MUEI

CSAI: CIFRADO DE VIGENERE

14 de abril de 2015

David Pereiro Lagares

Índice general

0.1. INTRODUCCIÓN

Esta práctica ha sido realizada en *scala*.

El repositorio del proyecto puede ser accedido en: `https://bitbucket.org/torce/vigenere`.

0.2. DESCIFRADO POR FUERZA BRUTA

En el descifrado por fuerza bruta simplemente generamos una secuencia de claves por orden, *A, B, ... , AA, AB, AC, ...*, y desciframos el texto con estas.

Para cada descifrado se comprueba la validez de la solución y si es válida se imprime por la salida seleccionada.

0.3. COMPROBACIÓN DE VALIDEZ

Para saber si una solución es válida, se buscan en ella palabras comunes en el lenguaje seleccionado, en nuestro caso en inglés, y si el número de palabras encontrado supera un parámetro proporcionado por el usuario se considera válida.

La lista de palabras considerada son las 20 palabras más frecuentes en Inglés. Hemos excluido las de un solo carácter, ya que estas no daban buenos resultados.

```
val commonWords: Seq[String] = Seq("THE", "BE", "TO", "OF", "AND",
                                   "IN", "THAT", "HAVE", "IT", "FOR",
                                   "NOT", "ON", "WITH", "HE", "AS",
                                   "YOU", "DO", "AT", "THIS", "BUT")
```

0.4. DESCIFRADO ADIVINANDO LONGITUD DE CLAVE

0.4.1. Longitud de la clave

Para tratar de adivinar cual es la longitud de la clave, el algoritmo que seguimos es recorrer el texto buscando caracteres repetidos con una distancia *n*. Cada vez que se encuentra un carácter repetido se suma 1 a la puntuación de la distancia n.

Tras repetir el proceso con varios valores de *n*, obtenemos una lista de posibles longitudes de clave ordenadas por su puntuación.

0.4.2. Algoritmo

El algoritmo básico es el siguiente.

1. Para cada posible longitud de clave, dividimos el texto en particiones, por ejemplo si la longitud de clave es 2, una partición tendrá los caracteres con índice par y otra los caracteres con índice impar.

2. Ahora cada una de estas particiones es un cifrado monoalfabético que podemos resolver empleando una tabla de frecuencias.

3. En base al carácter más repetido en la partición y los más frecuentes en el lenguaje elegido, calculamos el posible delta aplicado.

4. Realizamos este proceso para todas las particiones y luego las combinamos en un solo texto.

5. Se realiza una validación del texto buscando en el palabras comunes en el lenguaje elegido. El número de palabras mínimo para que una solución sea válida es configurable.

6. Si el texto pasa la validación, se imprime en la salida un trozo de este.

7. El proceso continúa hasta que se prueban todas las combinaciones o el usuario encuentra un texto válido y lo detiene.

0.4.2.1. Combinación de las particiones

Existen múltiples formas de combinar las distintas particiones en función de que carácter del idioma se haya considerado para calcular el delta (el más frecuente, el segundo más frecuente, etc).

Suponiendo dos particiones, en el proceso de descifrado en ambas se buscará el carácter mas usado y se calculará el delta entre este y el más usado en el idioma, ya que es la combinación más probable.

En la siguiente iteración se descifrará la primera partición usando el **segundo** carácter más usado en el lenguaje y la segunda con el más usado, ya que es la siguiente combinación más probable.

Tomando como ejemplo dos particiones, la secuencia de combinaciones óptima sería:

Partición 0	Partición 1
0	0
1	0
0	1
1	1
2	0
2	1
...	...

0.4.3. Otras optimizaciones

Una de las optimizaciones que más ha permitido mejorar los tiempos para textos largos ha sido el realizar un descifrado de solamente un fragmento del texto y realizar las comprobaciones de validez sobre este.

Si el texto es una solución válida, entonces se descifra al completo con la clave descubierta, si no se prueba la siguiente clave.

0.5. PROBLEMAS CONOCIDOS

- La validación buscando palabras comunes se realiza sin importar mayúsculas o minúsculas, lo cual puede llevar a que se muestren algunas soluciones que podrían ser no válidas porque se encuentran en ellas palabras con mezcla de mayúsculas y minúsculas.

- La forma en la que buscamos la longitud de clave tiene el problema de que tiende a otorgar una puntuación importante a los múltiplos de la longitud de clave. Por ejemplo si la clave es de 4 caracteres, los valores de longitud 8 y 12 también tendrán una puntuación elevada y en algunos casos incluso superior al valor real de la clave. Lo cual haría que el algoritmo resolviese una clave de 8 equivalente a una de 4, "passpass"para la clave "pass".

  Pese a esto, resolver una clave de 8 caracteres equivalente a una de 4 con este sistema sigue siendo mucho más rápido que por fuerza bruta.

0.6. PRUEBA

La prueba se ha realizado empleando como entrada a cifrar la página de la wikipedia *Philosophy* `http://en.wikipedia.org/wiki/Philosophy`.

Para repetir esta prueba, simplemente hay que ejecutar el fichero `test.sh` y examinar la salida.

Como alfabeto de cifrado se ha empleado solo minúsculas, para que la salida fuera más clara debido al problema citado en 0.5.

En la figura 1, puede verse como el índice de coincidencia del texto cifrado desciende a medida que se aumenta la longitud de clave, aunque este descenso es muy lento.

La prueba de fuerza bruta solamente la hemos realizado hasta claves de longitud 5, debido a que los tiempos de ejecución aumentan exponencialmente con la longitud de la clave. Los resultados pueden verse en la figura 2.

Empleando el algoritmo explicado antes, la clave correcta suele ser la primera salida del programa, debido a que se prueban las claves por orden de probabilidad, por eso el tiempo de ejecución no varía demasiado en función de la longitud de clave.

Si que hemos notado que a mayor longitud de clave, las particiones son más pequeñas y por lo tanto la información estadística que empleamos para romper cada uno de los n cifrados monoalfabéticos se vuelve menos precisa. A consecuencia de esto, la clave correcta ya no es la primera clave probada por el programa y los tiempos se alargan. Los tiempos pueden verse en la figura 3, estos no incluyen el tiempo empleado en adivinar la longitud de la clave, solo el descifrado.

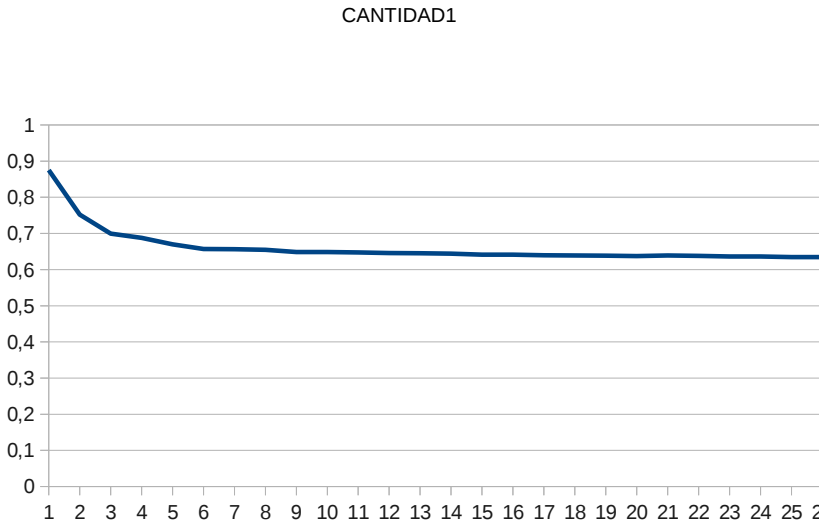
El tiempo que se tarda en adivinar la longitud de la clave parece crecer de forma lineal con el tamaño de esta, aunque necesitaríamos realizar más mediciones para confirmarlo, figura 4.

En conclusión, la eficiencia del método de fuerza bruta está determinada únicamente por la longitud de la clave empleada, mientras que empleando particiones y tabla de frecuencias está determinada por la relación entre la cantidad de texto cifrado que se disponga y la longitud de la clave.

0.7. MEJORAS POSIBLES

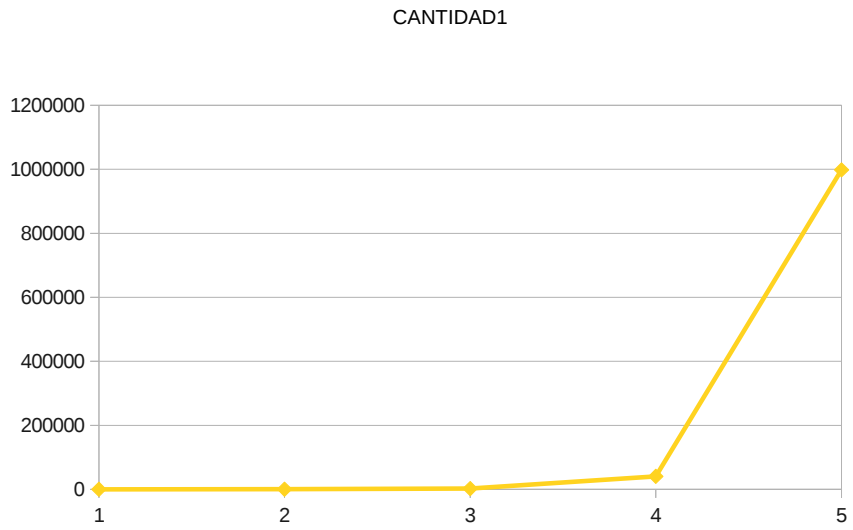
0.7.1. Mejora del algoritmo

Se ha observado que en muchos casos, especialmente cuando se intenta descifrar un texto que emplea una clave larga, el algoritmo encuentra muchas posibles claves muy similares



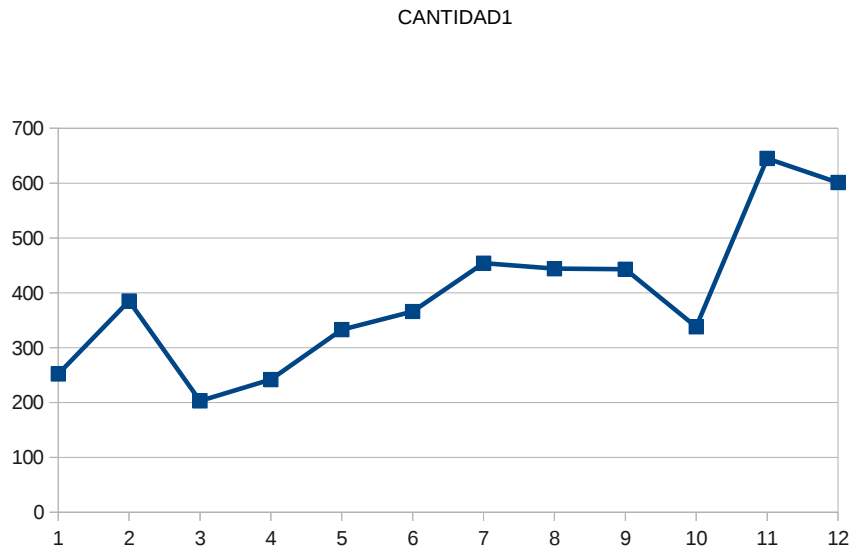
Error: el cálculo no converge 1

Figura 1: índice de coincidencia en función de la longitud de clave



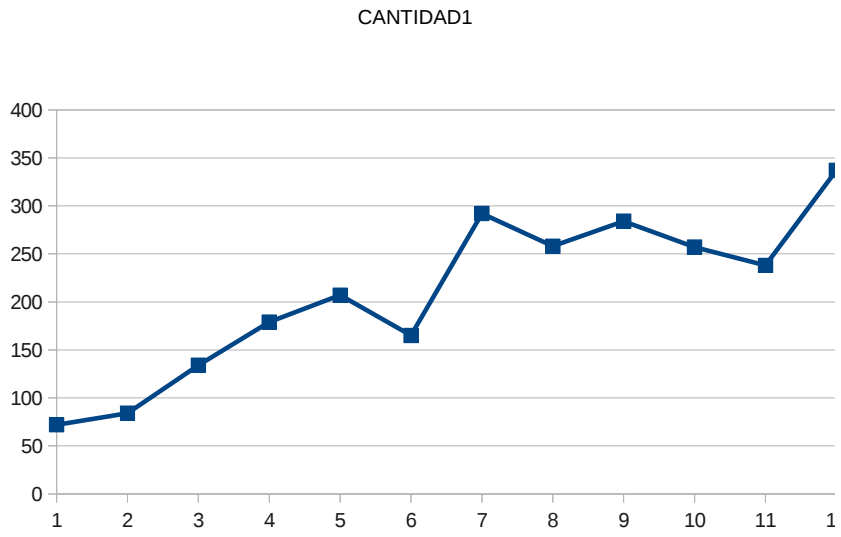
Error: el cálculo no converge 1

Figura 2: Fuerza bruta, tiempos en función de la longitud de clave, milisegundos



Error: el cálculo no converge 1

Figura 3: Algoritmo con tabla de frecuencias, tiempos en función de la longitud de clave milisegundos



Error: el cálculo no converge 1

Figura 4: Tiempos de cálculo de la longitud de clave, milisegundos

a la clave correcta. Por ejemplo, si la clave es `passwordlong`, es probable que el algoritmo encuentre claves tales como:

```
passaordkong
passaordling
```

Una opción para llegar a la clave real a partir de alguna de las claves parciales, por ejemplo `passaordkong`, sería realizar un análisis del texto descifrado, buscando palabras mal formadas a las que reemplazándoles caracteres den como resultado una palabra correcta en el idioma. Por ejemplo, podríamos encontrarnos en el texto descifrado palabras como `housr` y `orangt`, a las que cambiándoles un carácter, coincidirían con `house` y `orange` respectivamente.

En teoría estos caracteres del texto que no corresponden con la palabra original se corresponderían con las partes de la clave que no son correctas. Calculando el *delta* de los caracteres de estas palabras podríamos corregir partes de la clave y llegar a la clave correcta a partir de una clave similar, en nuestro caso, llegaríamos a `passwordlong` a partir de `passaordkong`.

0.7.2. Mejora de la implementación

Como cada una de las pruebas de claves es completamente independiente de las demás, los algoritmos podrían ser fácilmente paralelizados tanto el de fuerza bruta como el de tabla de frecuencias.

0.8. FORMA DE USO

`java - jar vigenere-assembly-1.0.jar <parámetros>`
Al menos se le debe indicar uno de los siguientes parámetros

- -c: Realiza un cifrado
- -d: Realiza un descrifrado
- -b: Intenta romper un cifrado usando fuerza bruta
- -bg: Intenta romper un cifrado adivinando la longitud de la clave

Si se elige cifrado o descifrado, se debe especificar con *-k* la clave.
La entrada puede ser un fichero o un argumento por linea de comandos

- -s: La entrada es el *string* que siga a este argumento
- -f: La entrada es el fichero proporcionado por este argumento

La salida por defecto es la pantalla, con *-o* se puede especificar un fichero de salida.
El alfabeto a utilizar se elige con *-x*, los valores posibles son:

- A: Mayusculas
- a: Minusculas
- 0: Números
- $: Símbolos

Estos valores para el alfabeto pueden combinarse entre si.
En caso de querer romper un cifrado, se deben proporcionar las siguientes opciones.

- -l: Longitud máxima posible de la clave
- -m: Número de palabras encontradas en una solución para que se considere válida. Véase 0.3

Otras opciones posibles para romper un cifrado serían

- -a: Número de caracteres que se prueban cuando se realiza el descifrado por tabla de frecuencia. Por defecto se prueban todos los caracteres del lenguaje, desde el más probable hasta el menos probable por orden. Véase 0.4.2.1
- -n: Longitud del *snippet* que se descifra para comprobar la validez de la clave, en lugar de descifrar el texto completo. Por defecto es 512, un valor que ha funcionado bien en las pruebas. Véase 0.4.3

0.9. COMPILACIÓN

Para poder construir el proyecto es necesario tener instalada la herramienta de construcción *sbt*, `http://www.scala-sbt.org/`, y *scala* 2.10.4.

Para construir el programa, es necesario ejecutar:

```
sbt assembly
```

Lo cual descargará las dependencias del proyecto, compilará el código fuente y tras pasar los test, generará un *fat jar*, esto es, un fichero *.jar* que contiene todas las dependencias necesarias y puede ser ejecutado directamente con `java -jar <argumentos del programa>`.