Memoria práctica 2 AUTLEN

Borja Pérez Bernardos y Alejandro Pascual Pozo

1. Estructuras creadas

1.1 Lista de enteros

Reutilizamos la lista de enteros de la práctica anterior.

```
lista de enteros como estructura:
    Contiene “valores” como vector de enteros.
    Contiene “tamaño” como entero.
    Contiene “tamaño máximo” como entero.
```

1.2 Lista de estados

Utilizamos una versión simplificada de la lista de estados de la práctica anterior. Nos permite trabajar con estados formados por un conjunto de subestados. La utilizamos para agrupar estados equivalentes.

```
lista de estados como estructura:
    Contiene “subestados” como vector de lista de enteros.
    Contiene “tamaño” como entero.
    Contiene “tamaño máximo” como entero.
```

2. Pseudocódigo del algoritmo implementado

2.1 Resumen de los pasos a seguir

```
Inicializar “estados originales” a estados del AFD.
Inicializar “estados accesibles” a “estados originales” filtrados.
Inicializar “estados accesibles útiles” a “estados accesibles” filtrados.
Inicializar “estados reducidos” a “estados accesibles útiles” agrupados.
Construir “AFD reducido” con “estados reducidos”.
```

2.2 Filtrar los estados accesibles

```
Inicializar “estados accesibles” a estado inicial del AFD.
Para cada “estado” en “estados accesibles”:
  Para cada “estado accesible” en accesibles desde “estado”:
    Si “estado accesible” no está en “estados accesibles”:
      Añadir “estado accesible” a “estados accesibles”.
```

2.3 Filtrar los estados útiles

```
Inicializar “estados útiles” a estados finales del AFD.
Para cada “estado” en “estados útiles”:
  Para cada “estado útil” en accesibles desde “estado”:
    Si “estado útil” no está en “estados útiles”:
      Añadir “estado útil” a “estados útiles”.
```

2.4 Agrupar estados equivalentes

```
Inicializar “cociente actual” a estado finales y no finales del AFD.
Hacer:
  Inicializar “cociente siguiente” vacío.
  Para cada “clase actual” en “cociente actual”
    Para cada “estado actual” en “clase actual”:
      Para cada “clase siguiente” válida en “cociente siguiente”:
        Si cabeza de “clase siguiente” y “estado actual” son misma clase:
          Añadir “estado actual” a “clase siguiente”.
      Si “estado actual” no se ha añadido a ninguna clase:
        Inicializar “clase siguiente” a “estado actual”.
        Añadir “clase siguiente” a “cociente siguiente”.
  Almacenar tamaños de “cociente actual” y “cociente siguiente”.
  Asignar “cociente siguiente” a “cociente actual”.
Mientras el tam de “cociente actual” y “cociente siguiente” sea distinto.
```

2.5 Construir AFD reducido dados los estados reducidos

```
Inicializar “AFD reducido”.
Añadir “estados reducidos” a “AFD reducido”.
Añadir todos los símbolos de AFD a “AFD reducido”.
Para cada “estado origen” en estados de “AFD reducido”:
  Para cada “estado destino” en estados de “AFD reducido”:
    Para cada “símbolo” en símbolos de “AFD reducido”:
      Para cada “subestado origen” en “estado origen”:
        Para cada “subestado destino” en “estado destino”:
          Si de “subestado origen” a “subestado destino” con “símbolo”:
            AFD transición “estado origen” “símbolo” “estado destino”.
```

3. Decisiones de diseño más allá del enunciado

Nuestra implementación se caracteriza principalmente por dos aspectos:

- Hacemos un proceso equivalente al de encontrar los estados accesibles para los estados útiles. Llamamos estados útiles a aquellos para los que existe una secuencia de símbolos que, estando en dicho estado, te lleven a un estado final. Si un estado es inútil no puede marcar la diferencia entre aceptar o rechazar una cadena, por lo que lo eliminamos. Ver 2.3.
- Los estados equivalentes los identificamos con un método algo distinto al propuesto. Realizamos muchas menos comparaciones aprovechando la transitividad de la pertenencia a una clase. Comparamos los nuevos elementos solo con las cabezas de cada clase, sabiendo que el resultado será el mismo con el resto de los elementos. Ver 2.5.

4. Pruebas realizadas

Se han realizado pruebas con diferentes autómatas, comprobando que la salida es la misma para la versión reducida y la original.

Se adjuntan los ficheros necesarios para la compilación y ejecución de una serie de tests automatizados. Para ello se deberán ejecutar los siguientes comandos:

make

make run_tests

La salida es comprobada de manera automática, indicándose al final si los tests se han ejecutado correctamente.

Hemos elegido testear el autómata de las diapositivas. También hemos probado con autómatas centrados en cada una de las tres secciones: con estados intuitivamente inaccesibles, inútiles o equivales.

5. Correcto uso de la memoria

Hemos comprobado que no se produzcan pérdidas de memoria al utilizar nuestro programa. Se puede comprobar ejecutando los comandos:

make

make debug

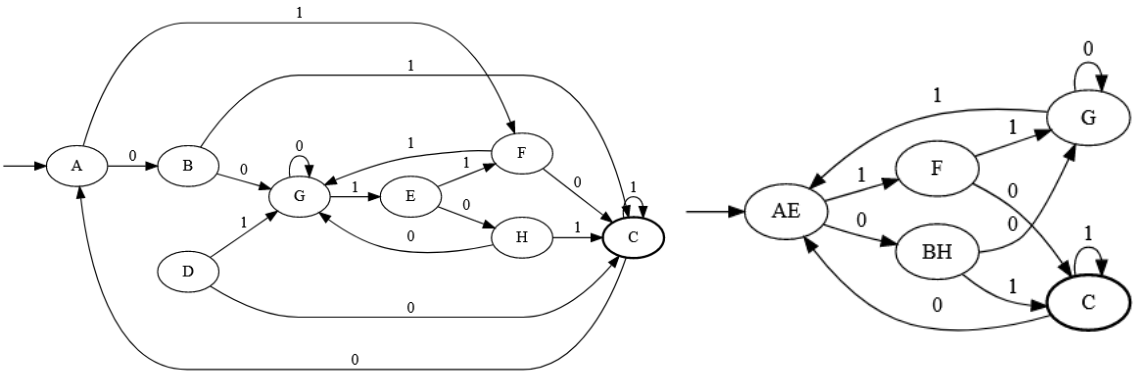
O para el caso de los tests:

make

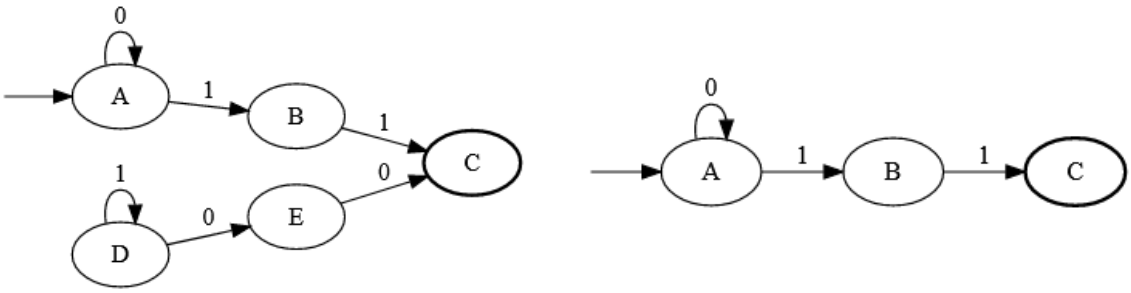
make debug_tests

6. Autómatas generados en las pruebas

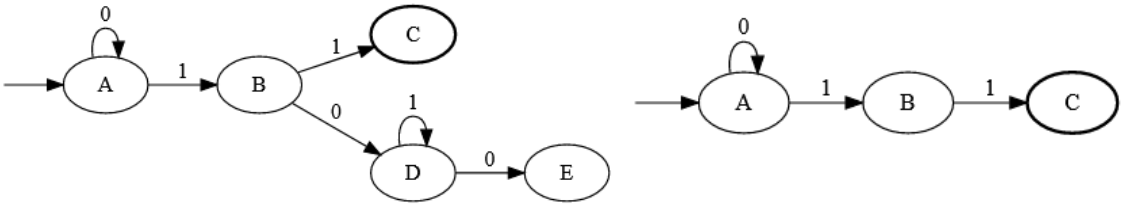
6.1 Autómata de las diapositivas



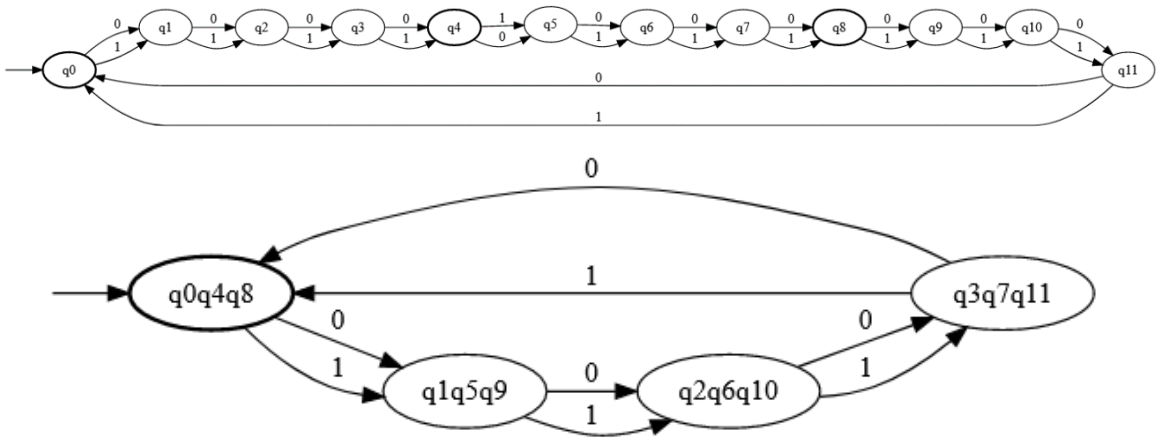
6.2 Autómata del lenguaje 0*11 con estados inaccesibles



6.3 Autómata del lenguaje 0*11 con estados inútiles



6.4 Autómata del lenguaje ((0+1) (0+1) (0+1) (0+1))* con 4 iteraciones



7. Cadenas probadas en las pruebas

7.1 Autómata de las diapositivas

Positivas: “01”, “011”, “01001” y “00101”.

Negativas: “0”, “11”, “000” y “0011”.

7.2 Autómata del lenguaje 0*11 con estados inaccesibles

Positivas: “11”, “011”, “0011” y “00011”.

Negativas: “0”, “1”, “00”, “01”, “10”, “000”, “111” y “100”.

Las cuatro cadenas de menor longitud aceptadas y diversas rechazadas.

7.3 Autómata del lenguaje 0*11 con estados inútiles

Mismas pruebas que el anterior.

7.4 Autómata del lenguaje ((0+1) (0+1) (0+1) (0+1))* con 4 iteraciones

Positivas: “0000”, “1111”, “0101”, “1010” y “01010101”.

Negativas: “0”, “1”, “00”, “11”, “000”, “111”, “00000” y “11111”.

Diversas cadenas, de las cuales una parte son de longitud múltiplo de cuatro.