Universidad Autónoma del Estado de México

Facultad de Ingeniería

Un algoritmo para el conteo de modelos de fórmulas booleanas en 2-Forma Normal Conjuntiva

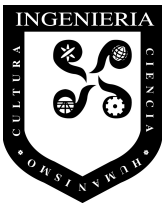
TESIS

QUE PARA OBTENER EL TÍTULO DE:

Ingeniero en Computación

PRESENTA:

Marco Antonio López Medina



ASESOR:
Dr. J. Raymundo Marcial Romero
COASESOR:
Dra. Rosa María Valdovinos Rosas

Toluca, México, Julio 2016

Resumen.

En este trabajo se presenta un algoritmo basado en grafos para obtener el número de asignaciones que pueden hacer verdadera una fórmula booleana en 2-Forma Normal Conjuntiva.

Se pretende resolver una parte del problema #SAT que es conocer el número de asignaciones que hacen verdadera una fórmula booleana que esta dada por disyunciones de conjunciones o expresado en su notación $c = (x_1 \vee \bar{x_2}) \wedge (x_2 \vee \bar{x_3})$.

Así mismo se pretende comparar este algoritmo con algunos ya existentes y que han sido probados de manera práctica para resolver problemas reales. Dado que los existentes han sido implementados, *sharpSAT* y *relsat*, para resolver problemas de una base de datos llamada *satlib* también se realizó la implementación del algoritmo presentado ya que en su documentación no proporcionan cuál es la complejidad del algoritmo que utilizan.

Para realizar las pruebas comparativas entre la herramienta desarrollada y las existentes se generaron instancias de prueba en las cuales se busca una ventaja para las otras herramientas y comparar cuales son los peores casos generados, en los que esta herramienta tarda un mayor tiempo que las otras en regresar un resultado.

Los resultados obtenidos fueron satisfactorios ya que aunque no se pude mejorar la herramienta más reciente, dado por su pre-procesamiento de la entrada y el cache de componentes de la entrada se pudo mejorar el desempeño de la segunda herramienta que se tomó como referencia en muchos casos y que es al menos la segunda más eficiente reportada en la literatura.

Como trabajo futuro se contempla integrar al algoritmo algunas estrategias que fueron implementadas en las herramientas *sharpSAT* y *relsat*, con lo que se espera que se alcance o supere la eficiencia de *sharpSAT*.

Índice general

Introducción

La lógica proposicional es una rama de la lógica clásica que estudia las variables proposicionales o sentencias lógicas, sus posibles implicaciones, evaluaciones de verdad y en algunos casos su nivel absoluto de verdad.

Sea $X = \{x_1, \ldots, x_n\}$ un conjunto de n variables booleanas (es decir que pueden tomar únicamente dos valores de verdad) [8]. Una literal es una variable x_i o la variable negada $\bar{x_i}$. Una cláusula es una disyunción de literales distintas. Una fórmula booleana en forma normal conjuntiva (CNF, Conjunctive Normal Form) F es una conjunción de cláusulas. Sea $v(Y)$ el conjunto de variables involucradas en el objeto Y, donde Y puede ser una literal, una cláusula o una fórmula booleana. Por ejemplo, para la cláusula $c = \{x_1 \vee \bar{x_2}\}, v(c) = \{x_1, \alpha_2\}$ y $Lit(c) = \{x_1, x_2, \bar{x_1}, \bar{x_2}\}$ es el conjunto de literales que aparecen en c.

Una asignación s es un conjunto de literales tomadas de una fórmula F, con la condición de que sólo una de las literales x_i o $\bar{x_i}$ pertenece a s. Puede también considerarse como un conjunto de pares de literales no complementario. Si $x_i \in s$, siendo s una asignación, entonces s convierte x_i en verdadero y $\bar{x_i}$ en falso. Por otro lado si $\bar{x_i} \in s$ entonces s convierte a $\bar{x_i}$ en verdadero y x_i en falso. Considerando una cláusula c y una asignación s como un conjunto de literales, se dice que c se satisface por s sí y solo si $c \cap s \neq \emptyset$ y si para toda $x_i \in c, \bar{x_i} \in s$ entonces s falsifica a c. Sea F una fórmula booleana en CNF, se dice que F se satisface por la asignación s si cada cláusula en F se satisface por s. Por otro lado, se dice que F se contradice por s si al menos una cláusula de F se falsifica por s. Un modelo de F es una asignación para $v(F)$ tal que satisface F. Se denota como $SAT(F)$ al conjunto de modelos de la fórmula F. Dada una fórmula F en CNF, SAT consiste en determinar si F tiene un modelo, mientras que #SAT consiste en contar el número de modelos que tiene F sobre $v(F)$. Por otro lado, #2SAT denota #SAT para fórmulas en 2-CNF.

El problema #SAT pertenece a la clase #P completo [11], por tanto no existe algoritmo eficiente que resuelva el problema en general. Existen implementaciones que permiten calcular los modelos de una fórmula dada como por ejemplo, relsat [13] la cual es una herramienta basada en el algoritmo DPLL (Davis-Putnam-Logemann-Loveland algorithm) de búsqueda exhaustiva de asignaciones. La mejora que provee en comparación a otras herramientas es

el poder procesar rápidamente fórmulas disjuntas, es decir, que las variables de una fórmula no intervengan en alguna otra. Cachet [10] es una herramienta basada en el modelo de conteo Caching que es sucesora del árbol de búsqueda de un modelo de conteo DPLL; ajustando variables y simplificando la fórmula, ya que se pueden encontrar sub-fórmulas que han aparecido en una rama del árbol de búsqueda. Si esto sucede, no es necesario volver a contar los modelos, sólo es necesario saber el conteo de la rama de búsqueda anterior y utilizarlo. Por otro lado sharpSAT [9] está basada en el modelo caching e implementa un mejor razonamiento en cada nodo que evalúa. Utiliza una técnica conocida como *look ahead* que permite hacer una prueba sobre alguna variable en la fórmula para ver si es necesario que sea siempre verdadera o falsa, permitiendo que solo se cuenten los modelos necesarios para el resto de la fórmula.

En esta tesis, se propone un algoritmo para el conteo de modelos convirtiendo la fórmula de entrada en un grafo. En [12] se mostró como contar modelos en tiempo polinomial en grafos cuya forma es un camino o un árbol, por lo que se tomará el procedimiento como referencia para mostrar como contar en otro tipo de grafos. Así mismo, se comparará con sharpSAT que es la herramienta más eficiente (en términos de tiempo y espacio) para el conteo de modelos .

Justificación

La complejidad computacional estudia la "dificultad" inherente de problemas de importancia teórica y/o práctica. El esfuerzo necesario para resolver un problema de forma eficiente puede variar.

Un problema se denomina #P si no existe un algoritmo eficiente para encontrar una solución óptima. Probar que un problema es #P es importante puesto que permite encontrar un algoritmo para la solución óptima y centrarse en objetivos realizables (encontrar algoritmos para obtener soluciones a una parte del problema.

El desarrollo de algoritmos para resolver problemas #P, en este caso #2SAT tiene como aplicaciones estimar el grado de veracidad en teorías proposicionales [5], generación de explicaciones a preguntas proposicionales, reparación de bases de datos inconsistentes, inferencia bayesiana [2, 3, 4, 5, 6, 7] entre otras.

Existen formas de representar una base de conocimiento mediante modelos representativos, con esto la verificación de validez de una proposición puede reducirse a un problema combinatorio y no a utilizar un método de razonamiento como los motores de inferencia, lo que permite saber si la base de conocimiento implica a la proposición, lo que la hace verdadera.

Las implementaciones actuales como sharpSAT [9] o relsat [13] no presentan un análisis de complejidad de sus algoritmos, por lo que no se conoce cómo se comportaran en diferentes instancias. En esta propuesta, se estudiará la complejidad computacional del algoritmo para conocer en qué instancias es mas eficiente.

Para desarrollar esta propuesta se realizará una transformación de la fórmula booleana en un grafo el cuál será descompuesto en árbol y coárbol, que será la entrada al algoritmo.

Hipótesis

Existe un algoritmo basado en la técnica de descomposición árbol-coárbol de la fórmula de entrada que pueda resolver algunas instancias de la base de datos de satlib [14] para el problema #2SAT de forma eficiente comparado a las implementaciones sharpSAT [9] y relsat [13].

Objetivo general

Desarrollar un algoritmo para el conteo de modelos de fórmulas booleanas en 2-forma normal conjuntiva, basándose en su representación mediante grafo y su descomposición en árbol-coárbol.

Capítulo 1

Marco Teórico

1.1. Grafo

Muchas situaciones del mundo real pueden ser descritas por medio de un diagrama que consiste de un conjunto de puntos y de líneas que unen pares de ellos.

Por ejemplo los puntos pueden representar personas, y las líneas unen pares de amigos; o los puntos pueden ser centros de comunicación y las líneas representan las comunicaciones entre ellos. En ciencias de la computacón a estos diagramas se les conoce como grafos y se definen formalmente como:

Definición 1 *Un grafo G es un par ordenado $(V(G), E(G))$ donde $V(G)$ es el conjunto de vértices y $E(G)$ es el conjunto de aristas, unidas con una función de incidencia φ_G que asocia cada arista de G con un par no ordenado de (no necesariamente distintos) vértices de G. Si e es una arista y u y v son vértices tal que $\varphi_G(e) = \{u, v\}$ entonces se dice que e une u y v, y los vértices u y v son llamados extremos de e. Para denotar el número de vértices y aristas en G se utiliza $v(G)$ y $e(G)$ llamados orden y tamaño de G, respectivamente.*

Se dice que si un vértice está conectado con otro mediante una arista éstos son adyacentes, y uno incide con el otro, además si los vértices unidos son distintos se dice que son vecinos. Tomando el concepto anterior la representación gráfica de una arista con vértices u y v es:

$$u \text{ — } v$$

Una arista con extremos idénticos es llamado un ciclo simple o loop, dos o más aristas con el mismo par de extremos son llamadas aristas paralelas, la Figura 1.1 muestra un ejemplo. Un grafo simple es aquel que no tiene loops o aristas paralelas.

Un *camino* es un grafo simple cuyos vértices pueden ser ordenados en una secuencia lineal de tal forma que dos vértices son adyacentes si son consecutivos en la secuencia, o no

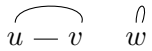


Figura 1.1: De izquierda a derecha se muestra un par de aristas paralelas y un ciclo simple

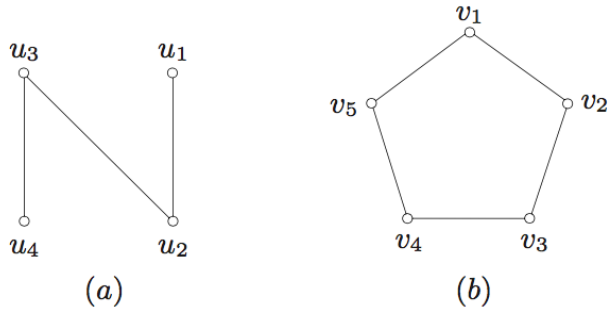


Figura 1.2: (a) Un camino de longitud tres, y (b) un ciclo de longitud cinco

adyacentes si no lo están. Un ciclo en tres o más vértices es un grafo simple cuyos vértices pueden ser ordenados en una secuencia cíclica de manera que dos de ellos son adyacentes si son consecutivos en la secuencia, o no adyacentes si no lo están; un ciclo en un vértice consiste en un vértice con un loop, y un ciclo en dos vértices consiste de dos vértices unidos por un par de aristas paralelas. La longitud de un camino o un ciclo es el número de aristas que incluye (Figura 1.2).

Un grafo es conectado si, para cada partición de sus vértices dentro de dos conjuntos no vacíos X y Y, existe una arista con un extremo en X y un extremo en Y, de otra forma se considera un grafo no conectado. En otras palabras, un grafo es desconectado si su conjunto de vértices puede ser dividido en dos conjuntos no vacíos X y Y y que ninguna arista tenga un extremo en X y otro en Y.

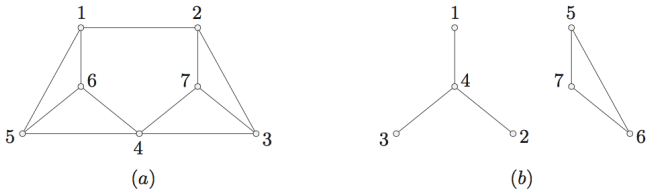


Figura 1.3: (a) Un grafo conectado, y (b) un grafo no conectado

1.1.1. Subgrafos

Dado un grafo G, existen dos caminos naturales para derivar en grafos más pequeños de G. Si e es una arista de G, se puede obtener un grafo con $m-1$ aristas eliminando a e de G pero dejando los vértices y las aristas restantes intactas. El grafo resultado se denota por $G\backslash e$. De manera similar, si v es un vértice de G, se puede obtener un grafo con $n-1$ vértices eliminando de G el vértice v junto con las aristas incidentes en él. El grafo resultado se denota por $G-v$.

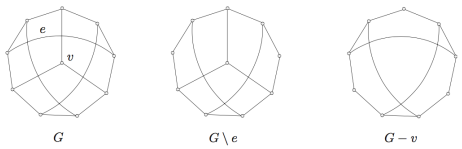


Figura 1.4: Subgrafos resultantes de eliminar la arista e y el vértice v respectivamente

Un grafo es acíclico si no contiene un ciclo. Esto quiere decir que un grafo acíclico debe contener un vértice de grado menor a dos.

Una arista de corte es aquella que al ser eliminada de un grafo conectado, lo convierte en un grafo desconectado.

Un subgrafo de expansión de un grafo G es un subgrafo obtenido por medio de eliminación de aristas solamente, en otras palabras es un subgrafo cuyo conjunto de vértices es el conjunto de vértices completo de G. Si S es el conjunto de aristas eliminados, el subgrafo de G se denota como $G\backslash S$.

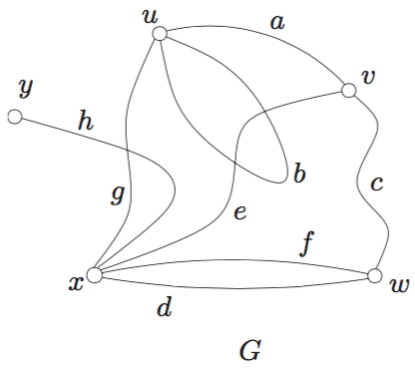
Un grafo acíclico conectado es llamado un árbol, cada componente de un grafo acíclico es un árbol, por esta razón son llamados bosques.

Si G es un grafo conectado y no es un árbol, y e es una arista de ciclo de G, entonces $G \setminus e$ es un subgrafo de expansión de G, ya que e no es una arista de corte de G. Repitiendo este proceso de eliminación de aristas en ciclos hasta que cada arista que se mantenga sea una arista de corte, se obtiene un árbol de expansión de G.

El complemento $E\backslash T$ de un árbol de expansión T es llamado coárbol, y se denota $\bar{T}$. Para cada arista $e = xy$ de un coárbol $\bar{T}$ de un grafo G, existe un camino xy único en T que conecta a los extremos, llamado $P = xTy$. Entonces $T + e$ contienen un ciclo único, este ciclo es llamado ciclo fundamental de G con respecto a T y e.

1.1.2. Búsqueda primero en profundidad

La matriz de adyacencia de G es de tamaño $n \times n$ $A_G := (a_{uv})$, dónde a_{uv} es el número de aristas que unen los vértices u y v, cada loop cuenta como dos aristas en la matriz.



	u	v	w	x	y
u	2	1	0	1	0
v	1	0	1	1	0
w	0	1	0	2	0
x	1	1	2	0	1
y	0	0	0	1	0

A

Figura 1.5: Grafo G y matriz de adyacencia A

La búsqueda primero en profundidad está basada en el concepto de árbol, en la cual el vértice añadido al árbol T en cada etapa es aquel que es vecino de la adición más reciente a T, en otras palabras se debe realizar una búsqueda en la lista de adyacencia para el vértice más recientemente añadido x para un vecino que no está en T. Si existe ese vecino, es añadido a T, si no se retrocede al vértice añadido a T justo antes de x y se examinan sus vecinos, hasta que todos los vértices son añadidos a T.

1.2. Lógica

1.2.1. ¿Qué es una lógica?

Una lógica siempre constará de dos partes:

- **Sintaxis**. Especifica los símbolos de los que dispone el lenguaje y la manera en la que se puedan combinar dichos símbolos para construir fórmulas.
- **Semántica**. Incluye los elementos necesarios para asignar significados a los símbolos y a las fórmulas de dicha lógica. Concretamente debe contener:
  - Una definición del concepto de *interpretación* I, que especifíca el significado que puede asignarse a cada símbolo.

- Una definición del concepto de *satisfacción*, que establece las condiciones que deben cumplirse para que una interpretación I *satisfaga* una fórmula F, esto es, para que podamos afirmar que la fórmula F es *cierta* en la interpretación I.

En una lógica también suele haber métodos de deducción, que describen cómo a partir de unas fórmulas dadas se pueden inferir otras nuevas.

1.2.2. Lógica proposicional

Sintaxis. En esta lógica, el vocabulario es un conjunto de *variables* P . Una *fórmula* de la lógica proposicional sobre P se define como:

- Toda variable x_i de P es una fórmula.
- Si F y G son fórmulas, entonces $(F \vee G)$ y $(F \wedge G)$ son fórmulas.
- Si F es una fórmula, entonces $\neg F$ es una fórmula.
- Nada más es una fórmula.

Interpretación. Una *interpretación* I sobre el vocabulario P es una función $I : P \to \{0, 1\}$, es decir, I es una función que, para cada variable de enunciado, dice si es 1 (cierto, *true*) o 0 (falso, *false*).

Satisfacción. Sean I una interpretación y F una fórmula, ambas sobre el vocabulario P. La *evaluación* en I de F, denotada $eval_i(F)$, es una función que por cada fórmula da un valor 0 o 1. Definimos $eval_i(F)$ para todos los casos posibles de F, usando *min, max* y -, que denotan respectivamente el mínimo, el máximo, y la resta (sobre números del conjunto {0,1}), dónde , $min(0,0) = min(0,1) = min(1,0) = 0$ y $min(1,1) = 1, max(1,1) = max(0,1) = max(1,0) = 1$ y $max(0,0) = 0$:

- si F es un símbolo p de P entonces $eval_i(F) = I(p)$.

$eval_i(F \vee G) = min(eval_i(F), eval_i(G))$

- $eval_i(F \wedge G) = max(eval_i(F), eval_i(G))$

$eval_i(\neg F) = 1 - eval_i(F)$

Se define: si $eval_i(F) = 1$ entonces I satisface F, denotado $I \models F$. En este caso también se dice que F *es cierta en* I, o que I *es un modelo de* F.

1.2.3. Satisfactibilidad, tautología, consecuencia y equivalencia

1. Una fórmula F es *satisfactible* si tiene algún modelo, es decir, si existe alguna interpretación I tal que $I \models F$. Una fórmula F es insatisfactible (o es una contradicción)

si no es satisfactible.

2. Una fórmula F es una tautología (o es válida), si toda interpretación es modelo de F, es decir, si para toda interpretación I se tiene $I \models F$. Por ejemplo: $(x \vee \bar{x})$.
3. Sean F y G fórmulas construidas sobre un vocabulario P. F y G son lógicamente equivalentes si tienen los mismos modelos, es decir, si para toda interpretación I sobre P se tiene que $I \models G$ si y sólo si $I \models F$. Esto se denotará por $G = F$.

1.2.4. Forma Normal Conjuntiva (CNF)

Una expresión lógica, la cual contiene variables x_i y los conectivos lógicos $\wedge, \vee$ y $\neg$ (AND, OR y NOT respectivamente). Se usa la notación $\bar{x_i}$ para denotar la negación $\neg x_i$, y el término literal para referirse a una x_i o a una $\bar{x_i}$. El que la expresión esté en *forma normal conjuntiva* significa que es una conjunción:

$$C_1 \wedge C_2 \wedge \ldots \wedge C_c$$

de subexpresiones C_i, cada una de las cuales es una disyunción de literales

$$(x_1 \vee x_3 \vee x_4) \wedge (\bar{x_1} \vee x_3) \wedge (\bar{x_1} \vee x_4 \vee \bar{x_2}) \wedge \bar{x_3} \wedge (x_2 \vee \bar{x_4})$$

es una expresión en Forma Normal Conjuntiva con cinco *elementos conjuntivos*. En general una variable podría aparecer más de una vez en un elemento conjuntivo, y este último incluso podría duplicarse.

El problema de satisfacción-CNF (o CNF-Sat) es el siguiente: Dada una expresión en forma normal conjuntiva, ¿existe una asignación de verdad que la satisfaga?.

El problema de conteo de modelos, mejor conocido por #SAT consiste en calcular el número de modelos de una fórmula booleana dada, es decir, el número de asignaciones de verdad para las cuales la fórmula se evalúa como verdadera. En este caso el problema que se aborda directamente es #2SAT que es el conteo de modelos sobre una fórmula booleana en forma Normal Conjuntiva de dos variables (2-Conjunctive Normal Form). El problema #2SAT pertenece a la clase #P completo, por tanto no existe algoritmo eficiente que resuelva el problema en general.

1.3. Problemas tratables

Se considera que un problema es tratable si se puede resolver en tiempo polinómico $O(n^k)$ para algún k. El conjunto de estos problemas se llama clase de complejidad P. Por ejemplo: ordenar una lista de valores o buscar el camino mínimo entre dos ciudades en un mapa.

Otros problemas para los cuales no se conocen algoritmos que los resuelvan en tiempo polinómico, se denominan NP. Por ejemplo: el problema de suma de subconjuntos.

NP es el conjunto de problemas cuya solución se puede comprobar en tiempo polinómico pero no existe procedimiento que obtenga la solución en tiempo polinómico. Por ejemplo: en la suma de subconjuntos se puede comprobar si un subconjunto es solución sumando sus elementos ($O(n)$), pero encontrar esa solución es la parte difícil.

Una Máquina de Turing es una máquina que manipula símbolos en una cinta según un conjunto de reglas. Se compone de:

- Una *cinta* infinitamente larga (por la derecha) donde se pueden leer/escribir símbolos.
- Una *cabeza lectora/escritora* que apunta a una posición de la cinta y se puede mover en ambas direcciones.
- Un *control de estado finito*:
  - Un registro que almacena el *estado* en que está la máquina en un momento dado.
  - La máquina utiliza el estado actual, y el símbolo leído de la cinta, para decidir qué hacer.

Definición 2 *Una* máquina de turing *(MT) es una 5-tupla* $T = (Q, \Sigma, \Gamma, q_0, \delta)$*, donde:*
Q *es un conjunto finito de estados, del cual se supone que no incluye* h_a *ni* h_r*, los dos estados de detención (se usan los mismos símbolos para los estados de detención de todas las MT);*
Σ *y* Γ *son conjuntos finitos, los alfabetos de* entrada *y de* cinta*, respectivamente, con* $\Sigma \subseteq \Gamma$*;* Γ *se supone que no contiene* Δ*, el símbolo del espacio en* blanco*;*
q_0 *el estado inicial, es elemento de* Q*;*
δ*:* $Q \times (\Gamma \cup \{\Delta\}) \to (Q \cup \{h_a, h_r\}) \times (\Gamma \cup \{\Delta\}) \times \{R, L, S\}$ *es una función parcial (es decir, posiblemente indefinida en ciertas partes).*

Definición 3 ***Clase de complejidad*** P*: Conjunto de problemas de decisión que se pueden resolver con una Máquina de Turing que termina su ejecución en un número de pasos acotado por una función Polinómica,* $O(n^k)$*.*

Definición 4 ***Clase de complejidad*** NP*: Conjunto de problemas de decisión que se pueden resolver con una máquina de Turing* no determinista *que termina su ejecución en un número de pasos acotado por una función polinómica,* $O(n^k)$*.*

Una máquina de Turng no determinista permite resolver problemas de complejidad exponencial en un tiempo polinómico. Partiendo de la suposición de que la máquina toma la mejor decisión de su árbol computacional.

Todo lo que se puede computar con una Máquina de Turing no determinista se puede hacer también con una determinista, pero no con la misma eficiencia, de ahí que pueda haber

problemas en NP pero no en P.

1.3.1. El grafo restringido de una 2-CNF

Existen algunas representaciones gráficas de una Forma Normal Conjuntiva, por ejemplo el grafo primal signado o también conocido como grafo restringido.

Sea F una 2-CNF, su grafo restringido se denota por $G_F = (V(F), E(F))$ con $V(F) = v(F)$ y $E(F) = \{\{v(x), v(y)\} : \{x \vee y\} \in F\}$, esto es, los vértices de G_F son las variables de F, y para cada cláusula $\{x \vee y\}$ en F existe una arista $\{v(x), v(y)\} \in E(F)$. Para $x \in V(F), \delta(x)$ denota su grado, es decir el número de aristas incidentes en x. Cada arista $c = \{v(x), v(y)\} \in E(F)$ se asocia con un par (s_1, s_2) de signos, que se asignan como etiquetas de la arista que conecta las variables de la cláusula. Los signos s_1 y s_2 pertenecen a las variables x y y respectivamente. Por ejemplo la cláusula x^0, y^1 determina la arista con etiqueta "$x \overset{-+}{—} y$", que es equivalente a la arista "$y \overset{+-}{—} x$".

Sea $S = \{+, -\}$ un conjunto de signos. Un grafo con aristas etiquetadas en S es el par (G, ψ), dónde $G = (V, E)$ es un grafo restringido, y ψ es una función con dominio E y rango S. $\psi(e)$ se denomina a la etiqueta de la arista $e \in E$. Sea $G = (V, E, \psi)$ un grafo restringido con aristas etiquetadas en $S \times S$ y x y y nodos en V, si $e = \{x, y\}$ es una arista y $\psi(e) = (s, s')$, entonces $s(s')$ es el signo adyacente a $x(y)$.

Sea G un grafo conectado de n vértices, un árbol de expansión de G es un subconjunto de $n-1$ aristas tal que forman un árbol de G. Se denomina co-árbol al subconjunto de aristas que son el complemento de un árbol.

1.4. Formato Dimacs

Es un formato utilizado para almacenar fórmulas booleanas en forma normal conjuntiva. La forma de almacenamiento de una fórmula en el archivo está dada por algunas reglas:

- El archivo puede iniciar con lineas de comentario. La primera letra de cada línea de comentario debe ser una letra “c” minúscula, pueden ir en cualquier parte del archivo.
- Después de las lineas de comentario iniciales sigue la línea que describe el problema. Ésta inicia con una “p” minúscula seguida de un espacio, después el tipo de problema que para los archivos CNF es “cnf”, un espacio seguido de el número de variables, un espacio y por último el número de cláusulas.
- El resto del archivo contiene las lineas que definen las cláusulas una a una.

- Una cláusula se define listando los índices de una literal positiva y un índice negativo para cada literal negativa, el índice 0 no es permitido.
- La definición de una cláusula se puede extender mas allá de una línea de texto. La definición de una cláusula es terminada por un valor final de 0.
- El archivo termina después de que se define la última cláusula.

Ejemplo 1 *Dada la expresión* $(x_1 \vee \neg x_3) \wedge (x_1 \vee \neg x_2)$ *el archivo en formato Dimacs es:*

c
c
p cnf 3 2
1 -3 0
1 -2 0

1.5. relsat (1997)

Es una herramienta basada en el algoritmo DPLL (Davis-Putnam algorithm, equivalente a una búsqueda hacia atrás con verificación hacia adelante) de búsqueda exhaustiva de asignaciones. La mejora que provee en comparación a otras herramientas es el poder procesar rápidamente fórmulas disjuntas, es decir, que las variables de una fórmula no intervengan en alguna otra.

Implementa una mejora del algoritmo DPLL para resolver instancias de 3SAT que serían difíciles o imposibles de resolver de forma eficiente sin esta mejora, realiza una mejora sobre *look-back* en el algoritmo.

1.6. cachet

Es una herramienta basada en el modelo de conteo Caching que es sucesora del árbol de búsqueda de un modelo de conteo DP; ajustando variables y simplificando la fórmula, ya que se pueden encontrar sub-fórmulas que han aparecido en una descomposición del algoritmo DP. Si esto sucede, no es necesario volver a contar los modelos, sólo es necesario saber el conteo de la fórmula que ya fue evaluada y utilizarlo.

1.7. sharpSAT (2012)

Es una herramienta para resolver #SAT basada en un algoritmo de búsqueda exhaustiva (DPLL) el cuál requiere conocer qué literales en la fórmula puede utilizar para realizar una ramificación, requiere heurísticas sobre cómo elegir estas literales. También utiliza descomposición de componentes y cache de componentes.

1.8. Flex

Flex es una herramienta para buscar patrones léxicos en un texto, la cuál requiere una descripción de reglas mediante expresiones regulares y código en C, con esto genera un ejecutable que analiza las ocurrencias de las expresiones regulares y ejecuta el código en C correspondiente.

1.9. Bison

Bison es un generador de parsers de propósito general que convierte una gramática libre de contexto en un parser LR, que permite utilizar un método de lookahead para saber cómo evaluar los símbolos más recientes. Requiere conocimientos de C o C++ para ser utilizado.

1.10. GMP (GNU Multiple Precision arithmetic library)

GMP es una librería libre para precisión aritmética, opera con enteros con signo, números racionales y números de punto flotante. No tiene un límite práctico en la precisión, excepto la memoria disponible en la computadora que lo utiliza.

Está diseñado cuidadosamente para ser tan rápido como sea posible para pequeños y grandes operandos, utilizando algoritmos rápidos con un código ensamblador altamente optimizado para un conjunto de CPUs.

Capítulo 2

Algoritmos para el conteo de modelos para fórmulas en 2-CNF

2.1. Conteo de Modelos en Grafos Acíclicos

El conteo de modelos para fórmulas booleanas cuyo grafo restringido es acíclico se puede realizar en tiempo polinomial [15]. A continuación se presenta un algoritmo que, mediante un recorrido en preorder, permite contar modelos en un grafo acíclico.

2.1.1. #2SAT para fórmulas en 2-CNF que representan un camino

Se dice que un grafo G_F representa un camino para una 2-CNF F, si

$$F = \{C_1, C_2, \ldots, C_m\} = \{\{x_1^{\epsilon_1} \vee x_2^{\delta_1}\}, \{x_2^{\epsilon_2} \vee x_3^{\delta_2}\}, \ldots, \{x_m^{\epsilon_m} \vee x_{m+1}^{\delta_m}\}\},$$

donde $\delta_i, \epsilon_i \in \{0, 1\},\ i \in [1..m]$. Sea f_i una familia de cláusula de la fórmula F construida como sigue: $f_1 = \emptyset; f_i = \{C_j\}_{j<i},\ i \in [1..m]$. Note que $n = |\upsilon(F)| = m + 1,\ f_i \subset f_{i+1},\ i \in [1..m-1]$. Sea $SAT(f_i) = \{s : s \text{ satisface } f_i\},\ A_i = \{s \in SAT(f_i) : x_i \in s\},\ B_i = \{s \in SAT(f_i) : \overline{x}_i \in s\}$. Sea $\alpha_i = |A_i|;\ \beta_i = |B_i|$ y $\mu_i = |SAT(f_i)| = \alpha_i + \beta_i$.

Para cada nodo $x \in G_F$ se calcula el par (α_x, β_x), donde α_x indica el número de veces que la variable x toma el valor 'verdadero' y β_x indica el número de veces que la variable x toma el valor 'falso' en el conjunto de modelos de F. El primer par es $(\alpha_1, \beta_1) = (1, 1)$ ya que x_1 puede tomar el valor verdadero o falso para satisfacer a f_1. Los pares (α_x, β_x) asociados a cada nodo $x_i,\ i = 2, \ldots, m$ se calculan de acuerdo a los signos (ϵ_i, δ_i) de las literales en la cláusula c_i por la siguiente ecuación de recurrencia:

$$(\alpha_i, \beta_i) = \begin{cases} (\beta_{i-1} & ,\alpha_{i-1} + \beta_{i-1}) \text{ if } (\epsilon_i, \delta_i) = (-,-) \\ (\alpha_{i-1} + \beta_{i-1} & ,\beta_{i-1}) \text{ if } (\epsilon_i, \delta_i) = (-,+) \\ (\alpha_{i-1} & ,\alpha_{i-1} + \beta_{i-1}) \text{ if } (\epsilon_i, \delta_i) = (+,-) \\ (\alpha_{i-1} + \beta_{i-1} & ,\alpha_{i-1}) \text{ if } (\epsilon_i, \delta_i) = (+,+) \end{cases} \tag{2.1}$$

Note que, si $F = f_m$ entonces $\#SAT(F) = \mu_m = \alpha_m + \beta_m$. Se denota con $\rightarrow$ la aplicación de una de las cuatro reglas de recurrencia (2.1).

Ejemplo 2 *Sea* $F = \{(x_1, x_2), (\overline{x}_2, \overline{x}_3), (\overline{x}_3, \overline{x}_4), (x_4, \overline{x}_5), (\overline{x}_5, x_6)\}$ *una fórmula en 2-CNF cuyo grafo restringido representa un camino. Las series* $(\alpha_i, \beta_i), i \in [1..6]$*, se calculan como:* $(\alpha_1, \beta_1) = (1,1) \rightarrow (\alpha_2, \beta_2) = (2,1)$ *ya que* $(\epsilon_1, \delta_1) = (+,+)$*, y la regla 4 tiene que aplicarse. En general, aplicando la regla correspondiente de la recurrencia (2.1) de acuerdo a los signos de* (ϵ_i, δ_i)*,* $i = 2, ..., 5$*, se tiene que* $(2,1) \rightarrow (1,3) \rightarrow (3,4) \rightarrow (3,7) \rightarrow (\alpha_6, \beta_6) = (10,7)$*, y entonces,* $\#SAT(F) = \mu_6 = \alpha_6 + \beta_6 = 10 + 7 = 17$*.*

2.1.2. #2SAT para fórmulas en 2-CNF cuyo grafo contiene aristas paralelas

Se considera el caso dónde en una fórmula F en 2-CNF existen dos cláusulas que involucran las mismas variables. En este caso, el cálculo debe considerar cuatro signos diferentes para calcular #2SAT como el caso de un camino. Suponiendo que las dos cláusulas son $c_k = (x_{i-1}^{\epsilon_k}, x_i^{\delta_k})$ y $c_j = (x_{i-1}^{\epsilon_j}, x_i^{\delta_j})$ las cuáles involucran las variables x_{i-1} y x_i. Entonces se calculan los valores para (α_i, β_i) asociados al nodo x_i, de acuerdo a los signos (ϵ_k, δ_k) y (ϵ_j, δ_j) como sigue:

$$(\alpha_i, \beta_i) = \begin{cases} (\alpha_{i-1} & ,\alpha_{i-1}) \text{ if } (\epsilon_k, \delta_k) = (1,1) \textit{ and } (\epsilon_j, \delta_j) = (1,0) \\ (\mu_{i-1} & ,0) \text{ if } (\epsilon_k, \delta_k) = (1,1) \textit{ and } (\epsilon_j, \delta_j) = (0,1) \\ (\beta_{i-1} & ,\alpha_{i-1}) \text{ if } (\epsilon_k, \delta_k) = (1,1) \textit{ and } (\epsilon_j, \delta_j) = (0,0) \\ (\alpha_{i-1} & ,\beta_{i-1}) \text{ if } (\epsilon_k, \delta_k) = (1,0) \textit{ and } (\epsilon_j, \delta_j) = (0,1) \\ (0 & ,\mu_{i-1}) \text{ if } (\epsilon_k, \delta_k) = (1,0) \textit{ and } (\epsilon_j, \delta_j) = (0,0) \\ (\beta_{i-1} & ,\beta_{i-1}) \text{ if } (\epsilon_k, \delta_k) = (0,1) \textit{ and } (\epsilon_j, \delta_j) = (0,0) \end{cases} \tag{2.2}$$

Siendo F una fórmula en 2-CNF tal que tres cláusulas en F involucran las mismas variables, entonces el valor de (α_i, β_i) es calculado por la recurrencia (2.3).

$$(\alpha_i, \beta_i) = \begin{cases} (0 \quad, \alpha_{i-1}) \text{ if } \{(x_{i-1}, x_i), (x_{i-1}, \overline{x}_i), (\overline{x}_{i-1}, \overline{x}_i)\} \subseteq F \\ (\mu_{i-1}, 0 \quad) \text{ if } \{(x_{i-1}, x_i), (x_{i-1}, \overline{x}_i), (\overline{x}_{i-1}, x_i)\} \subseteq F \\ (\beta_{i-1}, \alpha_{i-1}) \text{ if } \{(x_{i-1}, x_i), (\overline{x}_{i-1}, x_i), (\overline{x}_{i-1}, \overline{x}_i)\} \subseteq F \\ (\alpha_{i-1}, \beta_{i-1}) \text{ if } \{(\overline{x}_{i-1}, x_i), (x_{i-1}, \overline{x}_i), (\overline{x}_{i-1}, \overline{x}_i)\} \subseteq F \end{cases} \tag{2.3}$$

En el caso de que cuatro aristas paralelas con los mismos extremos y distinta combinación de signos indica que la fórmula F es insatisfactible y por lo tanto $\#2\text{SAT}(F) = 0$.

Procesamiento de Cláusulas Unitarias: Una cláusula unitaria representa un loop en el grafo de una fórmula en 2-CNF. Cuando (α_i, β_i) son calculados sobre un nodo x_i el cual tiene una arista de loop, se aplica la recurrencia (2.4).

$$(\alpha_i, \beta_i) = \begin{cases} (0, \beta_i) & \text{if} \quad (\overline{x}_i) \in U \\ (\alpha_i, 0) & \text{if} \quad (x_i) \in U \end{cases} \tag{2.4}$$

Si una cláusula unitaria solo determina los valores de su variable. Cuando ambos x_i y $\bar{x}_i$ existen en la fórmula F como cláusulas unitarias, entonces F es insatisfactible.

Aristas paralelas y cláusulas unitarias deben ser consideradas en un pre-procesamiento de la fórmula antes de aplicar el algoritmo de conteo general.

2.1.3. Conteo en grafos acíclicos

Sea F una fórmula en 2-CF donde su grafo asociado G_F es acíclico. Se puede asumir que G_F tiene un nodo raíz, un recorrido del grafo permite generar un árbol que tiene un nodo raíz ya que es acíclico. Un árbol tiene tres clases de nodos: raíz, interior y hojas.

Se denota con (α_v, β_v) el par asociado con el nodo v ($v \in G_F$). Se calcula $\#SAT(F)$ mientras se recorre G_F en post-order con el siguiente algoritmo.

Algoritmo Conteo_Modelos_para_arbol(G_F)

Entrada: G_F - un grafo que representa un árbol.

Salida: El número de modelos de F

Procedimiento:
Recorrer G_F en post-order, para cada nodo $v \in G_F$, asignar:

1. $(\alpha_v, \beta_v) = (1, 1)$ si v es un nodo hoja en G_F.

2. Si v es un nodo interior con una lista de nodos hijos asociados, i.e., $u_1, u_2, ..., u_k$ son los nodos hijos de v, una vez que se han visitado los hijos, los pares calculados son $(\alpha_{u_j}, \beta_{u_j})$ $j = 1, ..., k$. Sean $e_1 = v^{\epsilon_1}u_1^{\delta_1}, e_2 = v^{\epsilon_2}u_2^{\delta_2}, \ldots, e_k = v^{\epsilon_k}u_k^{\delta_k}$ las aristas que conectan v con cada uno de sus nodos hijos. El par $(\alpha_{e_j}, \beta_{e_j})$ se calcula para cada arista e_j basado en la recurrencia (2.1) donde α_{e_j-1} is α_{u_j} y β_{e_j-1} es β_{u_j} para $j = 1, \ldots k$. Entonces, sea $\alpha_v = \prod_{j=1}^{k} \alpha_{e_j}$ y $\beta_v = \prod_{j=1}^{k} \beta_{e_j}$. Note que este paso incluye el caso en que v tiene solo un nodo como hijo.

3. Si v es el nodo raíz de G_F entonces regresar $(\alpha_v + \beta_v)$.

Este procedimiento regresa el número de modelos de F en tiempo $O(n+m)$ [16] el cual es el tiempo para recorrer un grafo en post-order.

Ejemplo 3 *Si* $F = \{(x_1, x_2), (x_2, x_3), (x_2, x_4), (x_2, x_5), (x_4, x_6), (x_6, x_7), (x_6, x_8)\}$ *es una fórmula en 2-CF, si* x_1 *es el nodo raíz, después de realizar un recorrido post-order el número de modelos a cada nivel del árbol se muestran en la Figura 2.1. El procedimiento Conteo_Modelos_para_arbol regresa* $\alpha_{x_1} = 41$, $\beta_{x_1} = 36$ *y el número total de modelos es:* $\#SAT(F) = 41 + 36 = 77$.

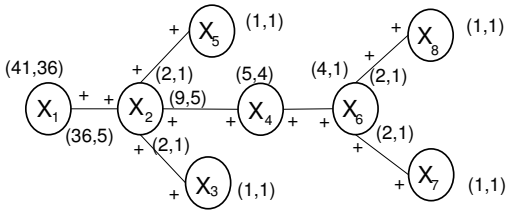


Figura 2.1: Conteo de modelos en grafos que representan árboles

2.2. Conteo de Modelos en Grafos Cíclicos

El problema #2SAT es #P-completo, para fórmulas cuyo grafo restringido es cíclico. En [15] se presenta un algoritmo para el conteo de modelos basado en el números de aristas del co-árbol. Por cada arista del co-árbol el algoritmo duplica el grafo de entrada eliminando la arista del co-árbol. Por lo tanto la complejidad del algoritmo esta dada por 2^r donde r es el número de aristas del co-árbol. Sin embargo, ya que las aristas del co-árbol son aquellas que forman cíclos, un grafo con r cíclos requiere 2^r árboles.

En esta sección mostramos cómo se puede disminuir el número de árboles generados para así hacer más eficiente el procedimiento para cierto tipo de grafos. Los árboles de expansión que se mencionan a continuación se construyen utilizando el método Depth First Search (DFS). El método DFS garantiza que todas las aristas del co-árbol son de retroceso es decir, que si se incorporan al árbol conectan nodos descendientes con nodos ancestros y

no nodos que están en diferentes ramas del árbol. Por tal motivo se puede dividir el grafo original en r sub-árboles cada uno con su respectivo co-árbol, donde r es el número de nodos hijos asociados al nodo que se eligió como raíz en el grafo original.

Definición 5 *Sea F una fórmula cuyo grafo restringido $G_F = \rho(F)$ es cíclico. Sea T el árbol de expansión de G_F y $\overline{T}$ su co-árbol; se construye una familia de conjuntos $\mathcal{P}$ de $\overline{T}$ como sigue: un par de aristas $e_1, e_2 \in \overline{T}$ pertenecen al mismo conjunto sí y sólo si $path(e_1) \cap path(e_2) \neq \emptyset$ donde $path(e_i), i = 1, 2$ es el camino de los vértices de las aristas e_1 y e_2 en el árbol T.*

Lema 1 *La familia de conjuntos $\mathcal{P}$ de la Definición 5 es una partición de $\overline{T}$.*

***Prueba* 1** *Sean $X, Y \in \mathcal{P}$, por definición $X \cap Y = \emptyset$ ya que para cada par de aristas $e_1, e_2 \in \overline{T}$ hay un único camino en T. Dado que cada arista $e \in \overline{T}$ pertenece a una única partición entonces $\bigcup_{X \in \mathcal{P}} X = \overline{T}$*

Ahora se puede construir una partición de G_F.

Definición 6 *t*

1. *Para cada $P \in \mathcal{P}$, se construye un subgrafo como sigue: $\forall e \in P$,*

$$G_P(V(path(e)), E(path(e) \cup e))$$

2. *Se define $G_R = G_F \setminus \bigcup G_{P \in \mathcal{P}}$.*

Lema 2 *Los conjuntos $E(G_P), P \in \mathcal{P}$ junto con $E(G_R)$ forman una partición de $E(G_F)$.*

Lema 3 *Para cada par de grafos G_{P_1}, G_{P_2}, del Lema 2, ya sea que $V(G_{P_1}) \cap V(G_{P_2}) = \emptyset$ o $V(G_{P_1}) \cap V(G_{P_2}) = \{v\}$ es decir es vacío o un conjunto de un solo elemento.*

***Prueba* 2** *Por contradicción, supóngase que $V(G_{P_1}) \cap V(G_{P_2}) \neq \emptyset$ y $V(G_{P_1}) \cap V(G_{P_2}) \neq \{v\}$ lo que significa que hay al menos dos vértices v_1, v_2 en la intersección, lo que significa que la arista $e = (v_1, v_2)$ pertenece a la intersección contradiciendo la hipótesis de que G_{P_1} and G_{P_2} tienen un conjunto de aristas disjuntas.*

Teorema 1 *Para cada par de grafos G_{P_1}, G_{P_2} del lema 2*

1. *Si $G_{P_1} \cap G_{P_2} = \emptyset$ entonces los modelos que representa G_{P_1} son independientes de los modelos que representa G_{P_2} es decir $Modelos(G_{P_1} \cup G_{P_2}) = Modelos(G_{P_1}) \times Modelos(G_{P_2})$.*
2. *Si $G_{P_1} \cap G_{P_2} = \{v\}$ entonces*

$$\begin{aligned} Modelos(G_{P_1} \cup G_{P_2}) = Modelos(G_{P_1}|_{v^1}) &\times Modelos(G_{P_2}|_{v^1}) \\ &+ \\ Modelos(G_{P_1}|_{v^0}) &\times Modelos(G_{P_2}|_{v^0}) \end{aligned}$$

***Prueba* 3** *1. Si $G_{P_1} \cap G_{P_2} = \emptyset$ ninguno de los vértices o aristas son compartidos. Ya que cada vértice de los grafos representan una variable de la fórmula de entrada, $\rho^{-1}(G_{P_1}) \cap \rho^{-1}(G_{P_2}) = \emptyset$, es decir no hay variables en común de las fórmulas representadas por cada sub-grafo. Es bien sabido que los modelos de fórmulas sin variables en común se pueden calcular como el producto de los modelos de cada fórmula.*

2. Que la intersección sea un conjunto con un solo elemento significa que si $F_1 = \rho^{-1}(G_{P_1})$ y $F_2 = \rho^{-1}(G_{P_2})$ entonces $\nu(F_1) \cap \nu(F_2) = \{x_1\}$, es decir hay una sola variable en común para ambas sub-fórmulas. Una estrategia branch and bound se puede utilizar, donde una rama cuenta los modelos donde x_1 se fija con el valor verdadero y la otra rama cuenta los modelos donde x_1 se fija con el valor falso en ambas sub-fórmulas y al final se suman los modelos. Una vez que x_1 se fijo con un valor ya sea verdadero o falso, no existen mas variables en común entre las sub-fórmulas , así que por 1, sus modelos se pueden calcular de forma independiente y posteriormente realizar el producto.

Del Teorema 1 se puede concluir que el número de modelos de cada uno de los $G_{P \in \mathcal{P}}$ se puede calcular de forma independiente.

Dado que G_F es conectado, hay sub-grafos G_{P_1} y G_{P_2} tal que $V(G_{P_1}) \cap V(G_{P_1}) = \emptyset$ por lo que debe existir un camino en G_R que los una. Afortunadamente, los modelos de un camino se pueden calcular en tiempo polinomial con el procedimiento que se presentó en la sección 2.1.1.

Así, se concluye que

Teorema 2 *Sea F una fórmula en 2-CNF, sea $G = \rho(F)$ su grafo restringido. Si P es una partición de G como se estableció en la Definición 5 y $\overline{T_i}, i = 1, \ldots r$ son las particiones que contienen aristas en el co-árbol, entonces la complejidad en tiempo para calcular $Modelos(G)$ es $O(2^{max\{|E(\overline{T_i})|\}} \cdot poly(|E(T)|))$, donde poly es una función polinomial.*

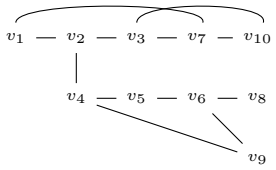
***Prueba* 4** *Por el Teorema 1 cada $\overline{T_i}$ se puede calcular de forma independiente. Así la duplicación máxima del grafo esta dado por $k = max\{|\ E(\overline{T_i})|\}$.*

Ejemplo

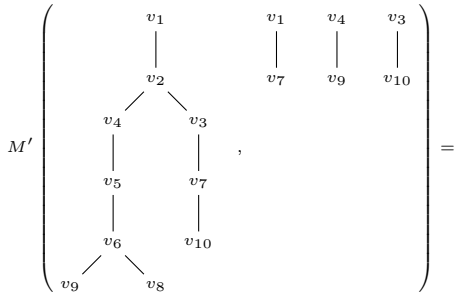
Sea

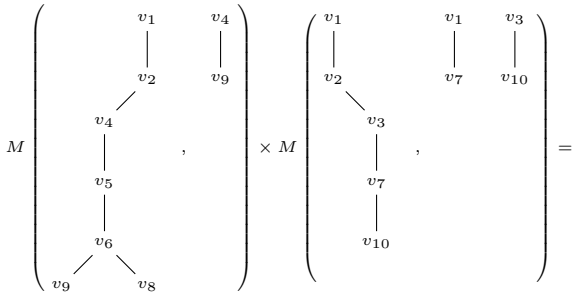
$$F = \{\{v_1, v_2\}, \{v_1, v_7\}, \{v_2, v_4\}, \{v_2, v_3\}, \{v_3, v_7\}, \{v_3, v_{10}\}, \{v_4, v_5\}, \{v_4, v_9\}, \{v_5, v_6\}, \{v_7, v_{10}\}, \{v_6, v_8\}, \{v_6, v_9\}\}.$$

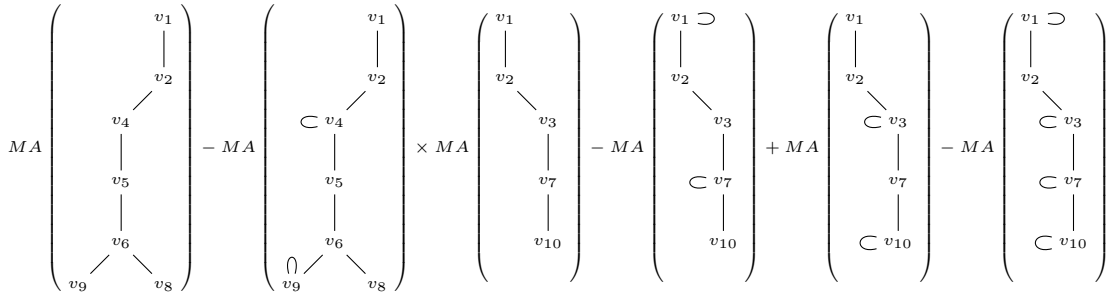
El grafo restringido $\rho(F)$ es (se omiten los signos ya que todos son positivos):



La aplicación de DFS del grafo restringido da la siguiente descomposición en árbol y co-árbol donde la artista del co-árbol (v_4, v_9) pertenece a una de las particiones y las aristas del co-árbol $(v_1, v_7), (v_3, v_{10})$ pertenecen a la otra partición. Así sus modelos se pueden calcular de forma independiente como se muestra abajo.







Capítulo 3

Desarrollo de la implementación

3.1. Herramientas utilizadas

.

Para poder realizar el procesamiento del archivo de entrada se decidió utilizar las herramientas *flex* y *bison* para poder evaluar fórmulas en 2-CNF expresadas de la forma $(x_i \vee x_{i+1}) \wedge (x_{i+1} \vee x_{i+2}) \wedge \ldots \wedge (x_{k-1}, x_k)$. Para las fórmulas que están representadas en archivos en formato *dimacs* se realiza el procesamiento con un programa (parser) que identifica sólo cláusulas de dos variables.

Para realizar la programación del algoritmo se decidió utilizar el lenguaje C para poder utilizar de manera eficiente las características del algoritmo, la especificación de las estructuras que se usarán, reducir al mínimo el uso de memoria de la computadora y la integración con librerías para manejo de números muy grandes como *GMP*, así como herramientas para manejar los archivos de entrada.

3.2. Diagrama general del sistema

Para poder realizar el método de conteo de modelos se realizaron cinco fases generales:

1. Transformar la fórmula de entrada en un grafo
2. Construcción del árbol y coárbol a partir del grafo
3. Evaluar cláusulas paralelas y unitarias dentro del árbol
4. Identificar las particiones en el árbol
5. Evaluación de las particiones en el árbol

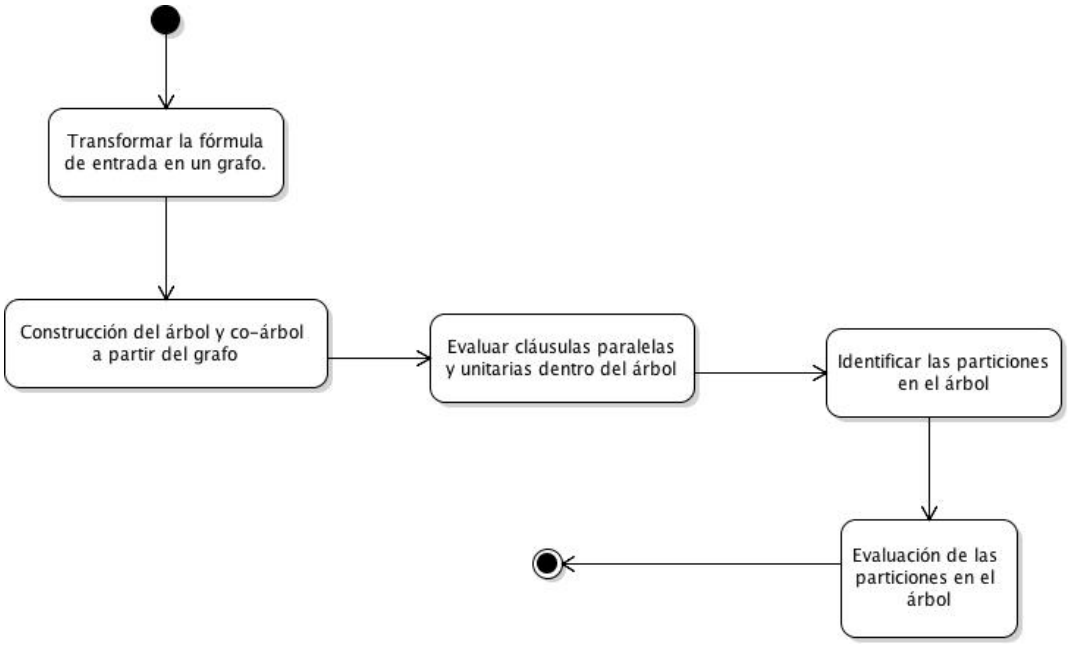


Figura 3.1: Diagrama general del sistema

3.3. Conversión de una fórmula en un grafo.

Para poder convertir una fórmula dada en 2-Forma Normal Conjuntiva en un grafo se utiliza la herramienta *flex* para poder identificar a las variables dentro de la fórmula, las cuales pueden contener letras y números, para identificar los conectores $\wedge$ y $\vee$ se utilizan Y y O respectivamente, para reconocer una variable en su forma negada el símbolo que se busca es $\neg$.

El código que se utiliza se muestra a continuación:

```
 DIGITO [0-9]
ENTERO {DIGITO}+

LETRA [a-zA-Z]
ID ({ENTERO}|{LETRA})+


%%

"O"         {return tOR;}
"Y"         {return tAND;}

{ID}    {yylval.name=yytext;ide=strlen(yytext);return tID;}

"¬"         {return tNEG;}

"("         {return tPA;}
")"         {return tPC;}
```

Para poder reconocer la fórmula y decidir que accción tomar se utiliza *bison*, para hacerlo se requiere poder representar mediante una gramática libre de contexto lo que se quiere hacer.

El código que se utiliza se muestra a continuación:

```
%union
{
    char    * name;
    int     val;
}
```

```
%token tNEG tOR tAND tPA tPC tID

%type<name> tID


%%

enunciado: enunlist      {if(ordenaARB()==1){while(tabla!=NULL){if(
ordenaARB()==0)break;}COMPARB *c=NULL;c=todo;while(c!=NULL){printf
("%d\n",conteo);if(c->co!=NULL){}c=c->next;conteo++;}}duplicaTodo
();}
         ;

enunlist: enunlist tAND enun
         |enun
         ;

enun: tPA act {izq=contNeg;contNeg=0;} tID {strncpy(varname1,$4,id
e);varname1[ide]='\0';} tOR act {der=contNeg;contNeg=0;} tID {strn
cpy(varname2,$9,ide);varname2[ide]='\0';} tPC {addTAB(izq,der,tabl
a,varname1,varname2);}
         ;

act: act tNEG    {if(contNeg==0){contNeg=1;}else{contNeg=0;}}
         |
         ;

%%
```

3.4. Representación en el lenguaje

En el siguiente ejemplo visto en la sección 2.1.3 se observa que es necesario tener control sobre el conteo de modelos en las aristas del grafo, así como una forma de saber si las variables que se relacionan por medio de las aristas se encuentran en su forma normal o negada, además es requerido tener un conteo en cada nodo del mismo. Para el conteo de modelos es necesario tener dos campos α y β y para los "signos" de las variables dos campos que tendrán un 0 o 1 asociado.

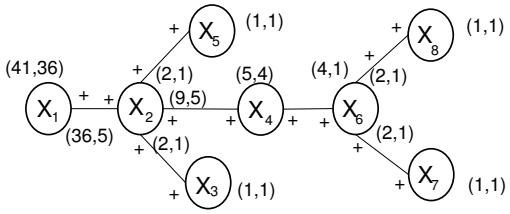


Figura 3.2: Conteo de modelos en grafos que representan árboles

Para representar un vértice o nodo en el árbol que se va a generar se utiliza una estructura que contiene ocho campos, un nombre, la dirección de la lista de conteo del vértice (la cuál contiene todas las aristas que conectan a un nodo con sus hijos), la dirección de la lista de conteo del vértice padre (la arista que conecta a un vértice con su vértice padre), un campo para el conteo α , un campo para el conteo β , dos campos para identificar cláusulas unitarias además de servir para desarrollar el método de conteo.

Las aristas del árbol se representan por medio de una lista de conteo para cada vértice, un elemento de una lista de conteo conecta a dos vértices o nodos del árbol y son usadas como un conteo temporal de modelos que siguiendo el algoritmo el resultado final se refleja en el nodo al que pertenece la lista de conteo, los campos que conforman a la lista son: la dirección del vértice padre, la dirección del vértice hijo, los indicadores para conocer si una variable está en su forma normal o negada uno para el vértice padre y otro para el vértice hijo, los campos de conteo temporal α y β, la dirección del siguiente elemento de la lista de conteo y por último un indicador utilizado para la evaluación de las particiones.

```
//nodos del arbol
typedef struct nodo {
        char nombre[30];
        struct link *listaConteo;
        struct link *listaSuperior; //
        mpz_t a,b; //los campos a = alfa b = beta
        int unit_a, unit_b; //indicadores para loops
        } ;
```

```
//aristas del arbol
typedef struct link {
        struct nodo *padre; // nodo padre
        struct nodo *hijo; // nodo hijo
        int t_padre,t_hijo; // indicadores de forma normal o
        // negada para padre e hijo respectivamente
        mpz_t a,b; // los campos para alfa y beta temporales
        struct link *siguiente; // siguiente elemento de la lista
        //de conteo
        int no_modifica; // indicador para marcar como
        //no modificable
        } LINK;
```

3.5. Construcción del árbol y coárbol a partir del grafo.

Teniendo el grafo representado mediante una lista de relaciones G de la forma $g_i = (x, y, s_1, s_2)$ cada una de las cuales tiene dos variables (x, y) y un indicador para saber si esta en su forma normal o negada, para cada una (s_1, s_2), se aplica el método de depth first search para construir el árbol y el coárbol como sigue:

1. Para cada elemento g_i de G.
2. Si A está vacío, el elemento x_i de g_i se convierte en la raíz del árbol, en otro caso se agrega x_i como hijo del último nodo que se evaluó en el árbol.
3. Si y_i existe en el árbol y no es x_i el elemento g_i se agrega al coárbol en otro caso si $x_i = y_i$ se evalúa esta cláusula como unitaria si y_i no existe en el árbol se agrega como hijo de x_i asignando los indicadores (s_{i1}, s_{i2}).
4. Se toma un elemento g_i de G que contenga a la variable y_i en el lugar de x_i (y_i, y_i') y se procede al paso 3, si no existe un elemento que contenga a y_i se procede a evaluar el nodo utilizando la recurrencia vista en 2.1 desde y_i hasta la raíz del árbol para propagar el resultado, después se procede a buscar un elemento g_i que contenga a la variable x_i (x_i, y_i') y se procede al paso 3.
5. Si $G \neq \emptyset$ y no hay algún elemento g_i que se pueda añadir al árbol se procede a generar un árbol y coárbol nuevos y regresar al paso 1.

Con el procedimiento anterior es posible evaluar fórmulas que representen grafos desconectados y asegura que al terminar de evaluar un árbol y éste no tenga un coárbol no sea

necesario recorrerlo nuevamente para obtener el número de modelos, la raíz del árbol ya tendrá el número de modelos resultante.

Para realizar los procedimientos de evaluación se renombran todos los nodos del árbol recorriéndolo primero en profundidad y comenzando en 1 .

3.6. Evaluación de cláusulas paralelas.

Teniendo el árbol y coárbol la evaluación de cláusulas paralelas se lleva a cabo buscando todos los elementos del coárbol que involucren a las mismas variables y se aplican las siguientes reglas:

- Si dos cláusulas involucran las mismas variables se usa la recurrencia 2.2.
- Si tres cláusulas involucran las mismas variables se usa la recurrencia 2.3.
- En el caso de cuatro cláusulas el conteo es 0.

Para poder realizar lo anterior se realizan los siguientes pasos:

1. Para cada nodo del árbol n_i.
   1.1 Si existe más de un elemento en el coárbol que conecte a n_i a otro nodo n_{i-1} se aplican las reglas anteriores.
   1.2 Se evalúa cada nodo desde n_i a la raíz aplicando la recurrencia 2.1 para reflejar el cambio en el nodo raíz.

3.7. Evaluación de cláusulas unitarias

Cuando existe una cláusula unitaria y la variable que aparece en ella está en su forma normal x_i o negada $\bar{x_i}$ entonces se aplica la recurrencia 2.4, posterior mente se evalúa cada nodo desde x_i hasta la raíz del árbol aplicando la recurrencia 2.1 para reflejar el cambio en el nodo raíz.

3.8. Identificar las particiones en el árbol

Para realizar este procedimiento de manera eficiente se necesita que los nodos del árbol estén nombrados con números de la manera que se especificó en la sección 3.5 y por otro lado teniendo el coárbol $\bar{T}$ ordenado de manera descendente de tal forma que teniendo dos

elementos $t_i = (x_i, y_i, s_{i_1}, s_{i_2})$ y $t_{i+1} = (x_{i+1}, y_{i+1}, s_{i+1_1}, s_{i+1_2})$ se cumpla que $y_i \geq y_{i+1}$ y se procede como sigue:

1. Se comienza con una nueva partición p_i.
2. Para cada elemento t_i del coárbol $\bar{T}$.
3. Se genera el camino r que se sigue desde x_i a y_i.
4. Teniendo p un conjunto de nodos asociados N_i, si $|r \cap N_i| \geq 2$ se realiza la unión $N_i = r \cup N_i$ y se agrega t_i al conjunto T_i de p_i, si .

3.9. Evaluación de las particiones en el árbol

Al tener identificado un conjunto de particiones P y al haber asegurado que los elementos del coárbol se encontrarán ordenados de manera descendente las particiones también lo estarán lo que permite que al resolver una por una y sustituir el número de modelos que se obtiene de cada una en el árbol no se altere el resultado al final de la evaluación, permitiendo reducir el tamaño del árbol en cada iteración.

1. Para cada p_i de P.
2. Se evalúa siguiendo el procedimiento en 2.2 con T_i como el coárbol y tomando como raíz el elemento con el valor más pequeño de N_i, se sustituyen los valores obtenidos mediante la evaluación en el nodo raíz de la partición p_i. .
3. Se evalúa de la raíz de p_i, usando la recurrencia 2.1, hasta la raíz del árbol para reflejar el cambio realizado.

Capítulo 4

Resultados

4.1. Enfoque de la herramienta

En el Capítulo 2 se mostró que la complejidad del algoritmo propuesto esta dado en términos de grupos de ciclos, por lo tanto el tipo de grafos que se resuelven de manera eficiente son aquellos que tengan agrupaciones de ciclos. Entre los grafos de este tipo se tiene los dispersos y Cactus.

4.2. Grafos Dispersos

Son aquellos grafos dónde el número de aristas es cercano al número de vértices, lo que se pretende es utilizar grafos dónde existan grupos de ciclos dispersos, sin que el número de aristas llegue al máximo posible.

Sea $G = (V, E)$ un grafo simple con $n = |V|$, $m = |E|$. La densidad de un grafo se define con la siguiente fórmula:

$$d = \frac{2m}{n(n-1)}$$

Si d es cercano a 0 entonces se dice que G es un grafo disperso, si es igual a 0 entonces es un grafo dónde los vértices son aislados. Si d es cercano a 1 entonces G es un grafo denso, si es igual a 1 es un grafo completo.

Para poder realizar pruebas con este tipo de grafos se decidió utilizar grafos como los de la Figura 4.1, en la que se representan particiones en las ramas del árbol con $0 \leq d \leq 1$.

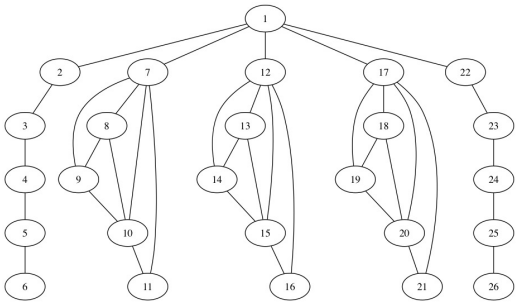


Figura 4.1: Grafo Sparse

4.2.1. ¿Cómo generar las instancias de prueba para los grafos dispersos?

Para poder generar grafos dónde los ciclos que contienen estuvieran divididos en particiones se generó un programa que dado el número de nodos n que tenga el grafo y el número de ciclos m que se quiere en cada partición, genera un grafo dónde a partir de la raíz se crean ramas con el número de nodos k necesarios para asegurar que el número de ciclos que se requiere se cumpla, dada la siguiente fórmula:

$$k = \frac{1+\sqrt{1+8m}}{2} + 1$$

Para hacer una corrección en k en caso de que sea un número con decimales se realiza un redondeo hacia arriba, ya que indica que se necesita al menos otro nodo para poder tener el número de ciclos indicados por m.

Para obtener el número de particiones p que tendrá el grafo resultante se utiliza la siguiente fórmula:

$$p = \frac{(n-1)}{k}$$

El número de aristas a que tendrá el grafo se obtiene mediante la siguiente fórmula:

$$a = p(k + m)$$

Para realizar la creación del grafo se muestra en el ejemplo 4 el grafo obtenido.

Ejemplo 4 *Dado un grafo de 16 nodos se requiere que cada partición tenga 4 ciclos por lo que el número de nodos necesarios en cada partición es 5 y el número de particiones que se pueden generar es 3 con un número de aristas de 27.*

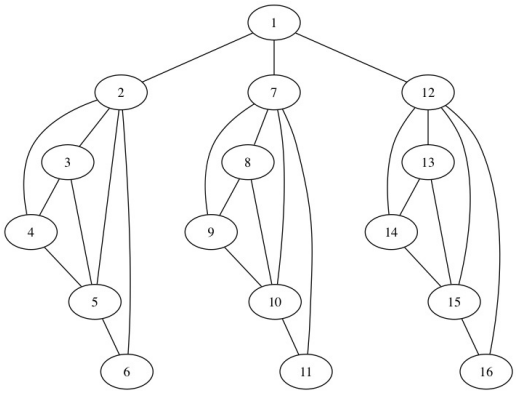


Figura 4.2: Grafo Sparse con n=16, m=4, p=3 y a=27

4.2.2. Pruebas en grafos con $densidad = 0$

En el cuadro 4.1 se muestran resultados obtenidos en grafos acíclicos.

Nodos	Aristas	Modelos ≈
15999	15998	3111×10^{3814}
19999	19998	5437×10^{4768}
25999	25998	1256×10^{6200}
29999	29998	2196×10^{7154}

Tabla 4.1: Pruebas en grafos sin ciclos

La columna *Nodos* muestra el número de variables, la columna Aristas muestra el número de cláusulas.

4.2.3. Pruebas en grafos con $0 < densidad < 1$

En el cuadro 4.2 se muestran resultados obtenidos en grafos en los que se introdujeron un menor número de ciclos en cada partición.

Nodos	Aristas	Ciclos	Ciclos por partición	Tiempo(s)
15996	23994	7998	2	28
15995	31990	15995	5	60
15996	42656	26660	10	159
19996	29994	9998	2	49
19995	39990	19995	5	115
19998	53328	33330	10	269
25996	38994	12998	2	118
25995	51990	25995	5	276
25998	69328	43330	10	469
29996	44994	14998	2	170
29995	59990	29995	5	362
29998	79984	49990	10	663

Tabla 4.2: Pruebas en grafos con $0 < densidad < 1$

La columna ciclos muestra el total de ciclos que se introdujeron en el árbol, la columna Ciclos por partición muestra el número de ciclos que se introdujeron en cada rama del árbol y la columna de Tiempo(s) muestra el tiempo que tarda la herramienta en resolver cada instancia con respecto al número de ciclos que tenga.

En el cuadro 4.2 se muestran resultados obtenidos en grafos en los que se introdujeron 2, 5 y 10 ciclos por partición, para poder saber cuál era el límite de la herramienta se utilizaron grafos con varios ciclos, éste límite está dado por la cantidad de memoria física necesaria para realizar el conteo de modelos.

4.3. Grafos Cactus

Un grafo $G = (V_G, E_G)$ es un cactus si:

- Cada arista de E_G pertenece a lo más a un ciclo.
- Cualquier par de ciclos comparten a lo más un vértice.

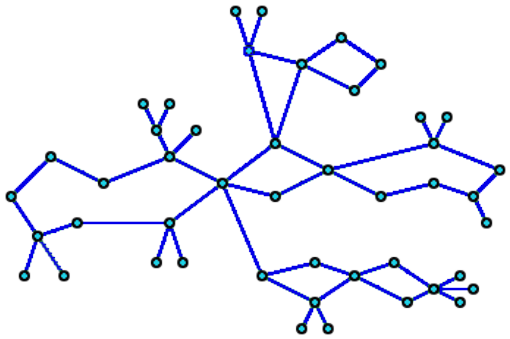


Figura 4.3: Grafo Cactus

4.3.1. ¿Cómo generar las instancias de prueba para los grafos Cactus?

Para poder generar instancias de prueba se utiliza un programa que realiza la conexión de variables creando grupos de cuatro (Figura 4.4) y uniéndolos por medio de una de las variables que componen a cada grupo, cumpliendo que dos grupos diferentes comparten a lo más una variable entre ellos.

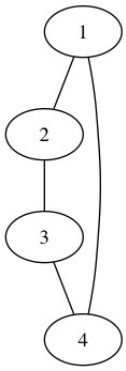


Figura 4.4: Parte de los grafos cactus generados

Para este tipo de grafos sólo se pretende manipular el número de nodos n que tendrá el grafo, la forma en cómo se conectan los grupos es simple:

1. Se genera el primer grupo de variables $\{x_i, x_{i+1}, x_{i+2}, x_{i+3}\}$ y se agregan a una lista L.
2. Si no se ha llegado al límite de variables se procede a generar otro grupo de tres variables $\{x_j, x_{j+1}, x_{j+2}\}$.
3. Se toma el primer elemento de L y se conecta con las tres variables generadas en el paso anterior.
4. Se procede a realizar el paso 2 hasta que se haya alcanzado el número máximo de variables n.

4.3.2. Pruebas en Grafos Cactus

El cuadro 4.3 muestra los tiempos obtenidos en grafos cactus generados por el método visto en la sección 4.3.1 dónde se incrementó el número de variables en cada ejemplo hasta llegar al límite de la herrramienta.

Nodos	Aristas	Tiempo(s)
9382	12508	6
10000	13332	8
16000	21332	19
19999	26664	37
25999	34664	61
30001	40000	118
36001	48000	175
40000	53332	225
46000	61332	302
49999	66664	389
55999	74664	445
60001	80000	549
66001	88000	622

Tabla 4.3: Pruebas de la herramienta sharpSAT en los grafos sparse y cactus

4.4. Comparación con otras herramientas

Se probaron los grafos presentados en los cuadros 4.1,4.2,4.3 en las herramientas sharpSAT y relsat ya que son las más recientes y aplican las características de *cachet*.

Nodos	Cláusulas	Ciclos	Sharp SAT	relsat	Esta herramienta
			Tiempo (s)	Tiempo (s)	Tiempo(s)
15999	15998	0	5.3	60	0
19999	19998	0	8.3	94	0
25999	25998	0	14.2	153	0
29999	29998	0	19.2	201	0
15997	23994	7998	4	47	28
15996	31990	15995	3.3	55	49
15997	42656	26660	3	55	128
19997	29994	9998	6.2	78	42
19996	39990	19995	5	90	79
19999	53328	33330	4.6	92	210
25997	38994	12998	10.1	130	81
25996	51990	25995	8.3	158	148
25999	69328	43330	7.5	149	340
29997	44994	14998	13	182	123
29996	59990	29995	11	211	230
29995	79984	49990	9.9	203	600
9382	12508	3126	0.5	3	6
19999	26664	6665	1.4	18	27
10000	13332	3332	0.6	4	5
16000	21332	5332	1	10	16
25999	34664	8665	2	33	49
30001	40000	9999	2.4	40	69
36001	48000	11999	3.1	61	104
40000	53332	13332	3.5	82	133
46000	61332	15332	4.2	107	184
49999	66664	16665	4.7	125	219
55999	74664	18665	5.5	169	280
60001	80000	19999	6.1	194	337
66001	88000	21999	7	264	398

Tabla 4.4: Pruebas en grafos Cactus

De las 29 pruebas realizadas solo en 4 de ellas la herramienta desarrollada en este trabajo

Nodos	Cláusulas	Ciclos	Sharp SAT		Esta herramienta	
			Modelos	Tiempo (s)	Modelos	Tiempo (s)
13	15	3	401	0	401	0
19	18	0	15660	0	15660	0
19	33	15	3512	0.1	3512	19
19	34	15	-	-	5520	18
37	52	15	-	-	28058400	18
37	66	30	-	-	6294080	37
60	59	0	1.33×10^{13}	0	1.33×10^{13}	0
70	69	0	1.85×10^{16}	0	1.85×10^{16}	0
85	84	0	4.38×10^{19}	0	4.38×10^{19}	0

Tabla 4.5: Pruebas del algoritmo general en grafos que contienen un número pequeño de ciclos

mejora el tiempo de ejecución en comparación a *sharpSAT*. Por lo tanto, se considera que *sharpSAT* tiene una mejor estrategia para el conteo de modelos en grafos que tienen ciclos anidados.

En el caso de *relsat* en los casos en que el número de ciclos es menor al número de nodos se mejoro el tiempo de ejecución, la estrategia de *cache de componentes* muestra una mejoría importante en los últimos trece casos de la Tabla 4.4.

En la tabla 4.5 se generaron instancias pequeñas en las cuales la herramienta sharpSAT no pudo regresar un resultado.

Como se puede apreciar, en tres instancias sharpSAT no pudo regresar el número de modelos. Por lo tanto se envio un correo al autor de la herramienta pero no hubo respuesta al respecto.

Capítulo 5

Conclusiones

Aunque el problema #SAT es en general #P-completo, existen diferentes instancias que se pueden resolver de forma eficiente. Si el grafo restringido de la entrada tiene ciclos, se muestra un procedimiento que permite calcular el número de modelos con complejidad del orden $O(2^{max\{|E(\overline{T_i})|\}} \cdot poly(|E(T)|))$, donde $poly$ es una función polinomial y los $T_i, \overline{T_i}$ son los árboles y co-árboles respectivamente de la descomposición del grafo original . Este es un método que muestra que ciertas instancias se pueden resolver de manera eficiente.

Aunque no se pudieron resolver de manera "eficiente" la mayoría de los grafos presentados con respecto a sharpSAT, el método probó ser eficiente en casos especiales dónde ésta herramienta si puede regresar un resultado y sharpSAT no.

Con respecto a *relsat* los tiempos de ejecución fueron cercanos a los de ésta herramienta, superándola en algunos casos, lo que indica que es posible mejorar su eficiencia y llegar al tiempo de ejecución de *sharpSAT*.

Se utilizó el tiempo de ejecución de las herramientas como comparativo porque *sharpSAT* y *relsat* no mencionan cual es la complejidad del algoritmo que utilizan.

Como trabajo a futuro se contemplan las siguientes vertientes.

- Actualmente se tiene un método para el conteo de modelos en grafos cactus sin tener que duplicar los subgrafos que contiene ciclos, es decir, el método para el conteo de modelos en grafos cactus es polinómico. Con esta estrategía, se considera se puede igual el tiempo de sharpSAT.
- Para grafos dispersos, existen estrategias que permiten disminuir el número de ciclos de la fórmula de entrada, entre ellos se encuentran la aplicación de resolución unitaria y eliminación de literales puras, por lo que la aplicación de estos procedimientos puede disminuir el tiempo de ejecución.
- Existen procedimientos para descomponer una fórmula en subfórmulas reducidas en

las que se limita la aparición de cada variable a 5 o menos y posteriormente se reduce su aparición a 3 por cada subfórmula.

- Se han implementado estrategias de cache de componentes que han probado ser eficientes en fórmulas grandes, consiste en evaluar subfórmulas y agregarlas a un cache, al evaluar una nueva subfórmula si ésta ya fue evaluada, se toma el número de modelos que se calculó sin tener que volver a calcular.

Bibliografía

[1] Bondy J. A. and Murty U.S.R.: Graph Theory. Springer Verlag, Graduate Texts in Mathematics (2010).

[2] Darwiche A., On the Tractability of Counting Theory Models and its Application to Belief Revision and Truth Maintenance, Jour. of Applied Non-classical Logics, , pp. 11-34, (2001).

[3] Dahllöf V., Jonsson P., Wahlström M., Counting models for 2SAT and 3SAT formulae., Theoretical Computer Sciences, 332(1-3), pp. 265-291, (2005).

[4] Khardon R., Roth D., Reasoning with Models, Artificial Intelligence, Vol. 87, No. 1, pp. 187-213, (1996).

[5] Roth D., On the hardness of approximate reasoning, Artificial Intelligence 82, pp. 273-302, (1996).

[6] Russ B., Randomized Algorithms: Approximation, Generation, and Counting, Distinguished dissertations Springer, (2001).

[7] Vadhan Salil P., The Complexity of Counting in Sparse, Regular, and Planar Graphs, SIAM Journal on Computing, Vol. 31, No.2, pp. 398-427, (2001).

[8] Bubley R.: Randomized Algorithms: Approximation, Generation, and Counting. Distinguished dissertations Springer (2001).

[9] Thurley, M., sharpSAT - Counting Models with Advanced Component Caching and Implicit BCP. Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT 2006), pp. 424-429, 2006.

[10] Sang, T., Fahiem Bacchus, Paul Beame, Henry Kautz, and Toniann Pitassi. Combining Component Caching and Clause Learning for Effective Model Counting. Seventh International Conference on Theory and Applications of Satisfiability Testing, Vancouver, Canada, 2004.

[11] Brightwell, G. R.; Winkler, Peter (1991). Çounting linear extensions". Order 8: 225-242.

[12] Marcial-Romero J. R., De Ita G., Hernández, J. A., Valdovinos, R. M. A Parametric Polynomial Deterministic Algorithm for #2SAT, to appear in Lecture Notes in Computer Science, 2015.

[13] Bayardo, R. and R.Schrag, Using CSP look-back techniques to solve real-world SAT instances. In: Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97). New Providence, RI, pp. 203-208.

[14] Satlib. www.satlib.org, consultada el 17 de Noviembre del 2015.

[15] De Ita G., Marcial-Romero J.R., Mayao Y. An Enumerative Algorithm for #2SAT, Electronic Notes in Discrete Mathematics, Volume 46, pp 81–88, 2015.

[16] Levit V.E., Mandrescu E., The independence polynomial of a graph - a survey, To appear, Holon Academic Inst. of Technology, 2005.