Universidad Auto´noma del Estado de Me´xico

Facultad de Ingenier´ıa

Un algoritmo para el conteo de modelos de fo´rmulas booleanas en 2-Forma Normal Conjuntiva

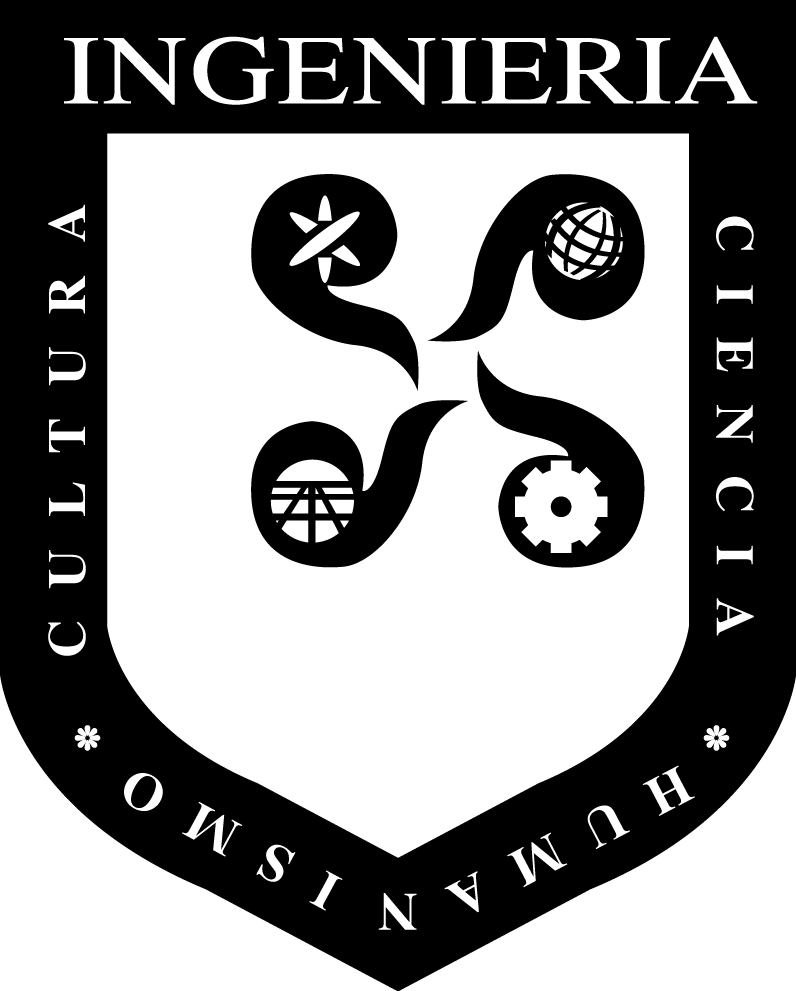
TESIS

QUE PARA OBTENER EL T´ITULO DE:

### Ingeniero en Computacio´n

PRESENTA:

### Marco Antonio L´opez Medina

ASESOR:

Dr. J. Raymundo Marcial Romero COASESOR:

Dra. Rosa Mar´ıa Valdovinos Rosas

Toluca, M´exico, Julio 2016

**Resumen.**

En este trabajo se presenta un algoritmo basado en grafos para obtener el nu´mero de asignaciones que pueden hacer verdadera una f´ormula booleana en 2-Forma Normal Conjuntiva.

Se pretende resolver una parte del problema #SAT que es conocer el nu´mero de asignaciones que hacen verdadera una f´ormula booleana que esta dada por disyunciones de conjunciones o expresado en su notacio´n *c* = (*x*1 *∨x*¯2)*∧*(*x*2 *∨x*¯3).

As´ı mismo se pretende comparar este algoritmo con algunos ya existentes y que han sido probados de manera pra´ctica para resolver problemas reales. Dado que los existentes han sido implementados, *sharpSAT* y *relsat*, para re- solver problemas de una base de datos llamada *satlib* tambi´en se realizo´ la implementacio´n del algoritmo presentado ya que en su documentacio´n no pro- porcionan cu´al 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 gen- erados, 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 m´as reciente, dado por su pre-procesamiento de la en- trada y el cache de componentes de la entrada se pudo mejorar el desempen˜o de la segunda herramienta que se tomo´ como referencia en muchos casos y que es al menos la segunda m´as 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*.

´Indice general

1. Marco Te´orico [7](#_bookmark0)
   1. Grafo [7](#_bookmark1)
      1. Subgrafos [9](#_bookmark4)
      2. Bu´squeda primero en profundidad [10](#_bookmark5)
   2. L´ogica [10](#_bookmark6)
      1. ¿Qu´e es una l´ogica? [10](#_bookmark7)
      2. L´ogica proposicional [11](#_bookmark8)
      3. Satisfactibilidad, tautolog´ıa, consecuencia y equivalencia [11](#_bookmark9)
      4. Forma Normal Conjuntiva (CNF) [12](#_bookmark10)
   3. Problemas tratables [12](#_bookmark11)
      1. El grafo restringido de una 2-CNF [14](#_bookmark12)
   4. Formato Dimacs [14](#_bookmark13)

1.5 relsat (1997) [15](#_bookmark14)

1.6 cachet [15](#_bookmark15)

1.7 sharpSAT (2012) [16](#_bookmark16)

* 1. Flex [16](#_bookmark17)
  2. Bison [16](#_bookmark18)
  3. GMP (GNU Multiple Precision arithmetic library) [16](#_bookmark19)

1. Algoritmos para el conteo de modelos para f´ormulas en 2-CNF [17](#_bookmark20)
   1. Conteo de Modelos en Grafos Ac´ıclicos [17](#_bookmark21)
      1. #2SAT para f´ormulas en 2-CNF que representan un camino [17](#_bookmark22)
      2. #2SAT para f´ormulas en 2-CNF cuyo grafo contiene aristas paralelas [18](#_bookmark24)
      3. Conteo en grafos ac´ıclicos [19](#_bookmark28)
   2. Conteo de Modelos en Grafos C´ıclicos [20](#_bookmark29)
2. Desarrollo de la implementaci´on [25](#_bookmark33)
   1. Herramientas utilizadas [25](#_bookmark34)
   2. Diagrama general del sistema [26](#_bookmark35)

vii

* 1. Conversio´n de una f´ormula en un grafo. . . . . . . . . . . . . . . . . . .

|  |  |
| --- | --- |
| . . | [27](#_bookmark36) |
| . . | [29](#_bookmark37) |
| . . | [30](#_bookmark38) |
| . . | [31](#_bookmark42) |
| . . | [31](#_bookmark43) |
| . . | [31](#_bookmark44) |
| . . | [32](#_bookmark45) |
|  | [**33**](#_bookmark46) |
| . . | [33](#_bookmark47) |
| . . | [33](#_bookmark48) |
| . | [34](#_bookmark50) |
| . . | [35](#_bookmark52) |
| . . | [35](#_bookmark54) |
| . . | [36](#_bookmark56) |
| . . | [37](#_bookmark57) |
| . . | [38](#_bookmark60) |
| . . | [39](#_bookmark62) |
|  | [**41**](#_bookmark65) |
|  | [**43**](#_bookmark66) |

* 1. Representaci´on en el lenguaje . . . . . . . . . . . . . . . . . . . . . . . .
  2. Construccio´n del ´arbol y co´arbol a partir del grafo. . . . . . . . . . . . .
  3. Evaluaci´on de cl´ausulas paralelas. . . . . . . . . . . . . . . . . . . . . . .
  4. Evaluaci´on de cl´ausulas unitarias . . . . . . . . . . . . . . . . . . . . . .
  5. Identificar las particiones en el ´arbol . . . . . . . . . . . . . . . . . . . .
  6. Evaluaci´on de las particiones en el ´arbol . . . . . . . . . . . . . . . . . .

### Resultados

4.1 Enfoque de la herramienta . . . . . . . . . . . . . . . . . . . . . . . . . .

4.2 Grafos Dispersos . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

* + 1. ¿C´omo generar las instancias de prueba para los grafos dispersos?
    2. Pruebas en grafos con *densidad* = 0 . . . . . . . . . . . . . . . .
    3. Pruebas en grafos con 0 *< densidad <* 1 . . . . . . . . . . . . . . 4.3 Grafos Cactus . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

4.3.1 ¿C´omo generar las instancias de prueba para los grafos Cactus? .

4.3.2 Pruebas en Grafos Cactus . . . . . . . . . . . . . . . . . . . . . .

4.4 Comparacio´n con otras herramientas . . . . . . . . . . . . . . . . . . . .

### Conclusiones

### Bibliograf´ıa

viii

La l´ogica proposicional es una rama de la l´ogica cl´asica que estudia las variables proposicio- nales o sentencias l´ogicas, sus posibles implicaciones, evaluaciones de verdad y en algunos casos su nivel absoluto de verdad.

Sea *X* = *{x*1*, . . . , xn}* un conjunto de *n* variables booleanas (es decir que pueden tomar u´nicamente dos valores de verdad) [[8](#_bookmark73)]. Una literal es una variable *xi* o la variable negada *x*¯*i* . Una cl´ausula es una disyuncio´n de literales distintas. Una f´ormula booleana en forma normal conjuntiva (CNF, Conjunctive Normal Form) *F* es una conjuncio´n de cl´ausulas. Sea *v*(*Y* ) el conjunto de variables involucradas en el objeto *Y* , donde *Y* puede ser una literal, una

cl´ausula o una f´ormula booleana. Por ejemplo, para la cl´ausula *c* = *{x*1*∨x*¯2*}, v*(*c*) = *{x*1*, α*2*}*

y *Lit*(*c*) = *{x*1*, x*2*, x*¯1*, x*¯2*}* es el conjunto de literales que aparecen en *c*.

Una asignaci´on *s* es un conjunto de literales tomadas de una f´ormula *F* , con la condicio´n de que s´olo una de las literales *xi* o *x*¯*i* pertenece a *s*. Puede tambi´en considerarse como

un conjunto de pares de literales no complementario. Si *xi ∈ s*, siendo *s* una asignaci´on,

entonces s convierte *xi* en verdadero y

*x*¯*i* en falso. Por otro lado si

*x*¯*i ∈ s* entonces s

convierte a *x*¯*i* en verdadero y *xi* en falso. Considerando una cl´ausula *c* y una asignaci´on *s* como un conjunto de literales, se dice que *c* se satisface por *s* s´ı y solo si *c ∩ s ƒ*= *∅* y si para toda *xi ∈ c, x*¯*i ∈ s* entonces *s* falsifica a *c*. Sea *F* una f´ormula booleana en CNF, se dice que

*F* se satisface por la asignaci´on *s* si cada cl´ausula en *F* se satisface por *s*. Por otro lado, se dice que *F* se contradice por *s* si al menos una cl´ausula de *F* se falsifica por *s*. Un modelo de *F* es una asignaci´on para *v*(*F* ) tal que satisface *F* . Se denota como *SAT* (*F* ) al conjunto de modelos de la f´ormula *F* . Dada una f´ormula *F* en CNF, SAT consiste en determinar si *F* tiene un modelo, mientras que #SAT consiste en contar el nu´mero de modelos que tiene *F* sobre *v*(*F* ). Por otro lado, #2SAT denota #SAT para f´ormulas en 2-CNF.

El problema #SAT pertenece a la clase #P completo [[11](#_bookmark76)], por tanto no existe algoritmo efi- ciente que resuelva el problema en general. Existen implementaciones que permiten calcular los modelos de una f´ormula dada como por ejemplo, relsat [[13](#_bookmark78)] la cual es una herramienta basada en el algoritmo DPLL (Davis-Putnam-Logemann-Loveland algorithm) de bu´squeda exhaustiva de asignaciones. La mejora que provee en comparacio´n a otras herramientas es

el poder procesar r´apidamente f´ormulas disjuntas, es decir, que las variables de una f´ormu- la no intervengan en alguna otra. Cachet [[10](#_bookmark75)] es una herramienta basada en el modelo de conteo Caching que es sucesora del ´arbol de bu´squeda de un modelo de conteo DPLL; ajus- tando variables y simplificando la f´ormula, ya que se pueden encontrar sub-fo´rmulas que han aparecido en una rama del ´arbol de bu´squeda. Si esto sucede, no es necesario volver a contar los modelos, s´olo es necesario saber el conteo de la rama de bu´squeda anterior y utilizarlo. Por otro lado sharpSAT [[9](#_bookmark74)] esta´ basada en el modelo caching e implementa un mejor razonamiento en cada nodo que evalu´a. Utiliza una t´ecnica conocida como *look ahead* que permite hacer una prueba sobre alguna variable en la f´ormula para ver si es necesario que sea siempre verdadera o falsa, permitiendo que solo se cuenten los modelos necesarios para el resto de la f´ormula.

En esta tesis, se propone un algoritmo para el conteo de modelos convirtiendo la f´ormula de entrada en un grafo. En [[12](#_bookmark77)] se mostro´ como contar modelos en tiempo polinomial en grafos cuya forma es un camino o un ´arbol, por lo que se tomara´ el procedimiento como referencia para mostrar como contar en otro tipo de grafos. As´ı mismo, se comparara´ con sharpSAT que es la herramienta m´as eficiente ( en t´erminos de tiempo y espacio) para el conteo de modelos .

2

La complejidad computacional estudia la “dificultad” inherente de problemas de importan- cia teo´rica y/o pr´actica. 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 solu- ci´on ´optima. Probar que un problema es #P es importante puesto que permite encontrar un algoritmo para la solucio´n ´optima 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 co- mo aplicaciones estimar el grado de veracidad en teor´ıas proposicionales [[5](#_bookmark70)], generaci´on de explicaciones a preguntas proposicionales, reparacio´n de bases de datos inconsistentes, inferencia bayesiana [[2](#_bookmark67), [3](#_bookmark68), [4](#_bookmark69), [5](#_bookmark70), [6](#_bookmark71), [7](#_bookmark72)] entre otras.

Existen formas de representar una base de conocimiento mediante modelos representati- vos, con esto la verificacio´n de validez de una proposici´on puede reducirse a un problema combinatorio y no a utilizar un m´etodo de razonamiento como los motores de inferencia, lo que permite saber si la base de conocimiento implica a la proposici´on, lo que la hace verdadera.

Las implementaciones actuales como sharpSAT [[9](#_bookmark74)] o relsat [[13](#_bookmark78)] no presentan un an´alisis de complejidad de sus algoritmos, por lo que no se conoce c´omo se comportaran en diferentes instancias. En esta propuesta, se estudiara´ la complejidad computacional del algoritmo para conocer en qu´e instancias es mas eficiente.

Para desarrollar esta propuesta se realizara´ una transformacio´n de la f´ormula booleana en un grafo el cu´al sera´ descompuesto en ´arbol y co´arbol, que sera´ la entrada al algorit- mo.

Existe un algoritmo basado en la t´ecnica de descomposici´on ´arbol-co´arbol de la f´ormula de entrada que pueda resolver algunas instancias de la base de datos de satlib [[14](#_bookmark79)] para el problema #2SAT de forma eficiente comparado a las implementaciones sharpSAT [[9](#_bookmark74)] y relsat [[13](#_bookmark78)].

Objetivo general

Desarrollar un algoritmo para el conteo de modelos de f´ormulas booleanas en 2-forma normal conjuntiva, basa´ndose en su representacio´n mediante grafo y su descomposici´on en

´arbol-co´arbol.

Cap´ıtulo 1

Marco Teo´rico

# Grafo

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

Por ejemplo los puntos pueden representar personas, y las l´ıneas unen pares de amigos; o los puntos pueden ser centros de comunicaci´on y las l´ıneas representan las comunicaciones entre ellos. En ciencias de la computaco´n a estos diagramas se les conoce como grafos y se definen formalmente como:

**Deftnicio´n 1** *Un grafo tt es un par ordenado* (*V* (*tt*)*, E*(*tt*)) *donde V* (*tt*) *es el conjunto de v´ertices y E*(*tt*) *es el conjunto de aristas, unidas con una funcio´n de incidencia ϕtt que asocia cada arista de G con un par no ordenado de (no necesariamente distintos) v´ertices de*

*tt. Si e es una arista y u y v son v´ertices tal que ϕtt*(*e*) = *{u, v} entonces se dice que e une*

*u y v, y los v´ertices u y v son llamados extremos de e. Para denotar el nu´mero de v´ertices*

*y aristas en tt se utiliza v*(*tt*) *y e*(*tt*) *llamados orden y taman˜o de tt, respectivamente.*

Se dice que si un v´ertice esta´ conectado con otro mediante una arista ´estos son adyacentes, y uno incide con el otro, adem´as si los v´ertices unidos son distintos se dice que son vecinos. Tomando el concepto anterior la representacio´n gr´afica de una arista con v´ertices *u* y *v* es:

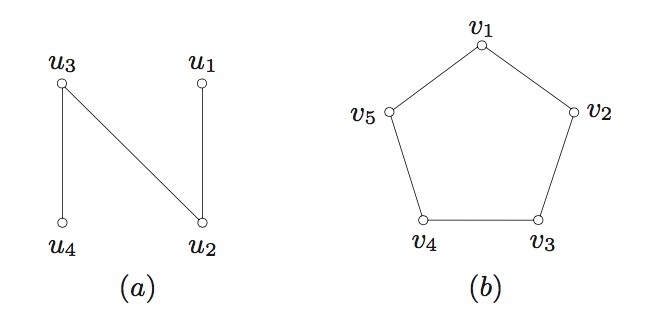
*u v*

Una arista con extremos id´enticos es llamado un ciclo simple o loop, dos o m´as aristas con el mismo par de extremos son llamadas aristas paralelas, la Figura [1.1](#_bookmark2) muestra un ejemplo. Un grafo simple es aquel que no tiene loops o aristas paralelas.

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

*u v w*

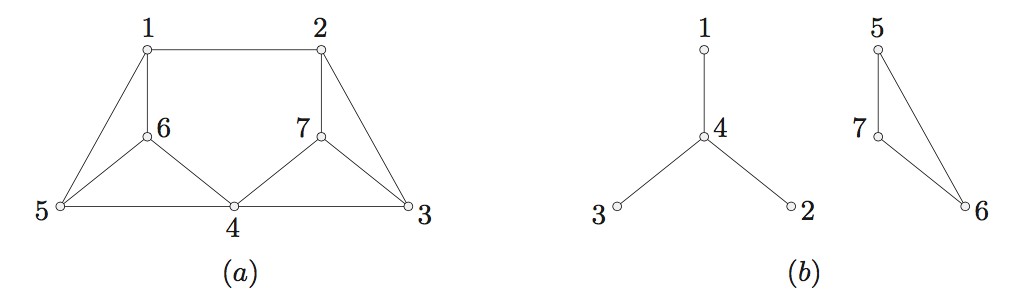
**Figura 1.1:** De izquierda a derecha se muestra un par de aristas paralelas y un ciclo simple



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

adyacentes si no lo esta´n. Un ciclo en tres o m´as v´ertices es un grafo simple cuyos v´ertices pueden ser ordenados en una secuencia c´ıclica de manera que dos de ellos son adyacentes si son consecutivos en la secuencia, o no adyacentes si no lo esta´n; un ciclo en un v´ertice consiste en un v´ertice con un loop, y un ciclo en dos v´ertices consiste de dos v´ertices unidos por un par de aristas paralelas. La longitud de un camino o un ciclo es el nu´mero de aristas que incluye (Figura [1.2](#_bookmark3)).

Un grafo es conectado si, para cada particio´n de sus v´ertices dentro de dos conjuntos no vac´ıos *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 v´ertices puede ser dividido en dos conjuntos no vac´ıos *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

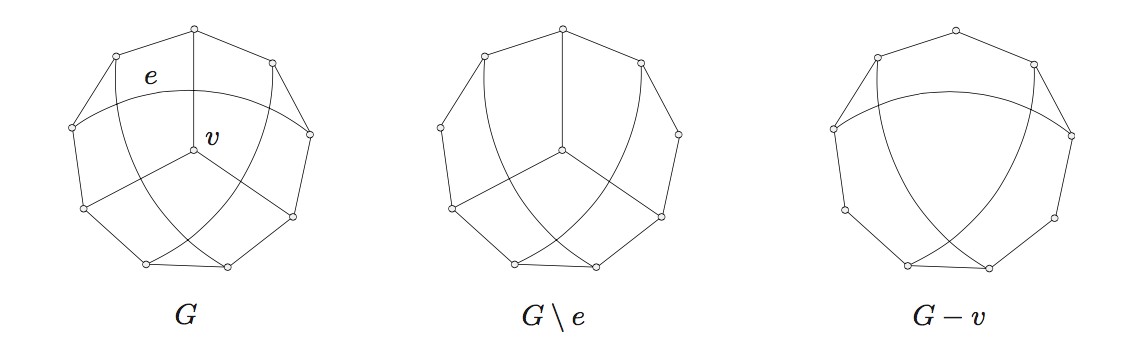
## Subgrafos

Dado un grafo *tt*, existen dos caminos naturales para derivar en grafos m´as pequen˜os de *tt*. Si *e* es una arista de *tt*, se puede obtener un grafo con *m −* 1 aristas eliminando a *e* de *tt*

pero dejando los v´ertices y las aristas restantes intactas. El grafo resultado se denota por

*tt\e*. De manera similar, si *v* es un v´ertice de *tt*, se puede obtener un grafo con *n−*1 v´ertices

eliminando de *tt* el v´ertice *v* junto con las aristas incidentes en ´el. El grafo resultado se denota por *tt − v*.



**Figura 1.4:** Subgrafos resultantes de eliminar la arista *e* y el v´ertice *v* respectivamente

Un grafo es ac´ıclico si no contiene un ciclo. Esto quiere decir que un grafo ac´ıclico debe contener un v´ertice 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 expansio´n de un grafo *tt* es un subgrafo obtenido por medio de eliminacio´n de aristas solamente, en otras palabras es un subgrafo cuyo conjunto de v´ertices es el conjunto de v´ertices completo de *tt*. Si S es el conjunto de aristas eliminados, el subgrafo

de *tt* se denota como *tt\S*.

Un grafo ac´ıclico conectado es llamado un ´arbol, cada componente de un grafo ac´ıclico es un ´arbol, por esta razo´n son llamados bosques.

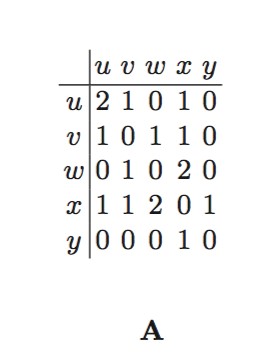
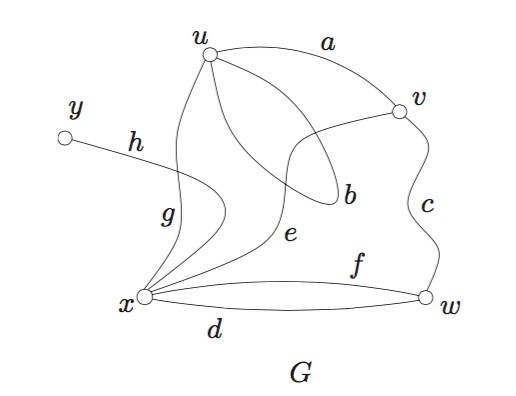
Si *tt* es un grafo conectado y no es un ´arbol, y *e* es una arista de ciclo de *tt*, entonces *tt \ e* es un subgrafo de expansio´n de *tt*, ya que *e* no es una arista de corte de *tt*. Repitiendo este proceso de eliminacio´n de aristas en ciclos hasta que cada arista que se mantenga sea una arista de corte, se obtiene un ´arbol de expansio´n de *tt*.

El complemento *E\T* de un ´arbol de expansio´n *T* es llamado co´arbol, y se denota *T*¯. Para

cada arista *e* = *xy* de un co´arbol *T*¯ de un grafo *tt*, existe un camino *xy* u´nico en *T* que conecta a los extremos, llamado *P* = *xTy*. Entonces *T* + *e* contienen un ciclo u´nico, este ciclo es llamado ciclo fundamental de *tt* con respecto a *T* y *e*.

## Bu´squeda primero en profundidad

La matriz de adyacencia de *tt* es de taman˜o *n × n Att* := (*auv* ), d´onde *auv* es el nu´mero de aristas que unen los v´ertices *u* y *v*, cada loop cuenta como dos aristas en la matriz.



**Figura 1.5:** Grafo *tt* y matriz de adyacencia *A*

La bu´squeda primero en profundidad esta´ basada en el concepto de ´arbol, en la cual el v´ertice an˜adido al ´arbol *T* en cada etapa es aquel que es vecino de la adicio´n m´as reciente a *T* , en otras palabras se debe realizar una bu´squeda en la lista de adyacencia para el v´ertice m´as recientemente an˜adido *x* para un vecino que no esta´ en *T* . Si existe ese vecino, es an˜adido a *T* , si no se retrocede al v´ertice an˜adido a *T* justo antes de *x* y se examinan sus vecinos, hasta que todos los v´ertices son an˜adidos a *T* .

# L´ogica

## ¿Qu´e es una lo´gica?

Una l´ogica siempre constara´ de dos partes:

**Sintaxis**. Especifica los s´ımbolos de los que dispone el lenguaje y la manera en la que se puedan combinar dichos s´ımbolos para construir f´ormulas.

**Sem´antica**. Incluye los elementos necesarios para asignar significados a los s´ımbolos y a las f´ormulas de dicha l´ogica. Concretamente debe contener:

* + - * Una definicio´n del concepto de *interpretaci´on I*, que especif´ıca el significado que puede asignarse a cada s´ımbolo.
* Una definicio´n del concepto de *satisfacci´on*, que establece las condiciones que

deben cumplirse para que una interpretacio´n *I satisfaga* una f´ormula *F* , esto es,

para que podamos afirmar que la f´ormula *F* es *cierta* en la interpretacio´n *I*.

En una l´ogica tambi´en suele haber m´etodos de deduccio´n, que describen c´omo a partir de unas f´ormulas dadas se pueden inferir otras nuevas.

## Lo´gica proposicional

**Sintaxis**. En esta l´ogica, el vocabulario es un conjunto de *variables P* . Una *f´ormula* de la l´ogica proposicional sobre *P* se define como:

Toda variable *xi* de *P* es una f´ormula.

Si *F* y *tt* son f´ormulas, entonces (*F ∨ tt*) y (*F ∧ tt*) son f´ormulas. Si *F* es una f´ormula, entonces *¬F* es una f´ormula.

Nada m´as es una f´ormula.

**Interpretacio´n**. Una *interpretaci´on I* sobre el vocabulario *P* es una funcio´n *I* : *P → {*0*,* 1*}*, es decir, *I* es una funcio´n que, para cada variable de enunciado, dice si es 1 (cierto, *true*) o 0 (falso, *false*).

**Satisfaccio´n**. Sean *I* una interpretacio´n y *F* una f´ormula, ambas sobre el vocabulario *P* . La *evaluaci´on* en *I* de *F* , denotada *evali*(*F* ), es una funcio´n que por cada f´ormula da un valor 0 o 1. Definimos *evali*(*F* ) para todos los casos posibles de *F* , usando *min, max* y -, que denotan respectivamente el m´ınimo, el m´aximo, y la resta (sobre nu´meros del conjunto

*{*0,1*}*), d´onde , *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 s´ımbolo *p* de *P* entonces *evali*(*F* ) = *I*(*p*).

*evali*(*F ∨ tt*) = *min*(*evali*(*F* )*, evali*(*tt*)) *evali*(*F ∧ tt*) = *max*(*evali*(*F* )*, evali*(*tt*)) *evali*(*¬F* ) = 1 *− evali*(*F* )

Se define: si *evali*(*F* ) = 1 entonces *I* satisface *F* , denotado *I |*= *F* . En este caso tambi´en se dice que *F es cierta en I*, o que *I es un modelo de F*.

## Satisfactibilidad, tautolog´ıa, consecuencia y equivalencia

1. Una formula *F* es *satisfactible* si tiene algu´n modelo, es decir, si existe alguna inter- pretacio´n *I* tal que *I |*= *F* . Una f´ormula *F* es insatisfactible (o es una contradicci´on)

si no es satisfactible.

1. Una f´ormula *F* es una tautolog´ıa (o es v´alida), si toda interpretacio´n es modelo de

*F* , es decir, si para toda interpretacio´n *I* se tiene *I |*= *F* . Por ejemplo: (*x ∨ x*¯).

1. Sean *F* y *tt* f´ormulas construidas sobre un vocabulario *P* . *F* y *tt* son l´ogicamente equivalentes si tienen los mismos modelos, es decir, si para toda interpretacio´n *I* sobre

*P* se tiene que *I |*= *tt* si y s´olo si *I |*= *F* . Esto se denotara´ por *tt* = *F* .

## Forma Normal Conjuntiva (CNF)

Una expresio´n l´ogica, la cual contiene variables *xi* y los conectivos l´ogicos *∧, ∨* y *¬* (AND, OR y NOT respectivamente). Se usa la notaci´on *x*¯*i* para denotar la negacio´n *¬xi*, y el

t´ermino literal para referirse a una *xi* o a una *x*¯*i*. El que la expresio´n est´e en *forma normal conjuntiva* significa que es una conjuncio´n:

*C*1 *∧ C*2 *∧ . . . ∧ Cc*

de subexpresiones *Ci*, cada una de las cuales es una disyunci´on de literales (*x*1 *∨ x*3 *∨ x*4) *∧* (*x*¯1 *∨ x*3) *∧* (*x*¯1 *∨ x*4 *∨ x*¯2) *∧ x*¯3 *∧* (*x*2 *∨ x*¯4)

es una expresio´n en Forma Normal Conjuntiva con cinco *elementos conjuntivos*. En general una variable podr´ıa aparecer m´as de una vez en un elemento conjuntivo, y este u´ltimo incluso podr´ıa duplicarse.

El problema de satisfaccio´n-CNF (o CNF-Sat) es el siguiente: Dada una expresio´n en forma normal conjuntiva, ¿existe una asignaci´on de verdad que la satisfaga?.

El problema de conteo de modelos, mejor conocido por #SAT consiste en calcular el nu´mero de modelos de una f´ormula booleana dada, es decir, el nu´mero de asignaciones de verdad para las cuales la f´ormula se evalu´a como verdadera. En este caso el problema que se aborda directamente es #2SAT que es el conteo de modelos sobre una f´ormula 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.

# Problemas tratables

Se considera que un problema es tratable si se puede resolver en tiempo polino´mico *O*(*nk*) para algu´n k. El conjunto de estos problemas se llama clase de complejidad *P* . Por ejem- plo: ordenar una lista de valores o buscar el camino m´ınimo entre dos ciudades en un mapa.

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

*NP* es el conjunto de problemas cuya solucio´n se puede comprobar en tiempo polin´omico pero no existe procedimiento que obtenga la solucio´n en tiempo polin´omico. Por ejemplo: en la suma de subconjuntos se puede comprobar si un subconjunto es solucio´n sumando sus elementos (*O*(*n*)), pero encontrar esa solucio´n es la parte dif´ıcil.

Una M´aquina de Turing es una m´aquina que manipula s´ımbolos en una cinta segu´n un conjunto de reglas. Se compone de:

Una *cinta* infinitamente larga (por la derecha) donde se pueden leer/escribir s´ımbolos.

Una *cabeza lectora/escritora* que apunta a una posici´on de la cinta y se puede mover en ambas direcciones.

Un *control de estado finito*:

* Un registro que almacena el *estado* en que esta´ la m´aquina en un momento dado.
* La m´aquina utiliza el estado actual, y el s´ımbolo le´ıdo de la cinta, para decidir qu´e hacer.

**Deftnicio´n 2** *Una* m´aquina de turing *(MT) es una 5-tupla T* = (*Q,* Σ*,* Γ*, q*0*, δ*)*, donde:*

Q *es un conjunto finito de estados, del cual se supone que no incluye ha ni hr, los dos estados de detenci´on (se usan los mismos s´ımbolos para los estados de detenci´on de todas las MT);*

Σ *y* Γ *son conjuntos finitos, los alfabetos de* entrada *y de* cinta*, respectivamente, con*

Σ *⊆* Γ; Γ *se supone que no contiene* ∆*, el s´ımbolo del espacio en* blanco*;*

*q*0 *el estado inicial, es elemento de Q;*

*δ: Q ×* (Γ *∪ {*∆*}*) *→* (*Q ∪ {ha, hr}*) *×* (Γ *∪ {*∆*}*) *× {R, L, S} es una funcio´n parcial (es decir,*

*posiblemente indefinida en ciertas partes).*

**Deftnicio´n 3 *Clase de complejidad*** *P : Conjunto de problemas de decisi´on que se pue- den resolver con una M´aquina de Turing que termina su ejecuci´on en un nu´mero de pasos acotado por una funcio´n Polino´mica, O*(*nk*)*.*

**Deftnicio´n 4 *Clase de complejidad*** *NP : Conjunto de problemas de decisi´on que se pueden resolver con una m´aquina de Turing* no determinista *que termina su ejecuci´on en un nu´mero de pasos acotado por una funcio´n polin´omica, O*(*nk*)*.*

Una m´aquina de Turng no determinista permite resolver problemas de complejidad expo- nencial en un tiempo polin´omico. Partiendo de la suposici´on de que la m´aquina toma la mejor decisio´n de su ´arbol computacional.

Todo lo que se puede computar con una M´aquina de Turing no determinista se puede hacer tambi´en con una determinista, pero no con la misma eficiencia, de ah´ı que pueda haber

problemas en *NP* pero no en *P* .

## El grafo restringido de una 2-CNF

Existen algunas representaciones gr´aficas de una Forma Normal Conjuntiva, por ejemplo el grafo primal signado o tambi´en conocido como grafo restringido.

Sea *F* una 2-CNF, su grafo restringido se denota por *ttF* = (*V* (*F* )*, E*(*F* )) con *V* (*F* ) = *v*(*F* ) y *E*(*F* ) = *{{v*(*x*)*, v*(*y*)*}* : *{x ∨ y} ∈ F}*, esto es, los v´ertices de *ttF* son las variables de *F* , y para cada cl´ausula *{x ∨ y}* en *F* existe una arista *{v*(*x*)*, v*(*y*)*} ∈ E*(*F* ). Para

*x ∈ V* (*F* )*, δ*(*x*) denota su grado, es decir el nu´mero de aristas incidentes en *x*. Cada arista

*c* = *{v*(*x*)*, v*(*y*)*} ∈ 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 cl´ausula. Los signos *s*1 y *s*2 pertenecen

a las variables *x* y *y* respectivamente. Por ejemplo la cl´ausula *x*0*, y*1 determina la arista con etiqueta ”*x−*+ *y*”, que es equivalente a la arista ”*y* +*− x*”.

Sea *S* = *{*+*, −}* un conjunto de signos. Un grafo con aristas etiquetadas en *S* es el par (*tt, ψ*), d´onde *tt* = (*V, E*) es un grafo restringido, y *ψ* es una funcio´n con dominio *E* y

rango *S*. *ψ*(*e*) se denomina a la etiqueta de la arista *e ∈ E*. Sea *tt* = (*V, E, ψ*) un grafo restringido con aristas etiquetadas en *S × S* y *x* y *y* nodos en *V* , si *e* = *{x, y}* es una arista y *ψ*(*e*) = (*s, st*), entonces *s*(*st*) es el signo adyacente a *x*(*y*).

Sea *tt* un grafo conectado de *n* v´ertices, un ´arbol de expansio´n de *tt* es un subconjunto de

*n−* 1 aristas tal que forman un ´arbol de *tt*. Se denomina co-a´rbol al subconjunto de aristas

que son el complemento de un ´arbol.

# Formato Dimacs

Es un formato utilizado para almacenar f´ormulas booleanas en forma normal conjuntiva. La forma de almacenamiento de una f´ormula en el archivo esta´ dada por algunas reglas:

El archivo puede iniciar con lineas de comentario. La primera letra de cada l´ınea de comentario debe ser una letra “c” minu´scula, pueden ir en cualquier parte del archivo.

Despu´es de las lineas de comentario iniciales sigue la l´ınea que describe el problema. E´sta inicia con una “p” minu´scula seguida de un espacio, despu´es el tipo de problema que para los archivos CNF es “cnf”, un espacio seguido de el nu´mero de variables, un espacio y por u´ltimo el nu´mero de cl´ausulas.

El resto del archivo contiene las lineas que definen las cl´ausulas una a una.

Una cl´ausula se define listando los ´ındices de una literal positiva y un ´ındice negativo para cada literal negativa, el ´ındice 0 no es permitido.

La definicio´n de una cl´ausula se puede extender mas all´a de una l´ınea de texto. La definicio´n de una cl´ausula es terminada por un valor final de 0.

El archivo termina despu´es de que se define la u´ltima cl´ausula.

**Ejemplo 1** *Dada la expresi´on* (*x*1 *∨ ¬x*3) *∧* (*x*1 *∨ ¬x*2) *el archivo en formato Dimacs es: c*

*c*

*p cnf 3 2*

*1 -3 0*

*1 -2 0*

# relsat (1997)

Es una herramienta basada en el algoritmo DPLL (Davis-Putnam algorithm, equivalente a una bu´squeda hacia atr´as con verificacio´n hacia adelante) de bu´squeda exhaustiva de asig- naciones. La mejora que provee en comparacio´n a otras herramientas es el poder procesar r´apidamente f´ormulas disjuntas, es decir, que las variables de una f´ormula no intervengan en alguna otra.

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

# cachet

Es una herramienta basada en el modelo de conteo Caching que es sucesora del ´arbol de bu´squeda de un modelo de conteo DP; ajustando variables y simplificando la f´ormula, ya que se pueden encontrar sub-fo´rmulas que han aparecido en una descomposici´on del algoritmo DP. Si esto sucede, no es necesario volver a contar los modelos, s´olo es necesario saber el conteo de la f´ormula que ya fue evaluada y utilizarlo.

# sharpSAT (2012)

Es una herramienta para resolver #SAT basada en un algoritmo de bu´squeda exhaustiva (DPLL) el cu´al requiere conocer qu´e literales en la f´ormula puede utilizar para realizar una ramificacio´n, requiere heur´ısticas sobre c´omo elegir estas literales. Tambi´en utiliza descomposici´on de componentes y cache de componentes.

# Flex

Flex es una herramienta para buscar patrones l´exicos en un texto, la cu´al requiere una descripci´on de reglas mediante expresiones regulares y c´odigo en C, con esto genera un ejecutable que analiza las ocurrencias de las expresiones regulares y ejecuta el c´odigo en C correspondiente.

# Bison

Bison es un generador de parsers de propo´sito general que convierte una gram´atica libre de contexto en un parser LR, que permite utilizar un m´etodo de lookahead para saber c´omo evaluar los s´ımbolos m´as recientes. Requiere conocimientos de C o C++ para ser utilizado.

# GMP (GNU Multiple Precision arithmetic library)

GMP es una librer´ıa libre para precisio´n aritm´etica, opera con enteros con signo, nu´meros racionales y nu´meros de punto flotante. No tiene un l´ımite pra´ctico en la precisio´n, excepto la memoria disponible en la computadora que lo utiliza.

Esta´ disen˜ado cuidadosamente para ser tan r´apido como sea posible para pequen˜os y gran- des operandos, utilizando algoritmos r´apidos con un c´odigo ensamblador altamente opti- mizado para un conjunto de CPUs.

Cap´ıtulo 2

Algoritmos para el conteo de modelos para

fo´rmulas en 2-CNF

# Conteo de Modelos en Grafos Ac´ıclicos

El conteo de modelos para f´ormulas booleanas cuyo grafo restringido es ac´ıclico se puede realizar en tiempo polinomial [[15](#_bookmark80)]. A continuacio´n se presenta un algoritmo que, mediante un recorrido en preorder, permite contar modelos en un grafo ac´ıclico.

## #2SAT para fo´rmulas en 2-CNF que representan un camino

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

*F* = *{C*1*, C*2*, . . . , Cm}* = *{{xs*1 *∨ xδ*1 *}, {xs*2 *∨ xδ*2 *}, . . . , {xsm ∨ xδm*

*}},*

1 2 2 3 *m m*+1

donde *δi, si ∈ {*0*,* 1*}, i ∈* [1*..m*]. Sea *fi* una familia de cl´ausula de la f´ormula *F* construida como sigue: *f*1 = *∅*; *fi* = *{Cj}j<i*, *i ∈* [1*..m*]. Note que *n* = *|υ*(*F* )*|* = *m* + 1, *fi ⊂ fi*+1, *i ∈* [1*..m −* 1]. Sea *SAT* (*fi*) = *{s* : *s* satisface *fi}*, *Ai* = *{s ∈ SAT* (*fi*) : *xi ∈ s}*, *Bi* = *{s ∈ SAT* (*fi*) : *xi ∈ s}*. Sea *αi* = *|Ai|*; *βi* = *|Bi|* y *µi* = *|SAT* (*fi*)*|* = *αi* + *βi*.

Para cada nodo *x ∈ ttF* se calcula el par (*αx, βx*), donde *αx* indica el nu´mero de veces que la variable *x* toma el valor ‘verdadero’ y *βx* indica el nu´mero de veces que la variable *x* toma el valor ‘falso’ en el conjunto de modelos de *F* . El primer par es (*α*1*, β*1) = (1*,* 1) ya que *x*1 puede tomar el valor verdadero o falso para satisfacer a *f*1. Los pares (*αx, βx*) asociados a cada nodo *xi*, *i* = 2*, . . . , m* se calculan de acuerdo a los signos (*si, δi*) de las literales en la cl´ausula *ci* por la siguiente ecuacio´n de recurrencia:

(*βi−*1 *,αi−*1 + *βi−*1) if (*si, δi*) = (*−, −*)

(*αi−*1 + *βi−*1*,βi−*1 ) if (*si, δi*) = (*−,* +) (*αi, βi*) = (*αi* 1 *,α* + *β* ) if (*s , δ* ) = (+*, −*)

*− i−*1 *i−*1 *i i*

(2.1)

(*αi−*1 + *βi−*1*,αi−*1 ) if (*si, δi*) = (+*,* +)

Note que, si *F* = *fm* entonces #*SAT* (*F* ) = *µm* = *αm* + *βm*. Se denota con *→* la aplicaci´on

de una de las cuatro reglas de recurrencia ([2.1](#_bookmark23)).

**Ejemplo 2** *Sea F* = *{*(*x*1*, x*2)*,* (*x*2*, x*3)*,* (*x*3*, x*4)*,* (*x*4*, x*5)*,* (*x*5*, x*6)*} una f´ormula en 2-CNF cuyo grafo restringido representa un camino. Las series* (*αi, βi*)*, i ∈* [1*..*6]*, se calculan como:* (*α*1*, β*1) = (1*,* 1) *→* (*α*2*, β*2) = (2*,* 1) *ya que* (*s*1*, δ*1) = (+*,* +)*, y la regla 4 tiene que aplicarse.*

*En general, aplicando la regla correspondiente de la recurrencia (*[*2.1*](#_bookmark23)*) de acuerdo a los signos de* (*si, δi*)*, i* = 2*, ...,* 5*, se tiene que* (2*,* 1) *→* (1*,* 3) *→* (3*,* 4) *→* (3*,* 7) *→* (*α*6*, β*6) = (10*,* 7)*, y*

*entonces,* #*SAT* (*F* ) = *µ*6 = *α*6 + *β*6 = 10 + 7 = 17*.*

## #2SAT para fo´rmulas en 2-CNF cuyo grafo contiene aristas para- lelas

Se considera el caso d´onde en una f´ormula *F* en 2-CNF existen dos cl´ausulas que involucran las mismas variables. En este caso, el c´alculo debe considerar cuatro signos diferentes para calcular #2SAT como el caso de un camino. Suponiendo que las dos cl´ausulas son

*ck* = (*xsk , xδk* ) y *cj* = (*xsj*

*, xδj* ) las cu´ales involucran las variables *xi* 1 y *x* . Entonces

*i−*1 *i*

*i−*1 *i −* *i*

se calculan los valores para (*αi, βi*) asociados al nodo *xi*, de acuerdo a los signos (*sk, δk*) y

(*sj, δj* ) como sigue:

(*αi−*1*,αi−*1) if (*sk, δk*) = (1*,* 1) *and* (*sj, δj* ) = (1*,* 0)



(*µi−*1 *,*0 ) if (*s , δ* ) = (1*,* 1) *and* (*s , δ* ) = (0*,* 1)

*k k j j*



(*αi, βi*) = (*βi−*1 *,αi−*1) if (*sk, δk*) = (1*,* 1) *and* (*sj, δj* ) = (0*,* 0)

(*αi−*1*,βi−*1 ) if (*sk, δk*) = (1*,* 0) *and* (*sj, δj* ) = (0*,* 1)



(2.2)

(0 *,µi−*1 ) if (*sk, δk*) = (1*,* 0) *and* (*sj, δj* ) = (0*,* 0)

(*βi−*1 *,βi−*1 ) if (*sk, δk*) = (0*,* 1) *and* (*sj, δj* ) = (0*,* 0)

Siendo *F* una f´ormula en 2-CNF tal que tres cl´ausulas en *F* involucran las mismas variables, entonces el valor de (*αi, βi*) es calculado por la recurrencia ([2.3](#_bookmark26)).

(0 *,αi−*1) if *{*(*xi−*1*, xi*),(*xi−*1*, xi*),(*xi−*1*, xi*)*} ⊆ F*

(*µi−*1 *,*0 ) if *{*(*xi−*1*, xi*),(*xi−*1*, xi*),(*xi−*1*, xi*)*} ⊆ F*

(*αi, βi*) = (*βi* 1 *,α* ) if *{*(*x , x* ),(*x , x* ),(*x , x* )*} ⊆ F*

 *− i−*1 *i−*1 *i i−*1 *i i−*1 *i*

(*αi−*1*,βi−*1 ) if *{*(*xi−*1*, xi*),(*xi−*1*, xi*),(*xi−*1*, xi*)*} ⊆ F*

(2.3)

En el caso de que cuatro aristas paralelas con los mismos extremos y distinta combinacio´n de signos indica que la f´ormula *F* es insatisfactible y por lo tanto #2SAT(*F* ) = 0.

**Procesamiento de Cl´ausulas Unitarias:** Una cl´ausula unitaria representa un loop en el grafo de una f´ormula en 2-CNF. Cuando (*αi, βi*) son calculados sobre un nodo *xi* el cual tiene una arista de loop, se aplica la recurrencia ([2.4](#_bookmark27)).

(*αi, βi*) =

. (0*, βi*) if (*xi*) *∈ U* (2.4) (*αi,* 0) if (*xi*) *∈ U*

Si una cl´ausula unitaria solo determina los valores de su variable. Cuando ambos *xi* y *x*¯*i* existen en la f´ormula *F* como cl´ausulas unitarias, entonces *F* es insatisfactible.

Aristas paralelas y cl´ausulas unitarias deben ser consideradas en un pre-procesamiento de la f´ormula antes de aplicar el algoritmo de conteo general.

## Conteo en grafos ac´ıclicos

Sea *F* una f´ormula en 2-CF donde su grafo asociado *ttF* es ac´ıclico. Se puede asumir que *ttF* tiene un nodo ra´ız, un recorrido del grafo permite generar un ´arbol que tiene un nodo ra´ız ya que es ac´ıclico. Un ´arbol tiene tres clases de nodos: ra´ız, interior y hojas.

Se denota con (*αv, βv* ) el par asociado con el nodo *v* (*v ∈ ttF* ). Se calcula #*SAT* (*F* ) mientras se recorre *ttF* en post-order con el siguiente algoritmo.

**Algoritmo Conteo Modelos para arbol(***ttF* ) **Entrada:** *ttF* - un grafo que representa un ´arbol. **Salida:** El nu´mero de modelos de *F* **Procedimiento:**

Recorrer *ttF* en post-order, para cada nodo *v ∈ ttF* , asignar:

* + - 1. (*αv, βv* ) = (1*,* 1) si *v* es un nodo hoja en *ttF* .
      2. Si *v* es un nodo interior con una lista de nodos hijos asociados, i.e., *u*1*, u*2*, ..., uk*

son los nodos hijos de *v*, una vez que se han visitado los hijos, los pares calculados

son (*αuj , βuj* ) *j* = 1*, ..., k*. Sean *e*1 = *v*

*s*1

*uδ*1 *, e*2 = *vs*2

*uδ*2 *, . . . , ek* = *vsk*

*uδk* las aristas

que conectan *v* con cada uno de sus nodos hijos. El par (*αej , βej* ) se calcula para cada arista *ej* basado en la recurrencia ([2.1](#_bookmark23)) donde *αej−*1 is *αuj* y *βej−*1 es *βuj* para

1

2

*k*

*j* = 1*, . . . k*. Entonces, sea *αv* = Q*k αe* y *βv* = Q*k βe* . Note que este paso incluye

*j*=1

*j*=1

*j j*

el caso en que *v* tiene solo un nodo como hijo.

* + - 1. Si *v* es el nodo ra´ız de *ttF* entonces regresar (*αv* + *βv* ).

Este procedimiento regresa el nu´mero de modelos de *F* en tiempo *O*(*n* + *m*) [[16](#_bookmark81)] 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 f´ormula en 2-CF, si x*1 *es el nodo ra´ız, despu´es de realizar un recorrido post-order el nu´mero de modelos a cada nivel del ´arbol se muestran en la Figura 2.1. El procedimiento Conteo Modelos para arbol regresa αx*1 = 41*, βx*1 = 36 *y el nu´mero total de modelos es:*

*#SAT*(*F* ) = 41 + 36 = 77*.*

(41,36)

X1

+ + (36,5)

(1,1)

(1,1)



+

X

(1,1)

+

5

X

+

+

(2,1)

8

X

2

(2,1)

(9,5)

+

(5,4)

(4,1)

+

X

4 +

+

X

6

+ (2,1)

+

(2,1)

+

X3 (1,1)

+

X7

**Figura 2.1: Conteo de modelos en grafos que representan ´arboles**

# 2.2. Conteo de Modelos en Grafos C´ıclicos

El problema #2SAT es #P-completo, para f´ormulas cuyo grafo restringido es c´ıclico. En [[15](#_bookmark80)] se presenta un algoritmo para el conteo de modelos basado en el nu´meros de aristas del co-a´rbol. Por cada arista del co-a´rbol el algoritmo duplica el grafo de entrada eliminando la arista del co-a´rbol. Por lo tanto la complejidad del algoritmo esta dada por 2*r* donde *r* es el nu´mero de aristas del co-a´rbol. Sin embargo, ya que las aristas del co-a´rbol son aquellas que forman c´ıclos, un grafo con *r* c´ıclos requiere 2*r* ´arboles.

En esta seccio´n mostramos c´omo se puede disminuir el nu´mero de ´arboles generados para as´ı hacer m´as eficiente el procedimiento para cierto tipo de grafos. Los ´arboles de expansio´n que se mencionan a continuacio´n se construyen utilizando el m´etodo Depth First Search (DFS). El m´etodo DFS garantiza que todas las aristas del co-a´rbol son de retroceso es decir, que si se incorporan al ´arbol conectan nodos descendientes con nodos ancestros y

no nodos que esta´n en diferentes ramas del ´arbol. Por tal motivo se puede dividir el grafo original en *r* sub-a´rboles cada uno con su respectivo co-a´rbol, donde *r* es el nu´mero de nodos hijos asociados al nodo que se eligio´ como ra´ız en el grafo original.

**Deftnicio´n 5** *Sea F una f´ormula cuyo grafo restringido ttF* = *ρ*(*F* ) *es c´ıclico. Sea T el*

*´arbol de expansi´on de ttF y T su co-´arbol; se construye una familia de conjuntos* P *de T como sigue: un par de aristas e*1*, e*2 *∈ T pertenecen al mismo conjunto s´ı y s´olo si path*(*e*1) *∩ path*(*e*2) *ƒ*= *∅ donde path*(*ei*)*, i* = 1*,* 2 *es el camino de los v´ertices de las aristas*

*e*1 *y e*2 *en el a´rbol T .*

**Lema 1** *La familia de conjuntos* P *de la Definici´on* [*5*](#_bookmark30) *es una partici´on de T .*

***Prueba* 1** *Sean X, Y ∈* P*, por definici´on X ∩ Y* = *∅ ya que para cada par de aristas e*1*, e*2 *∈ T hay un u´nico camino en T . Dado que cada arista e ∈ T pertenece a una u´nica partici´on entonces* S*X∈*P *X* = *T*

Ahora se puede construir una particio´n de *ttF* .

### Deftnicio´n 6 *t*

1. *Para cada P ∈* P*, se construye un subgrafo como sigue: ∀e ∈ P ,*

*ttP* (*V* (*path*(*e*))*, E*(*path*(*e*) *∪ e*))

1. *Se define ttR* = *ttF \* S *ttP ∈*P*.*

**Lema 2** *Los conjuntos E*(*ttP* )*, P ∈* P *junto con E*(*ttR*) *forman una partici´on de E*(*ttF* )*.*

**Lema 3** *Para cada par de grafos ttP*1 *, ttP*2 *, del Lema* [*2*](#_bookmark31)*, ya sea que V* (*ttP*1 ) *∩ V* (*ttP*2 ) = *∅*

*o V* (*ttP*1 ) *∩ V* (*ttP*2 ) = *{v} es decir es vac´ıo o un conjunto de un solo elemento.*

***Prueba* 2** *Por contradicci´on, supo´ngase que V* (*ttP*1 ) *∩ V* (*ttP*2 ) *ƒ*= *∅ y V* (*ttP*1 ) *∩ V* (*ttP*2 ) *ƒ*=

*{v} lo que significa que hay al menos dos v´ertices v*1*, v*2 *en la intersecci´on, lo que significa*

*que la arista e* = (*v*1*, v*2) *pertenece a la intersecci´on contradiciendo la hip´otesis de que ttP*1

*and tt**P*2 *tienen un conjunto de aristas disjuntas.*

**Teorema 1** *Para cada par de grafos ttP*1 *, ttP*2 *del lema* [*2*](#_bookmark31)

1. *Si ttP*1 *∩ ttP*2 = *∅ entonces los modelos que representa ttP*1 *son independientes de los modelos que representa ttP*2 *es decir Modelos*(*ttP*1 *∪ ttP*2 ) = *Modelos*(*ttP*1 ) *×*

*Modelos*(*ttP*2 )*.*

1. *Si ttP*1 *∩ ttP*2 = *{v} entonces*

*Modelos*(*ttP*1 *∪ ttP*2 ) = *Modelos*(*ttP*1 *|v*1 ) *×Modelos*(*ttP*2 *|v*1 )

+

*Modelos*(*ttP*1 *|v*0 ) *×Modelos*(*ttP*2 *|v*0 )

***Prueba* 3** *1. Si ttP*1 *∩ ttP*2 = *∅ ninguno de los v´ertices o aristas son compartidos. Ya*

*que cada v´ertice de los grafos representan una variable de la f´ormula de entrada,*

*ρ−*1(*ttP* ) *∩ ρ−*1(*ttP* ) = *∅, es decir no hay variables en comu´n de las f´ormulas repre-*

1 2

*sentadas por cada sub-grafo. Es bien sabido que los modelos de f´ormulas sin variables*

*en comu´n se pueden calcular como el producto de los modelos de cada f´ormula.*

*2. Que la intersecci´on sea un conjunto con un solo elemento significa que si F*1 =

*ρ−*1(*ttP* ) *y F*2 = *ρ−*1(*ttP* ) *entonces ν*(*F*1) *∩ ν*(*F*2) = *{x*1*}, es decir hay una so-*

1 2

*la variable en comu´n para ambas sub-fo´rmulas. Una estrategia branch and bound se*

*puede utilizar, donde una rama cuenta los modelos donde x*1 *se fija con el valor ver- dadero y la otra rama cuenta los modelos donde x*1 *se fija con el valor falso en ambas sub-fo´rmulas 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 comu´n entre las sub-fo´rmulas , as´ı que por 1, sus modelos se pueden calcular de forma independiente y posteriormente realizar el producto.*

Del Teorema [1](#_bookmark32) se puede concluir que el nu´mero de modelos de cada uno de los *ttP ∈*P se puede calcular de forma independiente.

Dado que *ttF* es conectado, hay sub-grafos *ttP*1 y *ttP*2 tal que *V* (*ttP*1 ) *∩ V* (*ttP*1 ) = *∅* por lo que debe existir un camino en *ttR* que los una. Afortunadamente, los modelos de un camino se pueden calcular en tiempo polinomial con el procedimiento que se present´o en la secci´on [2.1.1](#_bookmark22).

As´ı, se concluye que

**Teorema 2** *Sea F una f´ormula en 2-CNF, sea tt* = *ρ*(*F* ) *su grafo restringido. Si P es una partici´on de tt como se estableci´o en la Definici´on* [*5*](#_bookmark30) *y Ti, i* = 1*, . . . r son las particio- nes que contienen aristas en el co-´arbol, entonces la complejidad en tiempo para calcular*

*Modelos*(*tt*) *es O*(2*max{|E*(*Ti*)*|} · poly*(*|E*(*T* )*|*))*, donde poly es una funci´on polinomial.*

***Prueba* 4** *Por el Teorema* [*1*](#_bookmark32) *cada Ti se puede calcular de forma independiente. As´ı la duplicaci´on m´axima del grafo esta dado por k* = *max{| E*(*Ti*)*|}.*

### 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 *ρ*(*F* ) es (se omiten los signos ya que todos son positivos):

*v*1 *v*2 *v*3

*v*7 *v*10

*v*4 *v*5 *v*6 *v*8

*v*9

La aplicaci´on de DFS del grafo restringido da la siguiente descomposici´on en ´arbol y co-

´arbol donde la artista del co-a´rbol (*v*4*, v*9) pertenece a una de las particiones y las aristas del co-a´rbol (*v*1*, v*7)*,* (*v*3*, v*10) pertenecen a la otra partici´on. As´ı sus modelos se pueden calcular de forma independiente como se muestra abajo.

 *v*1











*v*1 *v*4 *v*3 























*Mt* 







*v*2 *v*7

*v*4 *v*3

*,*

*v*9 *v*10 















=





 *v*

5











 *v*6







*v*9 *v*8

*v* 



7









*v*10 







 *v*1









 *v*2







 *v*4





*M* 

*v*4 









*v* 

9













*,* 

 *v*1







 *v*2







 *v*3







*v*1 *v*3 







*v*7 *v*10 













*,*  =

  *× M*  

 *v*1 

 

 

 

 

 *v* 

2

 

 

 

 



 *v*



5











 *v*6







*v*9 *v*8

 *v*1 

 

 

 

 

 *v* 

2

 

 

 

 

























 *v*1







 *v*2











 *v*



7











 *v*10





  *v*1

 

 

 

  *v*2





 

 

 























  *v*1 

  

  

  

  *v*2 







  

  

  

 *v*1 

 

 

 

 *v*2 





 

 

 

 *v*4 

 

*MA*  

 *v*4 

 

 

 *v*3 

 

 

 *v*3 

 

 

 *v*3 

 

 

 *v*3 

 

 

  *− MA* 

 *× MA* 

 *− MA* 

 + *MA* 

 *− MA*  

 

 





*v*



5



















 *v*6 





 

 

*v*9 *v*8

 

 





*v*



5



















 *v*6 





 

 

*v*9 *v*8

 

 

 

*v*7

 

 

 

 

 

 *v*10 





 

 

 

 

*v*7

 

 

 

 

 

 *v*10 





 

 

 

 

*v*7

 

 

 

 

 

 *v*10 





 

 

 

 

*v*7

 

 

 

 

 

 *v*10 





 

Cap´ıtulo 3

Desarrollo de la implementacio´n

# Herramientas utilizadas

.

Para poder realizar el procesamiento del archivo de entrada se decidio´

utilizar las he-

rramientas *flex* y *bison* para poder evaluar f´ormulas en 2-CNF expresadas de la forma (*xi ∨ xi*+1) *∧* (*xi*+1 *∨ xi*+2) *∧ . . . ∧* (*xk−*1*, xk*). Para las f´ormulas que esta´n representadas

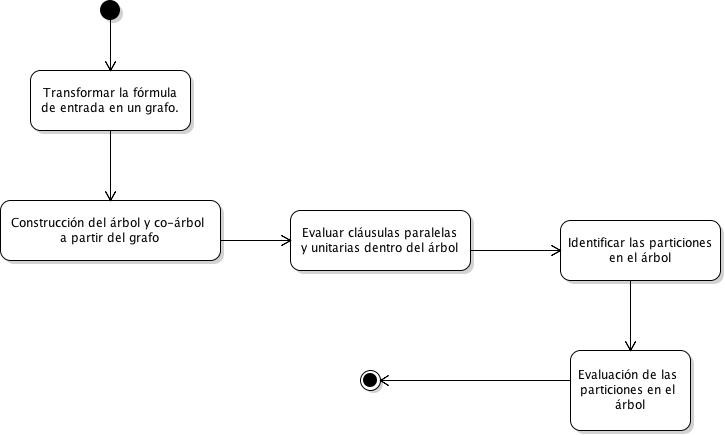
en archivos en formato *dimacs* se realiza el procesamiento con un programa (parser) que identifica s´olo cl´ausulas de dos variables.

Para realizar la programacio´n del algoritmo se decidio´ utilizar el lenguaje C para poder uti- lizar de manera eficiente las caracter´ısticas del algoritmo, la especificacio´n de las estructuras que se usara´n, reducir al m´ınimo el uso de memoria de la computadora y la integracio´n con librer´ıas para manejo de nu´meros muy grandes como *GMP*, as´ı como herramientas para manejar los archivos de entrada.

# Diagrama general del sistema

Para poder realizar el m´etodo de conteo de modelos se realizaron cinco fases genera- les:

* + 1. Transformar la f´ormula de entrada en un grafo
    2. Construccio´n del ´arbol y co´arbol a partir del grafo
    3. Evaluar cl´ausulas paralelas y unitarias dentro del ´arbol
    4. Identificar las particiones en el ´arbol
    5. Evaluaci´on de las particiones en el ´arbol



**Figura 3.1: Diagrama general del sistema**

# Conversi´on de una f´ormula en un grafo.

Para poder convertir una f´ormula dada en 2-Forma Normal Conjuntiva en un grafo se

utiliza la herramienta *flex* para poder identificar a las variables dentro de la f´ormula, las cuales pueden contener letras y nu´meros, para identificar los conectores *∧* y *∨* se utilizan

*Y* y *O* respectivamente, para reconocer una variable en su forma negada el s´ımbolo que se busca es *¬*.

El c´odigo que se utiliza se muestra a continuacio´n:

DIGITO [ 0 *−* 9 ]

ENTERO *{*DIGITO*}*+

LETRA [ a*−*zA*−*Z ]

ID ( *{*ENTERO*} | {*LETRA*}*)+

% %

”O” *{* **return** tOR; *}*

”Y” *{* **return** tAND; *}*

*{* ID*} {* y y l v a l . name=yytext ; i de=s t r l e n ( yytext ) ; **return** tID ; *}* ”*¬*” *{* **return** tNEG; *}*

” ( ” *{* **return** tPA ; *}*

” ) ” *{* **return** tPC ; *}*

Para poder reconocer la f´ormula y decidir que accci´on tomar se utiliza *bison*, para hacerlo se requiere poder representar mediante una gram´atica libre de contexto lo que se quiere hacer.

El c´odigo que se utiliza se muestra a continuacio´n:

% **union**

*{*

**char**

**int**

*∗* name ;

val ;

*}*

% token tNEG tOR tAND tPA tPC tID

% type*<*name*>* tID

% %

enunciado : e n u n l i s t *{* **i f** ( ordenaARB()==1)*{* **while** ( tabl a !=NULL) *{* **i f** ( ordenaARB()==0)**break** ; *}*COMPARB *∗* c=NULL; c=todo ; **while** ( c !=NULL) *{* p r i n t f ( ” %d *\* n” , conteo ) ; **i f** ( c*−>*co !=NULL) *{}* c=c*−>*ne xt ; conteo ++;*}}* duplicaTodo ( ) ; *}*

;

e n u n l i s t : e n u n l i s t tAND enun

*|* enun

;

enun : tPA act *{* i z q=contNeg ; contNeg =0;*}* tID *{* strncpy ( varname1 , $4 , id e ) ; varname1 [ i de ]= ’ *\*0 ’ ; *}* tOR act *{* der=contNeg ; contNeg =0;*}* tID *{* s trn cpy ( varname2 , $9 , i de ) ; varname2 [ i de ]= ’ *\*0 ’ ; *}* tPC *{*addTAB( i zq , der , tabl a , varname1 , varname2 ) ; *}*

;

act : act tNEG *{* **i f** ( contNeg==0)*{* contNeg =1;*}* **else** *{* contNeg =0;*}}*

*|*

;

% %

* 1. **Representacio´n en el lenguaje**

En el siguiente ejemplo visto en la seccio´n [2.1.3](#_bookmark28) se observa que es necesario tener control sobre el conteo de modelos en las aristas del grafo, as´ı como una forma de saber si las variables que se relacionan por medio de las aristas se encuentran en su forma normal o negada, adem´as 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 tendra´n un 0 o 1 asociado.

(41,36)

X1

+ + (36,5)

(1,1)

(1,1)



+

X

(1,1)

+

5

X

+

+

(2,1)

8

X

2

(2,1)

(9,5)

+

(5,4)

(4,1)

+ 4 +

X

+

X

6

+ (2,1)

+

(2,1)

+

X3 (1,1)

+

X7

**Figura 3.2: Conteo de modelos en grafos que representan ´arboles**

Para representar un v´ertice o nodo en el ´arbol que se va a generar se utiliza una estructura que contiene ocho campos, un nombre, la direccio´n de la lista de conteo del v´ertice (la cu´al contiene todas las aristas que conectan a un nodo con sus hijos), la direccio´n de la lista de conteo del v´ertice padre (la arista que conecta a un v´ertice con su v´ertice padre), un campo para el conteo *α* , un campo para el conteo *β* , dos campos para identificar cl´ausulas unitarias adem´as de servir para desarrollar el m´etodo de conteo.

Las aristas del ´arbol se representan por medio de una lista de conteo para cada v´ertice, un elemento de una lista de conteo conecta a dos v´ertices o nodos del ´arbol 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 direccio´n del v´ertice padre, la direccio´n del v´ertice hijo, los indicadores para conocer si una variable esta´ en su forma normal o negada uno para el v´ertice padre y otro para el v´ertice hijo, los campos de conteo temporal *α* y *β*, la direccio´n del siguiente elemento de la lista de conteo y por u´ltimo un indicador utilizado para la evaluaci´on de las particiones.

*// nodos d e l a rb o l*

**typedef struct** nodo *{*

**char** nombre [ 3 0 ] ;

**struct** l i n k *∗* l i s ta C o nte o ;

**struct** l i n k *∗* l i s t a S u p e r i o r ; *//*

mpz t a , b ; *// l o s campos a = a l f a b = b e ta*

**int** unit a , uni t b ; *// ind ic a d o re s para lo o p s*

*}* ;

*// a r i s t a s d e l a rb o l*

**typedef struct**

**struct struct**

l i n k *{*

nodo *∗* padre ; *// nodo padre*

nodo *∗* h i j o ; *// nodo h i j o*

**int** t padre , t h i j o ; *// ind ic a d o re s de forma normal o*

*// negada para padre e h i j o re s pe c t ivame nte*

mpz t a , b ; *// l o s campos para a l f a y b e ta te mporale s*

**struct** l i n k *∗* s i g u i e n t e ; *// s i g u i e n t e e lemento de l a l i s t a*

*// de conteo*

**int** no modi f i ca ; *// indicador para marcar como*

*//no m o d i f ic a b l e*

*}* LINK ;

* 1. **Construccio´n del ´arbol y co´arbol a partir del grafo.**

Teniendo el grafo representado mediante una lista de relaciones *tt* de la forma *gi* = (*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 m´etodo de depth first search para construir el ´arbol y el co´arbol como sigue:

* + 1. Para cada elemento *gi* de *tt*.
    2. Si A esta´ vac´ıo, el elemento *xi* de *gi* se convierte en la ra´ız del ´arbol, en otro caso se agrega *xi* como hijo del u´ltimo nodo que se evalu´o en el ´arbol.
    3. Si *yi* existe en el ´arbol y no es *xi* el elemento *gi* se agrega al co´arbol en otro caso si *xi* = *yi* se evalu´a esta cl´ausula como unitaria si *yi* no existe en el ´arbol se agrega como hijo de *xi* asignando los indicadores (*si*1*, si*2).
    4. Se toma un elemento *gi* de *tt* que contenga a la variable *yi* en el lugar de *xi* (*yi*, *yt* ) y se procede al paso [3](#_bookmark40), si no existe un elemento que contenga a *yi* se procede a evaluar el nodo utilizando la recurrencia vista en [2.1](#_bookmark23) desde *yi* hasta la ra´ız del ´arbol para

*i*

propagar el resultado, despu´es se procede a buscar un elemento *gi* que contenga a la variable *x**i* (*xi*, *yt* ) y se procede al paso [3](#_bookmark40).

*i*

* + 1. Si *tt ƒ*= *∅* y no hay algu´n elemento *gi* que se pueda an˜adir al ´arbol se procede a generar un ´arbol y co´arbol nuevos y regresar al paso [1](#_bookmark39).

Con el procedimiento anterior es posible evaluar f´ormulas que representen grafos desco- nectados y asegura que al terminar de evaluar un ´arbol y ´este no tenga un co´arbol no sea

necesario recorrerlo nuevamente para obtener el nu´mero de modelos, la ra´ız del ´arbol ya tendra´ el nu´mero de modelos resultante.

Para realizar los procedimientos de evaluaci´on se renombran todos los nodos del ´arbol recorri´endolo primero en profundidad y comenzando en 1 .

* 1. **Evaluacio´n de cl´ausulas paralelas.**

Teniendo el ´arbol y co´arbol la evaluaci´on de cl´ausulas paralelas se lleva a cabo buscando todos los elementos del co´arbol que involucren a las mismas variables y se aplican las siguientes reglas:

Si dos cl´ausulas involucran las mismas variables se usa la recurrencia [2.2](#_bookmark25). Si tres cl´ausulas involucran las mismas variables se usa la recurrencia [2.3](#_bookmark26). En el caso de cuatro cl´ausulas el conteo es 0.

Para poder realizar lo anterior se realizan los siguientes pasos:

* + 1. Para cada nodo del ´arbol *ni*.
       1. Si existe m´as de un elemento en el co´arbol que conecte a *ni* a otro nodo *ni−*1 se aplican las reglas anteriores.
       2. Se evalu´a cada nodo desde *ni* a la ra´ız aplicando la recurrencia [2.1](#_bookmark23) para reflejar el cambio en el nodo ra´ız.
  1. **Evaluacio´n de cl´ausulas unitarias**

Cuando existe una cl´ausula unitaria y la variable que aparece en ella esta´ en su forma normal *xi* o negada *x*¯*i* entonces se aplica la recurrencia [2.4](#_bookmark27), posterior mente se evalu´a cada nodo desde *xi* hasta la ra´ız del ´arbol aplicando la recurrencia [2.1](#_bookmark23) para reflejar el cambio en el nodo ra´ız.

* 1. **Identificar las particiones en el ´arbol**

Para realizar este procedimiento de manera eficiente se necesita que los nodos del ´arbol est´en nombrados con nu´meros de la manera que se especifico´ en la seccio´n [3.5](#_bookmark41) y por otro lado teniendo el co´arbol *T*¯ ordenado de manera descendente de tal forma que teniendo dos

elementos *ti* = (*xi, yi, si*1 *, si*2 ) y *ti*+1 = (*xi*+1*, yi*+1*, si*+11 *, si*+12 ) se cumpla que *yi ≥ yi*+1 y

se procede como sigue:

* + 1. Se comienza con una nueva particio´n *pi*.
    2. Para cada elemento *ti* del co´arbol *T*¯.
    3. Se genera el camino *r* que se sigue desde *xi* a *yi*.
    4. Teniendo *p* un conjunto de nodos asociados *Ni*, si *|r ∩ Ni| ≥* 2 se realiza la unio´n

*Ni* = *r ∪ Ni* y se agrega *ti* al conjunto *Ti* de *pi*, si .

* 1. **Evaluacio´n de las particiones en el ´arbol**

Al tener identificado un conjunto de particiones *P* y al haber asegurado que los elementos del co´arbol se encontrara´n ordenados de manera descendente las particiones tambi´en lo estara´n lo que permite que al resolver una por una y sustituir el nu´mero de modelos que se obtiene de cada una en el ´arbol no se altere el resultado al final de la evaluaci´on, permitiendo reducir el taman˜o del ´arbol en cada iteracio´n.

* + 1. Para cada *pi* de *P* .
    2. Se evalu´a siguiendo el procedimiento en [2.2](#_bookmark29) con *Ti* como el co´arbol y tomando como ra´ız el elemento con el valor m´as pequen˜o de *Ni*, se sustituyen los valores obtenidos mediante la evaluaci´on en el nodo ra´ız de la particio´n *pi*. .
    3. Se evalu´a de la ra´ız de *pi*, usando la recurrencia [2.1](#_bookmark23), hasta la ra´ız del ´arbol para reflejar el cambio realizado.

Cap´ıtulo 4

Resultados

* 1. **Enfoque de la herramienta**

En el Cap´ıtulo [2](#_bookmark20) se mostro´ que la complejidad del algoritmo propuesto esta dado en t´ermi- nos 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.

* 1. **Grafos Dispersos**

Son aquellos grafos d´onde el nu´mero de aristas es cercano al nu´mero de v´ertices, lo que se pretende es utilizar grafos d´onde existan grupos de ciclos dispersos, sin que el nu´mero de aristas llegue al m´aximo posible.

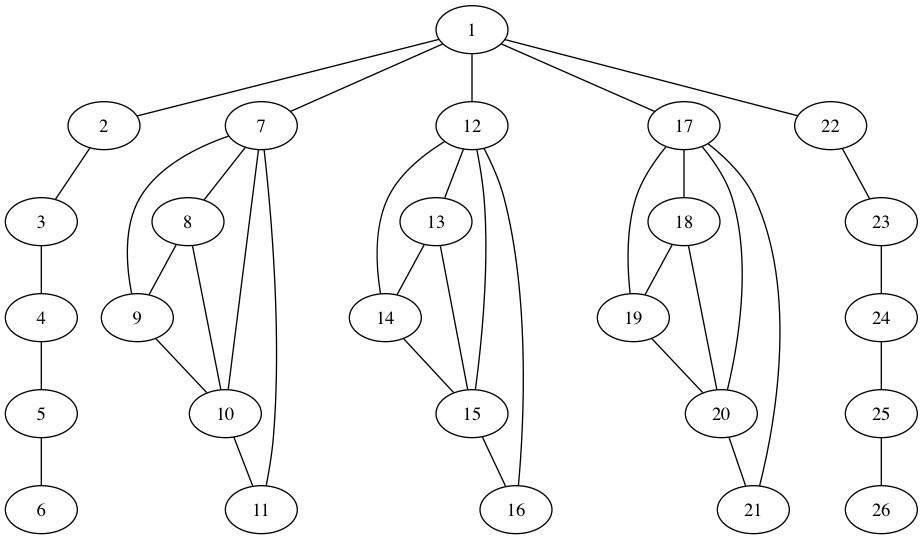
Sea *tt* = (*V, E*) un grafo simple con *n* = *|V |*, *m* = *|E|*. La densidad de un grafo se define con la siguiente f´ormula:

*d* = 2*m*

*n*(*n−*1)

Si d es cercano a 0 entonces se dice que *tt* es un grafo disperso, si es igual a 0 entonces es un grafo d´onde los v´ertices son aislados. Si d es cercano a 1 entonces *tt* es un grafo denso, si es igual a 1 es un grafo completo.

Para poder realizar pruebas con este tipo de grafos se decidio´ utilizar grafos como los de la Figura [4.1](#_bookmark49), en la que se representan particiones en las ramas del ´arbol con 0 *≤ d ≤* 1.



**Figura 4.1: Grafo Sparse**

* + 1. **¿C´omo generar las instancias de prueba para los grafos disper- sos?**

Para poder generar grafos d´onde los ciclos que contienen estuvieran divididos en particiones se gener´o un programa que dado el nu´mero de nodos *n* que tenga el grafo y el nu´mero de ciclos *m* que se quiere en cada particio´n, genera un grafo d´onde a partir de la ra´ız se crean ramas con el nu´mero de nodos *k* necesarios para asegurar que el nu´mero de ciclos que se requiere se cumpla, dada la siguiente f´ormula:

*√*

*k* = 1+ 1+8*m*

2 + 1

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

Para obtener el nu´mero de particiones *p* que tendra´ el grafo resultante se utiliza la siguiente f´ormula:

*p* = (*n−*1)

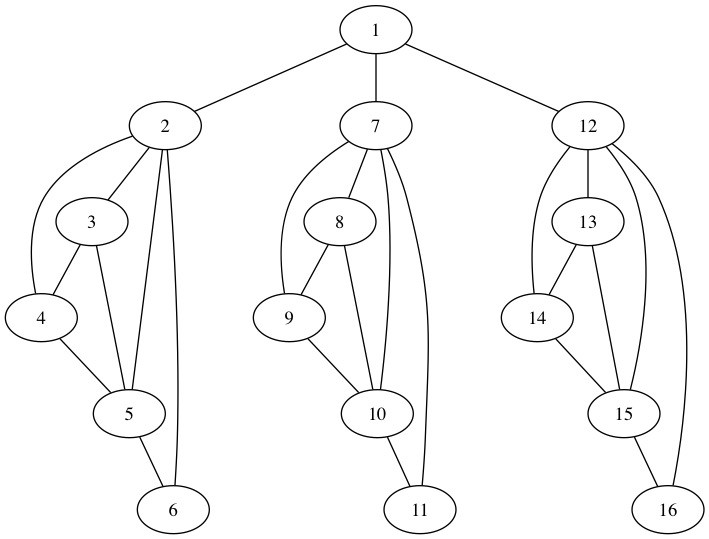
*k*

El nu´mero de aristas *a* que tendra´ el grafo se obtiene mediante la siguiente f´ormula:

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

Para realizar la creacio´n del grafo se muestra en el ejemplo [4](#_bookmark51) el grafo obtenido.

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



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

* + 1. **Pruebas en grafos con** *densidad* = 0

En el cuadro [4.1](#_bookmark53) se muestran resultados obtenidos en grafos ac´ıclicos.

|  |  |  |
| --- | --- | --- |
| Nodos | Aristas | Modelos *≈* |
| 15999 | 15998 | 3111 *×*103814 |
| 19999 | 19998 | 5437 *×*104768 |
| 25999 | 25998 | 1256 *×*106200 |
| 29999 | 29998 | 2196 *×*107154 |

**Tabla 4.1:** Pruebas en grafos sin ciclos

La columna *Nodos* muestra el nu´mero de variables, la columna Aristas muestra el nu´mero de cl´ausulas.

* + 1. **Pruebas en grafos con** 0 *< densidad <* 1

En el cuadro [4.2](#_bookmark55) se muestran resultados obtenidos en grafos en los que se introdujeron un menor nu´mero de ciclos en cada particio´n.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nodos | Aristas | Ciclos | Ciclos por particio´n | 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 ´arbol, la columna Ciclos por partici´on muestra el nu´mero de ciclos que se introdujeron en cada rama del

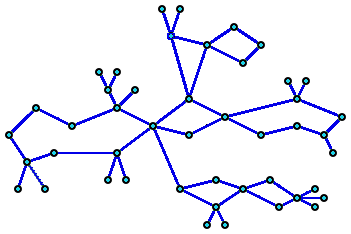
´arbol y la columna de Tiempo(s) muestra el tiempo que tarda la herramienta en resolver cada instancia con respecto al nu´mero de ciclos que tenga.

En el cuadro [4.2](#_bookmark55) se muestran resultados obtenidos en grafos en los que se introdujeron 2, 5 y 10 ciclos por particio´n, para poder saber cu´al era el l´ımite de la herramienta se utilizaron grafos con varios ciclos, ´este l´ımite esta´ dado por la cantidad de memoria f´ısica necesaria para realizar el conteo de modelos.

**4.3. Grafos Cactus**

Un grafo *tt* = (*Vtt, Ett*) es un cactus si:

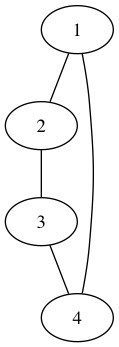
Cada arista de *Ett* pertenece a lo m´as a un ciclo. Cualquier par de ciclos comparten a lo m´as un v´ertice.



**Figura 4.3: Grafo Cactus**

* + 1. **¿C´omo generar las instancias de prueba para los grafos Cactus?**

Para poder generar instancias de prueba se utiliza un programa que realiza la conexio´n de variables creando grupos de cuatro (Figura [4.4](#_bookmark58)) y uni´endolos por medio de una de las variables que componen a cada grupo, cumpliendo que dos grupos diferentes comparten a lo m´as una variable entre ellos.



**Figura 4.4: Parte de los grafos cactus generados**

Para este tipo de grafos s´olo se pretende manipular el nu´mero de nodos *n* que tendra´ el grafo, la forma en c´omo se conectan los grupos es simple:

* + - 1. Se genera el primer grupo de variables *{xi, xi*+1*, xi*+2*, xi*+3*}* y se agregan a una lista

*L*.

* + - 1. Si no se ha llegado al l´ımite de variables se procede a generar otro grupo de tres variables *{xj, xj*+1*, xj*+2*}*.
      2. Se toma el primer elemento de *L* y se conecta con las tres variables generadas en el paso anterior.
      3. Se procede a realizar el paso [2](#_bookmark59) hasta que se haya alcanzado el nu´mero m´aximo de variables *n*.
    1. **Pruebas en Grafos Cactus**

El cuadro [4.3](#_bookmark61) muestra los tiempos obtenidos en grafos cactus generados por el m´etodo visto en la seccio´n [4.3.1](#_bookmark58) d´onde se increment´o el nu´mero de variables en cada ejemplo hasta llegar al l´ımite 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. Comparacio´n con otras herramientas**

Se probaron los grafos presentados en los cuadros [4.1](#_bookmark53),[4.2](#_bookmark55),[4.3](#_bookmark61) en las herramientas sharpSAT y relsat ya que son las m´as recientes y aplican las caracter´ısticas de *cachet*.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Nodos | Cl´ausulas | 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 | Cl´ausulas | 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*×*1013 | 0 | 1.33*×*1013 | 0 |
| 70 | 69 | 0 | 1.85*×*1016 | 0 | 1.85*×*1016 | 0 |
| 85 | 84 | 0 | 4.38*×*1019 | 0 | 4.38*×*1019 | 0 |

**Tabla 4.5:** Pruebas del algoritmo general en grafos que contienen un nu´mero pequen˜o de ciclos

mejora el tiempo de ejecucio´n en comparacio´n 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 nu´mero de ciclos es menor al nu´mero de nodos se mejoro el tiempo de ejecucio´n, la estrategia de *cache de componentes* muestra una mejor´ıa importante en los u´ltimos trece casos de la Tabla [4.4](#_bookmark63).

En la tabla [4.5](#_bookmark64) se generaron instancias pequen˜as en las cuales la herramienta sharpSAT no pudo regresar un resultado.

Como se puede apreciar, en tres instancias sharpSAT no pudo regresar el nu´mero de mo- delos. Por lo tanto se envio un correo al autor de la herramienta pero no hubo respuesta al respecto.

Cap´ıtulo 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 nu´mero de modelos con complejidad del

orden *O*(2*max{|E*(*Ti*)*|} · poly*(*|E*(*T* )*|*)), donde *poly* es una funcio´n polinomial y los *Ti, Ti* son

los ´arboles y co-a´rboles respectivamente de la descomposici´on del grafo original . Este es un

m´etodo que muestra que ciertas instancias se pueden resolver de manera eficiente.

Aunque no se pudieron resolver de manera “eficiente” la mayor´ıa de los grafos presentados

con respecto a sharpSAT, el m´etodo probo´ ser eficiente en casos especiales d´onde ´esta

herramienta si puede regresar un resultado y sharpSAT no.

Con respecto a *relsat* los tiempos de ejecucio´n fueron cercanos a los de ´esta herramienta, super´andola en algunos casos, lo que indica que es posible mejorar su eficiencia y llegar al tiempo de ejecuci´on de *sharpSAT*.

Se utilizo´ el tiempo de ejecucio´n 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 m´etodo para el conteo de modelos en grafos cactus sin tener que duplicar los subgrafos que contiene ciclos, es decir, el m´etodo para el conteo de modelos en grafos cactus es polin´omico. Con esta estrateg´ıa, se considera se puede igual el tiempo de sharpSAT.

Para grafos dispersos, existen estrategias que permiten disminuir el nu´mero de ciclos de la f´ormula de entrada, entre ellos se encuentran la aplicaci´on de resolucio´n unitaria y eliminacio´n de literales puras, por lo que la aplicaci´on de estos procedimientos puede disminuir el tiempo de ejecucio´n.

Existen procedimientos para descomponer una f´ormula en subf´ormulas reducidas en

las que se limita la aparici´on de cada variable a 5 o menos y posteriormente se reduce su aparici´on a 3 por cada subf´ormula.

Se han implementado estrategias de cache de componentes que han probado ser eficientes en f´ormulas grandes, consiste en evaluar subfo´rmulas y agregarlas a un cache, al evaluar una nueva subfo´rmula si ´esta ya fue evaluada, se toma el nu´mero de modelos que se calculo´ sin tener que volver a calcular.

Bibliograf´ıa

[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] Dahll*o*¨f V., Jonsson P., Wahlstr¨om 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, Distin- guished 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. Dis- tinguished 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 Appli- cations of Satisfiability Testing (SAT 2006), pp. 424-429, 2006.

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

[11] Brightwell, G. R.; Winkler, Peter (1991). C¸ ounting linear extensions”. Order 8: 225- 242.

[12] Marcial-Romero J. R., De Ita G., Hern´andez, J. A., Valdovinos, R. M. A Parame- tric 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 Intelli- gence (AAAI-97). New Providence, RI, pp. 203-208.

[14] Satlib. [www.satlib.org,](http://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.