Facultad de Informática

UNIVERSIDAD DE MURCIA

AUTÓMATAS Y LENGUAJES FORMALES

GRADO EN INGENIERÍA INFORMÁTICA

Práctica 1 - Curso 2019/2020

Autores:
Eduardo MARTÍNEZ GRACIÁ
edumart@um.es
Mercedes VALDÉS VELA
mdvaldes@um.es
Santiago PAREDES MORENO
chapu@um.es
José Manuel JUÁREZ HERRERO
jmjuarez@um.es

13 de octubre de 2019

Introducción

En las transmisiones digitales a través de canales con interferencias, es necesario proteger la secuencia de bits de la señal que se quiere transmitir para evitar la pérdida de información. La protección de la señal consiste en añadir información redundante para que, en caso de alteración de la señal, y bajo ciertas condiciones, se pueda detectar el error y la señal pueda ser recuperada. Cuanta más información redundante se añade, mayor protección se obtiene, pero también es mayor el coste de la transmisión de un bit de la señal. En ingeniería nos encontramos con infinidad de situaciones en las que hay que llegar a un compromiso entre bueno y barato.

Los sistemas actuales de transmisión digital inalámbrica (como las transmisiones de teléfonos móviles o la televisión digital terrestre) utilizan los denominados *códigos convolucionales*. El nombre puede amedrentar, pero la idea es realmente sencilla, ya que se basa en el uso de registros de desplazamiento a nivel de bits y de operaciones XOR.

Esta práctica consiste en modelar con un AFD el lenguaje de las secuencias de bits válidas que pueden generarse con un codificador convolucional sencillo.

Código convolucional

La figura 1 muestra el codificador convolucional que emplearemos en esta práctica. Por cada bit de entrada (x) se generan dos bits (y_1 e y_2) para transmitir. La forma de generar los bits de salida consiste en combinar el valor del bit de entrada junto con los valores de algunas de las posiciones de un registro de desplazamiento de bits que memoriza hasta tres entradas anteriores. Inicialmente este registro se encuentra con todas sus entradas a 0.

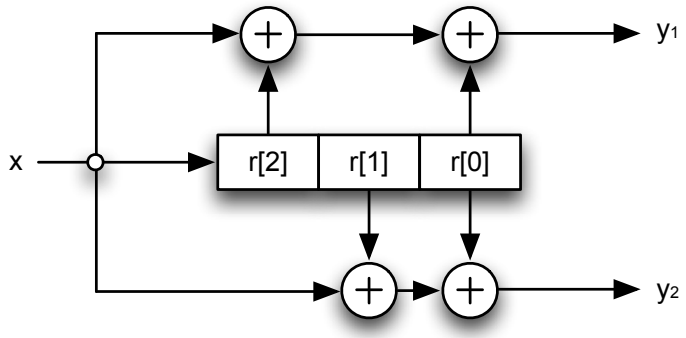


Figura 1: Codificador convolucional en la configuración inicial.

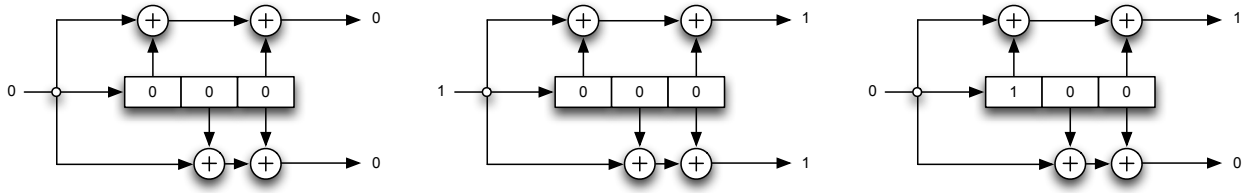
Vamos a denominar r al registro de desplazamiento, de modo que $r[i]$ representa el valor de la posición i-ésima. Si empleamos el operador $+$ como un XOR binario, la forma de calcular los dos bits de salida es la siguiente:

$$y_1 = x + r[2] + r[0]$$
$$y_2 = x + r[1] + r[0]$$

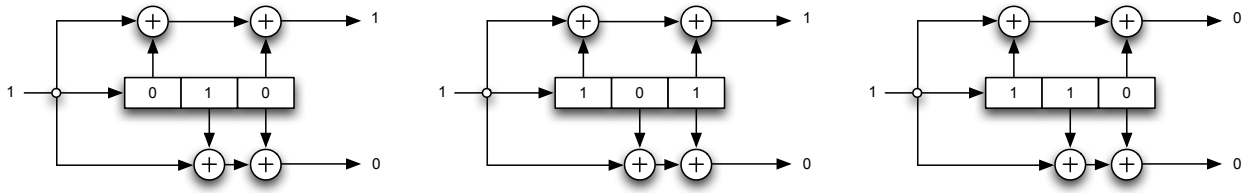
Es decir, y_1 es un bit de paridad par de x, $r[2]$ y $r[0]$, mientras que y_2 es un bit de paridad par de x, $r[1]$ y $r[0]$. La actualización del registro se realiza después de calcular los valores de la salida:

$$
\begin{aligned}
r[0] &= r[1] \\
r[1] &= r[2] \\
r[2] &= x
\end{aligned}
$$

Veamos un ejemplo de codificacion de la secuencia binaria 010111. La secuencia se debe transmitir de izquierda a derecha, es decir, el primer valor de x es 0, luego 1, etc.



(a) $x = 0$ (b) $x = 1$ (c) $x = 0$



(d) $x = 1$ (e) $x = 1$ (f) $x = 1$

Figura 2: Secuencia de estados del codificador con la entrada 010111

Concatenando las salidas para cada bit de la entrada 010111, indicadas en la figura 2, se obtiene la secuencia codificada 001110101000.

La práctica

Esta práctica tiene dos partes:

1. Implementar con JFLAP un AFD que permita validar la salida de un codificador convolucional como el indicado en la sección previa.
2. Implementar en Python un programa que emplee el AFD anterior para verificar secuencias de bits almacenadas en un fichero.

El autómata

El autómata debe tener en cuenta las siguientes reglas:

1. Debe ser un autómata finito determinista.

2. El estado inicial del autómata representa la configuración del codificador con el registro de bits a 0.
3. El autómata puede tener un alfabeto de entrada distinto de $V = \{0, 1\}$ siempre y cuando la conversión de una cadena de V^* al nuevo alfabeto se realice en orden lineal.

El programa

El programa debe implementarse en Python haciendo uso del paquete `jflap` que aparece descrito en el capítulo 8 de los *Apuntes de Python*. El programa debe realizar los siguientes pasos:

1. Solicitar un nombre de fichero al usuario.
2. Si el nombre de fichero es una cadena vacía, el programa debe terminar.
3. Si el fichero no existe, el programa debe indicarlo y volver a solicitar un nombre nuevo.
4. Si el nombre es correcto, el programa debe procesar cada línea del fichero como una secuencia de bits que sale del codificador convolucional.
5. El programa debe imprimir un mensaje con el resultado de la validación de cada línea con el siguiente formato:

   `Línea X CADENA: RESULTADO`

   donde `X` es el número de línea, `CADENA` es la secuencia de bits de la línea, y `RESULTADO` es o bien `válida` o bien `no válida en N`; `N` indica la posición (empezando en 1) del primer bit en el que la validación de la cadena falla.

Por ejemplo, un fichero `prueba.txt` con el siguiente contenido:

```
001110101000
11011101011010
1101001111
11011010
00001101111110
```

genera la siguiente salida:

```
Línea 1 001110101000: válida
Línea 2 11011101011010: válida
Línea 3 1101001111: válida
Línea 4 11011010: no válida en 6
Línea 5 00001101111110: no válida en 12
```

La validación se debe implementar mediante el algoritmo general de simulación de un AFD que se encuentra en la página 19 de los apuntes del tema 2.

La evaluación

Este ejercicio puntúa un máximo de 2 puntos: 1 punto por el autómata y 1 punto por el programa Python. La entrega de la práctica consistirá en dos ficheros:

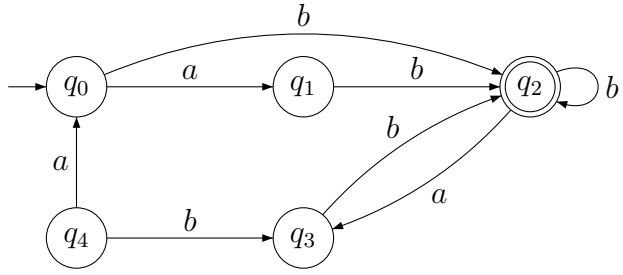
- `practica1.jff`: autómata desarrollado con JFLAP.
- `practica1.py`: programa Python de validación.

Alternativa: minimización

Tienes una alternativa a la práctica 1. En lugar de diseñar el autómata y el programa de validación, puedes obtener 3 puntos si implementas un programa en Python que minimice un autómata finito determinista. El programa debe realizar los siguientes pasos:

1. Solicitar un nombre de fichero JFLAP al usuario.
2. Si el nombre de fichero es una cadena vacía, el programa debe terminar.
3. Si el fichero no existe, el programa debe indicarlo y volver a solicitar un nombre nuevo.
4. Si el nombre es correcto, el programa debe leer el fichero JFLAP y generar una salida con la tabla de transiciones del autómata mínimo equivalente.

Supongamos que la entrada al programa es el siguiente autómata:



La salida deberá tener un formato similar al siguiente:

```
   Estados   | a         | b
-----------------------------------
 ->{q0}      | {q1,q3}   | {q2}
   {q1,q3}   | {qerror}  | {q2}
  #{q2}      | {q1,q3}   | {q2}
   {qerror}  | {qerror}  | {qerror}
```

El nombre del estado de error puede ser distinto a `qerror`. Si decides optar por esta segunda alternativa, la entrega de la práctica consistirá en un único fichero:

- `practica1.py`: programa Python para la minimización.

Se valorará que el código esté estructurado en funciones, documentado con comentarios y que la implementación sea eficiente.