Universidad de La Habana
Facultad de Matemática y Computación

Título de la tesis

Autor:
Raudel Alejandro Gómez Molina

Tutores:
Fernando Rodríguez Flores

Trabajo de Diploma
presentado en opción al título de
Licenciado en Ciencia de la Computación

Fecha

https://github.com/raudel25/my-thesis

Dedicación

Agradecimientos

Agradecimientos

Opinión del tutor

Opiniones de los tutores

Resumen

Resumen en español

Abstract

Resumen en inglés

Índice general

Índice de figuras

Ejemplos de código

Introducción

Capítulo 1

Preliminares

1.1. Teoría de Lenguajes

En esta sección se definen los principales conceptos de teoría de lenguajes que sirven de base al contenido de las secciones posteriores.

1.1.1. Conceptos básicos

Alfabeto: Un alfabeto, denotado como Σ, es un conjunto finito y no vacío de símbolos.

Por ejemplo, el alfabeto $\Sigma = \{1, 0\}$ está formado por los símbolos 0 y 1.

Cadena: Una cadena es una sucesión finita de símbolos del alfabeto.

Por ejemplo: 11 y 101 pueden ser cadenas sobre el alfabeto Σ.

Lenguaje: Un lenguaje es un conjunto de cadenas definido sobre un alfabeto.

Por ejemplo: el lenguaje de la representación binaria de todos los números pares $L = \{w \,|\, \text{last}(w) = 0\}$, $\text{last}(w)$ representa el último caracter de la cadena w.

Dados los conceptos básicos, ahora es necesario definir las operaciones básicas entre los elementos de la teoría de lenguajes.

1.1.2. Operaciones con Lenguajes

Como los lenguajes son conjuntos, todas las operaciones sobre conjuntos también se definen para lenguajes.

Unión: La unión de dos lenguajes L_1 y L_2 se define como el conjunto de cadenas que pertenecen a L_1 o a L_2:

$$L_1 \cup L_2 = \{w \,|\, w \in L_1 \vee w \in L_2\}.$$

Intersección: La intersección de dos lenguajes L_1 y L_2 se define como el conjunto de cadenas que pertenecen a L_1 y a L_2:

$$L_1 \cap L_2 = \{w \,|\, w \in L_1 \wedge w \in L_2\}.$$

Concatenación: La concatenación de dos lenguajes L_1 y L_2 se define como el conjunto de cadenas que resultan de concatenar una cadena de L_1 con una cadena de L_2:

$$L_1 \circ L_2 = \{w_1 w_2 \,|\, w_1 \in L_1 \wedge w_2 \in L_2\}.$$

Complemento: El complemento de un lenguaje L se define como el conjunto de cadenas que no pertenecen a L:

$$\overline{L} = \{w \,|\, w \notin L\}.$$

Clausura de Kleene: La clausura de Kleene de un lenguaje L se define como el conjunto de cadenas que resultan de concatenar cero o más cadenas de L:

$$L^* = \{w_1 w_2 \dots w_n \,|\, n \geq 0 \,\text{y}\, w_i \in L\}$$

Homorfismo: Dados un alfabeto Σ y un alfabeto Γ, un homomorfismo es una función:

$$h : \Sigma^* \to \Gamma^*$$

tal que:

1. Para cada $a \in \Sigma$, $h(a)$ es una cadena en Γ^*.
2. Para cualquier par de cadenas $u, v \in \Sigma^*$, se cumple que:

   $$h(uv) = h(u)h(v),$$

   es decir, el homomorfismo preserva la concatenación.

Por ejemplo sobre el alfabeto $\Sigma = \{a, b\}$, se define el homomorfismo h, tal que $h(a) = 0$ y $h(b) = 1$, entonces $h(ab) = 01$.

Todos los conceptos anteriormente planteados definen la base de la teoría de lenguajes y de estos se derivan algunas preguntas como determinar si una cadena pertenece a un lenguaje o determinar si un lenguaje es vacío. A continuación se presentan 3 problemas que serán usados en el desarrollo de este trabajo.

1.1.3. Problemas relacionados con Lenguajes

Los problemas relacionados con la teoría de lenguajes permiten vincular los problemas de otra naturaleza con su representación en la teoría de lenguajes formales.

Problema de la palabra: Consiste en determinar si una cadena pertenece a un lenguaje dado.

Todo problema en Ciencia de la Computación puede ser reducido a un problema de la palabra, ya que cualquier problema puede ser codificado como un lenguaje formal [10].

Problema del vacío: Consiste en determinar si un lenguaje es vacío.

Problema de la equivalencia: Consiste en determinar si dos L_1 y L_2 lenguajes son iguales (es decir si se cumple que $L_1 \subseteq L_2 \wedge L_2 \subseteq L_1$).

En la próxima sección se presentan las gramáticas, un mecanismo que permite definir un lenguaje.

1.1.4. Gramáticas

Una **gramática** es un formalismo utilizado para describir lenguajes formales. Se define como una 4-tupla:

$$G = (N, \Sigma, P, S),$$

donde:

- N: Es un conjunto finito de **símbolos no terminales**, que representan variables o categorías intermedias.
- Σ: Es un conjunto finito de **símbolos terminales**, que constituyen el alfabeto del lenguaje. Se cumple que $N \cap \Sigma = \emptyset$.
- P: Es un conjunto finito de **reglas de producción**, cada una de la forma:

$$\alpha \to \beta, \quad \text{donde } \alpha \in (N \cup \Sigma)^* \wedge \beta \in (N \cup \Sigma)^*.$$

- S: Es el **símbolo inicial**, $S \in N$, que define el punto de partida para derivar cadenas del lenguaje.

Una derivación en la gramática consiste en seleccionar una **regla de producción** $\alpha \to \beta$ y sustituir una ocurrencia de α en una cadena w por β.

Una cadena w, $w \in \Sigma^*$, se puede generar por la gramática G si existe una secuencia de derivaciones que comienza con S y termina con la cadena w.

El lenguaje generado por una gramática G se denota como:

$$L(G) = \{w \in \Sigma^* \mid S \overset{*}{\to} w\},$$

donde $\overset{*}{\to}$ indica una derivación en cero o más pasos.

A continuación se presenta la jerarquía de Chomsky, que clasifica a los lenguajes formales de acuerdo con su poder de generación.

1.1.5. Jerarquía de Chomsky

La **Jerarquía de Chomsky** (Figura 1.1) clasifica las gramáticas en cuatro tipos, según las restricciones en sus reglas de producción y la capacidad expresiva de los lenguajes que generan [8].

1. **Tipo 0: Gramáticas irrestrictas**
   - No tienen restricciones en las reglas de producción.
   - Cada regla tiene la forma: $\alpha \to \beta$, donde $\alpha, \beta \in (N \cup \Sigma)^*$ y $\alpha \neq \varepsilon$.
   - Todo lenguaje generado por una gramática irrestricta se denomina **lenguaje recursivamente enumerable**.
2. **Tipo 1: Gramáticas dependientes del contexto**
   - Cada regla tiene la forma: $\alpha A \gamma \to \alpha \beta \gamma$, donde $A \in N$, $\alpha, \beta, \gamma \in (N \cup \Sigma)^*$, y $|\beta| \geq 1$.
   - Todo lenguaje generado por una gramática dependiente del contexto se denomina **lenguaje dependiente del contexto**.
   - Todo lenguaje dependiente del contexto es también un lenguaje recursivamente enumerable.
3. **Tipo 2: Gramáticas libres del contexto**
   - Cada regla tiene la forma: $A \to \beta$, donde $A \in N$ y $\beta \in (N \cup \Sigma)^*$.
   - Todo lenguaje generado por una gramática libre del contexto se denomina **lenguaje libre del contexto**.
   - Todo lenguaje libre del contexto es también un lenguaje dependiente del contexto.
4. **Tipo 3: Gramáticas regulares**

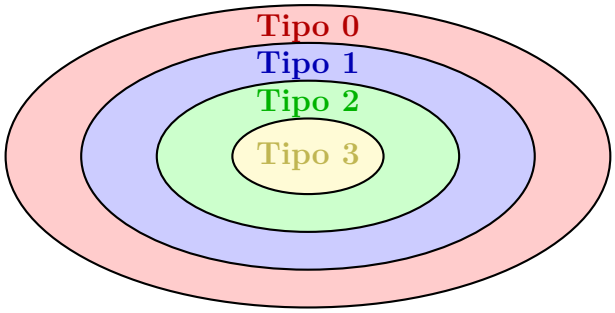

Figura 1.1: Esquema de la Jerarquía de Chomsky

- Las reglas tienen la forma:

$$A \to aB \quad \text{o} \quad A \to a,$$

  donde $A, B \in N$ y $a \in \Sigma$.
- Todo lenguaje generado por una gramática regular se denomina **lenguaje regular**.
- Todo lenguaje regular es un lenguaje libre del contexto.

Las diferencias entre los elementos de la Jerarquía de Chomsky se pueden ilustrar con el lenguaje *Copy* sobre un alfabeto Σ, que se define como $L_{copy} = \{w^+ \,|\, w \in Z^*\}$. Si se toma un caso particular de L_{copy}, al cual se le llama $L^n_{copy} = \{w^n \,|\, w \in Z^*\}$, L^1_{copy} es un lenguaje regular, mientras que L^2_{copy} es un lenguaje libre del contexto y por último $L^k_{copy} \forall\, k \geq 3$ es un lenguaje dependiente del contexto [10]. El lenguaje L_{copy} se usa en los restantes capítulos como base de formalismos que describen el tema analizado en este trabajo.

En la próxima sección se presentan los principales conceptos relacionados con las gramáticas libres del contexto y las gramáticas regulares. Estos contenidos sirven de base para el resto de los capítulos y secciones.

1.2. Autómatas

Un autómata es un modelo de cómputo en la teoría de lenguajes. En esta sección se abordan los principales conceptos sobre los autómatas.

1.2.1. Autómata regular

Un autómata regular [10], también conocido como autómata finito, es un modelo matemático que permite reconocer lenguajes regulares. Este tipo de autómata se

define como una máquina abstracta que procesa cadenas de símbolos de un alfabeto finito y determina si una cadena pertenece a un lenguaje regular.

Un autómata regular se define como una 5-tupla

$$\mathcal{A} = (Q, \Sigma, \delta, q_0, F),$$

donde:

- Q: Es un conjunto finito de **estados**.
- Σ: Es el **alfabeto** finito de entrada.
- δ: Es la **función de transición**, $\delta : Q \times \Sigma \to Q$, que define cómo el autómata cambia de estado en función del símbolo leído.
- $q_0 \in Q$: Es el **estado inicial** desde donde comienza la computación.
- $F \subseteq Q$: Es el conjunto de **estados de aceptación o estados finales**.

El autómata comienza en el estado inicial q_0 y procesa una cadena de entrada símbolo por símbolo. En cada paso, la función de transición δ determina el siguiente estado del autómata. Si, después de procesar toda la cadena, el autómata termina en un estado de aceptación $q \in F$, entonces la cadena se acepta; de lo contrario, se rechaza.

Se puede extender el concepto de autómata finito añadiendo un nuevo tipo de transición que no consume ningún caracter, la cual recibe el nombre de transición ε. Se puede demostrar [10] que el conjunto de lenguajes reconocido por un autómata finito sin transiciones ε (*autómata finito determinista*) es equivalente al conjunto de lenguajes reconocido por un autómata finito con transiciones ε (*autómata finito no determinista*).

A continuación se presenta el concepto de autómata que reconoce a los lenguajes libres del contexto.

1.2.2. Autómata de pila y Gramáticas libres del contexto

Un autómata de pila [10] es un modelo matemático de computación que extiende de los autómatas finitos al incluir una estructura de datos adicional: una pila. Este modelo es capaz de reconocer lenguajes libres de contexto, proporcionando una conexión directa con las gramáticas libres de contexto *CFG*. Es decir dada una gramática libre del contexto se puede construir un autómata de pila que reconozca el lenguaje generado por la gramática y viceversa [10].

Un autómata de pila se define como una 7-tupla

$$\mathcal{A} = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F),$$

donde:

- Q: Es un conjunto finito de **estados**.
- Σ: Es el **alfabeto** finito de entrada.
- Γ: Es el **alfabeto** finito de la pila (conjunto de símbolos que se pueden almacenar en la pila).
- δ: Es la función de transición, $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma^*)$, que describe cómo cambia el estado, el contenido de la pila y la posición en la entrada.
- $q_0 \in Q$: Es el **estado inicial** desde donde comienza la computación.
- $Z_0 \in \Gamma$: Es el símbolo inicial en la pila.
- $F \subseteq Q$: Es el conjunto de **estados de aceptación o estados finales**.

Un autómata de pila procesa una cadena de entrada desde el estado inicial q_0 y puede utilizar transiciones ε (sin consumir entrada). En cada paso, la función δ determina el nuevo estado, los símbolos que se insertan o sacan de la pila, y el avance en la entrada. Tras procesar toda la cadena, el autómata termina en un estado de aceptación o con una pila vacía (dependiendo del criterio de aceptación), la cadena se acepta.

A continuación se presenta una combinación de un autómata finito con un homomorfismo.

1.2.3. Transductor finito

Un transductor finito [9] es un modelo computacional que extiende los autómatas finitos al incluir tanto entradas como salidas. Formalmente, un transductor finito es un autómata finito con una función de transición extendida que asocia una salida a cada transición.

Un transductor finito puede representarse como una 6-tupla:

$$T = (Q, \Sigma, \Gamma, \delta, q_0, F),$$

donde:

- Q es el conjunto finito de estados.

- Σ es el alfabeto de entrada.
- Γ es el alfabeto de salida.
- $\delta : Q \times \Sigma \to Q \times \Gamma^*$ es la función de transición, que mapea una combinación de estado actual y símbolo de entrada a un nuevo estado y una salida.
- $q_0 \in Q$ es el estado inicial.
- $F \subseteq Q$ es el conjunto de estados finales.

Observe que un homorfismo es un transductor finito de un solo estado y tantas transiciones hacia el mismo estado como transformaciones de símbolos en el homomorfismo.

Por ejemplo se define el siguiente transductor finito T:

$$T = (\{q_0, q_1\}, \Sigma, \Gamma, \delta, q_0, Q),$$

donde:

- $Q = \{q_0, q_1, q_2\}$ es el conjunto de estados.
- $\Sigma = \{a, b\}$ es el alfabeto de entrada.
- $\Gamma = \{0, 1\}$ es el alfabeto de salida.
- δ es la función de transición definida por:

δ	a	b
q_0	$(q_0, 0)$	$(q_1, 1)$
q_1	$(q_2, 0)$	$(q_1, 1)$
q_2	$(q_2, 0)$	$(q_2, 1)$

- q_0 es el estado inicial.
- $F = \{q_1\}$ es el conjunto de estados finales.

Entonces la cadena *aaaabbbbb* es reconocida por T y $T(aaaabbbbb) = 000011111$.

A continuación se describe el modelo de cómputo más general que describe el resto de los elementos de la Jerarquía de Chomsky.

1.3. Máquina de Turing

Una Máquina de Turing [10] es un modelo abstracto de computación universal introducido por Alan Turing en 1936. Este modelo consiste en los siguientes componentes:

- **Cinta**: Un medio de almacenamiento infinito dividido en celdas, donde cada celda contiene un símbolo de un alfabeto finito.
- **Cabezal de lectura/escritura**: Un dispositivo que puede leer el contenido de una celda, escribir un nuevo símbolo y moverse a la izquierda o derecha.
- **Conjunto de estados**: Una colección finita de estados internos que describen la configuración actual de la máquina.
- **Función de transición**: Un conjunto de reglas que determinan cómo la máquina cambia de estado, escribe en la cinta y mueve el cabezal en función del estado actual y el símbolo leído.

Máquina de Turing determinista (*DTM*): En una Máquina de Turing determinista, para cada estado y cada símbolo leído, existe como máximo una transición definida.

Máquina de Turing no determinista (*NTM*): En una Máquina de Turing no determinista, para cada estado y símbolo leído, pueden existir múltiples transiciones posibles.

La Máquina de Turing es fundamental en el estudio de la computabilidad, ya que establece un marco teórico para definir qué problemas son resolubles mediante algoritmos. Este modelo formalizó el concepto de *función computable*, es decir, aquellas funciones que pueden ser calculadas mediante una secuencia finita de pasos definidos por una máquina de Turing [10].

Todas las operaciones con lenguajes y los problemas relacionados con ellos tienen una dificultad y para medir esta dificultad se utiliza un marco teórico llamado complejidad computacional, el cual se presenta en la próxima sección.

1.4. Complejidad computacional

En esta sección se definen los principales conceptos de complejidad computacional y las clases de problemas.

A continuación se presenta una notación para describir el tiempo que demora un algoritmo en realizar determinado cómputo.

1.4.1. Notación asintótica

La notación asintótica se utiliza para describir el comportamiento de una función $f(n)$ a medida que n crece hacia el infinito. A continuación se definen las notaciones más comunes:

- **Notación** $O(f(n))$: Una función $g(n)$ pertenece a $O(f(n))$ si existen constantes positivas c y n_0 tales que:

$$g(n) \leq c \cdot f(n) \quad \text{para todo } n \geq n_0.$$

  Esta notación proporciona un límite superior asintótico para $g(n)$.

- **Notación** $\Omega(f(n))$: Una función $g(n)$ pertenece a $\Omega(f(n))$ si existen constantes positivas c y n_0 tales que:

$$g(n) \geq c \cdot f(n) \quad \text{para todo } n \geq n_0.$$

  Esta notación proporciona un límite inferior asintótico para $g(n)$.

- **Notación** $\Theta(f(n))$: Una función $g(n)$ pertenece a $\Theta(f(n))$ si:

$$g(n) \in O(f(n)) \quad \text{y} \quad g(n) \in \Omega(f(n)).$$

  Es decir, $f(n)$ acota a $g(n)$ tanto superior como inferiormente.

La notación asintótica permite describir el tiempo de ejecución un algoritmo con respecto al número de operaciones básicas realizadas por un modelo formal de cómputo (por ejemplo una máquina de Turing). Algoritmos como determinar el mínimo y el máximo de un arreglo son $\Theta(n)$, ya que necesitan realizar una cantidad n de operaciones básicas en relación con la cantidad de elementos del arreglo.

Se dice que un algoritmo tiene un tiempo polinomial si puede resolverse en una complejidad de $O(n^k)$, donde n es el tamaño de la entrada del algoritmo y k es una constante. Por ejemplo encontrar el mínimo y el máximo de un arreglo tiene un tiempo polinomial.

En la próxima sección se presenta la clasificación de los problemas de acuerdo a su complejidad computacional.

1.4.2. Clases de problemas

Los problemas computacionales [10] se agrupan en diferentes clases según los recursos necesarios para resolverlos.

Problemas en la clase P: Un problema pertenece a la clase P si puede resolverse en tiempo polinomial.

Problemas en la clase NP: Un problema pertenece a NP si su solución puede verificarse en tiempo polinomial mediante una Máquina de Turing determinista. Alternativamente, un problema está en NP si puede resolverse en tiempo polinomial mediante una Máquina de Turing no determinista [10].

Problemas en la clase NP-Completo: Un problema pertenece a la clase NP-Completo, si pertenece a NP y además es tan difícil como cualquier otro problema en NP. Esto significa que cualquier problema en NP puede reducirse a este problema en tiempo polinómico [10].

Problemas en la clase NP-Duro: Un problema pertenece a la clase NP-Duro, es tan difícil como cualquier otro problema en NP, pero no necesariamente pertenece a NP [10].

Problemas no decidibles: Un problema es no decidible si no existe una Máquina de Turing que pueda resolverlo correctamente para todas las entradas posibles. Esto significa que no hay algoritmo que garantice una respuesta en tiempo finito en todos los casos [10]. Un ejemplo clásico de problema no decidible es el *Problema de la Parada* [10], que consiste en determinar si una Máquina de Turing se detendrá para una entrada dada.

En la siguiente sección se realiza la comparación entre las clases P y NP.

1.4.3. P vs NP

La relación entre las clases P y NP es uno de los problemas abiertos más importantes en la teoría de la computación [10]. Hasta la fecha, se desconoce si $P = NP$ o si $P \neq NP$, es decir no se conoce si realmente los problemas en NP son más difíciles que los problemas en P. Por otro lado el conjunto de problemas NP-Completo brinda una base sólida para el problema anterior, ya que dada su definición, cualquier problema perteneciente a este conjunto que sea soluble en tiempo polinomial implica que todos los problemas en NP lo son. Mientras que los problemas en NP-Duro pueden resultar aún más difíciles, es decir aunque resultara que $P = NP$ no se puede asegurar que no existan problemas en NP-Duro que no se puedan resolver en tiempo polinomial [10].

Por otra parte existen problemas para los cuales no existe ningún algoritmo, como los problemas indecidibles.

A continuación se presenta el problema que sirve de base a los problemas de la clase NP-Completo y que a su vez, la relación de este con la teoría de lenguajes, es el foco de atención de este trabajo.

1.5. Problema de la satisfacibilidad booleana

El problema de la satisfacibilidad booleana (SAT), es un problema fundamental en la teoría de la computación y la lógica matemática [10]. El objetivo principal del problema es determinar si existe una asignación de valores a las variables de una expresión booleana tal que la expresión sea verdadera.

En las próximas secciones se definen los principales elementos del SAT.

1.5.1. Variables booleanas

Una variable booleana es una variable que puede tomar uno de dos valores posibles: *true* (verdadero) o *false* (falso). Estas variables se utilizan para construir expresiones lógicas.

1.5.2. Literales

Un literal es una variable booleana o su negación. Formalmente, si x es una variable booleana, entonces x y $\neg x$ (la negación de x) son literales. Un literal puede tomar los valores $true$ o $false$ dependiendo de la asignación de valores a las variables.

1.5.3. Cláusulas

Una cláusula es una disyunción (operador **OR**) de uno o más literales. Por ejemplo, la cláusula $(x \vee \neg y \vee z)$ es una disyunción de tres literales: x, $\neg y$ y z. Una cláusula es verdadera si al menos uno de sus literales es verdadero. Si todos los literales son falsos, la cláusula será falsa.

1.5.4. Fórmulas en forma normal conjuntiva

Una fórmula booleana en forma normal conjuntiva (CNF) es una conjunción (operador **AND**) de cláusulas. En otras palabras, es una expresión booleana que se puede escribir como una serie de cláusulas unidas por el operador **AND**. Por ejemplo:

$$(x \vee \neg y \vee z) \wedge (\neg x \vee y) \wedge (x \vee \neg z)$$

1.5.5. Fórmulas booleanas equivalentes

Dos fórmulas booleanas se consideran equivalentes si, para cualquier asignación de valores a sus variables, ambas producen el mismo resultado lógico. Por ejemplo, las fórmulas $x \vee (y \wedge z)$ y $(x \vee y) \wedge (x \vee z)$ son equivalentes, ya que para cualquier combinación de valores x, y, z, ambas tienen el mismo valor lógico.

Para cualquier fórmula booleana existe una fórmula booleana equivalente en CNF [10] y el algoritmo para encontrarla es polinomial, por lo tanto se puede asumir que toda fórmula booleana está en CNF.

1.5.6. Definición del problema de la satisfacibilidad booleana

El problema de la satisfacibilidad booleana, o SAT, consiste en determinar si existe una asignación de valores $true$ o $false$ a las variables de una fórmula booleana tal que la fórmula completa sea verdadera. En términos formales, dado un conjunto de cláusulas en CNF, el problema es encontrar una asignación de valores a las variables que haga que la conjunción de las cláusulas sea verdadera.

Formalmente, se dice que una fórmula booleana en CNF es satisfacible si existe una asignación de valores a las variables tal que todas las cláusulas de la fórmula sean verdaderas simultáneamente.

- Si existe tal asignación, la fórmula es *satisfacible*.
- Si no existe tal asignación, la fórmula es *insatisfacible*.

Un SAT con exactamente n variables distintas en cada cláusula se denomina n-SAT.

1.5.7. SAT como Problema NP-Completo

El SAT es el primer problema demostrado como NP-Completo [10] y juega un rol central en la teoría de la complejidad computacional. Se define en la clase NP porque, dada una asignación de valores a las variables de la fórmula booleana, se puede verificar en tiempo polinómico si dicha asignación satisface la fórmula.

Además, la prueba de que SAT es NP-Completo fue una de las contribuciones principales de Stephen Cook en 1971 [10], marcando el inicio de la teoría de la NP-completitud.

1.5.8. Equivalencia entre SAT y 3-SAT

Para el problema 2-SAT existe una solución polinomial que determina si la fórmula booleana es satisfacible o no [7], pero para el problema 3-SAT no se conoce ningún

algoritmo polinomial que permita determinar si una fórmula booleana es satisfacible o no [10].

Cualquier fórmula booleana del problema n-SAT se puede reducir a una fórmula booleana equivalente del problema 3-SAT, por lo tanto, SAT es equivalente a 3-SAT en términos de complejidad computacional [10].

1.5.9. Problemas SAT solubles en tiempo polinomial

Como se mencionó anteriormente no se conoce ningún algoritmo polinomial para resolver el problema SAT en general, pero existen casos particulares del problema que sí pueden ser resueltos en tiempo polinomial. A continuación se presentan los principales casos:

- **1-SAT:** El problema 1-SAT es una instancia particular de SAT donde cada cláusula tiene a lo sumo un literal. Este problema puede ser resuelto en tiempo polinomial mediante un algoritmo de asignación de valores de booleanos.
- **2-SAT:** El problema 2-SAT es una instancia de SAT donde cada cláusula contiene exactamente dos literales. Este problema puede ser resuelto en tiempo polinomial mediante una modelación basada en grafos, utilizando algoritmos como la detección de componentes fuertemente conexas en el grafo de implicación.
- **Horn-SAT:** El problema Horn-SAT es una generalización del problema SAT, donde cada cláusula tiene a lo sumo un literal positivo. Este problema puede ser resuelto en tiempo polinomial mediante el algoritmo de resolución de Horn.
- **XOR-SAT:** El problema XOR-SAT es una instancia de SAT donde cada cláusula representa una operación XOR sobre los literales. Puede ser resuelto en tiempo polinomial transformando el problema en un sistema de ecuaciones lineales modulares y aplicando eliminación de Gauss.

En este capítulo se abordó todo el contenido de la teoría de lenguajes y el SAT necesario para el contenido de los capítulos posteriores. En el siguiente capítulo se abordan los formalismos de escritura regulada que se usan en los capítulos 3, 4 y 5, estos formalismos reconocen lenguajes que pertenecen a los lenguajes dependientes del contexto en la Jerarquía de Chomsky.

Capítulo 2

Formalismos de escritura regulada

En los capítulos posteriores se presentan estrategias para la solución del SAT que emplean formalismo de la teoría de lenguajes, dichos formalismos utilizan formalismos de escritura regulada para controlar la asignación de valores para las variables del SAT. La mayoría de estos formalismos describen lenguajes más generales que los lenguajes libres del contexto.

En el presente capítulo se presentan las gramáticas de concatenación de rango simple y las gramáticas matriciales simples que serán usadas en el capítulo 3, las gramáticas de índice global que serán usadas en el capítulo 4 y finalmente las gramáticas de concatenación de rango, las cuales representan el foco de atención de este trabajo, que serán usadas en el capítulo 5.

2.1. Gramáticas Matriciales

Una gramática matricial [11] de grado n n-MG es una 4-tupla:

$$G_n = (V, P, S, \Sigma)$$

donde:

- V es un conjunto finito de **símbolos no terminales**.
- Σ es un conjunto finito de **símbolos terminales**, con $V \cap \Sigma = \emptyset$.
- P es un conjunto finito de matrices. Cada matriz es una secuencia ordenada de **producciones** libres del contexto de la forma:

  $$[P_1, P_2, \ldots, P_k]$$

  donde cada P_i es una regla $A \to \alpha$, con $1 \leq k \leq n$, $A \in N$ y $\alpha \in (N \cup T)^*$.

- S es el **símbolo inicial**.

Observe que das CFG son gramáticas matriciales de grado 1, es decir 1-MG.

2.1.1. Proceso de derivación de una Gramática Matricial

El proceso de derivación en una gramática matricial se realiza de la siguiente manera:

1. Se selecciona una matriz $[P_1, P_2, \ldots, P_k] \in P$.
2. Las reglas $P_1, P_2, \ldots, P_k$ se aplican de manera secuencial a la cadena actual.
3. La derivación continúa hasta que la cadena derivada contenga solo símbolos terminales, es decir, pertenezca a T^*.

Un ejemplo de gramática matricial G_3 que genera el lenguaje L^3_{copy}, sobre el alfabeto $\Sigma = \{a, b, c\}$ es el siguiente:

$$G_3 = (V, P, S, \Sigma)$$

donde:

- $V = \{S, X, Y, Z\}$.
- $\Sigma = \{a, b, c\}$.
- S es el símbolo inicial.
- P está compuesto por las siguientes matrices de producción:

$$\begin{aligned}
&M_1 : [S \to XYZ] \\
&M_2 : [X \to aX, Y \to aY, Z \to aZ] \\
&M_3 : [X \to bX, Y \to bY, Z \to bZ] \\
&M_4 : [X \to cX, Y \to cY, Z \to cZ] \\
&M_5 : [X \to \varepsilon, Y \to \varepsilon, Z \to \varepsilon]
\end{aligned}$$

Ahora se presenta un caso particular de las gramáticas matriciales.

2.1.2. Gramáticas Matriciales Simples

Una gramática matricial simple de grado n *n-SMG* es una (n+3)-tupla:

$$G_n = (V_1, V_2, \ldots, V_n, P, S, \Sigma)$$

donde:

- $V_1, V_2, \ldots, V_n$ son conjuntos finitos de **símbolos no terminales** disjuntos 2 a 2.
- Σ es un conjunto finito de **símbolos terminales**, con $V_i \cap \Sigma = \emptyset \forall 1 \leq i \leq n$.
- P es un conjunto finito de matrices. Cada matriz es una secuencia ordenada **producciones** que cumplan con una de las siguientes reglas:
  - $[S \to w]$, donde $w \in \Sigma^*$.
  - $[S \to a_{11}A_{11}\ldots A_{1k}a_{21}A_{21}\ldots A_{2k}\ldots A_{n1}a_{n1}\ldots A_{nk}b]$, donde $\forall i, j$ con $1 \leq i \leq n \wedge 1 \leq j \leq k$ se cumple que $A_{ij} \in V_i$, $a_{ij} \in \Sigma^*$ y $b \in \Sigma^*$.
  - $[A_1 \to w_1, \ldots, A_n \to w_n]$, donde $A_i \in V_i \wedge w_i \in \Sigma^* \ \forall i\, 1 \leq i \leq n$.
  - $[A_1 \to a_{11}A_{11}\ldots a_{1k}A_{1k}b_1, \ldots, A_n \to a_{n1}A_{n1}\ldots a_{nk}A_{nk}b_n]$, donde $\forall i, j$ con $1 \leq i \leq n \wedge 1 \leq j \leq k$ se cumple que $A_{ij} \in V_i$, $a_{ij} \in \Sigma^*$ y $b_i \in \Sigma^*$.
- S es el **símbolo inicial**.

Observe que la restricción impuesta sobre las n-MG es cada matriz de producciones debe contener exactamente n reglas de producción donde cada regla de producción utiliza no terminales de conjuntos distintos o puede contener una única producción cuya secuencia de no terminales esta compuesta por una secuencia de subsecuencias de terminales de conjuntos distintos.

En la próxima sección se presentan las gramáticas de índice global.

2.2. Gramáticas de Índice Global

Una gramática de índice global *GIG*, es una extensión de las CFG, que añaden un mecanismo de memoria al proceso de derivación, esta característica permite la generación de lenguajes más generales que los generados por las CFG [6]. El mecanismo de memorización consiste en una pila en la cual se pueden almacenar símbolos que pertenecen a un conjunto predeterminado, en cada producción se puede realizar una operación de insertar o eliminar de la pila o dejarla en su estado actual.

Una GIG es una 6-tupla:

$$G = (N, \Sigma, I, S, \#, P)$$

donde:

- N es un conjunto finito de **símbolos no terminales**.
- Σ es un conjunto finito de **símbolos terminales**, $\Sigma \cap N = \emptyset$.
- I es un conjunto finito de **índices de pila**, $\Sigma \cap I = \emptyset \wedge I \cap I = \emptyset$.
- $S \in N$ es el **símbolo inicial**.
- $\#$ es el **símbolo inicial de la pila**, $\# \notin \Sigma \cup N \cup I$.
- P es un conjunto finito de **producciones** que tienen la siguiente forma, donde $x \in I \cup N \cup \Sigma$ y $y \in I \cup N$:
  - $A \underset{\varepsilon}{\rightarrow} \alpha$ o $A \rightarrow \alpha$ (reglas épsilon o reglas libres del contexto).
  - $A \underset{[y]}{\rightarrow} \alpha$ o $[..]A \rightarrow [..]\alpha$ (reglas épsilon o reglas con restricciones).
  - $A \underset{x}{\rightarrow} a\beta$ o $[..]A \rightarrow [x..]a\beta$ (reglas de *push* o apertura de paréntesis).
  - $A \underset{\overline{x}}{\rightarrow} \alpha$ o $[x..]A \rightarrow [..]\alpha$ (reglas de *pop* o cierre de paréntesis).

Como se puede observar la primera regla de producción consiste en dejar la pila intacta y puede ser interpretada como una regla de derivación libre del contexto. La segunda regla consiste en dejar la pila intacta pero solo se puede realizar si el caracter en el tope de la pila es el especificado en la regla de producción. La tercera regla consiste en añadir un caracter a la pila y la cuarta regla consiste en eliminar un caracter de la pila. Una gramática GIG solo con producciones de la primera regla de producción es equivalente a una CFG.

A continuación se describe el proceso de producción de las GIG.

2.2.1. Proceso de derivación

Como se mencionó anteriormente el proceso de derivación en las GIG es idéntico al proceso de derivación de las CFG, con la diferencia que en cada paso de la derivación se puede realizar una operación de insertar o eliminar de la pila, además de las operaciones de sustitución de símbolos. Otra restricción adicional es que en el proceso de derivación en las GIG solo se pueden usar producciones de extrema izquierda.

Una cadena es reconocida por una GIG si existe una secuencia de derivaciones desde S que genere la cadena y que además la pila termine vacía al final de la derivación (con el símbolo $\#$ en el tope de la pila).

Se define el lenguaje generado por una GIG, G como $L(G) = \{w \,|\, \#S \xrightarrow{*} \#w \wedge w \in \Sigma^*\}$.

El lenguaje L^{+}_{copy} se define como: $L^{+}_{copy} = \{ww^{+} \,|\, w \in Z^{*}\}$. A continuación se presenta una GIG que genera el lenguaje L^{+}_{copy} sobre el alfabeto $\Sigma = \{a, b\}$ [6]:

$$G_{ww^{+}} = (N, \Sigma, I, S, \#, P)$$

donde:

- $N = \{S, R, A, B, C\}$.
- $\Sigma = \{a, b\}$.
- $I = \{i, j\}$.
- S es el **símbolo inicial**.
- $\#$ es el **símbolo inicial de la pila**.
- P es un conjunto finito de **producciones**:
  - $S \underset{\varepsilon}{\rightarrow} AS \,|\, BS \,|\, C$
  - $C \underset{\varepsilon}{\rightarrow} RC \,|\, L$
  - $R \underset{\bar{i}}{\rightarrow} RA$
  - $R \underset{\bar{j}}{\rightarrow} RB$
  - $R \underset{[\#]}{\rightarrow} \varepsilon$
  - $A \underset{i}{\rightarrow} a$
  - $B \underset{j}{\rightarrow} b$
  - $L \underset{\bar{i}}{\rightarrow} La \,|\, a$
  - $L \underset{\bar{j}}{\rightarrow} Lb \,|\, b$

El funcionamiento de la gramática anterior es el siguiente: a cada caracter de Σ le corresponde un no terminal y un símbolo de la pila por el que se puede producir dicho caracter almacenando el caracter el símbolo de la pila correspondiente. El funcionamiento de la gramática lo compone además un mecanismo de recursión por el que se puede producir un no terminal asociado a un caracter solo eliminando el símbolo de la pila asociado a dicho caracter. Por último el mecanismo de recursión produce la cadena vacía solo si el símbolo inicial de la pila se encuentra en el tope.

En la próxima sección se mencionan las propiedades de las GIG que son de interés para este trabajo.

2.2.2. Propiedades de las GIG

En esta sección se describen las principales propiedades de las GIG que serán empleadas en este trabajo:

- **Cerradas bajo homomorfismo** Dada una GIG G, el homomorfismo de un lenguaje reconocido por G es un lenguaje de índice global [6].
- **Cerradas bajo transducción finita:** Dada una GIG G, la transducción finita del lenguaje que reconoce G es un lenguaje de índice global [6].

En la siguiente sección se describen las gramáticas de concatenación de rango.

2.3. Gramáticas de Concatenación de Rango

Las gramáticas de concatenación de rango (RCG) [5] son un formalismo de gramáticas desarrollado en 1988 como una propuesta de Pierre Boullier, un investigador en el campo de la lingüística computacional. Su objetivo principal era proporcionar un modelo más general y expresivo que las CFG para describir lenguajes que van más allá de los libres del contexto. Las RCG fueron diseñadas con el fin de capturar propiedades de lenguajes naturales y formales que requieren dependencias más complejas, como las que se encuentran en los lenguajes de programación y ciertas estructuras lingüísticas humanas.

2.3.1. Definiciones

Rango: un rango es una tupla (i, j) que representa un intervalo de posiciones en la cadena, donde i y j son enteros no negativos tales que $i \leq j$.

Gramática de Concatenación de Rango Positiva: una gramática de concatenación de rango positiva ($PRCG$) se define como una 5-tupla:

$$G = (N, T, V, P, S),$$

donde:

- N: Es un conjunto finito de **predicados o símbolos no terminales**: Cada predicado tiene una **aridad**, que indica el número de argumentos que toma.
- T: Es un conjunto finito de **símbolos terminales**.
- V: Es un conjunto finito de **variables**.
- P: Es un conjunto finito de **cláusulas**, de la forma:

  $$A(x_1, x_2, \ldots, x_k) \rightarrow B_1(y_{1,1}, y_{1,2}, \ldots, y_{1,m_1}) \ldots B_n(y_{n,1}, y_{n,2}, \ldots, y_{n,m_n}),$$

  donde $A, B_i \in N$, $x_i, y_{i,j} \in (V \cup T)^*$, y k es la aridad de A.
- $S \in N$: Es el **predicado inicial** de la gramática.

Gramática de Concatenación de Rango Simple: las gramáticas de concatenación de rango simple (*SRCG*) son un subconjunto de las RCG que restringen la forma de las cláusulas de producción. Una RCG G es **simple** si los argumentos en el lado derecho de una cláusula son variables distintas, y todas estas variables (y no otras) aparecen una sola vez en los argumentos del lado izquierdo. Para cada CFG existe una SRCG equivalente que genera el mismo lenguaje [5].

Por ejemplo una cláusulas de una SRCG pueden tener la siguiente estructura:

$$A(X_1, X_2, \ldots, X_k) \rightarrow B_1(X_1) \ldots B_n(X_n).$$

Los predicados B_i pueden tener más de un argumento siempre y cuando estos sean variables distintas en este caso la cantidad de predicados B_i debe ser menor que n ya que todas las variables en el lado derecho deben aparecer en el lado izquierdo de la cláusula y viceversa:

$$A(X_1, X_2, \ldots, X_k) \rightarrow B_1(X_1, X_n), B_2(X_2) \ldots B_{n-1}(X_{n-1}).$$

Sustitución de rango: una sustitución de rango es un mecanismo que reemplaza una variable por un rango de la cadena. Por ejemplo dado el predicado $A(Xa)$ donde $X \in V \wedge a \in T$, si se instancia la cadena baa en A, X puede ser asociada con el rango ba de la cadena original.

En la próxima sección se describe el proceso de derivación de las RCG.

2.3.2. Proceso de derivación

La principal idea detrás de las RCG, para realizar una derivación, se basa en encontrar para cada argumento del predicado izquierdo de una cláusula todas las posibles sustituciones en rango de la cadena, entonces es necesario asociar los valores de las variables que se reconocen en los argumentos del predicado izquierdo a los argumentos de los predicados derechos y continuar el proceso de derivación en los predicados derechos.

Por ejemplo, dada la cláusula $A(X, aYb) \rightarrow B(aXb, Y)$, donde X y Y son símbolos variables y a y b son símbolos terminales, la cadena predicado $A(a, abb)$ deriva como $B(aab, b)$, porque $A(a, abb)$ coincide con $A(X, aYb)$ cuando $X = a \wedge Y = b$.

Las RCG a diferencia de las gramáticas anteriores no generan cadenas, su funcionamiento se basa en reconocer si una cadena pertenece o no al lenguaje.

Una secuencia de argumentos se reconocen por un predicado si existe una secuencia de derivaciones que comienza en dicho predicado y termina en la cadena vacía. Dada la siguiente cláusula $A(X_1, \ldots, X_n) \rightarrow B_1(X_1) \ldots B_n(X_n)$, para los rangos $w_1, \ldots, w_n$ asociados a las variables $X_1, \ldots, X_n$ respectivamente, si existe una secuencia de derivaciones para cada uno de los predicados $B_1(w_1), \ldots, B_n(w_n)$ que derive en la cadena vacía, entonces se reconoce el predicado $A(w_1, \ldots, w_n)$.

Por ejemplo, dada la siguiente RCG:

$$G = (N, T, V, P, S),$$

donde:

- N=$\{A, S\}$.
- T=$\{a, b, c\}$.
- V=$\{X, Y, Z\}$.
- El conjunto de cláusulas P es el siguiente:

$$S(XYZ) \rightarrow A(X, Y, Z)$$
$$A(aX, aY, aZ) \rightarrow A(X, Y, Z)$$
$$A(bX, bY, bZ) \rightarrow A(X, Y, Z)$$
$$A(cX, cY, cZ) \rightarrow A(X, Y, Z)$$
$$A(\varepsilon, \varepsilon, \varepsilon) \rightarrow \varepsilon$$

- El símbolo inicial es S.

La cadena *abcabcabc* es reconocida por la RCG anterior, ya que se puede derivar de la siguiente manera:

$$S(abcabcabc) \rightarrow A(abc, abc, abc) \rightarrow A(bc, bc, bc) \rightarrow A(c, c, c) \rightarrow A(\varepsilon, \varepsilon, \varepsilon) \rightarrow \varepsilon$$

.

En el primer paso de la derivación se toma la primera cláusula, la sustitución en rango asocia las variables $X = abc$, $Y = abc$ y $Z = abc$ a los rangos $w[0 \ldots 2]$, $w[3 \ldots 5]$ y $w[6 \ldots 8]$ ($w[i \ldots j]$ representa el rango que va desde el i-ésimo caracter hasta el j-ésimo con la cadena indexada en 0) respectivamente, derivando en el predicado $A(abc, abc, abc)$. En el segundo paso se toma la segunda cláusula, la sustitución en rango asocia las variables $X = bc$, $Y = bc$ y $Z = bc$ a los rangos $w[1 \ldots 2]$, $w[4 \ldots 5]$ y $w[7 \ldots 8]$ respectivamente, derivando en el predicado $A(bc, bc, bc)$. En el tercer paso se toma la tercera cláusula, la sustitución en rango asocia las variables $X = c$, $Y = c$ y $Z = c$ a los rangos $w[2 \ldots 2]$, $w[5 \ldots 5]$ y $w[8 \ldots 8]$ respectivamente, derivando en el predicado $A(c, c, c)$. En el cuarto paso se toma la cuarta cláusula, la sustitución en rango asocia las variables $X = \varepsilon$, $Y = \varepsilon$ y $Z = \varepsilon$ respectivamente, derivando en el predicado $A(\varepsilon, \varepsilon, \varepsilon)$. Finalmente en el último paso se toma la última cláusula que deriva en la cadena vacía, por lo que de esta manera se reconoce la cadena *abcabcabc*.

El lenguaje que reconoce la RCG anterior es L^3_{copy}, sobre el alfabeto $\Sigma = \{a, b, c\}$.

A continuación se presentan las principales propiedades de las RCG.

2.3.3. Propiedades de las RCG

A continuación se describen las principales propiedades de las RCG [3]:

- **Cerradura bajo unión:** Dadas dos RCG G_1 y G_2, la unión de los lenguajes reconocidos por G_1 y G_2 se reconoce por una RCG [5].
- **Cerradas bajo intersección:** Dadas dos RCG G_1 y G_2, la intersección de los lenguajes reconocidos por G_1 y G_2 es reconocida por una RCG [5].
- **Cerradas bajo complemento:** Dada una RCG G, el complemento del lenguaje reconocido por G se reconoce por una RCG [5].
- **Cerradas bajo concatenación:** Dadas dos RCG G_1 y G_2, la concatenación de los lenguajes reconocidos por G_1 y G_2 se reconoce por una RCG [5].
- **Cerradas bajo clausura de Kleene:** Dada una RCG G, la clausura de Kleene del lenguaje se reconoce por G es reconocida por una RCG [5].
- **No cerradas bajo homomorfismo:** Dada una RCG G, el homomorfismo de un lenguaje reconocido por G no es necesariamente se reconoce por una RCG [3].
- **No cerradas bajo transducción finita:** Dada una RCG G, la transducción finita de un lenguaje reconocido por G no es necesariamente se reconoce por una RCG. Esto es una consecuencia de la propiedad anterior ya que como se mencionó en el capítulo anterior un homomorfismo es un caso particular de un transductor finito. Esta propiedad trae implicaciones en una cuestión analizada en el capítulo 5.

2.3.4. Problema de la palabra, problema del vacío y equivalencia de 2 RCG

Problema de la palabra

En general en la mayoría de los casos este problema es polinomial y pasa por un algoritmo de memorización sobre las cadenas que son instanciadas en los rangos de los predicados de la RCG [5]. Como la cantidad máxima de rangos de la cadena es n^2 y la máxima aridad de un predicado es constante, este proceso de memorización cuenta con cantidad polinomial de estados, en una complejidad de $O(|P|n^{2h(l+1)})$ donde h es la máxima aridad en un predicado, l es la máxima cantidad de predicados en el lado derecho de una cláusula y n es la longitud de la cadena que se reconoce.

Pero existen casos en los que el problema de la palabra no es polinomial [3], el ejemplo presentado en [3] muestra una RCG que se enfoca en el problema de contar, en este trabajo se analiza otro caso en el que este problema no es polinomial.

Problema de la palabra no polinomial

El algoritmo de reconocimiento presentado en la sección anterior utiliza un proceso de memorización sobre los rangos de la cadena, la idea fundamental para esto y lo que acota la complejidad del algoritmo es que la cantidad de estados asociados a la memorización es igual a la cantidad de rangos de la cadena, el cual es polinomial con respecto a la longitud de la cadena.

Ahora ¿que pasaría si algún predicado de la gramática trabajara con rangos que no pertenecen a la cadena original? En este caso si se emplea el algoritmo anterior ya la complejidad no depende de la cantidad de rangos de la cadena original porque pueden aparecer otros rangos que no pertenezcan a dicha cadena.

Por ejemplo a continuación se presenta una RCG que no tiene uso real porque existe otra RCG equivalente que reconoce el mismo lenguaje, pero ilustra lo descrito anteriormente:

$$G = (N, T, V, P, S),$$

donde:

- N=$\{A, B, Eq, S\}$.
- T=$\{0, 1\}$.
- V=$\{X, Y\}$.
- El conjunto de cláusulas P es el siguiente:

$$S(X) \to A(X, X)$$
$$A(1X, Y) \to B(X, 0, Y)$$
$$A(1X, Y) \to B(X, 1, Y)$$
$$A(0X, Y) \to B(X, 1, Y)$$
$$A(0X, Y) \to B(X, 0, Y)$$
$$B(1X, Y, Z) \to B(X, 1Y, Z)$$
$$B(1X, Y, Z) \to B(X, 0Y, Z)$$
$$B(0X, Y, Z) \to B(X, 0Y, Z)$$
$$B(0X, Y, Z) \to B(X, 1Y, Z)$$
$$B(\varepsilon, Y, Z) \to Eq(Y, Z)$$

- El símbolo inicial es S.

El funcionamiento de la gramática anterior se basa en dada una cadena w generar todas las posibles cadenas q, tales que $|w| = |q|$ y luego comprobar si $w = q$. Observe que como se dijo anteriormente está gramática no tiene caso de uso ya que para toda cadena w siempre va a existir una cadena q tal que $w = q$, por lo que se puede modelar con solamente la cláusula $S(X) \to \varepsilon$. Pero sin embargo la complejidad del reconocimiento de G es mayor que 2^n (con n igual al tamaño de la cadena d entrada), ya que esta es la cantidad de rangos posibles que puede recibir el predicado B. En el capítulo 5 se presenta una RCG de este estilo, pero que si tiene caso de uso en los problemas de la Ciencia de la Computación.

En la próxima sección se aborda el problema del vacío para las RCG.

Problema del vacío

El problema del vacío para una RCG es indecidible [3]. La razón principal para esto es que como para toda CFG existe una RCG equivalente y como las RCG son cerradas bajo intersección existen RCG que describen la intersección de 2 lenguajes libres del contexto y determinar si dicha intersección es vacía es un problema indecidible [3].

En el caso de las SRCG este problema es polinomial [5]. En el capítulo 3 se analiza un caso de uso de las SRCG donde esta propiedad juega un rol importante.

En este capítulo se analizaron los principales formalismos de escritura regulada, que son necesarios para la compresión de los siguientes capítulos. Todos los formalismos descritos generan lenguajes dependientes del contexto y sirven para solucionar problemas que modelan el SAT utilizando teoría de lenguajes.

Capítulo 3

Estrategia para la solución del SAT usando el problema del vació

En el presente capítulo se presenta una estrategia para la solución del SAT, la cual dado un SAT, construye un formalismo que genera todas todas las interpretaciones que la hace verdadera la fórmula booleana de dicho SAT, entonces para comprobar si el SAT es satisfacible solo resta comprobar si el conjunto de cadenas generadas por el formalismo es no vacío.

3.1. Antecedentes

Como parte del estudio del problema SAT, se han desarrollado anteriormente en la Facultad de Matemática y Computación de la Universidad de La Habana una serie de trabajos utilizando el enfoque que se describe en este capítulo basado en formalismos de la teoría de lenguajes, buscando resolver instancias específicas del SAT, las cuales serán abordadas en las secciones del presente capítulo.

La idea principal que se aborda en [1] consta de tres partes: asumir que todas las variables en la fórmula son distintas, construir un autómata finito que reconozca cadenas de 0 y 1 hagan verdadera esa fórmula (asumiendo que todas las variables son distintas), y por último intersectar ese lenguaje con algún formalismo que garantice que todas las instancias de la misma variable tenga el mismo valor. Luego de esos tres pasos, se obtiene un lenguaje libre del contexto de las cadenas de 0 y 1 que satisfacen la fórmula y que además respeta los valores de las variables duplicadas. Finalmente, para determinar si la fórmula es satisfacible o no, basta con determinar si el lenguaje es vacío. Si es vacío, no es satisfacible, si lo es, pues no es satisfacible. Todo el algoritmo descrito anteriormente tiene un tiempo polinomial en relación con el tamaño de la fórmula booleana.

En la siguiente sección se presenta un mecanismo para reconocer si una cadena de 0 y 1 satisface una fórmula booleana (asumiendo que todas las variables son distintas).

3.2. Autómata booleano

El primer paso para el proceso descrito anteriormente es construir un autómata finito que depende de la estructura de la fórmula booleana y verifica si una cadena de 0 y 1 satisface dicha fórmula (asumiendo que todas las variables son distintas), dicho autómata se denomina **autómata booleano**.

La idea detrás del autómata booleano y es representar las reglas de la lógica proposicional en transiciones entre los estados de un autómata finito, donde cada estado del autómata representa un valor de verdad positivo o negativo lo cual significa que hasta ese momento (solo tomando las instancias de las variables asociadas a los caracteres reconocidos) la fórmula se evalúa positiva o negativa respectivamente [1].

3.3. Problema de la satisfacibilidad booleana libre del contexto

El problema satisfacibilidad booleana libre del contexto (*CF-SAT*) [1], es una instancia específica del SAT donde las variables donde la fórmula booleana es una fórmula booleana libre del contexto.

Fórmula booleana libre del contexto: una fórmula booleana se considera libre del contexto (*CF-BF*) si para cualquier par de instancias de una variable x_i y x_j con $i < j$ se cumple que si existe otra variable con instancia x_k con $i < k < j$ entonces todas las instancias de esta nueva variable ocurren entre x_i y x_j.

Dada la definición anterior posible clasificar las instancias de cada variable en una fórmula booleana:

1. **Clase libre:** a esta clase pertenecen todas las instancias de las variables que ocurren una única vez en la fórmula booleana.

2. **Clase primera:** a esta clase pertenecen todas las instancias que representan la primera ocurrencia de una variable en la fórmula booleana (la variable debe ocurrir más de una vez).

3. **Clase intermedia:** a esta clase pertenecen todas las instancias que no representan ni la primera, ni la última ocurrencia de una variable en la fórmula booleana (la variable debe ocurrir más de una vez).

4. **Clase última:** a esta clase pertenecen todas las instancias que representan la última ocurrencia de una variable en la fórmula booleana (la variable debe ocurrir más de una vez).

A continuación se describe la solución al CF-SAT.

3.3.1. Solución al CF-SAT

El orden que siguen las instancias de las variables en una CF-BF puede ser reconocido por una CFG que a su vez trae asociado un autómata de pila, al cual de le denomina autómata de pila booleano [1].

Para la construcción del autómata de pila booleano correspondiente a una CF-BF se toma el autómata booleano asociado a la fórmula booleana y se le añade un mecanismo de memoria. Dicho mecanismo de memoria funciona de la siguiente manera:

- Si el caracter analizado corresponde a una instancia de clase libre, dicho caracter no se almacena en la pila.
- Si el caracter analizado corresponde a una instancia de clase primera, dicho caracter se almacena en la pila.
- Si el caracter analizado corresponde a una instancia de clase intermedia, dicho caracter no se almacena en la pila y se comprueba que su valor sea igual al del tope de la pila.
- Si el caracter analizado corresponde a una instancia de clase última, dicho caracter no se almacena en la pila, se comprueba que su valor sea igual al del tope de la pila y se retira el valor del tope de la pila.

El autómata de pila booleano reconoce una cadena si termina en un estado final con la pila vacía.

En [1] se describe como dado un autómata de pila encontrar la CFG correspondiente y luego se emplea un algoritmo que permite determinar si el lenguaje generado con por la gramática es vacío o no [10]. Posteriormente se enuncia un algoritmo para generar todas las cadenas que pertenecen a dicha gramática. Todo el proceso anteriormente descrito tiene una complejidad de $O(n^3)$, donde n es la cantidad de instancias de variables en la CF-BF.

A continuación se presenta una generalización del CF-SAT.

Generalización de la solución al CF-SAT

De manera general el algoritmo presentado en [1] puede ser generalizado a otras instancias de SAT. Al seguir la idea del autómata booleano (asumiendo que las instancias de las variables son distintas), solo es necesario emplear un formalismo que verifique que los valores de las mismas instancias de las variables sean iguales e interceptarlo con el autómata. Dicho formalismo debe cumplir 2 propiedades: ser cerrado bajo la intersección con lenguajes regulares y poder resolver el problema del vacío en tiempo polinomial. Por tanto si se selecciona un formalismo más general que el empleado en [1] pueden resolverse instancias más generales de SAT.

Otro resultado interesante es encontrar un algoritmo que permita generar todas las posibles cadenas que pertenecen al lenguaje generado por la intersección del autómata booleano y el formalismo seleccionado, con estas cadenas se pueden obtener los valores de verdad para cada una de las variables que satisfacen la fórmula booleana.

En las próximas secciones se presentan los problemas SAT asociados a los trabajos que siguen el enfoque descrito anteriormente que han sido 2: el primero usa las gramáticas de concatenación de rango simple para definir el orden de las variables y el segundo requiere un representación de un SAT como una cadena y usa esta representación para interceptar el autómata booleano con una gramática matricial simple que sirve de mecanismo de control para el valor de las instancias de las variables.

3.4. Problema de la satisfacibilidad booleana de concatenación de rango simple

Una secuencia de instancias de variables de longitud n tiene un orden de concatenación de rango simple [12], si es posible construir una SRCG, con un tamaño polinomial en n, que genere el lenguaje de todas las cadenas $w_1 w_2 \dots w_n$, de longitud n sobre el alfabeto 0,1, tal que para todo par de instancias de variables (x_i, x_j) que pertenezcan a la misma variable se cumple que $w_i = w_j$.

El problema de la satisfacibilidad booleana de concatenación de rango simple (*SRC-SAT*) es una instancia de SAT donde la secuencia de instancias de sus variables posee un orden de concatenación de rango simple.

3.4.1. Solución al SRC-SAT

Para la solución del SRC-SAT en [12] se procede de manera similar a [1], pero esta vez el autómata booleano se intersecta con una SRCG que describe el orden de la fórmula booleana.

Luego solo queda resolver el problema del vacío para el formalismo resultante. Para esto se obtiene una CFG que representa el lenguaje de todas las posibles formas

de generar la menor cadena en la intersección de la SRCG con el autómata booleano y solo queda verificar que dicha CFG sea no vacía como se muestra en [1]. La complejidad de todo este proceso es $O(n^k)$ donde n es el número de instancias de variables en la fórmula booleana y k es la aridad de la SRCG, observe que este algoritmo es polinomial siempre y cuando la aridad de la gramática sea constante.

El SRC-SAT constituye un paso de avance con respecto al CF-SAT ya que describe un espacio mucho más amplio de instancias de SAT que pueden ser resueltas en un tiempo polinomial. Como se mencionó en el capítulo anterior para todo CFG existe una SRCG equivalente

En las próximas secciones se aborda el primer enfoque para analizar la solución de un SAT de manera general, es decir, no solo se resuelven instancias específicas como en las secciones anteriores, para ello primeramente es necesario definir la representación de cualquier fórmula booleana en CNF como una cadena.

3.5. Transformación de una fórmula booleana a una cadena

Dada una fórmula booleana F en CNF se puede definir la siguiente estructura:

$$F = X_1 \wedge X_2 \wedge \ldots \wedge X_n$$

donde cada cláusula X_i es una disyunción de literales:

$$X_i = L_{i1} \vee L_{i2} \vee \ldots \vee L_{im}$$

y cada literal L_{ij} es una variable booleana o su negación. En cada cláusula X_i las variables que aparecen en F, puede tener cada una 3 estados posibles: a si la variable aparece positiva, b si la variable aparece negada y c si la variable no pertenece a ninguno de los literales de la cláusula.

Por ejemplo la siguiente fórmula booleana en CNF:

$$F = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3)$$

para la primera cláusula x_1 aparece positiva (a), x_2 aparece positiva (a) y x_3 no aparece (c), para la segunda x_1 aparece negada (b), x_2 aparece positiva (a) y x_3 aparece positiva (a) y para la tercera x_1 aparece positiva (a), x_2 aparece negada (b) y x_3 aparece positiva (a).

A partir de la afirmación anterior, se puede definir una cadena de símbolos w que representa a la cláusula X_i sobre una secuencia de variables $v_1, v_2, \ldots, v_p$ de la siguiente manera:

- w cuenta con exactamente p símbolos.
- Si la variable v_j aparece positiva en X_i, entonces el j-ésimo símbolo es a.
- Si la variable v_j aparece negada en X_i, entonces el j-ésimo símbolo es b.
- Si la variable v_j no aparece en X_i, entonces el j-ésimo símbolo es c.

Si se toma la secuencia de variables correspondiente a F, y se le aplica el procedimiento anterior a cada cláusula se obtiene una cadena de símbolos que representa a dicha cláusula en F.

Si ya se tiene una representación para cada cláusula de F solo resta obtener una cadena de símbolos que represente a F, esto se puede lograr concatenando las cadenas de símbolos de cada cláusula de F en el orden que aparecen con un separador en este caso se eligió el símbolo d.

Por ejemplo la siguiente fórmula booleana en *CNF*:

$$F = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3)$$

puede ser expresada como la cadena de símbolos:

$$w = aacdbaadabad$$

tomando como secuencia de variables x_1, x_2, x_3.

3.6. Primera aproximación a la solución del SAT usando gramáticas matriciales simples

En [13] se propone un nuevo enfoque a los 2 problemas anteriores, el cual se basa en eliminar la restricción de la estructura de la fórmula booleana y en su lugar se trabaja con la transformación de la misma en una cadena.

Con esta nueva estructura se modifica el autómata booleano correspondiente a dicha fórmula para que trabaje con cadenas que pertenezcan al lenguaje $L_m^n = \{w^m \,|\, w \in \{0,1\} \wedge |w| = n\}$, observe que w se puede interpretar como una asignación de valores 0 ó 1 para cada una de las variables, que representan su valor de verdad.

3.6.1. Lenguaje L_m^n

En [13] se define una gramática matricial simple que describe el lenguaje L_m^n:

$$G_n = (\{A_1, A_{11}, \ldots A_{1m}\}, \ldots, \{A_n, A_{n1}, \ldots A_{nm}\}, \Sigma, P, S, \Sigma)$$

donde:

- $\Sigma = \{0, 1\}$.
- P es un conjunto finito de matrices de producciones:
  - $[S \to A_1 A_2 \dots A_n]$
  - $[A_1 \to 1A_{12}, \dots, A_n \to 1A_{n2}]$
  - $[A_1 \to 0A_{12}, \dots, A_n \to 0A_{n2}]$
  - Para $2 \leq i < m$, $[A_{1i} \to 1A_{1(i+1)}, \dots, A_{ni} \to 1A_{n(i+1)}]$
  - Para $2 \leq i < m$, $[A_{1i} \to 0A_{1(i+1)}, \dots, A_{ni} \to 0A_{n(i+1)}]$
  - $[A_{1m} \to 1, \dots, A_{nm} \to 1]$
  - $[A_{1m} \to 0, \dots, A_{nm} \to 0]$
- S es el **símbolo inicial**.

En [11] se demuestra que esta gramática en efecto, genera le lenguaje L_m^n.

Ahora solo resta interceptar esta gramática con el autómata booleano para determinar si el SAT es satisfacible.

3.6.2. Intersección con el autómata booleano

En [13] se menciona que las $n - SMG$ son cerradas bajo la intersección con lenguajes regulares, pero cuando se analiza la intersección del autómata con la $n - SMG$ que describe el lenguaje L_m^n se obtiene una gramática con una cantidad de producciones exponencial en relación a la cantidad de variables de la fórmula booleana, lo cual impide analizar si el lenguaje resultante es vacío en un tiempo polinomial.

Este es el primer acercamiento para la solución de cualquier instancia del SAT (aunque sigue siendo una solución exponencial en términos de complejidad computacional), en el próximo capítulo se presenta una estrategia de solución diferente que busca otras alternativas para la solución de cualquier instancia del SAT.

Capítulo 4

Estrategia para la solución del SAT usando el problema de la palabra

En este capitulo se presenta un enfoque distinto al del capítulo anterior, que se basa en definir un lenguaje al cual pertenecen todos los problemas SAT que son satisfacibles y al cual se le denomina L_{S-SAT}. Una vez definido este lenguaje, para determinar si un SAT es satisfacible solo es necesario verificar si la cadena que lo representa pertenece a L_{S-SAT}.

4.1. Definición de L_{S-SAT}

El lenguaje de todas las fórmulas booleanas en CNF que son satisfacibles se define como $L_{S-SAT} = \{w \,|\, w \in L_{FULL-SAT} \wedge f_{SAT}(w)\}$, donde $L_{FULL-SAT}$ representa el lenguaje de todas las fórmulas booleanas en CNF y $f_{SAT}(w)$ es una función que determina si w es satisfacible.

En las próximas secciones se presenta el primer enfoque para definir L_{S-SAT}, el cual se basa en definir el lenguaje mediante un transductor finito seleccionando un formalismo que sea cerrado bajo transducción finita.

4.2. Transductor T_{SAT}

La idea para definir L_{S-SAT} es construir un transductor finito que acepte como entrada cadenas del lenguaje $L_{0,1} = \{wd\}^+$ donde $w \in \{0,1\}^*$ y devuelva como salida cadenas que representan una fórmula booleana en *CNF* que sea verdadera si se evalúa en la cadena que el transductor recibió como entrada.

Detrás de esta construcción se busca asociar cada carácter 0 ó 1 en la cadena de entrada al valor de la variable booleana correspondiente en la cadena de salida y

verificar que para dichos valores al evaluar la fórmula booleana se obtenga un valor de positivo.

A continuación se define el transductor finito T_{SAT} (Figura 4.2) que sigue la construcción definida anteriormente, para ello se define el transductor T_{CLAUSE} (Figura 4.1) que hace el proceso de transducción para los valores de las variables de una cláusula wd, donde $w \in \{0,1\}$:

$$T_{CLAUSE} = (Q, \Sigma, \Gamma, \delta, q_0, F),$$

donde:

- $Q = q_0, q_p, q_n$.
- $\Sigma = 0, 1$.
- $\Gamma = a, b, c$.
- $\delta : Q \times \Sigma \to Q \times \Gamma^*$ función de transición.
- $q_0 = q_0$ estado inicial.
- $F = q_p$ conjunto de estados finales.

se define la función de transición δ de la siguiente manera:

- transiciones para el estado q_0: representa el estado inicial. Si la entrada es un 1 el transductor puede escribir a, b y c, si escribe a pasa al estado positivo, si escribe b pasa al estado negativo y si escribe c permanece en el mismo estado. Por otro lado si la entrada es un 0 se intercambian los estados cuando se escribe a y b y cuando se escribe c permanece en el mismo estado.
  - $\delta_{SAT}(q_0, 1) = (q_p, a)$
  - $\delta_{SAT}(q_0, 0) = (q_n, a)$
  - $\delta_{SAT}(q_0, 1) = (q_n, b)$
  - $\delta_{SAT}(q_0, 0) = (q_p, b)$
  - $\delta_{SAT}(q_0, 1) = (q_0, c)$
  - $\delta_{SAT}(q_0, 0) = (q_0, c)$
- transiciones para el estado q_p (estado positivo de T_{CLAUSE}): representa que para los valores asignados a las variables se obtiene un valor de verdad positivo. Como la fórmula se encuentra ya en un estado positivo lo que significa que al menos un literal se evaluó positivo no importa la entrada y lo que el transductor escriba se mantiene en el mismo estado. Este es el estado de aceptación para el transductor lo cual significa que la cláusula toma un valor de verdad positivo.

- $\delta_{SAT}(q_p, 1) = (q_p, a)$
- $\delta_{SAT}(q_p, 0) = (q_p, a)$
- $\delta_{SAT}(q_p, 1) = (q_p, b)$
- $\delta_{SAT}(q_p, 0) = (q_p, b)$
- $\delta_{SAT}(q_p, 1) = (q_p, c)$
- $\delta_{SAT}(q_p, 0) = (q_p, c)$

- transiciones para el estado q_n (estado negativo de T_{CLAUSE}): representa que para los valores asignados a las variables se obtiene un valor de verdad negativo. Aquí las transiciones son idénticas a las del estado inicial, reemplazando el estado inicial por el estado negativo en los posibles resultados de la función de transición.

  - $\delta_{SAT}(q_n, 1) = (q_p, a)$
  - $\delta_{SAT}(q_n, 0) = (q_n, a)$
  - $\delta_{SAT}(q_n, 1) = (q_n, b)$
  - $\delta_{SAT}(q_n, 0) = (q_p, b)$
  - $\delta_{SAT}(q_n, 1) = (q_n, c)$
  - $\delta_{SAT}(q_n, 0) = (q_n, c)$

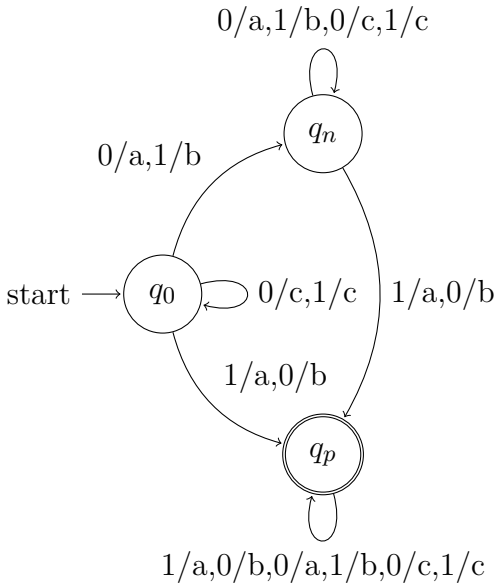

Figura 4.1: Transductor T_{CLAUSE}.

Para definir el transductor T_{SAT} se toman dos instancias del transductor T_{CLAUSE} (T_1 y T_2 respectivamente) y se concatenan añadiendo una transición del estado q_{p_1} (estado positivo de T_1) al estado q_{0_2} (estado inicial de T_2) con el símbolo d (tanto de lectura como de escritura) y además se agrega una clausura a T_2 con una transición del estado q_{p_2} (estado positivo de T_2) al estado q_{0_2} con el símbolo d (tanto de lectura como

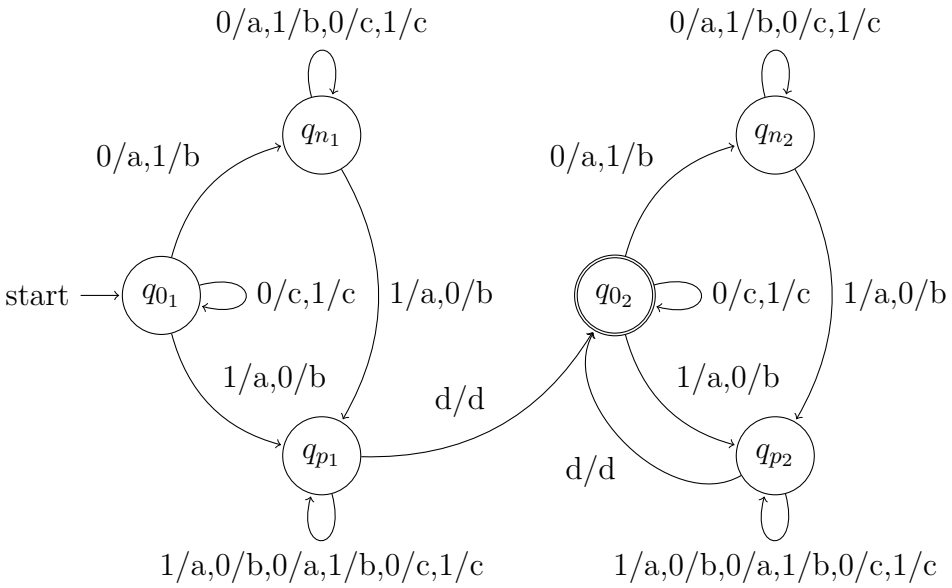

Figura 4.2: Transductor T_{SAT}.

de escritura). Entonces solo resta definir el estado inicial y el estado final de T_{SAT}, los cuales serían q_{0_1} (estado inicial de T_1) y q_{0_2} (estado inicial de T_2), respectivamente.

A continuación se define el lenguaje L_{S-SAT} usando transducción finita.

4.2.1. Definición del L_{S-SAT} usando transducción finita

Finalmente se define L_{S-SAT} como el lenguaje de todas las transducciones e que se obtienen del transductor T_{SAT}, a partir del lenguaje de cadenas de entrada $L_{0,1} = \{wd\}^+$ donde $w \in \{0,1\}^*$.

$$L_{S-SAT} = \{e \,|\, \exists w \in L_{0,1} \wedge e \in T_{SAT}(w)\}.$$

Aunque L_{S-SAT} contiene todas las fórmulas booleanas satisfacibles, este conjunto por si solo no sirve de mucho sin un formalismo que permita conocer si una cadena que representa una fórmula booleana pertenece al lenguaje o no. Para ello se necesita encontrar un formalismo que sea capaz de generar el lenguaje $L_{0,1}$ y al aplicarle el transductor T_{SAT} a dicho formalismo se obtenga un formalismo que cuente con un algoritmo de reconocimiento para reconocer si una cadena pertenece a dicho formalismo o no.

Encontrar un formalismo que genere el lenguaje $L_{0,1}$, y que sea cerrado bajo transducción finita es suficiente para generar el lenguaje L_{S-SAT}. Una pregunta relacionada con L_{S-SAT} sería saber si la existencia de dicho formalismo es una condición

necesaria para definir dicho lenguaje. Otra pregunta sería saber si existe un formalismo que sea capaz de describir el lenguaje L_{S-SAT} y el problema de la palabra en dicho formalismo sea polinomial (observe que de esta manera se estaría resolviendo el problema **P vs NP**).

Mediante la transformación $L_{0,1}$ por el transductor finito se mantiene la invariante fundamental del SAT que a dos instancias de la misma variable se les asocia el mismo valor de verdad.

Dado la transformación $L_{0,1}$ por el transductor finito se puede demostrar que el problema de la palabra de cualquier formalismo que genere el lenguaje $L_{0,1}$ y sea cerrado bajo transducción finita es NP-Duro, ya que puede ser reducido al SAT y por tanto a cualquier problema en NP.

Ahora se presenta un formalismo que sirve para generar L_{S-SAT} usando el transductor T_{SAT}.

4.3. L_{S-SAT} como lenguaje de índice global

En esta sección se presenta una una forma de generar el lenguaje $L_{0,1}$ empleando una gramática de índice global.

Dada esta gramática de índice global que describe el lenguaje L^{+}_{copy} que se define en el capítulo 2, es posible realizar una modificación para generar el lenguaje $L_{0,1}$, a esta nueva gramática se le denominará $G_{0,1}$, que se define como:

$$G_{0,1} = (N, \Sigma, I, S, \#, P)$$

donde:

- $N = \{S, R, A, B, C, D\}$.
- $\Sigma = \{0, 1, d\}$.
- $I = \{i, j, k\}$.
- S es el **símbolo inicial**.
- $\#$ es el **símbolo inicial de la pila**.
- P es un conjunto finito de **producciones**:
  - $S \underset{\varepsilon}{\rightarrow} AS \,|\, BS \,|\, DC$
  - $C \underset{\varepsilon}{\rightarrow} RC \,|\, L$
  - $R \underset{\bar{i}}{\rightarrow} RA$
  - $R \underset{\bar{j}}{\rightarrow} RB$
  - $R \underset{\bar{k}}{\rightarrow} RD$

- $R \underset{[\#]}{\rightarrow} \varepsilon$
- $A \underset{i}{\rightarrow} 1$
- $B \underset{j}{\rightarrow} 0$
- $D \underset{k}{\rightarrow} d$
- $L \underset{\bar{i}}{\rightarrow} L$
- $L \underset{\bar{j}}{\rightarrow} L$
- $L \underset{\bar{k}}{\rightarrow} L$
- $L \underset{[\#]}{\rightarrow} \varepsilon$

Como modificaciones a la gramática anterior se ha introducido un nuevo terminal, un no terminal y un símbolo de la pila manteniendo la invariante de correspondencia que se mencionó anteriormente entre los elementos de estos 3 conjuntos. Por otro lado se modificaron las producciones del no terminal L para que unicamente produzca la cadena vacía eliminando todos los elementos de la pila, con ello se puede generar el lenguaje $L_{0,1}$.

A continuación se presenta un ejemplo de una cadena que se obtiene como derivación de la gramática $G_{0,1}$.

4.3.1. Ejemplo de reconocimiento de $G_{0,1}$

En esta sección se presenta una secuencia de derivaciones que permiten obtener la cadena $110d110d110d$ mediante la gramática $G_{0,1}$ (para cada derivación primero se muestra la regla de producción. seguido del estado de la pila y del estado del la cadena generada):

1. $S \underset{\varepsilon}{\rightarrow} AS$ $[\#]$ AS
2. $A \underset{i}{\rightarrow} 1$ $[i\#]$ $1S$
3. $S \underset{\varepsilon}{\rightarrow} AS$ $[i\#]$ $1AS$
4. $A \underset{i}{\rightarrow} 1$ $[ii\#]$ $11S$
5. $S \underset{\varepsilon}{\rightarrow} BS$ $[ii\#]$ $11BS$
6. $B \underset{j}{\rightarrow} 0$ $[jii\#]$ $110S$
7. $S \underset{\varepsilon}{\rightarrow} DC$ $[jii\#]$ $110DC$
8. $D \underset{k}{\rightarrow} d$ $[kjii\#]$ $110dC$
9. $C \underset{\varepsilon}{\rightarrow} RC$ $[kjii\#]$ $110dRC$
10. $R \underset{\bar{k}}{\rightarrow} RD$ $[jii\#]$ $110dRDC$
11. $R \underset{\bar{j}}{\rightarrow} RB$ $[ii\#]$ $110dRBDC$
12. $R \underset{\bar{i}}{\rightarrow} RA$ $[i\#]$ $110dRABDC$
13. $R \underset{\bar{i}}{\rightarrow} RA$ $[\#]$ $110dRAABDC$
14. $R \underset{[\#]}{\rightarrow} \varepsilon$ $[\#]$ $110dAABDC$
15. $A \underset{i}{\rightarrow} 1$ $[i\#]$ $110d1ABDC$
16. $A \underset{i}{\rightarrow} 1$ $[ii\#]$ $110d11BDC$
17. $B \underset{i}{\rightarrow} 0$ $[jii\#]$ $110d110DC$
18. $D \underset{i}{\rightarrow} d$ $[kjii\#]$ $110d110dC$

19. $C \underset{k}{\rightarrow} d\ [kjii\#]\ 110d110dRC$
20. $R \underset{\overline{k}}{\rightarrow} RD\ [jii\#]\ 110d110dRDC$
21. $R \underset{\overline{j}}{\rightarrow} RB\ [ii\#]\ 110d110dRBDC$
22. $R \underset{\overline{i}}{\rightarrow} RA\ [i\#]\ 110d110dRABDC$
23. $R \underset{\overline{i}}{\rightarrow} RA\ [\#]\ 110d110dRAABDC$
24. $R \underset{[\#]}{\rightarrow} \varepsilon\ [\#]\ 110d110dAABDC$
25. $A \underset{i}{\rightarrow} 1\ [i\#]\ 110d110d1ABDC$
26. $A \underset{i}{\rightarrow} 1\ [ii\#]\ 110d110d11BDC$
27. $B \underset{i}{\rightarrow} 0\ [jii\#]\ 110d110d110DC$
28. $D \underset{i}{\rightarrow} d\ [kjii\#]\ 110d110d110dC$
29. $C \underset{\varepsilon}{\rightarrow} L\ [kjii\#]\ 110d110d110dL$
30. $L \underset{\overline{k}}{\rightarrow} L\ [jii\#]\ 110d110d110dL$
31. $L \underset{\overline{j}}{\rightarrow} L\ [ii\#]\ 110d110d110dL$
32. $L \underset{\overline{i}}{\rightarrow} L\ [i\#]\ 110d110d110dL$
33. $L \underset{\overline{i}}{\rightarrow} L\ [\#]\ 110d110d110dL$
34. $L \underset{[\#]}{\rightarrow} \varepsilon\ [\#]\ 110d110d110d$

Por tanto existe una secuencia de derivaciones desde S hasta la cadena $110d110d110d$, por lo que $G_{0,1}$ genera la cadena $110d110d110d$.

Como se mencionó anteriormente las GIG son cerradas bajo transducción finita, por lo que $G_{0,1}$ permite describir L_{S-SAT} usando el transductor T_{SAT}, el único inconveniente para este análisis es que luego de aplicar T_{SAT} a la gramática, se obtiene un gramática ambigua y en estos casos el problema de la palabra para las GIG no es polinomial [6].

Capítulo 5

Estrategia para la solución del SAT usando gramáticas de concatenación de rango

En este capítulo se abordan las estrategias presentadas en los 2 capítulos anteriores (resolver el SAT usando el problema del vacío y usando el problema de la palabra) utilizando gramáticas de concatenación de rango. Además se obtiene un resultado que permite demostrar que las RCG reconocen todos los problemas que pertenecen a la clase NP y se deja como problema abierto obtener una RCG que permita reconocer todas las instancias de SAT que son solubles en tiempo polinomial.

Para ello primeramente se muestra como reconocer $L_{0,1}$ con una RCG, el cual es la base para las RCG que permiten resolver el SAT usando el problema del vacío y el problema de la palabra, las cuales se presentan en próximas secciones.

5.1. $L_{0,1}$ como lenguaje de concatenación de rango

En esta sección se presenta una RCG que reconoce el lenguaje $L_{0,1} = \{wd\}^+$ donde $w \in \{0,1\}^*$, el cual sirve para la asignación de valores a las variables de un SAT. La gramática empleada se basa reconocer primeramente una cadena w, luego un caracter d y después comprobar que las siguientes cadenas sean iguales a w seguidas del caracter d.

Para reconocer $L_{0,1}$ se define la gramática $G_{0,1}$ como sigue:

$$G_{0,1} = (N, T, V, P, S),$$

donde:

- $N = \{S, A, B, C, Eq\}$

- $T = \{0, 1, d\}$.
- $V = \{X, Y, P\}$.
- El conjunto de cláusulas P es el siguiente:
  1. $S(X) \to A(X)$
  2. $A(XdY) \to B(Y, X)C(X)$
  3. $B(XdY, P) \to B(Y, P)C(X)Eq(X, P)$
  4. $B(\varepsilon, P) \to \varepsilon$
  5. $C(0X) \to C(X)$
  6. $C(1X) \to C(X)$
  7. $C(\varepsilon) \to \varepsilon$
- El **símbolo inicial** es S.

El predicado Eq se define en [5] y comprueba que dos cadenas sobre un alfabeto sean iguales. Por otro lado el predicado B se encarga de definir la sustitución en rango de la próxima cadena de 0 y 1 y comprobar que este sea igual al patrón inicial. De esta manera se pueden reconocer cadenas que pertenezcan al lenguaje $L_{0,1}$.

A continuación se presenta un ejemplo de como se reconoce la cadena $101d101d101d$ en $G_{0,1}$.

5.1.1. Ejemplo de reconocimiento de $G_{0,1}$

En esta sección se describen las derivaciones que permiten reconocer la cadena $101d101d101d$. Las derivaciones de la gramática son las siguientes (después de cada derivación se especifica la cláusula usada para la derivación y los rangos asociados a las variables):

1. $S(101d101d101d) \to A(101d101d101d)$: c-1, $X = 101d101d101d$
2. $A(101d101d101d) \to B(101d101d, 101)C(101)$: c-2, $X = 101$ $Y = 101d101d$
3. $B(101d101d, 101) \to B(101d, 101)C(101)Eq(101, 101)$: c-3, $X = 101$ $Y = 101d$ $P = 101$
4. $B(101d, 101) \to B(\varepsilon, 101)C(101)Eq(101, 101)$: c-3, $X = 101$ $Y = \varepsilon$ $P = 101$
5. $B(\varepsilon, 101) \to \varepsilon$: c-4, $P = 101$
6. $C(101) \to C(01)$: c-6, $X = 01$

7. $C(01) \to C(1)$: c-5, $X = 1$
8. $C(1) \to C(\varepsilon)$: c-6 $X = \varepsilon$
9. $C(\varepsilon) \to \varepsilon$: c-7

Como todos los predicados derivan en la cadena vacía entonces $101d101d101d$ es reconocida por $G_{0,1}$.

En las próximas 2 secciones se presentan 2 RCG que siguen la idea descrita en los capítulos 3 y 4 respectivamente.

5.2. Solución del SAT usando el problema del vacío

En esta sección se presentan 2 enfoques, el primero usa el lenguaje $L_m^n = \{w^m \,|\, w \in \{0,1\} \wedge |w| = n\}$ para asignar valores a las instancias de las variables de una fórmula booleana y el segundo busca obtener una gramática que describa el orden de las variables en una fórmula booleana.

Estos 2 enfoques siguen la línea de lo expuesto en el capítulo 3: asumir que todas las variables en la fórmula son distintas, construir un autómata finito que reconozca cadenas de 0 y 1 hagan verdadera esa fórmula (asumiendo que todas las variables son distintas), y por último intersectar ese lenguaje con algún formalismo que garantice que todas las instancias de la misma variable tenga el mismo valor (en ambos enfoques se emplea una RCG).

En la próxima sección se presentan una RCG que reconoce el lenguaje L_m^n.

5.2.1. L_m^n y $L_{0,1}$

En esta sección se presenta una RCG que permite reconocer el lenguaje L_m^n definido en [13].

Según las definiciones planteadas en los capítulos 3 y 4,

$$L_m^n = \{w^m \,|\, w \in \{0,1\}^* \wedge |w| = n\},$$

y

$$L_{0,1} = \{wd \,|\, w \in \{0,1\}^+\},$$

por tanto para un n y un m específico se cumple que $L_m^n \subset L_{0,1}$.

Luego también es posible reconocer el lenguaje L_m^n mediante una RCG haciendo modificaciones en $G_{0,1}$. Para ello se define la gramática G_m^n como sigue:

$$G_m^n = (N, T, V, P, S),$$

donde:

- $N = \{S, A, B_0, \ldots, B_m, C_1, \ldots, C_n, Eq\}$
- $T = \{0, 1, d\}$.
- $V = \{X, Y, P\}$.
- El **símbolo inicial** es S.

A continuación se desglosa el conjunto de cláusulas P en varias fases agrupando las cláusulas por funcionalidad:

- **Primera fase:** Representa las cláusulas iniciales de la gramática:
  1. $S(X) \to A(X)$
  2. $A(YdX) \to B_0(X, Y)C_1(X)$
- **Segunda fase:** Los predicados $B_i \forall i : 0 \leq i \leq m$ se encargan de definir las transiciones en la gramática que permiten contar el número de cadenas $w \in \{0, 1\}^*$ que existen en la cadena original, verificar que este número es m y comprobar que las cadenas w reconocidas sean iguales:
  3. $B_0(XdY, P) \to B_1(Y, P)C_1(X)Eq(X, P)$
  4. $B_1(XdY, P) \to B_2(Y, P)C_1(X)Eq(X, P)$

     $\vdots$

  5. $B_{m-1}(XdY, P) \to B_m(Y, P)C_1(X)Eq(X, P)$
  6. $B_m(\varepsilon, Y) \to \varepsilon$
- **Tercera fase:** Los predicados $C_i \forall i : 1 \leq i \leq n$ se encargan de verificar que las cadenas w reconocidas en la fase anterior solo estén conformadas por los caracteres 0 ó 1 y tengan exactamente longitud n:
  7. $C_1(0X) \to C_2(X)$
  8. $C_1(1X) \to C_2(X)$
  9. $C_2(0X) \to C_3(X)$

10. $C_2(1X) \to C_3(X)$

$\vdots$

11. $C_{n-1}(0X) \to C_n(X)$
12. $C_{n-1}(1X) \to C_n(X)$
13. $C_n(\varepsilon) \to \varepsilon$

Observe que las modificaciones que se hicieron a $G_{0,1}$, fueron añadir nuevos predicados y transiciones que permitan contar la cantidad de cadenas w reconocidas y la longitud de dichas cadenas. Además la nueva gramática obtenida tiene una cantidad de cláusulas en el orden $O(n+m)$.

Siguiendo la idea de [13] ahora solo resta interceptar G_m^n con el autómata booleano correspondiente a dicha fórmula y comprobar si el lenguaje generado es no vacío. La intercepción de una RCG con un autómata finito se expone en secciones posteriores.

A continuación se presenta una generalización de los trabajos [1] y [12], presentados en el capítulo 3, los cuales se basan en obtener un formalismo que permita describir el orden de las variables de instancias específicas de un SAT.

5.2.2. Orden de las instancias de las variables de un SAT

En [1] y en [12] se proponen 2 formalismos para reconocer el orden de las variables de instancias específicas del SAT que permiten dar una solución en tiempo polinomial. En esta sección se presenta un algoritmo para generar una RCG que permita reconocer el orden de las instancias de las variables de cualquier SAT.

Sea una fórmula booleana:

$$F = L_1\, S_1\, L_2\, S_2 \ldots S_{m-1}\, L_m,$$

donde los $L_i \forall i : 1 \leq i \leq m$ son literales que representan una variable o su negación (dos literales L_x y L_y pueden representar la misma variable) y los $S_i \forall i : 1 \leq i \leq m-1$ son los operadores booleanos de disyunción y conjunción según corresponda. Además se define la función `equals`(L_x, L_y) que recibe dos literales de F y devuelve verdadero o falso en dependencia de si los literales corresponden a la misma variable o no.

Ahora dada un fórmula booleana F, con m literales y n variables se define la gramática G_{ord} como sigue:

$$G_{ord} = (N, T, V, P, S),$$

donde:

- $N = \{S, A, B_1, B_2, \ldots, B_n, C_1, C_2, \ldots, C_n\}$

- $T = \{0, 1, d\}$.
- $V = \{X_1, X_2, \ldots, X_m, Y\}$.
- El conjunto de cláusulas P es el siguiente:
  1. $S(X) \to A(X)$
  2. $A(X_1 X_2 \ldots X_m) \to B_1(X_{\alpha_{1,1}} \ldots X_{\alpha_{1,\beta_1}}) B_2(X_{\alpha_{2,1}} \ldots X_{\alpha_{2,\beta_2}}) \ldots B_n(X_{\alpha_{n,1}} \ldots X_{\alpha_{n,\beta_n}})$
  3. $B_i(1Y) \to Eq(Y, \underbrace{111\ldots1}_{\beta_i}) \forall i : 1 \leq i \leq n$
  4. $B_i(0Y) \to Eq(Y, \underbrace{000\ldots0}_{\beta_i}) \forall i : 1 \leq i \leq n$
- El **símbolo inicial** es S.

Cada una de las variables $X_i \forall 1 \leq i \leq n$ en G_{ord} corresponde al literal L_i de F. Además se cumple que el conjunto $\{\alpha_{1,1}, \ldots, \alpha_{1,\beta_1}, \alpha_{2,1}, \ldots, \alpha_{2,\beta_2}, \ldots, \alpha_{n,1}, \ldots, \alpha_{n,\beta_n}\}$ es una permutación de los los números del 1 al m y para todos los conjuntos $S_i = \{\alpha_{i,1}, \ldots, \alpha_{i,\beta_i}\} \forall i : 1 \leq i \leq n$ se cumple $\texttt{equals}(L_x, L_y)$ es verdadera para todos los pares x, y tales que $x \neq y \wedge x, y \in S_i$.

Observe que en la construcción de G_{ord} a cada variable se le asigna un literal de F, lo que significa para una cadena reconocida por G_{ord} cada caracter asociado a una variable X_i representa el valor de verdad para la variable asociada a L_i en F. El funcionamiento de la gramática se basa en reconocer los rangos de la cadena de entrada que representa la asignación a los valores de las instancias de las variables en F y a cada rango se le asocia una variable de G_{ord}. Luego las variables que en G_{ord} que corresponden a una misma instancia de una variable en F se agrupan en un solo predicado comprobando que todas tengan el mismo valor de verdad.

Siguiendo la estrategia de [1] y [12] solo resta interceptar la gramática generada con el autómata booleano correspondiente a F y comprobar si el lenguaje generado es no vacío. En la próxima sección se aborda el problema de interceptar una RCG con un autómata finito.

5.2.3. Problema del vacío para la intercepción de una RCG con un autómata finito

Primeramente como se mencionó en el capítulo 1 toda CFG tiene una RCG equivalente asociada y a su vez todo autómata finito tiene una CFG equivalente asociada [10], por lo que dado un autómata finito se puede construir una RCG que sea equivalente. Entonces la intercepción de una RCG y un autómata finito puede ser descrita

mediante otra RCG, el problema aquí es como se mencionó anteriormente el problema del vacío para una RCG es indecidible. Por tanto este enfoque no es valido para analizar el problema descrito en esta sección.

En [2] se menciona que el problema del vacío para la intersección de un lenguaje regular y una RCG es un problema NP-Completo y se demuestra con una reducción al 3-SAT. En el caso que ocupa el estudio de este trabajo se puede hacer una reducción parecida tomando como referencia los resultados de las 2 secciones anteriores que permiten reducir este problema al SAT.

En la literatura consultada no se encontró un algoritmo que permitiera resolver este problema, por tanto encontrar un algoritmo específico para este problema puede constituir una vía diferente para resolver el SAT.

En siguiente sección se presenta otra vía de solución para el SAT que usa el problema de la palabra para las RCG.

5.3. Solución del SAT usando el problema de la palabra

En la presente sección se propone utilizar gramáticas de concatenación de rango para definir el enfoque descrito en el capítulo 4, el cual se basa en buscar un formalismo que reconozca las fórmulas booleanas satisfacibles.

5.3.1. L_{S-SAT} como lenguaje de concatenación de rango

Como se mostró en una sección anterior el lenguaje $L_{0,1}$ puede ser reconocido mediante una RCG. Entonces siguiendo con la idea abordada en las secciones del capítulo 4, que busca definir L_{S-SAT} mediante una transducción finita, se puede tratar de definir $L_{0,1}$ para la transducción finita mediante una RCG. El problema radica en que las RCG no son cerradas bajo transducción finita como se mencionó en el capítulo 1. Por ello no se puede asegurar que al aplicarle el transductor T_{SAT} a la RCG que reconoce el lenguaje $L_{0,1}$ se obtenga una RCG, por lo cual no se puede realizar el análisis del problema de la palabra como mismo se hizo con las gramáticas de índice global en el capítulo 4.

Ahora se pudiera profundizar en qué tipo de formalismo se obtiene al aplicar el transductor T_{SAT} sobre la RCG que reconoce $L_{0,1}$ y realizar un análisis de la complejidad del problema de la palabra para dicho formalismo. El cual automáticamente se demuestra que es NP-Duro, porque tiene una reducción directa al SAT. Otro aspecto a considerar sería investigar qué propiedades de las RCG limitan que estas no sean cerradas bajo transducción finita y dado esas propiedades identificadas comprobar si todavía es posible reconocer el lenguaje $L_{0,1}$.

A continuación se presenta otro enfoque distinto para generar L_{S-SAT} que no emplea la transducción finita, la cual se basa en una RCG que reconozca los problemas SAT satisfacibles.

5.3.2. Otro enfoque para generar L_{S-SAT}

En esta sección se presenta una RCG que es capaz de reconocer las fórmulas booleanas satisfacibles.

La idea detrás de esta gramática es la misma vista en el capítulo 4 cuando se definió L_{S-SAT} mediante una transducción finita: obtener un formalismo que permita reconocer si un SAT es satisfacible y para comprobar si un SAT es satisfacibles entonces solo es necesario analizar el problema de la palabra. En la definición de L_{S-SAT} mediante una transducción finita se genera mediante el lenguaje $L_{0,1}$ todas las posibles interpretaciones de cualquier fórmula booleana y luego el transductor T_{S-SAT} genera todas las posibles fórmulas booleanas satisfacibles según las interpretaciones generadas por $L_{0,1}$. Aquí en cambio se presenta un RCG que reconoce las fórmulas booleanas satisfacibles, la cual se basa en generar mediante el reconocimiento de la cadena original todas las posibles interpretaciones de la fórmula booleana que satisfacen la primera cláusula y luego comprobar si dicha intercepción satisface el resto de las cláusulas.

Para reconocer L_{S-SAT} define la siguiente RCG:

$$G_{S-SAT} = (N, T, V, P, S),$$

donde:

- $N = \{S, A, B, C, P, N, Cp, Cn\}$
- $T = \{a, b, c, d\}$.
- $V = \{X, Y, X_1, X_2\}$.
- El **símbolo inicial** es S.

A continuación se desglosa el conjunto de **cláusulas** P en varias en 4 fases agrupando las cláusulas por funcionalidad. La primera representa la derivación inicial de la gramática. La segunda se encarga de generar todas las posibles interpretaciones de las variables que satisfacen la primera cláusula. La tercera comprueba que la interpretación definida en la fase anterior sea satisfacible para el resto de las cláusulas. Por último la cuarta fase define el algoritmo para determinar si una intercepción satisface una cláusula dada. Finalmente si para una cadena que representa una fórmula booleana existe una secuencia de derivaciones desde el predicado inicial pasando por cada fase hasta la cadena vacía entonces la cadena se reconoce:

- **Primera fase:** Representa la cláusula de derivación inicial de la gramática:

  1. $S(X) \rightarrow A(X)$

- **Segunda fase:** El siguiente conjunto de cláusulas genera la cadena de 0 y 1 que que da valores a las variables de la fórmula booleana:

  2. $A(aX) \rightarrow P(X,1)$
  3. $A(aX) \rightarrow N(X,0)$
  4. $A(bX) \rightarrow N(X,1)$
  5. $A(bX) \rightarrow P(X,0)$
  6. $A(cX) \rightarrow N(X,1)$
  7. $A(cX) \rightarrow N(X,0)$
  8. $P(aX,Y) \rightarrow P(X,Y1)$
  9. $P(aX,Y) \rightarrow P(X,Y0)$
  10. $P(bX,Y) \rightarrow P(X,Y1)$
  11. $P(bX,Y) \rightarrow P(X,Y0)$
  12. $P(cX,Y) \rightarrow P(X,Y1)$
  13. $P(cX,Y) \rightarrow P(X,Y0)$
  14. $P(dX,Y) \rightarrow B(X,Y)$
  15. $N(aX,Y) \rightarrow P(X,Y1)$
  16. $N(aX,Y) \rightarrow N(X,Y0)$
  17. $N(bX,Y) \rightarrow N(X,Y1)$
  18. $N(bX,Y) \rightarrow P(X,Y0)$
  19. $N(cX,Y) \rightarrow N(X,Y1)$
  20. $N(cX,Y) \rightarrow N(X,Y0)$

  El predicado A representa el predicado por donde inician las derivaciones de esta fase, de este se deriva a los predicados P (representa que la cláusula de la fórmula booleana se encuentra en un estado de verdad positivo) y N (representa que la cláusula de la fórmula booleana se encuentra en un estado de verdad negativo) en dependencia del valor asignado. El predicado P deriva hacia sí mismo independientemente del símbolo, exceptuando el símbolo d, caso en el que se procede a la siguiente fase. El funcionamiento de esta fase es prácticamente el mismo que el del transductor T_{SAT}: a partir de un estado de una variable en la fórmula booleana y una asignación que se le haga a la misma pasar a un estado positivo o negativo que representa el valor de verdad actual de la cláusula.

- **Tercera fase:** El siguiente conjunto de cláusulas se encarga de un mecanismo de iteración que le permite a la gramática reconocer si la asignación realizada en la fase anterior es válida para las restantes cláusulas de la fórmula booleana.

  21. $B(X_1 dX_2,Y) \rightarrow C(X_1,Y)B(X_2,Y)$
  22. $B(\varepsilon,Y) \rightarrow \varepsilon$

  El predicado B permite realizar la iteración sobre las cláusulas restantes mientras que el predicado C comprueba que la cláusula de la fórmula booleana actual sea satisfacible, comportamiento que se encuentra definido en la cuarta fase.

- **Cuarta fase:** Solo resta definir el comportamiento de C, que recibe una cláusula y una intercepción de las variables y comprueba que dicha intercepción sea satisfacible para la cláusula analizada:

  23. $C(X,Y) \rightarrow Cn(X,Y)$
  24. $Cn(aX,1Y) \rightarrow Cp(X,Y)$
  25. $Cn(aX,0Y) \rightarrow Cn(X,Y)$
  26. $Cn(bX,1Y) \rightarrow Cn(X,Y)$
  27. $Cn(bX,0Y) \rightarrow Cp(X,Y)$
  28. $Cn(cX,1Y) \rightarrow Cn(X,Y)$
  29. $Cn(cX,0Y) \rightarrow Cn(X,Y)$
  30. $Cp(aX,1Y) \rightarrow Cp(X,Y)$
  31. $Cp(aX,0Y) \rightarrow Cp(X,Y)$
  32. $Cp(bX,1Y) \rightarrow Cp(X,Y)$
  33. $Cp(bX,0Y) \rightarrow Cp(X,Y)$
  34. $Cp(cX,1Y) \rightarrow Cp(X,Y)$
  35. $Cp(cX,0Y) \rightarrow Cp(X,Y)$
  36. $Cp(\varepsilon,\varepsilon) \rightarrow \varepsilon$

  Observe que este funcionamiento es exactamente igual al de la primera fase con un predicado que representa un estado positivo (Cp) y un predicado que representa un estado positivo (Cn) pero esta vez no se genera la cadena sino que se comprueba con un patrón, el cual se predefine en la segunda fase y pasa a las siguientes mediante las derivaciones de la gramática.

A continuación se demuestra que el lenguaje reconocido por G_{S-SAT} es exactamente igual al lenguaje que representa todas las fórmulas booleanas satisfacibles.

Demostración de la gramática G_{S-SAT}

La idea para la demostración de que el lenguaje reconocido por G_{S-SAT} es exactamente igual al lenguaje que representa todas las fórmulas booleanas satisfacibles se basa en demostrar el correcto funcionamiento de las 4 fases de la gramática. Primeramente demostrar que el predicado C que pertenece a la cuarta fase reconoce los argumentos X y Y si y solo si Y satisface la cláusula de la fórmula booleana asociada a X. Posteriormente demostrar que el predicado B asociado a la tercera fase reconoce los argumentos X y Y si y solo si Y satisface todas las cláusulas asociadas a X, funcionamiento presente en la segunda cláusula. Luego demostrar que el conjunto de cadenas Q formado por todas las cadenas Y tales que existe una secuencia de derivaciones desde del predicado $A(X_1)$ hasta $B(X_2,Y)$ es exactamente igual a al conjunto de todas las interpretaciones que hacen verdadera la primera cláusula de X_1, funcionamiento presente en la primera cláusula. Por último demostrar que el conjunto de cadenas X que son reconocidas por el predicado inicial S es exactamente igual al conjunto de la representación de todas las fórmulas booleanas satisfacibles.

A continuación se presenta la demostración para el funcionamiento de cada fase:

- **Cuarta fase:** Dado el predicado $C(X,Y)$.

  Para demostrar el funcionamiento de la cuarta fase primeramente suponga que la fórmula booleana asociada a X, la cual se denomina F_x es satisfacible por Y, entonces se debe demostrar que existe una secuencia de derivaciones desde $C(X,Y)$ hasta la cadena vacía. Como Y satisface a F_x debe existir una variable sin negar a la cual se le da como valor un 1 o un variable negada con valor 0, sin pérdida de la generalidad se dice que la primera variable que cumple esto es la i-ésima variable. Por tanto pueden darse 2 casos, el i-ésimo caracter de X es a y el i-ésimo caracter de Y es 1 o el i-ésimo caracter de X es b y el i-ésimo caracter de Y es 0. Del predicado C automáticamente se deriva al predicado Cn, la única derivación de la gramática donde se deriva del predica del predicado Cn a Cp es precisamente la combinación de una a y un 1 o de una b y un 0 y como Y satisface F_x esta combinación existe. Por último Cp siempre deriva en sí mismo o en la cadena vacía por lo tanto queda demostrado que existe una secuencia de derivaciones desde $C(X,Y)$ hasta la cadena vacía.

  La demostración del funcionamiento de la cuarta fase la completa el hecho de que si $C(X,Y)$ es reconocido entonces Y satisface a X. Por la estructura de la gramática si existe una secuencia de derivaciones desde $C(X,Y)$ hasta la cadena vacía entonces hay una derivación desde Cn hacia Cp y esta derivación solo es posible por una combinación de una a y un 1 o de una b y un 0, por lo tanto una de estas combinaciones existe. Luego existe en F_x una variable sin negar con valor 1 o una variable negada con valor 0 lo cual implica que Y satisface F_x.

- **Tercera fase:** Dado el predicado $B(X,Y)$.

  Para demostrar el funcionamiento de la tercera fase se hará una inducción sobre la cantidad de cláusulas n de la fórmula booleana asociada a X. Para $n=1$ se cumple que los rangos asociados a las variables X_1 y X_2 son X sin su último caracter y la cadena vacía respectivamente, por tanto $B(X,Y)$ se reconoce por la gramática si y solo si $C(X_1,Y)$ se reconoce y esto solo es posible si Y satisface a X_1, por lo que se demuestra el caso base. Se asume para $n=k$ y se demuestra para $k+1$, en todas las posibles sustituciones en rango de X_1 y X_2, $C(X_1,Y)$ solo se reconoce si $|X_1|=|Y|$, entonces el caso de sustitución en rango que ocupa a la demostración es si $|X_1|=|Y|$. Luego Y satisface todas las cláusulas de X si y solo si satisface X_1 y X_2 y precisamente $B(X,Y)$ se reconoce si y solo si se reconoce $C(X_1,Y)$ y $B(X_2,Y)$, $C(X_1,Y)$ se demuestra por el funcionamiento de la cuarta fase y $B(X_2,Y)$ se demuestra por hipótesis de inducción.

- **Segunda fase:** Dado el predicado $A(X)$.

Para la demostración del funcionamiento de la segunda fase se utilizan 2 conjuntos W representa el conjunto de todas las cadenas que satisfacen la primera cláusula de la fórmula asociada a X (F_{1x}) y Q que representa el conjunto de todas las cadenas Y tales que existe una secuencia de derivaciones desde $A(X)$ hasta $B(Z_x,Y)$, donde Z_x esta conformada por todas las cláusulas de X menos la primera. Dadas estas definiciones es necesario demostrar que $W = Q$, para ello se debe demostrar que $W \subseteq Q \wedge Q \subseteq W$.

Para demostrar que $W \subseteq Q$, sea w una cadena tal que $w \in W$, es decir, w satisface la primera F_{1x}. Por tanto en F_{1x} existe una variable sin negar con valor 1 en w o existe una variable negada con valor 0 en w, lo que representa una combinación de una a y un 1 o de una b y un 0 en una de las derivaciones de la segunda fase. Por la estructura de la gramática del predicado A solo hay derivaciones a P con una de estas 2 combinaciones, el resto son hacia el predicado N y del predicado N solo hay derivaciones a P con una de las combinaciones anteriores. Por tanto como existe una combinación de una a y un 1 o de una b y un 0, existe una secuencia de derivaciones que lleva del predicado A al predicado P pasando por N o sin pasar por N. Como el predicado P solo tiene derivaciones hacia sí mismo o hacia $B(Z_x,Y)$ por lo tanto se cumple que $w \in Q$.

Para demostrar que $Q \subseteq W$, sea q una cadena tal que $q \in Q$, es decir, existe una secuencia de derivaciones desde $A(X)$ a $B(X,Y)$ con $q = Y$. Por la estructura de la gramática solo se puede derivar al predicado B desde el predicado P y a su vez a este predicado solo se puede derivar mediante una combinación de una a y un 1 o de una b y un 0 en la gramática. Por tanto F_{1x} tiene una variable sin negar con valor 1 en q o una variable negada con valor 0 en q. Entonces se cumple que q satisface a F_{1x} por lo que $q \in W$. Por tanto queda demostrado que $Q = W$.

- **Primera fase:** Dado el predicado $S(X)$.

  Para demostrar que el lenguaje reconocido por G_{S-SAT} es exactamente igual al lenguaje que representa todas las fórmulas booleanas satisfacibles se define el lenguaje $L_{G_{S-SAT}}$ que representa el lenguaje de todas las cadenas reconocidas por G_{S-SAT}, es necesario demostrar que $L_{S-SAT} = L_{G_{S-SAT}}$.

  Sea una fórmula booleana satisfacible F y sea X su representación como cadena, entonces existe una cadena binaria w que satisface F. Como w satisface F entones w pertenece al conjunto de cadenas que satisfacen a la primera cláusula de F, entonces existe una secuencia de derivaciones desde el predicado $S(X)$ hasta $B(Z_x,w)$ y como w satisface todas las cláusulas de F entonces $B(Z_x,w)$ deriva en la cadena vacía por lo que X es reconocida por G_{S-SAT}, lo cual

implica que $L_{S-SAT} \subseteq L_{G_{S-SAT}}$.

Sea una cadena X reconocida por G_{S-SAT} y sea F la fórmula booleana asociada, entonces existe una cadena binara q tal que existe una secuencia de derivaciones desde $A(X)$ a $B(Z_x, q)$ y de $B(Z_x, q)$ a la cadena vacía, por tanto q satisface a la primera cláusula de F y a las restantes también, luego q satisface a F, por lo que F es satisfacible, lo cual implica que $L_{G_{S-SAT}} \subseteq L_{S-SAT}$. Por tanto se demuestra que $L_{G_{S-SAT}} = L_{S-SAT}$.

En la siguiente sección se presentan un ejemplo del funcionamiento de G_{S-SAT}.

Ejemplo de reconocimiento de G_{S-SAT}

En esta sección se presentan 2 ejemplos del funcionamiento de G_{S-SAT} en el primero se muestra como se reconoce la cadena asociada a la fórmula booleana $x_1 \vee x_2 \wedge x_1 \wedge \neg x_2$ y en el segundo se muestra como no se reconoce la fórmula booleana asociada a $x_1 \wedge \neg x_1$.

La cadena asociada a $x_1 \vee x_2 \wedge x_1 \wedge \neg x_2$ es *aadacdcbd*, la secuencia de derivaciones asociada a esta cadena en G_{S-SAT} es la siguiente (después de cada derivación se especifica la cláusula usada para la derivación y los rangos asociados a las variables):

1. $S(aadacdcbd) \rightarrow A(aadacdcbd)$: c-1, $X = aadacdcbd$
2. $A(aadacdcbd) \rightarrow P(adacdcbd, 1)$: c-2, $X = adacdcbd$
3. $P(adacdcbd, 1) \rightarrow P(dacdcbd, 10)$: c-9, $X = dacdcbd$ $Y = 1$
4. $P(dacdcbd, 10) \rightarrow B(acdcbd, 10)$: c-14, $X = acdcbd$ $Y = 10$
5. $B(acdcbd, 10) \rightarrow C(ac, 10)B(cbd, 10)$: c-21, $X_1 = ac$ $X_2 = cbd$ $Y = 10$
6. $B(cbd, 10) \rightarrow C(cb, 10)B(\varepsilon, 10)$: c-21, $X_1 = cb$ $X_2 = \varepsilon$ $Y = 10$
7. $B(\varepsilon, 10) \rightarrow \varepsilon$: c-22, $Y = 10$
8. $C(ac, 10) \rightarrow Cn(ac, 10)$: c-23, $X = ac$ $Y = 10$
9. $Cn(ac, 10) \rightarrow Cp(c, 0)$: c-24, $X = c$ $Y = 0$
10. $Cp(c, 0) \rightarrow Cp(\varepsilon, \varepsilon)$: c-35, $X = \varepsilon$ $Y = \varepsilon$
11. $Cp(\varepsilon, \varepsilon) \rightarrow \varepsilon$: c-36
12. $C(cb, 10) \rightarrow Cn(cb, 10)$: c-23, $X = cb$ $Y = 10$
13. $Cn(cb, 10) \rightarrow Cn(b, 0)$: c-28, $X = b$ $Y = 0$

14. $Cn(b,0) \rightarrow Cp(\varepsilon,\varepsilon)$: c-27, $X = \varepsilon\ Y = \varepsilon$
15. $Cp(\varepsilon,\varepsilon) \rightarrow \varepsilon$: c-36

Como todos los predicados derivan en la cadena vacía entonces *aadacdcbd* es reconocida por G_{S-SAT}, lo cual coincide con el hecho de que $x_1 \vee x_2 \wedge x_1 \wedge \neg x_2$, para la asignación de valores $x_1 = 1$ y $x_2 = 0$.

Ahora se presenta un caso asociado a una fórmula que no es satisfacible y por tanto la cadena asociada a dicha fórmula no es reconocida por G_{S-SAT}. La cadena asociada a $x_1 \wedge \neg x_1$ es *adbd*, la secuencia de derivaciones asociada a esta cadena en G_{S-SAT} es la siguiente (después de cada derivación se especifica la cláusula usada para la derivación y los rangos asociados a las variables):

1. $S(adbd) \rightarrow A(adbd)$: c-1, $X = abdbd$
2. $A(adbd) \rightarrow P(dbd,1)$: c-2, $X = dbd$
3. $A(adbd) \rightarrow N(dbd,0)$: c-3, $X = dbd$
4. $P(dbd,1) \rightarrow B(bd,1)$: c-14, $X = bd\ Y = 1$
5. $B(bd,1) \rightarrow C(b,1)B(\varepsilon,1)$: c-21, $X_1 = b\ X_2 = \varepsilon\ Y = 1$
6. $C(b,1) \rightarrow Cn(b,1)$: c-23, $X = b\ Y = 1$
7. $Cn(b,1) \rightarrow Cn(\varepsilon,\varepsilon)$: c-26, $X = \varepsilon\ Y = \varepsilon$

El predicado $C(b,1)$ en la quinta derivación no deriva en la cadena vacía por tanto, después de realizar todas las posibles derivaciones y las sustituciones en rango G_{S-SAT} no reconoce la cadena *adbd*. Esto coincide con el hecho de que $x_1 \wedge \neg x_1$ para ninguna asignación de valores a sus variables.

Como G_{S-SAT} reconoce las fórmulas booleanas satisfacibles solo se debe analizar el problema de la palabra para determinar si una fórmula es satisfacible, por lo que el siguiente paso es analizar el la complejidad del problema de la palabra para G_{S-SAT}.

Análisis de la complejidad computacional del reconocimiento en G_{S-SAT}

Como se mencionó en el capítulo 2 no todas las RCG tienen un algoritmo de reconocimiento polinomial y G_{S-SAT} es un ejemplo de ello. Observe que en la primera fase se genera la cadena binaria que representa la asignación de valores a las variables booleanas y dicha cadena participa en los predicados de fases posteriores, mediante las derivaciones de la gramática.

Si se analiza el algoritmo de reconocimiento descrito en [5] un factor en la complejidad del algoritmo de reconocimiento es la cantidad de rangos posibles para una

cadena que se reconoce por un predicado. En este caso la cadena que representa los valores de las variables de la fórmula booleana puede tomar 2^n valores distintos, donde n es la cantidad de variables en la fórmula booleana, ya que dicha cadena se genera durante la primera fase donde la gramática es ambigua y en cada derivación hay decisiones que generan valores distintos. Entonces la cantidad de rangos sería $n^2 2^n$, pero esta es una cota burda ya que una vez generada la cadena de asignación por la forma de la gramática solo se utiliza un solo rango que se va construyendo bajo demanda.

El resto de las fases de la gramática tienen una complejidad de m^2 donde m es la cantidad de caracteres en la cadena de entrada, por lo que la complejidad total sería $O(2^n m^2)$.

Este es resultado demuestra que no es necesario usar el transductor T_{SAT} para definir el lenguaje L_{S-SAT}, mediante un formalismo de escritura regulada.

En la siguiente sección se analiza una consecuencia directa del resultado de la gramática G_{S-SAT}.

5.4. Clases de problemas que reconocen las RCG

En [4] se menciona que para todo problema en P existe una RCG que lo reconoce en su representación como lenguaje formal. Ahora, como se mostró en la sección anterior con la gramática G_{S-SAT} existe una RCG que permite resolver el SAT, por tanto como el SAT se puede reducir a cualquier problema en NP en una complejidad polinomial, entonces para todo problema en NP también existe una RCG que lo reconoce en su representación como lenguaje formal.

En la próxima sección se presenta un primer acercamiento para resolver las instancias polinomiales del SAT usando gramáticas de concatenación de rango.

5.5. Instancias de SAT polinomiales empleando RCG

En esta sección se presenta una RCG que es capaz de reconocer problemas SAT, satisfacibles que pertenecen al 2-SAT, es decir, problemas SAT donde cada cláusula tiene a lo sumo 2 literales. La idea detrás de esta gramática es obtener una RCG que reconozca cuando la fórmula booleana pertenece al conjunto de fórmulas booleanas de 2-SAT y luego intersectar dicha gramática con G_{S-SAT}. Para ello se define la siguiente RCG:

$$G_{2-SAT} = (N, T, V, P, S),$$

donde:

- $N = \{S, A, A_0, A_1, A_2, A_3\}$

- $T = \{a, b, c, d\}$.
- $V = \{X, Y, X_1, X_2\}$.
- El conjunto de cláusulas P es el siguiente:

1. $S(X) \to A(X)$
2. $A(X_1 dX_2) \to A_0(X_1)A(X_2)$
3. $A(\varepsilon) \to \varepsilon$
4. $A_0(aX) \to A_1(X)$
5. $A_0(bX) \to A_1(X)$
6. $A_0(cX) \to A_0(X)$
7. $A_1(aX) \to A_2(X)$
8. $A_1(bX) \to A_2(X)$
9. $A_1(cX) \to A_1(X)$
10. $A_2(aX) \to A_3(X)$
11. $A_2(bX) \to A_3(X)$
12. $A_2(cX) \to A_2(X)$
13. $A_2(\varepsilon) \to \varepsilon$

- El **símbolo inicial** es S.

El funcionamiento de la gramática anterior es el siguiente: la segunda cláusula permite reconocer todas las cláusulas asociadas a la cadena original. Mientras que las cláusulas de la 4 a la 13 permiten contar la cantidad de a o b en la cláusula (osea la cantidad de literales de cada cláusula), para esto se definen 4 estados: se reconocieron 0 a o b, se reconocieron una a o b, se reconocieron 2 a o b y se reconocen más de 2 a o b, los cuales están representados por los predicados A_0, A_1, A_2 y A_3 respectivamente. Se crean las derivaciones entre los predicados que representan cada estado y se deriva en la cadena vacía desde el predicado A_2 cuando el argumento es la cadena vacía, lo que indica que la cláusula tiene 2 literales.

Como para todo problema en P existe una RCG que lo reconoce en su representación como lenguaje formal, entonces es posible resolver las todas las instancias polinomiales del SAT usando gramáticas de concatenación de rango.

Si se observa el enfoque seguido en la construcción de G_{S-SAT}, en la representación del SAT como cadena se trabaja con una instancia del SAT general por lo que no se tienen en cuenta las propiedades específicas del problema, que en el caso de las instancias polinomiales, es lo que permite que el algoritmo para las misma sea polinomial.

Entonces la gramática que reconoce los problemas 2-SAT satisfacible sería:

$$G_{S-2-SAT} = G_{S-SAT} \cap G_{2-SAT}.$$

Pero $G_{S-2-SAT}$ el problema de la palabra para $G_{S-2-SAT}$ es exponencial y se conoce que para el 2-SAT existe un algoritmo polinomial.

Como para todo problema en P existe una RCG que lo reconoce en su representación como lenguaje formal, entonces es posible resolver el 2-SAT y todas las todas

las instancias polinomiales del SAT usando gramáticas de concatenación de rango. Pero no se ha podido encontrar una RCG que permita reconocer problemas 2-SAT satisfacibles en un tiempo polinomial, esto se prepone como problema abierto ya que puede conducir a entontrar otras instancias de SAT solubles en tiempo polinomial.

Conclusiones

Conclusiones

Recomendaciones

Recomendaciones

Referencias

[1] Alina Fernández Arias. «El problema de la satisfacibilidad booleana libre del contexto». En: (2007) (vid. págs. 27-31, 45, 46).

[2] Eberhard Bertsch y Mark-Jan Nederhof. «On the Complexity of Some Extensions of RCG Parsing». En: *International Workshop/Conference on Parsing Technologies*. 2001. URL: https://api.semanticscholar.org/CorpusID:38424278 (vid. pág. 47).

[3] Pierre Boullier. *A Cubic Time Extension of Context-Free Grammars*. Research Report RR-3611. INRIA, 1999. URL: https://inria.hal.science/inria-00073067 (vid. págs. 24-26).

[4] Pierre Boullier. «Counting with range concatenation grammars». En: *Theor. Comput. Sci.* 293 (feb. de 2003), págs. 391-416. DOI: 10.1016/S0304-3975(01)00353-X (vid. pág. 55).

[5] Pierre Boullier. *Proposal for a Natural Language Processing Syntactic Backbone*. Research Report RR-3342. INRIA, 1998. URL: https://inria.hal.science/inria-00073347 (vid. págs. 21, 22, 24, 26, 42, 54).

[6] José M. Castaño. «Global Index Languages». Tesis doct. The Faculty of the Graduate School of Arts y Sciences Brandeis University, 2004 (vid. págs. 18, 20, 21, 40).

[7] GeeksforGeeks. *2-Satisfiability (2-SAT) Problem*. Accessed: 2025-01-09. n.d. URL: https://www.geeksforgeeks.org/2-satisfiability-2-sat-problem/ (vid. pág. 14).

[8] GeeksforGeeks. *Chomsky Hierarchy in Theory of Computation*. Accessed: 2025-01-09. n.d. URL: https://www.geeksforgeeks.org/chomsky-hierarchy-in-theory-of-computation/ (vid. pág. 5).

[9] GeeksforGeeks. *Finite State Transducer (FSTs) in NLP*. Accessed: 2025-01-09. n.d. URL: https://www.geeksforgeeks.org/finite-state-transducer-fsts-in-nlp/ (vid. pág. 8).

[10] John E. Hopcroft, Rajeev Motwani y Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. 3rd. Addison-Wesley, 2006. ISBN: 9780321455369 (vid. págs. 4, 6, 7, 10-15, 29, 46).

[11] Oscar H. Ibarra. «Simple matrix languages». En: *Information and Control* 17.4 (1970), págs. 359-394. ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(70)80034-1. URL: https://www.sciencedirect.com/science/article/pii/S0019995870800341 (vid. págs. 16, 33).

[12] Manuel Aguilera López. «Problema de la Satisfacibilidad Booleana de Concatenación de Rango Simple». En: (2016) (vid. págs. 30, 45, 46).

[13] José Jorge Rodríguez Salgado. «Gramáticas Matriciales Simples. Primera aproximación para una solución al problema SAT». En: (2019) (vid. págs. 32, 33, 43, 45).