Algorítmica y Modelos de Computación

Práctica 2

Autómatas Finitos

Universidad de Huelva

Realizado por: Álvaro Esteban Muñoz y Álvaro Díaz Rivero

Índice

Autómata Finito Determinista

Implementación del autómata

Para la implementación del autómata finito determinista se han elegido como datos a guardar:

- Un <Object> con el estado inicial para poder hacer uso del mismo al sacarlo del fichero, o en su defecto inicializarlo a '0' en caso de no usar un fichero.
- Un vector de <Object> con los estados finales.
- Una lista de <TransicionesAFD>, objecto definido en la clase con su mismo nombre (Usaremos la clase ArrayList de java para dicha lista).

Se ha decidido por sobrecargar el constructor de forma que se pueda hacer uso de uno al pasar un fichero y otro para pedir datos al usuario.

Los métodos usados en esta clase son los siguientes:

- **agregarTransición**: Este método es usado en ambos constructores para agregar las transiciones al atributo "transiciones" y usarlas para simular el comportamiento del autómata.
- **transición**: Gracias a este método podemos buscar dentro del atributo de las transiciones, el que concuerda con el estado y el símbolo aplicado para obtener el estado al que pasaría el autómata.
- **esFinal**: Nos permite saber si un estado es un estado final
- **reconocer**: Es la función última del autómata y la que es dada al usuario para que vea su comportamiento. Comprueba la cadena pasada por parámetro y dice si esta es aceptada o no por el autómata.

La funcionalidad de los métodos viene mejor definida en el javadoc, aquí solo se pretende dejar claro las decisiones que hemos decidido tomar mediante la breve exposición de los mismos.

TransicionesAFD

Para llevar un mejor control del autómata se han definido las transiciones del mismo como objetos diferentes, de esta forma, una transición para el AFD quedaría definida por los siguientes atributos:

- Object estado_origen
- Object estado_destino
- Object simbolo

Autómata Finito No Determinista

Implementación del autómata

Para la implementación del autómata finito no determinista se han tomado los atributos del AFD pero realizando algunos cambios:

- Tanto el atributo con el estado inicial como el atributo con los estados finales se mantienen iguales
- Una lista de <TransicionesAFND>, se diferencia en el tipo de objeto que guarda que son transiciones específicas del AFND.
- Una lista de <TransicionesLambda> que guardará aquellas transiciones que no requieran de input para cambiar de estado.

Se ha decidido por sobrecargar el constructor de forma que se pueda hacer uso de uno al pasar un fichero y otro para pedir datos al usuario.

Los métodos usados en esta clase son los siguientes:

- **agregarTransición** y **agregarTransiciónLambda** son prácticamente iguales que **agregarTrancisión** del AFD.
- **transición** se mantiene igual, pero hemos añadido un método sobrecargado cambiando el parámetro del estado por un macroestado, es decir, un array de estados. Será este último método el que uso al método transición para estados unitarios.
- **esFinal**: Funciona de la misma forma que en el AFD pero hemos sobrecargado el método con una versión para macroestados que usa este mismo método.
- **reconocer**: funciona de la misma forma que en el AFD
- **lambda_clausura**: Este método será necesario para sacar la lamba clausura a la hora de sacar el estado inicial.

TransicionesAFND

Al igual que en el AFD, para llevar un mejor control de las transiciones del autómata se han separado con objetos diferentes, estas transiciones se diferencian de las del AFD en que el estado destino puede ser multiple.

- Object estado_origen
- Object[ ] estado_destino
- Object símbolo

TransicionesLambda

Para poder llevar el control de las transiciones lambda se ha creado también un objeto a parte que se diferenciará de los anteriores en que no existe el atributo “símbolo”, es decir, quedaría definido por los siguientes atributos:

- Object estado_origen
- Object[ ] estado_destino

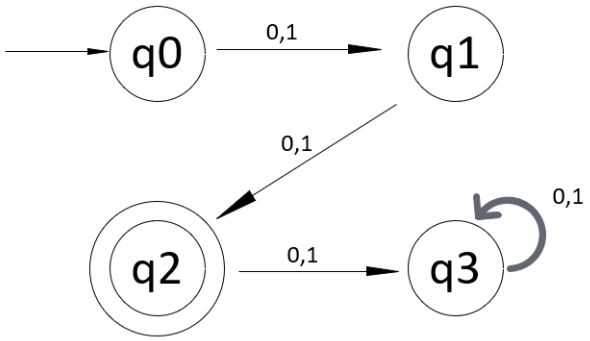
***Toda la información sobre la implementación de las clases usadas para la práctica se encuentra recogida de forma más explicativa en el javadoc de la aplicación.**

Autómatas usados en las pruebas

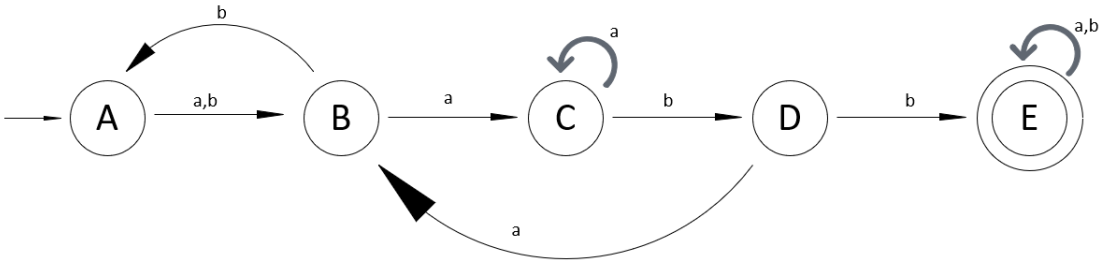
Para realizar las pruebas de funcionamiento hemos usado el siguiente conjunto de autómatas, todos vienen recogidos en sus respectivos ficheros dentro de la carpeta del proyecto. Estos autómatas pueden ser cargados dentro de la aplicación para probar su correcto funcionamiento.

Autómatas deterministas diseñados

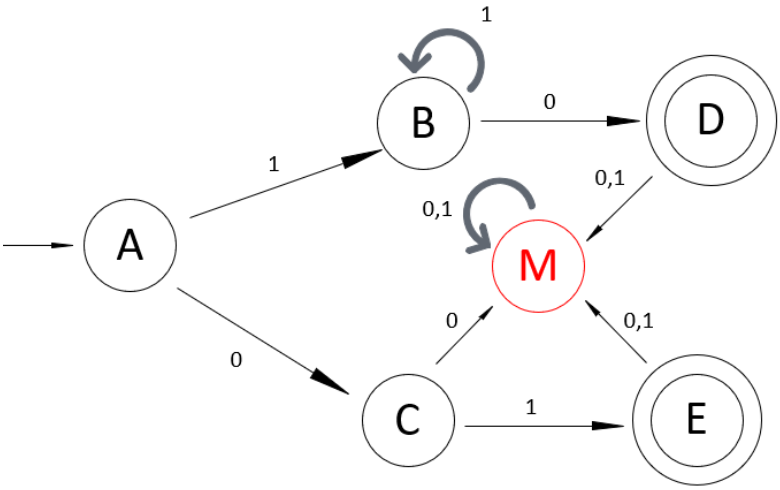
Acepta solo las cadenas de tamaño 2



Acepta todas las cadenas que contengan “aabb” en ellas.

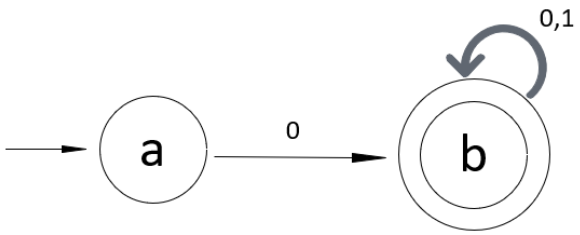


Acepta la cadena “01” o cualquier cadena que contenga al menos un ‘1’ seguido de un ‘0’ (En este ejemplo se incluye un “**estado muerto”** → M)

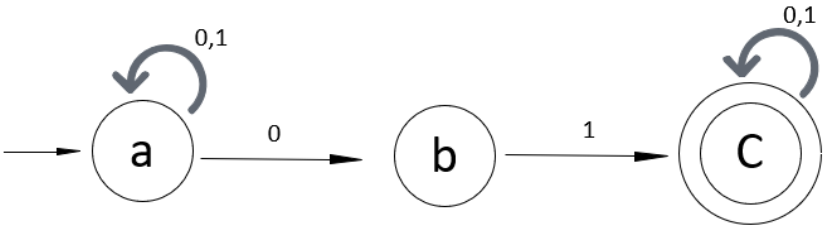


Autómatas no deterministas diseñados

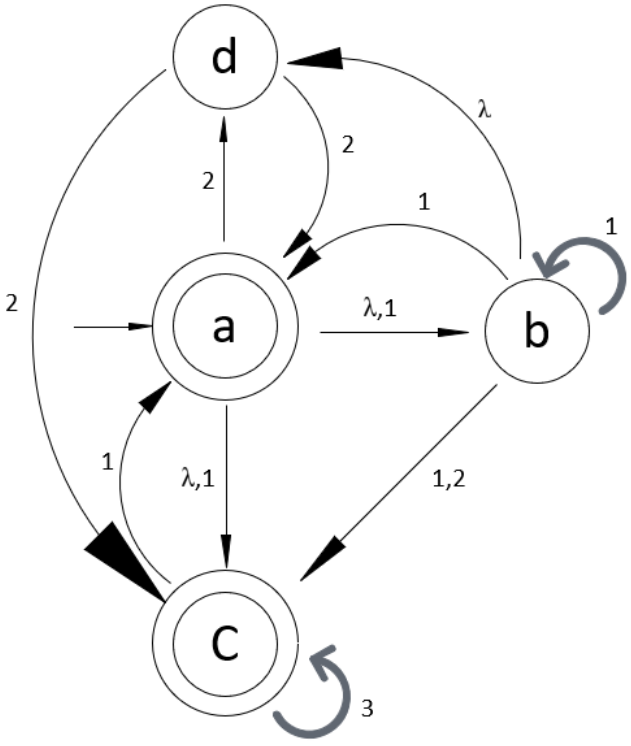
Acepta todas las cadenas que empiecen por ‘0’



Acepta todas las cadenas que contengan “01”



Ejemplo de autómata con transiciones lambda



Conclusiones y problemas encontrados

La simulación del autómata finito determinista no supone ninguna complicación, el autómata finito no determinista, sin embargo, trae algunos problemas para los cuales hemos tenido que pensar alguna forma estratégica de resolverlos.

1. Llevar la cuenta de todos los estados en los que nos encontramos en un determinado momento es complicado, la forma en la que hemos resuelto este problema es mediante un arrayList en el que añadiremos todos los estados destinos y que luego convertiremos a un array normal. De esta forma se hace muy sencillo seguir el estado en el que estamos.
2. Un problema encontrado en la implementación de ambos autómatas es la variedad de nomenclaturas usadas, para permitir que se pueda usar cualquier símbolo como estado y como letra del alfabeto hemos generalizado todos los atributos a Object, de esta forma, un estado puede ser tanto A, B, C, D, … como q0, q1, q2, q3, …
3. El constante cálculo de la lambda clausura para los diferentes estados es una operación que en largos autómatas resulta ineficiente, se podría barajar la posibilidad de que las lambdas clausuras de los estados se inicialicen junto al autómata y guardarlos en memoria para evitar la constante llamada al método.
4. El problema que más esfuerzo a conllevado para poder resolverlo ha sido el cálculo de la lambda clausura, que hemos necesitado hacer uso de recursividad para sacarla. Esto tratado de una forma no adecuada puede llegar a causar problemas de desbordamiento en la pila.

Para finalizar, podemos comentar que la práctica es bastante sencilla de implementar y produce resultados muy interesantes para analizar. Podría haber sido interesante añadir una memoria de pila para probar como simular un autómata con pila.

También se podía haber intentado diseñar algún autómata que reconociera algunas palabras reservadas concretas por el hecho de que queda más entendible para un humano ver como un autómata es capaz de reconocer palabras de su propio lenguaje.

Bibliografía

Theory of Computation & Automata Theory – Youtube Playlist by Neso Academy