Teoría de Lenguajes

Primer Cuatrimestre de 2015

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico 1

Grupo Estado Final

Integrante	LU	Correo electrónico
Montero, Victor	707/98	vmontero@dc.uba.ar
Román Gorojovsky	530/02	rgorojovsky@gmail.com
Lazzaro, Leonardo	147/05	lazzaroleonardo@gmail.com

Organización general del código.

El trabajo se realizó en *Python*. Se crearon tres módulos: `models` contiene la representación del autómata (clases `Automata` y `Node`), `parsers` contiene funciones para leer los archivos según los formatos definidos (regex y autómata) y convertirlos en objetos de clase `Automata`, y `writers` contiene funciones que escriben objetos de dicha clase en formato de texto o `DOT`.

Usando estas clases, se resolvieron los ejercicios en los archivos correspondientes, según el esqueleto provisto por la cátedra. Para cada ejercicio se implemento un breve unit test en `test/test_ejercicio_x`. También se implementaron unit tests para `models` y `parsers`. En algunos casos hay también un script `run_and_plot_ejercicio_x` que ejecuta el ejercicio y presenta gráficamente el resultado.

Clase `Automata`

La clase `Automata` contiene el estado inicial, la lista de estados, los símbolos y el conjunto de finales. Los estados están representados por la clase `Nodo`, y contienen un nombre y un diccionario cuyas claves son los símbolos y el valor un conjunto de estados a los que se puede transitar consumiendo ese símbolo. Claramente un nodo puede carecer de transiciones para un o más símbolo del lenguaje.

El autómata se puede crear pasando todos los elementos al constructor, o bien tan solo con el estado inicial y con una lista de estados finales, lo que nos permite mergear facilmente dos autómatas (por ejemplo solo juntando los finales con el inicial).

La clase `Automata` brinda también algunas funciones necesarias para resolver algunos de los ejercicios, que se discutirán oportunamente.

Parser de RegEx

Para parsear las expresiones regulares, se arma un diccionario donde la clave es la altura del árbol y el valor una lista de tuplas de la forma $\langle OPERADOR, VALOR \rangle$, donde $OPERADOR$ puede ser:

- $\{CONCAT\}$
- $\{STAR\}$
- $\{OR\}$
- $\{PLUS\}$
- $\{OPT\}$
- $\{SYMBOL\}$

y $VALOR$ puede ser la cantidad de operandos o el símbolo para el caso $\{SYMBOL\}$.

La altura del árbol la define la cantidad de tabs, y en el caso de que el tab es un símbolo en la segunda pasada del parser lo tratamos como un caso especial. Una vez obtenida esta estructura, la recorremos para armar autómatas que luego se van mergeando segun el algoritmo del ejercicio a (ver sección siguiente). No implementamos el algoritmo de mergeo, ya que pensamos a los estados como nodos y para juntar autómatas solo agregamos enlaces entre estos.

Ejercicio a: Minimizar autómatas

Para diseñar el algortimo que construye el autómata apartir de la expresión regular nos basamos en el teorema 3.7 del *Introduction to Automata Theory...* [1], que demuestra inductivamente que todo lenguaje definido por una expresión regular, está definido también por un autómata finito. Nuestro algortimo considera cada caso del la demostración y funciona recursivamente, tomando como los casos bases los correspondientes a la inducción.

Este algortimo nos devuelve un AFND-λ. Para pasar este autómata a AFD, utilizamos la idea que sugiere Hopcroft en la pagina 150: implementamos la función `lambda_closure`, la cual calcula, obviamente, la clausura lambda. Esto nos permite calcular la función delta del AFD que queremos obtener como resultado y, para obtener dado un conjunto de estados todos los estados alcazables usamos la función `move_set()`.

Pseudocodigo de algoritmo para convertir de AFND a AFD

```
def AFND_to_AFD(automata):
    clausura_inicial = lambda_closure(automata.initial)
    AFD_initial = DameNodo(clausura_inicial)
    states_queue = Queue()
    states_queue.put(AFD_initial)

    # aca construimos el nuevo delta
    mientras la queue no esta vacia:
        AFD_state = states_queue.get()
        for symbol in automata.symbols():
            current_state_closure=lambda_closure(automata.move_set(AFD_state.AFND_states, sy

            new_AFD_state=DameNodo(current_state_closure)

            AFD_state.add_transition(symbol, new_AFD_state)

    marcar_los_estados_finales()

    return res
```

Donde:

- Un nuevo nodo del AFD esta asociado a uno o mas nodos del AFND.
- `DameNodo()` se le pasa un conjunto de estados de la AFND y busca/crea el estado de la AFD que estamos creando, según corresponda. En la práctica, no implementamos este método en el código, pero aquí simplifica.
- `marcar_los_estados_finales()` revisa si todos los nuevos estados creados para la AFD estan asociados a algun nodo de la AFND que era final y si lo era lo marca como final.

Para minimizar usamos el algortimo que se dio en la clase práctica. En cuanto a tests, usamos los ejemplos del libro, el del enunciado del trabajo y uno un poco más largo hallado en wikipedia.

Ejercicio b: Decidir si una cadena pertenece al lenguaje de un autómata

La solucion a este ejercicio usa una de los métodos de la clase `Automata`, `move_sequence()`, que dada una cadena intenta "aplicarle" el autómata y devuelve un booleano indicando si fue se llegó a un estado final o no. Según este resultado se escribe `TRUE` o `FALSE`, según se pide en el enunciado. Para testearlo usamos el autómata que correspondería a la expresión regular `ac*(bf*)?`.

Ejercicio c: Escribir un autómata en formato `.dot`

Para la solucion de este ejercicio tampoco hay demasiada magia. Implementamos una función `save_dot()` en `writers` que toma un `Automata` y escribel el `.dot` correspondiente, y un módico test con un autómata sencillo, verificando el contenido archivo que se generaría.

Ejercicio d: Intersección de autómatas

El cálculo de la intersección fue el ejercicio que, innecesariamente, más dificultad nos causó, debido a que se trabajó primero con una definición errónea y luego con un ejemplo mal interpretado.

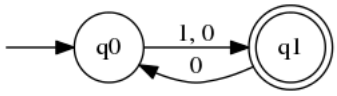
En un principio, dados dos autómatas $M_1 = \langle Q_1, \Sigma, q_0^1, \delta_1, F_1 \rangle$ y $M_2 = \langle Q_2, \Sigma, q_0^2, \delta_2, F_2 \rangle$ (ambos con el mismo alfabeto), definíamos la intersección M_I como:

$$M_I = \langle Q_1 \times Q_2, \Sigma, (q_0^1, q_0^2), \delta_I, \{(q_i^1, q_j^2)/q_i^1 \in F_1 \wedge q_j^2 \in F_2\} \rangle$$

donde

$$\delta_I((q_i^1, q_j^2), \mathrm{a}) = (q_k^1, q_l^2) <=> \delta_1(q_i^1, \mathrm{a}) = q_k^1 \wedge \delta_2(q_j^2, \mathrm{a}) = q_l^2$$

Esta definición genera muchos estados inútiles, es decir, inalcanzables partiendo del inicial consumiendo la misma cadena en los dos autómatas, por así decirlo. Por ejemplo, si tenemos el siguiente autómata:



y queremos calcular su intersección con si mismo, se va a generar un estado (q_0^1, q_1^2) que correspondería a "el primer autómata está en el estado 0 y el segundo en el estado 1", cosa que nunca puede ocurrir.

Para corregir esto, la definición del conjunto de estados en la intersección pasó a ser

$$\{(q_i^1, q_j^2)/(q_i^1, q_j^2) \in Q_1 \times Q_2 \wedge \exists \alpha \in \Sigma^*/\delta_1^*(q_0^1, \alpha) = q_i^1 \wedge \delta_2^*(q_0^2, \alpha) = q_j^2\}$$

Dicho esto, la implementación más sencilla es construír la primer versión y luego eliminar los nodos sobrantes, es decir, aquellos a los que no se llega partiendo del nodo inicial. En el ejemplo, los únicos nodos que quedarían son (q_0^1, q_0^2) y (q_1^1, q_1^2). Al resultado de esta construcción se le aplicó la minimización desarrollada en el ejercicio a.

La otra dificultad, como mencionamos, fue un ejemplo mal interpretado. La intersección entre "las cadenas que empiezan con 010" y "las cadenas que terminan con 010" no es "la cadena `'010'`" sino "las cadenas que empiezan y terminan con 010", y sin embargo eso fue lo que intentamos obtener como salida del algoritmo durante varias horas. Una vez resuelto el problema mental, quedó este ejemplo como uno de los tests, junto al caso evidente de la intersección de un autómata con si mismo que, efectivamente, genera el mismo autómata (usando el autómata del ejemplo) y la de dos autómatas cuya intersección es vacía, o mejor dicho dos autómatas que aceptan lenguajes cuya intersección es vacía.

Ejercicio e: Complemento de un autómata

El complemento de dos autómatas es más sencillo de definir: Dado $M = \langle Q, \Sigma, q0, \delta, F\rangle$, con su estado trampa qT, el complemento es $M^c = \langle Q, \Sigma, q0, \delta, Q - F\rangle$, y el código simplemente hace eso: define un nuevo autómata. Apenas tiene un test sencillo.

Ejercicio f: Equivalencia de autómatas

En este ejercicio explotamos el teorema 4.26 del *Introduction to Automata Theory...* [1], según el cual dado un autómata A y su minimización M, construída

según el algoritmo que estamos usando, entonces M tiene menos estados que cualquier autómata determinístico equivalente a A. Por lo que para decidir si dos autómatas son equivalentes:

- Minimizamos los dos autómatas
- Con un DFS recorremos determinísticamente el automata y renombramos los estados según ese orden.
- Verificamos si para cada estado del primer autómata, existe otro del mismo nombre y con tienen las mismas transiciones.

Para este ejercicio implementamos únicamente un unit test con el ejemplo que aparece en el libro.

Referencias

[1] Ullman J Hpcroft, J. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2001.