MEMORIA PRACTICA 2: Minimización

Autores de la práctica:

- Sofía Sánchez Fuentes
- Jesús Daniel Franco López

Descripción de la práctica

En esta práctica hemos implementado el algoritmo para minimizar un autómata finito determinista cualquiera. Para ello utilizaremos una librería ya existente que utilizamos en la primera práctica de esta asignatura.

Descripción de los ficheros de la práctica

Para realizar esta práctica hemos hecho uso de diferentes ficheros y hemos creado también una batería de pruebas para comprobar distintos ejemplos del algoritmo implementado.

Contamos con *afnd.c* y *afnd.h* proporcionados por el personal docente de la asignatura donde se encuentran todas las funciones de la API y de las cuales haremos uso en el algoritmo implementado.

También contamos con *intermedia.c* e *intermedia.h* que contienen tanto la definición de la estructura intermedia implementada (la cual necesita también de otra estructura transición definida aquí) como las funciones necesarias para el uso de dicha estructura. Estos ficheros son los mismos que utilizamos en la práctica anterior.

Además, contamos con *minimiza.c* y *minimiza.h* donde se encuentra la función *AFDMinimiza* en la cual desarrollaremos e implementaremos el algoritmo para minimizar un AFD, así como las funciones auxiliares que hemos creado nosotros mismos.

Por último, contamos con diferentes ficheros para la batería de pruebas. Estos son prueba1.c, prueba2.c y prueba3.c prueba_todo_minimiza.sh. En los *.c encontramos los main donde se forman los diferentes AFD y se minimizan y el .sh es un script de prueba que nos permite minimizarlos.

Decisiones de diseño

En cuanto a las decisiones de diseño explicaremos las que hemos tomado a lo largo de la práctica para la implementación de manera modulada y sencilla de este algoritmo, así como el pseudocódigo del algoritmo.

Para esta práctica hemos decidido utilizar la misma estructura intermedia que en la práctica anterior. La estructura es la siguiente:

```
struct _Intermedia{
  char nombre_estado[MAX_NOMBRE]; /* nombre del estado del autómata determinista */
  int *i_codificacion; /* codificacion del estado del automata determinista */
  int tipo; /*tipo que va a ser: final, inicial, etc*/ Transicion
  *transiciones[MAX_TRANSICIONES]; /* transiciones con las que podemos ir a otros estados
                                  destinos */
};
```

Como podemos apreciar también contamos de nuevo con la estructura llamada transición que era de la siguiente manera:

```
struct _Transicion{
  char simbolo[MAX_NOMBRE]; /* simbolo con el que puedo ir al estado destino */
  char estado_final[MAX_NOMBRE]; /* nombre del estado destino */
  int *t_codificacion; /* codificacion del estado destino */
};
```

Como ya hemos dicho, hacemos uso de intermedia.c e intermedia.h con el fin de modularizar la funcion AFDMinimiza y dividir el problema en subproblemas.

A continuación, mostraremos el pseudocódigo del algoritmo que hemos utilizado para implementar la función AFDMinimiza. Primero, debemos saber que contamos con una lista de estructuras intermedias llamada *estados_nuevos* donde guardaremos todos los nuevos estados que obtendremos al ir realizando el algoritmo.

Antes de explicar el algoritmo final, explicaremos también los algoritmos empleados para obtener los estados accesibles y los distinguibles:

- Accesibles: Para este algoritmo haremos uso de dos listas de enteros *accesibles* y *creados*. Accesibles será una lista de 0's y 1's que representa con un 1 si la posición(estado) es accesible y con un 0 si no lo es. Creados es una lista de enteros donde guardaremos las posiciones (que representan los estados) para comprobar que no miramos de nuevo si es accesible o no; es decir, iremos insertando los siguientes a explorar en el caso de que no estén ya en esta lista.

```
Mientras que tengamos en creados nuevos estados del autómata:
   Para cada símbolo:
      Para otro_estado del autómata:
         Si hay transición desde estado a otro_estado con símbolo:
               Comprobamos que no esté en creados. Si no está en creados, lo metemos y
               accesibles[otro_estado] = 1
               Creados[numero creados] = otro_estado
         Si el estado es inicial o inicial y final
               accesibles[otro_estado] = 1
   Incrementamos el número de creados
Liberamos memoria reservada para creados
Devolvemos la lista de accesibles
```

- Distinguibles:

```
Marcamos en la matriz de distinguibles aquellos pares de estados donde almenos uno de ellos sea
final o inicial y final.
Flag = 1 → para que entre por primera vez en el while
Mientras que el flag este a 1:
   Flag = 0 → para comprobar si hay cambios en la matriz
   Miramos por pares (i,j) donde i != j si están marcados o no.
   Si no están marcados:
      Para cada símbolo:
         Para cada i_cada_estado del autómata:
            Miramos si hay transiciones desde i a i_cada_estado. Si hay
               New_pos1 = i_cada_estado
            Miramos si hay transiciones desde j a i_cada_estado. Si hay
               New_pos2 = i_cada_estado

         Si new_pos1 != new_pos2:
             Miramos si este nuevo par (new_pos1, new_pos2) esta marcado en la matriz de
             distinguibles. Si esta marcado:
                 Marcamos el par (i,j)
                 Ponemos el flag a 1 para indicar que hay un cambio
                 Hacemos un break

Devolvemos la matriz de distinguibles.
```

Pasamos a explicar el algoritmo de minimización

```
Obtenemos antes que nada la lista de estados accesibles y la matriz de distinguibles con TODOS
los estados del autómata inicial.
Para cada estado del a. inicial:
        Vemos si está en la lista de estados accesibles. En caso contrario pasamos al siguiente
        estado.
        Obtenemos su clase de equivalencia
        Comprobamos si ya la teníamos creada de algún estado anterior, y en caso contrario
                creamos su estructura intermedia asociada (el estado nuevo del a. minimizado) y
                la guardamos en la lista estados_nuevos

        En caso de no tenerla guardada previamente, tenemos que obtener sus transiciones.
                Obtenemos las transiciones por cada símbolo desde el estado que estamos
                comprobando, y obtenemos la clase de equivalencia de los estados a los que
                llega.
                Guardaremos en la estructura intermedia una transición a la clase de equivalencia
                que acabamos de sacar.

                Por ejemplo, si estamos en la iteración del estado A, obtenemos su clase de
                equivalencia (A-B). A partir de A, sacamos sus transiciones. Sabemos que A→H
                con un 1, así que sacamos la clase de equivalencia de H. (H-L-M). Concluimos
                que (A-B)→(H-L-M) con 1

Creamos ahora un nuevo autómata (el autómata minimizado) a partir de los estados de la lista
estados_nuevos y sus transiciones.
```

Cabe destacar que para la implementación de este algoritmo en AFDMinimiza hemos hecho uso de la API proporcionada puesto que facilita el trabajo. Además, también destacamos que el método de trabajo nos facilitó mucho la implementación de dicho algoritmo puesto que primero creamos las funciones para los estados accesibles, después para calcular la matriz de estados distinguibles y finalmente lo aplicamos en AFDMinimiza.

Banco de pruebas

Para comprobar el correcto funcionamiento de AFNDMinimiza hemos creado diferentes main's de prueba con distintos autómatas finitos deterministas que minimizaremos.

Hemos decidido realizar 3 pruebas diferentes. Para ello contamos, como ya hemos comentado anteriormente, con los ficheros prueba1.c, prueba2.c y prueba3.c.

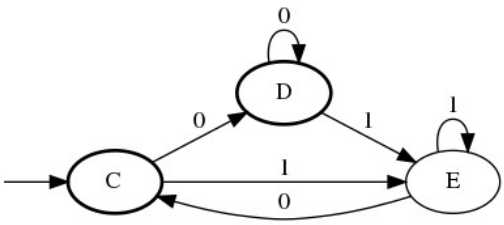
Para ejecutar las pruebas hemos creado el script *prueba_todo_minimiza.sh* el cual realiza make para generar los ejecutables de las pruebas, ejecuta dichas pruebas y genera la imagen final del autómata transformado con el nombre pruebaX.png correspondiente a cada prueba y finalmente realiza make clean. **Por lo tanto, para ejecutar el banco de pruebas proporcionado tan solo tendremos que introducir en la terminal el comando:**

bash pruebas.sh

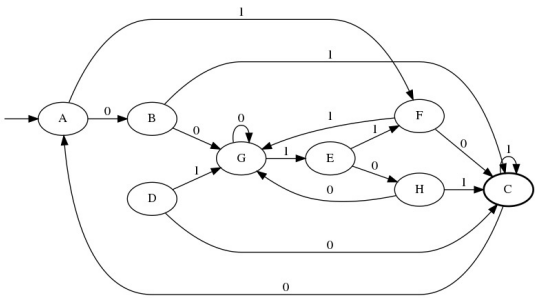
Si se decide probar cada ejercicio por separado, adjuntamos también pruebaX.sh para cada uno de los autómatas de prueba. Cada uno de estos scripts realiza un make clean al finalizar, por lo que si se desea comprobar el contenido del .dot habrá que hacer make y luego ejecutar la prueba en concreto ./pruebaX.

Los autómatas de las pruebas son los siguientes:

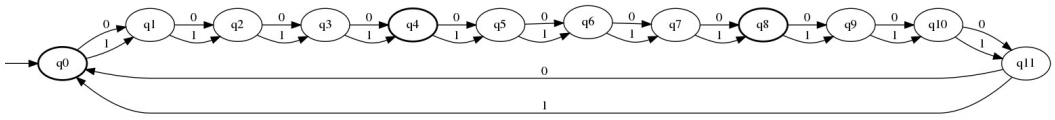
Prueba1:



Prueba 2:

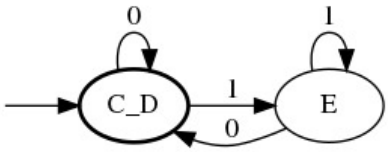


Prueba 3:

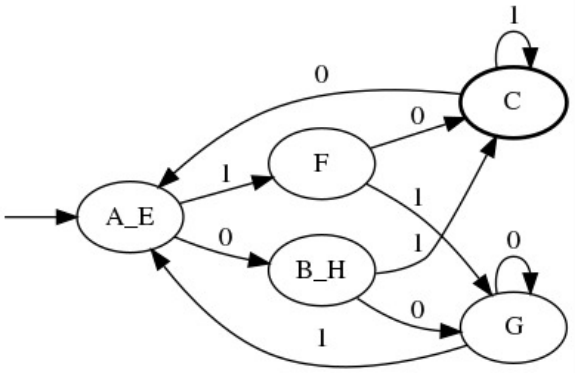


A continuación, se muestran los autómatas minimizados:

Prueba 1:



Prueba 2:



Prueba 3: