Universidad de La Laguna

Grado en Ingeniería Informática

Computabilidad y Algoritmia

PRÁCTICA 6

Simulación de DFAs

Factor de ponderación: 8

1. Objetivos

El objetivo de la práctica es consolidar los conocimientos adquiridos sobre Autómatas Finitos Deterministas (DFAs) 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 diseñó un autómata finito determinista (DFA) concreto (que permitía reconocer ciertos patrones presentes en una secuencia de ADN), en esta práctica desarrollaremos un programa que nos permita simular cualquier DFA especificado mediante un fichero de entrada. Esto es, el programa leerá desde un fichero las característicass de un DFA y, a continuación, simulará el comportamiento del autómata para las cadenas que se den como entrada.

Al igual que en las prácticas anteriores, también 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 deterministas

Un autómata finito determinista, AFD (o DFA de su denominación en inglés *Deterministic Finite Automaton*) es una máquina de estados finitos que acepta o rechaza una cadena de símbolos determinada, realizando una secuencia de transiciones entre sus estados, determinada únicamente por la cadena de símbolos. Si bien un DFA es un concepto matemático abstracto, con frecuencia se implementa en hardware y software para resolver diferentes problemas. Por ejemplo, un DFA puede modelar un software que decida si las entradas de un usuario en respuesta a una solicitud de dirección de correo electrónico son válidas o no.

Los DFAs reconocen exactamente el conjunto de lenguajes regulares, que son útiles entre otras finalidades, para hacer analizadores léxicos y detectores de patrones textuales.

Un DFA se define con un conjunto de estados y un conjunto de transiciones entre estados que se producen a la entrada de símbolos de entrada pertenecientes a un alfabeto Σ. Para cada símbolo de entrada hay exactamente una transición para cada estado. Hay un estado especial, denominado estado inicial o de arranque que es el estado en el que el autómata se encuentra inicialmente. Por otra parte, algunos estados se denominan finales o de aceptación. Así pues, un **Autómata Finito Determinista** se caracteriza formalmente por una quíntupla $M \equiv (\Sigma, Q, q_0, F, \delta)$ 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.
- Q es el *conjunto* finito de los *estados* del autómata. El autómata siempre se encontrará en uno de los estados de este conjunto.
- q_0 es el *estado inicial* o de arranque del autómata ($q_0 \in Q$). Se trata de un estado distinguido. El autómata se encuentra en este estado al comienzo de la ejecución.
- F es el *conjunto de estados finales* o de aceptación del autómata ($F \subseteq Q$). Al final de una ejecución, si el estado en que se encuentra el autómata es un estado final, se dirá que el autómata ha aceptado la cadena de símbolos de entrada.
- δ es la *función de transición*. $\delta : Q \times \Sigma \to Q$ que determina el único estado siguiente para un par (q_i, σ) correspondiente al estado actual y la entrada.

La característica esencial de un DFA es que δ es una función, por lo que debe estar definida para todos los pares del producto cartesiano $Q \times \Sigma$. 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.

El que sigue es un ejemplo de DFA definiendo los cinco elementos que lo componen:

1. $\Sigma = \{a, b\}$
2. $Q = \{q_0, q_1, q_2, q_3\}$
3. q_0
4. $F = \{q_1, q_2\}$
5. La función de transición, δ se define en la Tabla 1.

δ	a	b
q_0	q_1	q_3
q_1	q_1	q_2
q_2	q_1	q_3
q_3	q_3	q_3

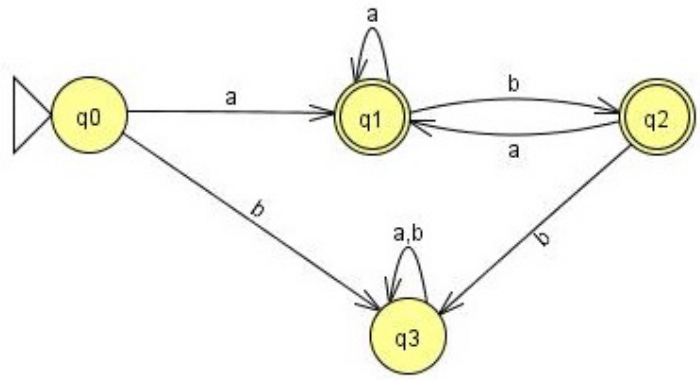
Tabla 1: La función de transición δ

En la Tabla 1, la primera columna contiene los estados posibles del autómata, y la primera fila todos los símbolos del alfabeto. Dado un estado actual q_i y un símbolo de entrada σ, la tabla define el estado al que transita el autómata cuando estando en el estado q_i, recibe la entrada σ: $\delta(q_i, \sigma) \in Q$

Con un DFA siempre se puede asociar un grafo dirigido que se denomina diagrama de transición. Su construcción es como sigue:

- Los vértices del grafo se corresponden con los estados del DFA.
- Si hay una transición desde el estado q hacia el estado p con la entrada i, entonces deberá haber un arco desde el nodo q hacia el nodo p etiquetado i.
- El estado de arranque se suele señalar mediante una flecha dirigida hacia ese nodo.
- Los estados de aceptación se suelen representar mediante un círculo de doble trazo.

A modo de ejemplo, en la Figura 1 se muestra el diagrama de transiciones para el DFA definido por la función de transición de la Tabla 1.



ejemplo.dfa

```
4
0
0 0 2 a 1 b 3
1 1 2 a 1 b 2
2 1 2 a 1 b 3
3 0 2 a 3 b 3
```

Figura 1: Diagrama de transiciones para un DFA de ejemplo

3. Ejercicio práctico

Desarrollar un programa que evalúe el reconocimiento de una serie de cadenas por parte de un DFA especificado a través de un fichero de entrada. El programa debe ejecutarse como:

```
$ ./dfa_simulation input.dfa input.txt output.txt
```

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

- Un fichero de texto en el que figura la especificación de un DFA.
- Un fichero de texto con extensión `.txt` en el que figura una serie de cadenas (una cadena por línea) sobre el alfabeto del DFA anterior.
- Un fichero de texto de salida con extensión `.txt` en el que el programa ha de escribir las mismas cadenas del fichero de entrada seguidas de un texto `-- Accepted / Rejected` indicativo de la aceptación (o no) de la cadena en cuestión.

El comportamiento del programa al ejecutarse en línea de comandos debiera ser similar al de los comandos de Unix. Este es un requisito que solicitaremos para todos los programas de prácticas de la asignatura a partir de ésta. Así por ejemplo, si se ejecuta sin parámetros el comando `grep` en una terminal Unix (se recomienda estudiar este programa) se obtiene:

```
$ grep
Modo de empleo: grep [OPCIÓN]... PATRÓN [FICHERO]...
Pruebe 'grep --help' para más información.
```

En el caso del programa a desarrollar:

```
$ ./dfa_simulation
```

```
Modo de empleo: ./dfa_simulation input.dfa input.txt output.txt
Pruebe 'dfa_simulation --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.

El autómata finito determinista vendrá definido en un fichero de texto con extensión `.dfa` y que tendrá el siguiente formato:

- Línea 1: Número total de estados del DFA.
- Línea 2: Estado de arranque del DFA.
- A continuación vendrá 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 enteros sin signo. La numeración de los estados corresponderá a los primeros números naturales comenzando por 0.
  - Un 1 si se trata de un estado de aceptación y un 0 si se trata de un estado de no aceptación.
  - Número de transiciones que posee el estado.
  - A continuación, para cada una de las transiciones, y separados por espacios en blanco, se detallará la información siguiente:
    - Símbolo de entrada necesario para que se produzca la transición.
    - Estado destino de la transición.

A modo de ejemplo, en la Figura 1 se muestra un autómata junto con la definición del mismo especificada mediante un fichero `.dfa`. El programa deberá detectar que no haya ningún error en la definición del autómata. Esto es, habría que analizar que se cumplen las siguientes condiciones:

- Existe uno y sólo un estado inicial para el autómata.
- Para cada estado del autómata siempre existe una y sólo una transición para cada uno de los símbolos del alfabeto.

El programa principal deberá crear un DFA a partir de la especificación dada en el fichero `.dfa` pasado por línea de comandos. Habrá que notificar al usuario si se produce algún error en la creación del DFA. 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 el DFA en cuestión acepta o rechaza dicha cadena de entrada. Por ejemplo, dado el autómata de la Figura 1, si tuviéramos el fichero de entrada siguiente

(considérese que se utiliza el símbolo `&` para representar a la cadena vacía):

```
&
a
b
aaaa
abab
bababa
```

Entonces, la salida sería la que se muestra a continuación:

```
& -- Rejected
a -- Accepted
b -- Rejected
aaaa -- Accepted
abab -- Accepted
bababa -- 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 cardinal del alfabeto de su autómata no debiera almacenarse en una variable cuyo identificador sea `N`. 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 `Dfa`. Estudie las componentes que definen a un DFA y vea cómo trasladar esas componentes a su clase `Dfa`.
- 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://campusingenieriaytecnologia2122.ull.es/mod/resource/view.php?id=4222

[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