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

En este capítulo se presentan las principales definiciones y conceptos que serán usados en el resto del trabajo, además se analizan los trabajos anteriores que exponen estrategias de solución del problema de la satisfacibilidad booleana mediante formalismos de la teoría de lenguajes.

1.1. Teoría de Lenguajes

En esta sección se presentan los principales conceptos de teoría de lenguajes que sirven de base al contenido de los capítulos y secciones posteriores, ya que en este trabajo se presentan estrategias para la solución del problema de la satisfacibilidad empleando mecanismos de la teoría de lenguajes.

1.1.1. Conceptos básicos

Los conceptos básicos de la teoría de lenguajes formales son alfabeto, cadena y lenguaje. Un alfabeto, denotado como Σ, es un conjunto finito y no vacío de símbolos; una cadena es una sucesión finita de símbolos del alfabeto y un lenguaje es un conjunto de cadenas definido sobre un alfabeto. Por ejemplo, el alfabeto $\Sigma = \{1, 0\}$ está formado por los símbolos 0 y 1, 11 y 101 son cadenas sobre el alfabeto Σ y un ejemplo de lenguaje es el conjunto de cadenas de 0 y 1 que terminan en 1.

Dados los conceptos básicos, a continuación se definen operaciones entre alfabetos y cadenas.

1.1.2. Operaciones con Lenguajes

Como los lenguajes son conjuntos, todas las operaciones sobre conjuntos también se definen para lenguajes: unión, intersección, complemento [7].

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. Si $w = a_1 a_2 \dots a_n$ es una cadena entonces

$$h(w) = h(a_1)h(a_2)\dots h(a_n).$$

Por ejemplo si sobre el alfabeto $\Sigma = \{a, b\}$, se define el homomorfismo h, tal que $h(a) = 0$ y $h(b) = 11$, entonces $h(ab) = 011$.

Estos conceptos son 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 2 problemas que serán usados en el desarrollo de este trabajo, el primero se utiliza en los capítulos 3 y 4 y el segundo es utilizado en [1] y [8] (estos trabajos serán descritos más adelante).

1.1.3. Problemas relacionados con Lenguajes

En esta sección se representa el problema de la palabra y el problema del vacío.

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

Por ejemplo dado el lenguaje $L = \{w \,|\, \text{last}(w) = 0\}$, last$(w)$ determinar si 1100100 pertenece a L.

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 [7].

Problema del vacío: Consiste en determinar si un lenguaje es vacío. Por ejemplo dado el lenguaje determinar si el conjunto de números pares mayores que 5, que sean primos es vacío.

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:

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

donde $\xrightarrow{*}$ 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 los lenguajes en cuatro tipos, según las restricciones en sus reglas de producción y la capacidad expresiva de los lenguajes que generan [**geeksforgeeks_chomsky_hierarchy**].

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**
   - 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 las 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 $L^n_{copy} = \{w^n \,|\, w \in Z^*\}$ y el lenguaje $L_{rev-copy} = \{ww_r \,|\, w \in Z^*\}$, donde w_r es w invertida. Se cumple que L^1_{copy} es un lenguaje regular, mientras que $L_{rev-copy}$ es un lenguaje libre del contexto y por último $L^k_{copy} \forall k \geq 2$ es un lenguaje dependiente del contexto [7]. El lenguaje L_{copy} se usa en los restantes capítulos como base de formalismos que describen el lenguaje de las fórmulas booleanas satisfacibles.

En la próxima sección se presentan los principales conceptos relacionados con los autómatas regulares, los cuales son usados en [1] para describir un mecanismo que

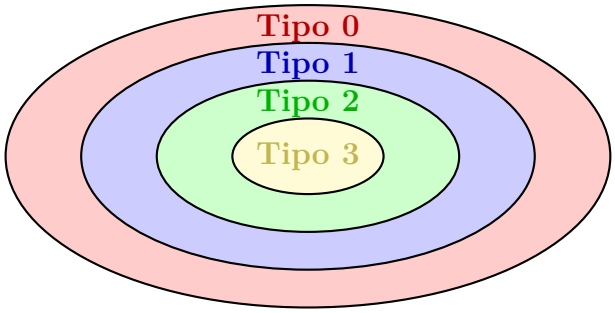

Figura 1.1: Esquema de la Jerarquía de Chomsky

determina si una asignación de valores de las variables satisface una fórmula booleana y en el capítulo 3 se utiliza una extensión de los autómatas regulares que permite para una asignación de valores de las variables generar todas las fórmulas booleanas satisfacibles por dichos valores.

1.2. Autómatas

Un autómata es una máquina abstracta que procesa cadenas de símbolos de un alfabeto finito y determina si una cadena pertenece a un lenguaje [7]. En esta sección se presentan los principales conceptos sobre los autómatas regulares.

1.2.1. Autómata regular

Un autómata regular [7], también conocido como autómata finito, es un modelo matemático que permite reconocer si una cadena pertenece a un lenguaje regular y 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 procede a leer el primer símbolo de la cadena. En cada paso, la función de transición δ determina, con base en el símbolo actual de la cadena, el siguiente estado al que debe pasar el autómata, avanzando al siguiente símbolo. Si, al finalizar la lectura del último símbolo de la cadena, el autómata se encuentra en un estado de aceptación $q \in F$, entonces la cadena se acepta; en caso 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 [7] que el conjunto de lenguajes que reconoce un autómata finito sin transiciones ε (*autómata finito determinista*) es equivalente al conjunto de lenguajes que reconoce un autómata finito con transiciones ε (*autómata finito no determinista*).

A continuación se presenta una extensión de los autómatas finitos, que es usada en el capítulo 3 para dada una asignación de valores de las variables, generar todas las fórmulas booleanas satisfacibles por dichos valores..

1.2.2. Transductor finito

Un transductor finito [5] es un modelo computacional que extiende los autómatas finitos al incluir una salida para cada transición.

Formalmente, un transductor finito es un autómata finito con una función de transición extendida que recibe un símbolo de la cadena de entrada y un estado, y devuelve el estado al cual pasa el transductor y un símbolo. Como resultado de la aceptación de una cadena el transductor genera una cadena de salida formada por todos los símbolos que se obtuvieron como resultado mediante la función de transición en el proceso de reconocimiento.

Un transductor finito se define 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.

Por ejemplo se puede definir el siguiente transductor finito T, el cual reconoce cadenas del alfabeto $\{a, b\}$ que tengan ninguna o más a seguida de una o más b y genere una cadena formada por una cantidad de 0 igual a la cantidad de a de la cadena original seguida de una cantidad de 1 igual a la cantidad de b de la cadena original:

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

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.

Para reconocer *aaaabbbbb*, primero T empieza por el estado q_0 y el primer caracter a, entonces pasa al estado q_0 y genera un 0. Luego, al leer el segundo caracter a, se mantiene en el estado q_0 y genera otro 0. Este proceso se repite hasta el quinto caracter que es b, entonces pasa al estado q_1. Luego, al leer el sexto caracter b, se mantiene en el estado q_1 y genera un 1. Este proceso se repite varias veces hasta que se llega al final de la cadena, por tanto se genera la cadena 000011111.

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

Seguidamente se presenta otra máquina abstracta, la cual representa el modelo de cómputo más general de la teoría de lenguajes y representa la formalización de un Algoritmo.

1.3. Máquina de Turing

Una Máquina de Turing [7] es un modelo abstracto de computación universal introducido por Alan Turing en 1936. Este modelo consiste en los siguientes componentes: una cinta, un cabezal de escritura/lectura, un conjunto de estados y una función de transición.

Una cinta es un medio de almacenamiento infinito dividido en celdas, donde cada celda contiene un símbolo de un alfabeto finito. Un cabezal de lectura/escritura es un dispositivo que puede leer el contenido de una celda, escribir un nuevo símbolo y moverse a la izquierda o derecha. El conjunto de estados es colección finita de estados internos que describen la configuración actual de la máquina. La función de transición es 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.

A continuación de presentan los conceptos de máquina de Turing determinista y máquina de Turing no determinista.

Máquina de Turing determinista (*DTM*): En una Máquina de Turing determinista, para cada estado y cada símbolo que se lee, se define como máximo una transición posible.

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

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: notación asintótica 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 define la notación que será utilizada en el resto del trabajo:

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 $g(n)$.

La notación asintótica permite describir el tiempo de ejecución de 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 $O(n)$, 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, ya que se necesita realizar una cantidad de operaciones proporcional al tamaño del arreglo.

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 [7] 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. En términos de teoría de lenguajes un lenguaje L pertenece a P si existe una Máquina de Turing determinista que reconozca L en un 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 [7].

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 [7].

Problemas en la clase NP-Duro: Un problema pertenece a la clase NP-Duro, si es tan difícil como cualquier otro problema en NP, pero no necesariamente pertenece a NP [7].

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 [7]. Un ejemplo clásico de problema no decidible es el *Problema de la Parada* [7], 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 [7]. 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 [7].

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: el problema de las satisfacibilidad booleana.

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 [7]. El objetivo es determinar si existe una asignación de valores a las variables de una fórmula booleana tal que la expresión sea verdadera.

A continuación se presentan los principales elementos del SAT:

- **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.

- **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.

- **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.

- **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:

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

- **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 [7] y el algoritmo para encontrarla es polinomial, por lo tanto se puede asumir que toda fórmula booleana está en CNF.

En la próxima sección define el problema de la satisfacibilidad booleana.

1.5.1. 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.

1.5.2. SAT como Problema NP-Completo

El SAT es el primer problema demostrado como NP-Completo [7] 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 [7], marcando el inicio de la teoría de la NP-completitud.

1.5.3. 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 [6], pero para el problema 3-SAT no se conoce ningún algoritmo polinomial que permita determinar si una fórmula booleana es satisfacible o no [7].

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 [7].

1.5.4. 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 la siguiente sección se describen 2 trabajos que vinculan el SAT con la teoría de lenguajes.

1.6. Solución de instancias del SAT usando Teoría de Lenguajes

Como parte del estudio del problema SAT, se han desarrollado anteriormente en la Facultad de Matemática y Computación de la Universidad de La Habana 2 trabajos utilizando un enfoque basado en formalismos de la teoría de lenguajes, buscando resolver instancias específicas del SAT, que tienen una solución polinomial.

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. Todo el algoritmo descrito anteriormente tiene un tiempo polinomial en relación con el tamaño de la fórmula booleana.

El autómata finito diseñado en [1], se denominó **autómata booleano**. La idea detrás de este 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].

Posteriormente en [1] se define el concepto del orden libre del contexto: una fórmula booleana tiene un orden libre del contexto 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.

Para terminar con lo expuesto en [1], para determinar si una fórmula booleana con un orden libre del contexto es satisfacible se intersecta el autómata booleano asociado a dicha fórmula con una gramática libre del contexto que comprueba que todas las instancias de las variables tienen el mismo valor, luego se comprueba si el lenguaje generado es no vacío.

El trabajo expuesto en [8] es una generalización de la idea de [1], pero esta vez se intersecta el autómata booleano asociado a la fórmula booleana con una gramática de concatenación de rango simple (esta gramática se define en el próximo capítulo) que describe el orden de las variables. Luego se procede igual que en [1] y se comprueba si el lenguaje generado es no vacío.

En este trabajo se sigue otro enfoque para resolver el SAT usando teoría de lenguajes, en vez de usar el problema del vacío se construye el lenguaje de todas las fórmulas booleanas satisfacibles y para determinar si un SAT es satisfacible se analiza el problema del vacío. Esto permite demostrar que muchos formalismos tienen el problema de la palabra NP-Duro (los que sean cerrados bajo transducción finita y su representación luego de la transducción sea $O(1)$, y además que genere un lenguaje que es subconjunto de L_{copy}, el cual se define en el capítulo 3). Por otro lado se obtiene una gramática de concatenación de rango que reconoce las fórmulas booleanas satisfacibles, lo que permite demostrar que las gramáticas de concatenación de rango reconocen todos los problemas que pertenecen a la clase NP. Pero antes de esto es necesario definir y analizar las gramáticas de concatenación de rango y a ello se dedica el próximo capítulo.

Capítulo 2

Gramáticas de concatenación de rango

En este capítulo se definen y analizan las gramáticas de concatenación de rango, las cuales se usan en el capítulo 4 para definir el lenguaje de todas las fórmulas booleanas satisfacibles.

Las gramáticas de concatenación de rango (RCG) [4] 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. Las RCG fueron diseñadas con el fin analizar propiedades y características del lenguaje natural.

En la próxima sección se presentan algunas nociones que sirven de introducción para las principales definiciones y conceptos de las gramáticas de concatenación de rango.

2.1. Nociones sobre las gramáticas de concatenación de rango

En esta sección se presentan algunos aspectos que sirven como base introductoria para comprender los conceptos y definiciones sobre gramáticas de concatenación de rango.

Para ello suponga que tiene una gramática con la siguiente regla de derivación:

$$A(XYZ) \rightarrow B(X)C(Y)D(Z).$$

La regla anterior significa que el no terminal A recibe una cadena, y las variables que se encuentran como argumento de A: XYZ, significan todas las formas de dividir la cadena que recibe A en 3 subcadenas de la cadena que no se solapen y que su

X	Y	Z
a	b	c
ab	ε	c
ab	c	ε
abc	ε	ε
ε	ab	c
ε	abc	ε
ε	ε	abc
a	ε	bc
$\vdots$	$\vdots$	$\vdots$
a	bc	ε

Figura 2.1: Posibles valores de las variables X, Y y Z

concatenación forme la cadena original. Por ejemplo, si el no terminal A recibe la cadena *abc* los valores de X, Y y Z pueden ser interpretados de la siguiente manera (Figura 2.1):

Entonces el primer paso es asignarle una subcadena de la cadena de entrada a cada variable. Supongamos que fue la primera, en la que $X = a$, $Y = b$, $Z = c$, y con esa asignación de variables se evalúa en los no terminales de su parte derecha: $B(X)C(Y)D(Z)$, que en este caso sería $B(a)C(b)D(c)$.

Con otra asignación de valores a las variables X, Y y Z, se tiene por ejemplo que si $X = ab$, $Y = \varepsilon$, $Z = c$, entonces la parte derecha de $A(XYZ)$ se evalúa de la siguiente forma: $B(ab)C(\varepsilon)D(c)$.

Con la idea anterior se puede hablar del concepto de rango, que no es más que un par de índices i y j, tales que $i \leq j$, estos representan la subcadena de la cadena de entrada que comienza en el i-ésimo caracter y termina en el j-ésimo caracter.

El concepto de rango se utiliza cuando se evalúa en un no terminal y se le asigna a cada variables un rango de la cadena, tales que estos no se solapen como se mostró en el ejemplo anterior.

Dadas estas nociones, a continuación se presentan las principales definiciones de las gramáticas de concatenación de rango.

2.2. Definiciones

Rango: un rango es una tupla (i, j) que representa un intervalo de posiciones en una 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 la dimensión del vector de cadenas que reconoce y cada cadena del vector es asociada a un argumento del predicado.
- 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:

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

  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, el cual la **aridad** es igual a 1.

Por ejemplo, la siguiente gramática reconoce el lenguaje $L^3_{copy} = \{www \,|\, w \in \{a, b, c\}^*\}$:

$$G^3_{copy} = (N, T, V, P, S),$$

donde:

- N=$\{A, S\}$.
- T=$\{a, b, c\}$.
- V=$\{X, Y, Z\}$.
- El conjunto de cláusulas P es el siguiente:
  1. $S(XYZ) \to A(X, Y, Z)$
  2. $A(aX, aY, aZ) \to A(X, Y, Z)$
  3. $A(bX, bY, bZ) \to A(X, Y, Z)$
  4. $A(cX, cY, cZ) \to A(X, Y, Z)$
  5. $A(\varepsilon, \varepsilon, \varepsilon) \to \varepsilon$
- El símbolo inicial es S.

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. Este es un caso particular de las RCG el cual es usado en [8] para describir el orden de las variables de una fórmula booleana.

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 $X \in V$ y $a \in T$, 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. Proceso de derivación

La idea principal para realizar una derivación en las RCG se basa en dado un vector de cadenas, identificar para cada argumento del predicado izquierdo de una cláusula, todas las posibles sustituciones dentro del rango de la cadena correspondiente. Esto implica asociar los valores de las variables reconocidas en los argumentos del predicado izquierdo con los argumentos de los predicados derechos. A partir de esta asociación, el proceso de derivación continúa aplicándose en los predicados derechos.

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

Las RCG, a diferencia de las gramáticas definidas en el capítulo 1 no generan cadenas, su funcionamiento se basa en reconocer si una cadena pertenece o no al lenguaje.

Un vector de cadenas se reconoce por un predicado A si existe una secuencia de derivaciones que comienza en A y termina en la cadena vacía.

Por ejemplo, dada la cláusula $A(X_1, X_2, X_3) \rightarrow B_1(X_1)B_2(X_2)B_3(X_3)$, el vector (w_1, w_2, w_3) se reconoce por A, si existe una secuencia de derivaciones para cada uno de los predicados $B_1(w_1)$, $B_2(w_2)$, $B_3(w_3)$ que derive en la cadena vacía.

A continuación se presenta un ejemplo de reconocimiento de una cadena por G^3_{copy}: La cadena $abcabcabc$ se reconoce por G^3_{copy}, ya que se puede derivar de la siguiente manera:

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

Para mostrar un ejemplo de sustitución en rango sobre la cadena w se define $w[i \ldots j]$ como el rango que va desde el i-ésimo caracter hasta el j-ésimo con la cadena

indexada en 0. Entonces si $w = abcabcabc$, al realizar el reconocimiento en rango sobre el predicado S se pueden asociar las variables $X = ab$, $Y = ca$ $Z = bcabc$ a los rangos $w[0\ldots1]$, $w[1\ldots2]$, $w[3\ldots8]$ respectivamente. Otra opción es asociar las variables $X = abca$, $Y = bc$ $Z = abc$ a los rangos $w[0\ldots3]$, $w[4\ldots5]$, $w[6\ldots8]$ respectivamente. De manera similar, se pueden hacer $\binom{8}{3}$ sustituciones en rango distintas. A continuación se detalla el proceso de reconocimiento de la cadena *abcabcabc* por la gramática G^3_{copy}.

- **Primer paso:** Se toma la primera cláusula, la sustitución en rango asocia las variables $X = abc$, $Y = abc$ y $Z = abc$ a los rangos $w[0\ldots2]$, $w[3\ldots5]$ y $w[6\ldots8]$.
- **Segundo paso:** Se toma la segunda cláusula, la sustitución en rango asocia las variables $X = bc$, $Y = bc$ y $Z = bc$ a los rangos $w[1\ldots2]$, $w[4\ldots5]$ y $w[7\ldots8]$ respectivamente, derivando en el predicado $A(bc, bc, bc)$.
- **Tercer paso:** Se toma la tercera cláusula, la sustitución en rango asocia las variables $X = c$, $Y = c$ y $Z = c$ a los rangos $w[2\ldots2]$, $w[5\ldots5]$ y $w[8\ldots8]$ respectivamente, derivando en el predicado $A(c, c, c)$.
- **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 $A(\varepsilon, \varepsilon, \varepsilon)$.
- **Quinto paso:** 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*.

A continuación se presentan las principales propiedades de las RCG que se utilizan en este trabajo.

2.4. Propiedades de las RCG

En esta sección se describen las principales propiedades de las RCG que se utilizan en este trabajo [2]:

- **No cerradas bajo homomorfismo:** Dada una RCG G, el homomorfismo de un lenguaje que se reconoce por G no es necesariamente se reconoce por una RCG [2].
- **No cerradas bajo transducción finita:** Dada una RCG G, la transducción finita de un lenguaje que se reconce 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.

En la siguiente sección se analiza el problema de la palabra para las RCG, el cual se utiliza en el capítulo 4, para construir el lenguaje de todas las fórmulas booleanas satisfacibles mediante una RCG.

2.5. Problema de la palabra

En [4] se menciona que el problema de la palabra para las RCG es polinomial y se resuelve mediante un algoritmo de memorización sobre las cadenas asignadas a los argumentos de los predicados de la RCG [4]. 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, y tiene 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.

Existen casos en los que el problema de la palabra no es polinomial [2]. El ejemplo presentado en [2] muestra una RCG que reconoce cadenas de unos donde la cantidad de unos es un cuadrado perfecto, en la siguiente sección se analiza otro caso en el que este problema no es polinomial.

2.5.1. Problema de la palabra no polinomial para las RCG

El algoritmo de reconocimiento mencionado 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.

¿Qué pasaría si algún predicado de la gramática trabajara con rangos que no pertenecen a la cadena de entrada, es decir, que se generan mediante el proceso de reconocimiento de la cadena de entrada? En este caso si se emplea el algoritmo anterior ya la complejidad no depende de la cantidad de rangos de la cadena de entrada porque pueden aparecer otros rangos que no pertenezcan a dicha cadena, que se generan mediante el reconocimiento de la cadena.

Por ejemplo, a continuación se presenta una RCG que reconoce el lenguaje $L = \{w \,|\, w \in \{0,1\}\}$. Esta RCG no tiene uso real porque existe otra RCG equivalente que reconoce el mismo lenguaje, pero ilustra una RCG donde se generan rangos que no pertenecen a la cadena de entrada durante el proceso de reconocimiento:

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

donde:

- N=$\{A, B, Eq, S\}$.
- T=$\{0, 1\}$.
- V=$\{X, Y\}$.
- El conjunto de cláusulas P es el siguiente:
  1. $S(X) \to A(X, X)$
  2. $A(1X, Y) \to B(X, 0, Y)$
  3. $A(1X, Y) \to B(X, 1, Y)$
  4. $A(0X, Y) \to B(X, 1, Y)$
  5. $A(0X, Y) \to B(X, 0, Y)$
  6. $B(1X, Y, Z) \to B(X, 1Y, Z)$
  7. $B(1X, Y, Z) \to B(X, 0Y, Z)$
  8. $B(0X, Y, Z) \to B(X, 0Y, Z)$
  9. $B(0X, Y, Z) \to B(X, 1Y, Z)$
  10. $B(\varepsilon, Y, Z) \to Eq(Y, Z)$
- 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 $w = q$. 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 la complejidad del reconocimiento de G es mayor que 2^n (con n igual al tamaño de la cadena de entrada), ya que esta es la cantidad de rangos posibles que puede recibir el predicado B, porque la gramática es ambigua y en cada derivación de B existen 2 posibles decisiones. En el capítulo 4 se presenta una RCG ambigua con el problema de la palabra no polinomial, pero que sí tiene aplicación en el problema de la satisfacibilidad booleana.

En este capítulo se analizaron las principales definiciones y propiedades de las RCG, que son utilizadas en el capítulo **??** para definir una gramática que reconozca las fórmulas booleanas satisfacibles. En el próximo capítulo se presenta un primer enfoque para definir el lenguaje de todas las fórmulas booleanas satisfacibles y a esta idea se le da continuidad en el capítulo 4, mediante las RCG.

Capítulo 3

Lenguaje de las fórmulas booleanas satisfacibles empleando transducción finita

En este capítulo se presenta un lenguaje formal, llamado L_{S-SAT}, al cual pertenecen todos los problemas SAT que son satisfacibles, y se muestra una forma de construirlo a partir de una transducción finita de una variante del lenguaje L_{copy} sobre el alfabeto $\{a, b, c, d\}$. Este lenguaje permitiría otra vía de solución para el SAT, que consiste en determinar si una fórmula dada pertenece a este lenguaje. Esto se puede hacer resolviendo el problema de la palabra.

Para definir el lenguaje L_{S-SAT} se presenta una vía para codificar una fórmula booleana mediante cadenas sobre el alfabeto $\Sigma = \{a, b, c, d\}$, y para construirlo se utiliza una transducción finita del alfabeto $\{0, 1, d\}$ en Σ.

La estructura de este capítulo es la siguiente: en la sección 3.1 se muestra como codificar una fórmula booleana usando el alfabeto $\{a, b, c, d\}$. En la sección 3.2 se muestra como interpretar las cadenas sobre el alfabeto $\{0, 1, d\}$ como asignaciones de las variables. Finalmente, en la sección 3.4 se presenta un transductor finito que convierte cadenas del lenguaje $L_{FULL-SAT}$ que se define en la sección 3.2 en cadenas sobre el alfabeto $\{a, b, c, d\}$ que representan fórmulas booleanas satisfacibles. Seguidamente se conjetura por qué la representación del lenguaje de las fórmulas booleanas satisfacibles, en cualquier formalismo que lo genere usando la estrategia propuesta en este capítulo, tiene un tamaño $O(1)$, lo que implica la que el problema de la palabra para estos formalismos que puedan generar el lenguaje, es NP-Duro.

A continuación se presenta cómo codificar una fórmula booleana mediante una cadena sobre el alfabeto $\{a, b, c, d\}$.

3.1. Codificación de una fórmula booleana a una cadena

Una fórmula booleana F, con v variables en CNF tiene la siguiente estructura:

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

donde cada cláusula X_i es una disyunción de literales

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

cada literal L_{ij} es una variable booleana o su negación y $m \leq v$.

Si se tiene una fórmula booleana F en forma normal conjuntiva se puede considerar que cada una de las v variables de la fórmula aparece en cada cláusula de F en uno de tres posibles estados: la variable aparece sin negar, la variable aparece negada, o la variable no aparece.

Por ejemplo, en la primera cláusula de la siguiente fórmula booleana en CNF con 3 variables:

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

la variable x_1 aparece sin negar, la variable x_2 aparece negada, y la variable x_3 no aparece.

El hecho de que se pueda asumir que en todas las cláusulas aparecen todas variables permite representar una cláusula de una una fórmula con v variables como una cadena de v símbolos, donde el símbolo en la posición i indica el estado de la variable x_i en la cláusula.

En este trabajo se propone usar los símbolos a, b y c para indicar el estado de una variable en una cláusula:

- a: la variable aparece sin negar,
- b: la variable aparece negada, y
- c: la variable no aparece.

Con este convenio, la primera cláusula de F se puede representar mediante la cadena *abc*.

Si se agrega un nuevo símbolo (por ejemplo d) que indique el final de una cláusula, una fórmula lógica con v variables y k cláusulas se puede representar mediante k bloques de longitud v, donde cada bloque está formado por los símbolos a, b, o c, y cada bloque se separa del siguiente por el símbolo d.

Con este convenio, la fórmula F se representa mediante la cadena:

$$abc\mathbf{d}baa\mathbf{d}aba\mathbf{d}$$

donde los símbolos **d** aparecen en negrita para facilitar la interpretación de la cadena como fórmula en forma normal conjuntiva.

Solo resta definir el proceso inverso, es decir, dada una cadena determinar la fórmula booleana en CNF que representa dicha cadena. Para que una cadena w pueda ser interpretada como una fórmula booleana debe cumplir con las siguientes condiciones: tener n bloques separados por d, cada bloque de las misma longitud v y en cada bloque solo pueden ser usados los caracteres a, b, c. Entonces la cadena w puede ser interpretada por como una fórmula booleana con n cláusulas y v variables y la estructura de las cláusula depende de los caracteres correspondiente al bloque de a, b y c asociado a dicha cláusula.

Por ejemplo la cadena $acc\mathbf{d}aba\mathbf{d}cba\mathbf{d}$, se puede interpretar como la siguiente fórmula booleana:

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

Entonces una vez definida la transformación de una fórmula booleana en una cadena se puede definir el lenguaje de todas las fórmulas booleanas en CNF:

$$L_{FULL-SAT} = \{w \,|\, \exists F : t_{SAT}(F) = w\},$$

donde F es una fórmula booleana en CNF y $t_{SAT}(x)$ es una función que recibe una fórmula booleana y devuelve su representación mediante los estados a, b y c y el separador d.

Seguidamente se muestra como interpretar una cadena como la asignación de los valores de las variables de una fórmula booleana.

3.2. Interpretar una cadena como asignaciones de variables para una fórmula booleana

En esta sección se muestra como dada una cadena q y una cadena $f \in L_{FULL-SAT}$, interpretar la cadena w como la asignación de los valores de las variables para la fórmula booleana que representa la cadena f.

Para que la cadena e pueda ser interpretada como una asignación de valores para f, es necesario al igual que en la codificación anterior poder distinguir los valores asignados a cada cláusula, esto se logra mediante el caracter d, y para representar los valores de las variables asignados a cada cláusula se utilizan cadenas binarias. Para cada cadena binaria se le asocia el valor del i-ésimo caracter al valor de la i-ésima

variable de la cláusula, si este caracter es un 1 se le asocia un valor de *true* (verdadero) y si el caracter es un 0 se le asocia el valor de *false* (falso).

Otra restricción importante es que la cadena binaria debe ser la misma para cada cláusula lo que mantiene la invariante de que a todas las instancias de la misma variable le corresponde el mismo valor y la longitud de la cadena binaria debe ser exactamente igual al número de variables de la fórmula booleana.

Por ejemplo, si $e = 101\mathbf{d}101\mathbf{d}101\mathbf{d}$ y $f = abc\mathbf{d}cbb\mathbf{d}acc\mathbf{d}$, fórmula booleana:

$$(x_1 \vee \neg x_2) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_1),$$

se evalúa de la siguiente manera:

$$(true \vee \neg false) \wedge (\neg false \vee \neg true) \wedge (true).$$

Entonces, se puede definir el lenguaje de todas las interpretaciones de las fórmulas booleanas representas en la codificación de $L_{FULL-SAT}$ como:

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

En la próxima sección se define el lenguaje L_{S-SAT}.

3.3. Definición de L_{S-SAT}

El lenguaje de todas las fórmulas booleanas en CNF que son satisfacibles se define como $L_{S-SAT} = \{w \,|\, w \in L_{FULL-SAT}\}$, donde $L_{FULL-SAT}$ representa el lenguaje de todas las fórmulas booleanas en CNF y w representa una fórmula booleana satisfacible.

En la próxima sección se presenta una vía para construir L_{S-SAT} usando transducción finita.

En la próxima sección se demuestra que L_{S-SAT} no es un lenguaje libre del contexto, por lo que el formalismo que lo genere necesariamente debe pertenecer a las gramáticas dependientes del contexto o las gramáticas irrestrictas.

3.3.1. Demostración de que L_{S-SAT} no es un lenguaje libre del contexto

En esta sección se demuestra que el lenguaje L_{S-SAT} no es libre del contexto. Una técnica que se emplea para demostrar que un lenguaje no es libre del contexto, es el lema del bombeo para los lenguajes libres del contexto [7].

El lema del bombeo establece que si L es un lenguaje libre del contexto existe una constante n tal que para toda cadena $q \in L$, w puede escribirse de la forma $q = uvwxy$ tal que: $|vwx| \leq n$, $vx \neq \varepsilon$ y $\forall i \geq 0\, uv^i wx^i y \in L$ [7].

Suponga que L_{S-SAT} es un lenguaje libre del contexto y la constante es n, entonces se cumple que

$$q = a\underbrace{c\ldots c}_{n+1}\mathbf{d}ca\underbrace{c\ldots c}_{n}\mathbf{d}\ldots\underbrace{c\ldots c}_{n}ac\mathbf{d}\underbrace{c\ldots c}_{n+1}a\mathbf{d},$$

pertenece a L_{S-SAT}, ya que la fórmula

$$(x_1) \wedge (x_2) \wedge \ldots \wedge (x_{n+2})$$

es satisfacible. Por tanto existen u, v, w, x y y con $vx \neq \varepsilon$ y $|vwx| \leq n$ tales que $q = uvwxy$.

Como $|vwx| \leq n$ y cada cláusula tiene $n+2$ caracteres se cumple que vwx tiene caracteres a lo sumo en 2 cláusulas, por tanto la cadena uv^2wx^2y tiene una o dos cláusulas con más caracteres que las restantes y esto contradice el hecho de que $uv^2wx^2y \in L_{S-SAT}$. Entonces, se cumple que L_{S-SAT} no es un lenguaje libre del contexto.

Seguidamente se muestra como construir L_{S-SAT} mediante una transducción finita.

3.3.2. Construcción del L_{S-SAT} usando transducción finita

La idea para definir L_{S-SAT} es construir un transductor finito, denominado T_{SAT}, que acepte como entrada cadenas $w \in L_{0,1}$ y devuelva cadenas de $f \in L_{FULL-SAT}$ tales que al evaluar w en f, f es verdadera.

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 $L_{0,1}$.

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

En la siguiente sección se define el transductor T_{SAT}.

3.4. Transductor T_{SAT}

En esta sección se define el transductor finito T_{SAT} (Figura 3.2), el cual se usa para construir L_{S-SAT}, mediante una transducción finita.

Para ello se construye el transductor T_{CLAUSE} (Figura 3.1) que dada una cadena binaria w genera todas las posibles cláusulas sobre el alfabeto $\{a, b, c, d\}$ que sean satisfacibles por w.

La idea detrás de T_{CLAUSE} es construir un transductor que genere todas las posibles cláusulas satisfacibles por los valores de las variables que determina la cadena de entrada. Este transductor tiene 3 estados: el estado inicial, el estado positivo (representa que la cláusula generada ya es satisfacible) y el estado negativo (representa

que la cláusula generada aún no es satisfacible). Las transiciones entre los estados se realizan dependiendo del caracter que se lee y el caracter que se escribe y además dependiendo si la asignación que se realiza en el momento de leer o de escribir satisface la cláusula o no.

A continuación de describe el funcionamiento de cada estado del transductor T_{CLAUSE}:

- 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 cuando se escribe a pasa al estado negativo y cuando se escribe una b pasa al estado positivo, mientras que cuando se escribe c permanece en el mismo estado.

- 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 estado es el estado de aceptación para el transductor lo cual significa que la cláusula se evalúa con un valor de verdad positivo.

- Estado q_n (estado negativo de T_{CLAUSE}): representa que para los valores asignados a las variables hasta el momento se obtiene un valor de verdad negativo. 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 cuando se escribe a pasa al estado negativo y cuando se escribe una b pasa al estado positivo, mientras que cuando se escribe c permanece en el mismo estado.

Seguidamente se define T_{CLAUSE}.

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

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:
  - $\delta_{SAT}(q_0, 1) = (q_p, a)$
  - $\delta_{SAT}(q_0, 0) = (q_n, a)$
  - $\delta_{SAT}(q_0, 1) = (q_n, b)$
  - $\delta_{SAT}(q_0, 0) = (q_p, b)$
  - $\delta_{SAT}(q_0, 1) = (q_0, c)$
  - $\delta_{SAT}(q_0, 0) = (q_0, c)$
- Transiciones para el estado q_p (estado positivo de T_{CLAUSE}):
  - $\delta_{SAT}(q_p, 1) = (q_p, a)$
  - $\delta_{SAT}(q_p, 0) = (q_p, a)$
  - $\delta_{SAT}(q_p, 1) = (q_p, b)$
  - $\delta_{SAT}(q_p, 0) = (q_p, b)$
  - $\delta_{SAT}(q_p, 1) = (q_p, c)$
  - $\delta_{SAT}(q_p, 0) = (q_p, c)$
- Transiciones para el estado q_n (estado negativo de T_{CLAUSE}):
  - $\delta_{SAT}(q_n, 1) = (q_p, a)$
  - $\delta_{SAT}(q_n, 0) = (q_n, a)$
  - $\delta_{SAT}(q_n, 1) = (q_n, b)$
  - $\delta_{SAT}(q_n, 0) = (q_p, b)$
  - $\delta_{SAT}(q_n, 1) = (q_n, c)$
  - $\delta_{SAT}(q_n, 0) = (q_n, c)$

Para definir T_{SAT} mediante T_{CALUSE}, la idea es modificar T_{CALUSE} para que cuando se genere una cláusula y termine en el estado positivo se continue leyendo los valores de la cadena de entrada que son necesarios para generar la próxima cláusula. Lo anterior se logra definiendo q_0 como el estado de aceptación y agregando una transición del estado q_p al estado q_0 que lea una d y escriba una d, de esta manera cuando una cláusula se genera con un valor de verdad positivo se pasa a generar la siguiente cláusula desde el estado inicial.

La construcción anterior tiene una dificultad y es que el transductor genera la cadena vacía, y la cadena vacía representa una fórmula booleana con 0 variables, por lo que no tiene sentido que se considere en L_{S-SAT}. Para solucionar lo anterior se pueden tomar 2 transductores T_{CLAUSE} y unirlos mediante una transición, esta idea se expone a continuación.

Para definir el transductor T_{SAT} (Figura 3.2) se toman 2 transductores T_{CLAUSE} (T_1 y T_2 respectivamente) y se concatenan añadiendo una transición del estado estado positivo de T_1 q_{p_1} al estado inicial de T_2 q_{0_2} con el símbolo d (tanto de lectura como

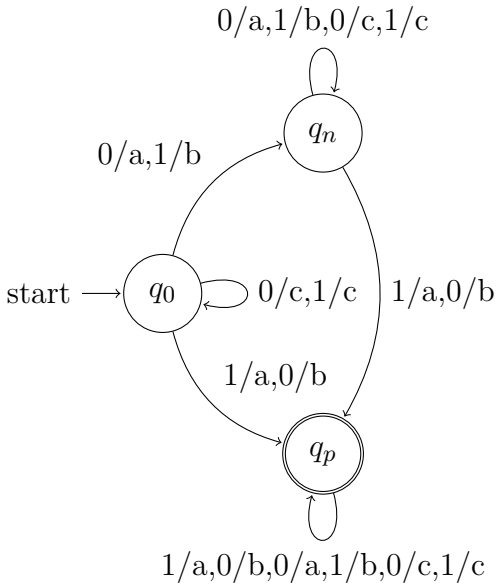

Figura 3.1: Transductor T_{CLAUSE}.

de escritura) y además se agrega una transición del estado del estado estado positivo de T_2 q_{p_2} al estado q_{0_2} con el símbolo d (tanto de lectura como de escritura). Para terminar se define el estado inicial y el estado final de T_{SAT}, los cuales serían el estado inicial de T_1 q_{0_1} y el estado inicial de T_2 q_{0_2}, respectivamente.

A continuación se presenta la demostración de que la construcción de L_{S-SAT} mediante una transducción finita reconoce todas las fórmulas satisfacibles.

3.5. Demostración de que la construcción de L_{S-SAT} mediante una transducción finita reconoce todas las fórmulas satisfacibles

La idea de la demostración es probar que para cualquier cadena binaria w, $T_{CLAUSE}(w)$ es el conjunto de todas las cláusulas que son satisfacibles por w. Después demostrar que si $e = (wd)^n \wedge w \in \{0,1\}^*$, $T_{SAT}(e)$ contiene todas las fórmulas de n cláusulas satisfacibles por la cadena w donde y por último demostrar que:

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

Para demostrar que dada una cadena binaria w, $T_{CLAUSE}(w)$ es el conjunto de todas las cláusulas que son satisfacibles por w primero suponga que $q \in T_{CLAUSE}(w)$. Esto significa que el transductor terminó en el estado q_p en el proceso que generó q y

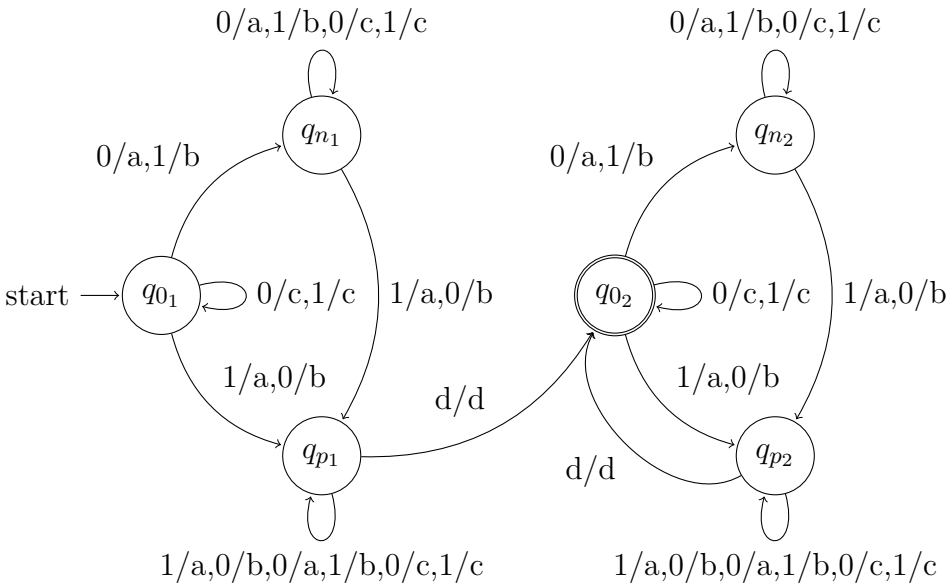

Figura 3.2: Transductor T_{SAT}.

como empezó en el estado q_0 ocurrió una transición desde q_0 a q_p o desde q_n a q_p. Esto solo es posible si el transductor leyó un 1 y escribió una a o si leyó un 0 y escribió una b. Lo cual significa que en la fórmula booleana que representa q hay una variable sin negar a la cual se le asigna un 1 o una variable negada a la que se le asigna un 0, por lo tanto se cumple que w satisface la fórmula booleana que representa q.

Sea una fórmula booleana F satisfacible por w, cuya representacion en $L_{FULL-SAT}$ es q, se cumple que en F hay una variable sin negar a la cual se le asigna un 1 en w o una variable negada a la cual se le asigna un 0 en w. Sin pérdida de la generalidad se asume que la primera variable que cumple lo anterior es la i-ésima.

Si se hace el reconocimiento de los primeros $i-1$ caracteres de la cadena de entrada por T_{CLAUSE} se pueden tomar las transiciones de tal manera que los primeros $i-1$ caracteres de la cadena generada sean iguales a los primeros $i-1$ caracteres de q. Dado este punto solo es posible que el transductor esté en el estado q_0 o q_n, pero como se cumple que la i-ésima variable está sin negar y se le asigna un 1 en w o está negada y se le asigna un 0 en w, entonces se puede tomar la opción de leer un 1 y escribir una a o leer un 0 y escribir una b según corresponda. De esta manera, según en el estado en que se encuentre el autómata, se pasa al estado q_p.

Luego se toman las restantes transiciones de manera que la cadena generada sea q y se mantiene en el mismo estado ya que q_p solo tiene transiciones hacia sí mismo. De esta manera se demuestra que $T_{CLAUSE}(w)$ es el conjunto de todas las cláusulas que son satisfacibles por w.

Para demostrar que $T_{SAT}(e)$ contiene todas las fórmulas satisfacibles por la cade-

na $e = (wd)^n$ donde $w \in \{0,1\}^*$ se hará una inducción sobre n. Se define los conjuntos $A_{w,n}$ como el conjunto formado por todas las cadenas que representan fórmulas booleanas de n cláusulas satisfacibles por w y B_w el conjunto formado por todas las cadenas que representan las cláusulas que son satisfacibles por w. El caso base $n = 1$ se demuestra porque la transducción se realiza solo sobre el primer transductor T_{CLAUSE} que conforma a T_{SAT} y como se demostró anteriormente $T_{CLAUSE}(w)$ contiene todas cláusulas satisfacibles por w.

Una vez demostrado el caso base corresponde asumir para $n = k$ y se demuestra para $n = k+1$.

Se cumple que el conjunto de todas las fórmulas booleanas satisfacibles por w con $k+1$ cláusulas es equivalente al conjunto que forman todas las fórmulas booleanas satisfacibles por w con k cláusulas concatenadas con todas las cláusulas satisfacibles por w:

$$A_{w,k+1} = \{xyd \,|\, x \in A_{w,k} \wedge y \in B_w\}.$$

Además por la estructura de T_{SAT} se cumple que el conjunto de todas las cadenas generadas por $T_{SAT}((wd)^{k+1})$ es igual al conjunto de todas las cadenas generadas por $T_{SAT}((wd)^k)$ concatenadas con todas las cadenas generadas por $T_{CLAUSE}(w)$:

$$T_{SAT}((wd)^{k+1}) = \{xyd \,|\, x \in T_{SAT}((wd)^k) \wedge y \in T_{CLAUSE}(w)\}.$$

Por hipótesis de inducción se cumple que $A_{w,k} = T_{SAT}((wd)^k)$ y además se cumple que $B_w = T_{CLAUSE}(w)$, lo cual implica que $A_{w,k+1} = T_{SAT}((wd)^{k+1})$.

Para concluir el tercer paso de la demostración, sea una cadena e tal que existe w, donde $w \in L_{0,1} \wedge e \in T_{SAT}(w)$ entonces se cumple que la fórmula booleana asociada a e es satisfacible por w. Ahora sea F una fórmula booleana satisfacible y sea e su cadena asociada, por tanto existe $w \in L_{0,1}$ tal que w satisface a F, luego se cumple que $e \in T_{SAT}(w)$. Por tanto se cumple que:

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

A continuación de demuestra que el problema de la palabra de cualquier formalismo que genere el lenguaje $L_{0,1}$, sea cerrado bajo transducción finita y su representación luego de la transducción sea $O(1)$, es NP-Duro

3.5.1. Demostración de que el problema de la palabra de cualquier formalismo que genere el lenguaje $L_{0,1}$, sea cerrado bajo transducción finita y su representación luego de la transducción sea $O(1)$, es NP-Duro

Para realizar la demostración suponga que existe un formalismo G que genera $L_{0,1}$, es cerrado bajo transducción finita y su representación luego de la transducción es

$O(1)$. Sea G' el formalismo que resulta de aplicarle el transductor T_{SAT} a G, entonces determinar si una cadena w pertenece al lenguaje generado por G', es equivalente a saber si la fórmula booleana a la cual representa w es satisfacible. Por tanto como la representación de G' es $O(1)$ entonces el problema de la palabra para G' es NP-Duro, porque tiene una reducción directa al problema de la satisfacibilidad booleana.

En este capítulo se presentó una estrategia para resolver el SAT usando teoría de lenguajes que se basa en definir y construir el lenguaje de todas las fórmulas satisfacibles. Además se presentó un primer acercamiento para construir L_{S-SAT}, mediante transducción finita lo cual demuestra que el problema de la palabra para todos los formalismos que generen $L_{0,1}$ y sean cerrados bajo transducción finita, es NP-duro. En el próximo capítulo se presenta otra estrategia distinta para definir L_{S-SAT}, la cual se basa en definir una RCG que reconozca las fórmulas booleanas satisfacibles.

Capítulo 4

Lenguaje de las fórmulas booleanas satisfacibles empleando gramáticas de concatenación de rango

En este capítulo se presenta como resolver el SAT usando el problema de la palabra, mediante gramáticas de concatenación de rango. Además obtiene una gramática de concatenación de tango que reconoce el lenguaje L_{S-SAT}, la cual permite demostrar que las RCG cubren 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 el desarrollo del capítulo primeramente se describe como reconocer el lenguaje $L_{0,1}$ mediante una gramática de concatenación de rango y a continuación se describe por qué no es posible usar las RCG para construir el lenguaje L_{S-SAT} mediante una transducción finita de un lenguaje de concatenación de rango. Posteriormente se presenta una RCG que reconoce todas las fórmulas booleanas satisfacibles y se demuestra que las RCG cubren todos los problemas en la clase NP. Por último se presenta una RCG que permite reconocer problemas 2-SAT, pero que tiene el problema de la palabra no polinomial, esto significa una vía distinta para resolver el SAT, lo cual puede contribuir a encontrar otras instancias polinomiales del SAT.

En la siguiente sección se describe una RCG que reconoce el lenguaje $L_{0,1}$.

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

En esta sección se presenta una RCG que reconoce el lenguaje $L_{0,1} = \{(wd)^+ \,|\, w \in \{0,1\}^+\}$, el cual sirve para la asignación de valores a las variables de un SAT. La gramática que reconoce $L_{0,1}$ se basa en 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 [4] y comprueba que dos cadenas sobre un alfabeto sean iguales. Por otro lado el predicado B recibe un patrón y una cadena y determina si algún prefijo de la cadena, que esté seguido de una d, es igual al patrón. Luego continúa la derivación del resto de la cadena en el predicado B con el mismo patrón. Finalmente si la cadena que recibe B es la cadena vacía entonces se deriva en la cadena vacía. Esto permite que $G_{0,1}$ reconozca el lenguaje $L_{0,1}$.

A continuación se presenta un ejemplo de cómo $G_{0,1}$ reconoce la cadena 101**d**101**d**101**d**.

4.1.1. Ejemplo de reconocimiento de una cadena por $G_{0,1}$

En esta sección se describen las derivaciones que permiten reconocer la cadena 101**d**101**d**101**d**. Las derivaciones de la gramática son las siguientes:

Notación que se emplea en los ejemplos de una derivación de una RCG: para mostrar un paso de la derivación primero se muestra el predicado izquierdo de la cláusula seleccionada, instanciado con los valores de los rangos correspondientes y los predicados en los que deriva. Luego se específica la cláusula seleccionada (ejemplo c-4, significa que se selecciona la cuarta cláusula) y después se muestran los valores que toma cada variable en la sustitución en rango.

1. $S(101\mathbf{d}101\mathbf{d}101\mathbf{d}) \rightarrow A(101\mathbf{d}101\mathbf{d}101d)$: c-1, $X = 101\mathbf{d}101\mathbf{d}101\mathbf{d}$
2. $A(101\mathbf{d}101\mathbf{d}101\mathbf{d}) \rightarrow B(101\mathbf{d}101\mathbf{d}, 101)C(101)$: c-2, $X = 101$ $Y = 101\mathbf{d}101\mathbf{d}$
3. $B(101\mathbf{d}101\mathbf{d}, 101) \rightarrow B(101\mathbf{d}, 101)C(101)Eq(101, 101)$: c-3, $X = 101$ $Y = 101\mathbf{d}$ $P = 101$
4. $B(101\mathbf{d}, 101) \rightarrow B(\varepsilon, 101)C(101)Eq(101, 101)$: c-3, $X = 101$ $Y = \varepsilon$ $P = 101$
5. $B(\varepsilon, 101) \rightarrow \varepsilon$: c-4, $P = 101$
6. $C(101) \rightarrow C(01)$: c-6, $X = 01$
7. $C(01) \rightarrow C(1)$: c-5, $X = 1$
8. $C(1) \rightarrow C(\varepsilon)$: c-6 $X = \varepsilon$
9. $C(\varepsilon) \rightarrow \varepsilon$: c-7

Como para todos los predicados existe una sustitución en rango, en la que estos derivan en la cadena vacía entonces $101\mathbf{d}101\mathbf{d}101\mathbf{d}$ es reconocida por $G_{0,1}$.

A continuación se analiza por qué no es posible construir L_{S-SAT} mediante una transducción finita usando una RCG

4.1.2. Construir L_{S-SAT} mediante una transducción finita usando una RCG

Como se mostró en la sección anterior el lenguaje $L_{0,1}$ se reconoce mediante una RCG. Siguiendo con la idea expuesta en las secciones del capítulo 3, donde se muestra cómo construir L_{S-SAT} mediante una transducción finita, se puede usar $G_{0,1}$ como formalismo para generar $L_{0,1}$. Esto no es posible porque las RCG no son cerradas bajo transducción finita como se mencionó en el capítulo 2. 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 para el formalismo que se obtiene después de las transducción como se propuso en el capítulo 3.

Se pudiera analizar 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.

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 muestra una RCG que reconoce el lenguaje L_{S-SAT}, lo cual demuestra que la transducción finita no es la única vía para construir este lenguaje.

4.2. Reconocer L_{S-SAT} mediante una RCG

En esta sección se presenta una RCG que reconoce las fórmulas booleanas satisfacibles. Esto permite demostrar que las RCG cubren todos los problemas de la clase NP.

En la construcción de L_{S-SAT} mediante una transducción finita se usa el lenguaje $L_{0,1}$ para generar todas las posibles interpretaciones de cualquier fórmula booleana y con estas interpretaciones el transductor T_{S-SAT} genera todas las posibles fórmulas booleanas satisfacibles. En esta sección, en cambio, se presenta un RCG que reconoce las fórmulas booleanas satisfacibles, la cual se basa en generar, durante el reconocimiento de la cadena de entrada, todas las posibles interpretaciones de la fórmula booleana que satisfacen la primera cláusula y luego comprobar si alguna de ellas satisface el resto de las cláusulas.

Para definir la gramática, se agrupan las cláusulas en 4 fases, en dependencia de las tareas que cumplen durante el reconocimiento:

- **Primera fase:** representa la derivación inicial de la gramática.
- **Segunda fase:** se encarga de generar todas las posibles interpretaciones de las variables que satisfacen la primera cláusula. En esta fase se definen 2 estados: positivo (significa que la cadena de 0 y 1 generada ya satisface la primera cláusula) y negativo (significa que la cadena de 0 y 1 generada aún no satisface la primera cláusulas). Estos estados se representan por los predicados P y N, respectivamente.
- **Tercera fase:** comprueba que la interpretación que se define en la fase anterior sea satisfacible para el resto de las cláusulas.
- **Cuarta fase:** define el algoritmo para determinar si una interpretación satisface una cláusula dada. En esta fase se definen 2 estados, positivo (significa que la interpretación ya satisface la cláusula actual) y negativo (significa que la interpretación aún no satisface la cláusula actual). Representados por los predicados Cp y Cn respectivamente.

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:

- **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 una cadena de 0 y 1 que asigna valores a las variables de la fórmula booleana, de forma que satisfagan la primera cláusula.

  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. $N(cX, Y) \rightarrow N(X, Y0)$

  A representa el predicado por donde inician las derivaciones de esta fase, P representa que la cláusula ya tiene un valor de verdad positivo y N representa que la cláusula de la fórmula booleana se encuentra aún con un valor de verdad negativo. De A se deriva a los predicados P y N en dependencia del valor asignado, a la variable del literal que se encuentra al inicio del rango actual. El predicado P deriva hacia sí mismo independientemente del símbolo, exceptuando el símbolo d, caso en el que se deriva en B y se procede a la siguiente fase.

- **Tercera fase:** El siguiente conjunto de cláusulas comprueba que la asignación de variables que se realiza en la fase anterior se verdadera para las restantes cláusulas.

  21. $B(X_1 d X_2, Y) \rightarrow C(X_1, Y) B(X_2, Y)$
  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:** En esta fase se define el comportamiento de C, que recibe una cláusula y una interpretación de las variables y comprueba que dicha interpretación sea verdadera 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 que reconoce G_{S-SAT} es exactamente igual al lenguaje que representa todas las fórmulas booleanas satisfacibles descritas mediante la transformación que se define en el lenguaje $L_{FULL-SAT}$.

Demostración de que la gramática G_{S-SAT} reconoce el lenguaje L_{S-SAT}

La idea para la demostración de que el lenguaje que reconoce G_{S-SAT} es exactamente igual al lenguaje que representa todas las fórmulas booleanas satisfacibles se basa en demostrar la tarea que cumple cada una de las 4 fases de la gramática durante el proceso de reconocimiento y probar que una representación de una fórmula booleana se reconoce por G_{S-SAT} si y solo si dicha fórmula es satisfacible.

Primeramente, se demuestra que el predicado C, reconoce las cadenas w y e si y solo si e satisface a la cláusula que representa w. Posteriormente se prueba que el predicado B reconoce las cadenas w y e si y solo si e satisface a todas las cláusulas de w. Luego, se demuestra que el conjunto de cadenas Q formado por todas las cadenas q tales que existe una secuencia de derivaciones desde del predicado $A(q)$ hasta $B(z_q, e)$ es exactamente igual a al conjunto de todas las interpretaciones que hacen verdadera la primera cláusula de q. Por último, se prueba que el conjunto de cadenas q que se reconocen 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 la tarea que cumple de cada fase. Primero se demuestra la tarea de la cuarta fase, luego la tercera, después la segunda y por último la primera, ya que la demostración de la tercera depende de la cuarta y la demostración de la primera depende de la segunda y de la tercera.

- **Cuarta fase:** Sean las cadenas w y e, asociadas a una cláusula y una interpretación de variables respectivamente, y el predicado $C(w, e)$, para demostrar e satisface a fórmula booleana asociada a w, la cual se denomina F_w si y solo si $C(w, e)$ deriva en la cadena vacía primeramente suponga que F_w es satisfacible por e, entonces se debe demostrar que existe una secuencia de derivaciones desde $C(w, e)$ hasta la cadena vacía.

  Como e satisface a F_w existe una variable sin negar con valor un 1 o un variable negada con valor 0. Del predicado C se deriva al predicado Cn, las únicas derivaciones de la gramática donde se deriva del predicado Cn a Cp es la combinación de una a y un 1 o de una b y un 0 y como e satisface F_w esta combinación existe. Por último, Cp siempre deriva en sí mismo o en la cadena vacía por lo que queda demostrado que existe una secuencia de derivaciones desde $C(w, e)$ hasta la cadena vacía.

  Para finalizar la demostración de la cuarta fase, es necesario probar que si $C(w, e)$ se reconoce entonces e satisface a F_w. Por la estructura de la gramática, si existe una secuencia de derivaciones desde $C(w, e)$ hasta la cadena vacía entonces hay una derivación desde Cn hacia Cp. 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. Por lo que existe una variable sin negar con valor 1 o una variable negada con valor 0 en F_w, lo cual implica que e satisface F_w.

- **Tercera fase:** Sean las cadenas w y e, asociadas a una fórmula booleana y una interpretación de variables respectivamente, el predicado $B(w, e)$ y la cláusula $B(X_1 d X_2, Y) \rightarrow C(X_1, Y) B(X_2, Y)$, para demostrar que e satisface todas las cláusulas de w se hará una inducción sobre la cantidad de cláusulas n de la fórmula booleana asociada a w.

Para $n = 1$ se cumple que, al realizar la sustitución en rango, los rangos asociados a las variables X_1 y X_2 son w sin su último caracter y la cadena vacía respectivamente. Por tanto $B(w, e)$ se reconoce por la gramática si y solo si $C(X_1, e)$ se reconoce y esto solo es posible si e satisface a X_1, por lo que se demuestra el caso base.

Una vez demostrado el caso base corresponde asumir para $n = k$ y demostrar para $k+1$.

En todas las posibles sustituciones en rango de X_1 y X_2, $C(X_1, e)$ solo se reconoce si $|X_1| = |e|$, entonces el caso de sustitución en rango que ocupa a la demostración es cuando $|X_1| = |e|$, porque para el resto de las sustituciones en rango $C(X_1, e)$ no se reconoce por la gramática. Como e satisface todas las cláusulas de w si y solo si satisface a la primera cláusula de w y el resto de las cláusulas de w, que en este caso están asociadas a las variables X_1 y X_2 respectivamente. Precisamente $B(w, e)$ se reconoce si y solo si se reconoce $C(X_1, e)$ y $B(X_2, e)$. $C(X_1, e)$ se demuestra por la cuarta fase y $B(X_2, e)$ se demuestra por hipótesis de inducción, ya que X_2 tiene k cláusulas.

- **Segunda fase:** Sea la cadena q asociada a una fórmula booleana y el predicado $A(q)$, para demostrar que durante el reconocimiento en esta fase se generan todas las cadenas que satisfacen a q se utilizan 2 conjuntos: W y Q. W representa el conjunto de todas las cadenas de 0 y 1 que satisfacen la primera cláusula de la fórmula asociada a q , denominada F_{1q}, y E, representa el conjunto de todas las cadenas e tales que existe una secuencia de derivaciones desde $A(q)$ hasta $B(z_q, e)$, donde z_q esta conformada por todas las cláusulas de q menos la primera.

  Dadas estas definiciones es necesario demostrar que $W = E$, para ello se debe demostrar que W es subconjunto de E y E es subconjunto de W.

  Para demostrar que $W \subseteq E$, se toma una cadena w tal que $w \in W$, es decir, w satisface a F_{1q}. Por tanto en F_{1q} 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 hacia 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 $B(z_q, w)$ se cumple que $w \in E$.

  Para demostrar que $Q \subseteq W$, se toma una cadena e tal que $e \in E$, por lo que existe una secuencia de derivaciones desde $A(q)$ a $B(z_q, e)$. 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_{1q} tiene una variable sin negar con valor 1 en e o una variable negada con valor 0 en e. Entonces se cumple que e satisface a F_{1q} por lo que $e \in W$. Con esto se demuestra que $E = W$.

- **Primera fase:** Sea la cadena q asociada a una fórmula booleana y el predicado $S(q)$, para demostrar que el lenguaje que reconoce 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 que se reconocen por G_{S-SAT}. Entonces es necesario demostrar que $L_{S-SAT} = L_{G_{S-SAT}}$.

  Para demostrar que $L_{S-SAT} \subseteq L_{G_{S-SAT}}$, sea una fórmula booleana satisfacible F y sea q su representación como cadena en el lenguaje $L_{FULL-SAT}$, entonces existe una cadena binaria w, con longitud igual a la cantidad de variables de F, que satisface a F.

  Como w satisface F entonces w pertenece al conjunto de cadenas que satisfacen a la primera cláusula de F. Esto significa que existe una secuencia de derivaciones desde el predicado $S(q)$ hasta $B(z_q, w)$ y como w satisface todas las cláusulas de F entonces $B(z_q, w)$ deriva en la cadena vacía por lo que q se reconoce por G_{S-SAT}, lo cual implica que $L_{S-SAT} \subseteq L_{G_{S-SAT}}$.

  Para completar la demostración es necesario demostrar que $L_{G_{S-SAT}} \subseteq L_{S-SAT}$. Sea una cadena q que se reconoce por G_{S-SAT} y sea F la fórmula booleana asociada a q, entonces existe una cadena binaria w tal que existe una secuencia de derivaciones desde $A(q)$ a $B(z_q, w)$ y de $B(z_q, w)$ a la cadena vacía, por tanto w satisface a la primera cláusula de F y a las restantes también, luego w satisface a F, por lo que F es satisfacible. Esto implica que $L_{G_{S-SAT}} \subseteq L_{S-SAT}$ y con esto se demuestra que $L_{G_{S-SAT}} = L_{S-SAT}$.

En la siguiente sección se presentan un ejemplo del reconocimiento de una cadena por 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 cómo 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 cómo 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 $aa\mathbf{d}ac\mathbf{d}cb\mathbf{d}$, la secuencia de derivaciones asociada a esta cadena en G_{S-SAT} es la siguiente:

1. $S(aa\mathbf{d}ac\mathbf{d}cb\mathbf{d}) \rightarrow A(aa\mathbf{d}ac\mathbf{d}cb\mathbf{d})$: c-1, $X = aa\mathbf{d}ac\mathbf{d}cb\mathbf{d}$
2. $A(aa\mathbf{d}ac\mathbf{d}cb\mathbf{d}) \rightarrow P(a\mathbf{d}ac\mathbf{d}cb\mathbf{d}, 1)$: c-2, $X = a\mathbf{d}ac\mathbf{d}cb\mathbf{d}$
3. $P(a\mathbf{d}ac\mathbf{d}cb\mathbf{d}, 1) \rightarrow P(\mathbf{d}ac\mathbf{d}cb\mathbf{d}, 10)$: c-9, $X = \mathbf{d}ac\mathbf{d}cb\mathbf{d}$ $Y = 1$
4. $P(\mathbf{d}ac\mathbf{d}cb\mathbf{d}, 10) \rightarrow B(ac\mathbf{d}cb\mathbf{d}, 10)$: c-14, $X = ac\mathbf{d}cb\mathbf{d}$ $Y = 10$
5. $B(ac\mathbf{d}cb\mathbf{d}, 10) \rightarrow C(ac, 10)B(cb\mathbf{d}, 10)$: c-21, $X_1 = ac$ $X_2 = cb\mathbf{d}$ $Y = 10$
6. $B(cb\mathbf{d}, 10) \rightarrow C(cb, 10)B(\varepsilon, 10)$: 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 en todas las instancias de los no terminales existe una asignación en rango que provoca que todos los predicados deriven en la cadena vacía, entonces $aa\mathbf{d}ac\mathbf{d}cb\mathbf{d}$ se reconoce por G_{S-SAT}. Esto coincide con el hecho de que $(x_1 \vee x_2) \wedge (x_1) \wedge (\neg x_2)$ es satisfacible, para la asignación de valores $x_1 = 1$ y $x_2 = 0$.

Seguidamente se presenta una cadena que representa una fórmula que no es satisfacible y por tanto la cadena asociada a dicha fórmula no se reconoce por G_{S-SAT}. La cadena asociada a $x_1 \wedge \neg x_1$ es $a\mathbf{d}b\mathbf{d}$, la secuencia de derivaciones asociada a esta cadena en G_{S-SAT} es la siguiente:

1. $S(a\mathbf{d}b\mathbf{d}) \rightarrow A(a\mathbf{d}b\mathbf{d})$: c-1, $X = a\mathbf{d}b\mathbf{d}$
2. $A(a\mathbf{d}b\mathbf{d}) \rightarrow P(\mathbf{d}b\mathbf{d}, 1)$: c-2, $X = \mathbf{d}b\mathbf{d}$
3. $A(a\mathbf{d}b\mathbf{d}) \rightarrow N(\mathbf{d}b\mathbf{d}, 0)$: c-3, $X = \mathbf{d}b\mathbf{d}$
4. $P(\mathbf{d}b\mathbf{d}, 1) \rightarrow B(b\mathbf{d}, 1)$: c-14, $X = b\mathbf{d}$ $Y = 1$

5. $B(b\mathbf{d},1) \rightarrow C(b,1)B(\varepsilon,1)$: 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 $N(\mathbf{d}b\mathbf{d},0)$ en la tercera derivación no deriva en la cadena vacía. Por otro lado el predicado $Cn(\varepsilon,\varepsilon)$ en la séptima 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 $a\mathbf{d}b\mathbf{d}$. Esto coincide con el hecho de que $x_1 \wedge \neg x_1$ no es satisfacible para ninguna asignación de valores a sus variables.

Como G_{S-SAT} reconoce las fórmulas booleanas satisfacibles para determinar si una fórmula es satisfacible se debe determinar si su cadena asociada es reconocida por G_{S-SAT}, por lo que el es necesario analizar 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.

En la primera fase se generan las cadenas binarias que representa la asignación de valores a las variables booleanas y estas cadenas participan en los predicados de fases posteriores, mediante las derivaciones de la gramática.

Si se analiza el algoritmo de reconocimiento descrito en [4] 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.

Como se pueden generar 2^n cadenas y cada cadena tiene n^2 rangos, la cantidad de rangos totales sería $n^2 2^n$, pero esta es una cota burda ya que para cada cadena no se utilizan todos los posibles rangos en el proceso de derivación de la gramática, por lo que la complejidad es mucho menor pero sigue siendo exponencial con respecto al tamaño de la cadena de entrada.

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 $O(2^n m^2)$.

En la siguiente sección se analiza una consecuencia directa de G_{S-SAT}, que demuestra que para todo problema en NP existe una RCG que reconoce el lenguaje que se asocia a dicho problema.

4.3. Clases de problemas que cubren las RCG

En [3] se menciona las RCG cubren todos los problemas de la clase P. Como se mostró en la sección anterior con la gramática G_{S-SAT} existe una RCG que reconoce el lenguaje de las fórmulas satisfacibles, 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 a las instancias polinomiales del SAT, usando gramáticas de concatenación de rango para determinar si una fórmula booleana asociada una de estas instancias polinomiales es satisfacible.

4.4. 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. $A(X_1 d X_2) \to A_0(X_1)A(X_2)$
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. Las cláusulas de la 4 a la 13 permiten contar la cantidad de a o b en una cláusula, o sea, la cantidad de literales de cada cláusula. Para esto se definen 4 estados: A_0, A_1, A_2 y A_3. A_0 representa que se reconocieron 0 a o b, A_1 representa que se reconocieron una a o b, A_2 representa que se reconocieron 2 a o b y A_3 representa que se reconocen más de 2 a o b.

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.

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

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

Pero 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 las RCG cubren todos los problemas de la clase P, entonces es posible diseñar una RCG para el 2-SAT y para cada instancia polinomial del SAT, cuyo problema de la palabra sea polinomial.

En este capítulo se construyó el lenguaje L_{S-SAT}, mediante una RCG, lo que demuestra que no es necesario el transductor T_{SAT} para construir L_{S-SAT}.

Por otro lado G_{S-SAT} demuestra que todos los problemas en NP se reconocen por una RCG y se presenta un primer acercamiento para describir los problemas SAT polinomiales mediante un RCG, dejando abierto el problema de encontrar una RCG que permita reconocer el 2-SAT y el problema de la palabra para esta RCG sea polinomial.

Conclusiones

Conclusiones

Recomendaciones

Recomendaciones

Referencias

[1] Alina Fernández Arias. «El problema de la satisfacibilidad booleana libre del contexto». En: (2007) (vid. págs. 3, 5, 14, 15).

[2] 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. 20, 21).

[3] 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. 45).

[4] 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. 16, 21, 35, 44).

[5] Cristian Calude, Kai Salomaa y Tania Roblot. «Finite-State Complexity and the Size of Transducers». En: *Electronic Proceedings in Theoretical Computer Science* 31 (ago. de 2010), págs. 38-47. ISSN: 2075-2180. DOI: 10.4204/eptcs.31.6. URL: http://dx.doi.org/10.4204/EPTCS.31.6 (vid. pág. 7).

[6] Steven Halim y Felix Halim. *Competitive Programming 3: The New Lower Bound of Programming Contests*. 2-SAT Problem. Lulu.com, 2013, págs. 336-337 (vid. pág. 13).

[7] John E. Hopcroft, Rajeev Motwani y Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. 3rd. Addison-Wesley, 2006. ISBN: 9780321455369 (vid. págs. 2, 3, 5-7, 9-13, 26).

[8] Manuel Aguilera López. «Problema de la Satisfacibilidad Booleana de Concatenación de Rango Simple». En: (2016) (vid. págs. 3, 15, 19).