Universidad de La Laguna

Grado en Ingeniería Informática

Computabilidad y Algoritmia

PRÁCTICA 6

Simulación de Autómatas Finitos

Factor de ponderación: 10

1. Objetivos

El objetivo de la práctica es consolidar los conocimientos adquiridos sobre Autómatas Finitos (FA, del inglés *Finite Automata*) al mismo tiempo que se continúan desarrollando capacidades para diseñar y desarrollar programas orientados a objetos en C++. Mientras que en la anterior práctica se utilizó una herramienta externa para simular autómatas finitos deterministas (DFAs) y automátas finitos no deterministas (NFAs), en esta práctica desarrollaremos nosotros mismos un programa que nos permita simular cualquier autómata finito especificado mediante un fichero de entrada. El programa leerá desde un fichero las característicass de un autómata finito y, a continuación, simulará el comportamiento del autómata para las cadenas que se den como entrada (también a través de un fichero de entrada).

Al igual que en las prácticas anteriores, se propone que el alumnado utilice este ejercicio para poner en práctica aspectos generales relacionados con el desarrollo de programas en C++. Algunos de estos principios que enfatizaremos en esta práctica serán los siguientes:

- **Programación orientada a objetos**: es fundamental identificar y definir clases y objetos que permitan modelar adecuadamente el escenario de trabajo que se plantea.
- **Diseño modular**: el programa debiera escribirse de modo que las diferentes funcionalidades que se precisen sean encapsuladas en métodos concretos cuya extensión textual se mantenga acotada.
- **Pautas de estilo**: es imprescindible ceñirse al formato para la escritura de programas en C++ que se propone en esta asignatura [7]. Algunos de los principales criterios de estilo ya se han descrito reiteradamente en las prácticas anteriores de esta asignatura.

- **Documentación**: se requiere que los comentarios del código fuente sigan el formato especificado por Doxygen [2]. Además, se sugiere seguir esta referencia [3] para mejorar la calidad de la documentación de su código fuente.
- Compilación de programas utilizando `make` [5, 6].

2. Autómatas finitos

Para reconocer lenguajes regulares existen dos tipos de autómatas: los automátas finitos deterministas (DFAs) y los autómatas finitos no deterministas (NFAs). De forma general, los autómatas finitos se definen con un conjunto de estados y un conjunto de transiciones entre estados que se producen a la entrada de símbolos pertenecientes a un determinado alfabeto.

Así pues, un *Autómata Finito* (FA) se caracteriza formalmente por una quíntupla $M \equiv (\Sigma, Q, s, F, \delta)$ donde cada uno de estos elementos tiene el siguiente significado:

- Σ es el *alfabeto de entrada* del autómata. Se trata del conjunto de símbolos que el autómata acepta como entradas. Esto es, las cadenas que forman parte del lenguaje reconocido por el autómata serán secuencias de símbolos sobre dicho alfabeto Σ.
- Q es el *conjunto* finito de los *estados* del autómata.
- s es el *estado inicial* o de arranque del autómata ($s \in Q$). Se trata de un estado distinguido y es único (no es un conjunto). El autómata se encuentra en este estado al comienzo de la ejecución.
- F es el *conjunto de estados finales* o de aceptación del autómata ($F \subseteq Q$).
- δ es la *función o relación de transición* que indica a qué estados transitar en función de los estados actuales y los símbolos de entrada.

  Para un DFA esta función se define como:

  $$\delta : Q \times \Sigma \to Q$$

  Para un NFA δ es una relación definida como:

  $$\delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$$

  Puesto que δ es una función, por lo que debe estar definida para todos los pares del producto cartesiano $Q \times \Sigma$. Por lo tanto, en un DFA la función de transición determina el único estado siguiente para un par (q, σ), con $q \in Q$ y $\sigma in \Sigma$. Por ello, sea cual fuere el símbolo actual de la entrada y el estado en que se encuentre el autómata, siempre hay un estado siguiente y además este estado se determina de forma unívoca. Dicho de otro modo, la función de transición siempre determina

unívocamente el estado al que ha de transitar el autómata dados el estado en que se encuentra y el símbolo que se tiene en la entrada.

Sin embargo, la función de transición en un NFA asigna a un par (q, σ) un elemento de 2^Q (partes de Q, es decir, el conjunto de todos los subconjuntos de Q):

$$(q, \sigma) \rightarrow \{q_1, q_2, \ldots, q_n\} \;\; tal\ que \;\; q \in Q, \sigma \in (\Sigma \cup \{\epsilon\})$$

Por lo tanto, en un NFA, dados un estado $q \in Q$ y un símbolo $\sigma \in \Sigma$ tenemos un *conjunto* de estados alcanzables o siguientes y no únicamente un solo estado. Es decir, desde un estado y ante un determinado símbolo de entrada, podemos no tener un estado siguiente, o bien, podemos tener varios estados siguientes. Otra diferencia de los NFAs con respecto a los DFAs es que los NFAs pueden tener transiciones vacías (transiciones etiquetadas con la cadena vacía), permitiendo al autómata transitar de un estado a otro sin necesidad de entrada, esto es, sin consumir símbolos de la cadena de entrada.

Teniendo esto en cuenta lo anterior, todo DFA es un caso particular de NFA en el que hay uno y solo un estado siguiente desde cada estado y para cada símbolo de entrada: el conjunto de estados siguientes o alcanzables tendrá un único elemento para todos los pares (q, σ). Por lo tanto, cualquier DFA se podría representar como un NFA y es por ello que en esta práctica nos centraremos en simular el comportamiento de los NFAs.

Para simular el comportamiento de un NFA tendremos que partir del estado inicial e ir leyendo (uno a uno y de izquierda a derecha) los símbolos de la cadena de entrada para así determinar los estados siguientes a los que transitaremos. Esto implica que en cada momento tendremos un "conjunto" de estados actuales (inicialmente será solo uno, s) pero a medida que la simulación avanza, este conjunto podrá tener más de un elemento o incluso ninguno (conjunto vacío). Si se tratara de un DFA, entonces este conjunto tendrá siempre un elemento durante todo el proceso de simulación. Es importante recordar que la principal diferencia con respecto a un DFA es que un NFA permite que desde un estado se realicen cero, una o más transiciones mediante el mismo símbolo de entrada. Además, en un NFA podemos tener ϵ-transiciones que nos permitirían alcanzar desde algunos estados, otros estados, sin consumir ningún símbolo de la entrada.

Cuando el autómata finalice de analizar la cadena de entrada, tendremos que analizar el conjunto de estados actuales que tenemos en ese momento: si entre ellos se encuentra al menos un $q \in F$ se dirá que el autómata ha aceptado la cadena de entrada. En caso contrario, se dirá que el autómata ha rechazado la cadena de entrada.

3. Ejercicio práctico

Esta práctica consistirá en la realización de un programa escrito en C++ que lea desde un fichero las especificaciones de un autómata finito (FA) y, a continuación, simule el comportamiento del autómata para una serie de cadenas que se suministren como entrada.

El programa debe ejecutarse como:

```
$ ./p06_automata_simulator input.fa input.txt
```

Donde los dos parámetros pasados en la línea de comandos corresponden en este orden con:

- Un fichero de texto en el que figura la especificación de un autómata finito.
- Un fichero de texto con extensión `.txt` en el que figura una serie de cadenas (una cadena por línea) sobre el alfabeto del autómata especificado en el primero de los ficheros.

Tal y como hemos venido recomendando en las prácticas de la asignatura, el comportamiento del programa al ejecutarse en línea de comandos debiera ser similar al de los comandos de Unix. Así por ejemplo, si se ejecuta el programa sin parámetros, se debería obtener información sobre el uso correcto del simulador:

```
$ ./p06_automata_simulator
Modo de empleo: ./p06_automata_simulator input.fa input.txt
Pruebe 'p06_automata_simulator --help' para más información.
```

La opción `--help` en línea de comandos ha de producir que se imprima en pantalla un breve texto explicativo del funcionamiento del programa. Una información que puede ser de especial ayuda para los usuarios del simulador sería precisamente el formato del fichero que contendrá la especificación del autómata.

El autómata finito vendrá definido en un fichero de texto con extensión `.fa`. Los ficheros `.fa` deberán tener el siguiente formato:

- Línea 1: Símbolos del alfabeto separados por espacios.
- Línea 2: Número total de estados del autómata.
- Línea 3: Estado de arranque del autómata.
- A continuación figurará una línea para cada uno de los estados. Cada línea contendrá los siguientes números, separados entre sí por espacios en blanco:
    - Número identificador del estado. Los estados del autómata se representarán mediante números naturales. La numeración de los estados corresponderá a los primeros números comenzando en 0.

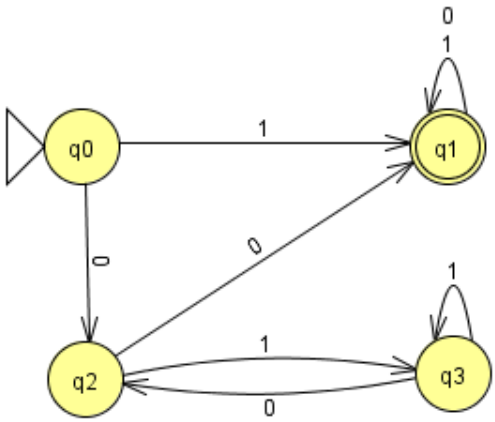

Figura 1: Especificación de un DFA de ejemplo

- Un 1 si se trata de un estado de aceptación y un 0 en caso contrario.
- Número de transiciones que posee el estado.
- A continuación, para cada una de las transiciones, y utilizando espacios en blanco como separadores, se detallará la información siguiente:
  - Símbolo de entrada necesario para que se produzca la transición. Para representar la cadena vacía (el no consumir símbolo de la entrada) se utilizará el carácter `&`
  - Estado destino de la transición.

A modo de ejemplo, en la Figura 1 se muestra un DFA que reconoce el lenguaje de las cadenas sobre el alfabeto $\Sigma = \{0, 1\}$ tales que comienzan por 1 o no tienen dos ceros consecutivos. Siguiendo el formato de ficheros `.fa` descripto anteriormente, este DFA se especificaría de la siguiente manera:

```
0 1
4
0
0 0 2 0 2 1 1
1 1 2 0 1 1 1
2 0 2 0 1 1 3
3 0 2 0 2 1 3
```

Otro ejemplo sería de autómata sería el NFA de la Figura 2. Este autómata reconoce el lenguaje formado por todas las cadenas sobre el alfabeto $\Sigma = \{0, 1\}$ tales que tienen un 1 en la antepenúltima posición. De manera análoga, y siguiendo el formato de ficheros `.fa` descripto anteriormente, este NFA se especificaría de la siguiente manera:

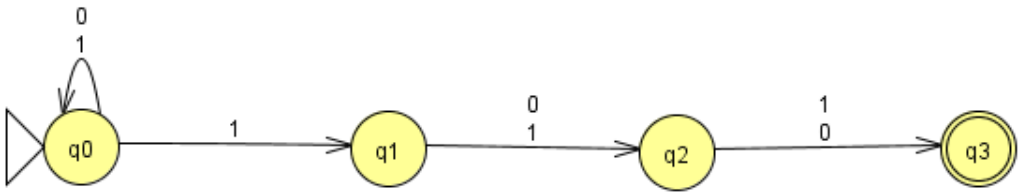

Figura 2: Especificación de un NFA de ejemplo

```
0 1
4
0
0 0 3 0 0 1 0 1 1
1 0 2 0 2 1 2
2 0 2 0 3 1 3
3 1 0
```

El programa principal deberá crear un autómata a partir de la especificación dada en el fichero `.fa` pasado por línea de comandos. Habrá que notificar al usuario si se produce algún error en la creación del autómata. Esto es, el programa deberá detectar y notificar adecuadamente al usuario en caso de que se detecte algún error o inconsistencia en la definición del autómata. Algunos ejemplos de condiciones que debe cumplir todo fichero de especificación son:

- Existe uno y sólo un estado inicial para el autómata.
- Hay una línea en el fichero por cada uno de los estados del autómata. Esto implica que para aquellos estados que no tengan transiciones salientes, deberá indicarse en la línea correspondiente del fichero, que el estado en cuestión tiene cero transiciones.
- Los símbolos de las transiciones deben ser símbolos del alfabeto sobre el que se define el autómata.
- Los estados a los que se transita son estados que forman parte del propio autómata.

Si el autómata ha sido creado correctamente, entonces se procederá a leer una a una las cadenas contenidas en el fichero `.txt` de entrada. Para cada cadena en el fichero de entrada se deberá simular el comportamiento del autómata para determinar si dicha cadena de entrada es aceptada o rechazada por el autómata. Por ejemplo, dado el autómata de la Figura 2, si tuviéramos el fichero de entrada siguiente (considérese que se utiliza el símbolo `&` para representar a la cadena vacía):

```
&
0
11
000
100
010101
1111
0000100
0101010
```

Entonces, el programa debería mostrar por pantalla una salida como la que se muestra a continuación:

```
& --- Rejected
0 --- Rejected
11 --- Rejected
000 --- Rejected
100 --- Accepted
010101 --- Accepted
1111 --- Accepted
0000100 --- Accepted
0101010 --- Rejected
```

4. Consideraciones de implementación

La idea central de la Programación Orientada a Objetos, OOP (*Object Oriented Programming*) es dividir los programas en piezas más pequeñas y hacer que cada pieza sea responsable de gestionar su propio estado. De este modo, el conocimiento sobre la forma en que funciona una pieza del programa puede mantenerse local a esa pieza. Alguien que trabaje en el resto del programa no tiene que recordar o incluso ser consciente de ese conocimiento. Siempre que estos detalles locales cambien, sólo el código directamente relacionado con esos detalles precisa ser actualizado.

Las diferentes piezas de un programa de este tipo interactúan entre sí a través de lo que se llama interfaces: conjuntos limitados de funciones que proporcionan una funcionalidad útil a un nivel más abstracto, ocultando su implementación precisa. Tales piezas que constituyen los programas se modelan usando objetos. Su interfaz consiste en un conjunto específico de métodos y atributos. Los atributos que forman parte de la interfaz se dicen públicos. Los que no deben ser visibles al código externo, se denominan privados. Separar la interfaz de la implementación es una buena idea. Se suele denominar encapsulamiento.

C++ es un lenguaje orientado a objetos. Si se hace programación orientada a objetos, los programas forzosamente han de contemplar objetos, y por tanto clases. Cualquier programa que se haga ha de modelar el problema que se aborda mediante la definición de clases y los correspondientes objetos. Los objetos pueden componerse: un objeto “coche”

está constituido por objetos "ruedas", "carrocería" o "motor". La herencia es otro potente mecanismo que permite hacer que las clases (objetos) herederas de una determinada clase posean todas las funcionalidades (métodos) y datos (atributos) de la clase de la que descienden. De forma inicial es más simple la composición de objetos que la herencia, pero ambos conceptos son de enorme utilidad a la hora de desarrollar aplicaciones complejas. Cuanto más compleja es la aplicación que se desarrolla mayor es el número de clases involucradas. En los programas que desarrollará en esta asignatura será frecuente la necesidad de componer objetos, mientras que la herencia tal vez tenga menos oportunidades de ser necesaria.

Tenga en cuenta las siguientes consideraciones:

- No traslade a su programa la notación que se utiliza en este documento, ni en la teoría de Autómatas Finitos. Por ejemplo, el conjunto de estados de su autómata no debiera almacenarse en una variable cuyo identificador sea `Q`. Al menos por dos razones: porque no sigue lo especificado en la *guía de estilo* [7] respecto a la elección de identificadores y más importante aún, porque no es significativo. No utilice identificadores de un único carácter, salvo para situaciones muy concretas.
- Favorezca el uso de las clases de la STL, particularmente `std::array`, `std::vector`, `std::set` o `std::string` frente al uso de estructuras dinámicas de memoria gestionadas a través de punteros.
- Construya su programa de forma incremental y depure las diferentes funcionalidades que vaya introduciendo.
- En el programa parece ineludible la necesidad de desarrollar una clase `Automata`. Estudie las componentes que definen a un autómata y vea cómo trasladar esas componentes a su clase.
- Valore análogamente qué otras clases se identifican en el marco del problema que se considera en este ejercicio. Estudie esta referencia [4] para practicar la identificación de clases y objetos en su programa.

5. Criterios de evaluación

Se señalan a continuación los aspectos más relevantes (la lista no es exhaustiva) que se tendrán en cuenta a la hora de evaluar esta práctica:

- Se valorará que el alumnado haya realizado, con anterioridad a la sesión de prácticas, y de forma efectiva, todas las tareas propuestas en este guión. Esto implicará que el programa compile y ejecute correctamente. Además, el comportamiento del programa deberá ajustarse a lo solicitado en este documento.
- También se valorará que, con anterioridad a la sesión de prácticas, el alumnado haya revisado los documentos que se enlazan desde este guión.
- Paradigma de programación orientada a objetos: el programa ha de ser fiel al paradigma de programación orientada a objetos. Se valorará que el alumnado haya identificado clases y objetos que permitan modelar adecuadamente el escenario de trabajo que se plantea.
- Paradigma de modularidad: se valorará que el programa se haya escrito de modo que las diferentes funcionalidades que se precisen hayan sido encapsuladas en métodos concretos cuya extensión textual se mantuviera acotada.
- Documentación: se requiere que todos los atributos de las clases definidas en el proyecto tengan un comentario descriptivo de la finalidad del atributo en cuestión. Además, se requiere que los comentarios del código fuente sigan el formato especificado por Doxygen [2].
- Se valorará que el código desarrollado siga el formato propuesto en esta asignatura para la escritura de programas en C++.
- Capacidad del programador(a) de introducir cambios en el programa desarrollado.

Si el alumnado tiene dudas respecto a cualquiera de estos aspectos, debiera acudir al foro de discusiones de la asignatura para plantearlas allí. Se espera que, a través de ese foro, el alumnado intercambie experiencias y conocimientos, ayudándose mutuamente a resolver dichas dudas. También el profesorado de la asignatura intervendrá en las discusiones que pudieran suscitarse, si fuera necesario.

Referencias

[1] Transparencias del Tema 2 de la asignatura: Autómatas finitos y lenguajes regulares, https://campusingenieriaytecnologia2223.ull.es/mod/resource/view.php?id=5919

[2] Doxygen http://www.doxygen.nl/index.html

[3] Diez consejos para mejorar tus comentarios de código fuente https://www.genbeta.com/desarrollo/diez-consejos-para-mejorar-tus-comentarios-de-codigo-fuente

[4] Cómo identificar clases y objetos http://www.comscigate.com/uml/DeitelUML/Deitel01/Deitel02/ch03.htm

[5] Makefile Tutorial: http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor

[6] C++ Makefile Tutorial: https://makefiletutorial.com

[7] Google C++ Style Guide, https://google.github.io/styleguide/cppguide.html