Universidad de La Laguna

Grado en Ingeniería Informática

Computabilidad y Algoritmia

PRÁCTICA 7

Simulación de NFAs

Factor de ponderación: 9

1. Objetivos

El objetivo de la práctica es consolidar los conocimientos adquiridos sobre Autómatas Finitos No Deterministas (NFAs) 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 simulador para autómatas finitos deterministas (DFAs) en esta práctica desarrollaremos un programa que nos permita simular cualquier NFA especificado mediante un fichero de entrada. El programa leerá desde un fichero las característicass de un NFA 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 no deterministas

Un *Autómata Finito No Determinista*, AFN (o NFA de su denominación en inglés *Non-Deterministic Finite Automaton*) suele introducirse como una modificación de un Automáta Finito Determinista (DFA) en el que se permitirán cero, una o más transiciones desde un estado con un mismo símbolo del alfabeto de entrada. En este sentido, podríamos decir también que todo DFA es a su vez un NFA, pues tener desde cada estado una y sola una transición para cada símbolo del alfabeto no es más que un caso particular de lo anterior.

Formalmente, un *Autómata Finito No Determinista* se define por una quíntupla $(\Sigma, Q, q_0, F, \delta)$ 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 y no vacío 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. En este caso, y a diferencia de lo que ocurre en los DFAs, la función de transición es una relación:

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

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

  Esto es, se asigna a un par (q, σ) un elemento de 2^Q (partes de Q, es decir, el conjunto de todos los subconjuntos de Q).

3. Ejercicio práctico

Esta práctica consistirá en la realización de un programa escrito en C++ que lea desde un fichero las especificaciones de un NFA y, a continuación, simule el comportamiento del autómata para una serie de cadenas que se suministren como entrada. Tal y como ya se ha mencionado, se puede considerar un NFA como una modificación de un DFA en el que se permiten cero, una o varias transiciones desde un estado con un símbolo del alfabeto. 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. El no determinismo viene dado porque el autómata puede transitar con una misma entrada hacia un *conjunto* de diferentes estados. Este conjunto de estados puede incluso ser vacío.

El programa debe ejecutarse como:

```
$ ./p07_nfa_simulator input.nfa 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 NFA.
- Un fichero de texto con extensión `.txt` en el que figura una serie de cadenas (una cadena por línea) sobre el alfabeto del NFA 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.

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:

```
$ ./p07_nfa_simulator
Modo de empleo: ./p07_nfa_simulator input.nfa input.txt output.txt
Pruebe 'p07_nfa_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 no determinista vendrá definido en un fichero de texto con extensión `.nfa`. Los ficheros `.nfa` deberán tener el siguiente formato:

- Línea 1: Número total de estados del NFA
- Línea 2: Estado de arranque del NFA
- 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.
  - 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.

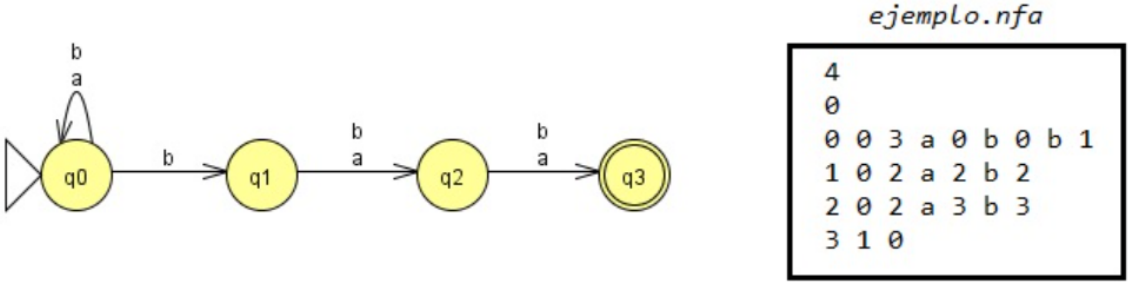

Figura 1: Especificación de un NFA de ejemplo

A modo de ejemplo, en la Figura 1 se muestra un autómata junto con la definición del mismo especificada mediante un fichero `.nfa`. El programa deberá detectar (y notificar al usuario de forma adecuada) si existe algún 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.
- 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.

El programa principal deberá crear un NFA a partir de la especificación dada en el fichero `.nfa` pasado por línea de comandos. Habrá que notificar al usuario si se produce algún error en la creación del NFA. 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 NFA 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
bb
abb
bab
bbbb
ababaa
```

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

```
& -- Rejected
a -- Rejected
bb -- Rejected
abb -- Rejected
bab -- Accepted
bbbb -- Accepted
ababaa -- Accepted
```

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 `Nfa`. Estudie las componentes que definen a un NFA y vea cómo trasladar esas componentes a su clase `Nfa`. Al realizar este análisis debería tener muy en cuenta la clase `Dfa` desarrollada durante la práctica anterior.
- 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