Programmation Avancée

Devoir à la Maison

Enigma

David Baelde*

Ce devoir à la maison tient lieu de partiel. Il est à rendre pour le 1er avril. Un défi sous forme d'un code secret à déchiffrer sera de plus donné le 29 mars, avec des points bonus pour qui le résoudra en premier.

L'objectif de ce projet est de coder un algorithme permettant de "casser" le chiffrement Enigma. Ce n'est pas un projet facile, mais il ne faut pas non plus se laisser impressionner par la longueur du présent énoncé. Au final, il n'y pas beaucoup plus de 500 lignes de code à produire, dont une bonne partie est facile. Les difficultés sont variées : connaissance de base du système de modules d'OCaml ; algorithmique des graphes ; recherche combinatoire avec backtracking et structure de données semi-persistante.

Table des matières

1 Introduction **2**

2 Structure du code et instructions générales **4**

3 Structures de données préliminaires **4**

4 Analyse des cycles **5**
4.1 Définition du graphe 5
4.2 Calcul des cycles 6

5 Cassage du chiffrement **6**
5.1 Câblages contraints 7
5.2 On finit en force 8

*Un grand merci à Lucca Hirschi pour l'idée!

1 Introduction

L'objectif de ce petit projet est de casser le chiffrement Enigma. Nous nous appuierons sur une fameuse observation de Turing, que nous exploiterons avec les moyens modernes du cours de programmation avancée. Nous célèbrerons ainsi le $0^{\text{ème}}$ anniversaire du premier blockbuster sur la vie de Turing. Je donne ci-dessous un minimum de contexte et d'explications concernant Enigma et l'attaque de Turing. Pour plus de détails, on pourra consulter l'une des pages suivantes :

`https://interstices.info/jcms/int_70884/turing-a-lassaut-denigma`
`http://en.wikipedia.org/wiki/Enigma_machine`
`http://www.ellsbury.com/enigmabombe.htm`
`http://en.wikipedia.org/wiki/Bombe`

Contexte. Au début du $XX^{\text{ème}}$ siècle, il est clair qu'un chiffrement consistant simplement à appliquer une permutation sur l'alphabet est trop naïf et facilement attaquable par des méthodes statistiques. La machine Enigma dépasse ce cadre en utilisant une permutation changeant à chaque lettre. Cette permutation dynamique est réalisée par un système de rotors se reconfigurant après le chiffrement de chaque lettre. Cependant, les rotors sont des pièces complexes qui doivent être préfabriquées, et un utilisateur d'Enigma ne peut transporter qu'un petit nombre de rotors avec lui. Pour éviter une attaque par force brute, la machine Enigma permet enfin une substitution statique, qui peut être cablée de façon arbitraire par l'opérateur. Cette substitution est obtenue en câblant un certain nombre de transpositions disjointes, ce qui rend la transformation involutive : si T est câblé avec B, alors B est câblé avec T. La machine Enigma constituait un saut en avant dans les techniques de chiffrement, et elle a été supposée inviolable pendant longtemps. Mais un autre saut en avant, dans le domaine de la cryptanalyse mathématique, a permis de briser ce moyen chiffrement pendant la seconde guerre mondiale.

Définitions. Nous considèrerons une machine Enigma composée d'un câblage frontal, trois rotors et un réflecteur. On considère les caractères A, B, ... Z comme des éléments du groupe $\mathbb{Z}/26 \times \mathbb{Z}$. Dans la suite nous utiliserons les symboles i, j, k pour dénoter des éléments de ce groupe, et les symboles l, m, n pour dénoter des entiers naturels usuels. Si f est une permutation, on pose $f_{(i)}(x) = f(x + i)$ son décalage de i caractères. Une machine M est donnée par un câblage frontal p, un réflecteur q, trois rotors r, s et t et leurs positions respectives i, j et k. Le chiffrement d'un caractère x par une telle machine est donné par :

$$M = p \circ r_{(i)} \circ s_{(j)} \circ t_{(k)} \circ q \circ t^{-1}_{(k)} \circ s^{-1}_{(j)} \circ r^{-1}_{(i)} \circ p$$

En utilisant le fait que p et q sont involutives, on obtient que M l'est aussi : ainsi, le chiffrement et le déchiffrement d'un message sont effectués par la même machine. Après le chiffrement, la position des rotors est mise à jour comme suit : le premier rotor se décale d'un vingt-sixième de tour ($i := i + 1$) ; s'il a

fait un tour complet, le second rotor se décale aussi d'un cran ($j := j + 1$ si $i = 0$) ; de même pour le troisième rotor ($k := k + 1$ si $i = j = 0$). On note $M_{(n)}$ la machine obtenue après chiffrement de n caractères. On observe alors que $M_{(n+26^3)} = M_{(n)}$: en pratique, la permutation sera donc a priori différente pour chaque caractère d'un message.

L'attaque de Turing. De façon évidente, on attend d'un système de chiffrement qu'il rende difficile l'obtention d'un message à partir de son chiffré pour qui ignore la *clé*, composée ici de la configuration initiale de la machine. En fait, un système de chiffrement doit aussi être robuste aux attaques dites *à texte clair connu* : étant donné un message et son chiffré, il doit être difficile de déduire la clé — qui pourrait alors être utilisée pour décrypter des chiffrés dont le texte clair n'est pas connu. Des textes clairs connus peuvent en effet être obtenus lors d'opérations militaires, ou bien devinés à partir d'information supposées présentes dans certains messages : salutations, bulletins météo, signalement d'évènement provoqués par l'ennemi, etc. Alan Turing a notoirement contribué à la cryptanalyse d'Enigma, sur ce type d'attaques. Il a observé une faiblesse dans le chiffrement qui permet de dériver, à partir d'une paire clair-chiffré connu, une quantité d'information significative sur le câblage frontal p utilisé. Il a aussi participé au développement de moyens électromécaniques pour faciliter l'exploitation de cette faiblesse afin de casser Enigma : c'est la fameuse *bombe*.

Nous présentons maintenant l'observation clé de Turing, que nous allons exploiter dans ce projet. Soit A un message de longueur l et B son chiffré par une machine Enigma de configuration inconnue. On notera $A(n)$ et $B(n)$ les caractères de ces messages, pour $0 \leq n < l$. Soit $G(A, B)$ le multi-graphe non-orienté dont les sommets sont les caractères, et qui contient pour tout n une arête étiquetée n entre $A(n)$ et $B(n)$. On remarque que l'existence d'un cycle [1] suivant un chemin étiqueté $i_1, i_2, \ldots i_n$ dans ce graphe ne dépend pas du câblage frontal p d'un hypothétique machine Enigma M associant A et B. En effet, considérons un chemin dans le graphe :

$$X_1 \overset{M_{(i_1)}}{--} X_2 \overset{M_{(i_2)}}{--} \ldots \overset{M_{(i_{n-1})}}{--} X_n \overset{M_{(i_n)}}{--} X_{n+1}$$

Notons $M'_{(i)}$ la machine identique à $M_{(i)}$ mais avec un câblage frontal égal à l'identité. Alors on a $X_{n+1} = X_1$ si et seulement si $p(X_1)$ est un point fixe de $M'_{(i_1)} \circ M'_{(i_2)} \circ \ldots M'_{(i_n)}$, puisque les passages par p en entrée et sortie de la machine à chaque étape s'annulent. Ainsi, la connaissance des cycles élémentaires de $G(A, B)$ permet d'éliminer de nombreuses configurations possibles des rotors pour M, mais aussi de déduire des informations sur le câblage frontal p. Par exemple, si pour une configuration initiale des rotors il n'y a qu'un point fixe Y, alors on sait que $p(X_1) = Y$. En pratique, cette technique rend aujourd'hui très facile l'attaque d'Enigma à texte clair connu.

1. Et même, d'un cycle élémentaire.

2 Structure du code et instructions générales

Une base de code OCaml est disponible en ligne. Elle contient l'implémentation des modules `Rotor`, `Involution` et `Enigma`, qui permettent de simuler la machine Enigma. Ces modules dépendent cependant du module `Symbol` dont seulement la signature est donnée, et dont l'implémentation constitue la première question du projet.

Le projet final comportera au final 9 modules, dont certains sont vraiment petits. Plusieurs fichiers `*.mli` sont donnés ; il est demandé de **respecter les interfaces**. Quand une interface déclare un type abstrait, il est rigoureusement interdit de briser cette abstraction. En général, on évitera d'ajouter aux interfaces autre choses que des fonctions de debug, affichage, etc. Les interfaces sont documentées, donnant des détails qui ne sont pas répétés ci-dessous : pensez à systématiquement aller les lire.

Le projet devra être rendu par mail sous la forme d'un tarball contenant les fichiers source finaux (`*.ml` et `*.mli`) ainsi que le `Makefile`, un éventuel `README`, mais aucun fichier compilé. Sauf bonus, la note sera basée sur la correction, l'efficacité et la clarté du code écrit en réponse aux questions de l'énoncé. Concernant le dernier point, les commentaires sont bien sûr bienvenus, dans le code ou dans un fichier `README`. Veillez à bien respecter les noms (de fonction, type, fichier, etc.) indiqués par l'énoncé, sans quoi je pourrais considérer une question comme non traitée.

Le Makefile. La base de code contient un fichier `Makefile` conçu pour s'adapter automatiquement à l'ajout de nouveaux fichiers `*.ml` et `*.mli`. Pour cela, le principe est de compiler à partir de tous les modules présents un unique exécutable nommé `enigma`, et de créer ensuite divers liens symboliques vers cet exécutable. L'exécutable se comportera différemment en fonction du nom utilisé lors son invocation, comme cela est illustré dans le fichier `enigma.ml` fourni. Par exemple, l'exécutable `enigma` servira de simulateur, l'exécutable `cycles` servira (à terme) à tester le calcul des cycles, etc. Dans la base de code fournie, `make` provoquera la compilation des fichiers `*.ml` et `*.mli` présents, ainsi que de leur documentation, générée dans le sous-répertoire `html`. Dès que le module `Symbol` sera disponible on ira ajouter `enigma` aux dépendances de la recette par défaut `all` dans le Makefile. Plus d'informations sur le Makefile y sont présentes en commentaire, notamment pour changer les options de compilation.

3 Structures de données préliminaires

Nous allons manipuler de façon intensive les caractères de A à Z mais aussi des fonctions de domaine ces caractères et des ensembles de ces caractères. Les modules `Map` et `Set` de la bibliothèque standard d'OCaml sont conçus pour des domaines ordonnés arbitraires ; on a tout intérêt à les éviter ici, et représenter directement les fonctions par des tableaux et les ensembles par des entiers.

Afin de garantir la bonne utilisation de ces structures de données, sans en diminuer l'efficacité, on exploite le typage statique d'OCaml et plus particulièrement les types abstraits : la signature du module `Symbol` cache les définitions concrètes des types `Symbol.sym`, `Symbol.Map.t` et `Symbol.Set.t`, afin de garantir qu'un `sym` est toujours un entier compris entre 0 et 25, qu'un `Map.t` est toujours un tableau de taille 26, et qu'un `Set.t` est toujours un entier $0 \leq n < 2^{26}$. Ces garanties présupposent bien sûr une implémentation correcte du module `Symbol`, ainsi que le respect du système de type !

Question. Implémenter le module `Symbol` dans le fichier `symbol.ml` en suivant la signature définie dans `symbol.mli`. Il est bien entendu rigoureusement interdit de modifier cette signature !

Vous pouvez désormais faire `make enigma` ; ajoutez la dépendance à la cible `all` pour vous simplifier le vie et éviter les oublis dans la suite. L'exécutable `enigma` prend des messages sur la ligne de commande ou l'entrée standard, et les traduit avec une machine de test prédéfinie. Le codage de `BONJOUR` devrait être `TEATGGS`.

4 Analyse des cycles

On s'intéresse ici à construire le graphe associé à un couple clair-chiffré, et à calculer ses cycles. Cette tâche sera réalisée par l'exécutable `cycles`.

4.1 Définition du graphe

Comme expliqué en introduction, l'attaque de Turing exploite les cycles élémentaires d'un certain multi-graphe non-orienté. Dans un premier temps, nous pouvons considérer le graphe où toutes les arêtes entre deux sommets ont été fusionnées en une arête étiquetée par l'ensemble des étiquettes fusionnées. Calculer les cycles élementaires du graphe fusionné est moins couteux, et permettra de retrouver ensuite les cycles du graphe original.

Question. Définir le module `Graph` selon la signature donnée dans `graph.mli`.

Question. Coder dans `cycles.ml` la fonction `graph_of_known_cipher` qui prend un message clair et son chiffré et renvoie le graphe décrit en introduction, de type `Graph.t`.

Question. Toujours dans `cycles.ml`, commencer à définir le comportement de l'exécutable `cycles` — ce que l'on détecte par `Sys.argv.(0)= "cycles"`, cf. fonctionnement similaire dans `enigma.ml`. On veut que, quand on exécute `./cycles <clair> <chiffré>`, le graphe correspondant à ce couple clair-chiffé soit calculé.

4.2 Calcul des cycles

Une façon de calculer tous les cycles élémentaires est de calculer, de façon plus générale, tous les chemins *élémentaires* et *ordonnés* d'un graphe. Un chemin est dit *élémentaire* s'il ne contient pas de sous-chemin strict qui soit cyclique, c'est à dire dont les sommets de départ et d'arrivée coïncident. Il est dit *ordonné* si le sommet de départ est inférieur à tous les autres. On remarque que chaque cycle élémentaire correspond à au plus deux chemins élémentaires, ordonnés et cycliques : un pour chaque sens de parcours du cycle, à partir du plus petit sommet du cycle.

Le calcul des chemins se fait aisément de façon incrémentale. Les chemins à un sommet et zéro arêtes, formant l'ensemble C_1, sont simplement les singletons de sommets du graphe. On peut ensuite calculer C_{n+1} à partir de C_n en étendant les chemins de C_n de toutes les façons qui préservent le caractère élémentaire et ordonné des chemins.

Afin d'implémenter cet algorithme efficacement, il faut choisir les bonnes structures de données : d'une part pour représenter les chemins, d'autre part pour organiser les chemins de C_k.

Question. Créer le module `Path` selon la signature fournie dans `path.mli`. Toutes les opérations sauf `compare` doivent être en temps constant, en fait quasi immédiates.

Question. Implémenter `Cycles.cycles` qui calcule tous les cycles utiles d'un graphe, avec un seul représentant par cycle non-orienté. Les cycles inutiles sont ceux qui comportent seulement deux arêtes ; ils n'apportent aucune information. (On pourra utiliser le fait qu'il n'y a jamais de cycle avec une seule arête, car le réflecteur et donc la machine n'ont pas de point fixe.) Le type de la valeur retournée par la fonction est libre : il dépendra des détails de votre implémentation, et des besoins ultérieurs.

Question. Implémenter dans le module `Cycles` deux fonctions qui à partir du résultat de la fonction `cycles` itèrent respectivement sur les cycles du graphe fusionné et sur les cycles du graphe original.

Question. Compléter le module pour que l'exécutable `cycles` affiche le nombre de cycles trouvés avant et après expansion [2] et liste les cycles (avant expansion) s'il y en a moins de 20. Déclarer dans `cycles.mli` une signature minimale pour ce module, qui fait du résultat de la fonction `cycles` un type abstrait.

5 Cassage du chiffrement

Dans cette dernière partie, nous écrivons finalement notre procédure pour casser Enigma. Ce sera notre *bombe*, même si l'appelation est abusive : la bombe

2. Autrement dit, dans le graphe fusionné et le graphe original.

historique effectue un calcul plus simple qu'ici — mais avec des moyens infiniment plus limités !

L'idée générale est de calculer, pour chaque configuration possible des rotors, quelles sont les contraintes induites par les cycles sur le câblage frontal, puis d'énumérer tous les câblages possibles sous ces contraintes pour trouver ceux qui permettent d'obtenir l'association clair-chiffré connue.

5.1 Câblages contraints

La principale difficulté réside dans la représentation des contraintes sur le câblage frontal. Nous allons définir dans le module `Board` un type `t` représentant un câblage contraint. Nous choisissons, pour des raisons d'efficacité, de le réaliser dans un style *semi-persistant* : l'ajout de contraintes aura un impact sur le câblage contraint, plutôt que de créer un nouveau câblage nouvellement contraint ; par contre, notre structure de données sera dotée d'un moyen efficace pour revenir à un état passé, ce qui la rend adaptée à une utilisation dans un algorithme avec backtracking. Ce deuxième point sera traité dans la prochaine et dernière sous-section.

Un câblage contraint représente essentiellement un ensemble de câblages possibles. Le câblage le moins contraint, $\top$, représente l'intégralité des câblages possibles. Le câblage le plus contraint, $\bot$, est l'ensemble vide. Entre deux, on a des câblages p pour lesquels pour chaque x les valeurs de $p(x)$ n'est pas un symbole mais un ensemble de symboles, signifiant que $p(x)$ pourrait être n'importe lequel de ces symboles. La contrainte d'involutivité se traduit sur les ensembles : pour tout $y \in p(x)$, $x \in p(y)$.

On ne considère qu'une façon primitive de contraindre un câblage : ajouter une contrainte interdisant l'association de deux symboles x et y. De façon immédiate, cette contrainte doit supprimer (si ce n'est déjà fait) x de $p(y)$, et vice versa. Si cette suppression engendre un ensemble vide, le câblage contraint n'est pas faisable. Si par cette suppression on a obtenu un ensemble singleton, par exemple $p(x) = \{z\}$, alors on va forcer l'association de z à x et supprimer toute association de z à y pour $y \neq x$. Ces suppressions, à leur tour, pourront provoquer des propagations de contraintes comme décrit ci-dessus.

Par exemple, sur quatre symboles, considérons le câblage contraint suivant, obtenu en interdisant les associations d'un symbole avec lui même, ainsi que l'association de A avec D :

$$p(A) = \{B, C\},\ p(B) = \{A, C, D\},\ p(C) = \{A, B, D\},\ p(D) = \{B, C\}$$

Si l'on interdit l'association de A et C, on se retrouve avec

$$p(A) = \{B\quad\},\ p(B) = \{A, C, D\},\ p(C) = \{\quad B, D\},\ p(D) = \{B, C\}$$

puis, comme l'association de A et B se retrouve forcée côté A,

$$p(A) = \{B\quad\},\ p(B) = \{A\qquad\},\ p(C) = \{\qquad D\},\ p(D) = \{B, C\}$$

et enfin, comme $p(C)$ est devenu un singleton :

$$p(A) = \{B \quad \},\ p(B) = \{A \qquad \},\ p(C) = \{ \qquad D\},\ p(D) = \{ \quad C\}.$$

Ainsi, l'ajout de la contrainte de non-association entre A et D a complètement déterminé notre câblage.

Question. Implémenter le module `Board` selon la signature fournie, dans laquelle les seules fonctionnalités non commentées sont `top` et `remove_assoc`. On pourra représenter les câblages (type `Board.t`) de façon assez proche de la description ci-dessus, avec quelques informations judicieusement stockées pour éviter de les recalculer très souvent.

Question. Implémenter dans `bombe.ml` la fonction `infer_plug_constraint` (dont le type précis est libre) qui prend un ensemble de cycles issus de l'analyse précédente et une configuration initiale (supposée) des rotors d'une machine, et calcule les contraintes induites par les cycles sur le cablage frontal de la machine Enigma.

Question. Dans `bombe.ml`, faire en sorte que l'exécutable `bombe` affiche le câblage contraint induit par les cycles, sous l'hypothèse que les rotors (et leurs positions) sont deux de la machine de test.

5.2 On finit en force

Notre bombe va devoir énumérer un certain nombre de câblages possibles. En d'autres termes, elle va tester tous les choix possibles, dans l'espace des câblages contraints d'après les cycles, et backtracker sur ces choix. Nous allons programmer ceci en utilisant des itérateurs ou, autrement dit, en travaillant (implicitement) dans la monade de non-déterminisme construite sur

```
type 'a m = ('a ->unit)-> unit.
```

On utilise cette définition de type dans la suite pour faciliter la lecture, mais il n'est pas requis de le définir explicitement dans votre code.

Question. Ecrire `Bombe.mpos : Rotor.t -> Rotor.t m` qui énumère toutes les positions possibles d'un rotor. Dit autrement, c'est un opérateur de choix sur l'ensemble de ces positions.

Question. Ecrire `Bombe.melem : 'a list -> ('a * 'a list)m` qui énumère tous les éléments d'une liste, en renvoyant avec chaque élément la liste contenant les autres éléments, dans un ordre non spécifié. Dit autrement, c'est un opérateur de choix non-déterministe sur les éléments d'une liste.

Question. Adapter le comportement de l'exécutable `bombe` pour qu'il teste toutes les configurations possibles des rotors, et affiche pour chacune le câblage contraint induit par les cycles, s'il n'est pas infaisable. (On suppose que les rotors

sont toujours pris dans `Rotor.rotors`, avec un seul rotor de chaque type, et que le réflecteur est toujours `Involution.reflector`.)

Nous souhaitons enfin créer un exécutable `brute` qui fait comme `bombe` mais cherche ensuite, de façon non-déterministe[3], à contraindre les câblages obtenus pour chaque configuration des rotors, jusqu'à trouver des câblages qui permettent effectivement d'obtenir l'association clair-chiffré attendue.

Considérons un exemple. Supposons qu'on a un couple clair-chiffré, et une configuration hypothétique des rotors, tel que le câblage contraint induit par les cycles est celui du début de l'exemple précédent sur quatre symboles. Si le premier symbole du message clair est un A, nous ne savons pas si le câblage associe A à B ou C. On va simplement tester les deux choix possibles. Si $p(A) = B$, alors on peut exécuter la machine sur l'entrée A, faire faire un aller-retour à B à travers les rotors ; si X est le symbole obtenu après cet aller-retour, on sait que $p(X)$ est le premier symbole du chiffré, disons Y. Ici, la première contrainte ($p(A) = B$) suffisait à déterminer le câblage, et il faut donc que la seconde contrainte ($p(X) = Y$) soit satisfaite par le câblage obtenu, sans quoi cette branche de l'exploration échoue immédiatemment. Après avoir testé ce choix, il faut revenir au câblage contraint initial pour explorer le cas où $p(A) = C$.

Étant donné que `Board.t` est codé de façon impérative/destructive, il faudrait pouvoir copier le câblage contraint pour implémenter le backtracking nécessaire. Mais une copie est une opération coûteuse. Pour faire mieux, nous allons rendre `Board.t` semi-persistant, en implémentant les fonctionalités `save` et `restore` en commentaire dans la signature. L'idée est simple : on enrichit le type `Board.t` avec un historique qui garde suffisamment d'information sur chaque modification du câblage contraint, de façon à pouvoir annuler ces modifications. Les seules modifications à considérer ici (encore une fois, grâce au fait que `Board.t` est abstrait) sont celles effectuées dans `remove_assoc`.

Question. Ajouter les fonctionalités d'historique au module `Board`.

Question. Implémenter, dans le module `Bombe`, les deux opérations suivantes :

```
val mget : Board.t -> Symbol.sym -> Symbol.sym m
val mforce_assoc :
          Board.t -> Symbol.sym -> Symbol.sym -> unit m
```

L'opération `mget p x` doit correspondre à une énumération des associations possibles de `x` dans `p`. Soit `f` la valeur passée à l'énumération, c'est à dire le troisième argument de `mget`. Pour chaque appel de `f` sur une valeur associée à `x`, l'état de `p` doit être mise à jour pour refléter, pendant l'appel `f y`, le fait que l'association de `x` et `y` est forcée dans `p`. L'état de `p` doit être identique avant et après l'appel de `mget p x f` — on pourra ici supposer que `f` ne lève jamais d'exception. L'opération `mforce_assoc p x y` énumère sur les tentatives de

3. C'est une vue de l'esprit : les choix sont tous explorés séquentiellement, de façon déterministe.

forcer l'association de `x` et `y` dans `p`. C'est une énumération d'au plus un élément, dont la valeur (de type `unit`) ne nous intéresse pas. La gestion de l'état de `p` est comme pour `mget`.

Question. Réaliser l'algorithme décrit plus haut pour l'exécutable `brute`. Pour chaque solution, le câblage obtenu sera affiché.