Matemáticas Computacionales: Generando lenguajes

Emilio E. G. Cantón Bermúdez

15/03/2019

Contexto

Se realizó un programa que reciba una definición recursiva de un lenguaje L y un número N, y obtenga el conjunto L_N de todos los *strings* que se generen aplicando N veces el paso recursivo. Se deben utilizar solo las letras válidas

Análisis

Definición de problema

Comenzamos por analizar de manera abstracta la cantidad de datos que vamos a recibir para estimar los datos que vamos a dar de output. De ésta manera sabemos que recibiremos un conjunto C de casos base y un conjunto de casos recursivos R. Definimos éstos conjuntos de la siguiente manera:

$$C = \{a_1, a_1, a_2, \cdots, a_n\}$$

Donde a_i para $1 \leq i \leq n$ es un caso base para el que puede ver $n \geq 1$ casos base.

$$R = \{r_1, r_1, r_2, \cdots, r_m\}$$

Así mismo r_j es un caso recursivo en donde $1 \geq j \geq m$ para cualquier $m \geq 1$.

Casos base

Primeramente podemos darnos cuenta que L_0 corresponde directamente con C, es decir $C \cong L_0$, ya que no se elabora el paso recursivo. Posteriormente damos razón que para L_1 se aplican m casos recursivos a n casos base que tomamos de

L_0. Definimos $r_j(l_{0,i})$ como la aplicación del caso recursivo $r_j \in R$ a $l_{0,i} \in L_0$ y elaboramos lo siguiente:

$N = 1$

$$r_1(l_{01}), r_2(l_{01}), r_3(l_{01}), \cdots, r_m(l_{01})$$
$$r_1(l_{02}), r_2(l_{02}), r_3(l_{02}), \cdots, r_m(l_{02})$$
$$r_1(l_{03}), r_2(l_{03}), r_3(l_{03}), \cdots, r_m(l_{03})$$
$$\vdots$$
$$r_1(l_{0n}), r_2(l_{0n}), r_3(l_{0n}), \cdots, r_m(l_{0n})$$

Observamos que se realizan $m \cdot n$ operaciones en este caso, al realizar un rectángulo de éstas dimensiones con los casos recursivos y base, respectivamente. Si continúamos de ésta misma manera...

$$r_1(r_1(l_{11})), r_2(r_1(l_{11})), \cdots, r_m(r_1(l_{11})$$
$$r_1(r_2(l_{11})), r_2(r_2(l_{11})), \cdots, r_m(r_2(l_{11})$$
$$\vdots$$
$$r_1(r_m(l_{11})), r_2(r_m(l_{11})), \cdots, r_m(r_m(l_{11})$$
$$r_1(r_1(l_{12})), r_2(r_1(l_{12})), \cdots, r_m(r_1(l_{12})$$
$$\vdots$$
$$r_1(r_m(l_{12})), r_2(r_m(l_{12})), \cdots, r_m(r_m(l_{12})$$
$$\vdots$$
$$r_1(r_m(l_{1n})), r_2(r_m(l_{1n})), \cdots, r_m(r_m(l_{1n})$$

Ahora cada caso de L_1 se realiza $m \cdot m$ veces. Lo que resulta en $m^N \cdot |L_{N-1}|$. Con esto podemos ver que si tomamos un ejemplo como:

Ejemplo

Valores exactos

$C = \{\lambda\}$

$R = \{uu0, w1u\}$

$N = 2$

$$L_0 = \{\lambda\} \Rightarrow |L_0| \leq m^0 \cdot |C| = 2^0 \cdot 1 = 1$$
$$L_1 = \{0, 1\} \Rightarrow |L_1| \leq m^1 \cdot |L_0| = 2^1 \cdot 1 = 2$$
$$L_2 = \{000, 110, 010, 111, 011, 01, 11, 10\} \Rightarrow |L_2| \leq m^2 \cdot |L_1| = 2^2 \cdot 2 = 8$$

Valores aproximados

$C = \{\lambda\}$

$R = \{uu0, w1u\}$

$N = 2$

$$L_0 = \{\lambda\} \Rightarrow |L_0| \leq m^0 \cdot |C| = 2^0 \cdot 1 = 1$$

$$L_1 = \{0, 1\} \Rightarrow |L_1| \leq m^1 \cdot |L_0| = 2^1 \cdot 1 = 2$$

$$L_2 = \{00, 10, 11\} \Rightarrow |L_2| \leq m^2 \cdot |L_1| = 2^2 \cdot 2 = 8$$

$$L_3 = \{000, 100, 110, 111\} \Rightarrow |L_3| \leq m^2 \cdot |L_2| = 2^3 \cdot 8 = 64$$

Esta aparente falla en la fórmula se debe a que es complicado calcular los valores que se repetirán en cada iteración, por lo tanto se puede perder exactitud muy rápidamente, sin embargo si realizamos el algoritmo en bruto (como se realizó en éste caso) éste número nos da el margen para trabajar.

Extra

Continúando el análisis de como éstos números crecen podemos observar que la fórmula puede deslindarse de lo recursivo, observemos que si para cualquier lenguaje con caso base λ:

Nota: Recordando la fórmula $m^N \cdot |conjuntoanterior|$.

$0 \geq N \geq 6$

$m = 2$

$$2^0 \cdot 1 = 1 = 2^0$$
$$2^1 \cdot 2 = 8 = 2^1$$
$$2^2 \cdot 8 = 64 = 2^3$$
$$2^3 \cdot 64 = 1024 = 2^6$$
$$2^4 \cdot 1024 = 32768 = 2^{10}$$
$$2^5 \cdot 32768 = \cdots = 2^{15}$$

Ahora intentamos otro ejemplo:

$0 \geq N \geq 4$

$m = 3$

$$3^0 \cdot 1 = 1 = 3^0$$
$$3^1 \cdot 3 = 9 = 3^1$$
$$3^2 \cdot 9 = 81 = 3^3$$
$$3^3 \cdot 27 = 729 = 3^6$$
$$3^4 \cdot 729 = 59049 = 3^{10}$$

Dadas las dos secuencias podemos ver que el número total de *strings* generadas por un lenguaje con caso base λ y con $m \in \mathbb{N}$ es menor o igual que $m^{\sum_{i=0}^{N}}$

Programa

r(u_int N)

Este método se encarga de preparar todo para realizar las combinaciones del paso recursivo y generar el lenguaje a partir de las permutaciones generadas por generate_combinations.

El `for` principal toma cada caso recursivo, `rs`, del conjunto R definido y lo separa entre las letras que son válidas como *strings* de acuerdo con lo definido (Contexto) y las guarda en un vector.

```
for (auto rs : R) {
    vector<string> s_matches;
    for (size_t i = 0; i < rs.length(); i++) {
        for (size_t j = 0; j < sizeof(valid); j++) {
            if (valid[j] == rs[i]) {
                stringstream ss;
                string s;
                ss << valid[j];
                ss >> s;
                s_matches.push_back(s);
            }
        }
    }
```

Luego, tomamos las combinaciones que genera generate_combinations e iteramos por ellas para poder generar las permutaciones. Las permutaciones se generan encontrando las posición del primer caracter correspondiente a la primera *string* dentro de la combinación en `combinations`, ésto se realiza hasta que el caracter se acabe ya que la palabra se repite para los caracteres similares, lo que no altera el númerod de combinaciones.

```
vector<vector<string>> combinations = generate_combinations(s_matches);
for (size_t i = 0; i < combinations.size(); i++) {
    string permutation = rs;
    for (size_t j = 0; j < s_matches.size(); j++) {
        int pos = permutation.find(s_matches[j]);
        while(pos != string::npos) {
            permutation.replace(pos, 1, combinations[i][j]);
            pos = permutation.find(s_matches[j]);
        }
    }
    strings_from_recursive_step.push_back(permutation);
}
```

Para teminar este método, tomamos las *strings* generadas y las insertamos (quitando a λ) en el mapa del lenguaje el cual contiene todas las strings como llave, es decir, sin repetición y con el valor de un entero que corresponde al paso recursivo que se tuvo que tomar para llegar a éste.

```
for (auto s : strings_from_recursive_step) {
    s = replace_lambda(s);
    language.insert(pair<string, int>(s, N));
}
```

generate_combinations

Dentro de éste código se realizan las posibles combinaciones a partir de los casos recursivos que se aplican. Iniciamos elevando el tamaño del lenguaje actual al número de variables que tenemos sin importar repetición, ya que luego se descarta.

```
int n_combinations = pow(language.size(), matches.size());
```

Creamos un vector bidimensional en el cual guardaremos éstas combinaciones y un vector donde tendremos el lenguaje actual por simplicidad. Consecutivamente calculamos el número que determina **cada cuánto cambia la *string* de valor en la columna correspondiente**, así como la string que actualmente se está utilizando del lenguaje.

```
vector<vector<string>> c(n_combinations, vector<string>(matches.size()));
vector<string> l;
for (auto const kv : language)
{
    l.push_back(kv.first);
}
    int loops = n_combinations / language.size();
    int curr_word = 0;
```

Finalmente, iteramos de arriba hacia abajo y de izquierda a derecha para generar todas las combinaciones. Ésto lo logramos al intercalar las *strings* del lenguaje en múltiplos del lenguaje. Como podemos ver se crean `n_combinations` de renglones con el número de variables que hay en cada caso recursivo. Dentro del loop tomamos en cuenta que tenemos que regresar al inicio si nos acabamos el lenguaje por lo que utilizamos modulos para lograrlo.

```
for (size_t j = 0; j < matches.size(); j++ {
    for (size_t i = 0; i < n_combinations; i++) {
        if (i % loops == 0 && i != 0) {
            curr_word = (curr_word + 1) % language.size();
        }
        c[i][j] = l[curr_word];
    }
    loops /= language.size();
}
return c;
```