Universidad de La Habana
Facultad de Matemática y Computación
Especialidad de Ciencias de la Computación

Trabajo de Diploma en Opción al Título de Licenciado en Ciencia de la Computación.

El Problema de la Satisfacibilidad Booleana Libre del Contexto. Un Algoritmo Polinomial.

Autor: Alina Fernández Arias

Tutor: MSc. Fernando Rodríguez Flores.

Ciudad de La Habana

Junio 2007

A mi papá,

Por ser mi papá.

Agradecimientos

A todas las personas que contribuyeron al desarrollo de este trabajo, directa o indirectamente. A los que fueron parte de los últimos cinco años.

De manera muy especial a:

A mi tutor, por todo.

Al claustro de profesores de la facultad.

A las personas que más han visto este trabajo en los últimos meses y con las que he compartido el tutor: Carmen, Raisa y Tatiana.

A mi grupo de estudio y de fiestas de fin de año: Eddy, Sandro, Voro, Yiselt, Gonzalo, Alain, Reynel, Yeny, Bolu, Hayden, Fernando.

A los que cuestionaron el trabajo y ayudaron a hacerlo mejor: Abelito, Ernesto y Bruno.

A mi familia por su apoyo incondicional y sobre todo a mi papá que dijo que era mejor estudiar Ciencia de la Computación en la Universidad de La Habana y no Informática en la CUJAE.

A Osqui, que ha estado cuando hace falta y que siempre dice que todo va a salir bien.

A los que mencioné y a los que no:

MUCHAS GRACIAS

Resumen

El problema de la satisfacibilidad booleana consiste en determinar si existe alguna interpretación verdadera de una fórmula lógica dada. Cuando la fórmula no tiene ninguna restricción es un problema NP-Completo. EL objetivo de este trabajo es presentar un algoritmo polinomial para resolver un caso particular del problema de la satisfacibilidad booleana: cuando las variables que aparecen en la fórmula cumplen una restricción de orden que se denomina **Orden Libre del Contexto**. El algoritmo que se propone construye, dada una fórmula lógica, una gramática libre del contexto que genera el lenguaje de todas la interpretaciones verdaderas de la fórmula de entrada. Para determinar si la fórmula es satisfacible, se verifica si el lenguaje generado por la gramática es vacío o no. En caso de ser vacío la fórmula no es satisfacible. En caso contrario se obtiene, como resultado del algoritmo, no solo que la fórmula es satisfacible, sino además una representación polinomial del lenguaje que contiene a todas sus interpretaciones verdaderas y la cantidad de cadenas que pertenecen a dicho lenguaje.

Tabla de contenido

Introducción 4
Capítulo 1 El problema de la satisfacibilidad booleana. 6
1.1 Preliminares. 6
1.1.1 Definición de los operadores lógicos. 7
1.1.2 Problema de la Satisfacibilidad Booleana. 7
1.2 Aplicaciones del problema SAT. 8
1.3 Algoritmos clásicos para resolver el problema SAT. 8
1.4 Complejidad Computacional. 9
1.5 Clases Solubles de SAT en tiempo polinomial. 12
1.6 Problema de la Satisfacibilidad Booleana Libre del Contexto. 13
Capítulo 2 Teoría de Lenguajes Formales. 15
2.1 Elementos básicos de la teoría de lenguajes formales. 15
2.2 Gramáticas. 15
2.2.1 Notación BNF. 16
2.3 Clasificación de la Gramáticas. Jerarquía de Chomsky. 16
2.4 Árboles de Derivación para Gramáticas Libres del Contexto. 17
2.5 Autómatas. 18
2.6 Árbol de Sintaxis Abstracta. 20
Capítulo 3 De la fórmula de entrada a la gramática libre del contexto. 21
3.1 Lenguaje Satisfacible. 21
3.2 Autómata Booleano. 22
3.3 Autómata Booleano Extendido. 30
3.4 Optimización del Autómata Booleano Extendido. 34
3.4.1: Algoritmo para eliminar las transiciones al estado *F*. 34
3.5 Mecanismo de Memoria. 35
3.6 Orden Libre del Contexto. 36
3.6.1: Algoritmo para determinar si una secuencia de variables tiene un orden libre del contexto. 36
3.7 Autómata de Pila. 37
3.8 Gramática Libre del Contexto. 40
3.8.1: Algoritmo para construir una gramática libre del contexto asociada a un autómata de pila booleano. 42

3.8.2 Problema del Vacío. 47
3.8.3: Limpieza de la gramática. 49
3.8.4: Contar la cantidad de soluciones. 49
3.9 K_SAT Libre del Contexto. 51
Capítulo 4 Detalles de Implementación. 55
4.1 Análisis Lexicográfico y Sintáctico. 56
4.2 Árbol de Sintaxis Abstracta. 58
4.3 Análisis Semántico: Verificar la restricción de orden libre del contexto. 59
4.4 Generación de Código: Construcción del autómata booleano extendido. 60
4.5 Autómata Booleano Extendido. 60
4.6 La gramática, los no terminales y las reglas. 61
4.7 Resultados Computacionales. 63
4.7.1 Generador de Casos de Prueba. 63
4.7.2 Resultados. 63
Conclusiones 64
Trabajos Futuros 64
Bibliografía y Referencias 65

Introducción

El problema de la satisfacibilidad booleana consiste en determinar si existe alguna interpretación verdadera para una fórmula lógica dada. Este problema está estrechamente relacionado con la lógica proposicional. Los resultados de la lógica proposicional bivalente tienen sus mayores aplicaciones en el desarrollo, construcción y diseño de circuitos lógicos digitales. Sin embargo, la importancia de éste problema va más allá del diseño de circuitos lógicos. Una de las áreas, dentro de la computación teórica, donde se estudia el problema de la satisfacibilidad booleana es la Teoría de la Complejidad Computacional. La importancia del problema de la satisfacibilidad booleana dentro de la teoría de la complejidad computacional viene dada porque este problema pertenece a la clase NP-Completo y, si se encuentra un algoritmo polinomial capaz de resolver un problema que pertenece a la clase NP-Completo, entonces todos los problemas de la misma pudieran ser resueltos en tiempo polinomial.

Son muchos los problemas NP que responde a situaciones de la vida real por lo que se hace necesario buscar alternativas para solucionarlos. Una de las vías más explotadas es la solución del problema mediante heurísticas. Otra alternativa es la solución de casos particulares. El algoritmo que se presenta, es un algoritmo polinomial para resolver un caso particular del problema de la satisfacibilidad booleana que se denomina **Problema de la Satisfacibilidad Booleana Libre del Contexto**. Para este caso, la restricción que se impone es referente al orden de las variables de la fórmula lógica, a diferencia de otras clases del problema de la satisfacibilidad booleana solubles en tiempo polinomial que parten de una fórmula en forma normal conjuntiva, donde las cláusulas tienen que cumplir un conjunto de restricciones; por ejemplo, para el problema 2 – SAT, la restricción que se impone es que todas las cláusulas tengan a lo sumo dos literales.

Para determinar si una fórmula lógica libre del contexto es satisfacible se construye una representación polinomial del lenguaje de todas sus interpretaciones verdaderas denominado $\hat{L}$. Este lenguaje se representa mediante una gramática libre del contexto $\hat{G}$ tal que $L(\hat{G}) = \hat{L}$. Si $L(\hat{G}) = \emptyset$ entonces la fórmula no es satisfacible; en caso contrario, se obtiene además, la gramática que genera todas las interpretaciones verdaderas de la misma y la cantidad de cadenas que pertenecen al lenguaje generado por la gramática.

Sea *A* una fórmula libre del contexto, la gramática que genera el lenguaje de todas sus interpretaciones verdaderas se construye a partir de un autómata de pila *P*, que reconoce, por el criterio de pila vacía, todas las las interpretaciones verdaderas de la fórmula *A*. El autómata de pila *P* se construye a partir de la unión de un autómata finito determinista que reconoce el lenguaje de todas las interpretaciones verdaderas de *A* asumiendo que todas las instancias de variables corresponden a variables distintas y una pila que garantiza que todas las instancias de una misma variable tomen el mismo valor. Determinar si el lenguaje generado por una gramática libre del contexto es vacío tiene un costo polinomial en la

longitud de la gramática. La longitud de una gramática es la cantidad de producciones de la misma.

El presente trabajo está dividido en cuatro capítulos. En el primer capítulo se presenta el problema de la satisfacibilidad booleana, se referencian algunas clases de este problema que son solubles en tiempo polinomial y algunas de las aplicaciones de este problema. Se presenta una pequeña introuducción a la teoría de la complejidad computacional y se explica la importancia del problema de la satisfacibilidad booleana dentro de la misma. Entre las clases solubles del problema de la satisfacibilidad booleana que se presentan en este capítulo está el problema de la satisfacibilidad booleana libre del contexto, que es precisamente el caso particular del problema de la satisfacibilidad booleana que se expone en este trabajo.

En el capítulo 2 se realiza una introducción a la teoría de lenguajes formales, presentando dos de las herramientas principales que brinda la teoría de lenguajes formales para representar lenguajes de forma compacta: los autómatas y las gramáticas.

En el tercer cápitulo se propone un algoritmo polinomial para resolver el problema de la satisfacibilidad booleana libre del contexto. Como resultado del algortimo se obtiene, además de si la fórmula es satisfacible o no, la cantidad de interpretaciones de la fórmula que son verdaderas y, la gramática que genera el lenguaje de todas las interpretaciones verdaderas de la fórmula de entrada. Lo más significativo de esta clase de problemas es que no parten de una fórmula en fórma normal conjuntiva. Aunque, si la fórmula de entrada está en forma normal conjuntiva, se resuelve como un caso particular del problema libre del contexto.

En el último capitulo se referencian algunos detalles de la implementación computacional del algoritmo propuesto.

Capítulo 1 El problema de la satisfacibilidad booleana.

Un problema muy importante dentro de la lógica proposicional bivalente es determinar si una fórmula es consistente o no, es decir, si es o no satisfacible. Este problema, dada su gran importancia dentro de la ciencia de la computación ha pasado a ser conocido como problema de la Satisfacibilidad Booleana.

Aunque es inmensa la importancia del problema de la Satisfacibilidad Booleana dentro de la lógica proposicional bivalente, su relevancia, dentro de la computación teórica, está dada por pertenecer a la clase de problemas NP-Completos.

Dos de las áreas fundamentales de la computación téorica son la Teoría de la Computabilidad y la Teoría de la Complejidad Computacional. La Teoría de la Computabilidad tiene como objetivo determinar si existe un procedimiento compuesto por un número finito de pasos para resolver un problema determinado. Por su parte, la Teoría de la Complejidad Computacional analiza los recursos necesarios para resolver el problema mediante un algoritmo determinado. La diferencia fundamental entre estas dos áreas radica en el hecho de que la primera tiene como objetivo encontrar un procedimiento mecánico para computar la solución de un problema; mientras que la segunda analiza si es factible resolverlo teniendo en cuenta los recursos necesarios para ello. El estudio del problema de la Satisfacibilidad Booleana se enmarca, fundamentalmente, dentro de la Teoría de la Complejidad Computacional.

1.1 Preliminares.

A continuación se definen los elementos necesarios para formular, matemáticamente, el problema de la satisfacibilidad booleana.

Definición 1.1.1: Una **variable booleana** es una variable binaria que toma valores del dominio {Verdadero, Falso}.

Siempre que no cause ambigüedad, se llamará "variable" a toda "variable booleana".

Definición 1.1.2: Una **fórmula lógica** se define recursivamente como sigue:

I. Toda variable es una fórmula.
II. Si A es una fórmula entonces $\neg A$ y (A) son fórmulas.
III. Si A y B son fórmulas entonces $A \wedge B$ y $A \vee B$ son fórmulas.
IV. No existen otras fórmulas que no sean las definidas por las reglas I, II y III.

De manera general en una fórmula lógica una variable aparece en más de una ocasión.

Como a las únicas fórmulas que se hace referencia en este documento son a las fórmulas lógicas, en lo adelante, siempre que se haga referencia a fórmula, se está haciendo referencia a fórmula lógica.

Definición 1.1.3: Sea x_i una variable que ocurre en una fórmula lógica A; entonces cada una de sus ocurrencias en A se denomina **instancia de** la **variable** x_i.

A partir de esta definición se puede redefinir la primera regla de formación de fórmulas como: Toda variable o instancia de ésta es una fórmula.

Definición 1.1.4: Una **interpretación** de una fórmula lógica es una asignación de valores de verdad a cada una de sus variables. Los posibles valores de verdad son Verdadero ó 1 y Falso ó 0.

Una interpretación de una fórmula *A*, determina un valor veritativo de *A*.

Definición 1.1.5: Una **fórmula** lógica es **satisfacible** si existe alguna interpretación se sus variables que la haga verdadera.

1.1.1 Definición de los operadores lógicos.

Definición 1.1.1.1: **Operador Negación.** Cambia el valor de verdad a la fórmula a la cual se aplica. Sea *A* una fórmula, $\neg A$ su negación y *U* una interpretación de *A*. Si *A* toma valor verdadero para *U* entonces $\neg A$ toma valor falso y viceversa.

Definición 1.1.1.2: **Operador Conjunción.** Sean *A* y *B* dos fórmulas lógicas, la fórmula $A \wedge B$ es verdadera para una interpretación *U* si y sólo si *A* y *B* son verdaderas para dicha interpretación; en cualquier otro caso $A \wedge B$ es falsa.

Definición 1.1.1.3: **Operador Disyunción.** Sean *A* y *B* dos fórmulas lógicas, la fórmula $A \vee B$ es verdadera para una interpretación *U* si al menos una de ellas es verdadera para dicha interpretación. En el único caso en que $A \vee B$ es falsa para *U* es cuando *A* y *B* son falsas para *U*.

1.1.2 Problema de la Satisfacibilidad Booleana.

Definición 1.1.2.1: Un **literal** es una variable o su negación.

Definición 1.1.2.1: Una **cláusula** es una disyunción de literales.

Definición 1.1.2.3: Una fórmula está en **forma normal conjuntiva** si es de la forma: $A_1 \wedge A_2 \wedge \ldots \wedge A_n$ donde cada $A_i, i = 1, \ldots, n$, es una cláusula.

Sea *A* una fórmula lógica, el problema de la satisfacibilidad booleana consiste en determinar si existe alguna asignación de valores de verdad que satisfagan a la fórmula *A*. Cuando la fórmula aparece en forma normal conjuntiva, el problema de la satisfacibilidad booleana se conoce como problema SAT. [2]

Definición 1.1.2.4: Dos **fórmulas** son **equivalentes** si y sólo si para una misma interpretación tienen el mismo valor de verdad.

Una consecuencia inmediata de de definición anterior es que si las fórmulas *A* y *B* son equivalentes entonces *A* es satisfacible si y sólo si *B* es satisfacible.

Teorema 1.1.2.1: Para toda fórmula lógica se puede construir una equivalente en forma normal conjuntiva.

Demostración: Véase [11].

Una instancia del problema SAT tiene dos elementos fundamentales:

1. Un conjunto de *n* variables $X = \{x_1, x_2, \dots, x_n\}$.
2. Un conjunto de *m* cláusulas distintas $C = \{C_1, C_2, \dots, C_m\}$

El objetivo del problema SAT es determinar si existe una asignación de valores de verdad para las variables del conjunto *X* tal que la fórmula en forma normal conjuntiva $C_1 \wedge C_2 \wedge \dots \wedge C_m$ sea verdadera.

1.2 Aplicaciones del problema SAT.

A continuación se mencionan algunas de las aplicaciones del problema SAT. [8]

Matemática:

- Criptografía Matemática.
- Resolución de cualquier problema de la clase NP – Completo.

Ciencia de la Computación e Inteligencia Artificial

- Razonamiento con restricciones.
- Programación lógica.

Diseño automático de circuitos integrados:

- Modelación de circuitos
- Verificación

Diseño de arquitecturas de computadoras:

- Optimización del conjunto de instrucciones.
- Diseño de circuitos aritmético – lógicos.
- Sistemas de tiempo análisis de la consistencia de flujo de datos.

1.3 Algoritmos clásicos para resolver el problema SAT.

Los algoritmos clásicos para resolver el problema SAT, a partir de fórmula en forma normal conjuntiva, construyen todas las posibles interpretaciones de la fórmula hasta llegar a una que sea verdadera, es ese caso dan como respuesta que la formula es satisfacible; si el algoritmo no encuentra ninguna asignación de valores de verdad que evalue verdadera la fórmula entrada entonces retorna que la fórmula no es satisfacible. En el caso peor, estos algoritmos tienen que construir todas las interpretaciones posibles de la fórmula de entrada, lo que es del orden de 2^m donde *m* es la cantidad de variables distintas que ocurren en la misma.

Un algoritmo para SAT es sano si para cada entrada en la que el procedimiento retorna **sí**, la fórmula es satisfacible y es completo, si además retorna **sí** para toda entrada satisfacible.

A continuación se presenta el algoritmo DPLL desarrollado por Davis, Putnam, Logemann y Loveland en 1960 [2, 6]. Este algoritmo hace una "búsqueda" de una interpretación que satisfaga la fórmula, en el conjunto de las interpretaciones posibles. Este es un algoritmo completo para el problema SAT.

Sea *A* una fórmula lógica escrita como conjuncion de disyunción de literales, DPLL hace una búsqueda de una interpretacion que satisfaga a *A* en el cojunto de las posibles interpretaciones de *A*. Se denomina espacio de búsqueda al espacio virtual en el que un algoritmo busca una solución a un problema. Una posible representación del espacio de búsqueda para un problema SAT es mediante un árbol donde los nodos están asociados a las variables de la fórmula y las aristas se etiquetan con los valores de verdad que toma la variable correspondiente al nodo de donde parten. Un camino desde la raíz hasta una hoja constituye una interpretación de la fórmula.

El DPLL construye parcialmente el espacio de búsqueda hasta llegar a una interpretación verdadera de la fórmula. Si hay una cláusula en *A* que es un literal, necesariamnete hay que asignar a la variable del literal el valor de verdad que lo haga verdadero, a continuación se sustituye la variable por su valor en todas sus ocurrencias en *A*, se *simplifica A* y se continúa aplicando DPLL a la simplificación de *A*. Como consecuencia de este paso las aristas que correspondan a un valor de verdad distinto del que se le asignó a la variable del literal jamás serán generadas. Si no hay cláusula unitaria en *A*, entonces se *selecciona* una variable de las que ocurre en *A*, que aún no haya sido analizada y se sustituye por un valor veritativo en todas sus ocurrencias en *A*, se *simplifica A* y se aplica DPLL a la simplificación de *A* denotada por S_A. Si S_A es satisfacible el algortimo termina. Si S_A no es satisfacible, entonces se sustituye a la variable en *A* por el otro valor veritativo, se simplifica nuevamente *A* en S_A y se aplica DPLL a S_A. Si no se encuentra ninguna interpretación verdadera de *A* entonces se dice que *A* no es satisfacible.

El procedimiento *selecciona*, escoge una variable del conjunto de variables de la fórmula *A* y luego de la selección la elimina del conjunto. El resultado de este procedimiento es la variable seleccionada.

El procedimiento *simplifica,* sustituye una variable por un valor de verdad y aplica las leyes de la lógica proposicional.

Otro de los métodos clásicos para resolver el problema SAT son los métodos incompletos. Estos métodos no pueden decir cuando una fórmula es no satisfacible, pero son mucho mejores que los métodos completos, como el DPLL, para encontrar una interpretación verdadera si ésta existe. Estos métodos emplean una especie de búsqueda local aleatoria. [2]

El problema SAT, juega un papel importante dentro de la complejidad computacional. En la siguiente sección se introducen algunos elementos de la Teoría de Complejidad Computacional que permiten comprender la importancia de resolver casos particulares del problema de la satisfacibilidad boolena en tiempo polinomial.

1.4 Complejidad Computacional.

La Teoría de la Complejidad computacional estudia los recursos necesarios para resolver un problema. El estudio del problema de la satisfacibilidad booleana se enmarca dentro de esta área. Dentro de teoría de la complejidad computacional, los recursos comúnmente estudiados son el espacio y el tiempo.

Definición 1.4.1: La **ecuación de complejidad computacional** es una función *f* que para todo valor *n* proporciona el tiempo que necesita un procedimiento *P* para encontrar una solución de una instancia *I* de tamaño *n*.

Definición 1.4.2: Sean $f(n)$ y $g(n)$ funciones, ***f* pertenece a *O* grande de *g*** y se denota por $f(n) \in O(g(n))$ o de forma abreviada $f \in O(g)$ si existe una constante c y un entero n_0 tal que $f(n) < c * g(n)\ \forall\ n \geq n_0$.

Definición 1.4.3: Un algoritmo es de **complejidad polinomial** si $f(n) \in O(p(n))$, donde p es un polinomio.

Un algoritmo es eficiente si es de complejidad polinomial. La complejidad de un problema es la complejidad del menor de los algoritmos que lo resuelven. El tiempo de ejecución de un algoritmo depende del tamaño de los datos. Actualmente las máquinas de cómputo pueden resolver problemas mediante algoritmos que tienen una complejidad o costo polinomial del tamaño de la entrada, estos problemas se agrupan en la clase *P*.

Un problema se puede definir formalmente como una relación binaria entre un conjunto de instancias del problema y un conjunto de soluciones del mismo. Por simplicidad, la Teoría de la Complejidad Computacional restringe su atención a los problemas de decisión. Un problema de decisión es un problema que para cada entrada da como salida una respuesta positiva o negativa; por ejemplo, el problema *"Es Satisfacible A"* se puede escribir como: Dada una fórmula *A*, responder **sí,** la fórmula es satisfacible o **no,** la fórmula no es satisfacible.

Definición 1.4.4: Un **problema de decisión** p consiste en un conjunto de instancias del problema D_p y un subconjunto de instancias del problema Y_p, $Y_p \subseteq D_p$, tal que la respuesta para cada instancia del conjunto Y_p es positiva.

Todo problema de decisión es equivalente a un lenguaje formal. El lenguaje equivalente a un problema de decisión dado es el conjunto de instancias del problema para el cual la respuesta es positiva.

Los problemas de decisión se clasifican en conjuntos de complejidad comparables llamados clases de complejidad.

Definición 1.4.5: Una **clase de complejidad** es el conjunto de problemas de decisión que pueden ser resueltos por una máquina *M* empleando $O(f(n))$ del recurso *R* (*n* es el tamaño de la entrada).

Como modelo de cómputo se emplea una máquina de Turing.

Definición 1.4.6: Una **máquina de Turing** es un quíntuplo $M = (Q, \Sigma, f, s, F)$ donde:

- Q: Conjunto de estados.
- Σ: Alfabeto.
- f: Función de transición. $f: Q \times \Sigma \longrightarrow \mathrm{Q} \times \Sigma \times \mathrm{D}$. *D* es el conjunto de posibles acciones. $D = \{izquierda,\ derecha,\ en\ el\ lugar\}$.
- s: Estado inicial de la máquina.
- F: Conjunto de estados finales. $F \subseteq Q$.

Definición 1.4.7: Sea $M = (Q, \Sigma, f, s, F)$ una máquina de Turing, un par $(q, w) \in Q \times \Sigma^*$ es una **configuración** de *M*.

Definición 1.4.8: Sea $M = (Q, \Sigma, f, s, F)$ una máquina de Turing, un **movimiento** en M es un cambio de configuración. Notación $(q, \alpha w) \vdash (p, \beta w)$, donde $q, p \in Q, \alpha, \beta \in \Sigma, w \in \Sigma^*$.

Los cambios de configuración posibles son los que están determinados por la función de transición. $(q, \alpha w) \vdash (p, \beta w)$ si y sólo si $f(q, \alpha) = (p, \beta)$.

La función de transición es un programa para la máquina de Turing. Esta función especifica para cada par $(q, \alpha) \in Q \times \Sigma$ una terna que representa la próxima acción a realizar por la máquina: $f(q, \alpha) = (p, \beta, d)$ donde $p \in Q, \beta \in \Sigma,\ d \in D$.

Definición 1.4.9: Una **máquina de Turing es determinista** si a partir de una configuración c_1 solo se puede llegar a una configuración c_2. Una **máquina de Turing es NO determinista** si a partir de una configuración $(q, \alpha w)$ se puede llegar a cualquier configuración *c* tal que $c \in f(p, \alpha)$.

Se dice que una máquina de Turing *M* acepta una cadena *w* si a partir de *w* llega a un estado final. El lenguaje aceptado por una máquina de Turing $M = (Q, \Sigma, f, s, F)$ es:

$$L(M) = \{w \in \Sigma^* \ tal\ que\ M\ acepta\ a\ w\}$$

Sea *M* una máquina de Turing. Se denota por $t_M(w)$ al número de pasos que realiza *M* para computar una entrada *w*. Si el proceso de reconocimiento nunca termina entonces $t_M(w) = \infty$. Para *n* que pertenece a los naturales, se denota por $T_M(n)$ el mayor tiempo de ejecución que necesita *M* para computar una entrada *w* de tamaño *n*, definido como: $T_M(n) = \max\{t_M(w) / w \in \Sigma^n\}$, donde Σ^n es el conjunto de cadenas de longitud *n* sobre el alfabeto Σ. El tiempo de ejecución de *M* es polinomial si existe *k* para todo *n* tal que: $T_M(n) \leq n^k + k$.

Definición 1.4.10: La **clase de complejidad *P*** es el conjunto de los problemas de decisión que pueden ser resueltos en una máquina de Turing determinista en tiempo polinomial.

Definición 1.4.11: La **clase de complejidad *NP*** es el conjunto de los problemas de decisión que pueden ser resueltos por una máquina de Turing **no** determinista en tiempo polinomial. Esta clase incluye muchos problemas que se desean resolver en la práctica como el *Problema de la Satisfacibilidad Booleana* y el Problema del Viajante.

De la definición se deduce que $P \subseteq NP$. Surge la pregunta ¿*P = NP*? Este es un problema abierto aunque se conjetura que $P \neq NP$. Esta pregunta ha introducido conceptos como *NP-Completo*. Los problemas que pertenecen a la clase *NP-Completo* son los problemas más difíciles de la clase *NP*. Su principal característica es que si se encuentra una solución en tiempo polinomial para algún problema de este conjunto, entonces existe una solución en tiempo polinomial para todos los problemas de la clase *NP-Completo*.

¿Cómo se prueba que un problema *p* pertenece a la clase *NP- Completo*?

En 1971 Cook publicó un teorema en el que demostró que el problema SAT es *NP- Completo* [7]. La idea general de la demostración del teorema de Cook es tomar un problema de decisión genérico $\pi \in NP$ y una instancia genérica del problema d, $d \in D_\pi$. Se resuelve la instancia *d* empleando la máquina de Turing no determinista hipotética que resuelva π. Luego se construye en tiempo polinomial una fórmula lógica $\varphi_{\pi,d}$ tal que $d \in Y_\pi$ si y sólo si $\varphi_{\pi,d}$ es satisfacible.

Definición 1.4.12: Sean p, r dos problemas de decisión. Decimos que $f: D_r \rightarrow D_p$ es una **reducción polinomial** de r en p si f se computa en tiempo polinomial y para todo $d \in D_r$ se cumple que $d \in Y_r \Leftrightarrow f(d) \in Y_p$.

A partir de la demostración de Cook, para probar que un problema $p \in NP - Completo$:

1. Se muestra que $p \in NP$.
2. Se elige convenientemente un problema $r \in NP - Completo$
3. Se construye una reducción polinomial f de r en p.

La clase de problemas *NP- Completos* incluye muchos problemas de gran aplicación práctica. La importancia de resolver casos particulares de un problema *NP-Completo* es que estas soluciones se pueden trasladar a casos particulares de otros problemas *NP-Completos*.

1.5 Clases Solubles de SAT en tiempo polinomial.

Existen problemas que pueden ser resueltos en teoría pero no en la práctica, a estos problemas se les llama problemas intratables. En general solo los problemas que tienen soluciones en tiempo polinomial son solubles para la mayoría de los casos. Entre los problemas intratables se incluyen los de tiempo exponencial. Si $P \neq NP$ entonces todos los problemas NP son también intratables. La razón por la que las soluciones de costo exponencial no son útiles en la práctica es porque si se tiene un problema que requiere 2^n operaciones para su resolución, donde n es el tamaño de la entrada; para $n = 100$, asumiendo que una computadora puede realizar 10^{10} operaciones por segundo, computar una solución tardaría cerca de $4 * 10^{12}$ años completarse, lo que es mucho más que la edad actual del universo.

El problema SAT es un problema que pertenece a la clase NP–Completo y parece improbable que se pueda encontrar un algoritmo polinomial para resolverlo [2]. Dada su importancia, dentro de la ciencia de la computación, se han realizado estudios de algunos casos particulares de este problema. Las principales investigaciones realizadas son a partir de fórmulas en forma normal conjuntiva, obteniéndose clases del problema SAT solubles en tiempo polinomial para aquellas fórmulas en las que sus cláusulas cumplen un conjunto de restricciones.

A continuación se referencian algunas de las subclases del problema SAT que en estos momentos tienen solución en tiempo polinomial:

- 2 – SAT: Una fórmula en forma normal disyuntiva es 2 – SAT si todas las cláusulas tienen a lo sumo dos literales.[8]
- Horn: Una fórmula en forma normal disyuntiva es Horn si todas las cláusulas tienen a lo sumo un literal positivo. Un literal positivo es un literal donde la variable no está negada. Existen otras clases derivadas de Horn que también son solubles en tiempo polinomial. [8]
- PLUR: La clase PLUR utiliza el concepto de literal eliminable y el hecho de que una fórmula proposicional es equivalente a la obtenida eliminando sus literales eliminables.[12]

Otra de las clases del problema de la satisfacibilidad, soluble en tiempo polinomial, es la que agrupa a las fórmulas lógicas libres del contexto, esta es la clase de problemas de la satisfacibilidad que se expone en este trabajo. En este caso, a diferencia de los mencionados

anteriormente donde se parte de una fórmula en forma normal conjuntiva con restricciones para sus cláusulas, la restricción que se impone es sobre el orden que tienen que seguir las variables de la fórmula, la fórmula no necesariamente tiene que estar en forma normal conjuntiva.

1.6 Problema de la Satisfacibilidad Booleana Libre del Contexto.

El problema de la satisfacibilidad booleana libre del contexto consiste en determinar si existe alguna interpretación verdadera de una fórmula lógica. La restricción que se impone a la fórmula es referente a orden que tiene que seguir las variables que aparecen en ellas. A continuación se realizan algunas definiciones necesarias para formalizar este problema matamáticamente.

Definición 1.6.1: Una **secuencia de instancias de variables** se define recursivamente como sigue:

i. Si xi_k es una instancia de la variable xi, entonces xi_k es una secuencia de instancias de variables.
ii. Si S_1 y S_2 son secuencias de instancias de variables, entonces la secuencia formada por todas las instancias de variables de S_2 puestas a continuación de todas las instancias de variables de la secuencia S_1, es una secuencia de instancias de variables.
iii. No existen otras secuencias de instancias de variables que no sean las formadas por las reglas i y ii.

Definición 1.6..2: La secuencia de instancias de variables que se obtiene de eliminar de una fórmula lógica todos los operadores y paréntesis se denomina **Secuencia de Variables asociada a una Fórmula Lógica**.

Cuando no cause ambigüedad, se llamará "secuencia de variables" a toda secuencia de instancias de variables.

Definición 1.6.3: La **longitud de una fórmula lógica** es la cantidad de instancias que tiene su secuencia de variables.

Por ejemplo: Sea la fórmula $A = x_1 \vee x_2 \wedge (x_3 \vee x_2) \wedge x_1$. La secuencia de variables de la fórmula A es $x_1\ x_2\ x_3\ x_2\ x_1$. Por tanto, la longitud de A es 5.

Definición 1.6.3: Una secuencia de variables tiene un **Orden Libre del Contexto** si se cumplen las siguientes condiciones:

1. Si hay una instancia de la variable $\mathbf{x_j}$ entre dos instancias consecutivas de la variable $\mathbf{x_i}$, entonces todas las instancias de $\mathbf{x_j}$ deben aparecer entre esas mismas dos instancias de la variable $\mathbf{x_i}$.
2. La condición 1 se cumple para todo par de variables $\mathbf{x_i}$ y $\mathbf{x_j}$ en la secuencia de variables.

Definición 1.6.4: Una **fórmula lógica es libre del contexto** si su secuencia de variables tiene un orden libre del contexto.

El problema de la satisfacibilidad booleana libre del contexto consiste en determinar si una fórmula lógica libre del contexto es satisfacible.

La diferencia significativa entre el problema de la satisfacibilidad booleana libre del conxtexto y el resto de las clases solubles del problema SAT en tiempo polinomial es que en el problema libre del contexto no es necesario que la fórmula de entrada esté en forma normal conjuntiva, para este caso solo interesa que la secuencia de variables tenga un orden libre del contexto. Un resultado importante es que existen problemas 3-SAT que son libres del contexto. Si la fórmula está en forma normal conjuntiva el problema de la satisfacibilidad booleana libre del contexto se resuelve de una manera más eficiente.

Es importante destacar que para este tipo de problema lo único significativo es el orden que tienen las instancias de variables en la fórmula. Esto significa que es posible encontrar dos fórmulas lógicas equivalentes, una libre del contexto y otra no. Por ejemplo, la fórmula $(x_1 \vee x_2) \wedge (\neg x_2 \vee \neg x_1)$ es libre del contexto, pero la "misma" fórmula escrita de otra forma: $(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$ no lo es. En este trabajo no se aborda el problema de transformar una fórmula lógica cualquiera en una fórmula libre del contexto, por dos razones fundamentales:

1. No todas las fórmulas se pueden transformar en una fórmula equivalente libre del contexto.

2. En el caso que se tenga una fórmula lógica que se pueda transformar en una fórmula equivalente libre del contexto, el proceso de transformación es demasiado casuístico.

Para determinar si una fórmula *A,* libre del contexto, es satisfacible, se construye una representación polinomial del lenguaje de todas sus interpretaciones verdaderas y se determina si dicho lenguaje es vacío o no, todo en tiempo polinomial. En caso de que el lenguaje de todas las interpretaciones verdaderas no sea vacío la fórmula es satisfacible y además se pueden contar la cantidad de interpretaciones verdaderas de la misma. Para construir y representar el lenguaje todas las interpretaciones verdaderas de la fórmula *A* se emplean elementos de la Teoría de Lenguajes Formales. En el siguiente capítulo se presentan algunos elementos básicos de la teoría de lenguajes formales.

Capítulo 2 Teoría de Lenguajes Formales.

2.1 Elementos básicos de la teoría de lenguajes formales.

Definición 2.1.1: Un **alfabeto** es un conjunto finito no vacío de símbolos.

Definición 2.1.2: Una **cadena** es una sucesión finita de símbolos de un alfabeto. La cadena que no contiene ningún símbolo se denomina cadena vacía y se denota por el símbolo especial ε que no pertenece a ningún alfabeto.

Definición 2.1.3: Un **lenguaje** es un conjunto de cadenas sobre un alfabeto.

Notación:

Σ: Alfabeto.

Σ^*: Todas las cadenas sobre Σ.

ε: Cadena vacía.

Definición 2.1.4: Sea α una cadena sobre el alfabeto Σ. Se denomina **longitud de** la **cadena** α y se denota por $|\alpha|$ a la cantidad de símbolos que forman la cadena.

2.2 Gramáticas.

Definición 2.2.1: Una **gramática** es un cuádruplo $G = (N, \Sigma, P, S)$ donde:

N: Alfabeto de no terminales. Por convenio se denotan con letras mayúsculas.

Σ: Alfabeto de terminales. Por convenio se denotan con letras minúsculas.

$N \cap \Sigma = \emptyset$. No debe exister ningún símbolo que pertenezca a la vez al alfabeto de no terminales y al de terminales.

P: Conjunto finito de pares ordenados (α, β) donde α, β son cadenas tales que $\alpha \in (N \cup \Sigma)^* - \Sigma^*$ y $\beta \in (N \cup \Sigma)^*$ (α es una cadena de símbolos terminales y no terminales que contiene al menos un símbolo terminal y β es una cadena de símbolos terminales y no terminales). De manera general, para referirse a los elementos $(\alpha, \beta) \in P$, se emplea la notación $\alpha \rightarrow \beta$.

S: Símbolo especial del alfabeto de no terminales que se denomina símbolo de partida.

El conjunto de producciones constituye el elemento fundamental de la gramática, pues describe como se forman las cadenas del lenguaje. Las producciones constituyen un mecanismo de sustitución en una gramática.

Por ejemplo, si el par *(AB, CDE)* es una producción de la gramática *G* y una cadena *α*, generada a partir de *G*, contiene la subcadena *AB,* entonces se puede formar una nueva cadena *β* remplazando una ocurrencia de la subcadena *AB* por la subcadena *CDE*.

Definición 2.2.2: Sean $\gamma, \delta \in (N \cup \Sigma)^*$; existe una **derivación directa** de $\boldsymbol{\gamma}$ en $\boldsymbol{\delta}$ si $\gamma = \varphi_1 \alpha \varphi_2$ y $\delta = \varphi_1 \beta \varphi_2$ y en la gramática *G* está la regla $\alpha \to \beta$. Luego $\boldsymbol{\gamma}$ deriva directamente en $\boldsymbol{\delta}$ ($\gamma \Rightarrow \delta$) usando la regla $\alpha \to \beta$.

Definición 2.2.3: Si $\gamma \Rightarrow \varphi_0 \Rightarrow \varphi_1 \Rightarrow \cdots \Rightarrow \varphi_n \Rightarrow \delta$, se dice que $\boldsymbol{\gamma}$ deriva en $\boldsymbol{\delta}$ empleando *n* reglas*, con* $n \geq 0$*,* o que existe una **secuencia de derivaciones** de $\boldsymbol{\gamma}$ a $\boldsymbol{\delta}$. Abreviadamente se escribe $\gamma \Rightarrow^* \delta$. Si $n \geq 1$ se dice que la secuencia de derivaciones es no trivial y se denota por $\Rightarrow^+$.

Definición 2.2.4: Una **forma oracional** α es una cadena formada por usa secuencia de símbolos terminales y no terminales ($\alpha \in (N \cup \Sigma)^*$) de una gramática $G = (N, \Sigma, P, S)$ que cumple que $S \Rightarrow^* \alpha$.

Definición 2.2.5: Una **oración** es una forma oracional donde $\alpha \in \Sigma^*$.

Definición: Sea $G = (N, \Sigma, P, S)$ una gramática. El **lenguaje generado** por *G* se define como:

$$L(G) = \{\alpha \; tal \; que \; \alpha \in \Sigma^* \wedge S \Rightarrow^+ \alpha\}$$

2.2.1 Notación BNF.

Es usual emplear una notación especial para describir las reglas de las gramáticas llamada BNF (Backus – Naus – Form). En la notación BNF:

Los símbolos no terminales se representan entre ángulos (**"<" " >"**).

Se emplea el símbolo **::=** para las producciones, en lugar del símbolo →.

Para indicar que un elemento es opcional se encierra entre corchetes (**"[" "]"**).

Para indicar que un elemento se repite cero o más veces se encierra entre llaves (**"{" "}"**).

El símbolo **|** se emplea para agrupar reglas.

Por ejemplo si las producciones $A \to bB$ y $A \to cC$ son producciones válidas de una gramática *G* entonces su notación en BNF es la siguiente: $\langle A \rangle ::= b\langle B \rangle | c\langle C \rangle$.

2.3 Clasificación de la Gramáticas. Jerarquía de Chomsky.

Definición 2.3.1: Las gramáticas se clasifican de acuerdo a la forma de sus producciones. Sea $G = (N, \Sigma, P, S)$ una gramática, decimos que *G* es:

- **Regular**: si todas las producciones son de la forma: $A \to xB$ ó $A \to x$, donde $A, B \in N$ y $x \in \Sigma^*$. Si $\varepsilon \in L(G)$ entonces $S \to \varepsilon$ y S no puede aparecer en ninguna parte derecha. Esta definición de gramática regular corresponde con una gramática regular a

la derecha; para el caso de una gramática regular a la izquierda es el simétrico, definiéndose como una gramática regular una gramática que sea regular a la izquierda o a la derecha. Para toda gramática regular a la izquierda existe una equivalente regular a la derecha.[10]

- **Libre del Contexto**: si todas las producciones son de la forma: $A \rightarrow \alpha$, donde $A \in N$ y $\alpha \in (N \cup \Sigma)^*$.
- **Dependiente del Contexto**: si todas las producciones son de la forma: $\alpha \rightarrow \beta$, donde $|\alpha| < |\beta|$
- **Irrestrictas 2.3.3s**: si no tiene ninguna restricción.

Convenio: Si un lenguaje es generado por una gramática de *"tipo x"* entonces se dice que el lenguaje es de *"tipo x"*.

Los cuatro tipos de gramáticas definidos anteriormente son referenciados en la Jerarquía de Chomsky.

Figura 1: Jerarquía de Chomsky.

2.4 Árboles de Derivación para Gramáticas Libres del Contexto.

Sea $G = (N, \Sigma, P, S)$ una gramática libre del contexto, $\alpha \in L(G)$ y $S \Rightarrow \varphi_0 \Rightarrow \cdots \Rightarrow \varphi_n \Rightarrow \alpha$ es la derivación completa de α en *G*. Emplear una representación lineal para la derivación no siempre es lo más ilustrativo para caracterizar su estructura; razón por la cual, frecuentemente, se emplea una estructura arbórea.

Definición 2.4.1: Sea $G = (N, \Sigma, P, S)$ una gramática libre del contexto, β una forma oracional tal que $A \Rightarrow^* \beta$ es una derivación en *G*, entonces el árbol asociado a la derivación anterior (**árbol de derivación**) se define recursivamente como sigue:

1. La raíz del árbol asociado a la derivación $A \Rightarrow^* \beta$, es un nodo con etiqueta *A* y nombrado *A*.
2. Sea *C* un nodo interior, si $C \rightarrow B_1 B_2 \ldots B_k$ con $B_i \in (N \cup \Sigma)$, es una regla de la gramática que se emplea en la derivación $A \Rightarrow^* \beta$, entonces las etiquetas de los hijos del nodo *C* son $B_1, B_2, \ldots B_n$, de izquierda a derecha.

3. Si la regla $C \to \varepsilon$ pertenece a la derivación; entonces el nodo *C* tiene como hijo al nodo etiquetado con ε.

Los nodos interiores del árbol de derivación están etiquetados con símbolos no terminales, mientras que los nodos hoja, están etiquetados con símbolos terminales o con ε.

2.5 Autómatas.

Los autómatas son mecanismos reconocedores de lenguajes. En el presente documento se hace referencia a los autómatas finitos y a los autómatas de pila.

Definición 2.5.1: Un **autómata finito** es un quíntuplo $A = (Q, \Sigma, f, q_0, F)$ donde:

Q: conjunto de estados.

Σ: Alfabeto de entrada.

f: Función de transición.

q_0: Estado inicial.

F: conjunto de estados finales.

Definición 2.5.2: Una **configuración** de un autómata finito es el par $(q, w), q \in Q \ \wedge \ w \in \Sigma^*$.

Definición 2.5.3: La configuración (q_0, w) donde *w* es la cadena que se quiere determinar si pertenece o no al lenguaje que reconoce el autómata, es la **configuración inicial** del autómata.

Definición 2.5.4: La configuración $(q, \varepsilon), q \in F$ es la **configuración final** del autómata.

Definición 2.5.5: Un **movimiento** es un cambio de configuración. Sean (q_1, uw) y (q_2, w) configuraciones del autómata finito A, un posible movimiento es el paso de la configuración (q_1, uw) a la (q_2, w). Notación: $(q_1, uw) \vdash (q_2, w)$.

Definición 2.5.6: Sean $(q_1, u_1u_2 \dots u_n), \dots, (q_k, u_k \dots u_n)$ configuraciones del autómata. Si $(q_1, u_1u_2 \dots u_n) \vdash \cdots \vdash (q_k, u_k \dots u_n)$ se dice que existe una **secuencia de movimientos** de $(q_1, u_1u_2 \dots u_n)$ a $(q_k, u_k \dots u_n)$. Notación: $(q_1, u_1u_2 \dots u_n) \vdash^* (q_k, u_k \dots u_n)$.

Definición 2.5.7: La **función de transición** de un autómata finito *A* es una función del par formado por los estados y el alfabeto de entrada en el conjunto de estados: $f: Q \times \Sigma \to$ Q. Para los autómatas deterministas a partir de un estado $q \in Q$ leyendo el símbolo *u* de la cadena de entrada, existe un único estado $r \in Q$ al que se puede mover el autómata, es decir, $f(q, u) = r$. En el caso de los autómatas finitos no deterministas $f(q, u) \in R \subseteq Q$.

Definición 2.5.8: El **lenguaje aceptado** por un autómata finito *A* es:

$$L(A) = \{w \ tal \ que \ w \in \Sigma^* \text{ y } (q_0, w) \vdash^* (q_f, \varepsilon), q_f \in F\}$$

Definición 2.5.9: Un **autómata de pila** es un séptuplo: $P = (Q, \Sigma, \Gamma, f, q_0, Z_0, F)$ donde:

Q: Conjunto de estados.

Σ: Alfabeto de entrada.

Γ: Alfabeto de pila.

f: Función de transición.

q_0: Estado inicial.

Z_0: símbolo inicial de pila.

F: Conjunto de estados finales.

Definición 2.5.10: La terna (q, w, λ) tal que $q \in Q,\ w \in \Sigma^*, \lambda \in \Gamma^*$ es una **configuración** del autómata *P*.

Definición 2.5.11: Sean $(q, uw, \beta\lambda), (r, w, \gamma\lambda)$ configuraciones de *P*, tal que $q \in Q, u \in \Sigma, w \in \Sigma^*, \beta \in \Gamma, \gamma, \lambda \in \Gamma^*$. Un posible **movimiento** es el paso de la configuración $(q, uw, \beta\lambda)$ a la $(r, w, \gamma\lambda)$. Notación: $(q, uw, \beta\lambda) \vdash (r, w, \gamma\lambda)$.

Definición 2.5.12: La **función de transición** de un autómata de pila *P* es una función $f: Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to \mathrm{Q} \times \Gamma^*$.

Definición 2.5.13: Sean $(q_1, u_1u_2 \dots u_n, \lambda_1), \dots, (q_k, u_k \dots u_n, \lambda_k)$ configuraciones del autómata. Si $(q_1, u_1u_2 \dots u_n, \lambda_1) \vdash \cdots \vdash (q_k, u_k \dots u_n, \lambda_k)$ se dice que existe una **secuencia de movimientos** de $(q_1, u_1u_2 \dots u_n, \lambda_1)$ a $(q_k, u_k \dots u_n, \lambda_k)$.

Notación: $(q_1, u_1u_2 \dots u_n, \lambda_1) \vdash^* (q_k, u_k \dots u_n, \lambda_k)$.

Definición 2.5.14: El **lenguaje aceptado** por el autómata de pila *P* por el criterio de estado final es:

$$L(P) = \{w\ tal\ que\ w \in \Sigma^* \wedge (q_0, w, Z_0) \vdash^* (q_f, \varepsilon, \lambda), q_f \in F, \lambda \in \Gamma^*\}$$

El autómata *P* reconoce una cadena *w* si llega a un estado final con la cadena vacía. Es importante destacar que un autómata de pila no realiza ningún cambio de configuración si la pila está vacía. Otro posible criterio de lenguaje aceptado es el criterio de pila vacía.

Definición 2.5.15: El **lenguaje aceptado** por el autómata de pila *P* por el criterio de pila vacía es:

$$L_\varepsilon(P) = \{w\ tal\ que\ w \in \Sigma^* \wedge (q_0, w, Z_0) \vdash^* (q_f, \varepsilon, \varepsilon), q_f \in F\}$$

La definición de autómata de pila se puede extender para lograr una función de transición capaz de remplazar una cadena finita de símbolos del tope de la pila por otra, en lugar de leer un solo símbolo de la pila en cada cambio de configuración [1].

Definición 2.5.16: Un **autómata de pila extendido** es un séptuplo: $P = (Q, \Sigma, \Gamma, f, q_0, Z_0, F)$ donde los símbolos *Q, Σ, Γ, q_0, Z_0, F* tienen el mismo significado que para el autómata de pila. Para este autómata la función de transición se define como $f: Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma^* \rightarrow Q \times \Gamma^*$. Las dos diferencias del autómata de pila extendido con el autómata de pila es que el primero:

1. Puede leer más de un símbolo del tope de la pila.
2. Se puede mover aunque la pila esté vacía.

2.6 Árbol de Sintaxis Abstracta.

Los árboles de sintaxis abstracta, que de aquí en adelante AST por sus siglas en ingles, son una representación de la información semántica de una cadena que pertenece al lenguaje generado por una gramática. Estos árboles se construyen siguiendo la idea de los árboles de derivación, lo que a diferencia de estos, obvian mucha información, sintácticamente útil, pero semánticamente irrelevante (punto y coma, paréntesis, etc.).

Una de las ventajas de escoger una estructura arbórea es el hecho de poder establecer relaciones jerárquicas entre los símbolos de la gramática.

Una vez definidos los elementos necesarios de la Teoría de Lenguajes Formales es posible construir una gramática libre del contexto que genere el lenguaje de las interpretaciones que hacen verdaderas a una fórmula. Para ello, se construye un autómata finito determinista que reconoce todas las cadenas verdaderas de la fórmula de entrada asumiendo que todas las variables son distintas. A éste autómata se le añade una pila como mecanismo de memoria externo para garantizar que todas las instancias de una misma variable tomen siempre el mismo valor. La combinación de un autómata finito con una pila da como resultado un autómata de pila, a partir del que se construye una grámatica libre del contexto que genera el lenguaje de todas las interpretaciones verdaderas de la fórmula de entrada. Si el lenguaje generado por la gramática no es vacío entonces la fórmula es satisfacible, en caso contrario no lo es. En el capítulo siguiente se detallan estos pasos.

Capítulo 3 De la fórmula de entrada a la gramática libre del contexto.

Para responder a la pregunta de si una fórmula lógica, cuyas variables aparezcan en un orden libre del contexto, es satisfacible o no, se construye una representación polinomial del lenguaje de **todas** sus interpretaciones verdaderas. Si este lenguaje es vacío entonces la fórmula no es satisfacible, en caso contrario, la fórmula es satisfacible y se obtiene además la cantidad de interpretaciones verdaderas de la misma y un mecanismo que permite generar todas sus interpretaciones verdaderas.

3.1 Lenguaje Satisfacible.

Sea A una fórmula cualquiera; si en ella ocurren n variables, entonces existen 2^n interpretaciones distintas de la misma. Como consecuencia, el lenguaje de todas las interpretaciones verdaderas es finito. Si se denota por L al lenguaje de todas las interpretaciones posible de la fórmula A, entonces el lenguaje L^* de todas las interpretaciones de A que la hacen verdadera es un subconjunto de L. Una posible alternativa para buscar todas las cadenas que pertenecen a L^* es recorrer todas las cadenas que pertenecen a L y determinar cuáles de éstas satisfacen la fórmula A. Debido a que la cantidad de cadenas que hay en L esta alternativa de solución es impracticable por lo que se opta por un mecanismo que permita representar el lenguaje L^* de forma más compacta, para ello se usan los principios de la Teoría de Lenguajes Formales, que como se comentaba en el capítulo 2, brindan herramientas para generación y reconocimiento de las cadenas que pertenecen a un lenguaje.

Antes de explicar el procedimiento empleado para construir el lenguaje L^* es necesario hacer una especificación formal de éste; para lo cual primeramente hay que realizar algunas definiciones.

Definición 3.1.1: Sea A una fórmula y U una interpretación de A; entonces la secuencia de valores de verdad resultante de sustituir en la secuencia de instancias de variables asociada a A los valores que toma cada una de las variables en U es una **Cadena de Valores de Verdad**.

Definición 3.1.2: Sea A una fórmula y U una interpretación que evalúa verdadera a A; entonces la secuencia de valores de verdad resultante de sustituir en la secuencia de instancias de variables asociada a A los valores que toma cada una de las variables en U es una **Cadena Verdadera**.

Definición 3.1.3: Sea A una fórmula y U una interpretación que evalúa falsa a A; entonces la secuencia de valores de verdad resultante de sustituir en la secuencia de instancias de variables asociada a A los valores que toma cada una de las variables en U es una **Cadena Falsa**.

Definición 3.1.4: Al lenguaje que contiene todas las Cadenas Verdaderas asociadas a una fórmula A lo llamaremos **Lenguaje Satisfacible** de A y lo denotaremos por $\tilde{L}$.

Para determinar si una fórmula A es satisfacible o no, se planteó como estrategia de solución la construcción de una representación polinomial del lenguaje de todas las interpretaciones

verdaderas: L^*. Para cada cadena x tal que $x \in \tilde{L}$, donde $\tilde{L}$ es el Lenguaje Satisfacible asociado a A existe una y solo una interpretación v tal que $v \in L^*$, esto se debe a la forma en que se construyen las cadenas de $\tilde{L}$. Luego se puede afirmar que $\tilde{L} \neq \emptyset$ si y sólo si $L^* \neq \emptyset$, de manera tal que es suficiente determinar si $\tilde{L} \neq \emptyset$ para poder afirmar que la fórmula A es satisfacible.

3.2 Autómata Booleano.

Para construir el lenguaje $\tilde{L}$ asociado a una fórmula lógica A, se supone que todas las instancias de variables de A corresponden a variables diferentes y se construye un autómata finito que reconoce todas las cadenas verdaderas de la fórmula. Una vez construido este autómata finito se le añade una pila para controlar que todas las instancias de variable asociadas a la misma variable tomen el mismo valor.

Definición 3.2.1: Un **autómata booleano** es un séptuplo $B = (Q, \Sigma, f, q_0, V, F)$ donde:

Q: Conjunto de estados.

Σ: Alfabeto de entrada. Dado que el autómata reconoce todas las cadenas verdaderas, el alfabeto de entrada está constituido por los dos posibles valores de verdad que pueden tomar las variables booleanas; estos son Verdadero, representado con V, 1 ó True y Falso, representado con F, 0, ó False.

f: Función de transición. $f: Q \times \Sigma \to Q$

q_0: Estado inicial del autómata.

V: Estado final del autómata. Representa que la cadena de entrada es una cadena verdadera.

F: Es un estado distinguido del autómata, no es considerado estado final. Representa que la cadena de entrada es una cadena falsa.

Definición 3.2.2: El par $(q, w) \in Q \times \Sigma^*$ es una **configuración** de B. La cadena w representa el fragmento de la cadena de entrada que todavía no se ha analizado.

Definición 3.2.3: Un **movimiento** en B está dado por un cambio de configuración, $(q, Iw) \vdash (r, w)$ donde $q, r \in Q,\ I \in \Sigma,\ w \in \Sigma^*$. Los cambios de configuración posibles son aquellos definidos por la función de transición; si $f(q, I) = r$ entonces $(q, Iw) \vdash (r, w)$. El movimiento anterior significa que estando en el estado q y teniendo a I como próximo símbolo a analizar de la cadena de entrada, se pasa al estado r.

Definición 3.2.4: La **configuración inicial** de un autómata booleano es (q_0, w).

Definición 3.2.5: La **configuración final** de un autómata booleano es (V, ε).

Definición 3.2.6: Sean $(q_1, u_1u_2 \dots u_n), \dots, (q_k, u_k \dots u_n)$ configuraciones del autómata booleano. Si $(q_1, u_1u_2 \dots u_n) \vdash \cdots \vdash (q_k, u_k \dots u_n)$ se dice que existe una **secuencia de movimientos** de $(q_1, u_1u_2 \dots u_n)$ a $(q_k, u_k \dots u_n)$.

Notación: $(q_1, u_1u_2 \dots u_n) \vdash^* (q_k, u_k \dots u_n)$.

Definición 3.2.7: El **lenguaje aceptado** por el autómata booleano es:

$$L(B) = \{w \in \Sigma \; tal \; que \; (q_0, w) \vdash^* (V, \varepsilon)\}$$

El modelo de construcción del autómata booleano parte de la construcción de un autómata booleano para cada instancia de la secuencia de variables correspondiente a la fórmula de entrada y siguiendo el proceso de formación de ésta, ir concatenando los autómatas en dependencia de los operadores que aparezcan en la fórmula. La combinación de dos autómatas da como resultado un nuevo autómata que reconoce todas las cadenas verdaderas correspondientes a la fórmula a partir de la cual se construye. Al finalizar el proceso de concatenación se obtiene un autómata que reconoce todas las cadenas verdaderas de la fórmula de entrada.

Cada estado del autómata booleano está asociado con una instancia de variable de la secuencia de variables de la fórmula de entrada. Esto es el resultado del proceso de construcción de autómata booleano, para más detalles véanse las definiciones 3.2.8, 3.2.11 y 3.2.12.

Para llevar a cabo el proceso de construcción se toman como base las reglas de formación de fórmulas. Se parte del automata asociado a una instancia de variable, se aplican las reglas para concatenar autómatas de acuerdo a los operadores lógicos y se obtiene el autómata asociado a la fórmula de entrada.

Definición 3.2.8: El **autómata booleano asociado a una instancia de variable** se define como:

$$B = (\{q_0, V, F\}, \{0,1\}, f, q_0, V, F) \qquad \begin{matrix} f(q_0, 0) = F \\ f(q_0, 1) = V \end{matrix}$$

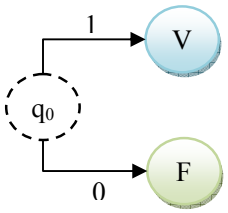

Figura 2: Representación gráfica de la función de transición.

El autómata asociado a una instancia de variable constituye la base sobre la que se construye el autómata asociado a una fórmula. El autómata booleano construido a partir de una fórmula lógica es una estructura por niveles.

Definición 3.2.9: El **nivel de un estado** se define recursivamente como sigue:

1. El nivel del estado inicial del autómata es cero.
2. Si el nivel del estado q es n, entonces para todo estado p tal que $f(q, I) = p$ el nivel del estado p es $n + 1$.

Definición 3.2.10: Si un **estado** tiene nivel n entonces está **asociado a** la **instancia de variable** que aparece en la posición n de la secuencia de instancias de variables, comenzando a contar a partir de cero.

Autómata asociado a la negación de una fórmula.

Definición 3.2.11: Sea *A* una fórmula lógica y *B* el autómata booleano asociado a *A* definido como: $B = (Q, \Sigma, f, q_0, V, F)$. Entonces el autómata booleano asociado a $\neg A$ se define como: $NB = (Q, \Sigma, f, q_0, F, V)$.

Al realizar la operación negación, todas las cadenas verdaderas de la fórmula pasan a ser cadenas falsas y viceversa, por lo que el significado de los estados V y F se invierte.

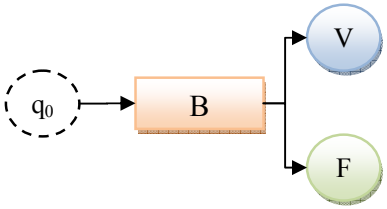

Figura 2: Autómata Booleano asociado a la fórmula A.

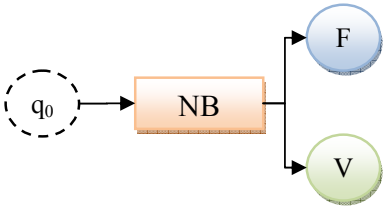

Figura 3: Autómata Booleano asociado a la fórmula ¬A.

Autómata booleano asociado a la conjunción y disyunción de dos fórmulas.

Definición 3.2.12: Sean A_1 y A_2 fórmulas lógicas y sean $B\_1 = (Q_1, \Sigma, f_1, q_1, V_1, F_1)$ y $B\_2 = (Q_2, \Sigma, f_2, q_2, V_2, F_2)$ los autómatas booleanos asociados a ellas. Entonces el autómata booleano asociado a:

1. $A1 \vee A2$ se define como: $D = (Q_1 \cup (Q_2 - q_2) \cup Q, \Sigma, f, q_1, V_2, F_2)$

Q es el conjunto de estados que se añaden al autómata resultante. La cantidad de elementos de *Q* es igual a la cantidad de instancias de variables que hay en la fórmula *B* menos uno.

El proceso de construcción del autómata *D* se divide en dos casos. Si la entrada se evalúa verdadera en el autómata *B_1*, entonces no importa el resultado de la evaluación del autómata *B_2*, pues ya el resultado del autómata *D* es verdadero. El otro caso posible es cuando el resultado de evaluar la cadena de entrada en *B_1* es falso; entonces el resultado final de la operación depende del resultado de *B_2*.

Caso1: El autómata B_1 se evalúa falso.

En este caso el resultado de la operación *disyunción* depende del resultado del autómata *B_2*. La forma de resolverlo es realizar una ε transición del F_1 (estado *Falso* de *B_1*) al estado q_2 (estado inicial de *B_2*); esta solución introduce una situación de indeterminismo al autómata por lo que en su lugar se realiza una "fusión" entre los mismos. Esta alternativa es equivalente a realizar la ε transición y luego eliminar las condiciones de indeterminismo.

Fusión de estados F_1 y q_2.

Sea *pv* y *pf* estados del autómata *B_2* y sean $f(q_2, 1) = pv$ y $f(q_2, 0) = pf$ transiciones válidas del autómata B_2. Se define la función de fusión $\bar{f}$ como sigue: $\bar{f}(F_1, 1) = pv$ y $\bar{f}(F_1, 0) = pf$

Caso2: El autómata B1 se evalúa verdadero.

En este caso no es necesario analizar el resultado del autómata *B_2* puesto que independientemente del valor que tomen las variables presentes en el autómata *B_2*, el

resultado de la operación *disyunción* es verdadero. Entonces se añade una secuencia de estados a partir del estado V_1 (estado *Verdadero* de *B_1*) hasta el estado V_2 (estado *Verdadero* de *B_2*), de manera que cada uno de éstos se corresponda con las variables presentes en el autómata *B_2*. Para este caso se construye una <u>función de extensión</u> $\check{f}$ definida de la siguiente forma:

$$\check{f} = \begin{cases} \check{f}(V_1, I) = a_1 \\ \check{f}(a_k, I) = a_{k+1} \quad k = 1, \dots, m \\ \check{f}(a_m, I) = V_2 \end{cases}$$
$$I \in \Sigma$$
$$a_i \in Q$$

Donde m es la cantidad de instancias de variables, menos una, que hay en la fórmula A_2. Los a_i, $i = 1 \dots m$ son los estados que se añaden.

Luego la función de transición del autómata *D* es una función por partes definida de la siguiente forma:

$$f(q, I) = \begin{cases} f_1(q, I) & q \in Q_1 \\ f_2(q, I) & q \in Q_2 - \{q_2\} \\ \bar{f}(q, I) & q = F_1 \\ \check{f}(q, I) & q \in Q \end{cases}$$

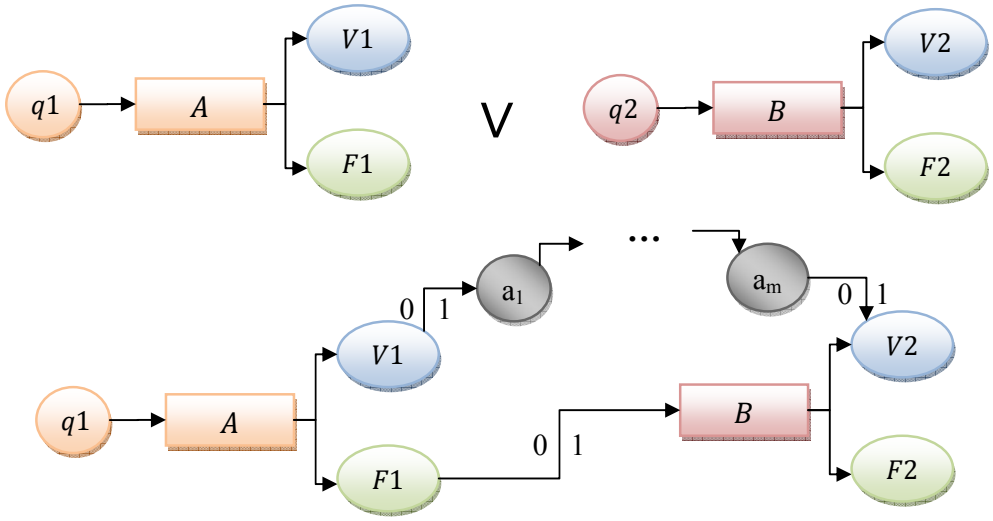

Figura 3: Proceso de construcción del autómata asociado a la fórmula resultante de efectuar la operación disyunción entre dos fórmulas A y B.

2. $A1 \wedge A2$ se define como: $C = (Q_1 \cup (Q_2 - q_2) \cup Q, \Sigma, f, q_1, V_2, F_2)$

Donde *Q* es el conjunto de estados que hay que añadir como resultado de la concatenación.

El proceso de construcción del autómata *C* es muy similar al del autómata *D*. La diferencia está dada por el hecho de que para la operación conjunción si el autómata *B_1* se evalúa verdadero entonces el resultado depende de la evaluación del autómata *B_2*. En caso contrario, el resultado de la operación conjunción es falso, independientemente del resultado del autómata *B_2*.

Para esta operación los estados que se fusionan son los estados V_1 (estado *Verdadero* de *B_1*) y q_2 (estado inicial de *B_2*), de manera análoga a como lo hacen los estados F_1 y q_2. Y la secuencia de estados adicionales se establece desde el estado F_1 (estado *Falso* de *B_1*) hasta el F_2 (estado *Falso* de *B_2*).

En este caso la función de fusión se define de la siguiente manera: sean *pv* y *pf* estados del autómata *B_2* y sean $f(q_2, 1) = pv$ y $f(q_2, 0) = pf$ transiciones válidas del autómata *B_2*, entonces : $\bar{f}(V_1, 1) = pv \; y \; \bar{f}(V_1, 0) = pf$.

De forma similar a cómo se realizó la extensión en *D*, para la operación conjunción la función de extensión queda definida como:

$$\check{f} = \begin{cases} \check{f}(F_1, I) = a_1 \\ \check{f}(a_k, I) = a_{k+1} \quad k = 1, \dots, m \\ \check{f}(a_m, I) = F_2 \end{cases}$$
$$I \in \Sigma$$
$$a_i \in Q$$

Al igual que en la disyunción, m es la cantidad de instancias de variables, menos una, que hay en la fórmula A_2. Los a_i, $i = 1 \dots m$ son los estados que se añaden.

Luego la función de transición correspondiente al autómata *C* es una función definida por partes de la siguiente forma.

$$f(q, I) = \begin{cases} f_1(q, I) & q \in Q_1 \\ f_2(q, I) & q \in Q_2 - \{q_2\} \\ \bar{f}(q, I) & q = V_1 \\ \check{f}(q, I) & q \in Q \end{cases}$$

Por convenio se denominan operadores de concatenación a los operadores conjunción y disyunción.

El proceso de construcción del autómata booleano es un proceso "bottom – up". Comenzando con la construcción de los autómatas asociados a cada una de las instancias de variable de la fórmula lógica y posteriormente, realizando un proceso de concatenación de los autómatas, en dependencia de los operadores que aparezcan en la fórmula y la prioridad establecida entre ellos.

Lema 3.2.1: Sean *A* y *B* dos autómatas booleanos asociados a dos fórmulas cuyas secuencias de variables son de longitud *p, q* respectivamente y tales que $p + q = n$. Como resultado del proceso de concatenación se añaden al autómata resultante $O(n)$ nuevos estados.

Demostración Lema 3.2.1:

Sean A, B autómatas booleanos.

Cantidad de estados de A = a. Notación: $\langle A \rangle$

Cantidad de estados de B = b. Notación: $\langle B \rangle$

Longitud de la fórmula asociada a A = p. Notación: $|A|$

Longitud de la fórmula asociada a B = q. Notación: $|B|$

Los autómatas se pueden concatenar empleando solamente las operaciones $\wedge$ y $\vee$, las cuales añaden la misma cantidad de estados (ver definición 3.2.11). Luego la cantidad de estados del autómata resultante es: $a + \underbrace{(b-1)}_{1} + \underbrace{(q-1)}_{2}$

1: Se elimina un estado de *B* como resultado del proceso de fusión que se realiza entre el estado inicial de *B* y un estado de *A* (Verdadero o Falso), en dependencia de la operación lógica.

2: Al autómata resultante se le añaden $|B| - 1 = q - 1$ nuevos estados.

Como $q - 1 < n$ entonces $(q-1) \in O(n)$. ■

Proposición 3.2.1: La cantidad de estados en el autómata booleano resultade de concatenar dos autómtas A y B es: $\langle A \rangle + \langle B \rangle - 1 + O(n)$.

Demostración: Se deduce del Lema 3.2.1.■

¿Cuántos estados tiene el autómata booleano asociado a una fórmula lógica con *n* instancias de variables?

Para responder a esta pregunta es necesario analizar cuántos estados se añaden al autómata booleano resultante como parte del proceso de concatenación de autómatas.

Teorema 3.2.1: La cantidad de estados en el autómata booleano es de orden cuadrático de la longitud de la secuencia de variables asociada a la fórmula de entrada.

Demostración Teorema 3.2.1:

Sea *A* una fórmula lógica de longitud *n*. Para construir el autómata booleano asociado a ella primero se construyen los autómatas asociados a cada una de las instancias de variable que aparecen en la secuencia. Una vez finalizada esta etapa se pasa a la concatenación de los mismos teniendo en cuenta el siguiente orden.

1. Efectuar las operaciones entre paréntesis.
2. Efectuar el operador negación.
3. Efectuar el operador conjunción.
4. Efectuar el operador disyunción.

Como parte del proceso de construcción, cada vez que se concatenen dos autómatas A_i y A_j tales que:

Cantidad de estados de $A_i = \langle A_i \rangle$

Cantidad de estados de $A_j = \langle A_j \rangle$

Longitud de la fórmula asociada a $A_i = |A_i|$

Longitud de la fórmula asociada a $A_j = |A_j|$

Se añaden al autómata resultante $|A_j| \in O(n)$ por Lema 3.2.1.

Durante el proceso de construcción del autómata booleano asociado a una fórmula *A* de longitud *n*, se efectúan $n-1$ concatenaciones dado que entre cualesquiera dos instancias de variable, precedidas o no del operador de negación, hay un operador de conjunción o disyunción y por cada uno de estos se efectúa una concatenación. Como resultado del proceso de construcción se tiene que la cantidad de estados del autómata booleano resultante es:

$$n * C + (n-1) * R - n \qquad (1)$$

Donde *C* es la cantidad de estados de un autómata booleano asociado a una instancia de variable. La cantidad de estados del autómata asociado a una instancia de variable es constante, $C \in O(1)$ y R es la cantidad de estados que se añaden a lo sumo en cada concatenación, $R \in O(n)$ por el Lema 3.2.1. La expresión anterior significa que la cantidad de estados del autómata resultante es igual a la cantidad de estados que se obtienen de los autómatas asociados a cada instancia de variable de la fórmula, más lo que se adiciona durante las $n-1$ concatenaciones realizadas para obtener el autómata asociado a la fórmula *A*. Al final se eliminan *n* estados porque durante el proceso de concatenación de dos autómatas se pierde siempre un estado como consecuencia de la fusión entre el estado inicial de A_j y un estado de A_i (Verdadero o Falso), en dependencia de la operación lógica.

Efectuando operaciones algebraicas en la expresión (1) tenemos que si *B* es el autómata booleano asociado a *A* entonces:

$$\begin{aligned}\langle B\rangle &= n * C + (n-1) * R - n \\ &= (n-1) * C + (n-1) * R \\ &= (n-1) * O(1) + (n-1) * O(n) \\ &= O(n) + O(n^2) = O(n^2)\end{aligned}$$

Con esto queda demostrado que la cantidad de estados del autómata booleano asociado a una fórmula lógica cuya secuencia de variables es de longitud *n* es a lo sumo de orden n^2. ■

El proceso de construcción del autómata booleano descrito anteriormente no "aprovecha" los caminos de extensión construidos con anterioridad. A continuación se expone un ejemplo donde hacer uso de los caminos de extensión previamente construidos disminuye el número de estados que se adicionan al autómata resultante.

Sea la fórmula $A = x1 \wedge \big(\neg(x2 \vee x3) \wedge (x3 \vee x1)\big)$. Para construir el autómata booleano se asume que todas las variables son distintas. Sea *B* el autómata booleano asociado a la fórmula $\check{A} = x1 \wedge \big(\neg(x2 \vee x3) \wedge (x4 \vee x5)\big)$ que es una fórmula obtenida a partir de renombrar las instancias de variable de *A* para que en $\check{A}$ todas las instancias de varaiable correspondan a variables distintas.

$$B = \begin{pmatrix}\{0,1,2,3,4,5,6,7,8,9,10,11,12, V, F\}, \\ \{0,1\}, f, 0, V, F\end{pmatrix}.$$

La función de transición del autómata B se ilustra en la figura 4.

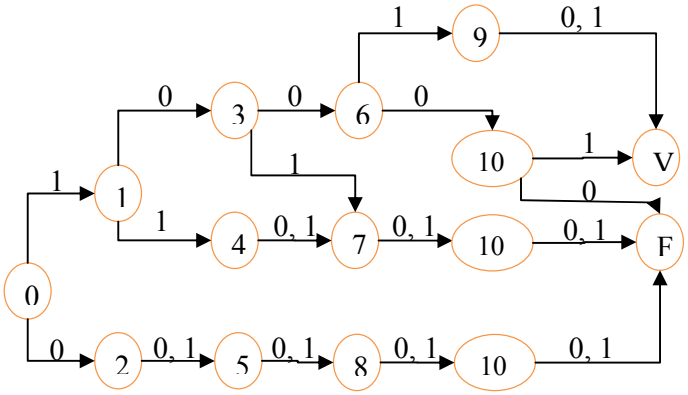

Figura 4: Función de transición del autómata booleano B.

El proceso de construcción se traduce en el ejemplo en concatenar los autómatas booleanos asociados a las siguientes fórmulas:

1. $A_1 = x2 \vee x3$
2. $A_2 = -A_1$
3. $A_3 = x3 \vee x1$
4. $A_4 = A_2 \wedge A_3$
5. $A_5 = x1 \wedge A_4$

Al analizar el ejemplo anterior se aprecia que al construir el autómata booleano asociado a la fórmula A_4 se adicionaron estados para conformar un "camino" que condujera al estado *F*. Obsérvese nuevamente, en la figura 5, el fragmento de la función de transición del autómata *B* que se corresponde con la función de transición del autómata asociado a A_4. En la figura 5 aparece resaltado el camino que conecta al estado 4 con el *F*; este es un camino de extensión porque independientemente del símbolo de entrada el resultado de evaluar la cadena es conocido, en este caso, el resultado será falso.

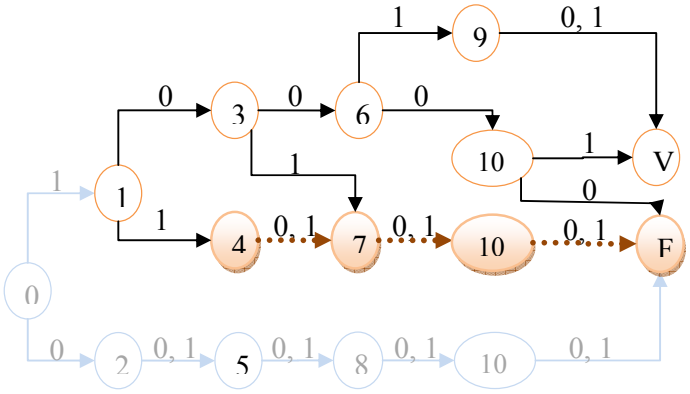

Figura 5: Función de transición asociada al autómata B aparece con puntos suspensivos en el fragmento correspondiente al autómata asociado a la fórmula A_4.

Cuando se construyó el autómata asociado a la fórmula $A_5 = x1 \wedge A_4$ se añadió otro camino de extensión que conduce al estado *F*. Sin embargo al realizar este proceso no se aprovechó la información que se tenía de procesos realizados anteriormente; lo que hubiese permitido que la cantidad de estados a adicionar fuese menor, dado que la función de transición representada en la figura 6 es equivalente a la de la figura 7.

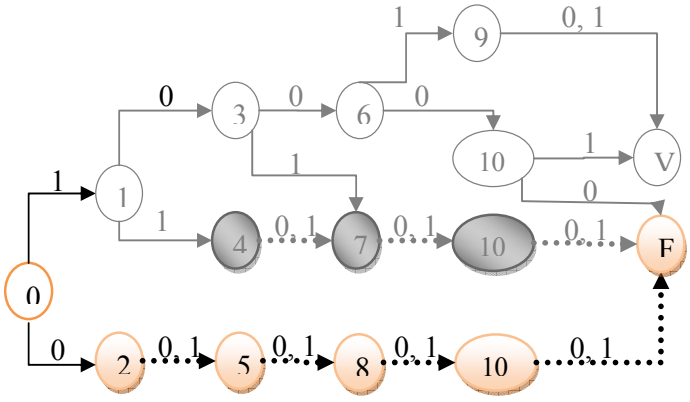

Figura 6: Función de transición del autómata booleano.

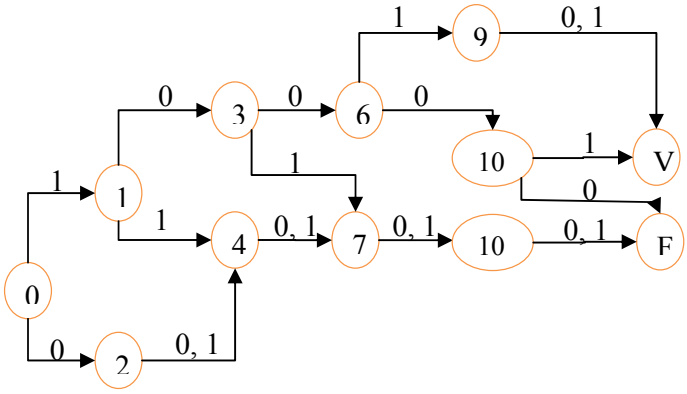

Figura 7: Función de transición correspondiente al autómata booleano extendido.

La modificación de la función de transición se realizó aprovechando los caminos de extensión que ya se habían construidos. En el ejemplo anterior se evidenció la utilidad que tiene aprovechar los caminos de extensión previamente construidos con el objetivo de minimizar la cantidad de estados del autómata resultante. En la sección siguiente se propone una extensión al autómata booleano con el objetivo de aprovechar, siempre que sea posible, los caminos de extensión previamente construidos.

3.3 Autómata Booleano Extendido.

En esta sección se define un nuevo tipo de autómata booleano capaz de aprovechar los caminos de extensión, lo que permite reducir la cantidad de estados del autómata booleano resultante.

Definición 3.3.1: Un **autómata booleano extendido** es un óctuplo

$$B\_ex = (Q, \Sigma, f, q_0, V, F, TV, TF)$$

Q: Conjunto de estados.

Σ: Alfabeto de entrada = {0,1}.

f: Función de transición. $f: Q \times \Sigma \rightarrow Q$

q_0: Estado inicial del autómata.

V: Estado final del autómata. Representa que la cadena de entrada es una cadena verdadera.

F: Es un estado distinguido del autómata, no es considerado estado final. Representa que la cadena de entrada es una cadena falsa.

TV: Estado a partir del cual, independientemente de los símbolos presentes en la cadena de entrada ya se sabe que es una cadena verdadera.

TF: Estado a partir del cual, independientemente de los símbolos presentes en la cadena de entrada ya se sabe que es una cadena falsa.

El proceso de construcción del autómata booleano extendido difiere únicamente en la inclusión en el autómata de los dos nuevos estados distinguidos.

Autómata Booleano Extendido asociado a una instancia de variable.

Definición 3.3.2: Sea *x* una instancia de variable de la fórmula de entrada, se define como el autómata booleano extendido asociado a *x* el siguiente:

$$B\_ex = (\{q_0, V, F\}, \{0,1\}, f, q_0, V, F, TV, TF) \quad \begin{matrix} f(q_0, 0) = F \\ f(q_0, 1) = V \end{matrix}$$

Autómata Booleano Extendido asociado a la negación de una fórmula.

Definición 3.3.3: Sea *A* una fórmula lógica y B_{ex} el autómata booleano extendido asociado a *A* definido como: $B_{ex} = (Q, \Sigma, f, q_0, V, F, TV, TF)$. Entonces el autómata booleano extendido asociado a la negación de *A* ($\neg A$) se define como: $N_{ex} = (Q, \Sigma, f, q_0, F, V, TF, TV)$.

En este caso, al igual que en el autómata booleano, se intercambian los estados *V* y *F*, por lo que consecuentemente con esto también se intercambiaran los estados *TV* con *TF*.

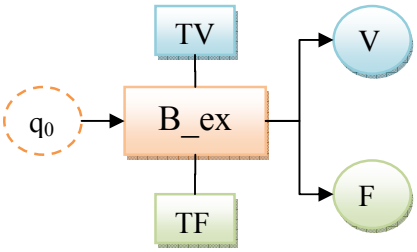

Figura 8: Función de transición del autómata B_{ex}.

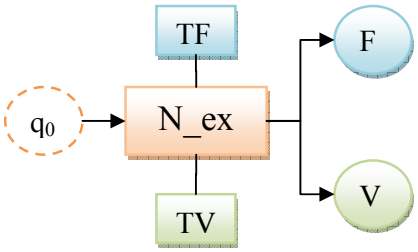

Figura 9: Función de transición del autómata N_{ex}.

En las figuras anteriores los estados TV y TF aparecen en el medio del autómata porque su nivel puede ser cualquiera.

La diferencia entre el autómata booleano extendido y el autómata booleano viene dada por la reutilización que hace este último de los caminos de extensión previamente construidos; lo que implica una disminución de la cantidad de estados del conjunto de estados adicionales y

por tanto un menor número de estados en el autómata booleano asociado a la fórmula de entrada.

Autómata Booleano Extendido asociado a la conjunción y disyunción de dos fórmulas.

Definición 3.3.4: Sean A_1 y A_2 fórmulas lógicas y sean B_{ex1} y B_{ex2} los autómatas booleanos extendidos asociados a ellas.

$$B_{ex1} = (Q_1, \Sigma, f_1, s_1, V_1, F_1, TV_1, TF_1)$$

$$B_{ex2} = (Q_2, \Sigma, f_2, s_2, V_2, F_2, TV_2, TF_2)$$

Entonces el autómata booleano extendido asociado a:

1. $A1 \vee A2$ se define como: $D_{ex} = (Q_1 \cup (Q_2 - s_2) \cup Q, \Sigma, f, s_1, V_2, F_2, TV_1, TF_2)$

Donde Q es el conjunto de estados adicionales. Para el autómata booleano extendido, la cantidad de estados del conjunto de estados adicionales es igual a la longitud de la secuencia de variables de la fórmula asociada al autómata B_{ex2} menos la longitud del camino de extensión del estado TV_2 al estado V_2.

Para el autómata booleano extendido se mantiene la definición de función de fusión dada para el autómata booleano, lo que cambia es la función de extensión, dado que el nuevo camino, en este caso (operación disyunción), conecta al estado V_1 (estado *Verdadero* de B_{ex1}) con el estado TV_2 (estado *TodosVerdaderos* de B_{ex2}). La función de extensión se define como:

$$\check{f} = \begin{cases} \check{f}(V_1, I) = a_1 \\ \check{f}(a_k, I) = a_{k+1} \quad k = 1, \dots, m \\ \check{f}(a_m, I) = TV_2 \\ I \in \Sigma \end{cases}$$

Donde m es el nivel del estado TV_2 menos uno. Los a_i, $i = 1 \dots m$ son los estados que se añaden.

Luego la función de transición del autómata D_{ex} es una función definida por partes como:

$$f(q, I) = \begin{cases} f_1(q, I) & q \in Q_1 \\ f_2(q, I) & q \in Q_2 - \{q_2\} \\ \bar{f}(q, I) & q = F_1 \\ \check{f}(q, I) & q \in Q \end{cases}$$

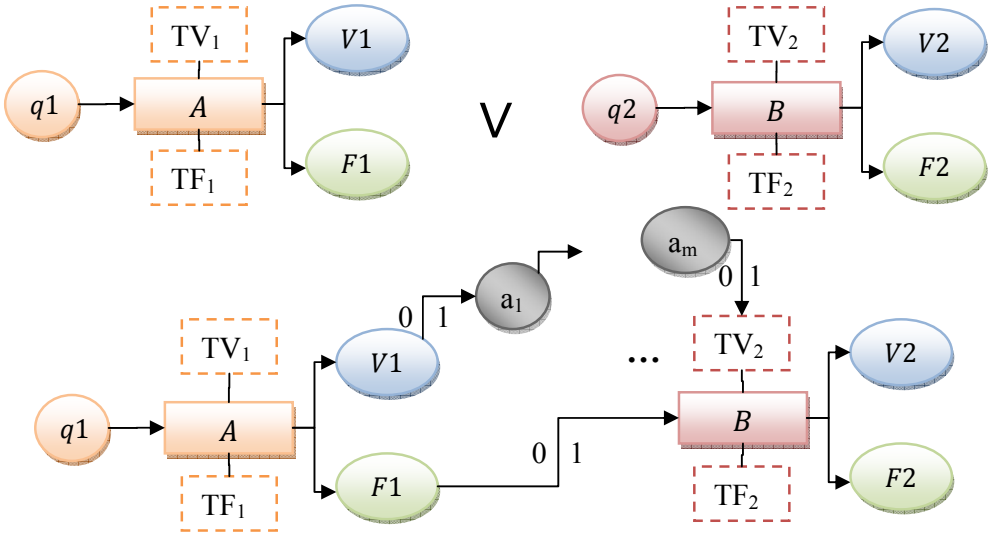

Figura 10: Proceso de construcción del autómata booleano extendido asociado a la fórmula resultante de efectuar la operación disyunción entre dos fórmulas A y B.

2. $A1 \wedge A2$ se define como: $C_{ex} = (Q_1 \cup (Q_2 - s_2) \cup Q, \Sigma, f, s_1, V_2, F_2, TV_2, TF_1)$

Donde Q es el conjunto de estados adicionales. Para el autómata booleano extendido, la cantidad de estados del conjunto de estados adicionales es igual a la longitud de la secuencia de variables de la fórmula asociada al autómata B_{ex2} menos la longitud del camino de extensión del estado TF_2 al estado F_2.

De manera análoga al autómata D_ex se redefine la función de extensión como:

$$\check{f} = \begin{cases} \check{f}(F_1, I) = a_1 \\ \check{f}(a_k, I) = a_{k+1} \quad k = 1, \dots, m \\ \check{f}(a_m, I) = TF_2 \\ I \in \Sigma \end{cases}$$

Donde m es el nivel del estado TF_2 menos uno. Los $a_i, i = 1 \dots m$ son los estados que se añaden.

La función de transición del autómata C_{ex} se define como:

$$f(q, I) = \begin{cases} f_1(q, I) & q \in Q_1 \\ f_2(q, I) & q \in Q_2 - \{q_2\} \\ \bar{f}(q, I) & q = V_1 \\ \check{f}(q, I) & q \in Q \end{cases}$$

Lema 3.3.1: Sean A_1 y A_2 dos fórmulas lógicas y sean B_1 y B_2 los autómatas booleanos extendidos asociados a ellas. En el autómata booleano extendido asociado a la fórmula resultante de efectuar la operación conjunción (disyunción) entre las fórmulas A_1 y A_2, el estado TodosVerdaderos (TodosFalsos) es el estado TodosVerdaderos (TodosFalsos) de B_2, por lo que su nivel aumenta en longitud de A_1 con respecto al nivel que tiene en B_2 y el estado

TodosFalsos (TodosVerdaderos) es el estado TodosFalsos (TodosVerdaderos) de A_1 por lo que su nivel es el mismo que tiene en A_1.

Demostración Lema 3.3.1:

Por definición de construcción del autómata booleano extendido asociado a la conjunción (diyunción) entre dos fórmulas. Véase definición 3.2.12.■

Esta modificación garantiza que la cantidad de estados del autómata booleano extendido asociado a una fórmula, es menor que la cantidad de estados del el autómata booleano.

3.4 Optimización del Autómata Booleano Extendido.

Hasta ahora se ha construido un autómata booleano que reconoce todas las cadenas que hacen verdadera la fórmula de entrada. Notar que si se dice que el estado *Falso* es un estado final entonces también se reconocerían las cadenas falsas la fórmula de entrada. Estas cadenas no interesan; luego, una vez finalizada la construcción del autómata booleano extendido asociado a una fórmula lógica se pueden eliminar de éste todas las transiciones que conduzcan al estado *Falso*. Permitiendo no solo disminuir el número de transiciones sino también la cantidad de estados, puesto que si en el proceso de eliminación de transiciones un estado queda sin transiciones se elimina, así como todas las transiciones incidentes en él.

3.4.1: Algoritmo para eliminar las transiciones al estado *F*.

Entrada: Autómata Booleano Extendido $B_{ex} = (Q, \Sigma, f, q_0, V, F, TV, TF)$

Salida: Autómata Booleano Extendido que no posee transiciones al estado Falso. $B_{ex} = (Q, \Sigma, f, q_0, V, \{\emptyset\}, TV, \{\emptyset\})$

Inicialización: R: Cola de estados a analizar. $R = \{F\}$

Instrucciones:

```
while R no vacía

    q es el próximo estado a analizar de la cola de estados.

    Q = Q - {q} //Eliminar el estado q del conjunto de estados
    del autómata.

    if ∀((r,I),q) ∈ f then f = f − {((r,I),q)} //Eliminar todas las
    transiciones al estado q.

    if ∀ f(r,I) = ∅ then R = R ∪ {r} //Si un estado ya no tiene
    ninguna transición entonces hay que eliminarlo.

end_while

return B_ex
```

Costo Computacional.

Lo peor que puede ocurrir es que se eliminen todos los estados del autómata, en ese caso se analizan todas las transiciones del mismo; luego el costo computacional es del orden de la cantidad de transiciones.

Como desde un estado solo se pueden realizar dos transiciones posibles entonces la cantidad de transiciones es 2*(cantidad de estados), entonces se puede decir que:

Proposición 3.4.1: La cantidad de transiciones del autómata booleano extendido es del orden de la cantidad de estados.

3.5 Mecanismo de Memoria.

Un autómata booleano es un autómata finito determinista que se construye en tiempo polinomial. Este autómata reconoce todas las interpretaciones verdaderas de la fórmula de entrada. Sin embargo este mecanismo es insuficiente puesto que asume que todas las instancias de variable de la fórmula de entrada son distintas. Por ejemplo:

Sea la fórmula $A = x1 \wedge (\neg(x2 \vee x3) \wedge (x3 \vee x1))$. Para construir el autómata booleano asociado a ella hay que reescribirla de manera que todas las variables sean distintas. Sea B es al autómata booleano extendido asociado a $\check{A} = x1 \wedge (\neg(x2 \vee x3) \wedge (x4 \vee x5)$

$$B = \begin{pmatrix} \{0,1,2,3,4,5,V,F\}, \\ \{0,1\}, f, 0, V, \emptyset, 4, \emptyset \end{pmatrix}$$

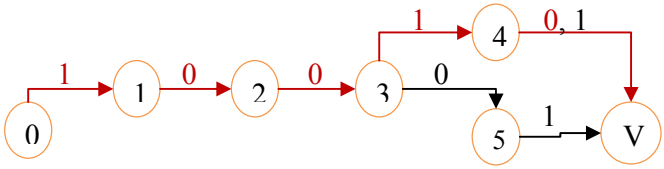

Figura 11: Representación gráfica de la función de transición de B.

En la figura 11 aparece señalado un camino del estado inicial al estado final. Por lo tanto la cadena "1 0 0 1 0" es una cadena verdadera para la fórmula $\check{A}$.

$$\check{A} = 1 \wedge \big(-(0 \vee 0) \wedge (1 \vee 0)\big) = 1$$

Sin embargo, la cadena "1 0 0 1 0" no es una cadena de valores de verdada de la fórmula *A* puesto que las instancias de las variables x1 y x3 toman valores diferentes. Esto ocurre porque al autómata booleano asociado a una fórmula lógica asume que todas las variables son distintas, dado que no cuenta con ningún mecanismo de memoria. Por esta razón es insuficiente este autómata para resolver el problema planteado; es necesario construir un mecanismo reconocedor de cadenas que disponga de un mecanismo de memoria capaz de garantizar que todas las instancias de una misma variable tomen siempre el mismo valor.

Se selecciona una pila como mecanismo de memoria porque la unión de un autómata booleano, que es un caso particular de un autómata finito, y una pila, da como resultado un autómata de pila. A partir de un autómata de pila se construye una gramática libre del contexto que genere el mismo lenguaje que reconoce el autómata por el criterio de pila vacía.

3.6 Orden Libre del Contexto.

En la sección anterior se explicó por qué no bastaba con un autómata booleano extendido para reconocer todas las interpretaciones verdaderas de la fórmula de entrada; además se planteó la necesidad de adicionar un mecanismo de memoria externo para suplir la deficiencia del autómata booleano que es su incapacidad de asignar a todas las instancias de una misma variable el mismo valor. El mecanismo de memoria seleccionado fue una pila, porque la unión de una pila con un autómata finito da un autómata de pila que es un mecanismo reconocedor de lenguajes libres del contexto. Este autómata de pila se obtiene a partir de transformaciones en la función de transición del autómata booleano.

En la pila se almacena el valor que se le asignana a cada variable. Aunque la cantidad de información que se puede almacenar en la pila es "infinita", en cada momento solo se puede acceder a la que está en el tope. La restricción de orden impuesta a la secuencia de variables garantiza que el valor de la próxima instancia a analizar es el que está en el tope de la pila.

La relación de orden que tienen que cumplir las instancias de la secuencia de variable de una fórmula lógica se denomina Orden Libre del Contexto, porque el lenguaje que reconoce un autómata de pila es un lenguaje libre del contexto.

3.6.1: Algoritmo para determinar si una secuencia de variables tiene un orden libre del contexto.

Sea $S = (s_1, s_2, \dots, s_n)$ la secuencia de variables asociadas a una fórmula lógica; donde cada s_i representa a la instancia de variable que se encuentra en la posición i-ésima. La idea del algoritmo es analizar cada una de las instancias de variables de la secuencia verificando que se cumplan las condiciones de la definición de orden libre del contexto (Véase epígrafe 1.6); para lo cual se cuenta con un conjunto Q, que es el conjunto de las variables que ya no pueden aparecer en la secuencia, y una estructura de datos tipo Pila.

Mientras se recorre la secuencia se apilan las variables. Si la próxima instancia a analizar, es instancia de una variable *v* que está en la pila, entonces todas las variables que están entre el tope de la pila y *v* son desapiladas, dejando *v* en el tope de la pila. Las variables que se sacaron de la pila son añadidas al conjunto Q para garantizar que se cumpla la primera condición de orden libre del contexto.

Si la próxima variable a analizar de la secuencia pertenece al conjunto Q, entonces la secuencia no cumple la restricción de orden libre del contexto. Si se pueden analizar todas las variables, entonces la secuencia es libre del contexto.

Algoritmo para determinar si una secuencia de variables tiene un orden libre del contexto.

Entrada: Secuencia de estados $S = (s_1, s_2, \dots, s_n)$

Salida:
Verdadero si la secuencia tiene un orden libre del contexto.
Falso si la secuencia no cumple la restricción de orden libre del contexto.

Inicialización:
$Q = \emptyset$
$P = \emptyset$

Instrucciones:
forall $s \in S$
if $s \in Q$ then return Falso //Si s está en el conjunto de estados de las variables que ya no pueden aparecer en la secuencia, entonces la secuencia no es libre del contexto.
else
if $s \notin P$ then $Push(P, s)$
else
while $s \neq Top(P)$
$p = Pop(P)$
$Q = Q \cup \{p\}$
end_while
end_forall
return Verdadero

Cada instancia de la secuencia se pone y se saca de la pila una única vez. Asumiendo que todas las operaciones que se realizan sobre la pila y el conjunto son constantes, el costo computacional del algoritmo está dado por la cantidad de operaciones realizadas. El algoritmo es $O(n)$, donde n es la longitud de la fórmula.

3.7 Autómata de Pila.

La combinación de una autómata booleano, que es un caso particular de un autómata finito, más una pila, es un autómata de pila. A partir de este autómata de pila se construye una gramática libre del contexto tal que el lenguaje aceptado por el autómata de pila es igual al lenguaje generado por la gramática. El lenguaje generado por la gramática es el lenguaje de todas las cadenas verdaderas de la fórmula de entrada. Luego para saber si la fórmula es satisfacible es suficiente determinar si el lenguaje generado por la gramática es vacío o no.

El autómata de pila se construye adicionando a cada una de las transiciones del autómata booleano la información almacenada en el mecanismo de memoria. Antes de explicar como se realiza este proceso es necesario establecer una clasificación de las instancias de la secuencia de variables asociada a la fórmula de entrada de acuerdo a su ocurrencia en la misma.

Definición 3.7.1: **Clase Libre.** La variable aparece una única vez por lo que su valor no está condicionado, en este caso no es necesario almacenarlo en el mecanismo de memoria.

El resto de las clasificaciones corresponden variables que aparecen más de una vez en la secuencia.

Definición 3.7.2: **Clase Primera.** A esta clase pertenecen todas las instancias de variables que corresponde a la primera ocurrencia de la variable.El resto de las instancias correspondiente a una misma variable tienen que tomar el valor que se le asigne la primera vez por lo que es necesario almacenar esta información en el mecanismo de memoria.

Definición 3.7.3 **Clase Intermedia.** A esta clase pertenecen todas las instancias de variables que se corresponden con una ocurrencia intermedia de la variable, por lo que su valor está condicionado al valor que tomó en su primera aparición, el cual está almacenado en el mecanismo de memoria. Como todavía quedan instancias de la misma variable por analizar, su valor tiene que volver a guardarse en la pila.

Definición 3.7.4: **Clase Última.** A esta clase pertenecen todas las instancias de variables que corresponden a la última ocurrencia de la variable., por lo que su valor está condicionado al valor que tomó en su primera ocurrencia pero como ya no aparece más en la secuencia de variables de la fórmula, se elimina del tope de la pila.

Definición 3.7.5: Un **autómata de pila booleano** es un séptuplo definido como:

$$P = (Q, \Sigma, \Gamma, q_0, Z_0, \tilde{f}, V).$$

Q: Conjunto de estados

Σ: Alfabeto de entrada = {0,1}

Γ: Alfabeto de pila = {0, 1, Z_0}

q_0: Estado inicial

Z_0: Símbolo inicial de pila

$\tilde{f}$: Función de transición $\tilde{f} : Q \times \{\Sigma \cup \{\varepsilon\}\} \times \Gamma^* \longrightarrow Q \times \Gamma^*$

V: Estado final

Definición 3.7.6: El par $(q, w, \alpha) \in Q \times \Sigma^* \times \Gamma^*$ es una **configuración** de P. La cadena w representa el fragmento de la cadena de entrada que todavía no se ha analizado y α representa el contenido de la pila; el símbolo más a la izquierda de α es el que está en el tope de la pila.

Definición 3.7.7: Un **movimiento** en P está dado por un cambio de configuración, $(q, Iw, T\alpha) \vdash (r, w, \xi\alpha)$ donde $q, r \in Q; I \in \Sigma \cup \{\varepsilon\}; w \in \Sigma^*; T \in \Gamma \cup \{\varepsilon\}; \alpha, \xi \in \Gamma^*$. Los cambios de configuración posibles son aquellos definidos por la función de transición; si $\tilde{f}(q, I, T) = (r, \xi)$ entonces $(q, Iw, T\alpha) \vdash (r, w, \xi\alpha)$. El movimiento anterior significa que

estando en el estado q, teniendo a I como próximo símbolo a analizar de la cadena de entrada y estando T en el tope de la pila, se pasa al estado r escribiendo la cadena ξ en el tope de la pila.

Una representación alternativa para la función de transición en mediante pares ordenados; es decir, la representación $\tilde{f}\ (q, I, T) = (r, \xi)$ es equivalente a $\big((q, I, T), (r, \xi)\big)$.

Definición 3.7.8: La **configuración inicial** de P es (q_0, w, Z_0)

Definición 3.7.9: La **configuración final** de P es $(V, \varepsilon, \varepsilon)$

Definición 3.7.10: Sean $(q_1, I_1 I_2 \ldots I_n, \lambda_1), \ldots, (q_k, I_k \ldots I_n, \lambda_k)$ configuraciones del autómata de pila. Si $(q_1, I_1 I_2 \ldots I_n, \lambda_1) \vdash \cdots \vdash (q_k, I_k \ldots I_n, \lambda_k)$ se dice que existe una **secuencia de movimientos** de $(q_1, I_1 I_2 \ldots I_n, \lambda_1)$ a $(q_k, I_k \ldots I_n, \lambda_k)$.

Notación: $(q_1, I_1 I_2 \ldots I_n, \lambda_1) \vdash^* (q_k, I_k \ldots I_n, \lambda_k)$.

Definición 3.7.11: El **lenguaje aceptado** por el autómata de pila booleano es:

$$L(P) = \{w \in \Sigma^* \ tal\ que\ (q_0, w, Z_0) \vdash^* (V, \varepsilon, \varepsilon)\}$$

Este es un autómata de pila extendido que reconoce por el criterio de pila vacía y que puede cambiar de configuración aunque la pila esté vacía (ver definición 2.5.16).

La idea general de la construcción del autómata de pila booleano consiste en modificar cada una de las transiciones del autómata booleano incluyendo la información almacenada en el mecanismo de memoria. Esta información depende de la clase a la que pertenezca la variable, por lo que es muy útil el hecho de haber asociado a cada estado del autómata booleano una instancia de variable de la secuencia de entrada.

Luego la función de transición del autómata de pila booleano $\tilde{f}$ se define a partir de la función de transición del autómata booleano f y de la clasificación establecida para las variables de la fórmula de entrada como sigue:

Si $f(q, I) = r$ y $q \in Clase\ Libre$ entonces $\tilde{f}(q, I, Z) = (r, Z)$

Si $f(q, I) = r$ y $q \in Clase\ Primera$ entonces $\tilde{f}(q, I, Z) = (r, IZ)$

Si $f(q, I) = r$ y $q \in Clase\ Intermedia$ entonces $\tilde{f}(q, I, I) = (r, I)$

Si $f(q, I) = r$ y $q \in Clase\ Última$ entonces $\tilde{f}(q, I, I) = (r, \varepsilon)$

Además, se añade el par $\tilde{f}(q_0, \varepsilon, Z_0) = (q_0, \varepsilon)$ para vaciar la pila porque el lenguaje que reconoce el autómata construido es el lenguaje que contiene a todas las cadenas w tales que partiendo del estado inicial con el símbolo Z_0 en el tope de la pila se llega al estado final con la pila vacía.

3.8 Gramática Libre del Contexto.

El autómata de pila booleano *P* es un mecanismo que reconoce todas las cadenas verdaderas de la fórmula de entrada. A partir de este autómata se construye una gramática libre del contexto *G*, tal que el lenguaje que reconoce *P* por el criterio de pila vacía es el mismo que genera *G*, $L_\varepsilon(P) = L(G)$. Para responder a la pregunta de si la fórmula es satisfacible, hay que determinar si el lenguaje generado por la gramática *L(G)* es vacío o no. Si el lenguaje no es vacío entonces existe al menos una interpretación verdadera para la fórmula por lo que ésta es satisfacible.

A partir de un autómata de pila booleano $P = (Q, \Sigma, \Gamma, f, q_0, Z_0, F)$ se construye una gramática libre del contexto $G = (N, \Sigma, \mathrm{P}, D)$ tal que la derivación extrema izquierda de una cadena *w*, $w \in L(G)$ se corresponde con los movimientos que realiza el autómata de pila *P* para reconocer la misma cadena. La **derivación extrema izquierda** de una cadena *w* que pertenece al lenguaje generado por una gramática *G* es una secuencia de derivaciones en la que partiendo del símbolo distinguido de *G* se llega a *w* y en cada derivación directa siempre se aplica la sustitución del símbolo no terminal que está más a la izquierda en la cadena que se está derivando [10].

En esencia el proceso de construcción de la gramática consiste en construir producciones para cada una de las transiciones válidas del autómata de pila booleano. Como cada transición de este autómata se construye a partir de las transiciones del autómata booleano, y la clasificación de las instancias de las variables de la secuencia de entrada en Libre, Primera, Intermedia y Última; una implementación computacional de este algoritmo tendrá en cuenta esto y realizará la construcción de la gramática a partir del autómata booleano en lugar de hacerlo a partir del autómata de pila booleano.

Las producciones de la gramática se construyen tomando como base un resultado de la Teoría de Lenguajes Formales que plantea cómo realizar la construcción de una gramática libre del contexto que genere el mismo lenguaje que reconoce un autómata de pila por el criterio de pila vacía.

Teorema 3.8.1:

Sea R un autómata de pila, entonces se puede construir una gramática G, libre del contexto, tal que el lenguaje generado por G es el mismo que reconoce R por el criterio de pila vacía.

Demostración Teorema 3.8.1: Véase [8].

Los No Terminales de la gramática se representan mediante tríos conformados de la siguiente forma: $[q \quad T \quad r]$ donde:

$q \in Q$, es el estado al que está asociado el no terminal;

$T \in \Gamma$, es el símbolo que está en el tope de la pila;

$r \in Q$, es uno de los estados a los que se llega a partir del estado q con T en el tope de la pila.

Las producciones de la gramática se construyen en dependencia de la longitud de la cadena que se pone en el tope de la pila. Dada la forma de la función de transición del autómata de pila, las producciones de la gramática, siguiendo lo planteado por el teorema 3.8.1, se construyen de la siguiente forma:

1. Para todos los pares $\big((q, I, T), (r, T)\big) \in f$ se añaden G producciones de la forma: $[q\ T\ t] \rightarrow I[r\ T\ t]\ \forall\ \mathrm{t} \in Q$.
1. Para todos los pares $\big((q, I, T), (r, IT)\big) \in f$ se añaden a G producciones de la forma: $[q\ T\ t_2] \rightarrow I[r\ I\ t_1][t_1\ T\ t_2]\ \forall\ (t_1, t_2) \in Q \times Q$.
2. Para todos los pares $\big((q, I, T), (r, \varepsilon)\big) \in f$ se añaden a G producciones de la forma: $[q\ T\ r] \rightarrow I$.

Definición 3.8.1: La gramática libre del contexto asociada a un autómata booleano se denomina $\boldsymbol{Sat_{gr}}$ y se define como $Sat_{gr} = (N, \Sigma, P, D)$ donde:

N: Conjunto de no terminales de la forma $[q\ T\ r]$.

$\sum$: Conjunto de terminales. $\Sigma = \{True, False\}$.

P: Conjunto de producciones. Las producciones de la gramática son de la forma:

$A \rightarrow a$

$A \rightarrow aB$

$A \rightarrow aBC$

$D \rightarrow A$: Solo para los no terminales en los que deriva el distinguido.

D: Símbolo distinguido.

Si se analiza detenidamente la forma en la que se construyen las producciones propuesta en la demostración del teorema 3.8.1 se nota que, por cada transición del autómata de la forma $f(q, I, T) = (r, T)$ se añaden a la gramática tantas producciones como estados tenga el mismo y por cada transición de la forma $f(q, I, T) = (r, IT)$, la cantidad de producciones que se añaden es del orden del cuadrado de la cantidad de estados.

Es necesario tener en cuenta también que muchos de los no terminales nuevos que se crean ni tan siquiera aparecen en la parte izquierda de alguna de las reglas de la gramática, puesto que la forma de la función de transición del autómata de pila booleano es un grafo acíclico y dirigido donde las aristas que salen de cada nodo inciden a algún nodo del nivel siguiente y a su vez todas las que incidentes en él, provienen de nodos del nivel anterior. Para que un

símbolo no terminal $[q\ T\ r]$ aparezca en la parte izquierda de alguna producción tiene que cumplir las siguientes restricciones:

1. El nivel en el que está q tiene que ser menor que el nivel en el que está r.
2. Las configuraciones del autómata de pila tienen que permitir que estando en el estado q con el símbolo T en el tope de la pila se pueda llegar al estado r en un número finito de transiciones.

Tomando estas ideas como base se diseñó un algoritmo que construye las reglas de la gramática con la información de los no terminales que tiene hasta el momento y luego mediante un mecanismo de Retorno hacia Atrás (Back Patch) se completa la información de cada regla.

3.8.1: Algoritmo para construir una gramática libre del contexto asociada a un autómata de pila booleano.

Este algoritmo tiene dos fases; en la primera fase se construye el conjunto de producciones de la gramática y en la segunda fase se construye la gramática.

Definición 3.8.1.1: Un **no terminal** de la forma $[q\ T\ r]$ es **completo** si el valor de cada uno de sus elementos está definido.

Definición 3.8.1.2: Un **no terminal** de la forma $[q\ T\ r]$ es **parcial** si el valor de alguno de sus elementos no ha sido todavía. Los valores que no se conocen se representan con el signo $-$.

Definición 3.8.1.3: Una **regla** es **completa** si todos los no terminales que aparecen en ella son completos.

Definición 3.8.1.4: Una **regla** es **parcial** en ella existe algún no terminal parcial.

Definición 3.8.1.5: El **nivel de una regla** es el nivel del no terminal que está en la parte izquierda de la misma.

Algoritmo para construir una gramática libre del contexto asociada a un autómata de pila booleano.

Entrada: $P = (Q, \Sigma, \Gamma, f, q_0, Z_0, F)$ Autómata de pila que reconoce por el criterio de pila vacía todas las cadenas verdaderas de la fórmula de entrada.

Salida: $G = (N, \Sigma, \mathrm{P}, D)$ Gramática libre del contexto que genera el lenguaje de todas las cadenas verdaderas de la fórmula de entrada.

Fase 1: Construcción del conjunto de producciones de la gramática.

Entrada: $P = (Q, \Sigma, \Gamma, f, q_0, Z_0, F)$ Autómata de pila que reconoce por el criterio de pila vacía todas las cadenas verdaderas de la fórmula de entrada.

Salida:

H: Diccionario <No Terminal, Conjunto de reglas en los que el no terminal aparece en parte derecha>.

L: Conjunto de reglas de la forma $A \to a$.

Inicialización:

$H = \emptyset$

$L = \emptyset$

Instrucciones:

forall $((q,I,T),(r,\gamma)) \in f$ //Para todo par definido en la función de transición del autómata de pila booleano.

Caso 1:

Si el par es de la forma $((q,I,T),(r,T)) \in f$ construir una regla de la forma: $Regla = [q\ T\ -] \to I[r\ T\ -]$

$H[r\ T\ -] = H[r\ T\ -] \cup \{Regla\}$//Añadir la regla al conjunto de reglas en las que el no terminal aparece en la parte derecha.

//Los pares de la forma $((q,I,T),(r,T))$ corresponden a las transiciones del autómata de pila booleano que se construyen cuando la clasificación de la instancia de variable asociada al estado q es Libre o Intermedia. En el caso de que la instancia de variable sea intermedia se tiene que cumplir la restricción adicional de que el símbolo que se lee de la cadena de entrada es el mismo que se lee del tope de la pila.

Caso 2:

Si el par es de la forma $((q,I,T),(r,IT)) \in f$ construir una regla de la forma: $Regla = [q\ T\ -] \to I[r\ I\ -][-\ T\ -]$.

$H[r\ I\ -] = H[r\ I\ -] \cup \{Regla\}$

Caso 3:

Si el par es de la forma $((q,I,S),(r,\varepsilon)) \in f$ construir una regla de la forma: $Regla = [q\ S\ r] \to I$.

$L = L \cup \{Regla\}$ //Añadir la regla al conjunto de reglas de la forma no terminal deriva en terminal.

BackPatch($[q\ T\ r]$*, H, P)*//Completar la información de todos las reglas en el no terminal $[q\ T\ r]$ aparece en parte derecha.

Fase 2: Construcción de la gramática.

Entrada:

H: Diccionario <No Terminal, Conjunto de reglas en los que el no terminal aparece en parte derecha>.

L: Conjunto de reglas de la forma $A \to a$.

Salida:

$G = (N, \Sigma, P, D)$, gramática libre del contexto que genera el lenguaje de que reconoce el autómata P por el criterio de pila vacía.

Inicialización:

P: Conjunto de producciones de la gramática. $P = \emptyset$.

N: Conjunto de no terminales de la gramática. $N = \emptyset$.

$\Sigma = \{0,1\}$: Conjunto de terminales.

$D = [0\ Z_0\ 0]$

Instrucciones:

Paso 1:

// Añadir al conjunto de producciones todas las reglas que están en el conjunto L y añadir al conjunto de no terminales todos los no terminales que aparecen en la parte izquierda de las reglas que están en L. Como P y N son conjuntos no tienen elementos repetidos.

forall $(A \to a) \in L$

$P = P \cup \{A \to a\}$

$N = N \cup \{A\}$

Paso 2:

// Añadir al conjunto de producciones todos los no terminales y todas las reglas completas que aparecen en H. Todo par de la forma <No Terminal, Conjunto de reglas en los que el no terminal aparece en parte derecha> que pertenece a H se representa por $< n, R >$.

foreall $< n, R > \in H$

c = 0 //Cantidad de reglas que se añaden.

forall $regla \in R$

Si regla es completa entonces $P = P \cup \{regla\};\ c = c + 1$

if $c > 0$ then $N = N \cup \{n\}$

Paso 3:

// El símbolo distinguido de la gramática deriva en todas las reglas de nivel cero.

forall $(A \to \alpha) \in P$ $//\alpha \in (N \cup \Sigma)^*$

Si el nivel de la regla es cero entonces $P = P \cup \{D \to A\}$

return $G = (N, \Sigma, P, D)$

En la fase de construcción del conjunto de producciones de la gramática que genera el lenguaje que reconoce el autómata de pila booleano se emplea un mecanismo de retorno hacia atrás (BackPatch) que tiene como objetivo completar la información de las reglas de la gramática.

Algoritmo de BackPatch.

Entrada:

$[q\ T\ r]$: No Terminal.

H: Diccionario <No Terminal, Conjunto de reglas en los que el no terminal aparece en parte derecha>.

Intrucciones:

forall $Regla \in H[q\ T -]$

Caso 1:

Si la regla es de la forma $[p\ T\ -] \to I[q\ T -]$ entonces se actualiza de la siguiente forma: $[p\ T\ r] \to I[q\ T\ r]$.

BackPatch($[p\ T\ r]$, H, P) //Se continúa el proceso de actualización a partir de la nueva información obtenida.

Caso 2:

Si la regla es de la forma $[p\ Z\ -] \to I[q\ T -][-\ Z\ -]$ entonces se actualiza de la siguiente forma: $[p\ Z\ -] \to I[q\ T\ r][r\ Z\ -]$

$H[r\ Z\ -] = H[r\ Z\ -] \cup \{Regla\}$ //Como ya se tiene información suficiente del segundo segundo no terminal de la parte derecha, se añade al diccionario.

Caso 3:

Si la regla es de la forma $[p\ T\ -] \to I[t\ I\ q][q\ T -]$ entonces se actualiza de la siguiente forma: $[p\ S\ r] \to I[t\ I\ q][q\ T\ r]$.

BackPatch($[p\ S\ r]$, H, P) //Se continúa el proceso de actualización a partir de la nueva información obtenida.

Para analizar el orden computacional de este algoritmo primeramente hay que tener en cuenta que desde un estado del autómata solo se pueden realizar 6 transiciones, 2 símbolos de

entrada × 3 símbolos de pila. Luego la cantidad de reglas parciales que se crean en la primera fase del algoritmo para construir la gramática asociada a un autómata de pila booleano es a lo sumo $6m$, donde *m* es la cantidad de estados del autómata.

En el algoritmo de BackPatch también pueden adicionarse nuevas reglas a la gramática, esto ocurre cuando la regla que hay que actualizar ya fue actualizada y en ese caso se crea una copia de la misma, se actualiza la copia y se continúa el proceso. ¿Cuántas reglas se adicionan? Lo peor que puede pasar es que cuando un no terminal vaya a actualizar las reglas en las que aparece en parte derecha, éstas ya hayan sido actualizadas, luego como consecuencia se crear una regla adicional por cada una de las reglas en las que el no terminal aparece en la parte derecha.

¿En cuántas reglas aparece un no terminal en la parte derecha? Esto depende del ancho del autómata. El **ancho del autómata** es la cantidad máxima de estados que hay en un nivel del autómata. Sea *k* el ancho del autómata, luego en todos los niveles hay a lo sumo *k* estados, por lo tanto suponiendo que todos los no terminales que se puedan formar a partir de estos estados derivaran en reglas que tienen al no terminal en cuestión en parte derecha, la cantidad máxima de reglas en las que puede estar un no terminal en la parte derecha es $6k$.

Lema 3.8.1: El ancho de un autómata booleano es del orden de la longitud de la fórmula de entrada a partir de la cual se construye el autómata.

Demostración Lema 3.8.1:

El ancho del autómata asociado a una instancia de variable es dos por la definición 3.3.2. El ancho de un autómata solo puede aumentar como resultado de un proceso de concatenación que es cuando se añaden al mismo los caminos de extensión a los estados Verdadero y/o Falso. En cada concatenación se construye un solo camino de extensión por definición 3.3.4. Como en el proceso de construcción del autómta booleano se realizan $n-1$ concatenaciones donde *n* es la longitud de la fórmula de entrada y por cada una de estas el nivel del autómata puede aumentar a lo sumo en uno, entonces el ancho del autómata es menor que *n*, por tanto pertenece a $O(n)$. ■

Luego la cantidad de reglas de la gramática es $O(nm)$, donde *n* es la longitud de la fórmula de entrada y *m* la cantidad de estados del autómata booleano. Como $m \in O(n^2)$ entonces la cantidad máxima de reglas que puede tener la gramática es $O(n^3)$. El algoritmo de BackPatch recorre todas las reglas de la gramática para realizar el proceso de actualización. Por lo el costo computacional del proceso completo de la construcción de la gramática libre del contexto asociada a un autómata de pila booleano es del Orden de la Cantidad de Producciones.

En la práctica, la cantidad de transiciones es mucho menor que $O(nm)$ porque una vez que se eliminan las trasiciones al estado *Falso* del autómata booleano se tiene que no todos los estados tienen dos transiciones posibles y no a todos los estados se puede llegar con todos los símbolos de pila. Además como el autómata booleano extendido aprovecha los caminos de

extensión previamente construidos, siempre que sea posible, el ancho del autómata es estrictamente menor que la longitud de la fórmula. $O(n^3)$ es una cota burda para el algoritmo, en estos momentos se está trabajando en un análisis amortizado de la complejidad computacional del mismo.

3.8.2 Problema del Vacío.

Después de construir la gramática libre del contexto Sat_{gr} que genera el lenguaje de todas las cadenas verdaderas de una fórmula lógica, para determinar si ésta es satisfacible o no, solo hace falta determinar si el lenguaje de todas sus cadenas verdaderas es vacío o no. Si el lenguaje no es vacío entonces existe al menos una interpretación verdadera para la fórmula de entrada por lo que esta es satisfacible. Luego, la fórmula es satisfacible si y sólo si el lenguaje generado por Sat_{gr} ***NO*** es vacío.

Para determinar si el lenguaje generado por Sat_{gr} es vacío o no se puede emplear el algoritmo para determinar cuando una gramática libre del contexto es vacía desarrollado por Hopcroft, Motwani y Ullman [9]. Este algoritmo está diseñado para resolver un problema general, por lo que es interesante analizar la forma de la gramática obtenida en busca de particularidades que puedan hacer aún más eficiente el proceso de determinar si el lenguaje generado por SAT_{gr} es vacío o no.

Recordando el proceso de construcción de la gramática se nota que las producciones de la misma está asociadas a los tuplos de la función de transición del autómata de pila boleano y regresando más atrás se observa que la función de transición de este autómata es un grafo acíclico y dirigido donde todos los caminos desde el nodo que representa al estado inicial hasta el nodo que representa al estado final tienen la misma longitud. Además como cada estado está asociado a una variable dentro de la secuencia, lo que determina el nivel del estado, y las transiciones se realizan de un estado que está en el nivel *k* hacia otro que está en el nivel *k+1*, se tiene que la función de transición del autómata booleano es un grafo por niveles; luego las reglas de la gramática, que de manera general tienen la siguiente forma: $[p\ Z\ r] \rightarrow I[s\ X\ q][q\ Y\ r]$, cumplen que $Nivel(p) < Nivel(s)$ y $Nivel(s) < Nivel(q)$.

Para determinar si el lenguaje generado por una gramática Sat_{gr} es vacío o no, se ordenan decrecientemente las reglas de la gramática teniendo en cuenta el nivel de cada una. Se analizan cada una de las reglas para determinar si producen y como existe un orden de nivel entre ellas cuando se esté analizando la regla $A \rightarrow aBC$ ya se sabe si los no terminales *B* y *C* producen, porque como estos están en un nivel superior al de *A*, ya fueron analizados.

Algoritmo para determinar si el lenguaje generado por la gramática asociada a un autómata de pila booleano es vacío.

Entrada: Gramática Libre del Contexto asociada a un Autómata de Pila Booleano. $G = (N, \Sigma, P, D)$

Salida:

Si, el lenguaje asociado a la gramática es vacío.

No, el lenguaje asociado a la gramática NO es vacío.

Inicialización:

P: conjunto ordenado decreciente por niveles de las producciones de la gramática G.

R: conjunto de los no terminales que producen. Un no terminal produce si a partir de él se puede obtener una cadena formada solo por símbolos terminales del lenguaje. $R = \emptyset$

Instrucciones:

forall $regla \in P$

if $regla = [q\ T\ r] \rightarrow I$ then //Si el no terminal de la parte izquierda de la regla deriva en terminal entonces produce.

$R = R \cup \{[q\ T\ r]\}$

else if $regla = [q\ T\ r] \rightarrow I[p\ T\ r]$ then // Si el no terminal de la parte izquierda de la regla deriva en un no terminal que produce entonces produce. En caso contrario se puede eliminar la regla del conjunto de reglas porque no va ser utilizada para generar cadenas del lenguaje.

if $[p\ T\ r] \in R$ then $R = R \cup \{[q\ T\ r]\}$

else $P = P - \{regla\}$

else $regla = [q\ T\ r] \rightarrow I[p\ I\ t][t\ T\ r]$ then // Si el no terminal de la parte izquierda de la regla deriva en dos no terminales que producen entonces produce. En caso contrario se puede eliminar la regla del conjunto de reglas porque no va ser utilizada para generar cadenas del lenguaje.

if $[p\ I\ t], [t\ T\ r] \in R$ then $R = R \cup \{[q\ T\ r]\}$

else $P = P - \{regla\}$

if $D \in R$ then //Si el distinguido produce entonces el lenguaje generado por la gramática no es vacío.

return **No**

else return Si

El orden computacional de este algoritmo es $O(Cantidad\ de\ producciones)$ puesto que se analizan todas las producciones y para cada una de ellas se realizan operaciones que requieren un tiempo constante.

3.8.3: Limpieza de la gramática.

Si el lenguaje generado por la gramática Sat_{gr} no es vacío entonces la fórmula es satisfacible. Además, el lenguaje de todas sus cadenas verdaderas es el lenguaje generado por Sat_{gr}. El trabajo pudiera acabarse aquí, pero sería bueno someter a la gramática a un proceso de limpieza; es decir, eliminar todos los no terminales que no son alcanzables desde el distinguido. El proceso de limpieza tiene un costo computacional del orden de la cantidad de reglas de la gramática.

Algoritmo de limpieza de la gramática.

Entrada: Gramática Libre del Contexto asociada a un Autómata de Pila Booleano. $G = (N, \Sigma, \mathrm{P}, D)$.

Inicialización:

P: conjunto ordenado creciente por niveles de las producciones de la gramática G

R: conjunto de los no terminales alcanzables. $R = \{D\}$. //Inicialmente el único no terminal alcanzable es el distinguido.

Instrucciones:

forall $A \to aBC \in P$ //Se analizan todas las reglas de la gramática para determinar cúales son los no terminales alcanzables.

if $A \in R$ then //Si el no terminal que está en la parte izquierda de la regla es alcanzable entonces los no terminales (uno o dos, en caso de que los halla) de su parte derecha son alcanzables.

$R = R \cup \{B\}$

$R = R \cup \{C\}$

else

$P = P - \{A \to aBC\}$ //Si el no terminal que está en la parte izquierda de la regla no es alcanzable entonces se puede eliminar la regla del conjunto de producciones porque no se va a utilizar para producir ninguna oración del lenguaje.

El orden computacional de este algoritmo es $O(Cantidad\ de\ producciones)$ puesto que se analizan todas las producciones, previamente organizadas por niveles, y para cada una de ellas se realizan operaciones que requieren un tiempo constante.

3.8.4: Contar la cantidad de soluciones.

Después de aplicarle a la gramática que genera el lenguaje de todas las cadenas verdaderas el algoritmo de limpieza expuesto en la sección anterior, se cuenta con una gramática donde todas sus producciones son alcanzables desde el símbolo distinguido y todas se emplean para generar las cadenas del lenguaje satisfacible asociado a la fórmula de entrada.

Para contar la cantidad de interpretaciones verdaderas que tiene la fórmula de entrada se ordenan las reglas de la gramática en orden decreciente de sus niveles.

Algoritmo para contar la cantidad de soluciones.

Entrada: Gramática Libre del Contexto asociada a un Autómata de Pila Booleano. $G = (N, \Sigma, \mathrm{P}, D)$.

Salida: Cantidad de cadenas que pertenecen al lenguaje generado por G.

Inicialización:

P: conjunto ordenado decreciente por niveles de las producciones de la gramática G

H: diccionario <Non terminal, cantidad de soluciones>

Instrucciones:

forall $regla \in P$

//Por cada regla de la forma $[q\ T\ r] \to I$ la cantidad de oraciones que produce el no terminal $[q\ T\ r]$ aumenta en uno.

if $regla = [q\ T\ r] \to I$ then $H[q\ T\ r] = H[q\ T\ r] + 1$

//Por cada regla de la forma $[q\ T\ r] \to I[p\ T\ r]$ la cantidad de oraciones que produce el no terminal $[q\ T\ r]$ aumenta en la cantidad de oraciones que produce $[p\ T\ r]$.

else if $regla = [q\ T\ r] \to I[p\ T\ r]$ then

$$H[q\ T\ r] = H[q\ T\ r] + H[p\ T\ r]$$

//Por cada regla de la forma $[q\ T\ r] \to I[p\ I\ t][t\ T\ r]$ la cantidad de oraciones que produce el no terminal $[q\ T\ r]$ aumenta en el producto de la cantidad de oraciones que producen los no terminales $[p\ I\ t]$ y $[p\ T\ r]$.

else $regla = [q\ T\ r] \to I[p\ I\ t][t\ T\ r]$ then

$$H[q\ T\ r] = H[q\ T\ r] + H[p\ I\ t] * [t\ T\ r]$$

return $H[D]$ //La cantidad de oraciones que produce la gramática es la cantida de oraciones que produce el símbolo distinguido.

El orden computacional de este algoritmo es $O(Cantidad\ de\ producciones)$ puesto que se analizan todas las producciones y para cada una de ellas se realizan operaciones que requieren un tiempo constante.

3.9 K_SAT Libre del Contexto.

Aunque en las secciones anteriores se explicó cómo resolver el problema de la satisfacibilidad libre del coxtexto, en esta sección se realiza un análisis detallado del mismo para el caso en que la entrada es una fórmula en forma normal conjuntiva. Las razones:

1. La mayor parte de las investigaciones realizadas sobre el problema de la satisfacibilidad booleana parten de una fórmula lógica en forma normal conjuntiva.
2. El algortimo propuesto para resolver el problema de la satisfacibilidad booleana libre del contexto cuando la fórmula está en forma normal conjuntiva tiene un costo computacional lineal de la longitud de la fórmula de entrada.

Lema 3.9.1: Toda fórmula lógica se puede representar mediante un árbol binario.

Demostración Lema 3.9.1

La demostración es constructiva. Tomando como base las reglas de formación de fórmulas el árbol asociado a una fórmula lógica se define recursivamente como sigue:

1. El árbol asociado a una instancia de variable es un nodo que identifica a la instancia de variable.
2. Si A es el árbol asociado a la fórmula F entonces el árbol asociado a la fórmula $\neg F$ es un árbol que tiene como raíz un nodo que representa al operador negación ($\neg$) y que tiene como único hijo a A.
3. Si A_1 y A_2 son los árboles asociados a las fórmulas F_1 y F_2 respectivamente, entonces, el árbol asociado a:

   3.1. $F_1 \wedge F_2$ es el árbol que tiene como raíz un nodo que representa al operador conjunción ($\wedge$), como hijo izquierdo a A_1 y como hijo derecho a A_2.

   3.2. $F_1 \vee F_2$ es el árbol que tiene como raíz un nodo que representa al operador disyunción ($\vee$), como hijo izquierdo a A_1 y como hijo derecho a A_2.■

Una representación arbórea para una fórmula refleja el orden en que se realizan las operaciones en el momento de evaluar la fórmula. Teniedo en cuenta que el proceso de evaluación de una fórmula se realiza de izquierda a derecha entonces en el árbol, los nodos que están más a la izquierda son los primeros en evaluarse.

Definición 3.9.1: El **árbol asociado a un literal** es un árbol que tiene un único nodo que representa al literal.

La definición anterior no cumple con el modelo de construcción del árbol asociado a una fórmula propuesto en la demostración del Lema 3.9.1; pero teniendo en cuenta que ésta representación se utiliza para ilustrar la construcción del autómata booleano extendido asociado a una fórmula y el autómata asociado a un literal tiene siempre tres estados independientemente de que el literal esté negado o no (Véanse definiciones 3.3.2 y 3.3.3).

Definición 3.9.2: El **árbol asociado a una cláusula** $l_1 \vee l_2 \vee \ldots l_{k-1} \vee l_k$ es un árbol que tiene como raíz al operador disyunción, como hijo derecho al árbol asociado al literal l_k y como hijo izquierdo al árbol asociado a la cláusula $l_1 \vee l_2 \vee \ldots l_{k-1}$. Si la cláusula tiene un solo literal entonces el árbol asociado a la cláusula es el árbol asociado al literal.

Definición 3.9.3: El **árbol asociado a una fórmula en forma normal conjuntiva** $c_1 \vee \ldots \vee c_{m-1} \vee c_m$ es un árbol que tiene como raíz al operador conjunción, como hijo derecho al árbol asociado a la cláusula c_m y como hijo izquierdo al árbol asociado a la fórmula $c_1 \vee \ldots \vee c_{m-1}$. Si la fórmula tiene una sola cláusula entonces el árbol asociado a la fórmula es el árbol asociado al literal.

Lema 3.9.2: Sea *c* una cláusula de tamaño *k* y *B* el autómata booleano extendido asociado a ella, entonces el estado TodosVerdaderos de B está en el nivel 1 y el estado TodosFalsos de *B* está en el nivel *k*.

Demostración Lema 3.9.2

La demostración se realiza por inducción en la longitud de la cláusula.

Hipótesis: Lema 3.9.2.

Para $k = 1$. El autómata booleano extendido asociado a un literal tiene tamaño tres. Los estados TodosVerdaderos y TodosFalsos están en el nivel 1. [Véanse definiciones 3.3.2 y 3.3.3].

Supóngase que la hipótesis se cumple para $k = n$.

Sea $k = n + 1$. Tómese una cláusula c de longitud *n* y sea *B* el autómata booleano extendido asociado a ella. Por hipótesis de inducción el nivel del estado *TodosVerdaderos* de *B* es 1 y el nivel de estado *TodosFalsos* de *B* es *n*.

Añadáse a *c* un literal *l* para formar la cláusula $\bar{c} = c \vee l$. La longitud de $\bar{c}$ es $n + 1$ por definición 1.6.3. El árbol asociado a $\bar{c}$ es un árbol que tiene como raíz al operador disyunción, como hijo izquierdo el árbol asociado a *c* y como hijo derecho el árbol asociado a *l*. Para construir el autómta booleano asociado a $\bar{c}$ hay que concatenar el autómata booleano asociado a *c* con el autómata booleano extendido asociado a *l*, denotado por *L*. Como consecuencia del proceso de concatenación:

1. Se fusiona el estado inicial de *L* con el estado Falso de *B*, por lo que se pierde un estado.
2. Se construye un camino de extensión de estado *Verdadero* de *B* hasta el estado *TodosVerdaderos* de *L*. Como el estado *TodosVerdaderos* de *L* está en el nivel 1 no se añade ningún estado adicional al autómata.
3. Sea *C* el autómata resultante de la concatenación entre los autómatas *B* y *L*.
    a. El estado TodosVerdaderos de *C* es el estado TodosVerdaderos de *B* que por hipótesis de inducción está en el nivel 1.

b. El estado *TodosFalsos* de *C* es el estado *TodosFalsos* de *L*. El nivel del estado *TodosFalsos* de *C* aumenta en longitud de *c* por Lema 3.3.1. Por tanto el nivel del *TodosFalsos* de *C* es $n + 1$ que es la longitud de $\bar{c}$.■

Corolario 3.9.1: Sea c una cláusula. Durante el proceso de construcción del autómata booleano asociado a ella no se añade ningún estado.

Lema 3.9.3: La cantidad de estados del autómata booleano extendido asociado a una cláusula *c* de longitud *k* es $2k + 1$.

Demostración Lema 3.9.3

Para construir el autómata booleano extendido asociado a una cláusula *c* de longitud *k* se construyen los autómatas booleanos asociados a cada uno de sus literales y posteriormente se concatenan empezado por los que están más a la izquierda en el árbol que representa a la cláusula. Como la cláusula es de longitud *k* entonces hay que realizar *k-1* concatenaciones y por cada una se pierde un estado como resultado de la fusión y no se añade ninguno. [Véase la demostración del lema 3.9.2] Por tanto el autómata booleano extendido asociado a la cláusula *c* tiene $3k - (k - 1) = 2k + 1$.■

Teorema 3.9.1: Sea la fórmula $c_1 \vee \ldots \vee c_{m-1} \vee c_m$ donde cada c_i es una cláusula de longitud *k*. La cantidad de estados del autómata booleano extendido asociado a ella es de orden lineal de la longitud de la fórmula.

Demostración Teorema 3.9.1:

La demostración se realiza por inducción en el número de cláusulas de la fórmula.

Hipótesis: Teorema 3.9.1.

Tómese una fórmula de una sola cláusula. Por el Lema 3.9.3 la cantidad de estados del autómata booleano extendido asociado a ella tiene $2k + 1$ estados. $2k + 1 \in O(k)$.

Supóngase que se cumple para una fórmula *f* de *m* cláusulas. Añádasele a *f* una cláusula *c* para formar una fórmula $\bar{f}$ de $m + 1$ cláusulas. El árbol asociado a $\bar{f}$ es un árbol que tiene como raíz al operador conjunción, como hijo izquierdo el árbol asociado a *f* y como hijo derecho el árbol asociado a *c*. Para construir el autómta booleano extendido asociado a $\bar{f}$ hay que concatenar el autómata booleano asociado a *f*, denotado por *B*, con el autómata booleano asociado a *c*, denotado por *C*. Como consecuencia del proceso de concatenación se crea un camino de extensión del estado *Falso* de *B* al estado *TodosFalsos* de *C*. El nivel del estado *TodosFalsos* de *C* es *k* por Lema 3.9.2, por tanto al autómata resultante se le añaden *k* estados por definición 3.3.4. Luego la cantidad de estados del autómata booleano resultante es:

$$\underbrace{O(km)}_{1} + \underbrace{2k + 1}_{2} + \underbrace{k - 1}_{3} - \underbrace{1}_{4}$$

1: Cantidad de estados del autómata *B*. Por hipótesis de inducción el autómata *B* tiene una cantidad de estados de orden lineal de la longitud de la fórmula de entrada. Para una fórmula de *m* cláusulas, donde cada cláusula tiene *k* literales la longitud de la secuencia es *km*.

2: Cantidad de estados del autómata asociado a una cláusula. Por lema 3.9.3.

3: Cantidad de estados que se añaden al autómata resultante en el camino de extensión.

4: Estado que se pierde como consecuencia de la fusión entre el estado *Verdadero* de *B* y el estado inicial de *C*.

$$O(km) + 2k + 1 + k - 1 - 1 \in O(k(m+1))\blacksquare$$

En el autómata booleano extendido asociado a una fórmula en forma normal conjuntiva solo se adicionan estados al autómata resultante cuando se construyen los caminos de extensión a estado Falso. Cuando se aplique el algoritmo que elimina las transiciones al estado Falso se elimina el camino de extensión creado. Luego, la cantidad de estados del autómata booleano se reduce a:

$$\underbrace{(2k+1)}_{1}\underbrace{m}_{2} - \underbrace{(m-1)}_{3} = 2km + 1 \in O(km) = O(longitud\ de\ la\ fórmula)$$

1: Cantidad de estados del autómata asociado a cada cláusula. Por lema 3.9.3.

2: Cantidad de cláusulas.

3: Se pierde un estado cada vez que se concatena una cláusula al autómata booleano extendido resultante. Como se realizan $m-1$ concatenaciones se pierde $m-1$ estados en total.

Al eliminarse el camino es extensión al estado Falso, el ancho del autómata booleano resultante se reduce a dos.

Haciendo uso de la proposición 3.4.1 se dice que la cantidad de transiciones del autómata booleano extendido es lineal de la longitud de la fórmula de entrada.

Como la cantidad de reglas de la gramática que genera el lenguaje de todas las interpretaciones verdaderas de la fórmula de entrada es del orden del ancho del autómata por la cantidad de estados, para una fórmula en forma normal conjuntiva, la cantidad de reglas es de orden lineal de la longitud de la fórmula de entrada y por tanto el proceso completo es de orden lineal de la longitud de la fórmula de entrada.

Capítulo 4 Detalles de Implementación.

Para determinar si una fórmula lógica, cuyas variables aparecen en un orden libre del contexto, es satisfacible, se construye una gramática libre del contexto *G* que genera el lenguaje de todas las cadenas verdaderas de la fórmula y se verifica si el lenguaje generado por *G* es vacío o no. En caso de ser vacío, la fórmula no es satisfacible. Si el lenguaje no es vacío se obtiene un generador de todas las cadenas verdaderas de la fórmula y la cantidad de interpretaciones verdaderas de la misma.

Dado un autómata de pila que reconozca por el criterio de pila vacía el lenguaje de todas las cadenas verdaderas de la fórmula de entrada, se construye, a partir de él, una gramática libre del contexto, tal que el lenguaje generado por la gramática es el mismo que reconoce el autómata por el criterio de pila vacía.

Este autómata de pila se construye a partir del autómata booleano asociado a la fórmula lógica y una pila en la que se almacena el valor que toma cada variable. Un autómata booleano es un autómata finito que reconoce todas las interpretaciones verdaderas de la fórmula asumiendo que todas las variables que aparecen en ella son distintas.

Se realizó una implementación de este algoritmo en la plataforma **.NET, Framework 2.0**, empleando **C#** como lenguaje de programación. En estos momentos se está trabajando en una implementación en **Python** del mismo algoritmo con el fin de obtener aplicación portable a otras plataformas.

Estrategia de Solución.

1. Determinar si la entrada constituye una fórmula válida.
2. Verificar si la secuencia de variables de la fórmula de entrada cumple la restricción de orden libre del contexto.
3. Construir el autómata booleano asociado a la entrada.
4. Construir una gramática libre del contexto que genere el lenguaje de todas las interpretaciones verdaderas a parir del autómata booleano.
5. Determinar si el lenguaje generado por la gramática es vacío o no.
6. Eliminar de la gramática los no terminales inalcanzables.
7. Contar la cantidad de cadenas que genera la gramática que son TODAS las interpretaciones verdaderas de la fórmula de entrada.

El proceso de solución se divide en dos fases, la primera de estas comprende los pasos del 1 al 3 y la segunda el resto. A la primera fase se le llama *Fase de Compilación* porque se establece cierta analogía entre los pasos que se realizan en esta fase y los que se llevan a cabo

durante un proceso de compilación. Como resultado de la misma se obtiene un autómata booleano extendido, que es un caso particular de un autómata finito, que reconoce todas las cadenas que corresponden con interpretaciones verdaderas de la fórmula de entrada, asumiendo que todas las variables presentes en ésta son distintas.

La segunda fase se le llama *Fase de Construcción* porque en esta fase se construye la gramática que genera el lenguaje de TODAS las cadenas verdaderas de la fórmula de entrada.

La analogía que se establece entre las fases de un compilador y los pasos que se realizan en la primera fase es la siguiente:

1: Análisis Lexicográfico: Agrupar los símbolos de entrada en unidades mínimas a la que se denominan Tokens. Los tokens válidos de la entrada son:

- Operadores Lógicos:
  - "and" y "&" para el operador $\wedge$
  - "or" y "|" para el operador $\vee$
  - "not" y "-" para el operador $\neg$
- Paréntesis: "(" ,")".
- Identificadores: Secuencia de letras y dígitos que comienza con letra. Los identificadores se emplean para nombrar a las variables. Las únicas secuencias que no son aceptadas son las cadenas "and", "or", "not" porque representan a los operadores lógicos.

2: Análisis Sintáctico: Determinar si la entrada es una fórmula lógica válida. En este paso se verifica que la entrada cumpla con las reglas de formación de fórmulas (Definición 1.1.2). Como resultado de este paso se obtiene además el árbol de sintaxis abstracta asociado a la fórmula de entrada.

3: Análisis Semántico: Verificar que la secuencia de instancias de variables de la fórmula de entrada cumpla la restricción de orden libre del contexto.

4: Generación de código: Construir el autómata booleano extendido asociado a la fórmula de entrada.

4.1 Análisis Lexicográfico y Sintáctico.

Una vez que han sido agrupados en tokens los símbolos de la entrada, es el momento de determinar si ésta se corresponde con una fórmula lógica válida; o lo que es lo mismo, si la entrada cumple con las reglas de formación de fórmulas. Si decimos que $\bar{F}$ es el lenguaje de todas las fórmulas válidas entonces se puede construir una gramática G_F que genere $\bar{F}$. Sea f una cadena de tokens, determinar si es una fórmula lógica es lo mismo que responder a la

pregunta: $¿f \in L(G_F)?$, esta pregunta es conocida como el problema de la palabra. La función del analizador sintáctico dar respuesta al problema de la palabra.

La gramática G_F se construye a partir de las reglas de formación de fórmulas y se define como sigue: $G_F = (N, \Sigma, P, fórmula)$ donde:

$N = \{fórmula, término, factor\}$

$\sum$: *Conjunto de tokens válidos:*

- ✓ *Operadores lógicos: "AND", "NOT", "OR"*
- ✓ *Paréntesis: "LP", "PR"*
- ✓ *Identificadores de variables: "ID"*

P: $\langle fórmula \rangle ::= \langle fórmula \rangle\ OR\ \langle término \rangle$
$\langle término \rangle ::= \langle término \rangle\ AND\ \langle factor \rangle$
$\langle factor \rangle ::= [NOT](\ ID \mid LP\ \langle fórmula \rangle\ RP)$

Las producciones de la gramática aparecen en notación BNF (Véase epígrafe 2.2.1).
Para realizar el análisis lexicográfico y sintáctico se empleó ANTLR, que es una herramienta para la generación automática de analizadores lexicográficos, sintácticos y semánticos. Las técnicas empleadas para realizar este tipo de análisis son técnicas LL(k) o LR(k). Los analizadores LR (k) son más potentes que los LL (k); esto se debe a que en los primeros ya se ha reducido parte de la entrada y mirando k tokens se sabe cual es la regla que hay que reducir; mientras que en los segundos se escoge una producción mirando los k próximos tokens. Por lo general un analizador LL (k) es más lento que uno LR (k). A favor de los analizadores LL(k) hay que destacar que son más fáciles de entender e implementar por un programador.

ANTLR emplea un analizador LL (k) extendido, conocido como analizador pred – LL (k). Este tipo de analizador tiene como ventajas más notables sobre un analizador LL (k) el hecho de que incluye predicados semánticos y lookahead de tamaño variable. Un predicado semántico es una condición que debe ser verificada durante ejecución antes de que se continúe con el proceso del análisis sintáctico. Éstos se utilizan para realizar validaciones o para evitar ambigüedades. Además se incluyen predicados sintácticos; los que se emplean para lograr un lookahead de tamaño variable.

Dado que la herramienta empleada genera analizadores LL(k) se necesita una gramática LL(k) que genere el lenguaje de todas las fórmulas lógicas. La gramática G_F es una gramática recursiva izquierda por lo que no es LL(k) para ningún k. Tomando como punto de partida la gramática G_F y realizando transformaciones sobre ésta para eliminar la recursividad izquierda se llega a la gramática $G_{\bar{F}}$ que sí es una gramática LL(k).

$G_{\bar{F}} = (N, \Sigma, P, fórmula)$ donde:

$N = \{fórmula, término, factor\}$

∑: *Conjunto de tokens válidos:*

- ✓ *Operadores lógicos: "AND", "NOT", "OR"*
- ✓ *Paréntesis: "LP", "PR"*
- ✓ *Identificadores de variables: "ID"*

P: $\langle fórmula \rangle ::= \langle término \rangle \{ OR \langle término \rangle \}$
$\langle término \rangle ::= \langle factor \rangle \{ AND \langle factor \rangle \}$
$\langle factor \rangle ::= [NOT](ID \mid LP \langle fórmula \rangle RP)$

4.2 Árbol de Sintaxis Abstracta.

Como resultado del análisis sintáctico se obtiene el árbol de sintaxis abstracta asociado a la fórmula de entrada, de ahora en adelante AST. El AST es una representación de la fórmula de entrada a partir de la cual se realizan los dos pasos restantes de la Fase de Compilación: la verificación de la restricción de orden y la construcción del autómata booleano extendido.

Las hojas del AST son las instancias de variable que aparecen en la fórmula y los nodos interiores están constituidos por los operadores conjunción, disyunción y negación.

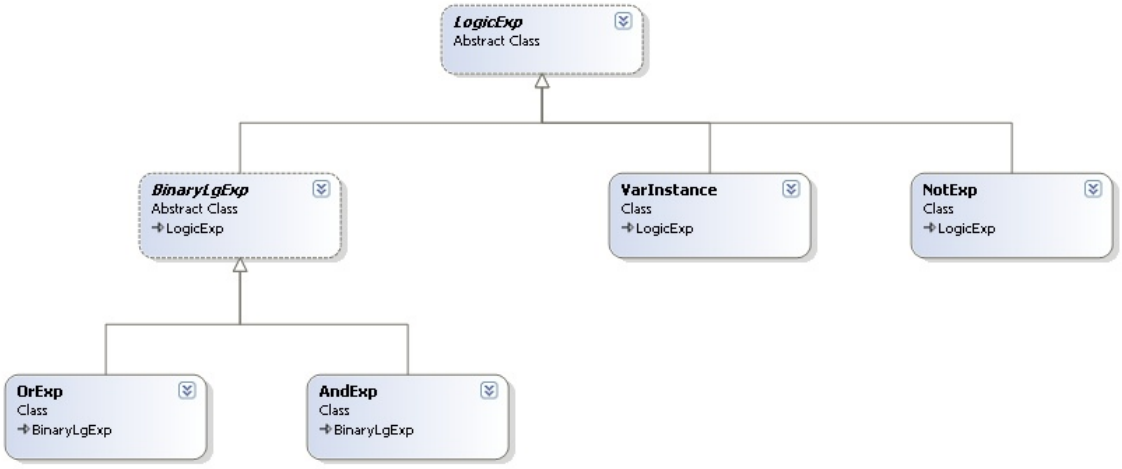

Figura 12: Jeranquía de clases de la implementación del AST.

La figura 12 ilustra la jerarquía de clases correspondiente a la implementación del AST. La clase base de esta jerarquía posee los siguientes métodos.

```
abstract void CheckContextFreeOrder(VarTable table, List<OrderError>
errors, Stack<string> stLevels);

abstract ExtendedBooleanAutomata BuildAutomata();
```

El método `CheckContextFreeOrder` se encarga de verificar que se cumpla la restricción de orden en la secuencia de variables asociada a la fórmula de entrada. Los parámetros que recibe este método son:

`VarTable table`: que es una estructura donde se almacena para cada variable cual es la primera y la última posición en la que aparece en la secuencia.

`List<OrderErrors> errors`: es una lista de los errores de orden que existen en la secuencia.

`Stack<string> stLevels`: es una pila que se usa para controlar el orden entre las instancias de variable de fórmula de entrada.

Si después de ejecutar el método la lista de errores está vacía entonces se afirma que la secuencia de variables de la fórmula de entrada es libre del contexto.

El método `BuildAutomata` tiene como función construir el autómata booleano extendido asociado a la fórmula de entrada.

4.3 Análisis Semántico: Verificar la restricción de orden libre del contexto.

La restricción de orden que se impone a la secuencia de variables de la fórmula es independiente de los operadores que aparezcan en ella. Para verificar si la secuencia es libre del contexto se recorren todas las hojas del AST, que es donde están almacenadas las variables.

Como parte del proceso de análisis semántico se construye la tabla de variables. Esta es una estructura en forma de diccionario en la que se almacenan todas las variables que aparecen en la fórmula de entrada. Los datos necesarios de una variable son:

- Identificador de la variable.
- Posición de la primera ocurrencia de la variable en la secuencia.
- Posición de la última ocurrencia de la variable en la secuencia.

A partir de esta información se pueden clasificar todas las instancias de variable de la secuencia de variables de la fórmula de entrada en cuatro categorías diferentes.

- *Libre (Free)*: La variable está representada por una única instancia.
- *Primera (First)*: A esta clase pertenecen todas las instancias de variable que corresponden a la primera ocurrencia de una variable en la fórmula.
- *Intermedia (Middle)*: Las clase intermedia agrupa a todas las instancias de variables que ocurren en más de una posición en la secuencia de variables de la fórmula pero que no ocupan ni la primera ni la última posición.
- *Última (Last)*: A esta clase pertenecen todas las instancias de variable que corresponden a la última ocurrencia de una variable en la fórmula.

Para representar estas clases se empleó un tipo enumerativo `LevelCategory{Free, First, Middle, Last}`. Esta clasificación se usa para construir el conjunto de producciones de la gramática que genera el lenguaje de todas las interpretaciones verdaderas de la fórmula de entrada.

4.4 Generación de Código: Construcción del autómata booleano extendido.

El proceso de construcción del autómata booleano extendido se realiza mediante llamados recursivos a partir de la raíz del AST hasta llegar a las hojas. El método que construye el autómata booleano extendido es el método `BuildAutomata`.

El proceso de construcción se basa en la definición de autómata booleano extendido asociado a una fórmula lógica. La función de transición del autómata booleano extendido es un grafo acíclico y dirigido por niveles donde cada nivel se corresponde con una instancia de variable de la secuencia de variables de la fórmula de entrada.

4.5 Autómata Booleano Extendido.

La versión computacional del autómata booleano extendido es la clase `ExtendedBooleanAutomata`; la cual tiene cinco campos que son los estados distinguidos del autómata booleano extendido; estos son:

- *Start*: Estado inicial del autómata.
- *FinalTrue*: Estado final del autómata.
- *FinalFalse*: Estado que representada que la cadena de entrada es una cadena falsa de la fórmula de entrada.
- *AllTrue*: Estado a partir del cual, independientemente del símbolo que se lea de la cadena de entrada, se llega al estado *FinalTrue*.
- *AllFalse*: Estado a partir del cual, independientemente del símbolo que se lea de la cadena de entrada, se llega al estado *FinalFalse*.

Los estados del autómata booleano extendido se representan mediante la clase `BooleanState`; los campos fundamentales de esta clase son:

- *Index*: Identificador del estado.
- *Level*: Nivel del estado.
- *NextTrue*: Estado al que se llega leyendo el símbolo verdadero de la cadena de entrada.
- *NextFalse*: Estado al que se llega leyendo el símbolo falso de la cadena de entrada.
- *Previous*: Lista de todos los estados que tienen transiciones a él.

La función de transición del autómata booleano extendido es un grafo acíclico y dirigido, donde los nodos del grafo son los estados del autómata y las aristas son las transiciones entre los estados. Este es un grafo por niveles, tal que cada nivel representa una instancia de variable de la secuencia de variables de la fórmula de entrada. De cada nodo del grafo parten a lo sumo dos aristas puesto que a partir de cada estado se pueden realizar solo dos transiciones, una por cada símbolo del alfabeto de entrada: {Verdadero, Falso}. Estas dos posibles transiciones se representan por los campos *NextTrue* y *NextFalse*.

El alfabeto de entrada se implementa mediante un tipo enumerativo `InputAlphabet {False, True, Epsilon}` donde:

- *True*: Representa al símbolo de entrada verdadero.
- *False*: Representa al símbolo de entrada falso.
- *Epsilon*: Representa a la cadena vacía.

El símbolo `Epsilon` del tipo `InputAlphabet` representa a la cadena vacía. Aunque teóricamente, la cadena vacía, no pertenece a ningún alfabeto, es necesario incluirlo en la versión computacional para representar las situaciones en la que no se lee ningún símbolo de la cadena de entrada.

Los métodos principales de la clase `ExtendedBooleanAutomata` son:

`void RemoveFinalFalse()`: Encargado de eliminar todas las transiciones que conducen al estado Falso del autómata booleano extendido, representado computacionalmente por el campo *FinalFalse*.

`List<Rule>[] BuildRulesSet(LevelTable lvTalble)`: Este método construye el conjunto de producciones de la gramática que genera el lenguaje de todas las interpretaciones verdaderas de la fórmula de entrada. Como parámetro recibe una estructura en la que se almacena la categoría a la que pertenece cada instancia de variable de la fórmula.

Una vez que se finaliza la construcción del autómata booleano extendido asociado a una fórmula lógica, se tiene un mecanismo que reconoce todas las interpretaciones verdaderas de la fórmula de entrada; sin embargo, si se quisieran reconocer además todas las interpretaciones falsas, bastaría con decir que el estado *FinalFalse*, es un estado final. Pero como lo que se desea es determinar es si la fórmula es satisfacible, para disminuir la cantidad de estados y por consiguiente, la cantidad de transiciones del autómata booleano extendido asociado a la fórmula de entrada, se eliminan de éste todas las transiciones que permitan reconocer interpretaciones falsas; dicho de otra forma, viendo el autómata como un grafo acíclico y dirigido, se eliminan todas los caminos que parten del nodo que corresponde al estado inicial del autómata y que llegan al estado *FinalFalse*. Para ello se emplea el método `RemoveFinalFalse`, este método no necesita parámetros.

Una vez que se ha eliminado del autómata booleano extendido todos los estados y transiciones innecesarios se aplica el método `BuildRulesSet` para construir el conjunto de producciones de la gramática que genera el lenguaje de todas las interpretaciones verdaderas. Este método constituye la implementación de la primera fase del algoritmo para construir una gramática libre del contexto asociada a un autómata de pila booleano, véase sección 3.7.1.

4.6 La gramática, los no terminales y las reglas.

La gramática que genera el lenguaje de todas las interpretaciones verdaderas es una gramática libre del contexto en la que todas las reglas son de alguna de las siguientes formas:

- $A \rightarrow B$

- $A \rightarrow a$
- $A \rightarrow aB$
- $A \rightarrow aBC$

Cada no terminal de la gramática se representa por una estructura (tipo por valor) denominada `NonTerminal` con los siguientes campos:

- *State*: Identificador del estado al que está asociado el no terminal.
- *Path*: Identificador del estado que determina el camino a seguir.
- *StSymbol*: Símbolo que está en el tope de la pila en el estado al que está asociado el no terminal. Los posibles símbolos de pila son elementos del alfabeto de pila.

El alfabeto de pila se representa por un tipo enumerativo `StackAlphabet{False, True, Epsilon, Z0}`, donde:

- *True*: Representa al símbolo de entrada verdadero.
- *False*: Representa al símbolo de entrada falso.
- *Epsilon*: Representa que la pila está vacía.
- *Z0*: Símbolo inicial de pila.

Dos no terminales son iguales si todos sus campos son iguales y son parcialmente iguales si tienen el mismo símbolo de pila (*StSymbol*) y están asociados al mismo estado (*State*). Los conceptos de igualdad e igualdad parcial son empleados por el método `BuildRulesSet`.

Las reglas de la gramática se representan mediante la clase `Rule`; como de manera general una regla de la gramática es de la forma $A \rightarrow aBC$ los campos de la clase son:

- *Left*: El no terminal *A*.
- *First*: El no terminal *B*.
- *Last*: El no terminal *C*.
- *Terminal*: El terminal a. Los terminales son elementos del alfabeto de entrada del autómata.

En el caso de que los no terminales *B* y/o *C* no aparezcan en la regla, los campos *First* y/o *Last* son estructuras `NonTerminal` vacías.

La versión computacional de la gramática asociada a un autómata booleano es la clase `SatGrammar`. Sus funcionalidades son:

`bool IsEmpty`: Este es el método retorna verdadero si el lenguaje generado por la gramática es vacío y falso en caso contrario.

`void CleanGrammar`: Este método elimina del conjunto de producciones de la gramática todas las reglas que no son alcanzables desde el distinguido.

`double AmountSolutions`: Este método cuenta la cantidad de cadenas del lenguaje generado por la gramática. Se empleó el tipo de dato `double` para contar la cantidad de

soluciones porque brinda una precisión de 16 dígitos e incluye notación decimal. Sin embargo en algunas situaciones resulta insuficiente porque hay fórmulas que tienen mayor cantidad de soluciones que el mayor número que puede ser representado mediante un `double`.

4.7 Resultados Computacionales.

Los resultados computacionales obtenidos se catalogan de satisfactorios. Para una fórmula con 6000 instancias de variables, se obtienen, en aproximadamente 2 minutos, la gramática que genera el lenguaje de todas las cadenas que corresponden a asignaciones verdaderas de la fórmula de entrada y la cantidad de cadenas que pertenecen a este lenguaje.

4.7.1 Generador de Casos de Prueba.

Para poder experimentar con fórmulas de grandes longitudes se construyó un generador aleatorio de fórmulas libres del contexto. La construcción de la fórmula se realiza en dos fases: una primera fase donde se construye la secuencia de variables de la fórmula garantizando que cumplan la restricción de orden libre del contexto y, una segunda fase en la que se añade, a la secuencia, los operadores lógicos y paréntesis.

La construcción de la secuencia de variables se realiza por "tramos", donde cada tramo corresponde a dos ocurrencias consecutivas de una misma variable. Inicialmente se tiene el conjunto de todas las variables que aparecerán en la fórmula y un único tramo correspondiente a toda la secuencia de variables. En cada iteración se toma una variable del conjunto de variables; se elimina de éste; se determina aleatoriamente la cantidad de ocurrencias de la variable en la fórmula y las posiciones que ocupa en la secuencia de variables. A partir de aquí, dos ocurrencias consecutivas de una misma variable determinan un tramo y a cada nuevo tramo se aplica el mismo proceso.

4.7.2 Resultados.

Fórmulas con paréntesis				
Longitud de la fórmula	Cantidad de Variables	Tiempo de Ejecución (segundos)	Cantidad de Estados	Cantidad de Reglas
6000	659	60	1623105	26611
6000	500	50	1500161	38499
3000	328	9	414461	13066
3000	346	8	412264	6689
1000	143	0.5	46473	2178

Fórmulas sin paréntesis				
Longitud de la fórmula	Cantidad de Variables	Tiempo de Ejecución (segundos)	Cantidad de Estados	Cantidad de Reglas
6000	587	1	15077	13205
6000	533	1	15039	13067
3000	317	0.6	7484	6634
3000	353	0.6	7564	6728
1000	156	0.3	2519	2325

Conclusiones

En este trabajo:

Se presentó una nueva clase de problema de la satisfacibilidad que puede ser resuelto en tiempo polinomial: **el problema de la satisfacibilidad booleana libre del contexto**. La diferencia fundamental con el resto de las clases solubles del problema SAT en tiempo polinomial es que no es necesario que la fórmula de entrada esté en forma normal conjuntiva. La restricción que se impone en este caso es referente al orden que tienen que cumplir las variables de la fórmula.

Se diseñó un algoritmo que resuelve en tiempo polinomial el problema de la satisfacibilidad libre del contexto, que además calcula la cantidad de intepretaciones verdaderas y construye una gramática libre del contexto que genera el lenguaje de **todas** las interpretaciones verdaderas de la fórmula de entrada.

El algoritmo propuesto para resolver el problema de la satisfacibilidad booleana libre del contexto tiene un costo computacional del orden del cubo de la longitud de la fórmula de entrada; aunque los resultados computacionales obtenidos refejan costos mucho menores. Para los problemas K-SAT, el algoritmo es lineal del la longitud de la fórmula de entrada.

Trabajos Futuros

A partir del trabajo realizado se proponen como temas para investigaciones futuras los siguientes:

Realizar un análisis amortizado del costo computacional del algoritmo.

Determinar cómo se manifientan las restricciones de orden libre del contexto en otros problemas que pertenecen a la clase NP – Completo.

Investigar otros tipos de lenguajes que permitan aplicar un razonamiento similar al usado en este trabajo. Aprovechar las características de dichos lenguajes para imponer condiciones al orden de las variables en la fórmula de entrada. Estos lenguajes deben cumplir que el problema del vacío se resuelve en tiempo polinomial y tienen que ser interceptables con un lenguaje regular.

Bibliografía y Referencias

1. **Aho, Alfred y Ullman, Jeffrey.** *The theory of parsing, translation and compiling.* s.l. : Prentice - Hall, 1972.

2. **Cook, Stephen y Mitchell, David.** Finding hard instances of the satisfiability problem: A survey. [aut. libro] Dingzhu Du, Jun Gu y Panos Pardalos. *Satisfiability Problem: Theory and Applications.* s.l. : American Mathematical Society, 1996.

3. **Cook, Stephen.** The P versus NP Problem. *Clay Mathematics Institute.* [En línea] [Citado el: 15 de abril de 2007.] http://www.claymath.org/millennium/P_vs_NP/pvsnp.pdf.

4. **Cormen, Thomas, y otros.** *Introduction to Algorithms.* Cambridge, Massachusetts : MIT Press, 2001.

5. **Cortéz, Augusto.** *Teoría de la Complejidad Computacional y Teoría de la Computabilidad.* Lima : Universidad Nacional Mayor de San Marcos, 2004.

6. **García Garrido, Luciano.** *Introducción a la Teoría de Conjuntos y a la Lógica.* Ciudad de La Habana : Univesidad de La Habana, 2002.

7. **Garey, Michael y Johnson, David.** *Computer and Intractability. A guide to the theory of NP- Completeness.* s.l. : Bell Thelephone Laboratories, 1979.

8. **Gu, Jun, y otros.** Algorithms for the satisfiability (SAT) problem: A survey . [aut. libro] Dingzhu Du, Jun Gu y Panos Pardalos. *Satisfiability Problem: Theroy and Applications.* s.l. : American Mathematical Society, 1996.

9. **Hopcroft, John, Ullman, Jeffrey y Motwani, Rajeev.** *Intruduction to Automta Theory, Language, and Computation.* s.l. : Addinson - Wesley, 2001.

10. **Navarrete, Isabel, y otros.** Teoría de Autómatas y Lenguajes Formales. *Universidad de Murcia. Departamento de Ingenieria de la Información y las Comunicaciones.* [En línea] [Citado el: 15 de abril de 2007.] http://perseo.dif.um.es/~roque/talf/Material/apuntes.pdf.

11. **Papadimitriou, Christos.** *Computational Complexity.* San Diego, California : Addison - Wesley, 1994.

12. **Portillo, JR y Rodríguez, JI.** *Un algoritmo polinomial para problemas de la satisfacibilidad.* 2001.

13. **Stoughton, Allen.** *An Introduction to Formal Language Theory that Integrates Experimentation and Proof.* Kansas : s.n., 2004.