Grupo Lunes 8:00-10:00 semanas B

- Práctica 4 -

Autor: Paula Soriano

NIP: 843710

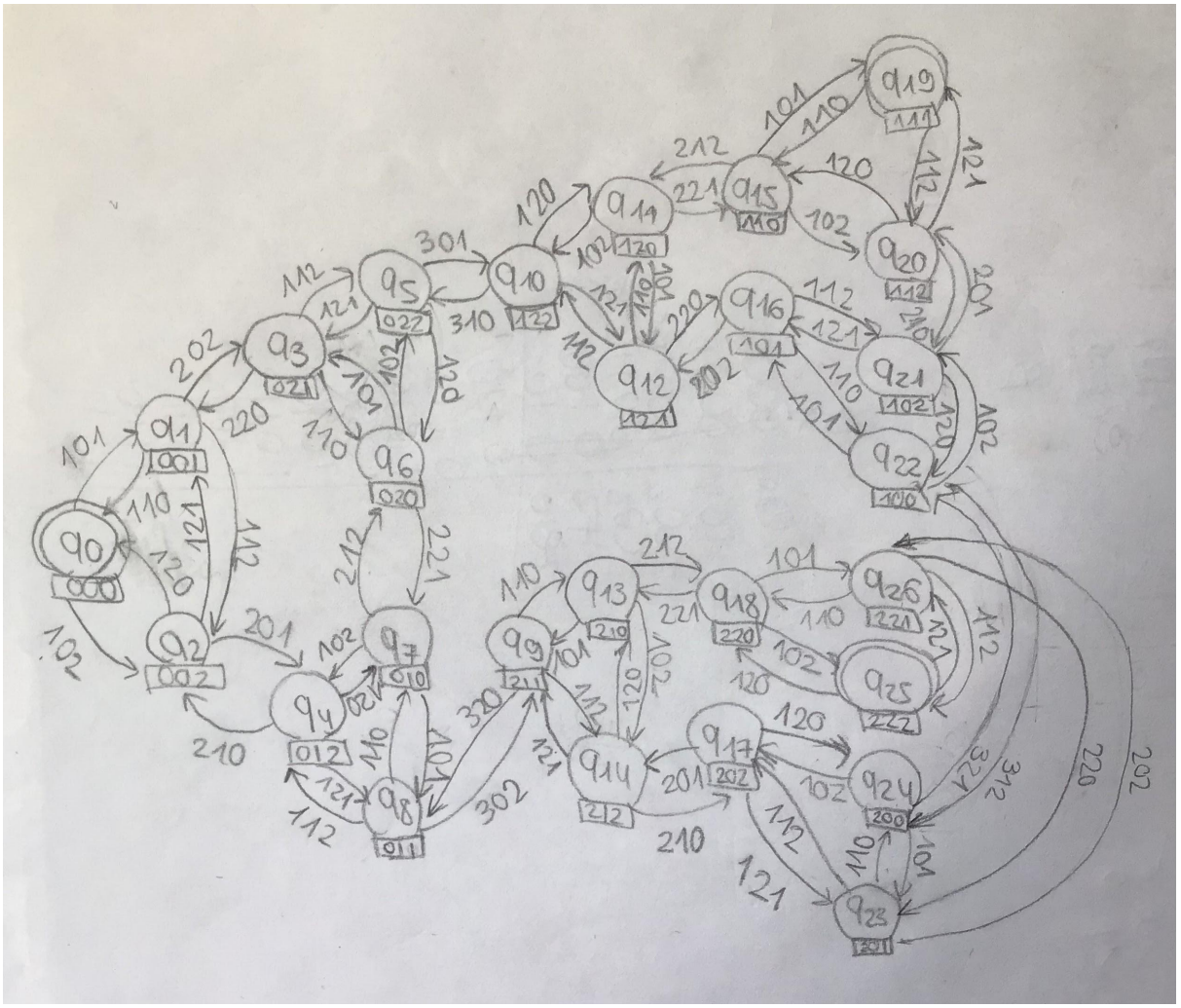
Autor: Carlota Moncasi

NIP: 839841

EJERCICIO 1.

A continuación se explica la codificación elegida para resolver el problema de las Torres de Hanoi para tres postes y tres discos.
Se conoce como estado regular cualquier distribución de los discos en los tres postes que satisfaga que todo disco está apoyado en un disco de mayor tamaño que él o sobre la placa base. Llamamos estado perfecto a un estado regular en el que todos los discos se encuentran apilados en el mismo poste.
Definimos el conjunto T como: T = {0, 1, 2} para etiquetar 0, 1 y 2 los postes del problema. El número total de discos lo hemos denominado como n > 0. Los discos los hemos etiquetado como 1, 2, . . . , n, con 1 el más pequeño, y asignando números crecientes con el orden creciente del diámetro de los discos.
Cada estado regular se representa de forma única con la n tupla s = sn . . . s1 ∈ Tn donde sk ∈ T es el poste en el que se encuentra el disco k. Por ejemplo, utilizando 3 discos, el estado perfecto en el que los tres discos se encuentran sobre el poste 0 se denota 000 = 0^3.

El autómata generado es el siguiente:



Para construir la especificación del problema de las Torres de Hanoi para 3 discos y 3 postes con ayuda de autómatas de estados finitos hemos seguido los siguientes pasos:

1. **Definición de los estados que pueden alcanzar el conjunto de postes y discos respetando las reglas de posicionamiento de los discos sobre los postes.**
   Los estados que pueden alcanzar el conjunto de postes y discos los hemos codificado de la siguiente manera: el primer dígito corresponde al número de poste en el que se encuentra el tercer disco, el segundo dígito al número de poste en el que se encuentra el segundo disco y el tercero al primer disco.
   Así pues, por ejemplo, el estado 001 corresponde a aquel estado en el que los discos 3 y 2 se encuentran en el poste codificado con el número 0 y el primer disco en el poste codificado con el número 1.

2. **Definición de las transiciones posibles entre estados.**
   El formato de las transiciones lo hemos definido de la siguiente manera: el primer dígito corresponde al número de disco cuya posición queremos modificar(1, 2 ó 3), el segundo, el número del poste en el que se encuentra actualmente el disco(0, 1 ó 2), y el tercero, el número del poste en el que queremos colocar el disco(0, 1 ó 2).

3. **Definición del estado inicial del autómata como la configuración de partida dada de los discos sobre los postes.**
   Hemos decretado como estado inicial el estado q0 (000), en el que los tres discos se encuentran en el primer poste (el codificado con el número 0).

4. **Definición del estado final del autómata como la configuración a alcanzar desde la de partida mediante movimientos de los discos conforme a las reglas establecidas.**
   Hemos designado como estados finales: q0(000), con el que los tres discos se posicionan en el primer poste (el codificado con el número 0), q19(111), con el que los tres discos se posicionan en el segundo poste (el codificado con el número 1) y q25 (222), con el que los tres discos se encuentran en el último poste (el codificado con el número 2).

EJERCICIO 2.

En este ejercicio se pide diseñar un lenguaje para la descripción de grafos. El lenguaje se formalizará mediante una gramática independiente de contexto a través de la cual se podrá construir una descripción textual de un grafo en un fichero de texto.
Hemos incluido en el fichero *thP3D3.txt* la descripción textual del autómata construido en el ejercicio 1. Esta descripción textual sigue el siguiente formato, de acuerdo con la codificación del autómata:
estadoOrigen ->estadoDestino1(transición),estadoDestino2(transición),...;

Se puede observar que, a excepción de los estados finales, los cuales tienen 2 transiciones (pues solo es posible mover el disco más pequeño), cada estado tiene 3 posibles transiciones que llevan a 3 estados diferentes (nunca al de origen).

Hemos elaborado una gramática para formalizar el lenguaje a través de la cual construimos la matriz de adyacencia.

En el fichero *th.y* hemos declarado los tokens siguientes, los cuales son reconocidos en el fichero *th.l*:

- **OP**: representa el carácter de apertura de paréntesis.
- **CP**: representa el carácter de cierre de paréntesis.
- **PC**: representa el carácter ";".
- **C**: representa el carácter ",".
- **EOL**: representa el carácter "\n", que es un salto de línea.
- **F**: representa los caracteres "->".
- **N**: representa uno o más dígitos comprendidos entre el 0 y 9.

La gramática que hemos desarrollado finalmente es la siguiente:

state → state origen F movements_list PC EOL
movements_list → movement | movement C movements_list
movement → N OP N CP
origen → N
F → ->
C → ,
PC → ;
EOL → \n
N → 0N | 1N | 2N | 3N | 4N | 5N | 6N | 7N | 8N | 9N | N epsilon

EJERCICIO 3.

Por un lado, hemos utilizado una función implementada por nosotras en el fichero *th.y* a la que hemos llamado *a_BaseDiez*, la cual sirve para convertir un número de base *PALOS* (en este caso es 3) a base decimal. Esta función es necesaria porque cada estado representa el número de filas de la matriz expresado en base PALOS, al igual que sucede con el número de columnas. A la función le pasamos como parámetro el entero n a modificar y su código consiste en:
una variable int *res* donde se devolverá el resultado, que inicializamos en n%10 (último dígito del número n). Después, recortamos el último dígito de n (n=n/10) y declaramos otra variable int *num* que usaremos para elevar *PALOS* a i. A continuación, utilizamos un bucle while(n!=0) para ir mirando cada dígito del número n, de derecha a izquierda.

Dentro del mismo, realizamos un bucle for para calcular *PALOS*^i, ya que, si tenemos por ejemplo n=331, debemos transformarlo a 3^0*1 + 3^1*3 + 3^2*3), así que empezando en i=1, con j=0 hasta i, multiplicamos *PALOS* por sí mismo, i veces. Después, en la variable *res* acumulamos *PALOS*^i * (n%10, que es el dígito en cuestión). Incrementamos i en uno y repetimos el bucle, hasta que no queden más dígitos en n por analizar, quedando *res* como:

PALOS^0*último dígito de n+ *PALOS*^1*penúltimo dígito de n + ...

Por otro lado, para determinar el mínimo número de movimientos desde un estado regular inicial a un estado regular destino a partir de la matriz de adyacencia que hemos construido, hemos utilizado el método basado en calcular las potencias de la matriz de adyacencia de un grafo.

Para ello, hemos diseñado una función de tipo void especificada en *th.y*, llamada *numMovimientos*, a la que le pasamos como parámetros el estado regular *origen* y el estado regular *destino* (ambos ya pasados a base diez) y la matriz de adyacencia del grafo, a la que hemos denominado *pot*.

En la función, hemos definido una variable local llamada *matriz*, que es una matriz que consta de *DIM* filas y *DIM* columnas de tipo puntero a carácter, y en la que hemos copiado la matriz que cuenta *tablaTr*. El punto clave de esta función consta de un bucle while cuya condición para mantenerse en este es que la celda *pot[origen][destino]* (siendo origen el nodo inicial y destino el nodo final) se encuentre vacía, lo que significa que no hay ningún movimiento posible para llegar desde el nodo origen hasta el nodo destino. Dentro de este bucle multiplicamos la matriz *pot* por sí misma y el resultado se le asigna a la matriz *pot*. que representa el conjunto de caminos de longitud k en el grafo que permiten alcanzar el estado regular *destino* desde el estado regular *origen*.

Para poder multiplicar la matriz *pot* por sí misma, ya que no se puede hacer directamente, multiplicamos las matrices *tablaTr* (que contiene la tabla de transición, y por tanto, el contenido inicial de *pot*) y *matriz* (que contiene el contenido actual de *pot*), y el resultado de dicha multiplicación queda guardado en *pot*. Además, dedicamos una línea más del interior del bucle a copiar la matriz *pot* en la matriz *matriz*, con la idea de que si hay una próxima iteración del bucle, la matriz *matriz* ya esté preparada, es decir, que cuente con el valor de *pot* actual. En el momento en el que salimos del bucle while significa que la celda *pot[origen][destino]* es no vacía y que por tanto hay un camino desde el cual se puede llegar al nodo final. El número de movimientos necesarios para llegar desde el nodo inicial hasta el nodo final se encuentran devueltos por la función en la matriz *pot* pasada como parámetro.

Los resultados obtenidos tras compilar y ejecutar nuestro programa han sido:
Nodo inicial: 000
Movimientos: 121-202-112
Nodo final: 222