

Benemérita Universidad Autónoma de Puebla

Facultad de Ciencias de la Computación



Tesis

Sistema para generar satisfactibilidad en fórmulas proposicionales

Presenta

Edgar Sahit Aguilar Hernández

Para obtener el título de Licenciado en Ingeniería en Ciencias de la
Computación

Asesor

Dr. Pedro Bello López

Diciembre 2022

Agradecimientos

A mi familia por el apoyo durante el tiempo de desarrollo de la licenciatura.

A mis amigos y compañeros de la Facultad de Ciencias de la Computación por su buena vibra.

A mi asesor Dr. Pedro Bello López por su amistad, compañerismo y un excelente facilitador de conocimiento para llegar a culminar este proyecto de tesis.

A mis profesores de la Licenciatura en Ingeniería en ciencias de la computación por aportar en cada curso sus conocimientos para enriquecer el trabajo desarrollado.

A Dios por darme salud.

A la Vicerrectoría de Docencia de la Benemérita Universidad Autónoma de Puebla por el apoyo para la culminación de este trabajo de tesis dentro de los proyectos investigación 2022: Diseño de algoritmos combinatorios y de inferencia lógica, del Programa Institucional para la Consolidación de los Cuerpos Académicos, en especial al cuerpo académico Algoritmos Combinatorios y Aprendizaje.

Edgar Sahit Aguilar Hernández

Índice general

Tabla de contenido

Tabla de contenido	3
Resumen.....	5
1. Introducción	7
1.1 Planteamiento del problema	8
1.2 Objetivo general y específico.....	9
2. Marco teórico	11
2.1 Simbolización de proposiciones	11
2.2 Literal.....	12
2.3 Cláusula	12
2.4 Conjunto de cláusulas.....	13
2.5 Conjunción.....	13
2.6 Negación.....	14
2.7 Disyunción	14
2.8 Forma normal conjuntiva	14
2.9 Forma normal disyuntiva.....	16
2.10 Consistencia	17
2.11 Tablas de certeza o tablas de verdad.....	19
2.12 Introducción a la NP-completitud	21
2.12.1 Las clases P y NP	22
2.12.2 Problemas NP-completos.....	22
2.13 Satisfactibilidad.....	23
3. Trabajo relacionado.....	24
3.1 El procedimiento DPLL	24
3.2 WolframAlpha.....	25
4. Algoritmo para generar satisfactibilidad en fórmulas proposicionales	27

4.1 Transformación a 1,0 y *	27
4.2 Revisión de la consistencia	28
4.2.1 Independencia	28
4.2.2 Cláusula subsumida	29
4.2.3 Generación de nuevas cláusulas	30
4.3 Generar consistencia	32
4.4 algoritmo	35
5. Sistema para generar satisfactibilidad en fórmulas proposicionales	39
5.1 Formato de la fórmula para la lectura por el sistema	40
5.2 Ejecución del sistema	41
5.3 Generador de cláusulas aleatorias	44
6. Conclusiones	48
BIBLIOGRAFÍA	50

Resumen

El problema de satisfactibilidad (SAT) es considerado el primer problema NP-completo demostrado por Stephen Cook en el año de 1971. El problema SAT es central en la teoría de la computación. En la práctica, SAT es fundamental en la resolución de problemas de razonamiento automático, en el diseño y fabricación asistido por computadora, en el desarrollo de base de datos, en el diseño de circuitos integrados, en la integración de módulos para lenguajes de alto nivel, entre otros. El problema SAT en general es un problema de decisión, lo definimos como: dada una fórmula booleana F en Forma Normal Conjuntiva (FNC), formada de un conjunto de cláusulas con n variables, se debe encontrar una asignación a dichas variables tal que la fórmula F sea satisfactible, es decir que exista al menos una asignación (modelo) de valores que haga verdadera la fórmula F .

El problema de satisfactibilidad proposicional es un caso especial del área de la Inteligencia Artificial (IA) y tiene una relación directa con la demostración Automática de Teoremas. El problema SAT es un problema típico de la clase NP-completo, por lo que muchos investigadores han tratado de identificar casos restringidos para el problema SAT, así como la versión de optimización y conteo: problemas MaxSAT y #SAT, tratando de que estos puedan ser resueltos eficientemente. Los problemas NP-completo son los problemas que caracterizan la clase de tal forma que al encontrar una solución del problema SAT se pueden resolver los demás problemas de la clase ya que se puede hacer una transformación en tiempo polinomial al problema correspondiente.

El problema de conteo de modelos de una fórmula booleana (problema #SAT) se puede reducir a diferentes problemas de razonamiento. Por ejemplo, para estimar el grado de confianza de una red de comunicación, para calcular el grado de confianza de la teoría proposicional, para la generación de argumentaciones, para consultas proposicionales, en inferencia Bayesiana, para reparar bases de datos inconsistentes.

De esta forma el problema a tratar en este trabajo de tesis es proponer un sistema para #SAT modelando una fórmula proposicional en FNC y generar una propuesta para la consistencia de F cuando no es consistente. Esto es, se presenta un algoritmo y su implementación computacional para contar el número de asignaciones (modelos) de una fórmula F en FNC, lo que significa que, si la fórmula tiene modelos, entonces F es consistente o satisfactible y en caso contrario se aplica un algoritmo que genera una tabla donde se indica que cláusula

es necesario quitar para lograr la satisfactibilidad de la fórmula. Aplicamos la programación en Python en la implementación de los algoritmos desarrollados, lo cual ayuda al mejor entendimiento de este tipo de problemas complejos.

A continuación, se describe la organización de la tesis.

- En el capítulo 1 se da una introducción al problema de estudio, así como el objetivo general y los objetivos particulares que enmarcan el desarrollo de este trabajo de tesis.
- En el capítulo 2 se describen los conceptos básicos requeridos para plantear el trabajo de investigación en el diseño de un algoritmo y sistema para la revisión de la consistencia de una fórmula proposicional en forma normal conjuntiva, se incluyen los temas de lógica proposicional.
- En el capítulo 3 se presenta el trabajo relacionado con la revisión de la consistencia en una fórmula proposicional y el conteo de modelos, así como determinar que cláusula eliminar para que sea satisfactible.
- En el capítulo 4 se presenta el diseño del algoritmo para generar satisfactibilidad en fórmulas proposicionales. Se describe cada una de las fases de la propuesta algorítmica.
- En el capítulo 5 se presenta el diseño del sistema para verificar la consistencia de la fórmula proposicional en FNC, además se presentan las pruebas de ejecución.
- En el capítulo 6 se presentan las conclusiones del trabajo desarrollado, las principales aportaciones y las mejoras al trabajo.

Capítulo 1

1. Introducción

Es claro que el avance de los algoritmos de razonamiento automático se basa en el diseño de algoritmos eficientes para resolver los problemas de Satisfactibilidad (SAT) y sus versiones de optimización (Máxima Satisfactibilidad) y conteo (#SAT). Aun cuando es reconocida la complejidad exponencial de estos problemas, es un problema abierto el reconocer las subclases de fórmulas donde tales problemas pueden resolverse eficientemente.

Existen varias aplicaciones de IA tal como planeación, sistemas expertos, minería de datos, razonamiento aproximado, etc. #SAT es tan difícil como el problema de decisión SAT, pero en muchos casos, aun cuando SAT es resuelto en tiempo polinomial, aun no existe un método eficiente conocido para #SAT. Por ejemplo, el problema 2-SAT, que es el problema SAT restringido a conjunciones de a lo más 2 literales por cláusula, se puede resolver en tiempo lineal. Sin embargo, el correspondiente problema de conteo #2-SAT es un problema #P-completo. Por lo que se consideran restricciones de #SAT para fórmulas monótonas, 2-HORN, y las clases de fórmulas $(2, 3\mu)$ -CF, $(2, 3\mu)$ -MON, $(2, 3\mu)$ -HORN (conjunción de cláusulas Monótonas y Horn, respectivamente, donde cada variable aparece a lo más tres veces), los cuales son todos problemas #P-completos, mientras que su respectiva versión SAT son resueltos eficientemente.

Existen diversas líneas de investigación en torno a estos problemas y las aplicaciones son muy variadas. Por lo que damos prioridad a estudiar diversos tópicos considerando el área de combinatoria centrándonos en algunas de las líneas de aplicaciones en particular problemas de satisfactibilidad, en particular en el conteo de modelos de una fórmula proposicional y la propuesta de generar satisfactibilidad en dicha fórmula cuando al evaluarla no es satisfactible.

El hablar de inteligencia artificial también implica el hablar de ciertas estructuras, una de ellas son las bases de conocimiento [1,12], desde hace varios años, entre las aplicaciones de mayor impacto en la inteligencia artificial aplicado a la industria o a la sociedad, son los sistemas basados en sistemas expertos y la teoría de revisión de creencias. En este tipo de sistemas: la presentación, revisión y poder mantener actualizada la base de conocimiento son una tarea preponderante en este tipo de sistemas basados en conocimiento.

Una base de conocimiento como se define en [2], por lo que es una forma de representar el conocimiento que un ser racional va adquiriendo a través del estudio, del tiempo, por experiencias o ya sea simplemente por lo que se vivió, tomado esto en cuenta cuando se recibe nuevo conocimiento la base de conocimiento se tiene que mantener actualizada, ya que si se da una condición conocida como contradicción con el conocimiento ya registrado entonces tendremos una inconsistencia lógica de la base de conocimiento algo que se debe de evitar y si se da el caso de una inconsistencia se tienen que aplicar medidas correctivas para mantener la consistencia lógica de la base de conocimiento. En este trabajo consideramos la base de conocimiento modelada como una fórmula proposicional en Forma Normal Conjuntiva (FNC), considerando además que existen mecanismos para convertir una fórmula proposicional a su FNC. Una FNC nos permite un mejor tratamiento computacional.

Como sabemos, muchos sistemas de inteligencia artificial como las redes neuronales toman como base el funcionamiento del cerebro humano y las bases de conocimiento no son la excepción, y en busca de como el cerebro representa su base de conocimiento se ha planteado modelar las bases de conocimiento mediante símbolos y conectores de tipo lógico, a la que llamaremos proposición. Como sabemos una proposición es una afirmación matemática organizada de tal forma que me permite determinar si es verdadera o falsa.

1.1 Planteamiento del problema

Recordando que el problema de satisfactibilidad (SAT) en general es un problema de decisión, se define como: dada una fórmula booleana F en forma normal conjuntiva (FNC), formada de un conjunto de cláusulas con n variables, se debe encontrar una asignación a dichas variables tal que F sea satisfactible, es decir que exista al menos una asignación de valores (a lo que llamamos modelos) que haga verdadera la fórmula F [3]. El problema del conteo #SAT consiste en contar un número de asignaciones (número de modelos en nuestro caso) que satisfacen a F [3].

El problema que se trata en este trabajo de tesis es proponer un sistema para #SAT para una fórmula F proposicional en FNC y al mismo tiempo se genera una propuesta para determinar que cláusula permite generar consistencia en F para el caso cuando F es inconsistente (no tiene modelos). Esto es, presentar un sistema en un lenguaje de programación como es Python para contar el número de asignaciones (modelos) de una fórmula F en FNC, lo que significa que, si la

fórmula tiene modelos, entonces la fórmula es consistente. Puede darse el caso de que la fórmula no tenga modelos, en otras palabras, que no sea consistente, en tal caso se tendrá que eliminar alguna cláusula y mantener la consistencia de dicha base de conocimiento, este proceso de generación de satisfactibilidad se puede apreciar en la Figura 1.1. El proceso inicia con la lectura de la fórmula F en FNC y la revisamos con el algoritmo propuesto de satisfactibilidad donde se obtiene el número de modelos (TM) de F . Si $TM > 0$ significa que F es satisfactible, si $TM = 0$ significa que la fórmula F es inconsistente por lo que se aplica el proceso para recuperar la satisfactibilidad como se muestra en la tabla correspondiente. El valor MR determina el número de modelos que se pueden recuperar al eliminar la cláusula indicada en la tabla de recuperación de modelos.

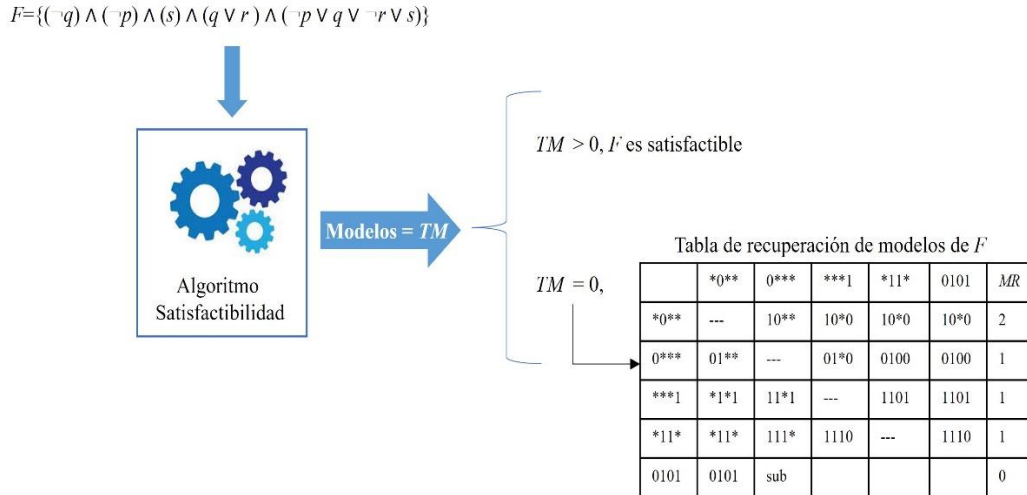


Figura 1.1 Modelo general para generar la propuesta de satisfactibilidad

1.2 Objetivo general y específico

Objetivo general. Desarrollar un sistema computacional que permita generar satisfactibilidad en fórmulas proposicionales que son modeladas en forma normal conjuntiva. El sistema debe determinar el número de modelos de la FNC y si no es satisfactible, genera una tabla que permite decidir que cláusula quitar para que la fórmula sea satisfactible.

Objetivos específicos.

- Diseñar el algoritmo propuesto para la revisión de consistencia de una fórmula proposicional en FNC.
- Diseñar un algoritmo y su implementación computacional para generar una propuesta que permita la satisfactibilidad en fórmulas que no sean satisfactibles.
- Realizar un conjunto de instancias de prueba con diferentes valores establecidos y valores aleatorios generados en el sistema propuesto.
- Implementación del sistema computacional para la revisión de las fórmulas en FNC y la generación de la tabla para determinar que cláusula permite recuperar la satisfactibilidad de la fórmula.

En el siguiente capítulo iniciaremos con el estudio del marco teórico del proyecto de tesis, donde se describe principalmente los fundamentos básicos de la lógica proposicional y de la Forma Normal Conjuntiva.

Capítulo 2.

2. Marco teórico

Actualmente, la lógica se ha convertido en una materia de gran amplitud y aplicación en muchas áreas de la ciencia. Para el desarrollo del trabajo fue necesario revisar estos conceptos teóricos que sirven como base para plantear el algoritmo.

2.1 Simbolización de proposiciones

Una proposición enunciada es una oración, frase o expresión matemática que puede ser falsa o verdadera, pero no ambas a la vez [8].

Para poder simbolizar proposiciones en lógica es necesario que conozcamos las partes que la conforman. Una proposición molecular está formada por una proposición atómica más un término de enlace, por lo menos. Una proposición atómica es aquella que no posee ningún término de enlace [3].

Entre los términos de enlace tenemos «y» y «o» los cuales ligan o trabajan sobre dos proposiciones a la vez, mientras que el término de enlace «no» solo actúa sobre una proposición. Una proposición molecular formada utilizando «y» es una «conjunción», una proposición molecular formada utilizando el término de enlace «o» es una «disyunción» y una proposición molecular formada utilizando el término de enlace «no» es una «negación» [3].

Cuando vamos a trabajar con lógica nos es conveniente trabajar con símbolos tanto para proposiciones y para los términos de enlace. Para proposiciones atómicas se usan letras mayúsculas tales como « P », « Q », « R », « S », y así sucesivamente. Los símbolos utilizados para los términos de enlace son: \wedge para conjunción, \vee para disyunción y \neg para la negación [3].

Cabe resaltar que en lógica la agrupación se expresa mediante paréntesis. La conjunción $(P \vee Q) \wedge R$ es distinta a la disyunción $P \vee (Q \wedge R)$ a pesar de tener las mismas proposiciones atómicas y los mismos términos de enlace [3].

2.2 Literal

En [4] se describe a un Literal como un símbolo de predicado p (literal positivo) o un símbolo de predicado negativo $\neg p$ (literal negativo). En la Figura 2.1 se muestra una fórmula con tres literales:

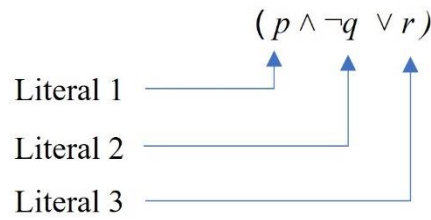


Figura 2.1: Fórmula con 3 literales

Definición 2.1: *Un literal es más bien una variable booleana o bien su negación. Una cláusula es o bien un literal o bien una disyunción de literales [6].*

2.3 Cláusula

Una cláusula es una disyunción de literales, es decir, una fórmula de la forma $l_1 \vee \dots \vee l_n$, donde cada l_j es una literal, o, equivalentemente, una fórmula, $p_1 \vee \dots \vee p_m \vee \neg q_1 \dots \vee \neg q_n$ donde q_i y q_j son símbolos de predicado [4].

En la Figura 2.2 vemos lo que es una cláusula con un ejemplo:

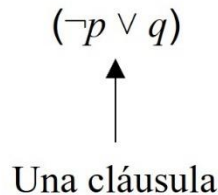


Figura 2.2. Ejemplo de una cláusula con 2 literales

2.4 Conjunto de cláusulas

Una fórmula en CNF es según [4] una conjunción de cláusulas que pueden verse como un conjunto de cláusulas.

En la Figura 2.3 podemos ver un conjunto de cláusulas:

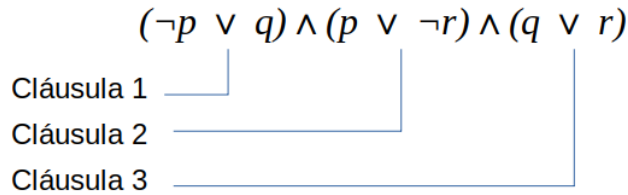


Figura 2.3: Ejemplo de un conjunto de tres cláusulas

2.5 Conjunción

Se inicia con la idea de que cada proposición tiene un valor de certeza ya sea cierta o falsa [3].

Conjunción. «y» como lo indica [3] es un término de enlace de certeza funcional, de manera que se puede decidir el valor de certeza de la proposición $P \wedge Q$ si se conocen los valores de certeza de la proposición P y de la proposición Q . La conjunción de dos proposiciones es cierta si y solo si ambas proposiciones son ciertas. Por tanto, si $P \wedge Q$ ha de ser una proposición cierta, entonces P ha de ser cierta y Q ha de ser cierta. No importa aquí cuáles sean las dos proposiciones que se han unido por medio del término de enlace «y». En lógica se pueden ligar dos proposiciones cualesquiera para formar una conjunción. No se requiere que el contenido de una de ellas tenga relación con la otra.

Para la conjunción tenemos 4 posibles combinaciones de valor de certeza:

P es verdadera y Q es verdadera.

P es verdadera y Q es falsa.

P es falsa y Q es verdadera.

P es falsa y Q es falsa.

Como regla general podemos decir que la conjunción de dos proposiciones es verdadera si y solo si ambas proposiciones son verdaderas.

Mientras que en [5] a la conjunción se define como una fórmula molecular que tiene el conectivo “ \wedge ” como dominante. La regla es que la conjunción es verdadera únicamente cuando sus dos componentes son verdaderos; en los demás casos es falsa.

2.6 Negación

Negación. El término de enlace «no» es de certeza funcional porque la certeza o falsedad de una negación depende enteramente de la certeza o falsedad de la proposición que niega. La regla práctica es: La negación de una proposición cierta es falsa y la negación de una proposición falsa es cierta [3].

Si P es cierta, entonces $\neg P$ es falsa.

Si P es falsa, entonces $\neg P$ es cierta.

2.7 Disyunción

Disyunción. El término de enlace «o» es también un término de enlace de certeza funcional, pero al considerar la certeza o falsedad de cada disyunción se ha de tener en cuenta que se ha utilizado el sentido influyente en la palabra «o». Esto significa que, en cualquier disyunción, por lo menos, una de las dos proposiciones es cierta y quizá ambas. Lo que se requiere es que por lo menos un miembro sea cierto. La regla práctica es: La disyunción de dos proposiciones es cierta si y solo si por lo menos una de las dos proposiciones es cierta [3].

2.8 Forma normal conjuntiva

Una fórmula está en forma normal conjuntiva (CNF, del inglés *Conjunctive Normal Form*) si es una conjunción de disyunciones de literales, es decir, si es de la forma:

$$(l_{1,1} \vee \dots \vee l_{1,k_1}) \wedge \dots \wedge (l_{n,1} \vee \dots \vee l_{n,k_n})$$

donde cada $l_{i,j}$ es una literal [4].

Ejemplo de una fórmula en forma normal conjuntiva con tres literales y tres cláusulas:

$$(\neg p \vee q) \wedge (p \vee \neg r) \wedge (q \vee r)$$

Definición 2.2 Una fórmula booleana está en fórmula normal conjuntiva (CNF) si es una cláusula o una conjunción de cláusulas. Está en la forma k-CNF para algún entero positivo k si está formada por cláusulas, cada una de las cuales contiene como máximo k literales [6].

TRANSFORMAR A FNC (FORMA NORMAL CONJUNTIVA)

Algoritmo: Aplicando a una fórmula F los siguientes pasos se obtiene una forma normal conjuntiva de F :

1. Eliminar los bicondicionales usando la equivalencia

$$A \leftrightarrow B \equiv (A \rightarrow B) \wedge (B \rightarrow A) \quad (1)$$

2. Eliminar los condicionales usando la equivalencia

$$A \rightarrow B \equiv \neg A \vee B \quad (2)$$

3. Interiorizar las negaciones usando las equivalencias

$$\neg(A \wedge B) \equiv \neg A \vee \neg B \quad (3)$$

$$\neg(A \vee B) \equiv \neg A \wedge \neg B \quad (4)$$

$$\neg(\neg A) \equiv A \quad (5)$$

4. Interiorizar las disyunciones usando las equivalencias

$$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C) \quad (6)$$

$$(A \wedge B) \vee C \equiv (A \vee C) \wedge (B \vee C) \quad (7)$$

EJEMPLO DE TRANSFORMACIÓN FNC

Ejemplo de cálculo de una FNC de $\neg(p \wedge (q \rightarrow r))$:

$$\neg(p \wedge (q \rightarrow r))$$

$$\equiv \neg(p \wedge (\neg q \vee r)) \quad [\text{por (2)}]$$

$$\begin{aligned}
 &\equiv \neg p \vee \neg(\neg q \vee r) && [\text{por (3)}] \\
 &\equiv \neg p \vee (\neg\neg q \wedge \neg r) && [\text{por (4)}] \\
 &\equiv \neg p \vee (q \wedge \neg r) && [\text{por (5)}] \\
 &\equiv (\neg p \vee q) \wedge (\neg p \vee \neg r) && [\text{por (6)}]
 \end{aligned}$$

2.9 Forma normal disyuntiva

Una Fórmula está en forma normal disyuntiva (DNF, del inglés *Disjunctive Normal Form*) si es una disyunción de conjunciones de literales, es decir, si es de la forma:

$$(l_{1,1} \wedge \dots \wedge l_{1,k_1}) \vee \dots \vee (l_{n,1} \wedge \dots \wedge l_{n,k_n})$$

donde cada $l_{i,j}$ es una literal [4].

Ejemplo de una fórmula en forma normal disyuntiva con tres literales y tres cláusulas:

$$(\neg p \wedge q) \vee (p \wedge \neg r) \vee (q \wedge r)$$

TRANSFORMAR A FND (FORMA NORMAL DISYUNTIVA)

Algoritmo: Aplicando a una fórmula F los siguientes pasos se obtiene una forma normal disyuntiva de F :

1. Eliminar los bicondicionales usando la equivalencia

$$A \leftrightarrow B \equiv (A \rightarrow B) \wedge (B \rightarrow A) \quad (1)$$

2. Eliminar los condicionales usando la equivalencia

$$A \rightarrow B \equiv \neg A \vee B \quad (2)$$

3. Interiorizar las negaciones usando las equivalencias

$$\neg(A \wedge B) \equiv \neg A \vee \neg B \quad (3)$$

$$\neg(A \vee B) \equiv \neg A \wedge \neg B \quad (4)$$

$$\neg(\neg A) \equiv A \quad (5)$$

4. Interiorizar las disyunciones usando las equivalencias

$$A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C) \quad (6)$$

$$(A \vee B) \wedge C \equiv (A \wedge C) \vee (B \wedge C) \quad (7)$$

EJEMPLO DE TRANSFORMACIÓN FND

Ejemplo de cálculo de una FND de $\neg(p \wedge (q \rightarrow r))$:

$$\begin{aligned} & \neg(p \wedge (q \rightarrow r)) \\ \equiv & \neg(p \wedge (\neg q \vee r)) && [\text{por (2)}] \\ \equiv & \neg p \vee \neg(\neg q \vee r) && [\text{por (3)}] \\ \equiv & \neg p \vee (\neg\neg q \wedge \neg r) && [\text{por (4)}] \\ \equiv & \neg p \vee (q \wedge \neg r) && [\text{por (5)}] \end{aligned}$$

2.10 Consistencia

Supongamos que tenemos tres proposiciones:

1. Trump gano las elecciones del año 2020.
2. Una hormiga puede cargar 0.5 kilogramos.
3. Edgar es más alto que Fátima y Edgar no es más alto que Fátima.

Como podemos ver, cada una de las proposiciones son falsas. Sin embargo, son falsas por distintas razones [3].

La primera proposición es falsa de hecho. Pero en distintas circunstancias pudo ser cierta, y lo mismo pasa con la proposición número 2, pero cuando llegamos a la proposición número tres, esta no puede ser cierta, nunca y en ninguna circunstancia, esta es una proposición lógica imposible. No hace falta el significado de «Edgar es más alto que Fátima» para saber que no puede ser cierta la proposición. Se deduce en su forma lógica [3].

Una proposición de la forma

$$(R) \wedge \neg(R)$$

se denomina una contradicción, Se dice que dos proposiciones son contradictorias si una es la negación de la otra [3].

Como se hace mención en [3] cada dos o más proposiciones que lógicamente no pueden ser ciertas a la vez se dice que son inconsistentes. Se dice que forman un conjunto inconsistente de proposiciones y justas implican una contradicción. En algunos casos lo que interesa no es deducir una conclusión, sino deducir si un conjunto de proposiciones es consistente o inconsistente.

En [5] se dice que las contradicciones, absurdos o proposiciones contradictorias son lo opuesto, o la negación, de las tautologías; por lo tanto: Las contradicciones (o absurdos) son fórmulas moleculares que siempre resultan falsas, cualesquiera que sean los valores veritativos de sus componentes atómicas.

Una forma de demostrar que una fórmula proposicional es una contradicción es haciendo su tabla de verdad.

El ejemplo que se presenta en [5]:

$$“[(p \rightarrow q) \wedge \neg p] \wedge p”.$$

Y su tabla de verdad sería la tabla 2.1 y nos muestra que esa fórmula es una contradicción porque la columna de su conectivo principal tiene solo F:

Tabla 2.1: Tabla de verdad de una contradicción

p	q	$p \rightarrow q$	$\neg p$	$[(p \rightarrow q) \wedge \neg p] \wedge p$
V	V	V	F	F
V	F	F	F	F
F	V	V	V	F
F	F	V	V	F

2.11 Tablas de certeza o tablas de verdad

En [3] se define a una tabla de certeza como un método para analizar los valores de certeza de proposiciones, es el de poner todas las posibilidades de certeza y falsedad en forma de una tabla. Estas tablas básicas de certeza indican rápidamente si una proposición molecular es cierta o falsa si se conoce la certeza o falsedad de las proposiciones que la forman.

Una tabla de verdad está formada por filas y columnas, y el número de filas depende del número de proposiciones diferentes que conforman una proposición compuesta. Asimismo, el número de columnas depende del número de proposiciones que integran la proposición y del número de operadores lógicos contenidos en la misma [8].

En general se tiene la siguiente expresión:

$$\text{Número de filas} = 2^n$$

A continuación, las tablas básicas de certeza:

Negación

P	$\neg P$
V	F
F	V

Conjunción

P	Q	$P \wedge Q$
V	V	V
V	F	F
F	V	F
F	F	F

Disyunción

P	Q	$P \vee Q$
V	V	V
V	F	V
F	V	V
F	F	F

En [5] se dice que las tablas de verdad son unos gráficos que nos muestran visualmente los valores veritativos que va teniendo una proposición molecular de acuerdo con las combinaciones de valores de sus proposiciones atómicas.

Veamos como se construye una tabla de verdad. Tomaremos como base la tabla de verdad 2.2, que corresponde a la fórmula:

$$F = \{(p \vee \neg q \vee r) \wedge (q \vee \neg r) \wedge (p \vee r)\}$$

Tabla 2.2: Tabla de verdad de F

p	q	r	$\neg p$	$\neg q$	$\neg r$	$(p \vee \neg q \vee r)$	$(q \vee \neg r)$	$(p \vee r)$	$(A \wedge B \wedge C)$
V	V	V	F	F	F	V	V	V	V
V	V	F	F	F	V	V	V	V	V
V	F	V	F	V	F	V	F	V	F
V	F	F	F	V	V	V	V	V	V
F	V	V	V	F	F	V	V	V	V
F	V	F	V	F	V	F	V	F	F
F	F	V	V	V	F	V	F	V	F
F	F	F	V	V	V	V	V	F	F

Los pasos a seguir que describe [5] son:

- Primeramente, escribimos la fórmula y a su izquierda las variables (letras, literales) que en ella entran. En este caso son tres (p, q, r). Así tenemos ya el encabezado.
- Para conocer el número de renglones aplicamos la fórmula, 2^n siendo n el número de variables. En este caso $2^n = 2^3$, o sea, $2 \times 2 \times 2 = 8$. Entonces trazamos 8 renglones.
- Debajo de cada una de las tres variables de la izquierda (p, q, r) escribimos una columna de valores. Empezando por la derecha anotamos una V y una F, etc., hasta completar el número de renglones (en este caso, ocho). La siguiente columna a la izquierda se forma escribiendo dos veces V y dos veces F, etc., hasta llenar los renglones. La siguiente columna se forma escribiendo cuatro veces V y cuatro veces F, etcétera. Así mismo se hace con las columnas de p, q y r negadas ($\neg p, \neg q, \neg r$), pero en lugar de V ponemos F y en el lugar de F ponemos V.
- Para calcular los valores de los conectivos se aplica la regla respectiva y se inicia por los más interiores. El último conectivo en ser calculado es aquel fuera de todo paréntesis. En este caso añadimos tres columnas, una por cada cláusula que tenemos, entonces, añadimos otras tres columnas y calculamos los valores, en la última columna va el valor resultante del cálculo de los valores de las tres cláusulas.

2.12 Introducción a la NP-completitud

En la vida existen muchos problemas prácticos para los cuales no se conoce ningún algoritmo eficiente, pero cuya dificultad intrínseca no ha conseguido demostrar nadie. Entre estos se cuentan problemas tan conocidos como el del viajante del comercio, el coloreo óptimo de grafos, la búsqueda del camino simple más largo de un grafo, y la satisfacción de una fórmula booleana [6].

2.12.1 Las clases P y NP

En computación, cuando el tiempo de ejecución de un algoritmo (mediante el cual se obtiene una solución al problema) es menor que un cierto valor calculado a partir del número de variables implicadas (generalmente variables de entrada) usando una fórmula polinómica, se dice que dicho problema se puede resolver en un tiempo polinómico P [7, 13].

Por ejemplo, si determinar el camino óptimo que debe recorrer un cartero que pasa por N casas necesita menos de $50N^2 + N$ segundos, entonces el problema es resoluble en un "tiempo polinómico" [7, 13].

De esa manera, tiempos $2n^2 + 5n$, $4n^6 - 7n^2$ o $n^{2^{127}}$ son polinómicos, pero 2^n no lo es [7, 13].

Dentro de los tiempos polinómicos, podemos distinguir los logarítmicos $O(\log n)$, los lineales $O(n)$, los cuadráticos $O(n^2)$, los cúbicos $O(n^3)$, etc. [7, 13].

Un algoritmo es eficiente si existe algún polinomio $p(n)$ tal que el algoritmo pueda resolver cualquier caso del tamaño n en un tiempo que está en $O(p(n))$ diremos que estos algoritmos son de tiempo polinómico [6].

Definición 2.3 P es la clase de problemas de decisión que se pueden resolver mediante un algoritmo de tiempo polinómico [6].

Llamamos P al conjunto de problemas en los que podemos encontrar una respuesta al problema en un tiempo razonable.

2.12.2 Problemas NP-completos

En teoría de complejidad llamamos NP al conjunto de problemas en los que podemos comprobar en un tiempo razonable (polinomial) si una respuesta al problema es correcta o no.

Definición 2.4 de [6] dice que NP es la clase de problemas de decisión que admiten un sistema de pruebas $F \subseteq XxQ$ tal que existe un polinomio $p(n)$ y un algoritmo de tiempo polinómico A tal que:

- *Para todo $x \in X$ exista un $q \in Q$ tal que $(x, q) \in F$ y además del tamaño de q es como máximo $p(n)$, donde n es el tamaño de x .*
- *Para todos los pares (x, q) , el algoritmo A puede verificar si (x, q) pertenece o no a F.*

Para este tema existen diferentes definiciones, pero todas tienen la misma idea, al menos en [4] se dice que un problema está en NP si hay algún algoritmo *NO-determinista Polinómico* que lo resuelve. Informalmente, esto significa que en tiempo polinómico podemos “adivinar” una posible solución y comprobar si efectivamente lo es.

Un problema que está en NP se dice que es NP-completo si además es posible utilizarlo para expresar en tiempo polinómico cualquier otro problema de NP [4].

2.13 Satisfactibilidad

En [4] se dice que una fórmula F es satisfactible si tiene algún modelo, es decir, si existe alguna interpretación I tal que I sea consecuencia lógica de F . Una fórmula F es insatisfactible (o es una contradicción) si F no es satisfactible. En otras palabras, mientras tengamos un modelo que sea verdadero, entonces F es satisfactible.

Definición 2.5 Una Fórmula es satisfactoria si existe al menos una forma de asignar valores a sus variables de tal modo que resulte ser verdadero. Denotaremos mediante SAT el problema de decidir, dada una fórmula booleana, si es o no satisfactoria [6].

Por ejemplo, $(p \vee q) \Rightarrow (p \wedge q)$ es satisfactoria porque da lugar a ser verdadero si asignamos el valor verdadero tanto a p como a q . Esta fórmula es viable a pesar de que existen otras asignaciones de valor para las variables que la hacen falsa, como $p = \text{verdadero}$ y $q = \text{falso}$. Por otra parte, $(\neg p) \wedge (p \vee q) \wedge (\neg q)$ no es viable porque sigue siendo falsa independientemente de los valores que asignemos a p y a q [6].

En principio, es posible decidir si una fórmula booleana es satisfactoria calculando su valor para todas las posibles asignaciones de sus variables booleanas. Sin embargo, esto no es práctico cuando el número n de variables booleanas implicadas es elevado, porque hay 2^n asignaciones posibles. Por tanto $SAT \in NP$ [6].

Definición 2.6 SAT-CNF es la restricción del problema SAT a fórmulas booleanas en CNF. Para todo k positivo, SAT- k -CNF es la restricción de SAT-CNF a fórmulas booleanas en k -CNF [6].

Capítulo 3

3. Trabajo relacionado

En este capítulo se presentan los trabajo relacionado con la generación de satisfactibilidad en fórmulas en forma normal conjuntiva.

3.1 El procedimiento DPLL

DPLL (Davis - Putnam - Logemann - Loveland) es un algoritmo de búsqueda exhaustivo, basado en retroceso, utilizado para decidir la satisfacción booleana de las fórmulas lógicas proposicionales en forma normal conjuntiva (CNF). Fue introducido en 1962 por Martin Davis, Hilary Putnam, George Logemann y Donald W. Loveland. El DPLL es un procedimiento muy eficiente, y después de más de 40 años todavía forma la base de los solucionadores SAT completos más eficientes [9].

El algoritmo de retroceso de la base se ejecuta eligiendo un literal, asignando un valor de verdad (verdadero o falso), simplificando la fórmula y luego verificando recursivamente si la fórmula simplificada es satisfactoria; si este es el caso, la fórmula original también es satisfactoria; de lo contrario, el mismo procedimiento recursivo, tomando el otro valor de verdad (falso o verdadero). Este procedimiento se conoce como la regla de división, ya que divide el problema en dos problemas secundarios más simples. El paso de simplificación esencialmente elimina todas las cláusulas que se han vuelto verdaderas en esa asignación parcial de la fórmula, y elimina de las cláusulas restantes todos los literales que se han vuelto falsos. El algoritmo DPLL mejora el retroceso con el uso forzado de estas reglas, en cada paso: la insatisfacción de una asignación parcial dada se verifica si una cláusula se vuelve vacía, es decir, si todas sus variables se han asignado de tal manera que los literales correspondientes sean falsos. La satisfacción de la fórmula se comprueba cuando todas las variables se asignan sin generar cláusulas vacías o, en implementaciones modernas, si se cumplen todas las cláusulas. La insatisfacción de la fórmula completa solo se puede verificar después de una investigación exhaustiva del problema [9].

3.2 WolframAlpha

Una de las herramientas más completas y que nos podría proporcionar información sobre si una fórmula es satisfacible o insatisfacible es la aplicación web llamada WolframAlpha, pero esta aplicación nos proporciona los modelos que tiene una fórmula mediante la tabla de verdad lo cual vuelve al proceso muy tedioso cuando tenemos muchas variables, ya que genera una tabla de verdad inmensa, además de que las fórmulas en forma normal conjuntiva con más de 10 literales ya no las puede trabajar, como lo podemos observar en la Figura 3.1 tenemos una fórmula en forma normal conjuntiva con 10 literales y ya no muestra la tabla de verdad:



Figura 3.1: WolframAlpha no genera tabla de verdad con muchas variables

Mientras que si quitamos una de las variables ya nos genera una tabla de verdad, pero es lo único que nos proporcionará y como se mencionó esta es muy grande, como se puede ver en la Figura 3.2:



Figura 3.2: WolframAlpha genera tabla de verdad con pocas variables

Entonces cuando trabajamos con fórmulas con muchas literales y queremos saber si una fórmula en forma normal conjuntiva es satisfactible, uno de los métodos es mediante la tabla de verdad, pero hemos visto en el ejemplo anterior no siempre es la mejor opción, ya que dependiendo el número de variables va a ser la cantidad de renglones, de acuerdo a:

$$2^n$$

Donde n es el número de literales que tiene la fórmula, entonces para el ejemplo de la Figura 3.1 que tiene una fórmula en FNC con 10 variables, nos generaría una tabla de verdad de $2^n = 2^{10} = 1024$ renglones, la cual no nos genera incluso si le ponemos el comando “truth table” para obligar a WolframAlpha a imprimir la tabla de verdad.

Cuando le quitamos una literal como en se puede ver en la Figura 3.2 ya nos genera la tabla de verdad con $2^n = 2^9 = 512$ renglones, que aun así para revisar el número de modelos manualmente ya no es viable.

Entonces a esto nos limitan estos sistemas que pueden generar tablas de verdad.

Mi sistema aparte de decir si la fórmula en FNC es satisfactible me dirá el número de modelos que tiene, y en caso de que sea insatisfactible generara una tabla de recuperación de modelos la cual me proporcionará información sobre que cláusula hace a la fórmula insatisfactible para eliminarla y así la fórmula pase a ser satisfactible.

Capítulo 4

4. Algoritmo para generar satisfactibilidad en fórmulas proposicionales

En este capítulo se presenta el diseño del algoritmo para generar satisfactibilidad en fórmulas proposicionales. Se describe cada una de las fases de la propuesta algorítmica.

4.1 Transformación a 1,0 y *

Como ya se ha mencionado, el problema de satisfactibilidad (SAT) en general es un problema de decisión en el cual, dada una fórmula booleana F en FNC, formada de un conjunto de cláusulas con n literales, se debe encontrar una asignación a dichas variables tal que exista al menos una asignación de valores que haga verdadera la fórmula F .

Para poder tratar una fórmula en Forma normal conjuntiva de manera que computacionalmente sea óptimo, se hace una transformación de las cláusulas a cadenas de ceros, unos y asteriscos (*), de forma que consideraremos para cada cláusula el mismo conjunto de n variables. En la Figura 4.1 tenemos un ejemplo de transformación:

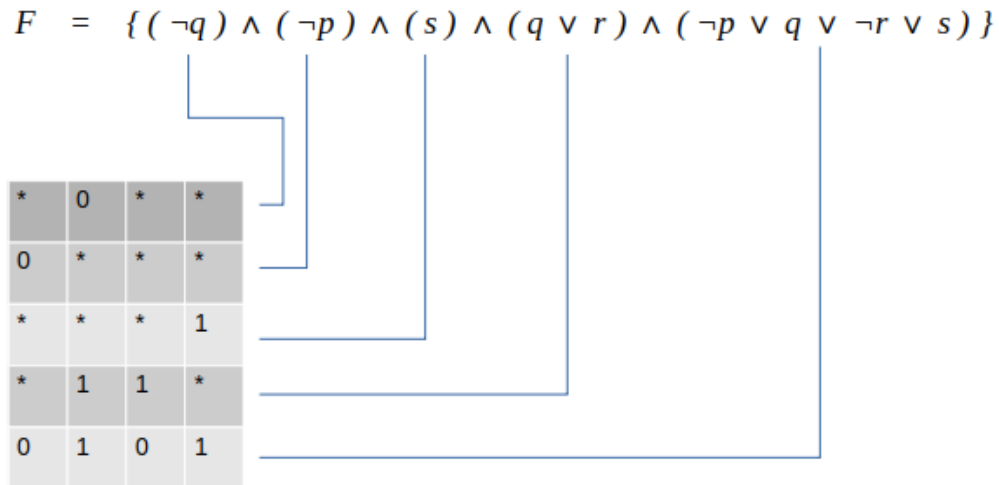


Figura 4.1: Ejemplo de transformación a 1, 0 y *

Como podemos observar de la Figura 4.1, cuando una literal no está en la cláusula esta posición pasa a ser ocupada por un asterisco (*), mientras que las literales negadas pasan a ser representadas por un cero (0) y las literales no negadas son representadas por un uno (1).

4.2 Revisión de la consistencia

Ahora que la fórmula fue transformada para un mejor manejo computacionalmente hablando, debemos considerar dentro del proceso general de la propuesta algorítmica para la revisión de la consistencia de una fórmula en FNC tres operaciones básicas:

4.2.1 Independencia

En [5] se dice que la relación de independencia se da cuando los valores de verdad de una proposición no influyen sobre los valores de verdad de la otra. Y se define como:

Definición 4.1 Dos proposiciones son independientes cuando la segunda es verdadera, falsa o indeterminada, sin importar que la primera sea verdadera o falsa.

Ejemplo de cómo se procede con el caso de subsumida:

		*0**	01**	11*1	1110
0101		0101	sub		

4.2.3 Generación de nuevas cláusulas

Generación de nuevas cláusulas: dadas K y C dos cláusulas se generan cláusulas si $Ci = *$ y Ki es una literal con valor en $\{0,1\}$.

Ejemplo del patrón básico de la operación de generación:

k	0	1	*	0	*
c	0	*	*	*	0

La operación de generación de nuevas cláusulas puede generar de 1 hasta n_c cláusulas nuevas, donde n_c depende del número de n variables, por lo que el máximo para n_c es $n-1$.

Ejemplo de cómo se genera una nueva cláusula:

		*0**
0***		01**

	K
C	Nueva cláusula

Como se puede notar en el ejemplo anterior, al generar una nueva cláusula siempre se busca la independencia de la nueva cláusula con k y las demás nuevas cláusulas en caso de que se genere más de 1.

Ejemplo de cómo se generan nuevas cláusulas:

	*****000
000*****	000***1**
	000***01*
	000***001

Después de que transformamos una cláusula $C \in F$ basándose en el mismo conjunto de n variables, tenemos que el número de modelos (Mod) satisfactibles está dado por $Mod(F) = 2^n - 2^{Ast(C)}$, donde $Ast(C)$ es el número de asteriscos de la cláusula C . Por ejemplo, sea $F = \{(1***0)\}$ con $n = 5$ variables, donde F tiene solo una cláusula $C = (1***0)$, por lo tanto, $Ast(C) = 3$, entonces $Mod(F) = 2^5 - 2^3 = 32 - 8 = 24$ modelos.

En el siguiente ejemplo se muestra una fórmula en FNC y como se calcula el número de modelos que tiene:

$$F = \{(*0**) (0***) (***) (11*) (0101)\}$$

$$\text{Total de modelos} = 2^4 = 16$$

$$F' = \{(*0**) \} = 2^3 = 8$$

$$\text{Total de modelos} = 16 - 8 = 8$$

	*0**
0***	01**

$$\text{Total de modelos} = 8 - 2^2 = 8 - 4 = 4$$

	*0**	01**
***1	*1*1	11*1

$$\text{Total de modelos} = 4 - 2^1 = 4 - 2 = 2$$

	*0**	01**	11*1
11	*11*	111*	1110

Total de modelos = $2 - 2^0 = 2 - 1 = 1$

	*0**	01**	11*1	1110
0101	0101	sub		

Por tanto, tiene un modelo y es satisfactible.

4.3 Generar consistencia

Cuando ya aplicamos las operaciones de independencia, subsumida y generación a una fórmula en forma normal conjuntiva y esta nos da como resultado cero modelos, entonces, es necesario aplicar un proceso de recuperación de modelos.

En este caso será necesario eliminar alguna cláusula con el fin de mantener la consistencia de la fórmula y de la base de conocimiento, otra cosa importante, es que la pérdida de información debe ser mínima con el objetivo de no eliminar creencias innecesarias.

La estrategia propuesta consiste en aplicar el proceso central de revisión con las operaciones *Ind()*, *Gen()* y *Sub()*.

En el siguiente ejemplo tenemos una fórmula en FNC, pero no es satisfactible, entonces aplicamos una tabla de recuperación de modelos para poder saber qué cláusula es la que hace que F sea insatisfacible y poder quitarla para que F sea satisfactible:

$$F = \{(*0**) (0***) (***1) (*1**) (**01)\}$$

$$\text{Total de modelos} = 2^4 = 16$$

$$F' = \{(*0**) \} = 2^3 = 8$$

Total de modelos = $16 - 8 = 8$

	*0**
0***	01**

Total de modelos = $8 - 2^2 = 8 - 4 = 4$

	*0**	01**
***1	*1*1	11*1

Total de modelos = $4 - 2^1 = 4 - 2 = 2$

	*0**	01**	11*1
*1**	*1**	11**	11*0

Total de modelos = $2 - 2^1 = 2 - 2 = 0$

Como podemos ver ya no tenemos modelos, por tanto, F es insatisfacible, continuando:

	*0**	01**	11*1	11*0
**01	*101	1101	subsumida	

Entonces para hacer satisfactible F recurrimos a la tabla de modelos a recuperar:

Tabla 4.1: Tabla de recuperación de modelos de una fórmula en FNC

F	*0**	0***	***1	*1**	**01	Modelos a recuperar
*0**	-----	10**	10*0	10*0	10*0	$2^1 = 2$
0***	01**	-----	01*0	sub		0
***1	*1*1	11*1	-----	sub		0
*1**	*1**	11**	11*0	-----	11*0	$2^1 = 2$
*1**	*101	1101	sub		-----	0

Entonces podemos observar que la cláusula (*0**) si la eliminamos nos recupera dos modelos y de igual manera la cláusula (*1**), entonces eliminamos la primera cláusula y comprobamos:

$$F \text{ original} = \{(*0**) (0***) (**1) (*1**) (**01)\}$$

$$F \text{ para recuperar modelos} = \{(0***) (**1) (*1**) (**01)\}$$

$$\text{Total de modelos} = 2^4 = 16$$

$$F = \{(0***)\} = 2^3 = 8$$

$$\text{Total de modelos} = 16 - 8 = 8$$

	0***
***1	1**1

$$\text{Total de modelos} = 8 - 2^2 = 8 - 4 = 4$$

	0***	1**1
*1**	11**	11*0

$$\text{Total de modelos} = 4 - 2^1 = 4 - 2 = 2$$

	0***	1**1	11*0
**01	1*01	subsumida	

Total de modelos = 2

Entonces como podemos ver ya recuperamos 2 modelos y ahora F ya es satisfactible.

4.4 algoritmo

Entrada:

Sea $F = \{C1, C2, \dots, Cm\}$ una base de conocimiento en FNC, con m cláusulas y n variables en cada cláusula.

Salida:

S : conjunto de cláusulas equivalentes resultantes del conteo

TM : total de modelos de la base de conocimiento

Funciones:

$Ind(C, K, n)$: se encarga de verificar si las cláusulas son independientes y si lo son devuelve True.

$Sub(C, K, n)$: se encarga de verificar si la cláusula C sea subsumida con respecto a K y en caso de que lo sea no devuelve nada, ya que, pues, se eliminaría esta cláusula.

$Gen(C, K, n)$: esta función se encarga de generar nuevas cláusulas y devuelve un arreglo con las nuevas cláusulas.

$Ope(C, K, n)$: esta función se encarga de decidir cuál es la operación a realizar entre la cláusula C y la cláusula K .

$Move(C, K, n, number)$: la función Move se encarga de moverse entre las cláusulas en la posición K . La variable *number* es de tipo int y va a contener el

número de cláusulas que existen en la posición K , esta se va a ir actualizando con cada iteración de la posición C .

En esta función se hace llamado recursivo a sí misma para cada cláusula en K .

Ejemplo:

		0***	1**1	11*0	Posición K , $number = 3$
Posición C	**01	1*01	sub		

Table (F , m , n): Esta genera la tabla de recuperación de modelos y lo único que regresa es un arreglo de enteros que sería el número de modelos a recuperar por cada cláusula, cada posición de este arreglo corresponde con el orden de cada cláusula.

Inicio:

1: $TM = 2^n - 2^{Ast(C)}$ // $Ast(C)$ = número de asteriscos de una cláusula

2: $K = C1$; //Consideramos $C1$ la cláusula con más asteriscos

3: **Para cada** Cláusula f_i **en** F , $i=2$, **hasta** m **Hacer**

4: $F = \text{Move}(\text{Cláusula}[i], K, n, number)$

5: $k = k + F$ //concatena las cláusulas para la posición K

6: $number = \text{TamañoDe}(k)$

7: **FinPara**

8: CalculaTM

9: **Si** TM es cero

10: $\text{ModRecuperar} = \text{Table}(F, m, n)$

11: Se elije la cláusula a eliminar de acuerdo a ModRecuperar

12: Se elimina la cláusula de F

13: **Para cada** Cláusula f_i **en** F , $i=2$, **hasta** m **Hacer**

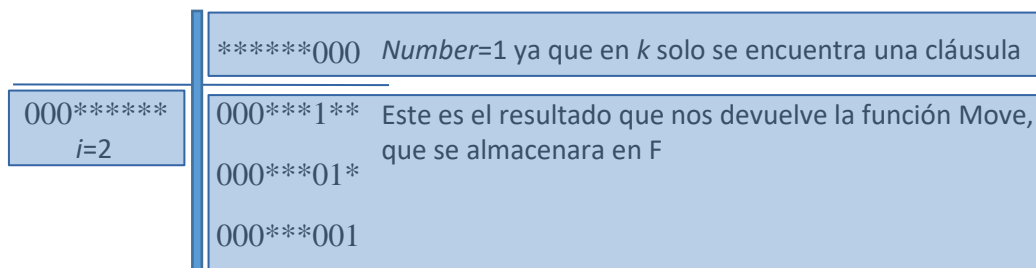
14: $F = \text{Move}(\text{Cláusula}[i], K, n, number)$

15: $k = k + F$
 16: $number = \text{TamañoDe}(k)$
 17: **FinPara**
 18: **FinSi**
 19: CalculaTM

Para ejemplificar el proceso del algoritmo lo veremos con un ejemplo de una fórmula en FNC que en este caso si es satisfactible.

$F = \{(\text{*****}000) (000\text{*****}) (\text{***}000\text{**})\}$

Entonces, como dice el algoritmo, asignamos la primera cláusula a k e inicia el **Para**, pero ya considerando desde la segunda cláusula, así que para el **primer ciclo** se llama a la función Move a la cual se le envía la segunda cláusula y la primera cláusula que está en k .

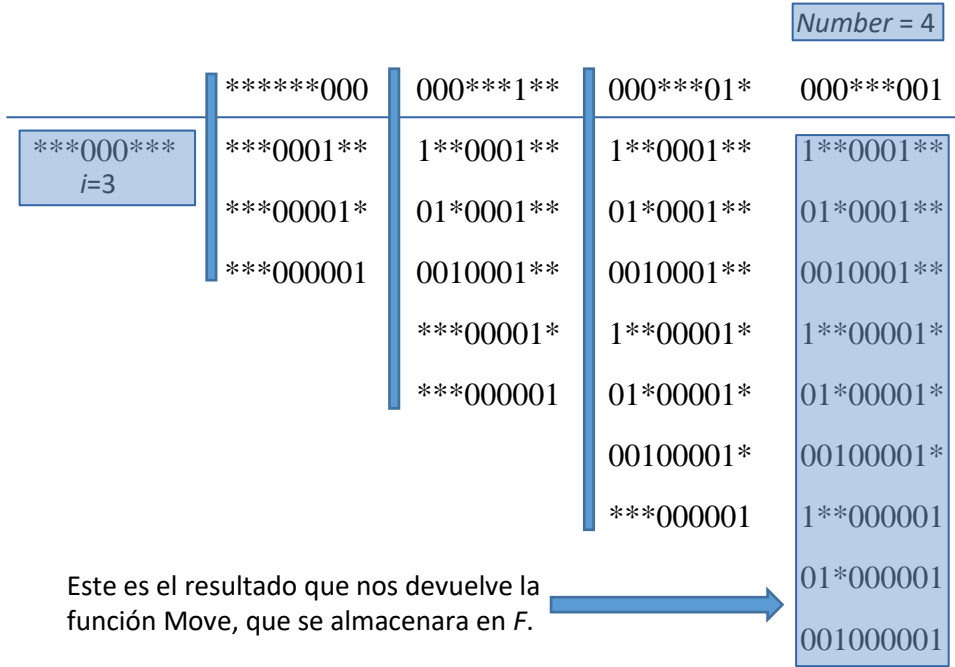


Después de este proceso se va a concatenar las cláusulas resultantes con k dando como resultado una k con las cláusulas:

$k = \{ (\text{*****}000) (000\text{***}1**) (000\text{***}01*) (000\text{***}001) \}$

También dentro de este primer ciclo actualizamos number con el tamaño de k que es igual a 4 cláusulas.

Cuando pasamos al **segundo ciclo de Para** con $i = 3$, llamamos a la función Move y le mandamos la cláusula número 3, con k y el *number* actualizado y esta función sigue el siguiente proceso y nos devuelve la última fila de cláusulas:



Concatenamos F con K y como fórmula final tendríamos:

$$F = \{ (*****000) (000***1**) (000***01*) (000***001) (1**0001**) (01*0001**) (0010001**) (1**00001*) (01*00001*) (00100001*) (1**000001) (01*000001) (001000001) \}$$

Con esta F resultante es con la que calculamos TM de acuerdo con el número de literales 2^9 menos la sumatoria de $2^{\#DeEnLaCláusula}$:

$$TM = 2^9 - (2^6 + 2^5 + 2^4 + 2^3 + 2^4 + 2^3 + 2^2 + 2^3 + 2^2 + 2^1 + 2^2 + 2^1 + 2^0)$$

$$TM = 512 - (169)$$

$$TM = 343 \text{ modelos}$$

Capítulo 5

5. Sistema para generar satisfactibilidad en fórmulas proposicionales

El sistema desarrollado es una aplicación que se ejecuta desde terminal implementando el algoritmo descrito en el capítulo 4 de este trabajo de tesis. Se utilizó el lenguaje de programación Python en su versión 3.8.10.

Python es un lenguaje de programación de alto nivel que se utiliza para desarrollar aplicaciones de todo tipo. A diferencia de otros lenguajes como Java o .NET, se trata de un lenguaje interpretado, es decir, que no es necesario compilarlo para ejecutar las aplicaciones escritas en Python, sino que se ejecutan directamente por el ordenador utilizando un programa denominado interpretador, por lo que no es necesario “traducirlo” a lenguaje máquina [10].

Gracias a que Python es un lenguaje de programación multiplataforma, algo que permite desarrollar aplicaciones en cualquier sistema operativo con una facilidad asombrosa [10], fue posible crear el sistema para generar satisfactibilidad en fórmulas en FNC en el sistema operativo Ubuntu, el cual es un Linux.

Para la creación del sistema que genera satisfactibilidad se ocupó la librería NumPy, para poder manejar eficientemente toda la cantidad de datos que se presentan.

NumPy es una librería de Python para computación científica. NumPy significa Python numérico [11]. Aquí está la descripción oficial de la librería indicada en su página web:

"NumPy es el paquete fundamental para la computación científica con Python. Contiene entre otras cosas:

- *un poderoso objeto de arreglo N-dimensional*
- *funciones sofisticadas*
- *herramientas para integrar código en C/C++ y Fortran*
- *útiles capacidades de álgebra lineal, transformación de Fourier y números aleatorios*

Además de sus obvios usos científicos, NumPy puede ser utilizado como un eficiente contenedor multidimensional de datos genéricos. Tipos de datos arbitrarios puede ser definidos. Esto permite que NumPy se integre sin problemas y con rapidez con una amplia variedad de bases de datos.

NumPy está licenciado bajo el formato BSD, lo que permite la reutilización con pocas restricciones".

NumPy es una librería de Python tan importante que hay otras librerías (incluyendo pandas) que están construidas enteramente sobre NumPy [11].

5.1 Formato de la fórmula para la lectura por el sistema

Las fórmulas en forma normal conjuntiva se van a almacenar en un archivo de texto, de forma que en cada renglón de este archivo va a ir una cláusula, cabe resaltar que estas deben de estar ya transformadas a *, 1 y 0, ya que este es el formato de entrada del sistema.

Por ejemplo, vamos a utilizar la fórmula F que se utilizó al final del capítulo 4 para ejemplificar el proceso del algoritmo:

$F = \{ (*****000) (000*****) (***000***) \}$

Esta fórmula al estar en el archivo .txt debe de estar sin los paréntesis y en cada renglón una cláusula, por tanto, el archivo de texto para esta fórmula solo tendrá 3 renglones, esto lo podemos ver en la Figura 5.1.

```
*****000
000*****
***000***
```

Figura 5.1: formato de una fórmula en FNC dentro de un archivo .txt

5.2 Ejecución del sistema

Paso 1:

Para poder ejecutar el sistema primero tenemos que editar una de las líneas en el cual va a ir el nombre del archivo .txt, como se puede apreciar en la Figura 5.2:

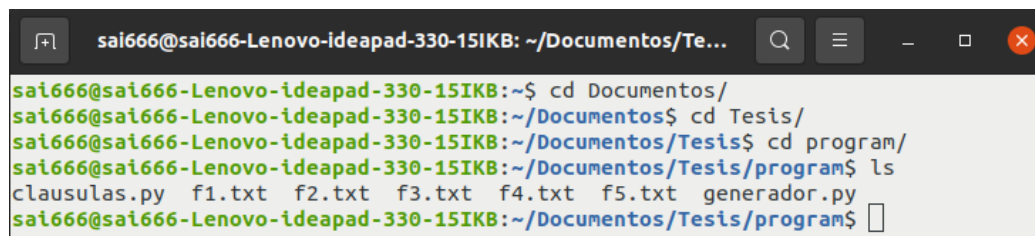
Esto es para que el sistema sepa con qué archivo de texto va a trabajar, cada archivo de texto solo va a contener una fórmula.

```
186 #-----  
187 file=open('f4.txt','r')
```

Figura 5.2: Línea de código para indicar el archivo a leer

Paso 2:

Ahora que ya tenemos un archivo de texto con una fórmula en forma normal conjuntiva en el formato correcto de lectura, lo siguiente es abrir una terminal de Ubuntu y nos dirigimos a la carpeta donde tenemos almacenado el sistema y los archivos .txt de las fórmulas, en la Figura 5.3 podemos ver cómo nos movemos:



```
sai666@sai666-Lenovo-ideapad-330-15IKB: ~/Documentos/Te...  
sai666@sai666-Lenovo-ideapad-330-15IKB:~$ cd Documentos/  
sai666@sai666-Lenovo-ideapad-330-15IKB:~/Documentos$ cd Tesis/  
sai666@sai666-Lenovo-ideapad-330-15IKB:~/Documentos/Tesis$ cd program/  
sai666@sai666-Lenovo-ideapad-330-15IKB:~/Documentos/Tesis/program$ ls  
clausulas.py f1.txt f2.txt f3.txt f4.txt f5.txt generador.py  
sai666@sai666-Lenovo-ideapad-330-15IKB:~/Documentos/Tesis/program$
```

Figura 5.3: comandos cd para moverse entre carpetas y ls para ver el contenido

Para ejecutar el programa utilizamos el comando:

python3 <nombre.py>

```

sai666@sai666-Lenovo-ideapad-330-15IK8:~/Documentos/Tests/program$ python3 clausulas.py
Original F
[['*', '*', '*', '*', '*', '*', 0, 0, 0], [0, 0, 0, '*', '*', '*', '*', '*'], ['*', '*', '*', 0, 0, 0, '*', '*', '*']]

-----

We apply the operations
working on the clause number: 1
working on the clause number: 2

-----

The new F is:
clause 0 : ['*', '*', '*', '*', '*', '*', 0, 0, 0]
clause 1 : [0, 0, 0, '*', '*', '*', 1, '*', '*']
clause 2 : [0, 0, 0, '*', '*', '*', 0, 1, '*']
clause 3 : [0, 0, 0, '*', '*', '*', 0, 0, 1]
clause 4 : [1, '*', '*', 0, 0, 0, 1, '*', '*']
clause 5 : [0, 1, '*', 0, 0, 0, 1, '*', '*']
clause 6 : [0, 0, 1, 0, 0, 0, 1, '*', '*']
clause 7 : [1, '*', '*', 0, 0, 0, 0, 1, '*']
clause 8 : [0, 1, '*', 0, 0, 0, 0, 1, '*']
clause 9 : [0, 0, 1, 0, 0, 0, 0, 1, '*']
clause 10 : [1, '*', '*', 0, 0, 0, 0, 0, 1]
clause 11 : [0, 1, '*', 0, 0, 0, 0, 0, 1]
clause 12 : [0, 0, 1, 0, 0, 0, 0, 0, 1]
Total of models in F is
343
-----
F is satisfiable

```

Figura 5.4: Resultados del sistema sobre una fórmula satisfactible

Como podemos ver en la Figura 5.4 la fórmula tiene un total de 343 modelos, por tanto, ya es satisfactible y no es necesario hacer algo más, pero para el caso de que no sea satisfactible tendríamos que crear una tabla de recuperación de modelos, en el siguiente ejemplo se probara con una fórmula que no sea satisfactible para que el sistema la evalúe y genere satisfactibilidad, la fórmula a utilizar es:

$$F = \{(*0**)(0***)(***1)(*1**)(**01)\}$$

Entonces generamos manualmente un archivo de texto con esta fórmula en el formato correspondiente, como se aprecia en la Figura 5.5:

```

1 *0**
2 0***
3 ***1
4 *1**
5 **01

```

Figura 5.5: Resultados del sistema sobre una fórmula satisfactible

Editamos la línea de código para que lea el archivo .txt en donde se encuentra la fórmula y después ejecutamos en una terminal de Ubuntu el sistema:

```

sat666@sat666-Lenovo-Ideapad-330-15IKB:~/Documentos/Tests/program$ python3 clausulas.py
Original F
[['*', 0, '*', '*'], [0, '*', '*', '*'], ['*', '*', '*', 1], ['*', 1, '*', '*'], ['*', '*', 0, 1]]

-----

We apply the operations
working on the clause number: 1
working on the clause number: 2
working on the clause number: 3
working on the clause number: 4

-----

The new F is:
clause 0 : ['*', 0, '*', '*']
clause 1 : [0, 1, '*', '*']
clause 2 : [1, 1, '*', 1]
clause 3 : [1, 1, '*', 0]
Total of models in F is
0

```

Figura 5.6: Resultados del sistema sobre una fórmula insatisfactible.

Y como podemos ver en la Figura 5.6, cuando se termina de ejecutar el sistema que trabaja con las operaciones de independencia, subsumida y generación sobre la fórmula, se tiene como resultado que esta fórmula tiene cero modelos, en otras palabras esta fórmula es insatisfactible, por tanto, el sistema recurre a aplicar una tabla de recuperación de modelos en la cual nos devuelve el número de modelos que nos recuperaría cada cláusula si es eliminada, los resultados del sistema se pueden apreciar en la Figura 5.7:

```

-----

F is not satisfiable so we apply models recuperation table
K is [['*', 0, '*', '*'], [0, '*', '*', '*'], ['*', '*', '*', 1], ['*', 1, '*', '*'], ['*', '*', 0, 1]]

C is ['*', 0, '*', '*'] and the result is: [1, 0, '*', 0]
C is [0, '*', '*', '*'] and the result is: None
C is ['*', '*', '*', 1] and the result is: None
C is ['*', 1, '*', '*'] and the result is: [1, 1, '*', 0]
C is ['*', '*', 0, 1] and the result is: None

and the models to get back of each clause are:
[2, 0, 0, 2, 0]

```

Figura 5.7: Resultados de la tabla de modelos a recuperar.

Ahora que ya sabemos los modelos que recuperaríamos de cada posible cláusula a eliminar, el sistema procede a eliminar la cláusula que recupera más modelos, en este caso eliminará la cláusula en la primera posición, ya que es una de las que recupera más modelos si es eliminada, esta información la imprime el sistema y se puede apreciar en la Figura 5.8:

```

-----
I am going to delete the clause with more models to get back
clause to delete is:
['*', 0, '*', '*']
in the position:
0
So, after delete the clause the result is:
[[0, '*', '*', '*'], ['*', '*', '*', 1], ['*', 1, '*', '*'], ['*', '*', 0, 1]]

```

Figura 5.8: Cláusula a eliminar.

Después de eliminar la cláusula seleccionada, el sistema vuelve a aplicar las operaciones de independencia, subsumida y generación sobre la nueva fórmula y manda a imprimir el resultado, este resultado se puede apreciar en la Figura 5.9:

```

-----
We apply the operations again with the new F
F' is :
[[0, '*', '*', '*'], [1, '*', '*', 1]]
F' is :
[[0, '*', '*', '*'], [1, '*', '*', 1], [1, 1, '*', 0]]
F' is :
[[0, '*', '*', '*'], [1, '*', '*', 1], [1, 1, '*', 0]]
The new F is:
[[0, '*', '*', '*'], [1, '*', '*', 1], [1, 1, '*', 0]]
Total of Models in F is
2
Now F is satisfiable

```

Figura 5.9: Resultados al generar satisfactibilidad en una fórmula no satisfactible.

Entonces ahora la fórmula ya es satisfactible, en otras palabras, el sistema ha generado satisfactibilidad en una fórmula no satisfactible.

5.3 Generador de cláusulas aleatorias

Para poder llevar al límite este sistema que evalúa si una fórmula en forma normal conjuntiva es satisfactible y si no lo es genera satisfactibilidad sobre la fórmula, se creó un sistema de generación de cláusulas, el cual genera cláusulas de manera aleatoria.

Para hacer este sistema se tienen que tomar en cuenta dos situaciones, la primera situación es que una cláusula sea de puros asteriscos (*):

$$c1 = (*, *, *, *, *, *, *, *, *, *, *, *, *, *, *)$$

Esta situación se tiene que evitar, ya que de esta manera nos quitaría todos los modelos dando como resultado puras fórmulas no satisfactibles desde el inicio, la forma de evitar esto es que cuando vallamos generando en las posiciones donde $i = j$ se obligue a generar ya sea un 1 o un 0:

$$c1 = (1, *, *, *, *, *, *, *, *, *, *, *, *, *, *)$$

$$c2 = (*, 0, *, *, *, *, *, *, *, *, *, *, *, *, *)$$

$$c3 = (*, *, 1, *, *, *, *, *, *, *, *, *, *, *, *)$$

$$c4 = (*, *, *, 1, *, *, *, *, *, *, *, *, *, *, *)$$

$$c5 = (*, *, *, *, 1, *, *, *, *, *, *, *, *, *, *)$$

$$c6 = (*, *, *, *, *, 0, *, *, *, *, *, *, *, *, *)$$

Así evitamos que una cláusula esté formada por puros asteriscos (*) de manera horizontal.

La otra situación que deberíamos evitar es cuando se generan puros (*) pero de manera vertical, por ejemplo:

$$\begin{array}{l} c1 = (1, *, *, *, *, *, *, *, *, *, *, *, *, *, *) \\ c2 = (*, 0, *, *, *, *, *, *, *, *, *, *, *, *, *) \\ c3 = (*, *, 1, *, *, *, *, *, *, *, *, *, *, *, *) \\ c4 = (*, *, *, 1, *, *, *, *, *, *, *, *, *, *, *) \\ c5 = (*, *, *, *, 1, *, *, *, *, *, *, *, *, *, *) \\ c6 = (*, *, *, *, *, 0, *, *, *, *, *, *, *, *, *) \end{array}$$

esto significaría que en ese lugar esa literal no existe, entonces para evitar esta situación obligamos a la primera cláusula a ser puros 1 o 0, sin posibilidad a ser *:

$c1 = (1,1,0,1,0,1,0,0,0,0,1,1,0,1,1,0,1)$
 $c2 = (*,0,*,*,*,*,*,*,*,*,*,*,*,*,*,*)$
 $c3 = (*,*,1,*,*,*,*,*,*,*,*,*,*,*,*,*)$
 $c4 = (*,*,*,1,*,*,*,*,*,*,*,*,*,*,*,*,*)$
 $c5 = (*,*,*,*,1,*,*,*,*,*,*,*,*,*,*,*,*)$
 $c6 = (*,*,*,*,*,0,*,*,*,*,*,*,*,*,*,*,*)$

Entonces de esta manera evitamos que una cláusula me elimine todos los modelos y de que una variable deje de existir, para probar el sistema vamos a generar un archivo .txt con 20000 cláusulas y cada una con 100 variables, como se puede ver en la Figura 5.10:

The screenshot shows a text editor window titled 'f1.txt' with the path '~\Documentos/Tesis/program'. The editor contains a large block of text representing 20000 clauses. Each clause is a binary string of length 100, where '*' represents a variable and '0' or '1' represents a literal. The clauses are numbered from 19964 to 20000. The text is displayed in a monospaced font, and the editor has a standard toolbar with 'Abrir', a dropdown arrow, and a 'Guardar' icon.

Figura 1.10: Fórmula con 20000 cláusulas y 100 variables.

Como vimos en capítulos anteriores, una forma de saber si una fórmula en forma normal conjuntiva es satisfactible es mediante la tabla de verdad, pero para casos en los que tenemos muchas variables ya no es óptimo este proceso, ya que de acuerdo a la fórmula

$$2^n$$

tendríamos que hacer una tabla de verdad con $2^{100} = 1\,267\,650\,600\,000\,000\,000\,000\,000\,000\,000\,000$ lo cual, pues, resulta nada óptimo tanto en recursos como en tiempo, pero cuando ejecutamos el sistema, esté en pocos minutos y con pocos recursos nos puede decir si una fórmula con esta cantidad de variables y cláusulas es satisfactible o no y en caso de que no, implemente un proceso en el cual generaría satisfactibilidad en la fórmula no satisfactible.

Para esta fórmula con 20000 cláusulas y 100 variables el resultado se puede apreciar en la Figura 5.11:

```

sal666@sal666-Lenovo-ideapad-330-15IKB: ~/Documentos/Tests/program
'*,*,1,0,'*,1,0,'*,1,1,1,0,0,1,0,0,0,1,'*']
clause 19990 : [1,0,1,1,0,1,0,0,1,0,1,1,1,1,1,0,'*',0,0,0,1,0,0,0,0,0,1,'*',*,0,1,1,0,'*',*,1,0,1,0,
1,0,'*',1,1,1,'*,1,0,0,1,1,1,'*,1,1,1,'*,0,'*',1,1,1,0,0,1,1,'*',0,'*',*,1,0,0,0,'*',1,0,'*',*,1
,1,1,0,0,0,'*',0,0,'*',1,0,0,0,1,1,'*',0,0,1,0]
clause 19991 : [1,1,0,'*',1,'*',1,0,1,1,1,'*',0,0,1,0,1,0,0,1,1,1,0,0,0,0,0,1,'*',0,1,0,1,0,1,1,0,1,0
,1,0,'*',1,0,0,0,0,'*',1,'*',*,1,0,0,0,0,1,0,1,0,0,1,'*',0,0,0,1,0,'*',1,'*',*,1,1,0,1,0,1,'*',1,1
,0,0,'*',1,'*',1,0,1,'*',*,1,0,0,1,0,0,'*',1]
clause 19992 : [0,0,1,0,0,1,1,1,0,'*',*,1,1,1,1,0,0,1,1,0,1,'*',1,1,'*',1,0,1,0,'*',*,*,0,0,1,1,1,0
,0,0,0,0,1,1,1,0,0,0,'*',0,0,'*',1,1,'*',*,0,1,0,1,1,0,'*',*,1,'*',1,0,1,'*',*,1,'*',1,0,0,0,1,
1,1,1,0,1,1,0,0,'*',1,1,0,0,0,1,0,0,0,0,1]
clause 19993 : [1,'*',*,*,1,'*',0,1,'*',*,0,0,1,'*',*,*,0,0,1,'*',*,*,0,0,0,1,0,0,0,1,0,1,0,'*',0,1
,1,1,1,0,0,0,0,'*',1,0,1,0,1,0,1,1,1,'*',1,1,0,'*',1,0,0,1,0,0,'*',1,1,0,0,'*',1,0,0,1,0,1,1,0,
0,0,0,0,0,0,1,1,0,1,1,'*',1,1,'*',*,0,1,1]
clause 19994 : [1,0,'*',0,1,0,0,1,0,0,1,'*',1,0,1,'*',1,1,0,'*',1,1,0,'*',*,*,*,0,1,1,0,1,0,0,1,'*',0,1,
1,0,'*',0,0,1,'*',1,1,1,1,'*',0,0,0,0,1,0,'*',0,1,0,0,0,1,1,0,'*',1,1,1,0,0,0,0,0,1,1,1,0,0,1,1,'*',
1,0,1,0,0,'*',1,1,1,1,1,0,0,0,'*',*,0]
clause 19995 : [0,'*',0,0,'*',1,1,1,0,0,0,0,'*',1,0,1,0,'*',0,0,1,0,1,1,0,0,'*',1,1,1,1,0,1,0,0,1,'*',1,
1,1,1,'*',1,0,1,0,0,1,1,0,1,1,'*',0,0,1,'*',0,0,1,'*',1,'*',1,0,0,1,1,'*',0,1,1,1,1,1,1,0,'*',1,
1,1,1,1,'*',1,1,1,'*',0,'*',1,0,0,1,1,'*',0,1]
clause 19996 : [1,0,0,'*',*,0,'*',0,0,1,1,0,1,0,1,1,1,'*',0,1,0,1,0,1,0,'*',0,1,'*',0,0,0,0,1,1,1,0,0,0,
0,1,1,'*',1,1,1,0,0,'*',1,0,0,0,0,0,1,'*',0,1,'*',0,1,1,1,'*',1,0,0,'*',1,0,0,1,0,1,0,1,'*',*,0,0,'*
',0,0,0,0,0,1,0,1,'*',*,0,0,1,1,'*',1,1,0,1]
clause 19997 : [1,0,1,1,1,0,1,0,'*',1,0,1,1,0,1,1,1,1,0,0,0,1,0,1,1,1,0,1,1,0,0,1,1,0,'*',0,1,'*',0,'*'
,*,1,'*',1,0,1,1,1,'*',1,1,1,'*',0,1,0,1,1,'*',1,0,0,0,1,0,0,'*',0,'*',*,0,1,'*',0,'*',0,0,1,'*'
,0,1,0,0,1,1,0,1,1,0,0,1,1,1,0,1,0,1,0,'*']
clause 19998 : ['*',1,1,0,1,0,1,0,0,0,1,1,1,1,0,1,1,0,1,1,0,0,1,0,0,1,1,1,1,0,0,0,1,1,0,1,1,0,0,'*',
0,1,0,0,1,1,'*',0,1,0,0,0,1,'*',1,'*',*,1,1,1,'*',1,1,'*',0,1,'*',0,0,1,1,1,1,0,0,0,1,'*',1,
0,'*',0,1,'*',1,1,0,'*',1,0,0,1,'*',1]
clause 19999 : [0,1,1,1,1,'*',0,0,1,1,1,0,1,0,1,0,1,1,1,'*',0,'*',1,0,1,1,0,'*',0,0,'*',*,*,*,1,1
,0,0,1,0,1,0,'*',0,1,0,0,0,'*',*,0,0,0,1,0,0,0,0,'*',0,0,'*',1,0,1,1,0,0,0,1,1,0,'*',*,1,
1,0,0,'*',*,0,1,1,1,1,0,1,0,1,1,0,0,1]
Total of models in F is
1267650600228181890272422789120
-----
F is satisfiable
sal666@sal666-Lenovo-ideapad-330-15IKB:~/Documentos/Tests/program$

```

Figura 5.11: Fórmula con 20000 cláusulas y 100 variables.

Como vemos en la imagen, esta fórmula resulto ser satisfactible.

Capítulo 6

6. Conclusiones

Actualmente, los sistemas para poder saber si una fórmula en forma normal conjuntiva es satisfactible es mediante tablas de verdad y contando sus modelos resultantes, lo cual en fórmulas con pocas variables puede ser factible, pero cuando hablamos de muchas variables ya no es buena idea porque las tablas de verdad se forman de acuerdo al número de variables que tiene la fórmula, por ejemplo para una fórmula de 20 variables ya nos generaría una tabla de 1048576 renglones lo cual ya no es óptimo tanto en tiempo y recursos.

La propuesta de representar una fórmula en FNC utilizando la representación de 0,1 y asteriscos (*), nos permite mejorar el proceso de conteo de modelos de manera algorítmica.

En cuanto a la complejidad del algoritmo viene dada por el número de m cláusulas y el número de n variables de cada cláusula, se procesa cada cláusula una a una y se generan a lo más $n-1$ nuevas cláusulas que también son revisadas para generar independencia.

Para general el sistema se utiliza un ciclo *for* que va a ser el que se mueva entre las cláusulas de F y dentro de este *for* se llama a una función, que esta es recursiva, en otras palabras, se llama a sí misma dependiendo si se generan más cláusulas.

Lo que hace el sistema desarrollado es poder decidir si una fórmula en forma normal conjuntiva es satisfactible, en otras palabras que tenga modelos, y en caso de que no sea satisfactible, que el número de modelos sea cero, el sistema implementa una tabla de recuperación de modelos la cual nos regresa el número de modelos que nos recuperaría cada cláusula al ser eliminada de la fórmula, entonces cuando ya tenemos la tabla de modelos a recuperar, automáticamente se elige la cláusula que recuperaría más modelos y se procede a eliminarla de la fórmula original, ahora que tenemos esta nueva fórmula, se aplica de nuevo el algoritmo para verificar la satisfactibilidad de la fórmula, imprimiendo como resultado final la fórmula resultante, el total de modelos que tiene y si es satisfactible o no.

Este tipo de algoritmo que genera satisfactibilidad en fórmulas en FNC puede ser ocupado en el área de razonamiento automático, en bases de conocimiento, en la revisión de creencias y en otras aplicaciones de la inteligencia artificial.

Como trabajo a futuro se piensa en la optimización del sistema, utilizando el multiprocesamiento, cuando hablamos de multiprocesamiento nos referimos al uso de dos o mas CPU (*Central Processing Unit*) para la ejecución de uno o varios procesos de manera concurrente.

Entonces, teniendo en cuenta el termino de multiprocesamiento, después de leer una fórmula en forma norma conjuntiva en el formato de entrada del sistema, este sistema en lugar de trabajar en un solo proceso, se podrían crear n procesos dependiendo del número de *cores* que tiene la computadora sobre la que se esta ejecutando el sistema que genera satisfactibilidad en fórmulas no satisfactibles.

En otras palabras, dividir la fórmula en n partes, con n igual al número de cores en la computadora, por ejemplo, si tenemos una fórmula con 20000 cláusulas y el sistema se esta ejecutando sobre una computadora que tiene dos microprocesadores, entonces, la fórmula se dividirá en dos partes, tocando 10000 para cada CPU, optimizando el proceso y disminuyendo el tiempo de ejecución.

En teoría, si una fórmula con 20000 clausulas y cada cláusula con 100 variables se tarda 25 minutos, al dividir la fórmula en dos procesos, la ejecución del sistema debería de tardar ahora unos 12.5 minutos.

BIBLIOGRAFÍA.

- [1] Pedro Bello, Guillermo DeIta: Inference Algorithm with Falsifying Patterns for Belief Revision. MCPR 2019: 347-356.
- [2] Fermé E. Revisión de creencias. inteligencia artificial. Revista Iberoamericana de Inteligencia Artificial,17–39, 2007.
- [3] Patrick Suppes, Shirley Hill, Introducción a la lógica matemática,968-6708-01-4, 2004.
- [4] Rafael Farré Cirera, Robert Nieuwenhuis, Pilar Nivelaa Alós, Albert Olivaras Llunell, Enric Rodríguez Carbonell, Josefina Sierra Santibáñes, Lógica para Informáticos, logica proposicional, ISBN 978-607-707-1655-5.
- [5] Pedro Chávez Calderón. Introducción a la ciencia del razonamiento. Lógica. ISBN 970-24-0264-6.
- [6] G. Brassard, PP. Bratley. Fundamentos de algoritmia, ISBN 0-13-335068-1, 2008.
- [7] Gustavo López, Ismael Jeder, Augusto Vega. Análisis y diseño de ALGORITMOS, Implementación en C y Pscal. Alfaomega, México 2009.
- [8] José A. Jiménez Murillo. Matemáticas para la computación, ISBN 978-970-15-1401-6, 2008.
- [9] kripkit (2021, 16 de noviembre). DPLL. URL: <https://kripkit.com/dpll/>.
- [10] Santander (2021, 09 de abril). Python: que es y porque deberías de aprender a utilizarlo. URL:<https://www.becas-santander.com/es/blog/python-que-es.html>.
- [11] Fernando Cardellino, freecodecamp (2021, 20 de marzo). La guía definitiva del paquete Numpy para computación científica en Python. URL: <https://www.freecodecamp.org/espanol/news/la-guia-definitiva-del-paquete-numpy-para-computacion-cientifica-en-python/>.
- [12] Guillermo De Ita, José Raymundo Marcial-Romero, José Antonio Hernández Servín: A bottom-up algorithm for solving #2SAT. Log. J. IGPL 28(6): 1130-1140 (2020)
- [13] Wikipedia (2022, 20 de junio). P (clases de complejidad). URL: [https://es.wikipedia.org/wiki/P_\(clase_de_complejidad\)](https://es.wikipedia.org/wiki/P_(clase_de_complejidad)).