Universidad de La Laguna

Grado en Ingeniería Informática

Computabilidad y Algoritmia

PRÁCTICA 5

Autómatas finitos y reconocimiento de patrones

Factor de ponderación: 8

1. Objetivos

El objetivo de la práctica es trabajar con autómatas finitos deterministas en C++. En particular, se diseñará un autómata que permita reconocer ciertos patrones presentes en una secuencia de ADN. Mientras que en la anterior práctica se utilizaban expresiones regulares para detectar la aparición o no de ciertas cadenas en un texto (programa fuente C++), en este caso, se utilizará un autómata finito determinista para reconocer un determinado patrón sobre una secuencia de ADN.

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 [5]. Algunos de los principales

criterios de estilo ya se han descrito reiteradamente en las prácticas anteriores de esta asignatura.

- **Autómatas finitos**: se aplicarán los conceptos relacionados con autómatas finitos para diseñar un mecanismo que permita reconocer las subcadenas que casan con un determinado patrón.
- Compilación de programas utilizando `make` [3, 4].

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. Y precisamente por esta característica utilizaremos este tipo de mecanismos para identificar patrones en una secuencia de símbolos.

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.

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.

3. Ejercicio práctico

Para poner en práctica lo estudiado hasta el momento, se desarrollará un programa en C++ que implemente un Autómata Finito Determinista (AFD o DFA) para extraer determinados patrones a partir de una secuencia de ADN de entrada. Las secuencias de ADN estarán representadas por secuencias de símbolos sobre el alfabeto $\{A, C, G, T\}$. A partir de estas secuencias de ADN, nos interesará extraer todas las posibles subsecuencias de ADN que empiecen y terminen por una `A` o por una `T`, y que tengan longitud mayor o igual a 2.

Para llevar a cabo esta extracción de secuencias de ADN, lo primero que haremos será diseñar un DFA que sea capaz de reconocer si una determinada entrada pertenece al lenguaje representado por la siguiente expresión regular:

```
1 A (A|T|G|C)* A | T (A|T|G|C)* T
```

Una vez diseñado el DFA reconocedor de subsecuencias de ADN, tendremos que utilizarlo para que analice cada una de las posibles subsecuencias de ADN y determine si la subsecuencia en cuestión cumple o no con el patrón. En este sentido, es importante no confundir las subsecuencias de ADN (partes sobre una cadena completa de ADN) con el concepto de subsecuencia en la teoría de autómatas y lenguajes formales. El DFA hará el reconocimiento sobre todas las posibles subcadenas del ADN de entrada pero sólo nos interesará conocer cuáles son las que cumplen con el patrón especificado.

El programa recibirá por línea de comandos una secuencia de ADN y un nombre de fichero de salida donde se almacenarán las subsecuencias de ADN que cumplen con el patrón anterior:

```
$./p05_dna_sequencer dna_input dna_subsequences.out
```

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, la opción `--help` en línea de comandos ha de producir que se imprima en pantalla un breve texto explicativo del funcionamiento del programa.

Suponiendo que el programa se ejecute con la siguiente secuencia de ADN:

```
$./p05_dna_sequencer CATTTGCAGGTG dna_subsequences.out
```

En el fichero de salida, `dna_subsequences.out` deberíamos obtener lo siguiente:

```
TT
TTT
ATTTGCA
TGCAGGT
TTGCAGGT
TTTGCAGGT
```

Tal y como se puede apreciar, el programa debería ser capaz de extraer todos los patrones (subcadenas presentes en la secuencia de ADN) que encajan con la expresión regular especificada anteriormente. En el fichero de salida los patrones detectados se almacenarán en orden creciente en función a su longitud. A igual tamaño, los patrones se ordenarán en base a su posición relativa dentro de la cadena de ADN. Además, no se tendrán en cuenta los patrones duplicados. Por ejemplo, en la secuencia de ADN anterior se detectan dos secuencias `TT` pero sólo una de ellas se escribe a fichero. Una de las secuencias comienza en la tercera posición de la cadena de entrada, mientras que la otra comienza en la cuarta posición.

A continuación, incluimos otro ejemplo de funcionamiento del programa:

```
$./p05_dna_sequencer TTTTAAAA dna_subsequences.out
```

En este caso, en el fichero de salida, deberíamos obtener lo siguiente:

```
TT
AA
TTT
AAA
TTTT
AAAA
```

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* [5] 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 `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 [2] 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.
- 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: se valorará que el alumnado haya identificado clases y objetos que permitan modelar adecuadamente el escenario de trabajo que se planeta.
- 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.
- Uso de autómatas finitos para la detección de patrones. En este sentido es imprescindible diseñar y utilizar un DFA para el reconocimiento. **No se utilizarán expresiones regulares ni librerías específicas para tal fin**.
- 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] Cómo identificar clases y objetos http://www.comscigate.com/uml/DeitelUML/Deitel01/Deitel02/ch03.htm

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

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

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