://github.com/Mahtwo/CryptoSharp>

3/10

Qu'est-ce que CryptoSharp

3/10

I) Introduction



II) Fonctionnement de

l'application

Gestion des cartes et du deck

Une image contenant texte

Description générée automatiquement

Figure : Méthode MoveCard

Dans la modélisation du projet, le premier « Objet » qui est évident c’est le paquet de carte. C’est le centre de notre codage et il est donc primordial de le modéliser ainsi que l’ensemble des cartes qui le compose. Pour cela, nous devons créer une liste de cartes avec l’ensemble des valeurs qu’elles peuvent prendre. Nous avons choisi la notation anglaise pour leurs noms et nous avons utilisé une énumération pour les stocker (pas besoin d’une classe pour cela). Celle-ci se situe dans le fichier Card.cs.

Une fois cette énumération présente, nous pouvons coder les différentes méthodes permettant les manipulations sur le Deck (ensemble de toutes les valeurs possibles de l’énumération). Parmi celles-ci, nous retrouvons quelques fonctions basiques :

* Deck() // Constructeur qui initialise la LinkedList « cards » avec toutes les valeurs de l’énumération.
* Deck(LinkedList<Card> cards) // Constructeur qui initialise la LinkedList « cards » avec celle en paramètre.
* GetCard(int index) // Retourne la carte à l’index en paramètre dans la LinkedList.
* Shuffle() // Mélange aléatoirement la LinkedList
* FindCardPosition(Card c) // Retourne la position de la carte en paramètre dans la LinkedList
* ToString() // Retourne une string qui affiche l’ensemble des cartes et leur index.

Une fois les opérations basiques codées, nous pouvons nous attaquer aux fonctions permettant d’implémenter les différentes opérations nécessaires à l’encodage. Pour commencer, nous avons une fonction pour changer la position d’une carte : MoveCard(Card c, int direction).

4/10

6/10

6 / 9

Une image contenant texte

Description générée automatiquement

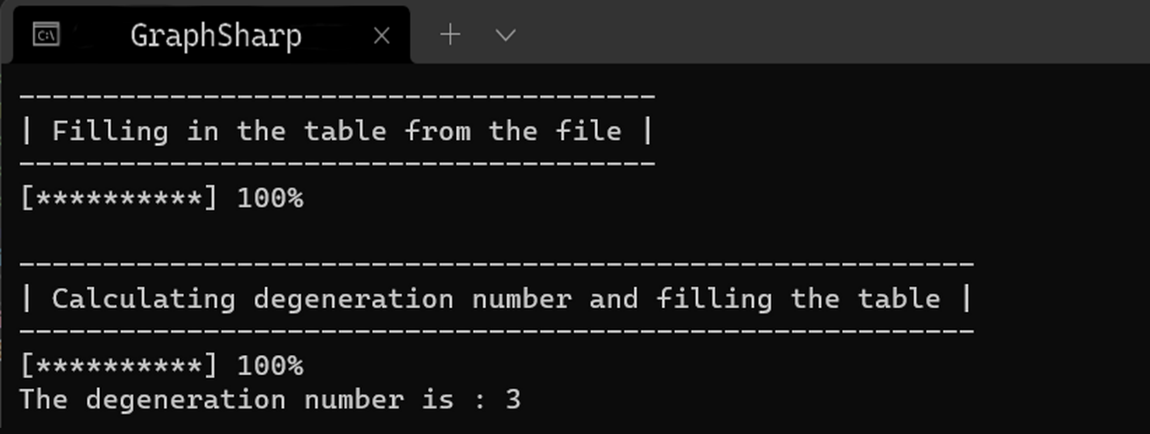
Cette fonction sert à déplacer une carte dans la liste de 1 ou plusieurs positions. Une position positive remontera la carte dans le paquet et une position négative la descendra dans le paquet. Le calcul de la nouvelle position utilise plusieurs modulos, c’est pour avoir des modulos fonctionnels avec des nombres négatifs (si on descend en dessous de la position 0 par exemple). On appelle cela un modulo canonique et ce n’est pas natif en C#. Le déplacement se fait de manière conventionnelle à part pour les bornes du deck.

En effet si on sort du dessus du deck, on ne se retrouve pas en dernière position, mais en avant dernière. De la même manière si on sort de la fin du deck, on ne retombe pas tout en haut du deck, mais en deuxième position. Cette fonction sert à réaliser les 2 premières opérations de la partie 3 du sujet :

*« 1 Recul du joker noir d’une position : Vous faites reculer le joker noir d’une place (vous le permutez avec la carte qui est juste derrière lui). Si le joker noir est en dernière position il passe derrière la carte du dessus (donc, en deuxième position).   
2 Recul du joker rouge de deux positions : Vous faites reculer le joker rouge de deux cartes. S’il était en dernière position, il passe en troisième position ; s’il était en avant dernière position il passe en deuxième. »*

La méthode suivante sert à réaliser la double coupe par rapport aux jokers : DoubleCutting()

3/11

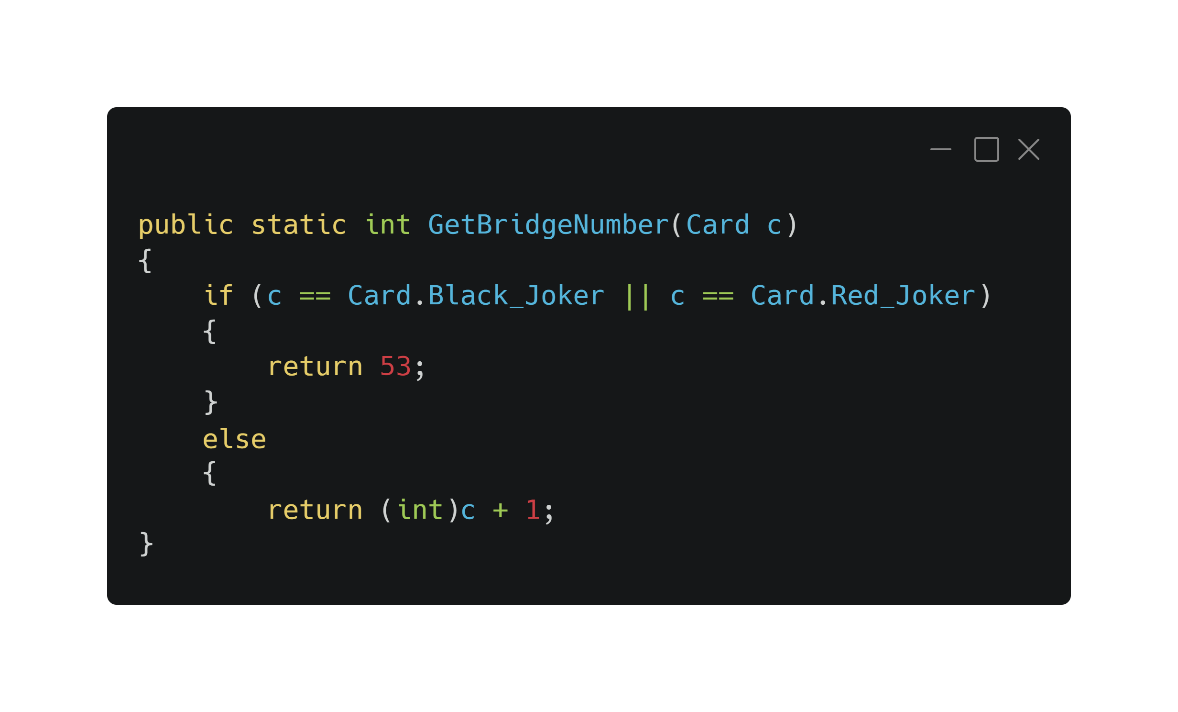
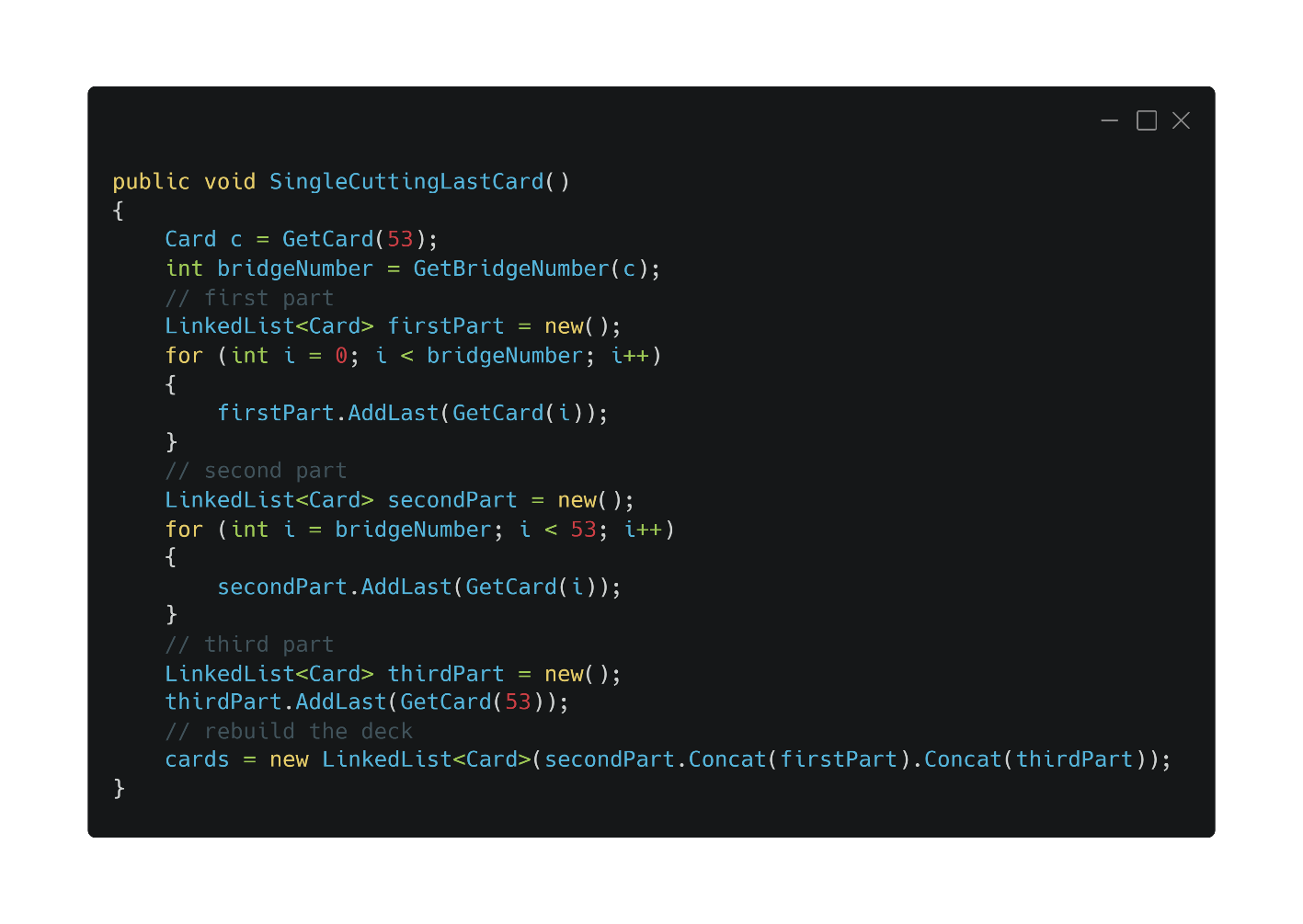


5/10

Figure : Méthode DoubleCutting

5/10

7/10



6/10

Figure 4 : Méthode SingleCuttingLastCard

Cette fonction toute bête nous sera très utile par la suite. Elle retourne simplement le numéro des cartes dans un deck dans l’ordre. L’ordre étant déjà le bon dans l’énumération, nous avons juste à récupérer le numéro de la carte dans celle-ci, sauf qu’elle commence à 0 et que les numéros du bridge commencent à 1. On rajoute juste 1 à cette valeur. Reste uniquement le cas des Joker, ceux-ci valent 53.

La fonction suivante nous sert à couper à partir de la dernière carte et de sa valeur de bridge : SingleCuttingLastCard()

Figure 3 : Méthode GetBridgeNumber

Comme dit précédemment, cette fonction sert à double couper le deck par rapport à la position des Jokers. On cherche donc d’abord les positions des deux Jokers avec les méthodes basiques épelées précédemment, puis on détermine celui qui a la position la plus haute, le Joker le plus proche du début du deck. On récupère le tas n°1 qui sont toutes les cartes entre les deux Jokers, puis le tas n°2 qui sont toutes les cartes entre le deuxième Joker et la fin du deck. Enfin, on intervertit ces deux tas. Notre double coupe est alors terminée. Cette fonction sert à réaliser la 3ème opération de la partie 3 du sujet :

*« 3* Double coupe par rapport aux jokers. Vous repérez les deux jokers et vous intervertissez le paquet des cartes situées au-dessus du joker qui est en premier avec le paquet de cartes qui est au-dessous du joker qui est en second. Dans cette opération la couleur des jokers est sans importance.*»*

La méthode suivante sert à récupérer le numéro du Bridge d’une carte : GetBridgeNumber(Card c)

Une image contenant texte

Description générée automatiquement

Cette fonction a pour but de récupérer la valeur de Bridge de la dernière carte du deck. Une fois cette valeur récupérée (que l’on appelle n par la suite), on prend donc les n première cartes puis on les met à l’avant dernière position du deck (en dessous du deck sauf qu’on garde la dernière carte à la fin). Cette fonction sert à réaliser la 4ème opération de la partie 3 du sujet :

*« 4 Coupe simple déterminée par la dernière carte : vous regardez la dernière carte et vous évaluez son numéro selon l’ordre du Bridge : trèfle-carreau-cœur-pique et dans chaque couleur as, 2, 3, 4, 5, 6, 7, 8, 9, 10, valet, dame et roi (l’as de trèfle a ainsi le numéro 1, le roi de pique a le numéro 52). Les jokers on par convention le numéro 53. Si le numéro de la dernière carte est n vous prenez les n premières cartes du dessus du paquet et les placez derrière les autres cartes à l’exception de la dernière carte qui reste la dernière »*

La prochaine méthode sert à vérifier le nombre de Bridge d’une carte à la position du nombre de Bridge d’une carte passée en paramètre et de vérifier si c’est un Joker : ReadingBridgeNumberNotJoker()

Une image contenant texte

Description générée automatiquement

Enfin, la dernière méthode sert à récupérer une lettre de l’alphabet à partir d’un nombre de Bridge : ReadingPseudoRandomLetter()

Figure 5 : Méthode ReadingBridgeNumberNotJoker



7/10

Pour cela on récupère le nombre de Bridge de la carte, puis le nombre de bridge de la carte à l’index du nombre de bridge précédent. Puis on force cette valeur entre 1 et 26 (Si c’est supérieur à 26, on retire 26). Le nombre ainsi obtenu sera une lettre de l’alphabet qui servira à créer la clé. Les deux fonctions précédentes servent à réaliser la 5ème et dernière opération de la partie 3 du sujet :

*« Lecture d’une lettre pseudo-aléatoire : Vous regardez le numéro de la première carte, soit n ce numéro. Vous comptez n cartes à partir du début et vous regardez la carte à laquelle vous êtes arrivé (la n + 1- i-ème), soit m son numéro. Si c’est un joker vous refaites une opération complète de mélange et de lecture (les points 1-2-3-4-5). Si m dépasse 26 vous soustrayez 26. Au nombre entre 1 et 26 ainsi obtenu est associée une lettre qui est la lettre suivante dans du flux de clefs. »*

Figure 6 : Méthode ReadingPseudoRandomLetter()

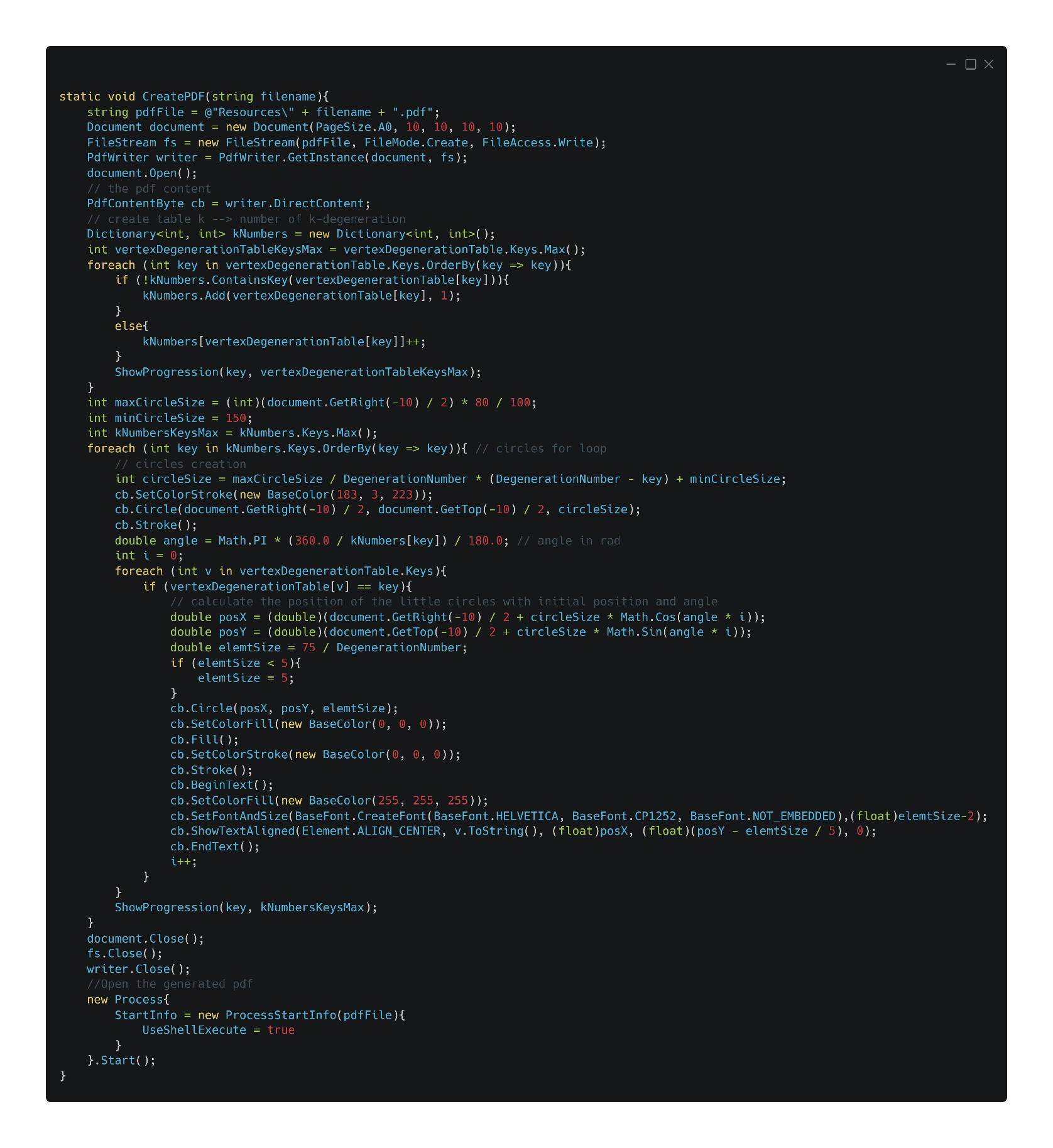
7/10



Génération d’un fichier PDF

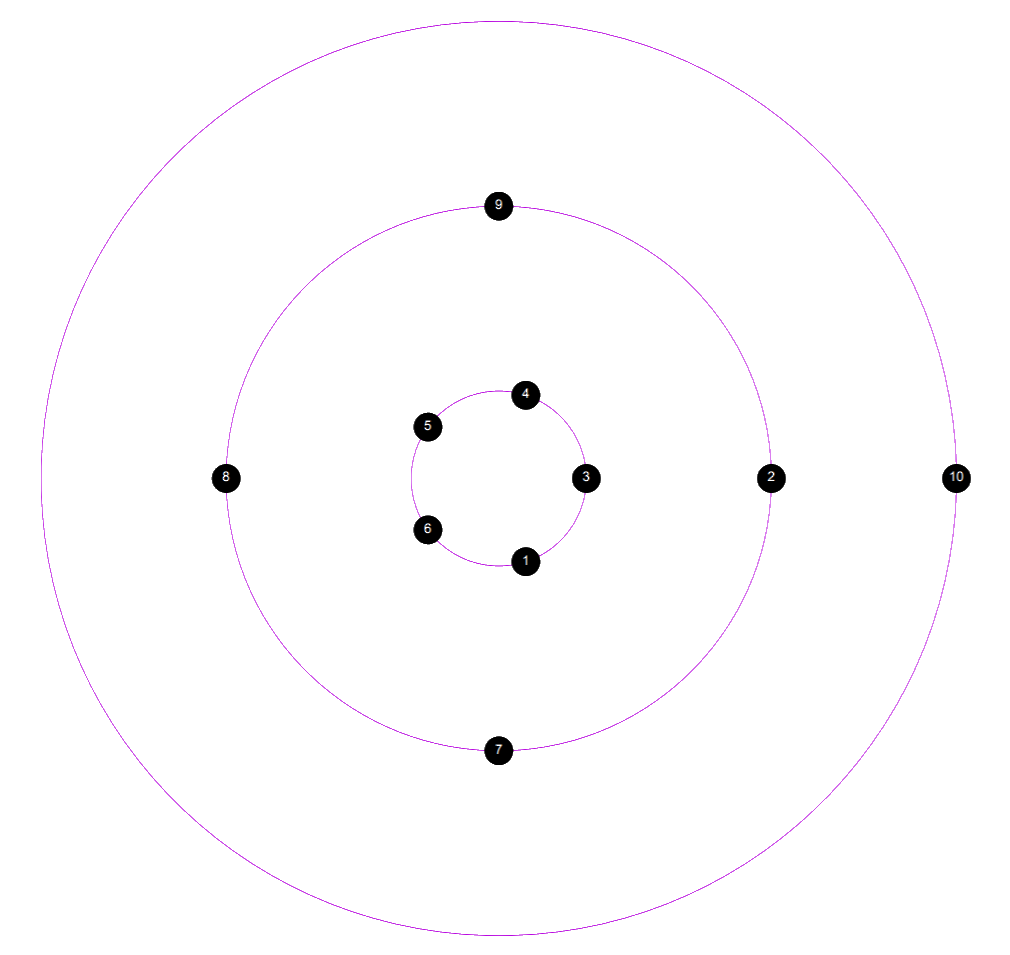
A partir de ce rendu, nous pouvons donc déduire que l’affirmation « on peux colorier un graphe k dégénéré en au plus k + 1 couleurs » est vrai dans notre cas d’observation. En effet la valeur **chromaticNumber** est toujours inférieure ou égale à la valeur **degenerationNumber** + 1.

On peut également supposer aux vues de ce rendu que l’écart entre la dégénérescence et le nombre chromatique est plutôt faible, mais que quelques exceptions sont tout de même présentes comme pour le fichier **Youtube.**



La dégénérescence de notre graphe est calculable, mais difficilement visualisable, en effet, on calcule la k-dégénérescence maximale, mais également la dégénérescence de chaque sommet du graphe. Nous pouvons donc visualiser ce graphe avec des cercles imbriqués, un cercle pour chaque k-dégénérescence. Le tout est mis dans un fichier PDF et chaque sommet est placé sur le cercle lui correspondant.

8/10



Le programme ouvre donc le fichier PDF généré, voici celui correspondant à l’exemple. Ainsi le cercle extérieur représente les sommets de dégénérescence 1 puis à l’intérieur le 2 et enfin le 3 :

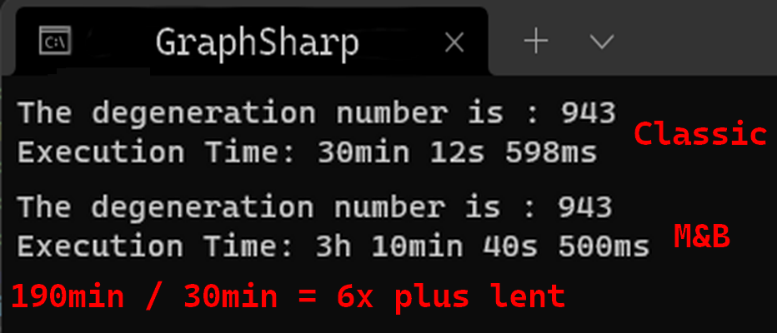
Algorithme de Matula & Beck

La dernière amélioration de notre programme est l’ajout de l’algorithme de Matula & Beck. En effet il permet également de calculer la dégénérescence d’un graphe, mais de manière beaucoup plus rapide.

La mise en place de l’algorithme de Matula & Beck s’est fait à l’aide de la thèse suivante : <https://schulzchristian.github.io/thesis/thesis_huebner.pdf>.

9/10



Une image contenant texte

Description générée automatiquement

Pour montrer cela, nous allons comparer la vitesse de calcul de la dégénérescence du graphe **Flickr**.   
Ce graphe est composé de plus de **8 500 000 arrêtes** et est stocké dans un fichier pesant près de **100Mo**. Le calcul est pourtant très rapide mais la différence est saisissante

En effectuant des essais sur des plus grands graphes, nous avons trouvé quelque chose d’étrange. Sur le graphe **UK-Domains** composé de **261 787 000 arrêtes** et est stocké dans un fichier de **4Go**. **Matula & Beck semble plus lent pour l’exécution de très grands fichiers !**

10/10