Universidad Nacional de Colombia
Facultad de Ciencias - Departamento de Matemáticas
Topics on Advanced Algorithms - 2024 II

Mateo Sebastian Ortiz Higuera mortizhi@unal.edu.co

Un algoritmo para la solución de problemas en Gramáticas independientes del contexto utilizando Autómatas

Resumen

Las gramáticas independientes del contexto (CFG) son una herramienta fundamental para modelar estructuras mediante reglas de producción, con aplicación en lingüística, lenguajes de programación y compiladores, entre otros. Por esto, es de esperar que varios de los problemas asociados con el diseño de una CGF pueden llegar a ser importantes en el modelamiento de dichas estructuras. En este trabajo utilizamos algunas herramientas vistas en clase para presentar el conjunto pre^* asociado a una CGF y los algoritmos desarrollados por Rossmanith et al. [1], Bouajjani et al.[2] y por Esparza et al.[3] para construir autómatas no deterministas finitos (NFA) que permiten aceptar palabras que pertenecen a pre^*. Los cuales permiten determinar diversos problemas asociados al diseño de CFGs.

1. Introducción

Varios de los problemas asociados a gramáticas independientes del contexto como la prueba de pertenencia, de vacuidad, de infinitud, de variables inútiles, entre otros; son discutidos en gran parte de la literatura como problemas independientes relacionados al diseño de CFGs [4][5][6], los cuales pueden influenciar tanto en el modelamiento de estructuras con reglas de producción, por ejemplo el diseño de compiladores [7]; tanto en los algoritmos asociados a autómatas con pila y a autómatas finitos, por ejemplo la verificación de sistemas [8].

Aunque, en la práctica, solucionar dichos problemas asociados a las gramáticas depende del diseño inicial del CFG, en la parte teórica son problemas interesantes: en el trabajo *String-Rewriting Systems*, sus autores Book & Otto [9] recopilan información sobre dichos sistemas de reescritura de cadenas, en donde definen cuándo un sistema es monádico y cómo, de esta forma, es posible hallar un NFA que acepta el conjunto de los descendientes de un lenguaje regular L módulo el sistema. Dichas definiciones, teoremas y algoritmos presentados en su trabajo fueron interpretados por Rossmanith et al. [1] para definir el conjunto de palabras, con variables, cuyas generaciones llevan al lenguaje generado por una gramática independiente del contexto $L(G)$, conocido como $pre^*(L(G))$ y, tras eso, la implementación de un par de algoritmos, de complejidad $O(p^2s^6)$ y $O(ps^3)$, para crear una NFA que permite determinar cuando una palabra pertenece a dicho conjunto. Además de eso, logran dar cuenta de que, con dicho autómata, se pueden

solucionar varios de los problemas iniciales que parecían ser independientes. De dicho trabajo Bouajjani et al. [2] retoman la idea actualizando el segundo algoritmo, aún de complejidad $O(ps^3)$.

Recientemente, con los trabajos de Esparza et al. [3] y de Rossmanith [10] se han diseñado algoritmos que construyen dichas NFAs en un tiempo del orden $O(ps^3)$ y espacio del orden $O(ps^2)$.

2. Preliminares

Antes de hablar del algoritmo vamos a presentar algunas definiciones y teoremas que serán útiles.

Definición 2.1. *Una Gramática Independiente del Contexto (CFG) es una tupla $G = (V, \Sigma, S, P)$, donde V es un conjunto de variables, Σ es el alfabeto de símbolos terminales, $S \in V$ es la variable inicial y $P \subseteq V \times \Gamma^*$, donde $\Gamma = V \cup \Sigma$, es un conjunto finito de producciones, en donde $V \cap \Sigma = \emptyset$.*

Dados $\alpha, \beta \in \Gamma^*$, una producción es de la forma $\alpha \longrightarrow \beta$.

Una generación es una secuencia $S \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow w$ ($S \Rightarrow^* w$), en donde $w \in \Sigma^*$. Y notaremos $u \Rightarrow^+ v$ si v es obtenida de u por una o más producciones.

Además, se define el lenguaje generado por la gramática G se define como sigue

$$L(G) = \{w \in \Sigma^* : S \Rightarrow^+ w\}$$

En particular, dados $\alpha, \beta \in \Gamma$, si $\alpha \Rightarrow \beta$, diremos que α es un predecesor inmediato de β. Si $\alpha \Rightarrow^* \beta$, diremos que α es un predecesor de β. De esto podemos tener las siguientes dos definiciones.

Definición 2.2. *Dado un lenguaje $L \subseteq \Sigma^*$, definimos el conjunto de predecesores inmediatos de L como sigue:*

$$pre(L) = \{\alpha \in \Gamma^* : \exists \beta \in L \text{ tal que } \alpha \Rightarrow \beta\}$$

Además, también definimos $pre^i(L)$ inductivamente por $pre^0(L) = L$ y $pre^{i+1}(L) = pre(pre^i(L))$.

Definición 2.3. *Dado un lenguaje $L \subseteq \Sigma^*$, definimos el conjunto de predecesores $pre^*(L) = \bigcup_{i \geq 0} pre^i(L)$, o de forma equivalente*

$$pre^*(L) = \{\alpha \in \Gamma^* : \exists \beta \in L \text{ tal que } \alpha \Rightarrow^* \beta\}$$

Observe que bajo esta definición el lenguaje L pertenece a $pre^*(L)$.

Definición 2.4. *Un Autómata No Determinista Finito (NFA) es un tupla $A = (\Sigma, Q, q_0, F, \Delta)$, en donde Σ es un alfabeto, Q es un conjunto finito de estados, $q_0 \in Q$ es el estado inicial, $F \subseteq Q$ es el conjunto de estados de aceptación y Δ es la función de transición $\Delta : Q \times \Sigma \longrightarrow \mathcal{Q}$, donde $\mathcal{Q}$ es una familia finita de Q, es decir $\Delta(q, s) = \{q', q'', ..., q^n\}$*[1].

[1]En este trabajo definimos los NFA porque los algoritmos funcionan para dichos autómatas, sin embargo, vamos a utilizar como ejemplo un DFA a lo largo de la revisión.

La principal utilidad de un autómata es la de aceptar/rechazar cadenas, por lo que podemos considerar que todo lenguaje $L \subseteq \Sigma^*$ (a lo mucho recursivo) tiene asociado un autómata[2] con la capacidad de decidir cuando una cadena $u \in \Sigma^*$ pertenece o no a L.
Como ejemplo consideremos $G = (V, \Sigma, S, P)$, en donde $V = \{S, A\}$, $\Sigma = \{a, b\}$ y

$$P = \begin{cases} S \longrightarrow aS \mid bB \mid \varepsilon \\ B \longrightarrow bB \mid b \mid \varepsilon \end{cases}$$

Entonces algunas posibles generaciones son

$$S \Rightarrow aS \Rightarrow aaS \Rightarrow^* a \cdots aS \Rightarrow a \cdots a\varepsilon = a \cdots a$$
$$S \Rightarrow bB \Rightarrow bbB \Rightarrow^* b \cdots bB \Rightarrow b \cdots b\varepsilon = b \cdots b$$
$$S \Rightarrow aS \Rightarrow^* a \cdots aS \Rightarrow a \cdots abB \Rightarrow^* a \cdots ab \cdots bB \Rightarrow a \cdots ab \ldots b\varepsilon = a \cdots ab \cdots b$$
$$S \Rightarrow \varepsilon$$

Por lo que podemos ver que el lenguaje generado es $L(G) = a^*b^*$.
Tomando esto último consideremos $w = aabb \in L(G)$, que se puede generar con las siguientes producciones:

$$S \Rightarrow aS \Rightarrow aaS \Rightarrow aabB \Rightarrow aabb$$
$$S \Rightarrow aS \Rightarrow aaS \Rightarrow aabB \Rightarrow aabbB \Rightarrow aabb\varepsilon = aabb$$

entonces podemos calcular $pre(w)$ y $pre^*(w)$:

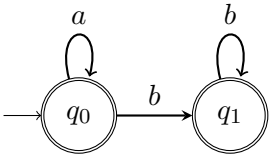
$$pre(w) = \{aabB, aabbB, aabbS\}$$
$$pre^*(w) = \{S, aS, aaS, aaSS, aaB, aaBB, aabB, aabS, aabbB, aabbS, \cdots, aabb\}$$

Y para $L(G) = a^*b^*$ tenemos

$$pre(L(G)) = \{a^n(S \cup B)^* : n \geq 0\} \cup \{b^n(S \cup B)^* : n \geq 1\} \cup \{a^n b^m(S \cup B)^* : n, m \geq 1\}$$

Sin embargo, no es trivial obtener el conjunto $pre^*(L(G))$.

Sabiendo que $L(G) = a^*b^*$, un autómata asociado que acepta dicho lenguaje generado está dado por $A = (\Sigma, Q, q_0, F, \Delta)$, donde $\Sigma = \{a, b\}$, $Q = \{q_0, q_1\}$, $F = Q$ y $\Delta(q_0, a) = q_0$, $\Delta(q_0, b) = q_1$ y $\Delta(q_1, b) = q_1$. Cuya representación en forma de grafo es de la forma:



[2]Considerando el tipo de lenguaje. Por ejemplo, los autómatas no deterministas finitos que hemos definido en este trabajo, solo poseen la capacidad de aceptar/rechazar lenguajes regulares. Sin embargo, los autómatas deterministas con pila pueden aceptar lenguajes independientes del contexto y los autómatas con doble pila pueden aceptar lenguajes recursivos.

Definición 2.5. *Una gramática $G = (V, \Sigma, S, P)$ se dice que es de la Forma Normal de Chomsky (FNC) si sus producciones son de la forma*

$$P = \begin{cases} A \longrightarrow BC, \ o \\ A \longrightarrow a, \ o \\ S \longrightarrow \varepsilon \end{cases}$$

donde $A, B, C \in V$ no necesariamente diferentes y $a \in \Sigma$.

Sin embargo, en el presente trabajo vamos a suponer que las gramáticas son de la forma normal de Chomsky con producciones unitarias y ε-producciones, es decir, $A \longrightarrow B$ y para cualquier $A \in V$, $A \longrightarrow \varepsilon$ respectivamente.

Teorema 2.6. *Para toda gramática independiente del contexto G, existe una gramática de la forma normal de Chomsky G' equivalente. Además, la transformación de G a G' toma un tiempo del orden $O(|P| + |V|)$.*

Demostración. Constructivamente, primero eliminamos las variables no terminales y las variables no alcanzables de G, estas dos cosas son del orden $O(|V|)$ cada una. Después eliminamos todas las producciones unitarias, es decir, de la forma $A \longrightarrow B$, estos dos pasos son del orden $O(|P|)$ cada uno.
Las producciones restantes son de la forma $A \longrightarrow a$ o $A \longrightarrow w$, con $|w| \geq 2$. Simulamos estas últimas producciones introduciendo para cada $a \in \Sigma$ una variable T_a con producción $T_a \longrightarrow a$. Y después de esto se introducen más variables con producciones binarias $T_j \longrightarrow T_{w_1} T_{w_2}$, que simulan el resto de producciones. Todo esto toma el tiempo del orden $O(|P|)$. □

Para verlo tomemos el ejemplo anterior, en donde $G = (V, \Sigma, S, P)$, en donde $V = \{S, B\}$, $\Sigma = \{a, b\}$ y

$$P = \begin{cases} S \longrightarrow aS \mid bB \mid \varepsilon \\ B \longrightarrow bB \mid b \mid \varepsilon \end{cases}$$

Primero eliminamos las variables no terminales y las variables no alcanzables, en este caso tanto S como B son terminales y alcanzables, por lo que la gramática sigue igual. Finalmente, para $\{a, b\}$ construimos T_a, T_b de tal forma que $T_a \longrightarrow a$ y $T_b \longrightarrow b$. Las producciones $S \longrightarrow aS \mid bB$ y $B \longrightarrow bB$ se simulan con $S \longrightarrow T_a S \mid T_b B$ y $B \longrightarrow T_b B$, respectivamente. Finalmente nuestra gramática G' es

$$P = \begin{cases} S \longrightarrow T_a S \mid T_b B \mid \varepsilon \\ B \longrightarrow T_b B \mid b \mid \varepsilon \\ T_a \longrightarrow a \\ T_b \longrightarrow b \end{cases}$$

Como último, vamos a definir la relación de transición $\hat{\Delta} \subseteq Q \times \Sigma \times Q$ (también representada como $q \xrightarrow{\alpha} q'$ donde $q, q' \in Q$ y $\alpha \in \Sigma$) inductivamente como sigue

- $\hat{\Delta}(q, \varepsilon) = \{q\}$. ($q \xrightarrow{\varepsilon} q$).
- $\hat{\Delta}(q, a) = \{q' : (q, a, q') \in \Delta\}$. (Si $(q, a, q') \in \Delta$ entonces $q \xrightarrow{a} q'$).

- $\hat{\Delta}(q, wa) = \{q'' : q'' \in \hat{\Delta}(q', a)$ para algún $q' \in \hat{\Delta}(q, w)\}$. (Si $(q, w, q') \in \Delta$ y $(q'a, q'') \in \Delta$ para algún $q' \in Q$, entonces $q \overset{wa}{\to} q''$).

Note que de esta forma asociamos un estado q a una relación dependiendo de las transiciones que vamos tomando, que en una gramática sería las posibles generaciones desde una palabra particular $\alpha \in \Gamma^*$. Entender esta relación es clave para entender por qué los algoritmos funcionan.

3. Algoritmos para el cómputo de pre^*

Dado el siguiente problema de decisión

Problema de Decisión 1.
Instancia: Una palabra $w \in \Gamma^*$ y un lenguaje $L(G)$ generado por una GFC G.
Decisión: ¿$w \in pre^*(L(G))$?

Los algoritmos propuestos producen un autómata A_{pre^*} que tiene la capacidad de aceptar o no si $w \in \Gamma^*$ es un predecesor de $L(G)$.

Ambos algoritmos van saturando el autómata inicial de forma iterativa hasta que se añadan todas las posibles transiciones de palabras $w \in V$. La principal diferencia radica en la forma de hacerlo, transformando primero la gramática G en una formal de Chomsky extendida y utilizando la forma de sus producciones para ahorrar llamadas.

3.1 Algoritmo de tiempo $O(lp^2s^6)$

Este primer algoritmo fue propuesto por Book & Otto [9] en base a los sistemas de reescritura de cadenas, y logra dar una primera idea del razonamiento para la construcción del autómata A_{pre^*} adaptándolo a gramáticas.

En particular, $l = \text{máx}_{\beta \in P} |\beta|$ donde $A \longrightarrow \beta$, $p = |P|$ y $s = |Q|$.

Algoritmo 1
Entrada: Una GFC $G = (V, \Sigma, S, P)$ y un autómata $A = (\Gamma, Q, q_0, F, \Delta)$
Salida: El autómata $A_{pre^*} = (\Gamma, Q, q_0, F, \Delta_{pre^*})$

```
1: Δ_pre* ← Δ
2: while Δ_pre* no cambie más do
3:     for q, q' ∈ Q, A ⟶ β ∈ P do
4:         if q' ∈ Δ̂_pre*(q, β) then
5:             Δ_pre* ← Δ_pre* ∪ {(q, A, q')}
6:         end if
7:     end for
8: end while
9: return A = (Γ, Q, q_0, F, Δ_pre*)
```

Para explicar el algoritmo utilicemos como ejemplo el autómata de la sección anterior: $A = (\Sigma, Q, q_0, F, \Delta)$, primero vamos a aumentar el tamaño del alfabeto de lectura con $\Gamma = \Sigma \cup V$, notemos que esto no supone alguna alteración al comportamiento del autómata original ya que

$\Sigma \cap V = \emptyset$ como se había definido en la definición 2.1, entonces $\Gamma = \{a, b, S, B\}$.
Ahora creamos nuevas funciones de transición $\Delta_{pre^*} = \Delta$ y empezamos a ver todas las transiciones actuales y las producciones, si dada una producción $A \longrightarrow \beta$, se tiene que $q' \in \Delta_{pre^*}(q, \beta)$, entonces agregamos la transición $\Delta_{pre^*}(q, A) = q'$ al nuevo autómata. En este caso notemos que $S \longrightarrow bB$ pero no existe transición $\Delta_{pre^*}(q, bB) = q'$, entonces seguimos buscando. La primera producción asociada a una transición es $B \longrightarrow b$ con $\Delta_{pre^*}(q_0, b) = q_1$ y $\Delta_{pre^*}(q_1, b) = q_1$, entonces añadimos las dos nuevas transiciones $\Delta_{pre^*}(q_0, B) = q_1$ y $\Delta_{pre^*}(q_1, B) = q_1$ y finaliza la primera iteración del bucle while. Ahora volvemos a buscar y veamos que para la producción $S \longrightarrow bB$ se tiene que $q_1 \in \hat{\Delta}(q_1, b)$ y $q_1 \in \hat{\Delta}(q_0, B)$, entonces $q_1 \in \hat{\Delta}(q_0, bB)$ y añadimos la nueva transición $\Delta_{pre^*}(q_0, S) = q_1$, similarmente añadimos $\Delta_{pre^*}(q_1, S) = q_1$. Continuando tenemos la producción $B \longrightarrow bB$, de donde tenemos que $q_1 \in \hat{\Delta}(q_1, B)$ y $q_1 \in \hat{\Delta}(q_1, b)$ entonces $q_1 \in \hat{\Delta}(q_1, bA)$ y se añade la transición $\Delta_{pre^*}(q_1, A) = q_1$ que ya estaba en como transición. Finalmente nos queda la producción $S \longrightarrow aS$, de donde sabemos que $q_1 \in \hat{\Delta}(q_0, S)$ y $q_0 \in \hat{\Delta}(q_0, a)$, luego $q_1 \in \hat{\Delta}(q_0, aS)$ y se añade la transición $\Delta_{pre^*}(q_0, S) = q_1$ que también ya estaba. Finalmente como no hay ninguna otra forma de cambiar Δ_{pre^*} entonces finaliza el bucle y retorna $A_{pre^*} = (\Gamma, Q, q_0, F, \Delta_{pre^*})$.

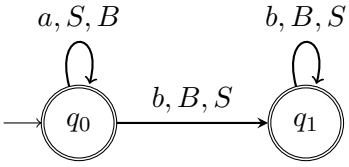


Figura 1: Autómata $A_{pre^*} = (\Gamma, Q, q_0, F, \Delta_{pre^*})$

Tomando como ejemplo $w = aabb \in L(G)$ y sabiendo que, por ejemplo $aabS \in pre^*(w)$, vemos que $aabS$ es aceptado por A_{pre^*}.

Teorema 3.1. *El algoritmo 1 tiene una complejidad temporal del orden de* $O(lp^2s^6)$

Demostración. Revisando línea por línea tenemos:

- *Línea 1.*: copiamos las transiciones a un nuevo conjunto de transiciones, luego tenemos una complejidad $O(ps^2)$, que es el peor caso cuando las transiciones toman todas las producciones y todos los estados.
- *Línea 2. - 8.*: tenemos dos bucles anidados. Analizando línea a línea tenemos:
  - *Línea 2.*: observemos que por iteración, por las líneas siguientes, al menos añadimos una sola transición y a lo mucho el mismo número de búsqueda de la línea 3., que sería cuando cada uno de los estados y producciones cumplen la condición de la línea 4. y saturamos por completo el autómata. Su complejidad es

$$\sum_{q \in Q}^{s} \sum_{q' \in Q}^{s} \sum_{f \in P}^{p} 1 = O(s^2 p)$$

  - *Línea 3.*: para cada par de estados, recorremos todas las producciones de G, entonces

su complejidad está dada por el bucle del while anterior

$$\sum_{q\in Q}^{s}\sum_{q'\in Q}^{s}\sum_{f\in P}^{p}\sum_{q\in Q}^{s}\sum_{q'\in Q}^{s}\sum_{f\in P}^{p} 1 = O(s^4p^2)$$

- *Líneas 4. - 5.*: revisamos si q' pertenece a $\hat{\Delta}_{pre^*}(q,\beta)$, si es así entonces añadimos a la lista de estados un nuevo estado, la complejidad de este par de líneas es de $O(ls^2)$, que es el tiempo en que se verifica si $q' \in \hat{\Delta}(q,\beta)$, y en el bucle tenemos

$$\begin{aligned}
\sum_{q\in Q}^{s}\sum_{q'\in Q}^{s}\sum_{f\in P}^{p}\sum_{q\in Q}^{s}\sum_{q'\in Q}^{s}\sum_{f\in P}^{p} O(ls^2) &= \sum_{q\in Q}^{s}\sum_{q'\in Q}^{s}\sum_{f\in P}^{p}\sum_{q\in Q}^{s}\sum_{q'\in Q}^{s}\sum_{f\in P}^{p} cls^2 \qquad c\in\mathbb{R}\\
&= \sum_{q\in Q}^{s}\sum_{q'\in Q}^{s}\sum_{f\in P}^{p}\sum_{q\in Q}^{s} cls^3p\\
&= \sum_{q\in Q}^{s}\sum_{q'\in Q}^{s} cls^4p^2\\
&= cls^6p^2\\
&= O(ls^6p^2)
\end{aligned}$$

Sumando cada línea tenemos:

$$O(ps^2) + O(s^2p) + O(s^4p^2) + O(ls^6p^2) = O(lp^2s^6)$$

□

Algo que hay que notar antes mostrar el siguiente resultado, es que en cada iteración del bucle while, estamos añadiendo transiciones de mayor profundidad de producciones, es decir, como se ha visto en el ejemplo anterior, en la primera iteración agregamos todas las producciones de longitud 1 ($A \longrightarrow a$), en la segunda todas las producciones de longitud 2 ($A \longrightarrow aB$) y allí acaba el algoritmo porque no existen producciones de mayor longitud.

Teorema 3.2. *(Correctitud del algoritmo):* $L(A_{pre^*}) = pre^*(L(G))$

En [2][1][3] presentan la prueba a través de dos lemas: probando que $L(A_{pre^*}) \subseteq pre^*(L(G))$ y $pre^*(L(G)) \subseteq L(A_{pre^*})$. Sin embargo, probaremos la correctitud utilizando el método de la invariante de ciclo.

Demostración. En la iteración i-ésima tenemos que $A_{pre^*} = (\Sigma, Q, q_0, F, \Delta_{pre^*})$ acepta cadenas $\gamma \in pre^*$ de longitud i, es decir, acepta $\bigcup_{0\le i\le l} pre^i(L(G))$, donde l es la longitud máxima de la parte derecha de las producciones de G.

1. *Inicialización:* En la iteración $i = 0$, es decir afuera del bucle de la línea 2., tenemos que $\Delta_{pre^*} = \Delta$, es decir, A_{pre^*} acepta $pre^0(L(G)) = L(G)$.
2. *Mantenimiento:* Supongamos que en la iteración $i = k$ se tiene que A_{pre^*} acepta $\bigcup_{0\le j\le k-1} pre^j(L(G))$, en particular, observe también que acepta $pre^{k-1}(L(G))$, por esto, todas las palabras $\gamma \in \Gamma^*$ donde $|\gamma| \le k-1$ que cumplen la condición de la línea 3. ya fueron ingresadas en Δ_{pre^*},

entonces ahora es posible verificar si existen $\beta \in \Gamma^*$ tales que $|\beta| = k$. Si existen, entonces se añaden a Δ_{pre^*} y ahora A_{pre^*} acepta $\bigcup_{0 \leq j \leq k} pre^j(L(G))$ y la invariante de ciclo de tiene.

3. *Terminación:* En la iteración $(l+1)$-ésima se tiene que A_{pre^*} acepta $\bigcup_{0 \leq j \leq l} pre^j(L(G))$, y como l es la longitud máxima de la parte derecha de las producciones, entonces ya no hay palabras $\gamma \in \Gamma^*$ que se añadan a Δ_{pre^*} y $\bigcup_{0 \leq j \leq l} pre^j(L(G)) = \bigcup_{0 \leq j} pre^j(L(G)) = pre^*(L(G))$. Y el bucle se acaba porque no se agregaron más estados.

Entonces $L(A_{pre^*}) = pre^*(L(G))$. □

3.2 *Algoritmo de tiempo $O(ps^3)$*

Este segundo algoritmo fue propuesto por Esparza et al. [3] aprovechando resultados observados anteriormente; lo primero es que por iteración i en el algoritmo 1 se revisan todas las producciones

Algoritmo 2

Entrada: Una GFC $G = (V, \Sigma, S, P)$ en la forma extendida de Chomsky y un autómata $A = (\Gamma, Q, q_0, F, \Delta)$
Salida: El autómata $A_{pre^*} = (\Gamma, Q, q_0, F, \Delta_{pre^*})$

```
1: Δ_pre* ← ∅; η ← Δ;
2: for q, q' ∈ Q, A → a ∈ P do
3:     η ← η ∪ {(q, A, q')}
4: end for
5: for q ∈ Q, A → ε ∈ P do
6:     η ← η ∪ {(q, A, q)}
7: end for
8: while η ≠ ∅ do
9:     remove t = (q, B, q') from η
10:    if t ∉ Δ_pre* then
11:        Δ_pre* ← Δ_pre* ∪ {t}
12:        for A → B ∈ P do
13:            η ← η ∪ {(q, A, q')}
14:        end for
15:        for A → BC ∈ P do
16:            for q'' such that (q', C, q'') ∈ Δ_pre* do
17:                η ← η ∪ {(q, A, q'')}
18:            end for
19:        end for
20:        for A → CB ∈ P do
21:            for q'' such that (q'', C, q) ∈ Δ_pre* do
22:                η ← η ∪ {(q'', A, q')}
23:            end for
24:        end for
25:    end if
26: end while
27: return A = (Γ, Q, q_0, F, Δ_pre*)
```

de longitud i, las cuales sabemos que, por lo menos, siempre van haber producciones de longitud 1 (producciones terminales). Y lo segundo es que podemos convertir una gramática en una equivalente en la forma normal de Chomsky con producciones unitarias y ε-producciones en tiempo polinomial (teorema 2.6), que dejan producciones de máximo longitud 2 en la parte derecha.

Similar al anterior algoritmo, toma como entrada una gramática G de la forma extendida de Chomsky y un autómata A que acepta $L(G)$.

Para explicar el algoritmo utilicemos como ejemplo el autómata de la sección anterior y la gramática transformada en la forma normal de Chomsky con ε-producciones. Primero creamos dos nuevas funciones de transiciones $\Delta_{pre^*} = \emptyset$ y $\eta = \Delta$. En los dos primeros for (línea 2. y línea 5.) revisamos todas las producciones terminales y añadimos a η transiciones generadas por las variables de dichas producciones, en nuestro caso tenemos que $B \longrightarrow b$, $T_a \longrightarrow a$ y $T_b \longrightarrow b$, por lo que agregamos $q_1 = \Delta(q_0, B), q_0 = \Delta(q_0, T_a)$ y $q_1 = \Delta(q_0, T_b)$ y $q_1 = \Delta(q_1, T_b)$ respectivamente. De igual forma con $S \longrightarrow \varepsilon$, $B \longrightarrow \varepsilon$ en el segundo bucle. Entonces antes del bucle principal tenemos una cola de la forma

$$\eta = \begin{cases} (q_0, a, q_0), (q_0, b, q_1), (q_1, b, q_1), (q_0, B, q_1) \\ (q_0, T_a, q_0), (q_0, T_b, q_1), (q_1, T_b, q_1), (q_0, S, q_0) \\ (q_1, S, q_1), (q_0, B, q_0), (q_1, B, q_1) \end{cases}$$

En el while (línea 8.) empezamos a vaciar todas las transiciones de η para añadirlas en Δ_{pre^*}: se desencola una transición t de η y se revisa si aún no está en Δ_{pre^*}. Como η funciona como cola, siempre vamos a sacar de primero transiciones con variables terminales: desencolamos $q_0 = \Delta(q_0, a)$, $q_1 = \Delta(q_0, b)$ y $q_1 = \Delta(q_1, b)$ y las añadimos en el if (línea 10.) y como tenemos variables terminales, no se realiza nada en el resto de ciclos. Después empezamos a desencolar las transiciones añadidas con variables no terminales: sacamos $q_1 = \Delta(q_0, B)$ y se añade a Δ_{pre^*}, como en este caso no tenemos variables unitarias no se hace el bucle de la línea 12. y procedemos a analizar las producciones de la forma $A \longrightarrow BC$, que tampoco existen; ahora vemos producciones de la forma $A \longrightarrow CB$, de donde tenemos $S \longrightarrow T_bB$ y $B \longrightarrow T_bB$, pero como aún no hemos ingresado transiciones de la forma $q' = \Delta(q, T_b)$ a Δ_{pre^*} acabamos con esta transición.

Continuamos desencolando de forma similar hasta llegar a $q_0 = \Delta(q_0, S)$, de donde existe la producción $S \longrightarrow T_aS$, de donde sabemos que $(q_0, T_a, q_0) \in \Delta_{pre^*}$ hace algunas iteraciones atrás, entonces añadimos a la cola $\{(q_0, S, q_0)\}$ y terminamos este bucle.
Realizamos lo mismo hasta que la cola η quede vacía y todas las transiciones en Δ_{pre^*}. Así terminamos el proceso y el autómata resultante es el mismo al de la figura 1.

La mejora de este algoritmo depende de varias suposiciones sobre el mismo, por ejemplo, el tomar una cola η y suponer que Δ_{pre^*} se implementa mediante un arreglo de bits, proporciona una complejidad espacial del orden de $O(ps^2)$ [3].

Teorema 3.3. *La complejidad temporal del algoritmo 2 es del orden $O(ps^3)$.*

Demostración. Revisando línea por línea tenemos:

- *Línea 1.*: Copiamos todas las transiciones Δ en una cola η. A lo mucho se puede tardar un tiempo $O(ps^2)$, que corresponde a que las transiciones usen todas las producciones y todos

los estados.

- *Línea 2.-3.*: Buscamos estados y producciones con variables terminales y las añadimos a la cola. Como añadir un elemento a una cola toma tiempo constante, entonces la complejidad total es
$$\sum_{i=1}^{s}\sum_{j=i}^{s}\sum_{k=1}^{p} O(1) = O(s^2p)$$

- *Línea 5.-6.*: Similarmente a las líneas anteriores, buscamos ahora solo un estado y ε-producciones y añadimos a la cola. Tenemos
$$\sum_{i=1}^{s}\sum_{j=1}^{p} O(1) = O(sp)$$

- *Línea 8.* Para el bucle principal, similarmente al algoritmo 1, el número de iteraciones depende de la iteraciones después del if. En particular la línea 9. se ejecuta $O(s^2p)$ veces.
  - *Línea 12.*: Se busca para cada producción los estados q, q'' de la transición desencolada en la línea 9. cuyo orden $O(s^2p)$ está dado por
$$\sum_{i=1}^{s}\sum_{j=1}^{s}\sum_{k=1}^{p} O(1) = O(s^2p)$$
  - *Línea 15. y Línea 20.*: En ambos bucles buscamos, además de las producciones de longitud dos, los dos estados q, q' y uno tercero q''. Tenemos así
$$\sum_{i=1}^{s}\sum_{j=1}^{s}\sum_{k=1}^{s}\sum_{l=1}^{p} O(1) = O(s^3p)$$

Sumando cada línea tenemos

$$O(ps^2) + O(sp) + O(s^2p) + 2 \cdot O(s^3p) = O(ps^3)$$

□

Ahora bien, observemos que como la propagación y la saturación del conjunto de estados Δ_{pre^*} se hace es por estado desencolado de η, la prueba de correctitud será diferente a la prueba realizada en el algoritmo anterior.

Teorema 3.4. *(Correctitud del algoritmo)* $L(A_{pre^*}) = pre^*(L(G))$

Demostración. Al inicio de la i-ésima iteración, el conjunto de transiciones Δ_{pre^*} contiene a lo mucho i transiciones resultantes de ir desencolando η.

1. *Inicialización:* En la iteración $i = 0$, es decir, afuera del bucle de la línea 8. tenemos que $\Delta_{pre^*} = \emptyset$ y $\eta = \Delta \cup \{(q, A, q') : q, q' \in Q \wedge A \in P\} \cup \{(q, A, q) : q \in Q \wedge A \in P)\}$.

2. *Mantenimiento:* Supongamos que en el comienzo de la iteración $i = k$ se tiene que $|\Delta_{pre^*}| = k-1$ y $|\eta| = m$ para algún $m \in \mathbb{N}^+$. Entonces se desencola la transición siguiente t en η y $|\eta| = m-1$. Si $t \notin \Delta_{pre^*}$ entonces se añade, por lo que $|\Delta_{pre^*}| = k$, y además se exploran los tres casos para encolar en η. Note que esto no afecta el rumbo de la prueba porque η siempre se va a desencolar y no siempre se va a entrar al condicional de la línea 10..
   Si $t \in \Delta_{pre^*}$ entonces no se añade y la invariante de ciclo se mantiene porque $|\Delta_{pre^*}| = k-1 \leq k$.

3. *Terminación:* Al comienzo de la última iteración, supongamos que es la iteración r-ésima, entonces $|\Delta_{pre^*}| = r < ps^2$ y $|\eta| = 1$, entonces se desencola t de η, por lo que $\eta = \emptyset$. Se revisa si $t \in \Delta_{pre^*}$, observe que la única opción que tenemos en una última iteración es que t ya esté en Δ_{pre^*} (de cualquier otra forma podríamos estar encolando más estados contradiciendo que sea la última iteración) por lo que $|\Delta_{pre^*}| = r < ps^2$.

De esta forma se puede concluir que $L(A_{pre^*}) = pre^*(L(G))$. □

Note que un paso importante del algoritmo es verificar si la transición t desencolada ya está en Δ_{pre^*}, de cualquier otra forma el algoritmo podría nunca terminar.

4. Aplicaciones

La principal aplicación de solucionar el problema de decisión 1 tiene que ver con la solución de otros problema relacionados con gramáticas independientes del contexto. Algunas de los problemas que se pueden identificar [3][2].

1. *Pertenencia:* Se presenta el problema de decisión como sigue

   Problema de Decisión 2.
   Instancia: Una palabra $w \in \Sigma^*$ y un lenguaje $L(G)$ generado por una GFC G.
   Decisión: ¿$w \in L(G)$?

   Se puede solucionar el problema verificando si dada la variable inicial de G se puede encontrar una generación a w, es decir, verificando si $S \in pre^*(\{w\})$. Por Early [11] se tiene un algoritmo en tiempo cuadrático, por lo que no es mucho más lento la construcción del autómata A_{pre^*} y su ejecución.

2. *Vacuidad*: Se presenta el problema de decisión como sigue

   Problema de Decisión 3.
   Instancia: Dado lenguaje $L(G)$ generado por una GFC G.
   Decisión: ¿$L(G) = \emptyset$?

   Verificar esto es tan simple como ver si $S \notin pre^*(\Sigma^*)$.

3. *Identificación de variables inútiles*: Decimos que una variable $A \in V$ es útil si existe una generación $S \Rightarrow^* w_1 A w_2 \Rightarrow^* w$, en donde $w_1, w_2 \in \Sigma^*$. Si no es así entonces se dice que

A es inútil, esto es que las producciones de A no están en ninguna generación de las palabras generadas por la gramática. En ese sentido, es equivalente pensar en que en alguna secuencia de producciones llegamos a alguna en donde se involucra A, por lo que para ver si A es útil, basta con ver si $S \in pre^*(\Sigma^* A \Sigma^*)$.

4. *Infinitud*:Dado el problema de decisión

> **Problema de Decisión 4.**
> **Instancia:** Un lenguaje $L(G)$ generado por una GFC G.
> **Decisión:** ¿$|L(G)| = \aleph_0$?

Un lenguaje es infinito si, dado $A \in V$ y $w_1, w_2 \in \Sigma^*$, existe una secuencia $A \Rightarrow^* w_1 A w_2$ en donde $w_1 w_2 \neq \varepsilon$. De esta forma se hace evidente que una solución es verificar si $A \in pre^*(\Sigma^+ A \Sigma^* \cup \Sigma^* A \Sigma^+)$ para toda variable A.

5. Conclusiones

Se puede evidenciar la gran capacidad, por lo menos desde un espacio teórico y pedagógico, de la resolución al problema de decisión de pre^* en su aplicación a otros problemas de decisión sobre gramáticas. Manteniendo una complejidad no muy grande con respecto a los algoritmos que solucionan dichos problemas individualmente.

La mejora en el segundo algoritmo muestra la posibilidad de mejorar el tiempo de ejecución en la solución del problema, realizando un análisis profundo sobre el funcionamiento general del primer algoritmo y de las demás herramientas matemáticas usadas como apoyo, como lo son las GIC en la forma normal de Chomsky.

Las distintas herramientas adquiridas a lo largo del curso dan cuenta de su importancia al poder ser utilizadas para realizar diversos análisis en los algoritmos, como lo son en la identificación del tipo de algoritmo, en las pruebas de complejidad temporal y de correctitud del algoritmo.

Referencias

[1] J. Esparza and P. Rossmanith, *An automata approach to some problems on context-free grammars*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 143–152. [Online]. Available: https://doi.org/10.1007/BFb0052083

[2] A. Bouajjani, J. Esparza, A. Finkel, O. Maler, P. Rossmanith, B. Willems, and P. Wolper, "An efficient automata approach to some problems on context-free grammars," *Information Processing Letters*, vol. 74, no. 5, pp. 221–227, 2000. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020019000000557

[3] J. Esparza, P. Rossmanith, and S. Schwoon, "A uniform framework for problems on context-free grammars," 2024. [Online]. Available: https://arxiv.org/abs/2410.19386

[4] D.-Z. Du and K.-I. Ko, *Problem solving in automata, languages, and complexity*. John Wiley & Sons, 2004.

[5] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to automata theory, languages, and computation*. ACM New York, NY, USA, 2001, vol. 32, no. 1.

[6] R. de Castro Korgi, *Introducción a la Teoría de la Comptuación, Lenguajes, autómatas, gramáticas*. Universidad Nacional de Colombia, 2004.

[7] M. K. Brown and B. Buntschuh, "A context-free grammar compiler for speech understanding systems." in *ICSLP*, vol. 94, 1994, pp. 21–24.

[8] A. Finkel and E. Gupta, "The well structured problem for presburger counter machines," 2023. [Online]. Available: https://arxiv.org/abs/1910.02736

[9] R. V. Book and F. Otto, *String-Rewriting Systems*. New York, NY: Springer New York, 1993, pp. 35–64. [Online]. Available: https://doi.org/10.1007/978-1-4613-9771-7_3

[10] P. Rossmanith, *Computing pre* for General Context Free Grammars*. Cham: Springer Nature Switzerland, 2024, pp. 255–280. [Online]. Available: https://doi.org/10.1007/978-3-031-56222-8_15

[11] J. Earley, "An efficient context-free parsing algorithm," *Commun. ACM*, vol. 13, no. 2, p. 94–102, Feb. 1970. [Online]. Available: https://doi.org/10.1145/362007.362035