Universidad de La Laguna

ESIT – INFORMÁTICA
Departamento de Ingeniería Informática y de Sistemas

Computabilidad y Algoritmia
Curso 2020-2021

PRACTICA 8: Autómatas Finitos No Deterministas

8.1. Objetivos

- Consolidar los conocimientos adquiridos sobre Autómatas Finitos No Deterministas (NFAs).
- Implementar en C++ una clase para representar NFAs.
- Profundizar en las capacidades de diseñar y desarrollar programas orientados a objetos en C++.

8.2. Rúbrica de evaluación del ejercicio

Se señalan a continuación los aspectos más relevantes (la lista no es exhaustiva) que el profesorado tendrá en cuenta a la hora de evaluar el trabajo que el alumnado presentará en la sesión de evaluación de la práctica:

- El alumnado ha de acreditar conocimientos para trabajar con la shell de GNU/Linux en su VM.
- El alumnado ha de acreditar capacidad para editar ficheros remotos en la VM de la asignatura usando VSC.
- El código ha de estar escrito de acuerdo al estándar de la guía de Estilo de Google para C++
- El programa desarrollado deberá compilarse utilizando la herramienta make y un fichero Makefile, separando los ficheros de declaración (`*.h`) y definición (`*.cc`) de clases.
- El comportamiento del programa debe ajustarse a lo solicitado en este documento.
- Ha de acreditarse capacidad para introducir cambios en el programa desarrollado.

- Modularidad: el programa ha de escribirse de modo que las diferentes funcionalidades que se precisen sean encapsuladas en funciones y/o métodos cuya extensión textual se mantenga acotada.
- El programa ha de ser fiel al paradigma de programación orientada a objetos.
- Se requiere que, en la sesión de evaluación de la misma, todos los ficheros con códido fuente se alojen en un único directorio junto con el fichero Makefile de compilación.
- Se requiere que todos los atributos de las clases definidas en el proyecto tengan un comentario descriptivo de la finalidad del atributo en cuestión.
- Se requiere que los comentarios del código fuente sigan el formato especificado por Doxygen [1].
- Utilice asimismo esta [2] referencia para mejorar la calidad de la documentación de su código fuente.

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.

8.3. Introducción

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.

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).

8.4. Objetivo

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 cadena que se suministre 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.

8.5. Especificación de NFAs

Un NFA vendrá definido en un fichero de texto con extensión `.nfa`. Un fichero `.nfa` deberá 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 (código ASCII 126).
- Estado destino de la transición.

A modo de ejemplo, en la Figura 8.1 se muestra un NFA junto con la definición del mismo especificada mediante un fichero `.nfa`

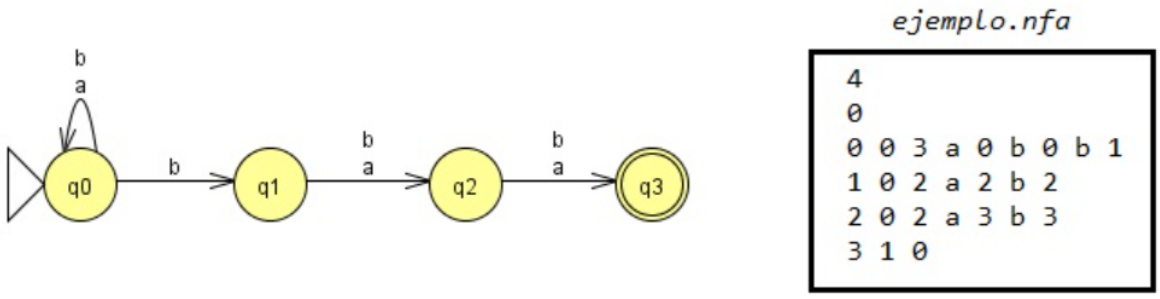

Figura 8.1: Especificación de un NFA de ejemplo

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.

8.6. Funcionamiento del programa

El programa debe ejecutarse como:

```
$ ./nfa_simulation 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 con extensión `.nfa` 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 especificado en el fichero `input.nfa`.

- 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 `-- Sí / No` indicativo de la aceptación (o no) de la cadena en cuestión. Este fichero debe ser sobreescrito por el programa en cada ejecución.

Tal y como se ha solicitado en prácticas anteriores, 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 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:

```
$ ./nfa_simulation
Modo de empleo: ./nfa_simulation input.nfa input.txt output.txt
Pruebe 'nfa_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.

8.7. Detalles 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 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` 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`.
- Valore análogamente qué otras clases identifica Ud. en el marco del problema que se considera en este ejercicio. Estudie esta referencia [3] para practicar la identificación de clases y objetos en su programa.

Bibliografía

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

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

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