

MEMORIA PRÁCTICA 1: Conversión a determinista

Autores de la práctica:

- Sofía Sánchez Fuentes
- Jesús Daniel Franco López

Descripción de la práctica

En esta práctica implementaremos un algoritmo para convertir un autómata finito no determinista (AFND), en un autómata finito determinista (AFD). Para ello utilizaremos una librería ya existente, que incorpora además funciones para dibujar el autómata en formato DOT, de forma que podamos obtener una representación gráfica de los autómatas, llamada *afnd* que consta de los ficheros *afnd.c* y *afnd.h*.

Ficheros que componen 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 implmentado.

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.

Además, encontramos *transforma.c* y *transforma.h* que es donde se encuentra la función *AFNDTransforma* en la cual desarrollaremos e implementaremos el algoritmo para convertir de un AFND a un AFD.

Por último, contamos con diferentes ficheros para la batería de pruebas. Estos son *prueba1.c*, *prueba2.c*, *prueba3.c*, *prueba4.c*, *prueba5.c* y *pruebas.sh*. En los *.c encontramos los main donde se forman los diferentes AFND y se transforman a AFD y el .sh es un script de prueba que nos permite transformarlos.

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.

En primer lugar, nos pedían construir una estructura intermedia. Para nosotros dicha estructura son en sí los nuevos estados del AFD por lo que encontraremos que cuenta con los siguientes atributos: con un nombre del estado, con el tipo de estado que es (INICIAL, INICIAL_Y_FINAL, NORMAL o FINAL), con una lista de transiciones que me llevarán a diferentes estados destinos y una “codificación” para este estado que nos permitirá identificarle de manera sencilla. Dicha codificación será una lista de números enteros de la longitud del número de estados que tenga el AFND marcando con 1 la posición que queremos representar. De esta manera si nos encontramos que el AFND posee 5 estados y queremos identificar el estado q3 lo haremos guardando en esta lista [0,0,0,1,0] (teniendo en cuenta que los estados empiezan a enumerarse desde q0).

Mostramos la implementación de la estructura para que se pueda ver aquí de manera más sencilla y representativa:

```
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, encontramos otra estructura llamada Transición la cual también hemos decidido implementar nosotros. Esta contará con el símbolo que representa la transición, el nombre del estado destino al que puede ir y la “codificación” del estado destino al que puede ir. Hemos decidido crear esta estructura para que la comprensión de la estructura intermedia sea más intuitiva y sencilla, de manera que separemos lo relacionado con las transiciones y lo relacionado con el estado actual en sí.

Mostramos también la implementación de esta estructura para que se pueda ver de manera sencilla y representativa:

```
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 */
};
```

Con el objetivo de modularizar la función AFNDTransforma y dividir el problema en subproblemas, hemos implementado diferentes funciones para el uso de ambas estructuras que se encuentran en los ficheros *intermedia.c* e *intermedia.h*.

Hemos necesitado implementar getters y setters de atributos de las estructuras Intermedia y Transicion para poder trabajar con ellas desde el fichero transforma.h

Además tenemos varias funciones que nos imprimen por pantalla los atributos que contienen las estructuras. Las creamos con el objetivo de ir comprobando que las cosas se iban guardando correctamente y con el fin de poder seguir el algoritmo a medida que lo implementamos para ver que funcionaba de manera correcta. Finalmente, decidimos no quitarlas en la entrega puesto que ya las habíamos implementado y pueden resultar útiles.

A continuación, mostraremos el pseudocódigo del algoritmo que hemos utilizado para implementar la función AFNDTransforma:

Primero, debemos de saber que contamos con una lista de estructuras intermedias llamada *creados* donde guardaremos todos los nuevos estados que obtendremos al ir realizando el algoritmo.

Para el estado inicial comprobamos si hay transiciones lambda a otro estado del AFND
Mientras que tengamos en creados nuevos estados del AFND (son estructuras intermedias)
"estado" es el que tomamos de la lista creados

Para cada símbolo del AFND:

Para cada sub-estado de "estado":

Comprobamos si hay transiciones con el símbolo a otro estado del AFND

Se forma en "nuevo_estado" la codificación de los estados destino

Para cada sub-estado de "nuevo_estado":

Comprobamos si hay transiciones lambda a otro estado del AFND

Si hay transiciones:

Formamos el nombre del nuevo estado, creamos la transición y se la ponemos al estado que cogemos de la lista de creados

Comprobamos si el estado nuevo estado que queremos crear añadir ya se encuentra en creados

Si no está en creados: creamos la estructura intermedia, es decir, el nuevo estado y lo añadimos a la lista de creados

Si esta en creados: no hacemos nada

Formamos el nuevo AFD:

Insertamos los mismos símbolos que teníamos en AFND

Insertamos los nuevos estados de la lista de creados

Insertamos todas las transiciones que tenemos en los estados de la lista de creados

Cabe destacar que para la implementación de este algoritmo en AFNDTransforma hemos hecho uso de la API proporcionada puesto que facilita el trabajo. Además, podemos destacar también que nosotros decidimos implementarlo en primer lugar sin tener en cuenta las transiciones vacías. Una vez que tuvimos que el algoritmo funcionaba correctamente, pasamos a introducir las comprobaciones de las transiciones vacías. Este método de trabajo nos facilitó la implementación de este algoritmo.

Banco de pruebas

Para comprobar el correcto funcionamiento de AFNDTransforma hemos creado diferentes *main's* de prueba con distintos autómatas finitos no deterministas que convertiremos a finitos deterministas.

Hemos decidido realizar 5 pruebas diferentes para AFND con transiciones vacías y sin ellas. Para ello contamos, como ya hemos comentado anteriormente, con los ficheros *prueba1.c*, *prueba2.c*, *prueba3.c*, *prueba4.c* y *prueba5.c*.

Para ejecutar las pruebas hemos creado el script *pruebas.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
```

Las imágenes de los autómatas transformados se encuentran en una carpeta llamada *pruebas*.

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*.