CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS
DEL INSTITUTO POLITÉCNICO NACIONAL

Unidad Zacatenco

Departamento de Computación

Demostración automática en paralelo en ambientes MPI, de memoria compartida y de tipo heterogéneo

Tesis que presenta

Renato Zacapala Zacapala

para obtener el Grado de

Maestro en Ciencias

en la Especialidad de

Ingeniería Eléctrica

Opción

Computación

Director de la Tesis: **Dr. Guillermo Morales Luna**

México, D.F. **Mayo 2007**

A mis padres; y a todos cuantos
hicieron posible éste momento.

Agradecimientos

Al Consejo Nacional de Ciencia y Tecnología por la beca otorgada, subvención necesaria para la realización de este posgrado.

A la Biblioteca de Ingeniería Eléctrica por haber proporcionado sus materiales de consulta durante el transcurso de la maestría.

A mi asesor el Dr. Guillermo Morales Luna, por dedicarme un poco de su preciado tiempo, tiempo durante el cual me orientó y brindo parte de su conocimiento que a la larga hicieron posible la finalización de esta tesis.

Quiero dar gracias también al Dr. José Oscar Olmedo Aguirre y al Dr. Sergio Víctor Chapa Vergara por sus valiosos comentarios durante la revisión del documento de tesis.

A Sofía Reza Cruz por su apoyo en la realización de todos aquellos trámites administrativos que se realizaron durante mi estancia, pero sobre todo por la actitud que siempre tuvo con todos nosotros.

Al personal de la Biblioteca de Ingeniería Eléctrica por su ayuda para encontrar aquellos libros y artículos que componen parte de este documento

Y finalmente, pero no por eso menos importante, a todos los amigos que tuve la fortuna de conocer y que hicieron más amena la estadía en el Centro.

Resumen

La demostración automática de teoremas consiste de métodos algorítmicos para lograr demostraciones por medio de una computadora. La dificultad para resolver un problema puede depender del tipo de lógica que se utilice, en nuestro caso, al utilizar la Lógica Proposicional el problema es decidible pero NP-completo. Por lo que para resolverlos solo existen algoritmos directos de tiempo exponencial en el peor caso. De ahí la importancia de disminuir el tiempo de ejecución de cualquier implementación.

La satisfactibilidad proposicional (SAT) es un problema importante en las ciencias de la computación debido a que la mayoría de las veces es más eficiente pasar un problema a SAT y resolverlo con alguno de los programas existentes, que desarrollar una versión particular para dicho problema. Recientemente ha sido muy importante para una amplia gama de aplicaciones practicas las cuales incluyen el diseño, síntesis y verificación de circuitos integrados, optimización de FPGA, inteligencia artificial y planificación de tareas. Una de las principales aplicaciones que han motivado un nuevo auge de los resolvedores de satisfactibilidad ha sido la verificación de circuitos integrados, dando como resultado un desarrollo acelerado de nuevos diseños.

En este trabajo presentamos diversas implementaciones del algoritmo de Davis-Putnam, que permiten la experimentación con el algoritmo por medio de la modificación de las variables que influyen directamente en su desempeño. Las versiones paralelas se desarrollaron con pThreads y MPI, con el fin de obtener una eficiencia mejor que la versión secuencial al valernos de la utilización de varios procesadores. Con las implementaciones en paralelo se experimentaron diversas estrategias de selección de cláusulas y literales para hacer más eficiente el algoritmo.

Abstract

The automatic demonstration of theorems consists of algorithmic methods to construct or to localize demonstrations. The difficulty to solve a problem can depend on the kind of logic that is utilized. In our case, when the problem use Logic Propositional, it is decidible but NP-complete. There are direct algorithms but they solve the SAT problem on exponential time. So we can see the importance to decrease the run time of any implementation.

The propositional satisfiability (SAT) is an important problem in computer sciences because most of the time it is more efficient to solve the problem with a SAT solver, that to develop a particular version for the aforementioned problem. The propositional satisfiability problem has been very important for an ample range of practical applications which include the design, synthesis and verification of chips, FPGA's optimization, artificial intelligence and planning of tasks. One of the principal applications that have motivated a new prosperity of SAT solvers has been the verification of chips, giving as a result a rapid development of new designs.

We presented a parallel version of Davis Putnam's algorithm, in order to have an algorithm with more efficiency than the sequential version. We develop several strategies of selection of clauses and literals to do the algorithm more efficient.

Índice general

Índice de figuras

Índice de cuadros

Índice de algoritmos

Lista de abreviaturas

Abreviatura	Significado
CNF	*Conjunctive Normal Form* (Forma Normal Conjuntiva)
DIMACS	*Center for Discrete Mathematics & Theoretical Computer Science* (Centro de Matemáticas Discretas y Teoría de la Computación)
DLL	Davis-Logemann-Loveland
DP	Davis-Putnam
DPLL	Davis-Putnam-Logemann-Loveland
k-CNF	Fórmula en CNF con un máximo de k variables por cláusula
LAM	*Local Area Multicomputer*
MAMS	MAXO + MOMS (una combinación de los dos métodos)
MAXO	*Maximum Occurrences* (Máximo número de apariciones para una variable)
MOMS	*Maximum Occurrences in Minimum Size Clauses* (Máximo número de apariciones de una variable en cláusulas de menor tamaño)
MPI	*Message Passing Interface* (Interfaz de paso de mensajes)
PThreads	*POSIX Threads* (Un conjunto de funciones y procedimientos escritos en C estándar, que permiten la ejecución de hilos)
SAT	Satisfactibilidad
SUP	*Selective Unit Propagation* (Propagación unitaria selectiva)
UP	*Unit Propagation* (Propagación unitaria)

Preámbulo

Objetivos

Se trata de desarrollar un algoritmo paralelo de Davis-Putnam para minimizar el tiempo de ejecución con respecto a la versión secuencial e implementarlo en una plataforma MPI.

1. Disminuir el tiempo de ejecución con respecto a la versión secuencial.
2. Tener una eficiencia aceptable.
3. Elegir un método de selección de átomos eficiente.
4. Minimizar el intercambio de datos entre los procesadores para evitar una sobrecarga de comunicaciones.

Descripción general

El problema consiste en elaborar un programa que tome como entrada un archivo de texto con una fórmula booleana representada en forma normal conjuntiva y que regrese SÍ, si es satisfactible y NO en caso contrario.

Nos planteamos elaborar un programa que reciba como entrada una fórmula booleana dada en forma normal conjuntiva desde un archivo de texto y que responda con un «SÍ», cuando ésta sea satisfactible y con un «NO» cuando no lo sea. Al basarnos en el algoritmos de Davis-Putnam se pretende al igual que éste, el programa sea finito, en el peor caso con complejidad exponencial en tiempo con respecto al número de variables.

Los archivos de entrada corresponderán a instancias generadas de forma aleatoria y a un conjunto de pruebas patrón o de referencia existentes que se han utilizado para probar otros demostradores.

Existe una gran cantidad de problemas que han sido modelados y que sirven como pruebas patrón entre los programas para resolver problemas de SAT, y dichas instancias en formato CNF se han convertido en un estándar de comparación.

Se utiliza el modelo de programación Maestro-Esclavo, la ejecución comienza con el proceso maestro y éste es el responsable de leer el archivo de entrada con la instancia del problema en forma normal conjuntiva. En este modelo el maestro es el que controla a los procesos esclavos. El proceso maestro llevará a cabo las funciones de control, como es inicio de los procesos esclavo, supervisión del estado de los esclavos.

Uno de los problemas a resolver es el determinar la afectación del intercambio de datos como resultado de las comunicaciones. En un momento dado puede ser más costoso el tiempo de transmisión de datos que el de procesamiento.

La paralelización se realiza dividiendo el problema en subproblemas, de esta manera se crearán varios árboles de búsqueda, los cuales serán asignados a distintos procesadores. A su vez cada árbol podrá crear otros dos procesos durante la paralelización de la regla de bifurcación.

Al final se pretende probar la versión paralela en una plataforma MPI, en dos ambientes distintos, uno utilizando equipos con memoria compartida y otro utilizando un ambiente heterogéneo con una conexión mediante una red de comunicaciones. De esta forma se analizará que tanto influye y afecta el intercambio de datos entre los procesadores.

Tareas

Las tareas realizadas en la tesis se encuentran en la lista que se muestra a continuación.

1. Implementación del algoritmo de Davis-Putnam para un procesador (versión secuencial del algoritmo en forma recursiva e iterativa).
   - *a*) Implementación de la regla de bifurcación.
   - *b*) Implementación de heurísticas de selección de átomos.
   - *c*) Ejecución de pruebas y análisis los resultados obtenidos.
2. Implementación del algoritmo de Davis-Putnam para varios procesadores en un ambiente de memoria compartida usando hilos (pThreads).
   - *a*) Paralelización la regla de bifurcación.

b) Paralelización la selección de átomos.

c) Ejecución de pruebas y analisis de los resultados obtenidos: Las pruebas se realizarán con instancias creadas artificialmente(de manera aleatoria) y con *benchmarks* existentes.

d) Descripción de los resultados.

3. Migración de la versión de pThreads a MPI.

Motivación

Uno de los principales motivos ha sido el crear una plataforma de experimentación, con la cual se puedan probar fácilmente nuevos métodos de selección de variables en paralelo, que trabajen de manera conjunta para encontrar la solución de la instancia dada.

El algoritmo de Davis y Putnam para cómputo paralelo y distribuido ha sido poco explorado para el problema de SAT.

Desde la década de los 90, el interés por el problema SAT se vio acrecentado, debido a los múltiples problemas en donde SAT tiene en una inmensa cantidad de aplicaciones, como en el área de diseño de circuitos, base de datos, optimización, búsqueda de k-soluciones, etcétera.

Descripción del documento

Capitulo 1 Se presentarán los conceptos que se usarán durante el documento; se presenta un panorama general de la demostración automática de teoremas, el problema de la satisfactibilidad; hasta llegar al estado del arte en los solucionadores del SAT, además se muestran las definiciones y símbolos que se usarán en el presente documento.

Capitulo 2 Se explica a detalle el algoritmo de Davis-Putnam y el usado para resolver el problema del SAT (DPLL).

Capitulo 3 se explica la forma en que se abordo el problema en cada una de las versiones, desde la versión secuencial hasta la versión en MPI, pasando por la versión de hilos.

Capitulo 4 Se describen las estructuras de datos usadas, las tres formas en que se representan las instancias dentro del algoritmo.

Capitulo 5 Se presentan los resultados obtenidos además de su análisis y la comparación con otras implementaciones.

Capitulo 6 se encuentran las conclusiones, además de los temas futuros que pueden abordarse para seguir con esta tesis.

Apéndice A Se presenta es un esbozo del manual de usuario de cada una de las versiones del algoritmo de Davis-Putnam, desde la versión secuencial, hasta las versiones paralelas.

Capítulo 1

Introducción

La demostración automática se remonta a los orígenes de la lógica misma con el uso de las primeras notaciones matemáticas formales en el siglo XVI y adquirió una gran importancia con la aparición de los resultados de Herbrand, Gödel y Church.

En el siglo XIII se presenta el primer intento de utilizar el razonamiento automático aplicado a la lógica aristotélica, por Ramón Llull (también conocido como Raimundo Lulio) en su *Ars Generalis Ultima* [23], conocida también como *Ars Magna*. Este consistió en un dispositivo mecánico que facilitaba el manejo de sistemas lógicos, orientado principalmente a la teología, con el dispositivo pretendía dar respuesta a lo que el llamaba las verdades universales. Posteriormente, aparecieron la máquina aritmética de Pascal y la máquina analítica de Babbage. Durante la Revolución Industrial, el conde de Stanhope (1753-1816) inventó una máquina para resolver silogismos de la lógica tradicional, W. Jevons inventó el piano lógico y en 1882 surgió la máquina de Marquarnd. En la década de los cincuenta del siglo XX, con la aparición de las primeras computadoras, se desarrollaron los primeros programas de demostración automática.

En 1960 Gilmore, Prawitz y posteriormente Martin Davis y Hilary Putnam lograron procedimientos de demostración para el cálculo de predicados con base en el teorema de Herbrand. Sus métodos consistían, esencialmente, en ir obteniendo de manera progresiva instancias de las cláusulas iniciales usando los términos del universo de Herbrand para ellas.

Desde un inicio, el algoritmo de Davis-Putnam-Longemann-Loveland (DPLL) ha sido uno de los más utilizados para resolver instancias de SAT. Las soluciones a los problemas prácticos, sin embargo, seguían siendo difíciles de alcanzar. Investigaciones recientes han aportado nuevos algoritmos con los cuales ha sido posible

resolver instancias de problemas prácticos intratables anteriormente. Sin embargo la mayoría de dichos algoritmos están basados en un enfoque de búsqueda local.

Con el aumento en la velocidad de las comunicaciones, ha sido posible desarrollar aplicaciones paralelas que resuelvan el problema de la satisfactibilidad con algoritmos completos en un tiempo razonable, aprovechando el cómputo paralelo que se da por la interconexión de las computadoras. Aún con aplicaciones paralelas y los avances significativos en la tecnología, existen numerosos problemas que no se pueden resolver. Algunos de ellos son generados artificialmente, pero otros problemas difíciles son propios de aplicaciones de la vida real.

Por la importancia de los resultados en las pruebas de satisfactibilidad se ha creado un conjunto de patrones para probar la eficacia de los programas resolvedores de SAT. Uno de los *benchmarks* más utilizados son los del *Center for Discrete Mathematics & Theoretical Computer Science* (DIMACS).

Actualmente el desarrollo de la ciencia y la tecnología no sólo busca una solución, sino la mejor solución dado un problema definido. Nos encontramos entonces, ante un problema de optimización.

Uno de los principales factores que afectan la eficiencia en los resolvedores de satisfactibilidad tiene que ver con el método de asignación de variables para la navegación por el espacio de búsqueda. Aún con un buen método, muchos problemas requieren computacionalmente mucho tiempo para resolverse.

Recientemente ha habido toda una variedad de esfuerzos para paralelizar la selección de átomos, es decir, fórmulas sin conectivos lógicos y así obtener una reducción en tiempo de cómputo. La mayoría de los métodos disponibles mantienen una base de datos de las soluciones parciales obtenidas durante el proceso de búsqueda. La actualización y mantenimiento de dicha base de datos se conoce como «aprendizaje».

1.1. Conceptos preliminares

La mayoría de los conceptos que se usarán a lo largo de la tesis se encuentra en la siguiente lista, los que no se encuentren se definirán conforme se vayan utilizando.

Definición 1 (Literal). Una literal l es una variable booleana x_i o su negación $\neg x_i$.

Definición 2 (Cláusula). Es la disyunción—separación por el conectivo lógico *o* ($\vee$)— de distintas variables booleanas(literales), ya sean positivas o negativas.

Definición 3 (Instancia). Es una fórmula booleana formada por un conjunto de cláusulas unidas por el conectivo lógico *y* ($\wedge$).

A continuación se muestra una lista de los símbolos y notación usada en la tesis.

- Φ o φ es una fórmula booleana en forma normal conjuntiva (CNF).
- $\emptyset$ representa que el conjunto de cláusulas que integran a φ esta vacío.
- k es el número de literales por cláusula en una instancia k-CNF o el máximo número de literales en una instancia CNF.
- l es una literal.
- m es el número de cláusulas en φ.
- n es el número de variables en φ.
- C es una cláusula o un conjunto de cláusulas.
- A es un conjunto de variables al que se le ha asignado un valor (literales)—cuyo tamaño puede variar desde 1 hasta n—, que al ser aplicados a Φ permiten saber si esta es satisfactible o no.
- $\square$ representa una cláusula vacía, esto es, sin literales en ella.
- C_i es cláusula número i.
- $|C_i|$ es el número de literales en la cláusula C_i.
- Las variables booleanas existentes en φ son representadas como $x_1, x_2, \ldots, x_n$
- $|x_i|$ es el número de veces que la variable x_i aparece en φ.
- C_p es el conjunto de cláusulas que contienen a p en φ.

- $C_{\neg p}$ es el conjunto de cláusulas que contienen a $\neg p$ en φ.
- $C_{\times p}$ es el conjunto de cláusulas que no contienen a p ni a $\neg p$
- C'_p es el conjunto C_p en donde han sido eliminadas todas las apariciones de p.
- $C'_{\neg p}$ es el conjunto $C_{\neg p}$ en donde han sido eliminadas todas las apariciones de $\neg p$.
- $C(p)$ es la unión de C'_p y $C_{\times p}$.
- $C(\neg p)$ es la unión de $C'_{\neg p}$ y $C_{\times \neg p}$.

1.2. El problema SAT

El problema de la satisfactibilidad booleana trata de un problema donde se desea saber si una expresión booleana en forma normal conjuntiva (FNC) tiene asociada una asignación de valores para sus variables que hace que la expresión sea verdadera.

Los componentes de un problema SAT son tres:

- Un conjunto de n variables: $x_1, x_2, ..., x_{n-1}, x_n$
- Un conjunto de literales. Una literal es una variable $(l = x)$ o una negación de una variable $(l = \neg x)$.
- Un conjunto m de distintas cláusulas: $C_1, C_2, ..., C_{m-1}, C_m$ Cada cláusula consiste de solo literales combinadas únicamente por conectivos lógicas.

Para resolver este tipo de problemas existen dos tipos de métodos, los completos y los incompletos.

A pesar de su antigüedad, el método de Davis-Putnam sigue siendo uno de los más completos procedimientos para satisfactibilidad. El método original fue basado en una regla de resolución que elimina las variables de una en una y añade todos los posibles resolventes al conjunto de cláusulas.

Desafortunadamente, este método requiere en general un espacio de búsqueda exponencial con respecto al número de variables. Davis, Logemann, y Loveland, reemplazaron la regla de resolución con una regla de partición que divide el problema en dos pequeños subproblemas (Davis, Logemann, Loveland, 1962).

No existe un algoritmo conocido que garantice la solución de algún problema de este tipo en un tiempo polinomial con respecto al número de variables, exepto para el problema 2-SAT.

Las investigaciones de SAT consideran los problemas en la Forma Normal Conjuntiva FNC, o CNF por sus siglas en inglés. Una fórmula está en CNF si es una conjunción de cláusulas; una cláusula es una disyunción de literales, donde una literal es una variable booleana negada o no negada.

Una fórmula P está en forma normal conjuntiva si tiene la forma

$$C_1 \wedge C_2 \wedge C_3 \wedge \ldots C_i \ldots \wedge C_{m-1} \wedge C_m$$

siendo $m \geq 1$ y cada C_i es una disyunción de literales

$$l_1 \vee l_2 \vee l_3 \vee \ldots l_i \ldots \vee l_{n-1} \vee l_n$$

Una cláusula que contiene solo una literal es llamada cláusula unitaria. Una cláusula que no contiene literales es llamada la cláusula vacía y es interpretada como falso. Pueden establecerse restricciones en el tamaño de las cláusulas del problema.

La Satisfactibilidad es expresada con variables discretas, pero algunos algoritmos hacen sus cálculos con variables continuas. Esto define el eje horizontal discreto-continuo en el espacio.

Satisfactibilidad tiene un conjunto de restricciones que deben de ser satisfechas exactamente, pero algunos procedimientos consideran cambios en los valores variables de la cláusulas que no satisfacen las restricciones. Esto define el eje vertical. Muchos algoritmos SAT son secuenciales, mientras que algunos han sido implementados en paralelo.

El problema de la satisfactibilidad proposicional es un ejemplo clásico de un problema NP-Completo—fue el primer problema que se demostró ser NP-Completo—, se sabe que cualquier problema NP-Completo es transformado a un problema SAT en tiempo polinomial, de ahí se puede entender la importancia que tendría el resolver el problema SAT en tiempo polinomial.

Si un problema es NP-completo, puede ser trasladado a SAT usando una codificación cuyo tamaño es limitado polinomialmente. Las investigaciones anteriores han demostrado este enfoque es muy efectivo en la práctica.

Durante los últimos años se han realizado diferentes investigaciones con el fin de encontrar un algoritmo para resolver el problema de manera eficiente; aun cuando se ha reducido el tiempo considerablemente con respecto a las versiones

iniciales, todavía siguen siendo intratables computacionalmente una buena cantidad de problemas.

Por tal motivo se ha recurrido a realizar algoritmos paralelos, con el fin de encontrar la solución en un tiempo menor que su contraparte secuencial y de volver tratables cierto tipo de problemas. Aunque esto también tiene un limite, dado que el problema es exponencial llega el momento en que las versiones paralelas sucumben ante el problema a resolver.

1.2.1. Variaciones de SAT

El problema SAT que consiste en encontrar una asignación de valores que satisfagan la fórmula, si es que tal asignación existe, esto es, Dada:

- Una fórmula booleana de n variables
- y m cláusulas.

se debe encontrar:

- Una asignación de valores que satisfaga la fórmula booleana, esto es que al evaluar la asignación en la fórmula ésta sea verdadera.
- La prueba de que la fórmula no tiene solución.

Se dice que una instancia φ, de un problema de satisfactibilidad proposicional, es satisfactible si existe una asignación de valores $A : V \rightarrow \{0, 1\}$, tal que A satisface cada $C \in \varphi$, donde C es una cláusula.

Sin embargo existen muchas otras variantes, algunas de las más conocidas se mencionan a continuación.

- Decisión SAT : ¿Existe alguna asignación que satisfaga la fórmula?. Para contestar esta pregunta se debe analizar la instancia hasta encontrar una asignación valida, en el peor caso se recorrerá toda la instancia sin encontrar una asignación valida.
- #SAT : Encontrar el número de asignaciones que satisfacen la fórmula.
- MAJ-SAT : Trata de verificar si la mayoría de las asignaciones satisfacen la fórmula.

- MAX-SAT : Encontrar una asignación que maximice el número de cláusulas satisfechas —o lo que es lo mismo, que minimice el número de cláusulas insatisfechas. En este problema la asignación de variables que da solución al problema es precisamente aquella que satisface a toda la fórmula.

Todas las versiones de SAT anteriores son NP-Completas, ya que todas tratan de encontrar al menos una asignación de variables satisfaga la fórmula, al igual que el SAT simple, eso es suficiente para que el problema sea NP-Completo, puesto que si la instancia es insatisfactible, cada versión tendría que comprobar todas las asignaciones posibles para probar verificar su objetivo.

Algoritmos completos

Los algoritmos completos para resolver el problema de la satisfactibilidad proposicional caen en dos principales categorías: búsqueda en *backtrack* (el procedimiento de Davis-Logemann-Loveland [11]) y resolución (el algoritmo original de Davis-Putnam[12] y y resolución direccional[4]).

El algoritmo de búsqueda en *backtrack* puede ser visto como una forma ordenada de descubrir el orden de asignación de las variables.

- Exploran todo el espacio de búsqueda
- Si existe una asignación que satisfaga la instancia, ésta será encontrada
- Si no se encuentra, la instancia es insatisfactible

Algoritmos incompletos

Por lo regular los algoritmos incompletos solo son usados en instancias que se sabe son satisfactibles, en las cuales solo es necesario saber cuales son los valores que la satisfacen.

- No exploran todo el espacio de búsqueda
- Si existe una asignación que satisfaga la instancia, ésta puede ser encontrada
- Si no se encuentra, la instancia puede o no ser insatisfactible

La ventaja de estos algoritmos es su rapidez, misma que se obtiene a base de sacrificar la certeza de saber si un resultado es insatisfactible, en caso de que sea así. En este tipo de algoritmos solo se puede estar seguro del resultado si este es satisfactible.

1.2.2. Descripción general

Como ya se mencionó anteriormente, los algoritmos de satisfactibilidad pueden dividirse en dos clases: algoritmos completos y algoritmos incompletos. La mayoría de los algoritmos completos están basados en el procedimiento de Davis-Putnam, mientras que los algoritmos incompletos más utilizados son algoritmos de búsqueda local.

Los algoritmos completos realizan una búsqueda a través del espacio de todas las posibles interpretaciones para probar que la fórmula es satisfactible (buscando una interpretación que satisfaga la fórmula) o insatisfactible (habiendo explorado todo el espacio de búsqueda, no se encuentra ninguna interpretación que satisfaga la fórmula).

Los algoritmos de búsqueda local no exploran, en general, todo el espacio de búsqueda. Empiezan generando una interpretación de forma aleatoria e intentan encontrar alguna interpretación que satisfaga la fórmula cambiando, en cada paso del algoritmo, el valor que tenga asignado una variable proposicional.

Tales cambios se repiten hasta que se encuentra una interpretación que satisfaga la fórmula o bien hasta que se realiza un número de cambios prefijado. Todo este proceso se repite hasta que se encuentre una solución o hasta que se realice un cierto número de intentos.

La principal diferencia entre los distintos algoritmos que se han implementado radica en la forma de seleccionar la variable a la que se cambia el valor de verdad.

Una dificultad que presentan los algoritmos de búsqueda local es que pueden quedar atrapados en mínimos locales, otra es que son incompletos y pueden fracasar en probar que una fórmula es insatisfactible: si encuentran una solución, la fórmula es satisfactible y el algoritmo termina, pero si el algoritmo termina sin encontrar una solución, no se puede concluir si la fórmula es satisfactible o insatisfactible. Al final una fórmula solo puede tener dos resultados posibles, bien es insatisfactible o satisfactible —con esto ultima opción incluso puede darse el caso de ser una tautología, cuando el resultado es siempre satisfactible no importando que valor tomen cada una de las variables que integran la fórmula.

$$\begin{aligned} A_1 &= \{l_{1,1}, l_{1,2}, \ldots, l_{1,n}\} \\ A_2 &= \{l_{2,1}, l_{2,2}, \ldots, l_{2,n}\} \\ \vdots &\qquad \ddots \\ A_{2^n} &= \{l_{2^n,1}, l_{2^n,2}, \ldots, l_{2^n,n}\} \end{aligned}$$

Cuando el algoritmo DPLL termina y encuentra una asignación de valores que satisfacen la fórmula, generalmente $|A| < n$, lo que quiere decir que no todas las

variables fueron instanciadas. Eso quiere decir que la instancia es satisfacitble no importando el valor que contengan las variables que no fueron instanciadas.

Podemos obtener otras asignaciones que hagan a φ satisfacible al asignarles valores a las variables que no fueron instanciadas.

Con la asignación de valores A que satisface a φ

$$A = \{l_1, l_2, \ldots, l_b, \ldots, l_n\}$$

Si $|A| < n$ entonces $b = |A|$ y $d = n - b$, con lo que se tienen d variables que no fueron instanciadas, y entonces podemos obtener 2^d asignaciones que también satisfacen a φ.

1.3. Situación actual

Existen pocas versiones paralelas en comparación con las secuenciales, además la mayoría de las versiones paralelas no son creadas ex profeso, sino que son modificaciones de alguna versión secuencial.

Como ejemplo de lo dicho anteriormente, se tienen a `SATO` y `PSATO`; `Satz` y `Parallel Satz`. Otras tantas veces, podría decirse que la mayoría, se toman ideas de otros programas como es el caso de `PaSAT` [32], que toma algunas ideas de Zhang y Silva-Sakallah. De las versiones paralelas, la mayoría utilizan conjuntos de funciones para el lenguaje C, que permiten la ejecución de los programas en varias máquinas, como lo son MPI y PVM.

En menor proporción encontramos versiones que utilizan PThreads, por la exigencia de equipos con memoria compartida y su consiguiente desventaja numérica. Existen también versiones distribuidas que se ejecutan en mallas, de todas es la que tiene más posibilidades de ejecutarse en un gran número de equipos y por lo tanto tiene un mayor poder de computo, sin embargo es la menos utilizada de todas. Y aun en menor medida están los programas desarrollados utilizando *sockets* como medio de comunicación entre procesos.

Algunos programas paralelos utilizan técnicas complejas de aprendizaje, lo cual es muy costoso en términos de comunicaciones, ya que al estar los procesadores trabajando en diferentes partes del espacio de búsqueda es necesario compartir información. En algunos casos se limita el número de procesadores a sólo unos pocos, para reducir costos en comunicaciones, tal como procede `PaSAT`.

Por otra parte, el tiempo de ejecución de los resolvedores de satisfactibilidad dependen en gran medida del problema por resolver, aún teniendo la misma canti-

dad de variables y cláusulas, algunos pueden resolverse rápidamente y otros tardar un tiempo considerable.

Es difícil partir el espacio de búsqueda para asignarlo a los distintos procesadores, por es motivo es más común el uso de resolvedores de SAT secuenciales.

`GrADSAT` es un resolvedor de satisfactibilidad que está basado en `zChaff`, el cual fue desarrollado por Sharad Malik en la Universidad de Princeton. `zChaff` es resolvedor completo, como tal garantiza encontrar una instancia de satisfatibilidad si el problema es satisfactible o probar que el problema es insatisfactible.

La paralelización que realiza `GrADSAT` al problema es por medio de la división del problema en subproblemas, idea que se tomara para nuestra versión paralela. De esta forma se evita tener una gran cantidad de comunicaciones, cada procesador tiene su propio árbol de búsqueda y solamente se comunica con el proceso maestro para averiguar si una parte del árbol ya ha sido resuelto por otro proceso.

1.4. Importancia de las aplicaciones

La importancia de SAT radica en la gran cantidad de problemas que pueden ser resueltos con él, dentro de sus muchas aplicaciones, se pueden mencionar las siguientes:

Lógica El problema de satisfacción restringido, el problema n-reinas, inferencia extendida, programación lógica, inferencia abductiva para sintetizar mezclas de hipótesis, procesamiento semántico de información, crucigramas, Coloreado de gráficas, detectar grafos y subgrafos isomórficos, criptología matemática, el problema del homomorfismo autómata, encontrar árboles de expansión y viajes Euler en un grafo, resolución del problema del vendedor viajero, y lógica aritmética..

Ciencias Computacionales Criptoaritmética, redes neuronales, planeación de tareas, configuración de tareas, procesamiento paralelo y distribuido.

Ingeniería de *software* Verificación de programas, corrección automática de errores, optimización de compiladores, optimización de conjuntos de instrucciones, sistemas de tiempo real.

Robótica Problema de visión vinculada, problema de empaques y problemas de planeación de tareas y trayectorias.

Bases de datos Operaciones de objetos, consistencia, mantenimiento, verificación de redundancia, respuesta a búsquedas, optimización de búsquedas, control de concurrencia, sistemas de bases de datos distribuidas, mantenimiento de verdad, problema de homomorfismo relacional y organización del conocimiento y reconocimiento de sistemas.

Verificación de circuitos Diseño automático de circuitos integrados, modelado de circuitos, minimizaciones lógicas, asignaciones de estados, minimización de estados, síntesis de circuitos asíncronos, codificación de entradas y salidas para máquinas secuenciales, particiones lógicas, análisis de interconexión, optimización del desempeño, generación de pruebas, diseño aritmético de circuitos lógicos.

Demostración automática Prueba de teoremas.

En este capitulo se presentó una breve explicación del concepto de SAT, sus requerimientos y la clasificación de los algoritmos de SAT, así como algunas de sus aplicaciones, tanto teóricas como prácticas. También se revisaron las clases básicas de algoritmos SAT.

Capítulo 2

Algoritmo de Davis-Putnam

El método de Davis-Logemann-Loveland o Davis-Putnam-Logemann-Loveland, mejor conocido como Davis-Putnam es el algoritmo completo más usado para resolver el problema de satisfactibilidad. El nombre del algoritmo varía debido a los nombres de los artículos en los que se describen originalmente, el algoritmo de Davis-Logemann-Loveland (DLL)debe su nombre a que esta basado en el articulo *A machine program for theorem-proving*[11] escrito por los tres autores mencionado, pero dicho articulo esta basado en su mayoría por el articulo previo *A Computing Procedure for Quantification Theory*, escrito por Davis-Putnam(DP), por lo que suele llamarsele así también. En otros artículos se les da crédito a los cuatro autores por lo que también se le conoce como Davis-Putnam-Logemann-Loveland (DPLL).

Es un algoritmo que utiliza un enfoque de *backtracking* para resolver el problema. El algoritmo de Davis-Putnam se puede describir recursivamente de la manera siguiente: primero se verifica si una fórmula F es satisfactible[1] (si no tiene cláusulas) o es insatisfactible[2] (cuando contiene una cláusula vacía) y en ambos casos el algoritmo termina. En caso contrario, se selecciona una literal l_i (una variable o su complemento) y se aplica el algoritmo de búsqueda en forma recursiva para encontrar una asignación de variables que satisfaga la fórmula $F\lceil l_i = 0$, que se obtiene al hacer cero todas las variables l_i en F ($l_i = 0$ en F). Si se encuentra,entonces se tiene una asignación para F. Si no se encuentra, se repite la búsqueda ahora asignando a l_i el valor de 1 ($l_i = 1$ en F). Si ninguna de las búsquedas encuentra una asignación que satisfaga a la fórmula F, entonces se

[1]Termino —aunque en menor medida— también conocido como **consistente**[12].

[2]Termino también conocido como inconsistente.

dice que F es no satisfactible.

Definición 4. Un algoritmo de demostración de teoremas es lógicamente completo si puede siempre probar la insatisfactibilidad de una fórmula siempre que ésta así lo sea.

Si C es un conjunto insatisfactible de cláusulas, al aplicar el algoritmo con C y alguna variable x, las instancias obtenidas $C(x)$ y $C(\neg x)$ también son insatisfactibles. Dado que el número de variables en $C(x)$ y $C(\neg x)$ es menor que el que existe en C, en algún momento se encontrará la cláusula vacía o se encontrará la fórmula vacía, por lo que el algoritmo es completo[7].

Entrada: Una instancia F en forma normal conjuntiva.
Salida: Si la instancia es satisfactible regresar `TRUE` y en caso de ser insatisfactible `FALSE`

```
1  Algoritmo:DPsat
2  begin
3      if F contiene una cláusula vacía then
4          return FALSE;
5      end
6      if F está vacío then
7          return TRUE;
8      end
9      if existe una literal pura L en F then
10         return DPsat(DPreducir(F, L));
11     end
12     if existe una literal unitaria L en F then
13         return DPsat(DPreducir(F, L));
14     end
15     L ← seleccionarVar(F);
16     return DPsat(DPreducir((F ∧ L) ∨ (F ∧ L)),O \ {L});
17 end
```

Algoritmo 1: Algoritmo de Davis-Putnam

2.1. Algoritmo secuencial

El siguiente es el algoritmo recursivo de Davis-Logemann-Loveland para resolver el problema de la satisfactibilidad proposicional:

Entrada: Una instancia F en forma normal conjuntiva y una asignación parcial de variables P
Salida: Si la instancia es satisfactible regresar `TRUE` y en caso de ser insatisfactible `FALSE`

```
1  Algoritmo:DPLLsat
2  begin
3      if F está vacío then                          /* Satisfactible */
4          return TRUE;
5      end
6      if F contiene una cláusula vacía then        /* Insatisfactible */
7          return FALSE;
8      end
9      if existe una literal pura L en F then
10         return DPLLsat(DPLLreducir(F, L), P ∪ L);
11     end
12     if existe una literal unitaria L en F then
13         return DPLLsat(DPLLreducir(F, L), P ∪ L);
14     end
15     PU(F, P);
16     L ← seleccionarVar(F);
17     if DPLLsat(DPLLreducir(F, L), P ∪ L) then
18         return TRUE;
19     else
20         return DPLLsat(DPLLreducir(F, ¬L), P ∪ ¬L);
21     end
22 end
```

Algoritmo 2: Algoritmo recursivo de Davis-Logemann-Loveland

En el Algoritmo 2, cuando F es satisfactible además de regresar $TRUE$ como respuesta, se asigna el contenido de la asignación parcial P con la que se satisface la fórmula a la asignación global de variables A para su futura ordenación y manejo de las variables que no fueron instanciadas.

Entrada: Una instancia F en forma normal conjuntiva, una asignación parcial de variables P y un contador de variables Q
Salida: Si F es satisfactible regresar `TRUE`, en caso contrario `FALSE`

```
1  Algoritmo:DPLLiter
2  begin
3      i ← 0, E[i] ← 0;
4      while i ≥ 0 do
5          PU (F_i,P_i);
6          if F_i contiene una cláusula vacía then     /* Insatisfactible */
7              i ← i − 1; continue;
8          if F_i está vacío then return TRUE ;        /* Satisfactible */
9          if E[i] = 0 then
10             if no hay variables en Q_i then
11                 i ← i − 1;
12                 continue;
13             else
14                 L ← seleccionarVar(F_i);
15                 E[i] ← 1;
16         if E[i] = 1 then valor ← TRUE;
17         if E[i] = 2 then valor ← FALSE;
18         if E[i] = 3 then
19             i ← i − 1;
20             continue;
21         if (E[i] = 1) || (E[i] = 2) then
22             E[i] ← E[i] + 1;
23             if valor = TRUE then
24                 P_i ← P_i ∪ L;
25                 DPLLreducir(F_i, L);
26             else
27                 P_i ← P_i ∪ ¬L;
28                 DPLLreducir(F_i, ¬L);
29             E[i + 1] ← 0;
30             i ← i + 1;
31     return FALSE;
32 end
```

Algoritmo 3: Algoritmo iterativo de Davis-Logemann-Loveland

Entrada: Una instancia F en forma normal conjuntiva, una lista de asignaciones parciales P
Salida: La instancia equisatisfactible[a] resultado de las modificaciones

```
1  Algoritmo:PU
2  begin
3      while exista una cláusula c ∈ C aún insatisfecha que contiene como máximo
       una literal l que no exista en P do
4          if cada literal en c ya tiene un valor asignado en P then
5              return error;
6          else
7              l = la variable en c que todavía no ha sido asignada a P; asignar l a P
               tal que la cláusula c quede satisfecha; DPLLreducir(F,l)
8          end
9          return P;
10     end
11 end
```

Algoritmo 4: Propagación unitaria

[a]Lógicamente equivalente.

Con el Algoritmo 3 iterativo, es posible agregar en un futuro *backtracking* no cronológico o *backjumping* o reinicios. Además permite mantener un control de las instancias que se van generando en cada nivel del árbol de búsqueda y con un manejo de memoria más eficiente que su contraparte iterativa.

El algoritmo iterativo recibe como entrada una instancia F en forma normal conjuntiva, una asignación parcial de variables P y un contador de variables Q. Una vez que se recibe (F, P, Q), cada cambio que se realice en un nivel i del árbol de búsqueda se almacena en un conjunto de cada uno de ellos (F_i, P_i, Q_i). El nivel i del árbol puede ir desde 1 hasta n. Cada nivel contiene un estado asociado E_i, el cual puede tener 4 valores $\{0, 1, 2, 3\}$. Cada estado indica la operación que se realizara en cada nivel cuando este sea recorrido o se realice el *backtrack*. Si $E_i = 0$ significa que el nivel i aun no tiene una literal asignada y hay que seleccionar una; en caso de que $E_i = 0$ y no haya variables por seleccionar, se realiza un retroceso en el árbol(*backtrack*). Si $E_i = 1$ significa que la reducción se realizara con la literal escogida L y si $E_i = 2$ con su negación $\neg L$. Por ultimo, si $E_i = 3$ significa que ya se procesaron ambas literales L y $\neg L$ si haber encontrado la satisfactibilidad, por lo que se retrocede un nivel en el árbol $i = i - 1$.

Entrada: Una instancia F en forma normal conjuntiva y una variable V
Salida: La instancia equisatisfactible resultado de las modificaciones

```
1  Algoritmo:DPLLreducir
2  begin
3      foreach cláusula C en F do
4          if A está en C y V[A]=TRUE o ¬A está en C y V[A]=FALSE then
5              eliminar C en F;
6          end
7          if A está en C y V[A]=FALSE then
8              eliminar A en C;
9          end
10         if (¬A está en C y V[A]=TRUE) then
11             eliminar ¬A en C;
12         end
13         return F;
14     end
15 end
```

Algoritmo 5: Función de reducción

Una instancia está vacía cuando no se encuentra ninguna cláusula dentro de ella; una cláusula está vacía cuando no tiene literales.

Una literal pura es aquella que no tiene negación dentro de una instancia dada. A la cláusula que contiene una sola literal se le llama cláusula unitaria, por lo anterior se le llama literal unitaria a esa única literal que se encuentra dentro de una cláusula.

Cuando se habla de reducción, nos estamos refiriendo a la aplicación de cada una de las reglas a una instancia dada.

Como se puede notar el algoritmo es naturalmente paralelizable, se puede aprovechar la regla de bifurcación para realizar el procesamiento en paralelo. Dividiendo el espacio de búsqueda, al asignar distintas ramas a los procesos, se consigue reducir el tiempo total de cómputo.

El algoritmo 1 tiene la desventaja de que al aumentar el número de variables de igual forma se incrementa el uso de memoria para manejar las instancias en cada nivel al realizarse una bifurcación.

Regla	Descripción
Cláusulas unitarias	Frecuente utilización después de la eliminación de literales.
Literal pura	Suele omitirse debido a la poca frecuencia de apariciones.
Literal positiva-negativa	Es muy poco frecuente en los primeros niveles del árbol de búsqueda.

Cuadro 2.1: Reglas de reducción.

2.1.1. Reglas de reducción

Las siguientes reglas son las descritas en el artículo de Davis-Logemann-Loveland [11], tienen la función de disminuir el tamaño de la instancia en cada aplicación. De todas ellas, la regla de la literal pura puede omitirse para mejorar el desempeño, ya que se ha demostrado que el tiempo que el algoritmo tarda en buscarla, por lo general es mayor que el tiempo que utilizarían las otras dos reglas en manejarla. Además de que es muy poco probable encontrar una variable pura en determinados momentos de la ejecución del algoritmo.

Eliminación de las cláusulas unitarias

Una cláusula es unitaria si contiene una sola literal en ella. Esta regla contiene a su vez varios incisos, cada uno de los cuales se aplican según concuerde con la regla.

1. Si una fórmula F contiene una cláusula unitaria con una literal l y también contiene otra cláusula unitaria con $\bar{l}$, entonces termina la ejecución del programa con resultado insatisfactible dado que F se contradice.

2. Si se encuentra una cláusula unitaria con l entonces se eliminan todas las cláusulas en donde aparezca la literal afirmada[3] y se elimina $\bar{l}$ de las cláusulas restantes. De la forma anterior obtenemos una fórmula F' equisatisfactible —también conocida como fórmula lógicamente equivalente—, es decir, que F' es insatisfactible si y solo si F lo es también.

[3] A una aparición de l sin una barra de negación se le llama ocurrencia afirmativa y a una con barra de negación se le llama ocurrencia negativa[12].

3. De igual manera que el inciso anterior, si se encuentra una cláusula unitaria con $\bar{l}$ entonces se eliminan todas las cláusulas en donde aparezca la literal en este caso negada y se elimina l de las cláusulas restantes, quedando la fórmula lógicamente equivalente F', que es consistente o inconsistente si y solo si F lo es también respectivamente.

A la aplicación sucesiva de los incisos de esta regla se le llama propagación unitaria.

Literal pura

Esta regla se puede omitir dado que alguna otra puede *atrapar* la literal pura, y generalmente con menor costo computacional. Esto es debido a que la literal pura se busca exhaustivamente en cada nivel y generalmente no se encuentra.

- Una literal pura es aquella que solo aparece una sola vez en la fórmula, si se encuentra alguna literal pura l se elimina la cláusula en donde aparece y también a su negación $\bar{l}$ en las cláusulas restantes.

Literal positiva-negativa

- Si una literal l aparece en una fórmula F solo en forma positiva o si l aparece solo en forma negativa, todas las cláusulas que contengan a l deben ser borradas. El resultado es una fórmula F' que es satisfactible si y solo si F lo es también y viceversa.

Eliminación de literales

También conocida como la regla de bifurcación, se aplica en caso de que la instancia no es afectada por ninguna de las reglas anteriores. Si siempre se diera el caso de que la instancia no es modificada por las reglas anteriores, el algoritmo sería como una versión más compleja de las tablas de verdad. A diferencia de las otra reglas, en ésta no se tiene un método especifico para elegir la variable a eliminar.

Se han desarrollado varias heurísticas para la bifurcación:

- Seleccionar literales con el máximo número de ocurrencias.
- Seleccionar literales con el máximo número de ocurrencias en las cláusulas pequeñas.

- Seleccionar literales que causen el mayor número de propagaciones unitarias.
- Seleccionar las literales aleatoriamente
- O hasta simplemente seleccionar las literales en forma consecutiva, aunque no con muy buenos resultados en la practica.

2.1.2. Procedimiento general del algoritmo

De igual manera el procedimiento para la utilización de las reglas de reducción está descrito en articulo de Davis-Logemann-Loveland [11].

Paso 1. Aplicar la regla para eliminar las cláusulas unitarias (Regla 1) a la instancia si es que la instancia contiene una cláusula con una sola literal, y seguir aplicando la misma regla a la fórmula hasta que no existan cláusulas unitarias en ella. Al ciclo anterior de la aplicación sucesiva de la regla 1 se le llama propagación unitaria. Si después de la propagación unitaria se tiene como resultado una fórmula vacía, entonces la fórmula es satisfactible. Por el contrario si el resultado después de la propagación unitaria es una fórmula aún con cláusulas —estas ya con más de una literal—, entonces se aplica la regla 2.

Paso 2. Aplicar la regla de la literal positiva-negativa (Regla 2) a la fórmula resultante del paso 1 o a la fórmula inicial en caso de que no se aplique la regla 1, a menos que cada una de las variables que aparezcan en la fórmula estén en forma positiva y negativa. Si como resultado obtenemos una cláusula unitaria, entonces regresamos al paso 1. Por el contrario si existe otra literal que aparezca solo afirmada o negada entonces volver a aplicar este paso 2. Si como resultado tenemos una fórmula vacía, la instancia es satisfactible. Hasta este punto, si la fórmula todavía contiene cláusulas con más de una literal en ellas y si cada una de las literales aparece en forma tanto positiva como negativa, entonces ir al paso 3.

Paso 3. A la fórmula que resulto después de haber aplicado las reglas anteriores hay que aplicar alguno de las métodos de selección de literales para obtener una. Después de esto se le aplicará la regla 3 a la fórmula por separado, con la literal elegida y a la misma literal negada. De esta forma se realiza la bifurcación de la ejecución —que posteriormente se aprovechará para la

versión paralela—. Si después de aplicar este paso —con las dos literales— alguna de las fórmulas está vacía (no contiene cláusulas) entonces la instancia es satisfactible.

Despues del paso 3 continuar aplicando las reglas 1, 2 y 3; hasta que se obtenga un resultado satisfactible de alguna de las reglas. Al final, si se llega a la regla 3 y se da el caso de que ya no existan más literales por escoger, entonces la instancia es insatisfactible.

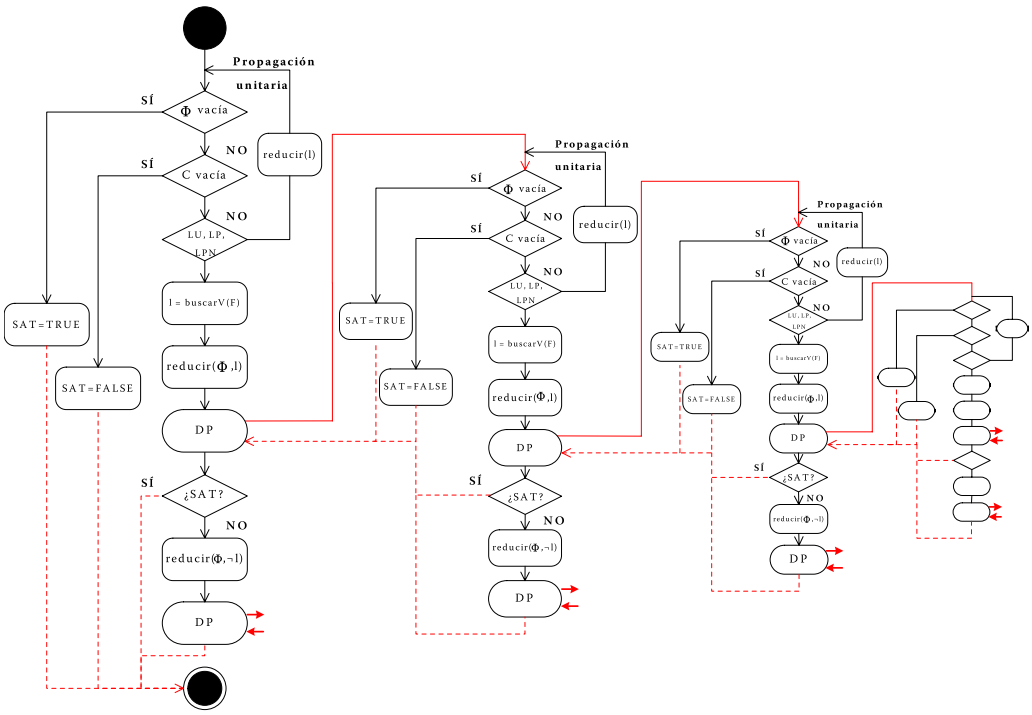


Figura 2.1: Diagrama de actividades del algoritmo secuencial recursivo

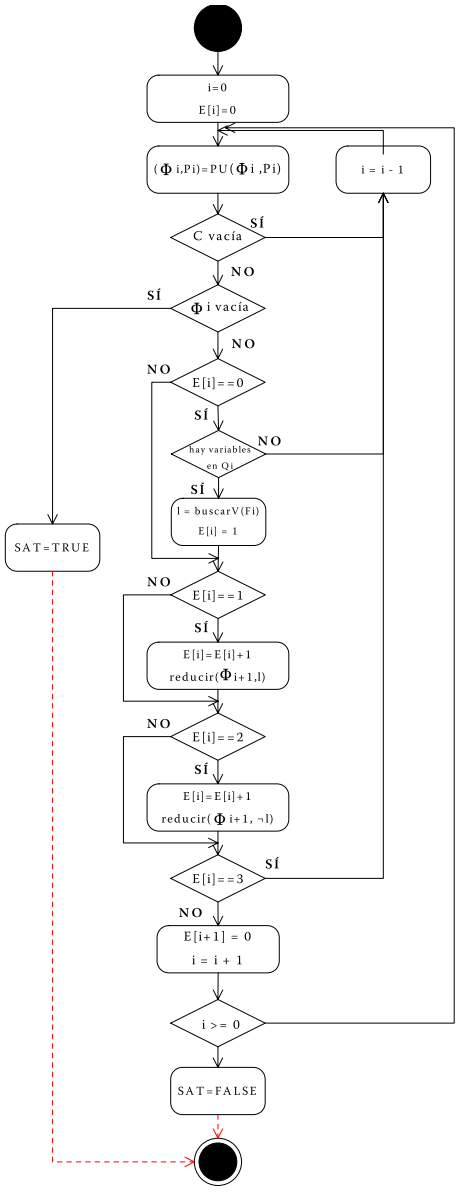


Figura 2.2: Diagrama de actividades del algoritmo secuencial iterativo

2.2. Algoritmo paralelo

La paralelización inicial está basada en la bifurcación del algoritmo de Davis-Putnam, en la cual se divide la instancia en dos partes. En cada parte se efectúa la reducción con la variable escogida.

La paralelización también está basada en el procesamiento de la instancia original con diferentes variables en cada proceso.

Conforme se desarrollen más y mejores heurísticas de solución de SAT, podrán resolverse problemas de satisfactibilidad más complejos, encontrando más áreas de aplicación.

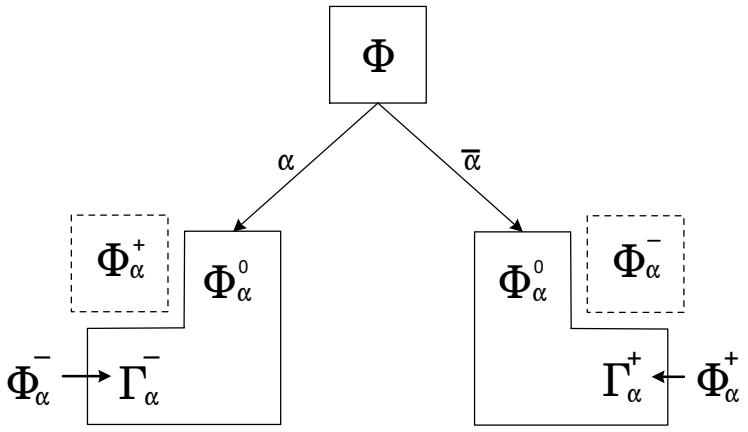


Figura 2.3: Reducción de la instancia

Si Φ es un conjunto de cláusulas C y l es una literal, definimos:

$$\Phi_l^+ = \{c \in \Phi | l \text{ aparece en } c\}$$

Son las cláusulas de Φ donde aparece l.

$$\Phi_l^- = \{c \in \Phi | \bar{l} \text{ aparece en } c\}$$

Son las cláusulas de Φ donde aparece $\bar{l}$.

$$\Phi_l^0 = \{c \in \Phi | l, \bar{l} \text{ no aparece en } c\}$$

Son las cláusulas de Φ donde no aparece l ni $\bar{l}$.

$$\Phi \equiv \Phi_0^+ \wedge \Phi_0^- \wedge \Phi_0^0$$

$$\forall x \in Xn(x) = \text{ número de ocurrencias de } x \text{ en } \Phi$$

En cada nivel se pregunta si se satisface la fórmula equisatisfactible, mientras no se satisfaga se continua con un nuevo nivel del árbol de búsqueda, para todas las variables $\alpha_1, \alpha_2, \alpha_3, ..., \alpha_n$. En caso de que terminemos con las variables y la instancia no encuentra una asignación de valores válidos se considera que la fórmula es insatisfactible.

La instancia es satisfactible cuando $|\Phi_\alpha^0 \cup \Gamma_\alpha^-| = 0$ para la literal α o bien $|\Phi_\alpha^0 \cup \Gamma_\alpha^+| = 0$ para la literal $\bar{\alpha}$

$$\Gamma_\alpha^- = \{c - \{\bar{\alpha}\}| \quad \forall \quad c \in \Phi_\alpha^-\}$$

$$\Gamma_\alpha^+ = \{c - \{\alpha\}| \quad \forall \quad c \in \Phi_\alpha^+\}$$

En la Figura 2.4 se ilustra la forma en que es reducida una instancia, se tiene como entrada una instancia Φ, a la que inicialmente se le aplica la regla de reducción número III, para eso se selecciona una literal α por medio de alguna heurística. Cuando se aplica la regla III, la literal escogida es asignada a un conjunto de asignaciones parciales y la instancia queda reducida en tamaño —y teóricamente también reducida en complejidad. La regla III se aplica a la literal α y a su negación $\neg\alpha$, con lo que se obtienen dos instancia equisatisfactibles ($\Phi_1 = \Phi - \{\alpha\}$ y $\Phi_2 = \Phi - \{\neg\alpha\}$), esto es si la instancia original Φ es satisfactible así también lo serán las dos instancias resultantes. Las demás reglas de reducción se aplican recursivamente[4] a cada una de las instancias para reducir aun más el problema.

En la figura 2.4 si Φ es insatisfactible, también serán insatisfactibles Φ_1 y Φ_2, pero si Φ es satisfactible no es posible saber el resultado de Φ_1 y Φ_2 a menos que alguno de ellos no contenga cláusulas—en cuyo caso el algoritmo habrá terminado. Cuando Φ_1 y Φ_2 contengan ambos la cláusula vacía $\square$ el algoritmo termina con la insatisfactibilidad como respuesta. pueden ser o bien ambos satisfactibles o bien solo uno de ellos. Si solo alguna de las instancias resultantes Φ_1 o Φ_2 contiene la cláusula vacía $\square$, ésta deja de procesarse y solo se continua con la instancia restante. Por ultimo si alguna de las fórmulas Φ_1 o Φ_2 tienen un conjunto vacío de cláusulas $\emptyset$ el algoritmo termina dado que la fórmula es satisfactible.

[4]Propagación unitaria(PU).

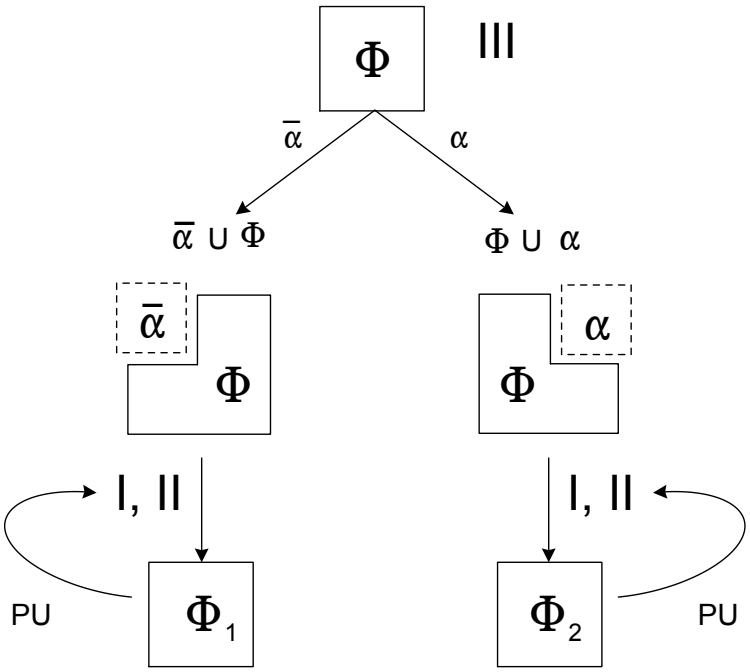


Figura 2.4: Aplicación de las reglas de reducción

2.2.1. Elementos de paralelización

El primer intento de paralelización se realizó como se ilustra en la Figura 2.5, que no es más que la ejecución en paralelo del mismo algoritmo secuencial recursivo con la misma instancia φ para todos los procesos hijo, únicamente varía la heurística que utilizaran cada uno de ellos. No es una forma eficiente, pero de esa forma se puede probar que heurística tiene un mejor desempeño y el tiempo total no puede ser mayor que el de la versión secuencial.

La paralelización posterior, que se muestra en la Figura 2.6, usó otro enfoque, en el que se divide el espacio de búsqueda entre los procesos que se ejecutaran en paralelo. Para ello primero la instancia φ que se recibe, es dividida en n instancias diferentes, para cada uno de los n procesos que se ejecutarán en paralelo. La división se realiza usando el algoritmo secuencial recursivo que usa las reglas de reducción sobre la instancia φ y a medida que va avanzando por los niveles del árbol de búsqueda va generando las instancias equisatisfactibles que tienen menor complejidad que φ.

Para la ejecución en 2 procesadores con ese enfoque, si φ tiene n variables, se pueden tener hasta 2^n posibles asignaciones, por lo que dividir el árbol en 2 sería

$$\frac{2^n}{2} = \frac{2^n}{2^1} = 2^{n-1}$$

esto es eliminar una variable n-1 y para eliminar una variable la tenemos que instanciar, por lo que si se usan 2 procesos cada uno de los procesos recibirá una instancia con un conjunto de cláusulas C en donde la variable x_1 ha sido instanciada $\varphi_1 = C(x_1)$ y $\varphi_2 = C(\neg x_1)$.

Una vez que se tienen a φ_1 y φ_2, estos se envían a los procesos hijo que continuaran con la ejecución del algoritmo DPLL, si se usa el algoritmo iterativo se tiene la ventaja de que con algunas modificaciones acepte alguna de las técnicas mas recientes como la de *backtracking* no cronológico o reinicios del algoritmo.

Si bien a φ_1 y φ_2 φ_1 y φ_2 les faltan las mismas variables por asignar, no significa que terminaran al mismo tiempo, la dificultad de una de las ramas puede ser mucho mayor que la otra, por lo que si la rama asignada a uno de los procesos hijo termina $FALSE$ como resultado, el proceso padre pedirá al otro proceso que regrese la instancia que esta procesando junto con su asignación parcial de variables, para que se vuelva a dividir el árbol con la finalidad de que un procesador quede ocioso.

Finalmente esta la paralelización usando la heurística de selección de variables, que se refiere a la utilización de una misma heurística para todos los pro-

cesos hijo, pero la búsqueda se hace en paralelo, al igual que la versión anterior mostrada en la Figura 2.6 también se realiza la división de la instancia de entrada, y se asignan a cada uno de los procesos hijo, en donde ahora cuando se necesite utilizar la regla de bifurcación, primero se verifica si alguno de los procesos ya realizó la búsqueda de una variable para dicho nivel, en cuyo caso solo se realiza la reducción con dicha literal, en caso de que ningún proceso lo haya hecho, éste realiza la búsqueda de una literal y la almacena en la lista global. Con esto se pretende reducir el tiempo usado en la búsqueda de variables, uno de los factores más importantes que influyen en el tiempo total de cómputo del algoritmo.

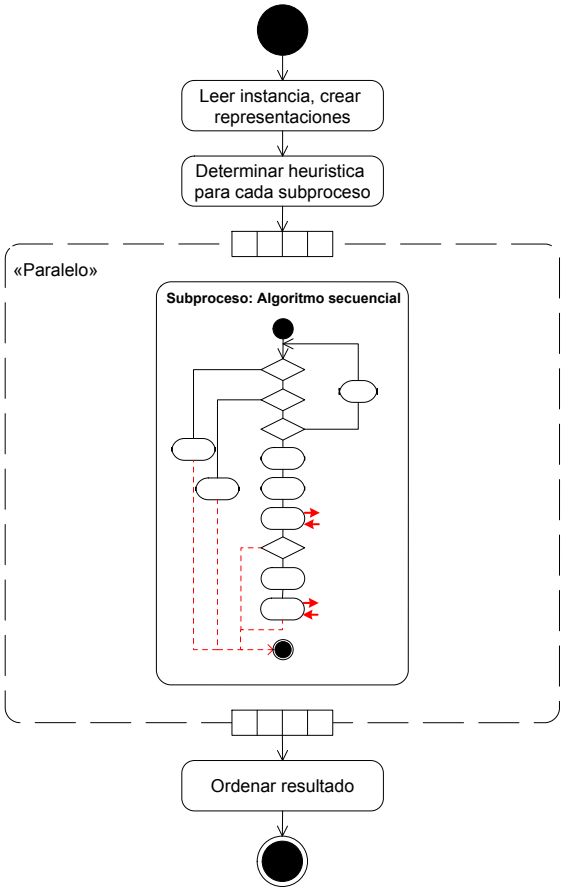


Figura 2.5: Paralelización 1

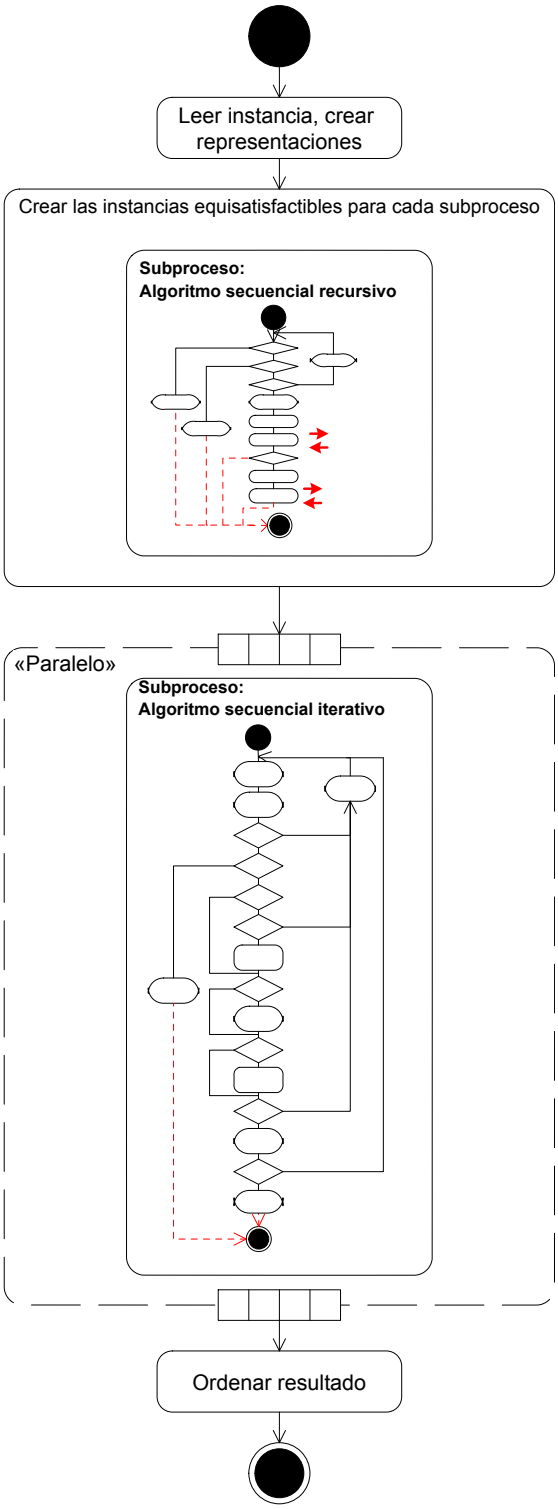


Figura 2.6: Paralelización 2

Capítulo 3

Tratamiento computacional

En años recientes los solucionadores del SAT han tenido grandes mejoras. Dicho progreso ha permitido desarrollar y resolver muchas instancias, que hasta hace poco eran computacionalmente intratables.

La satisfactibilidad proposicional es un problema NP-completo bien conocido, de importancia teórica y practica, con amplias aplicaciones en muchos campos de las ciencias de la computación e ingenierías, como son la inteligencia artificial y diseño electrónico por poner un ejemplo.

Los programas para resolver el problema de la satisfactibilidad incorporan técnicas complejas para reducir el espacio de búsqueda y para dirigirla. Algunos ejemplos de estrategias para la búsqueda son la búsqueda weak-commitment, el reinicio de la búsqueda y el *backtracking* aleatorio.

Recientemente se han desarrollado técnicas avanzadas para los algoritmos del SAT que utilizan *backtracking*, que han logrado grandes mejoras, y han mostrado cuan importante son para resolver instancias difíciles de SAT obtenidas de aplicaciones del mundo real.

Desde un punto de vista practico, los algoritmos más eficaces son los completos, además de que son capaces de probar lo que los algoritmos locales no consiguen, la insatisfactibilidad. Esto puede ser muy importante, sobre todo si no se conoce con anticipación si una instancia es satisfactible o no.

Sin embargo, está ampliamente aceptado que la búsqueda local puede a menudo tener claras ventajas con respecto a la búsqueda en *backtrack*, dado que le está permitido reiniciar la búsqueda desde cualquier punto cada vez que queda atrapado en una solución óptima local.

Esta ventaja de los algoritmos de búsqueda local ha motivado el estudio de enfoques para hacer más flexible las condiciones de retroceso en algoritmo de

backtracking. La idea central es que sin restricciones elegir el punto hasta el que se realizará el retroceso, para evitar el recorrido hasta el punto en que se pretende regresar.

Además, uno puede pensar en combinar diferentes formas para hacer reglas más flexibles para elegir el punto al cual retroceder.

El que un algoritmo sea completo no quiere decir que muestre todas las soluciones en caso de que ésta sea satisfactible, sino que en caso de que la instancia no tenga solución debe probar que la fórmula es insatisfactible, a diferencia de los algoritmos incompletos que solo pueden garantizar la satisfactibilidad de una instancia.

Se dice que una cláusula está *satisfecha* si al menos una de sus literales tiene 1 como valor, e *insatisfecha* si todas sus literales tienen valores de 0; y en en caso de que aun no se hayan asignado valores a las variables en dicha cláusula, esta aun está por resolver o en estado *irresoluto*.

Las literales a las que no se les ha asignado un valor de verdad son conocidas como *literales libres*. Una fórmula se dice estar satisfecha cuando todas sus cláusulas han sido satisfechas, e insatisfecha si al menos una cláusula es insatisfecha.

Una asignación de verdad en una fórmula es un conjunto de variables que han sido asignadas con sus respectivos valores de verdad. El problema SAT consiste en decidir si existe una asignación de verdad para las variables, tal que la fórmula se satisfaga.

Los algoritmos para SAT se pueden distinguir entre los completos e incompletos. Los algoritmos completos pueden comprobar la insatisfactibilidad de una fórmula, si se le da suficiente tiempo de CPU para que termine; en cambio los algoritmos incompletos al terminar sin encontrar una asignación que satisfaga la instancia, no pueden darnos la certeza de que el algoritmo sea insatisfactible.

Por consiguiente los algoritmos incompletos suelen usarse solamente para instancias satisfactibles. Algunos ejemplos de algoritmos completos e incompletos son los algoritmos de búsqueda con *backtracking* y de búsqueda local, respectivamente.

En el contexto de la búsqueda, los algoritmos completos son a menudo llamados sistemáticos, en tanto que los algoritmos incompletos son conocidos como no sistemáticos.

En años recientes se han creado una gran cantidad de algoritmos para resolver el problema SAT, la mayoría con mejoras de la versión original de Davis-Putnam, algoritmos utilizando *backtracking* hasta algoritmos de búsqueda local.

La gran mayoría de los algoritmos SAT que usan *backtrack*, han sido construi-

dos a partir del algoritmo original de *backtrack* propuesto por Davis, Logemann y Loveland.

El algoritmo de búsqueda en *backtrack* básicamente consiste en un proceso de búsqueda que recorre el espacio de 2^n asignaciones posibles, donde n es el número de variables.

Cada nueva asignación define una nueva ruta en el espacio de búsqueda. Un nivel de decisión consiste en la selección de una variable y la asignación de uno de los dos posibles valores.

La primera selección de una variable se realiza en la raíz del árbol de búsqueda, esto es, en el nivel 1, cada nivel de decisión aumenta según la cantidad de nuevas selecciones de variables y sus consiguientes asignaciones, por lo tanto cada árbol de búsqueda puede tener a lo más $\log_2 n$ niveles, donde n es la cantidad de variables. Puede darse el caso de que la instancia se satisfaga aun cuando no todas las variables hayan sido instanciadas.

Adicionalmente en cada nivel de decisión, se aplica la regla de la cláusula unitaria y opcionalmente también se puede aplicar la regla de la literal pura. Si una cláusula es unitaria, entonces

3.1. Análisis de las instancias

Una cantidad de variables o cláusulas en una instancia no necesariamente implica que ésta sea más difícil de resolver. Como se verá más adelante el tiempo que un programa se tarde en resolver una instancia está más ligado a la relación entre variables y cláusulas, que al número de ellas.

Existen básicamente dos tipos de instancias, primero están aquellas que son tomadas de problemas del mundo real, la mayoría de las cuales son relativamente fáciles de resolver y su complejidad radica en la cantidad de variables que contiene cada instancia. Y por otro lado se encuentran aquellas instancias generadas aleatoriamente —llamadas k-SAT o k-CNF—, a partir de tres parámetros (V, C, K), donde V es el número de variables, C el número de cláusulas y K es el número de literales por cláusulas. Cada instancia se forma con un conjunto aleatorio de cláusulas con una longitud exacta de K literales.

Koutsoupias y Papadimitriou demostraron que para instancia con $K = 3$ es relativamente fácil de satisfacer para un algoritmo de búsqueda local cuando la instancia tiene una solución [21]. Sin embargo David Mitchell, Bart Selman y Hector Levesque demostraron experimentalmente que las instancias generadas aleatoriamente son difíciles de resolver cuando tiene la misma probabilidad de

K	Proporción
3	4.24
4	9.88
5	21.05
6	43.31
7	87.70
8	176.41
9	353.88
10	708.78
⋮	⋮
20	726816.49

Cuadro 3.1: Proporciones de instancias aleatorias balanceadas k-SAT

ser satisfactible que insatisfactible [26]. La relación entre cláusulas y variables (C/V) para instancias difíciles de resolver varía según K. Para ($K = 3$) los problemas más difíciles se encuentran cuando la proporción C/V está entre 4,24 y 6,21 ($C/V = [4, 24.,6, 21]/V$). La mayoría de los *benchmarks* utilizan dicha proporción para la realización de sus pruebas.

Como se puede observar en el Cuadro 3.1 al incrementarse la cantidad de literales por cláusula se necesitan más cláusulas por cada variable si se desea crear una instancia difícil de resolver.

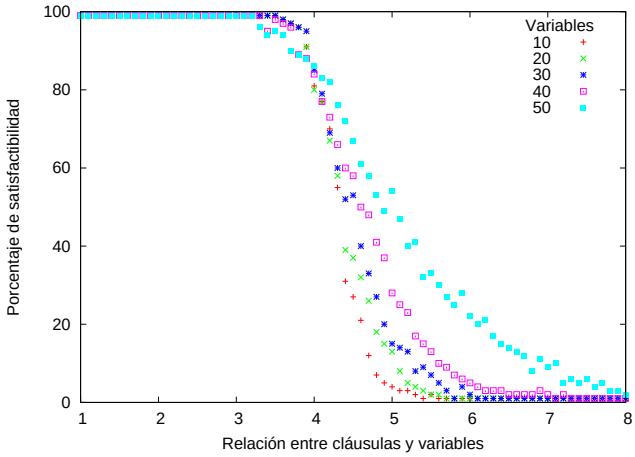


Figura 3.1: Relación entre cláusulas y variables, y el porcentaje de satisfactibilidad

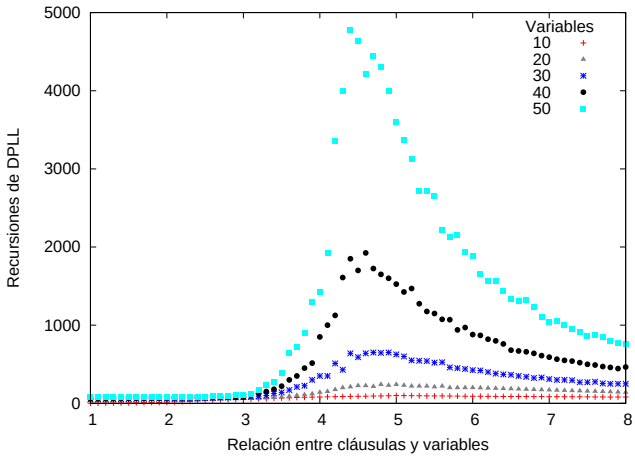


Figura 3.2: Proporción entre cláusulas y variables y su relación con el número de recursiones del algoritmo

3.2. Preprocesamiento

Estas son las tareas que se ejecutan previo a la ejecución del algoritmo y como primer tarea se tiene la lectura de la instancia en formato CNF, esa instancia inicialmente se guarda en la representación matricial. Cuando se ha leído la instancia completa se procede a crear las otras representaciones de los datos: la representación por literales y la representación por cláusulas.

Por defecto las variables que necesita la regla 3 son escogidas en el punto del árbol de búsqueda donde se realiza la bifurcación, pero otra de las tareas que se realizan en este punto es la creación de las listas de variables —en caso se necesita especificarlo como un parámetro— que será utilizada por la regla 3 y que en la versión secuencial es una sola lista, pero que en las versiones paralelas se crean varias listas, una por cada proceso, estas listas pueden contender los mismos o distintos valores —otra opción que también se debe especificar como parámetro—, si se eligen distintos valores, las listas se llenaran con los valores que den los diferentes métodos de selección de variables con los que se cuenta, en caso de que sean más procesos que métodos de selección de variables se volverá a seleccionar los métodos iniciales, de esta forma algunos de las listas serán idénticas —excepto para el método de selección de variables aleatorio—, pero tendrán un comportamiento distinto dado que se aplicarán sobre instancias equisatisfactibles distintas.

3.2.1. Ordenamiento estático de variables

El ordenamiento estático de variables consiste en asignar la misma variable en cada nivel del árbol de búsqueda, esto se logra ordenando previamente las variables dentro del preprocesamiento que realiza el proceso maestro, y este arreglo de variables se pasa a cada uno de los hijos que realiza la selección secuencial de las variables dentro del arreglo.

Cabe hacer mención que si bien es la misma variable en cada nivel, en la bifurcación se asignan una literal afirmada y otra negada para la misma variable.

Dentro del ordenamiento estático se propuso organizar a las variables que tienen más apariciones tanto positivas como negativas $|card(x_i^+) + card(x_i^-)|$ y que minimiza $|card(x_i^+) - card(x_i^-)|$, para toda i desde 1 hasta n.

Métodos de selección tal como el MOMS eligen a la literal según el número de apariciones que tenga en una cláusula. De esta forma al aplicar la regla de la eliminación de literales con este método se pretende reducir el tamaño del problema

considerablemente. Sin embargo este método no garantiza que la literal elegida sea la que reduzca más el tamaño del problema.

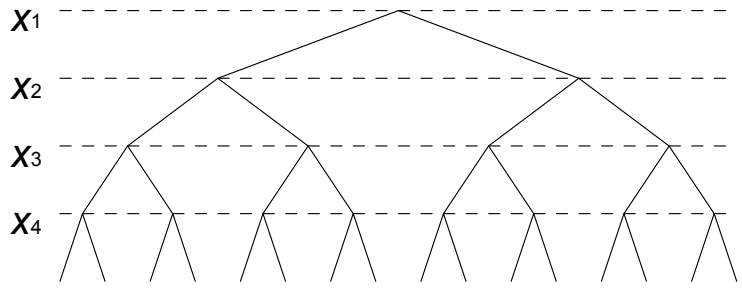


Figura 3.3: Ordenamiento estático de variables

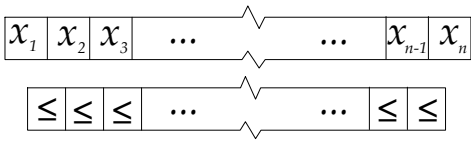
Cuando se elige una literal con este método, éste elimina todas las cláusulas en donde aparece, pero no garantiza una eliminación en la misma proporción de las apariciones negativas de la misma literal en el resto de las cláusulas.

$$\alpha = |card(x_i^+) + card(x_i^-)|$$

$$\beta = |card(x_{i+1}^+) + card(x_{i+1}^-)|$$

$$\alpha \geq \beta$$

(a) Ecuación para ordenar las literales.



(b) Representación gráfica.

$(|card(x_i^+) - card(x_i^-)|) \leq (|card(x_{i+1}^+) - card(x_{i+1}^-)|)$ cuando $\alpha = \beta$

Figura 3.4: Ordenamiento por modificación de MOMS.

3.2.2. Ordenamiento dinámico de variables

El ordenamiento dinámico de variables se da cuando existen diferentes variables en un mismo nivel del árbol de búsqueda, y en nuestro caso se da cuando el proceso maestro asigna a los procesos hijos una parte del árbol de búsqueda y cada hijo decide que variable elegir en cada nivel, por lo que el mismo nivel puede ser diferente en todos los procesos hijo.

Como ejemplo del ordenamiento dinámico está el método UP que en cada bifurcación debe realizar un preprocesamiento para detectar que variables realizan la mayor cantidad de propagaciones unitarias.

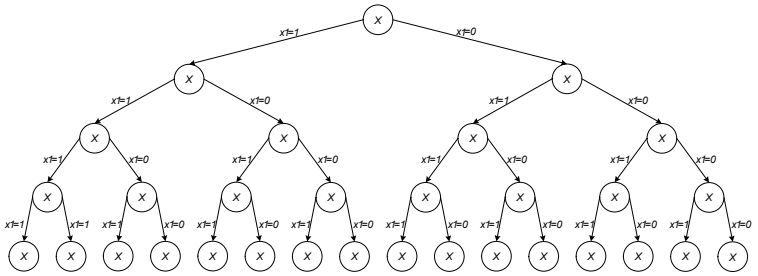


Figura 3.5: Árbol de búsqueda

Aunque el método puede utilizarse en el procesamiento y con eso generar una lista de literales estática, es conveniente utilizarlo en forma dinámica ya que la instancia va cambiando con cada aplicación de las reglas. Es aquí en donde entra la disyuntiva de elegir la forma de utilizar los métodos, la forma estática tiene la ventaja de que no gasta el tiempo de preprocesamiento por cada bifurcación y la dinámica tiene a su vez la ventaja de reducir la instancia con cada literal elegida.

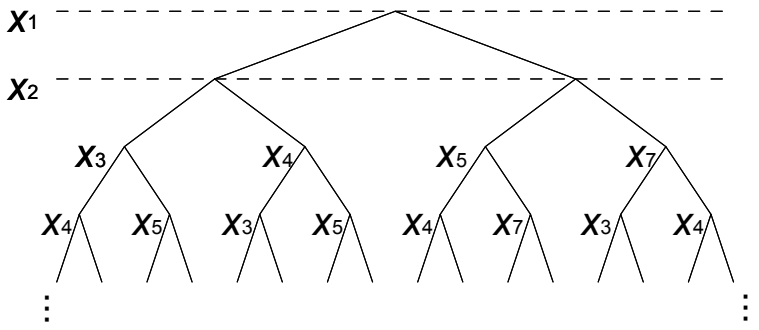


Figura 3.6: Ordenamiento dinámico de variables

3.3. Versión secuencial

Esta versión nos sirvió de base para las pruebas con varias heurísticas de selección de variables. Los algoritmos paralelos que se ejecutan en los procesos hijo de las versiones posteriores con hilos y MPI, prácticamente son copia fiel de esta versión, conteniendo solo algunas modificaciones.

La versión secuencial del algoritmo fue adoptando nuevos cambios, se comenzó con una implementación con las especificaciones iniciales del algoritmo, hechas por Davis y Putnam, se utilizó una matriz para representar la instancia y no hubo modificación a la heurística propuesta por ellos. Esa versión inicial tuvo varias modificaciones hasta obtener mejoras significativas agregando otras formas de representación de la instancia, que se complementara con la matriz ya existente y se realizaron pruebas con dos heurísticas nuevas —la de selección aleatoria y secuencial—. Una vez hechos estos cambios se modificaron las heurísticas ya existentes y los módulos para que aprovecharan los cambios hechos en la representación de los datos. Y finalmente se implementaron nuevas heurísticas de selección de literales.

3.4. Versión utilizando hilos

En sistemas UNIX® se desarrolló una interfaz de programación en lenguaje C estándar para la creación de hilos de ejecución, según lo especificado por la norma IEEE POSIX 1003.1c y los sistemas que se basan en ella son conocidos como *POSIX threads* o *PThreads*.

La razón por la que se desarrollo el programa con hilos, es porque estos pueden ser una gran ventaja, dado que aprovechan su máximo potencial, al ser utilizados en una arquitectura o ambientes de memoria compartida con varios procesadores, porque de esta manera pueden ser usados para llevar a cabo el paralelismo. Al utilizar memoria compartida se evitan los gastos en comunicaciones de otros sistemas distribuidos.

De inicio, con la versión con hilos se pretendía aprovechar las ventajas que nos ofrece el trabajar con la memoria compartida, eliminando la necesidad de costos computacionales como el tiempo de comunicación entre procesos, como es en el caso de la utilización de una solución que involucre la utilización de memoria compartida.

Antes de realizarse la bifurcación se debe escoger una variable sobre la cual aplicar las transformaciones. Las transformaciones se llevan a cabo en cada uno

de los hilos creados a partir de la bifurcación, en ellos se realizan las transformaciones utilizando la variable seleccionada y la misma variable pero negada.

El beneficio de la división del trabajo solo se produce cuando la instancia es lo suficientemente compleja, dado que la creación de hilos y la comunicación entre ellos requiere un costo en tiempo.

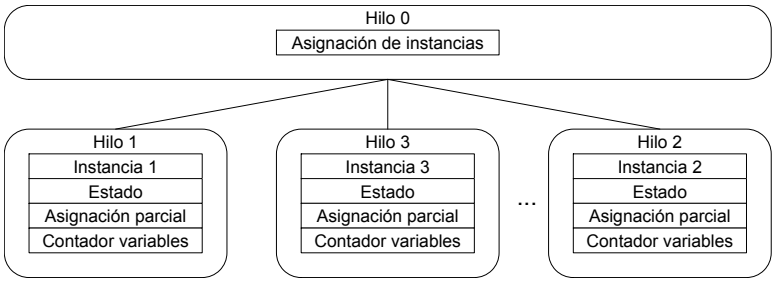


Figura 3.7: Esquema de asignación de tareas

La paralelización se realiza mediante la asignación de una rama de búsqueda del árbol a un hilo. El hilo solo termina cuando realiza la búsqueda por la rama que se le asigno, de esta forma se evita tener una gran cantidad de comunicaciones, cada procesador tiene su propio árbol de búsqueda y solamente se comunica con el proceso maestro para averiguar si una parte del árbol ya ha sido resuelta por otro proceso.

3.5. Versión con MPI

Esta versión fue escrita con el lenguaje C usando MPI con el conjunto de funciones de LAM/MPI, LAM (*Local Area Multicomputer*) es una implementación en código abierto del estándar de la Interfaz de Paso de Mensajes, MPI por sus siglas en inglés (*Message Passing Interface*).

Aun y cuando la versión con hilos tiene la ventaja de no tener gastos en comunicaciones, en la actualidad no son muy comunes sistemas que utilicen muchos procesadores con memoria compartida. Esta versión tiene la ventaja frente la versión con hilos, que no es necesario tener una plataforma de memoria compartida. Entonces la versión en MPI eliminó la necesidad de trabajar en equipos con ciertas características, como puede ser la obligatoriedad de utilizar una computadora con un solo procesador o varios procesadores con memoria compartida, como es el

caso de las versiones anteriores respectivamente. Todas las ventajas anteriores superan la desventaja del costo en comunicaciones que implica realizar un programa que utilice la librería MPI.

Para la paralelización se aprovecha la regla de bifurcación, en donde la instancia se divide en dos al asignar diferentes valor a una variable escogida previamente. Durante la bifurcación se divide la carga de trabajo en diferentes hilos.

Antes de realizarse la bifurcación se escoge una variable sobre la cual aplicar las transformaciones. Las transformaciones se llevan a cabo en cada uno de los hilos creados a partir de la bifurcación, en ellos se realizan las transformaciones utilizando la variable seleccionada y la misma variable pero negada.

El beneficio de la división del trabajo solo se produce cuando la instancia es lo suficientemente compleja, dado que la creación de hilos y la comunicación entre ellos requiere un costo en tiempo.

3.6. Estrategias de solución

La paralelización se realiza mediante la asignación de una rama de búsqueda del árbol a un hilo. El hilo solo termina cuando realiza la búsqueda por la rama que se le asigno, de esta forma se evita tener una gran cantidad de comunicaciones, cada procesador tiene su propio árbol de búsqueda y solamente se comunica con el proceso maestro para averiguar si una parte del árbol ya ha sido resuelto por otro proceso.

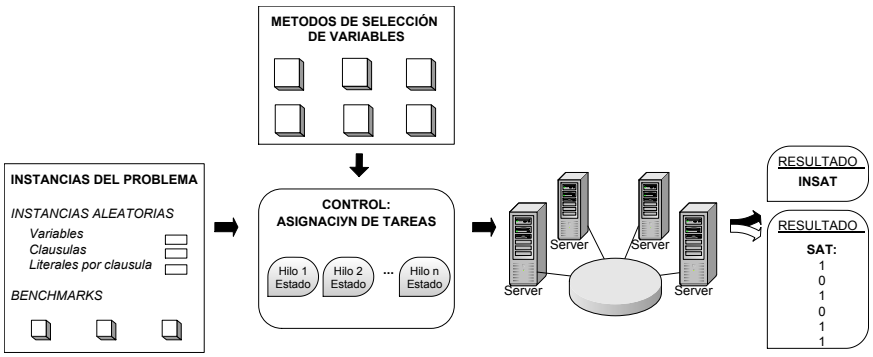


Figura 3.8: Esquema de paralelización

En la estructura de datos se utilizan dos formas de representar la instancia, una

se basa en las variables y la otra en las cláusulas, con la finalidad de disminuir el tiempo de búsqueda de la posición de una cláusula o literal dada.

Se desarrollaron diferentes versiones paralelas del algoritmo paralelo, ambas con hilos:

Versión con hilos en el cual la paralelización consiste en asignar a cada hilo ejecuta el algoritmo de Davis-Putnam sobre la instancia original, variando solamente la elección de la variable en la bifurcación.

Versión con hilos en el cual la paralelización se realiza mediante la división del árbol de búsqueda, de esta forma cada hilo ejecuta un instancia equisatisfactible, es decir una instancia que se satisface con los mismos valores de variables que la instancia original. Después de la bifurcación los hilos reciben una instancia y una variable con la cual se realizaran las transformaciones necesarias utilizando las reglas de reducción del algoritmo.

Versión con hilos en la que al igual que la anterior, los hilos reciben una instancia satisfactible diferente a los otros y una variable con la cual se realizaran las transformaciones necesarias utilizando las reglas de reducción del algoritmo. Sin embargo en esta versión en el hilo 0 existe un arreglo con las literales que han sido encontradas hasta el momento, y cada hilo al tener que aplicar la regla de bifurcación verifica si en el hilo padre existe una literal que no exista en su asignación parcial de variables que han sido instanciadas. Si existe esa literal, entonces se realiza la bifurcación con ella, si no, entonces se busca una literal con alguna de las heurísticas existentes, y cuando se encuentra, además de realizar la bifurcación con ella, se inserta en el arreglo del hilo padre. Con esto se tienen mejores resultados que si solo se realiza la asignación con variables estáticas y también se evita emplear mucho tiempo en la búsqueda de las variables.

3.7. División del árbol de búsqueda

La división del árbol de búsqueda se realiza en el instante en que se necesite aplicar la regla de la bifurcación, siempre y cuando exista disponibilidad de recursos, en este caso del procesador. Dado que de nada nos sirve asignar trabajos a los procesadores que están realizando otra tarea.

Mediante un procesamiento el proceso padre decide como realizar división inicial, que se le pasara a los procesos hijo. También existe la posibilidad de que se realicen otras bifurcaciones cuando no habiendo decidido si una instancia es satisfactible, termina uno de los hijos con su tarea asignada, por lo que se le asigna otra tarea; esto es muy importante en las instancias particularmente grandes,

porque si se realiza esto con instancias pequeñas el resultado puede ser contraproducente, ya que se necesita un procesamiento para decidir que parte del árbol reasignar y de que otro hijo. Y en instancias pequeñas el tiempo de este proceso puede ser mayor que el que nos podríamos ahorrar.

3.8. Métodos de selección de variables

Los algoritmos usados para la selección de variables fueron el secuencial, aleatorio, MAMS, MOMS, MOMS*, MAXO, UP y SUP, además de algunas combinaciones de ellos. Más adelante se profundizara en cada uno de ellos.

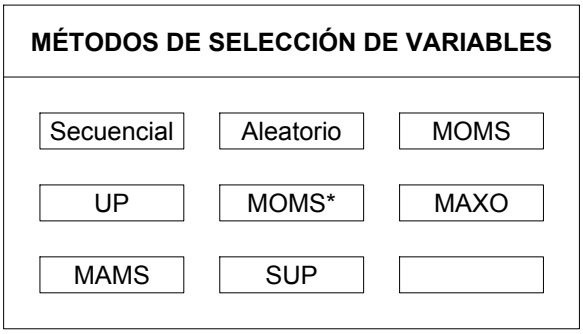


Figura 3.9: Métodos de selección de variables

3.9. Paralelización por búsqueda en orden de variables predefinidas

Una forma de trabajar con las heurísticas fue realizar un preprocesamiento con la instancia inicial, con la que se creo un orden preestablecido de las variables que se asignan a los hijos. Con esto se evita el calcular en cada bifurcación la variable a elegir. Sin embargo esto puede no siempre dar los mejores resultados.

La figura 3.10 y la figura 3.11 ejemplifican la forma en que el tiempo es destinado para las áreas principales del programa. En la figura 3.10 el área gris corresponde al preprocesamiento, durante el cual se realiza la lectura del archivo .cnf que contiene la instancia. Todas las cláusulas del archivo son leídas y asignadas

en un inicio a un arreglo bidimensional. Una vez que son leídas todas las cláusulas es realizada la conversión a las otras representaciones.

Posteriormente empieza la aplicación de las reglas de reducción —representado por el cuadro azul con la letra R— a la instancia inicial y en caso de ser necesaria la utilización de la regla de bifurcación se utilizara el método de selección de variables que se haya elegido —que está representado como la segunda parte del recuadro azul—. Esta forma de seleccionar las variables puede ser conveniente en ciertos casos puesto que la instancia va cambiando con cada aplicación de las reglas y con ciertas heurísticas se puede sacar provecho de utilizar la instancia actual. Pero también existe el inconveniente de que se debe utilizar tiempo adicional para seleccionar cada una de las variables por cada bifurcación.

El cuadro rojo indica que al aplicarse las reglas se encontró una contradicción en la rama que se estaba procesando, esto puede ser porque quedaron dos cláusulas unitarias con la misma literal con signo opuesto o porque se encontró una cláusula vacía. Cuando se encuentra una contradicción en una rama, ésta simplemente deja de procesarse.

Los cuadros en blanco indican el tiempo que dura procesar alguna de las dos ramas que se crean con la bifurcación. Los dos cuadros rojos contiguos indican que en una bifurcación ya no existen más literales por escoger, lo que significa que se ha llegado al final de esa rama, esto es de lo más costoso ya que significa que se recorrió toda una rama sin encontrar ninguna solución ni alguna contradicción que permitiera acortar la búsqueda.

Finalmente un cuadro negro representa que se ha encontrado el valor, que unido con los encontrados anteriormente satisface la instancia. Cuando se encuentra el cuadro negro en el caso de las versiones paralelas se envía ese resultado al proceso maestro y este manda terminar la ejecución de los demás procesos que aún siguen ejecutándose.

La segunda parte azul representa el tiempo invertido en la actualización de las diferentes representaciones de las instancias, después de que se ha seleccionado una literal por algún método de selección de variables. Esa mismo cuadro azul representa el posprocesamiento después de que se termina de explorar el árbol de búsqueda, en el se realiza la presentación de los valores que satisfacen la instancia y también el calculo del tiempo total del programa.

La figura 3.11 no difiere mucho de la anterior, excepto que se puede observar que se invierte en forma considerable en el tiempo destinado al preprocesamiento, durante este además de realizar la lectura de la instancia, se realiza el proceso de seleccionar las literales que serán utilizadas por la regla de bifurcación. Además al aplicar la regla de bifurcación también se tiene que analizar si la variable elegida

ya ha sido instanciada previamente por algunas de las otras reglas de reducción, y en ese caso elegir otra de la lista. Por lo tanto el cuadro azul representa ahora solo la aplicación de las reglas de reducción, dado que el tiempo empleado en acceder a la lista de variables es despreciable.

La aplicación de las heurísticas de esta forma no garantiza la reducción del tiempo con respecto a la aplicación dinámica, pero ninguna método lo hace, todos pueden funcionar bien con algunas instancias, pero no con todas.

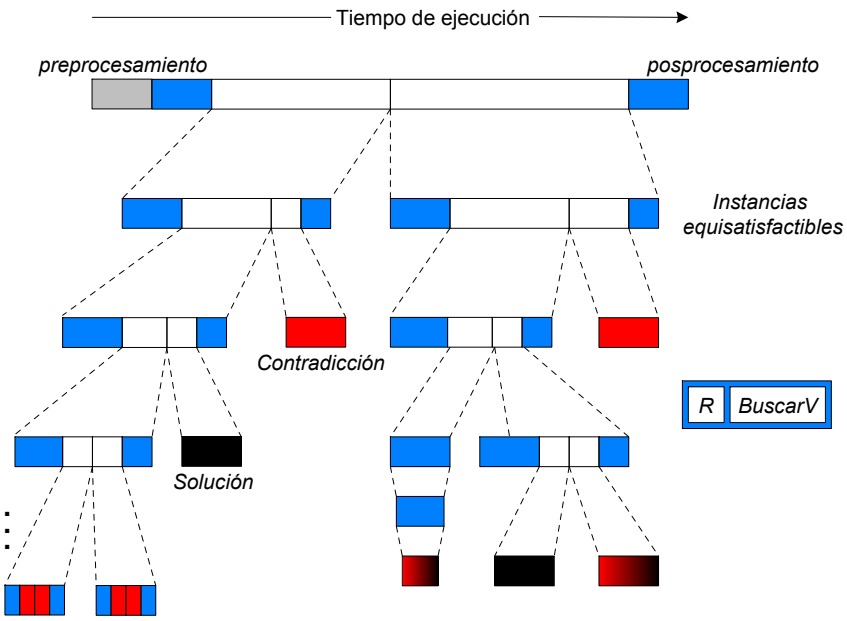


Figura 3.10: Llamadas recursivas

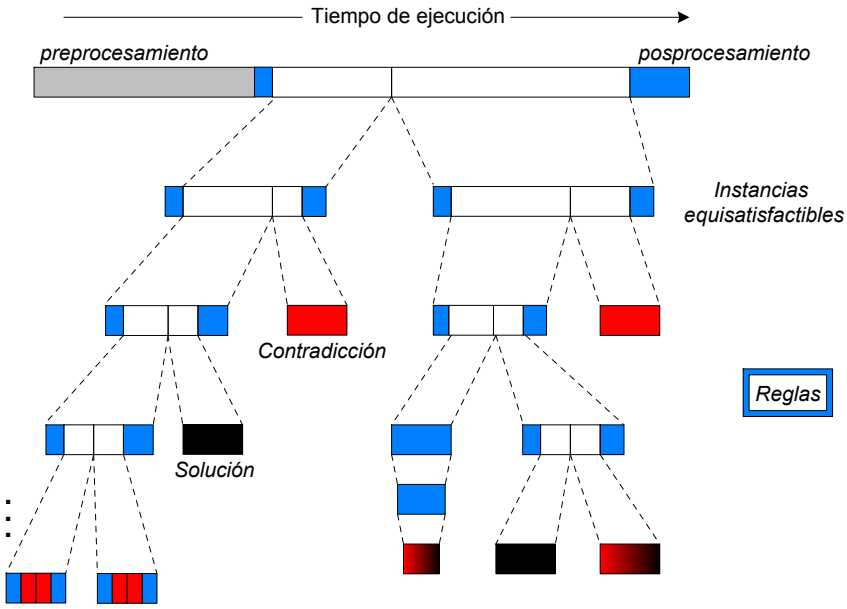


Figura 3.11: Llamadas recursivas 2

Capítulo 4

Características del sistema

4.1. Representación de la entrada

Uno de los formatos más comunes para representar las fórmulas es el CNF, sin embargo existen diferentes formas en que son representados en un archivo, en un inicio cada quien desarrollaba su propio formato de entrada. De todos los formatos en CNF, el más aceptado es el especificado por DIMACS[13], razón por la cual fue elegido este ultimo para usarlo como entrada de nuestros algoritmos.

El propósito de DIMACS es el de facilitar las pruebas entre distintos tipos de algoritmos y programas SAT, por medio de un formato común para representar las instancias de entrada.

4.1.1. Formato CNF

El archivo CNF es un documento `ASCII` con extensión .cnf, que consta principalmente de dos partes: El preámbulo y las cláusulas.

Preámbulo

El preámbulo contiene información acerca de la instancia, contenida en líneas, cada una de las cuales inicia con un carácter -que nos indica el tipo de información que contiene- y un espacio en blanco. La información que puede contener pueden ser comentarios o bien información del problema.

Comentarios Estas líneas son opcionales e inician con la letra *c* minúscula y son ignoradas por los los programas, su función solo es la de proporcionar

información adicional acerca de la instancia.

```
c Ejemplo de líneas
c de comentarios
```

Problema Línea obligatoria que inicia con la letra *p* minúscula seguida por el nombre del formato de representación que en este caso es *cnf*, el número de variables y el número de cláusulas. Debe aparecer una sola vez después de las líneas de comentarios si es que existen y antes de las líneas de cláusulas.

```
p FORMATO VARIABLES CLÁUSULAS
```

Donde `FORMATO` como ya se había mencionado antes es *cnf*, `VARIABLES` debe ser un número entero que indique la cantidad de variables que existen en la instancia y `CLÁUSULAS` debe ser un número entero que muestre la cantidad de cláusulas en la instancia.

Cláusulas

Las cláusulas inician inmediatamente después de la línea del problema, motivo por el cual no deben existir espacios en blanco después de la línea mencionada porque esto está fuera del estándar establecido y acarearía seguramente errores inesperados en la mayoría de los programas que lean el formato.

Las cláusulas se representan por números enteros que van desde el $-n$ hasta n; donde n es la cantidad total de variables en la instancia.

Cada cláusula se representa por una secuencia de números se parado por espacios; las variables afirmadas son representadas con un número positivo, mientras que las variables negadas se representan mediante un número negativo. El 0 se utiliza para terminar cada una de las cláusulas; cuando se encuentra este número automáticamente se pasa a la siguiente línea para buscar más cláusulas. Generalmente las cláusulas están ordenadas por el valor absoluto de las variables que lo integran, esto es, no se toma en cuenta su signo para el ordenamiento.

Por ejemplo las cláusulas

$$(x_1 \vee \bar{x}_3 \vee x_5) \wedge (x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_4) \wedge (\bar{x}_5) \wedge (x_3 \vee \bar{x}_4)$$

se representan en formato CNF en el cuadro 4.1.

4.2. Uso del sistema

Cada uno de las versiones toma como entrada solo el nombre del archivo en formato CNF, excepto para la versión con hilos que además del requisito anterior toma como parámetro adicional el número de hilos a ejecutar.

En la versión MPI hay que tener cuidado con la cantidad de procesos que se mandarán a crear, el número mínimo que se podrán crear son 2, esto es, siempre será necesario el proceso maestro para realizar el preprocesamiento, pero este por si solo no resuelve el problema, esto lo realizan los procesos hijos que el mismo crea.

Es por lo anterior que si solo se crea un proceso el programa no realizará ninguna acción visible. Entonces si solo se desea crear un proceso como si este fuera un programa secuencial, se deberá pasar un número dos en el parámetro que nos pide la cantidad de procesos a crear.

4.2.1. Modulo de selección de variables

Cada una de las versiones tiene varios métodos o heurísticas de selección de de variables, que a continuación se mencionan.

Secuencial: El método de selección secuencial elige las variables en un orden predefinido antes de la ejecución del algoritmo. Tiene la ventaja de que prácticamente no requiere tiempo para la elección de la variable, pero a su vez tiene la desventaja de que el orden de las variables es poco eficiente al realizar las transformaciones. Este método es el que ha arrojado los valores más grande de tiempo de ejecución con respecto a los demás métodos.

Aleatorio: El método de selección aleatorio, elige una variable que no haya sido instanciada aun de forma aleatoria. En general este método presenta un

```
1 1 −3 5 0
  2 −3 0
3 −1 −2 4 0
  −5 0
5 3 −4 0
```

Cuadro 4.1: Representación de las cláusulas en el formato CNF.

rendimiento intermedio entre el método secuencial y el MOMS.

MOMS: Por sus siglas en inglés de «Maximum Occurrences in Minimum Size Clauses», Apariciones máximas en cláusulas de menor tamaño. Selecciona la variable con mayor aparición en las cláusulas de menor tamaño. Cabe hacer mención que este método es igual a MAXO en caso de que todas las cláusulas contengan la misma cantidad de variables. Esta heurística es la que ha arrojado los mejores resultados en cuanto a tiempo de ejecución se refiere, esto es debido a que al elegir las variables con mayor aparición en la cláusula, al realizar las transformaciones correspondientes se eliminan más cláusulas. De esta forma queda una instancia equisatisfactible más pequeña y en teoría más fácil de resolver.

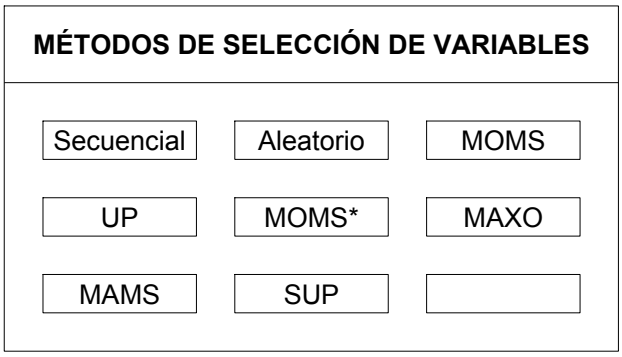


Figura 4.1: Métodos de selección de variables

Dado que la adecuada selección de variables afecta de manera directa en la eficiencia del algoritmo se agregaron los siguientes métodos de selección de variables, con el fin de mejorar los resultados obtenidos con los anteriores.

UP: Realiza asignaciones a las variables que aun están indefinidas, y cuenta con que variable se realiza la cadena más larga de propagaciones unitarias. En este método será necesario fijar un limite de búsqueda, puesto que puede ser mucho mayor el tiempo invertido en preprocesamiento que el tiempo que podría ahorrar.

MAXO: A diferencia de MOMS este método elige la variable con mayor ocurrencia dentro de la instancia, sin importar el tamaño de las cláusulas.

MAMS: Es una combinación de MAXO y MOMS, se trata de decidir según el tipo de problema, que método arroja los mejores resultados y con base en eso escoger el método adecuado. Generalmente cuando es un problema k-SAT se elige MAXO.

SUP : UP Selectivo, ejecuta UP solo en aquellas variables elegidas previamente por otros métodos, como MAXO, MOMS, MAMS, etcétera.

MOMS* A diferencia de MOMS esta heurística elige la variable con mayor número de apariciones pero balanceadas, esto es, que el numero de apariciones tanto negativas como positivas sean similares.

BAL Elige a las variables balanceadas, es decir, aquellas cuya diferencia entre las apariciones positivas y negativas tienda a cero, sin importar el número total de apariciones.

4.3. Estructura de datos

Para la representación de la instancia se utilizan tres distintas representaciones que al trabajar en conjunto con ellas superan sus ventajas individuales. A pesar del consumo adicional de memoria, su uso se justifica por la rapidez con que se puede encontrar, modificar o eliminar una variable o una cláusula sin tener que realizar una búsqueda.

Cada una de las representaciones son equisatisfactibles, esto es, aun cuando su forma y contenido varía, representan a la misma fórmula y su resultado no se modifica.

4.3.1. Representación matricial

En la representación matricial se utiliza un arreglo de n variables por m cláusulas, donde las columnas contiene las variables y las filas a las cláusulas. Cada elemento del arreglo puede contener uno de tres estados posibles: -1, 0, 1.

$$RM : arreglo[1:n] \times [1:m]$$

$$RM(i,j) = \begin{cases} +1, & \text{si } Q_{i,j} \text{ está afirmado en } C_i; \\ -1, & \text{si } Q_{i,j} \text{ está negado en } C_i; \\ 0, & \text{si } Q_{i,j} \text{ no aparece en } C_i. \end{cases}$$

Si la variable x_i, de la cláusula C_j, está negada, entonces el elemento del arreglo `matriz[i][j]` contendrá un -1; de la misma forma si ahora la variable está afirmada, entonces el elemento del arreglo contendrá un 1 y por ultimo, si la variable no se encuentra en una cláusula entonces la intersección contendrá un cero.

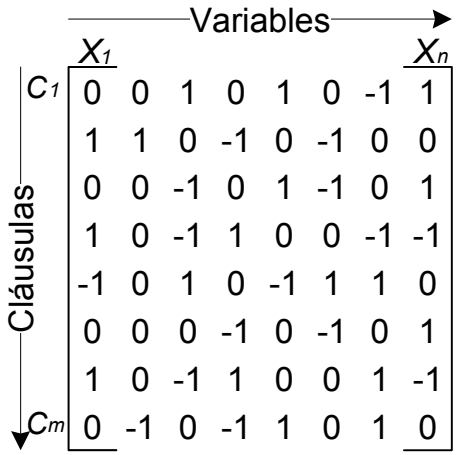


Figura 4.2: Representación matricial

4.3.2. Representación por cláusulas

En esta representación la primer columna contiene el número de cláusulas para cada una de las variables existentes, el resto de las columnas indican propiamente el número de la cláusulas en donde se encuentran la variables de cada una de las filas.

La primer fila nos da un acceso rápido si deseamos conocer que variable contiene el mayor número de apariciones en la fórmula, esta es una de las principales ventajas que caracteriza a esta representación. Las columnas tienen un tamaño fijo, asignado al momento de crearse, con el tamaño de la variable con más apariciones dentro de la fórmula.

El tamaño de las filas es dos veces la cantidad de variables, esto es necesario para saber en que cláusulas estás las variables, tanto afirmadas como negadas; la primera mitad de las filas contienen a las variables afirmadas $x_1, x_2, \ldots, x_{n-1}, x_n$ y de igual manera en la segunda mitad se encuentran las mismas variables pero

negadas $\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_{n-1}, \bar{x}_n$. Las cláusulas para una variable dada están en orden ascendente con respecto a las cláusulas en las que aparecen.

$$RC : arreglo[1 : k+1] \times [1 : 2n]$$

$$RC(g,h) = \begin{cases} |x_g|, & \text{si } h = 1 \text{ cuando } 1 \leqslant g \leqslant n; \\ |\bar{x}_g|, & \text{si } h = 1 \text{ cuando } n+1 \leqslant g \leqslant 2n; \\ i, & \text{si } Q_g \in C_i \; \forall \; i \in \mathbb{N} \mid i < m. \end{cases}$$

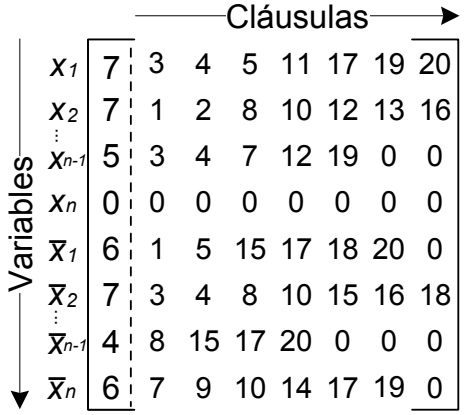


Figura 4.3: Representación por cláusulas

4.3.3. Representación por literales

Esta representación es parecida a la representación matricial, cada columna representa a las variables, excepto la primera que contiene la cantidad de variables por cada cláusula; y cada línea representa a las cláusulas.

La primer columna además de indicarnos si una cláusula ya no contiene variables, al tener un cero, nos puede indicar si alguna cláusula ha sido borrada; y una cláusula ha sido borrada si contiene el valor de -1 en la primer columna.

De igual forma en esta representación las variables están ordenadas en forma ascendente, sin tomar en cuenta el signo. Las variables afirmadas son positivas y las negadas están con valor negativo, cuando las variables se van eliminando se recorren hacia la izquierda y el lugar desocupado se llena con un cero.

Esta representación nos ofrece la ventaja de conocer rápidamente las cláusulas que siguen activas, esto es, las que no han sido eliminadas y que aún contienen variables, además de saber cuales variables hayan quedado huérfanas dentro de una cláusula (cuando en la primera fila contiene un valor de 1).

Generalmente se utilizan instancias k-CNF, por lo que en este caso en un inicio la primera columna contendrá el mismo valor, que posteriormente ira variando según se vayan eliminando las variables.

$$RL : arreglo[1 : n+1] \times [1 : m] \forall l_{i,j} \in \{-n : n\} \text{ y } l_{1,j} \in \mathbb{N}$$

$$RL(g,h) = \begin{cases} |C_g|, & \text{si } h = 1 \text{ cuando } 1 \leqslant g \leqslant m; \\ -1, & \text{si } C_g \ni \varphi; \\ Q_{g,j}, & \forall j \in \{1, \ldots, |C_g|\}; \\ 0, & \text{si } Q_{g,j} \ni C_g \text{ cuando } |C_g| < h < n. \end{cases}$$

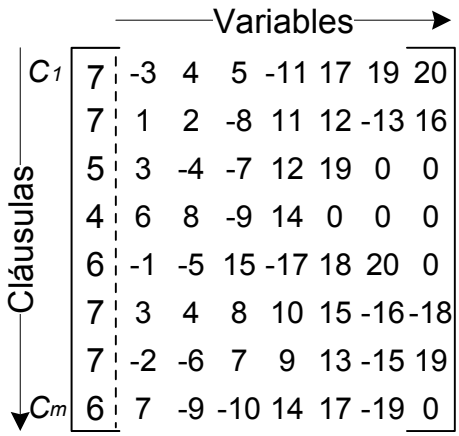


Figura 4.4: Representación por variables

4.4. Otras representaciones

Existen dos listas de literales —que se manejan internamente como un arreglo— las cuales sirven para separar las variables a las que no se les ha asignado un valor y a las que sí.

Al final en caso de que el resultado sea satisfactible, la segunda lista contendrá los valores que satisfacen la fórmula, si se da el caso de que la primera lista aun tenga literales, a estas se les puede usar como comodines, se les puede asignar cualquier valor booleano sin que afecte el resultado, lo cual podría verse como otras asignaciones, teniendo hasta 2^n asignaciones posibles más, donde n es la cantidad de variables en la primera lista.

Cuando existe un resultado satisfactible, la lista parcial es devuelta al proceso padre, y este la toma como la lista final de literales, la solución al problema. Y en caso de que la lista parcial de un subárbol no encuentre una solución, simplemente es desechada.

En adición existe otra lista que es útil para algunas heurísticas como auxiliar en la búsqueda de las variables óptimas, se trata de la lista que contiene al contador de literales. Como su nombre lo indica, el contador de literales contiene la cantidad de variables que existen en un momento determinado en la instancia actual.

El proceso de crear la lista requiere una buena parte del preprocesamiento, pero como tal solo se realiza una sola vez en la vida del programa, una vez creada la lista solo se modifica cuando ocurre la eliminación de una variable, en cuyo caso tal variable se borra de ella.

Como ya se mencionó el proceso padre crea la lista en el preprocesamiento, y lo envía tal como está a los hijos, junto con una lista de variables, es ahí donde ellos eliminan de la lista las variables que se le pasaron y comienzan su búsqueda de la solución en la rama del árbol que les toca.

Capítulo 5

Resultados

Las pruebas se realizaron en tres computadoras Sun Fire V20z Server con dos procesadores AMD Opteron a 2,4 GHz y 2 Gb de memoria DDR1-333 SDRAM cada uno, con Red Hat linux como sistema operativo.

Para probar la implementaciones que se realizaron del algoritmo DP, se usaron las instancias propuestas por DIMACS (SATLIB), las cuales aunque no muestran sus soluciones nos indican si el resultado esperado es satisfactible o insatisfactible.

Dentro de las pruebas realizadas, se pudo comprobar que la elección de la heurística de selección de variables afecta notablemente en el tiempo de ejecución del algoritmo. Las heurísticas más eficientes fueron MOMS y MAXO, para las instancias de los *benchmarks* de SATLIB hasta con 75 variables.

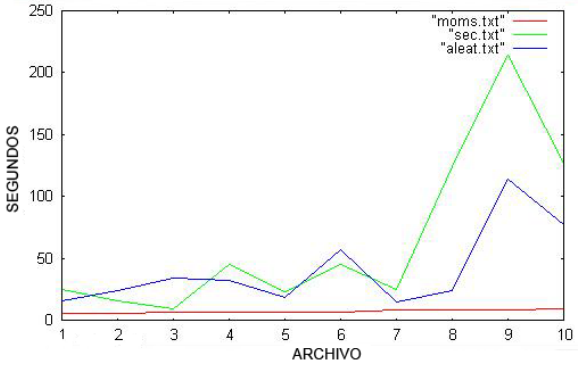


Figura 5.1: Resultados de una de las primeras versiones del algoritmo secuencial recursivo usando diferentes heurísticas de selección de variables con archivos de 75 variables y 375 cláusulas, todas ellas insatisfactibles.

También se comprobó que al aumentar la cantidad de hilos disminuye el tiempo de ejecución del algoritmo, cuando la cantidad de variables es pequeña muchas veces el tiempo de la versión secuencial es inferior al de la versión paralela, debido en gran medida al manejo de memoria para las estructura de datos, y por el costo implicado en la creación de los hilos.

En la implementación del algoritmo de Davis-Putnam influyen diferentes factores en la eficiencia, tales como la estructura de datos utilizada, el método de selección de variables, etcétera.

Durante el desarrollo se fueron haciendo diferentes pruebas con estas variables con el fin de disminuir el tiempo de cómputo.

Como se puede ver al aumentar la cantidad de variables nuestro algoritmo es más eficiente con respecto a Satzs. Llegando el punto en el que Satzs queda por muy por debajo del tiempo de ejecución de nuestra versión.

5.1. Resultados de la versión secuencial

En la versión secuencial la variable que se utiliza para manejar su eficiencia es el método de selección de variables elegido para realizar la bifurcación. La heurística que ha dado mejores resultados ha sido la MOMS.

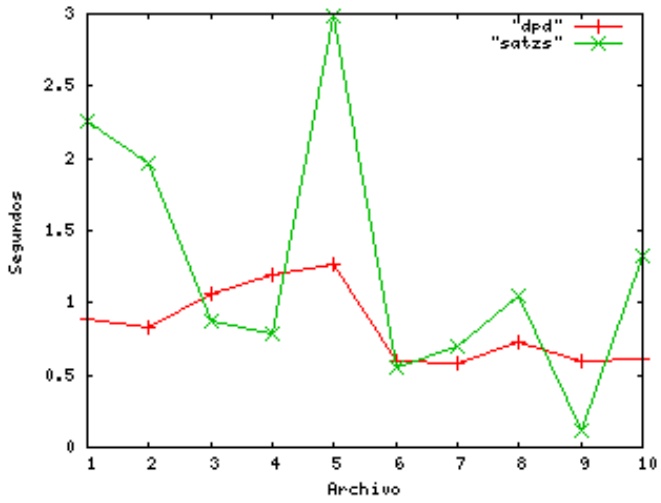
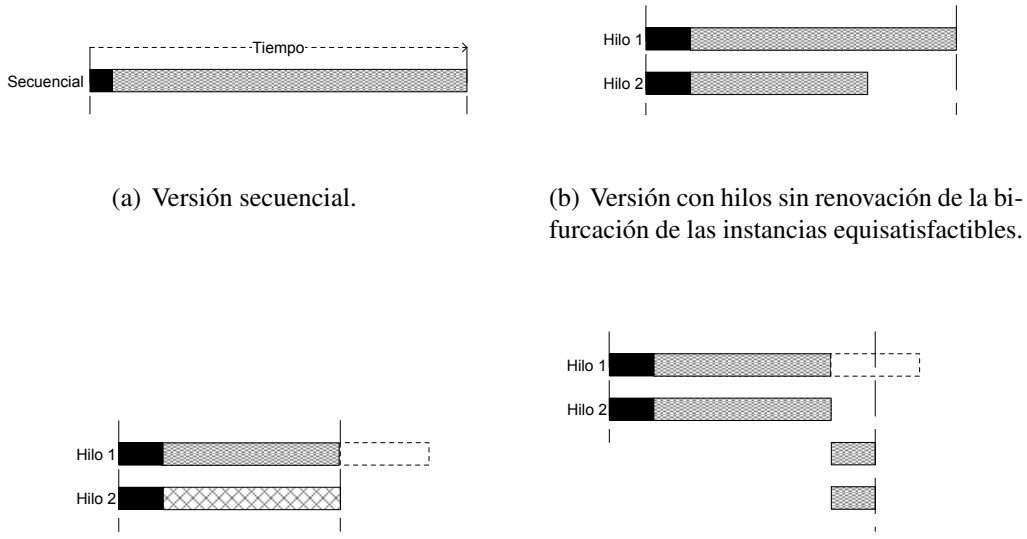


Figura 5.2: Comparación de nuestro programa con Satzs utilizando archivos de pruebas con 75 variables y 375 cláusulas, todas ellas insatisfactibles.



(a) Versión secuencial.

(b) Versión con hilos sin renovación de la bifurcación de las instancias equisatisfactibles.

(c) Versión con hilos que termina al encontrar una asignación que satisface a la fórmula.

(d) Versión con hilos que vuelve a dividir la instancia equisatisfactible cuando uno de los hilos termina sin encontrar una solución.

Figura 5.3: Comparación de la versión secuencial con la versión con hilos.

En general, de todas las heurísticas la secuencial es la que ha tenido el desempeño menos favorable, aunque no está muy lejos de la versión aleatoria.

En la comparación de la versión recursiva e iterativa, prácticamente no hay diferencia en cuanto a eficiencia, aunque esta ultima gasta un poco más de tiempo, aunque es una diferencia despreciable, pudiendo usarse indistintamente.

5.2. Resultados de la versión de hilos

Los resultados de esta versión utilizando un solo hilo son mayores que los de la versión secuencial; principalmente se debe al procesamiento previo que implica la creación de un nuevo hilo, y en menor medida al tiempo que invierte el sistema operativo en la creación y manejo de los hilos, pero este tiempo se puede considerar despreciable.

El tiempo de esta versión al utilizarse con 2 hilos es inferior al de la versión secuencial, sin embargo no llega a ser la mitad del tiempo secuencial. Esto se debe a que la división que se realiza en un inicio del árbol de búsqueda no asigna

dos ramas iguales en cuanto a dificultad. Si el programa tiene la opción de pedir una nueva rama al terminar la ejecución de una ellas con un resultado insatisfactible, se debe tener una instancia con 100 variables como mínimo para que no sea contraproducente volver a hacer otro particionamiento y asignación de las instancias. Lo anterior es debido al costo en tiempo que es eso conlleva. Si la instancia tiene menos de 100 variables es mejor dejar continuar el algoritmo, esperando la respuesta del segundo hilo.

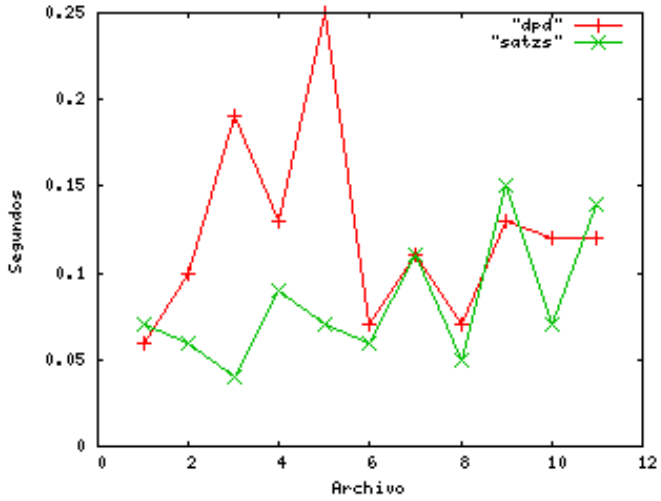


Figura 5.4: Comparación de nuestro programa con Satzs utilizando archivos de pruebas con 100 variables y 430 cláusulas, todas ellas insatisfactibles.

Dado que solo se probó esta versión con 2 hilos no se pudo hacer un análisis de su aceleración. Sin embargo los tiempos que arrojó esta versión son inferiores a la versión secuencial.

Al estar trabajando con memoria compartida se tuvo un mejor desempeño en esta versión comparado con la versión con MPI, particularmente se tuvo un acceso más rápido a las estructuras de datos. No se gasto en tiempo en comunicaciones, por lo que después de dividir la instancia original, prácticamente iniciaba la ejecución de los hilos.

5.3. Resultados de la versión de MPI

Sin las restricciones en la utilización de memoria compartida, con la versión de MPI se pudieron analizar las instancia utilizando mas procesos a comparación de la versión que usa hilos. En esta versión existe un costo considerable en comunicaciones dado que las instancias que se envían al inicio a los procesos hijos son de un tamaño considerable sobre todo al incrementarse la cantidad de variables y cláusulas, esto aunado que también se tienen que enviar la lista de literales a procesarse, la lista de asignaciones parciales y la lista de apariciones de las variables. Sin embargo esto se ve minimizado al pensar en las instancias de mayor tamaño que se pueden resolver.

Esta versión tiene una eficiencia menor comparada con la versión con hilos, al compararse la versión con anterior con dos hilos y la versión con MPI utilizando dos procesos, la primera tuvo una respuesta más rápida sobre todo por que tiene un acceso rápido de los datos. Sin embargo como ya se menciono antes con la versión en MPI se pueden utilizar muchos mas procesos para resolver una instancia grande simplemente añadiendo más maquinas a la red.

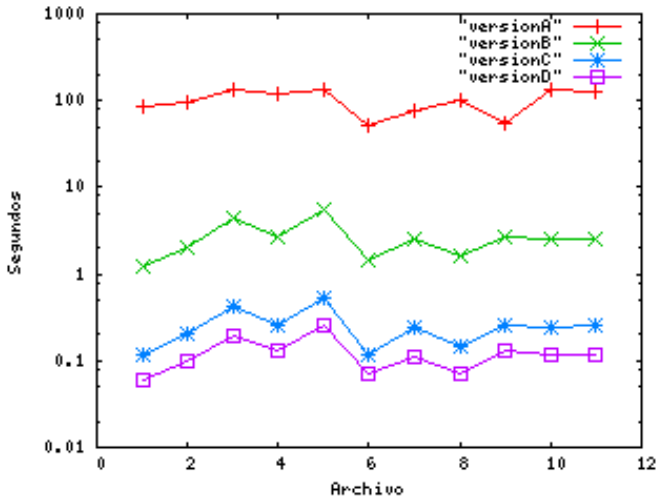


Figura 5.5: Resultados de las diferentes versiones, después de hacer ajustes, usando como pruebas archivos de 75 variables y 375 cláusulas, todas ellas insatisfactibles.

La aceleración en esta versión no es la óptima, sin embargo a medida que se incrementa el número de hilos se obtienen tiempos inferiores, dada la complejidad del problema.

5.4. Análisis de resultados

La diferencia entre los resultados de la versión con hilos y la de MPI es despreciable cuando la cantidad de variables es grande y cuando la cantidad de hilos es pequeña. Sin embargo cuando se utilizan cantidades de variables pequeñas la ventaja de la versión con hilos aumenta, sobre todo porque no invierte tiempo en comunicaciones.

La paralelización del algoritmo de Davis-Putnam por medio de la división del árbol de búsqueda mostró una disminución importante en el tiempo de ejecución con respecto a la versión secuencial. Aunque esto es para problemas suficientemente grandes, puesto que para problemas con pocas variables resulta más costoso el tiempo invertido en la paralelización que los beneficios de este.

También la elección de la heurística de selección de variables juega un papel muy importante en el algoritmo, por ello se está centrando el estudio de una combinación de las diferentes heurísticas estudiadas, con el fin de minimizar el tiempo de ejecución y maximizar la cantidad de variables que puedan resolverse en un tiempo razonable.

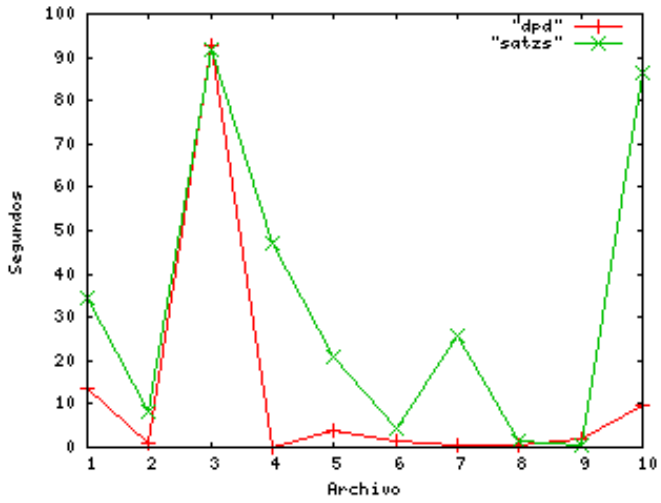


Figura 5.6: Comparación de nuestro programa con Satzs utilizando archivos de pruebas con 150 variables y 520 cláusulas, todas ellas satisfactibles.

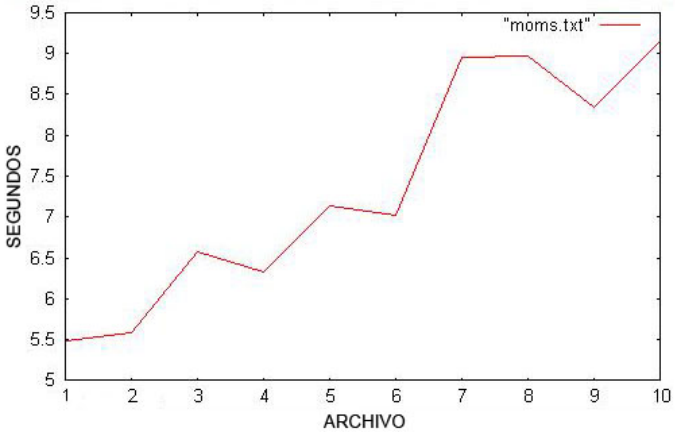


Figura 5.7: Resultados usando el método MOMS para selección de variables utilizando 2 hilos con una instancia de 20 variables.

5.5. Comparación con otras implementaciones

De otras implementaciones con las que se pudo contar con el código fuente, se compararon con ZRes[6], Small-Satz[34], CalcRes[5] y Satz[22].

Hay que aclarar que en el cuadro5.5 se presentan las comparación con las implementaciones con las que se tuvieron resultados favorables con respecto al tiempo total de cómputo.

ZRes ZRes es una implementación del algoritmo de Davis y Putnam tal y como fue planeado en 1960. ZRes usa ZBDD como la estructura de datos para representar en conjunto de cláusulas, que forman la instancia a procesar. El paquete usado para manipular ZBDD usa el paquete CUDD (un paquete de BDD), diseñado por Fabio Somezi.

ZRes fue creado por Laurent Simon y Philippe Chatalic en la Universidad de París, en el LRI (*Laboratoire de Recherche en Informatique*), en el año 2000.

Small-Satz Small-Satz fue escrito por Peter G. Stock en 1999, basado en la implementación de Satzs escrita por Chu Min Li. Es una modificación de Satzs, en la que se han eliminado los descubrimientos hechos por Chu Min Li y los avances y modificaciones que contiene Satzs basado en otros algoritmos. Solo se encuentra la versión del algoritmo DLL, junto con las heuristicas y la estructura original de Satzs.

benchmark	# variables	# cláusulas	# pruebas	small-Satz	zres	CalcRes	secuencial
uf20-01.cnf	20	91	5	.	0.05	0.50	0.00
uuf20-01.cnf	20	91	5	.	1.24	0.50	0.00
uf50-01.cnf	50	218	5	.	37.56	> 1800	0.00
uuf50-01.cnf	50	218	5	.	602.45	>1800	0.00
uf50-0100.cnf	50	218	.	.	.	>1800	.
uuf50-0100.cnf	50	218	.	.	.	>1800	.
uf75-01.cnf	75	325	.	.	.	.	.
uuf75-01.cnf	75	325	5	0.071	.	.	0.060
uf75-0100.cnf	75	325	.	.	.	.	.
uuf75-0100.cnf	75	325	5	0.040	.	.	0.190
uf75-0100.cnf	75	325	.	.	.	.	.
uuf75-0100.cnf	75	325	.	.	.	.	.
uf100-01.cnf	100	430	.	.	.	.	.
uuf100-01.cnf	100	430	5	2.250	.	.	0.890
uf125-01.cnf	125	538	.	.	.	.	.
uuf125-01.cnf	125	538	.	.	.	.	.
uf150-01.cnf	150	645	5	4.470	.	.	1.450
uuf150-01.cnf	150	645	.	.	.	.	.
uf175-01.cnf	175	753	5	258.750	.	.	38.110
uuf175-01.cnf	175	753	5	.	.	.	83.670

Cuadro 5.1: Comparación con otras implementaciones

CalcRes Es una implementación del algoritmo original de Davis-Putnam, tal y como fue escrito en 1960. A diferencia del algoritmo de Davis, Logeman y Loveland propuesto dos años después este algoritmo elimina cada variable una a una. Esta escrito en el lenguaje Objective Caml y fue diseñado por Laurent Simonde la Universidad de París, en el LRI (*Laboratoire de Recherche en Informatique*), en el año 1999.

La caracteristica de CalcRes es que usa árboles Trie[1] como estructura de datos, para manipular el conjunto de cláusulas; además utiliza una heurística dinámica para la selección de variables.

Satz Uno de los mejores programas para resolver el problema SAT que implementan el algoritmo DPLL es Satz. Usa MOMS como heurística de selección de variables, elige una literal l y ejecuta la regla de literal unitaria por cada una de esas de esas literales, logrando con estos la ejecución de propagaciones unitarias por cada literal.

Las literales obtienen un peso mayor si después de la propagación unitaria la fórmula es más pequeña comparada con el resultado de las otras literales. Esto optimiza en gran medida la heurística ya que las variables elegidas tienen un buen desempeño en las primeras decisiones, aquellas que se encuentran cerca de la raíz.

Las instancias probadas al igual que las usadas anteriormente fueron hechas por DIMACS y están disponibles en su página web. Las implementaciones anteriores son secuenciales, son las implementaciones con las que el algoritmo modificado propuesto presenta ciertas ventajas.

Todas las versiones están escritas en ANSI C, utilizando solo el conjunto de funciones matemáticas, ninguna requiere de un *software* especial, excepto por ZRes que para compilar necesita el paquete CUDD. Pueden ejecutarse sin ningún problema en cualquier equipo ejecutando Linux.

Como se observa ZRes es que presenta el desempeño más bajo, esto debido al manejo que tiene que realizar con la estructura de datos que representa a la instancia.

ZRes propone una mejora en cuanto a la estructura de datos para manejar la instancia, mientras que Small-Satz trata de mejorar por medio de las heurísticas.

El algoritmo secuencial presentado trata de mejorar en los dos aspectos, por medio de la estructura de datos empleada, que en este caso son tres, con el objetivo

[1] Un árbol ordenado como estructura de datos

de tener al alcance los datos evitando el procesamiento empleado en la búsqueda de estos después de procesar las cláusulas y variables.

También analizando las ventajas que nos pueden dar determinadas heurísticas de selección de variables con algunas instancia, por ejemplo aquellas que son generadas aleatoriamente, o las que provienen de un problema especifico, o las que tienen ciertas restricciones como las que provienen de las instancias k-CNF.

Lo anterior en el caso de la versión secuencial, en el caso de las versiones paralelas se pretende además de tener las ventajas anteriores, aprovechando el paralelismo para manejar distintas heurísticas.

Capítulo 6

Conclusiones y trabajo futuro

La paralelización del algoritmo de Davis-Putnam por medio de la división del árbol de búsqueda mostró una disminución importante en el tiempo de ejecución con respecto a la versión secuencial. Aunque esto es para problemas suficientemente grandes, puesto que para problemas con pocas variables resulta más costoso el tiempo invertido en la paralelización que los beneficios de este.

Se probó que un sistema no muy complejo puede dar muy buenos resultados; con la utilización de varias representaciones de las instancia se logro disminuir el tiempo de búsqueda con algunas heurísticas, todo esto con la desventaja de la utilización de más memoria.

Se implementaron los algoritmos de forma modular, esto aunado a estructuras de datos sencillas, permite una modificación más fácil del sistema. Todos esto posibilita la escalabilidad de los programas, por lo que se pueden agregar algunas otras técnicas recientes con pocos cambios.

También la elección de la heurística de selección de variables juega un papel muy importante en el algoritmo, por ello se está centrando el estudio de una combinación de las diferentes heurísticas estudiadas, con el fin de minimizar el tiempo de ejecución y maximizar la cantidad de variables que puedan resolverse en un tiempo razonable.

Se está trabajando en otros métodos de elección de literales y algunas técnicas que nos permitan deducir mediante un preprocesamiento, el beneficio o no que nos pueda traer el recorrer una ramificación del árbol de búsqueda. Tratando con lo anterior de reducir nuestro campo de búsqueda, para de esta forma reducir sustancialmente el tiempo de computo final.

Sin embargo el trabajo demuestra que existen heurísticas que pueden procesar instancias con una gran cantidad de variables y cláusulas en un tiempo razona-

ble, aun cuando los resultados no mejoren sustancialmente con respecto a otros programas.

6.1. Trabajo futuro

Como trabajo futuro se pueden agregar otras heurísticas para la selección de variables, se pueden combinar las ventajas que ofrecen las diversos métodos. Realizar un estudio más profundo en donde se muestre el impacto de la selección de variables en distintos tipos de instancias.

Se puede realizar una modificación a los algoritmos para lograr la tolerancia a fallos, dado que en este momento si existe algún problema con uno de los hilos o procesos, el sistema no es capaz de recuperarse. Esto es importante debido a que generalmente la respuesta se obtiene después de un tiempo de computo considerable.

Es posible también modificar la versión del algoritmo iterativo para que acepte *backtracking* no cronológico o *backjumping* o reinicios.

Apéndice A

Uso del sistema

A.1. Secuencial

```
>secuencial
-h=n          : Numero de la heuristica
-h2=n         : Segunda heuristica si se elige la 6 o 9
-orden=e      : Eleccion estatica
-orden=d      : Eleccion dinamica
-f=archivo    : Abre el archivo que contiene la instancia
-res=archivo  : Guarda el resultado en archivo

Las heuristicas son :
 0 : Secuencial
 1 : Aleatorio
 2 : MOMS [predeterminado]
 3 : UP
 4 : MAXO
 5 : MAMS
 6 : SUP
 7 : MOMS
 8 : BAL
 9 : Todos en ciclo
```

Cuadro A.1: Opciones de la versión secuencial

A.2. Hilos

```
>hilos
Uso : hilos archivo hilos [opciones]
-n=           : Numero de hilos
-h=n          : Numero de la heuristica
-h2=n         : Segunda heuristica si se elige la 6 o 9
-orden=e      : Eleccion estatica
-orden=d      : Eleccion dinamica
-modo=d       : Divide el arbol de busqueda
-modo=i       : Misma instancia con diferentes heuristicas
-f=archivo    : Abre el archivo que contiene la instancia
-res=archivo  : Guarda el resultado en archivo

Las heuristicas son :
 0 : Secuencial
 1 : Aleatorio
 2 : MOMS [predeterminado]
 3 : UP
 4 : MAXO
 5 : MAMS
 6 : SUP
 7 : MOMS
 8 : BAL
 9 : Todos en ciclo
```

Cuadro A.2: Opciones de la versión con hilos

A.3. MPI

```
>mpi
Uso : mpi archivo hilos [opciones]
-h=n          : Numero de la heuristica
-h2=n         : Segunda heuristica si se elige la 6 o 9
-orden=e      : Eleccion estatica
-orden=d      : Eleccion dinamica
-modo=d       : Divide el arbol de busqueda
-modo=i       : Misma instancia con diferentes heuristicas
-f=archivo    : Abre el archivo que contiene la instancia
-res=archivo  : Guarda el resultado en archivo

Las heuristicas son :
 0 : Secuencial
 1 : Aleatorio
 2 : MOMS [predeterminado]
 3 : UP
 4 : MAXO
 5 : MAMS
 6 : SUP
 7 : MOMS
 8 : BAL
 9 : Todos en ciclo
```

Cuadro A.3: Opciones de la versión en MPI

Bibliografía

[1] Cook S. A. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on the Theory of Computing*, pages 151–158, 1971.

[2] Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. Solving difficult instances of boolean satisfiability in the presence of symmetry. *IEEE Transactions on CAD*, 22(9):1117–1137, September 2003.

[3] Gilles Audemard, Daniel Le Berre, Olivier Roussel, Inês Lynce, and João Marques-Silva. OpenSAT: An open source SAT software project.

[4] Peter Barth. A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization. Research Report MPI-I-95-2-003, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, January 1995.

[5] P. Chatalic and L. Simon. Davis and putnam 40 years later: a first experimentation, 2000. 5.5

[6] P. Chatalic and L. Simon. ZRes: The old Davis-Putnam procedure meets ZBDDs. In David McAllester, editor, *17th International Conference on Automated Deduction (CADE'17)*, number 1831, pages 449–454, 2000. 5.5

[7] Wen-Tsuen Chen and Lung-Lung Liu. A parallel approach for theorem proving in propositional logic. *Information Sciences*, 41(1):61–76, 1987. 2

[8] W. Chrabakh and R. Wolski. GrADSAT: A parallel SAT solver for the grid. Technical Report Number 2003-05, University of California, Santa Barbara, Febrero 2003.

[9] Stephen A. Cook and David G Mitchell. Finding hard instances of the satisfiability problem: A survey. In Du, Gu, and Pardalos, editors, *Satisfiability Problem: Theory and Applications*, volume 35, pages 1–17. American Mathematical Society, 1997.

[10] R. Cowen and K. Wyatt. Breakup: A preprocessing algorithm for satisfiability testing of CNF formulas. *Notre Dame J. Formal Logic 34*, pages 602–606, 1993.

[11] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962. 1.2.1, 2, 2.1.1, 2.1.2

[12] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960. 1.2.1, 1, 3

[13] DIMACS. Satisfiability: Suggested format, mayo 1993. 4.1

[14] Franco and Ho. Probabilistic performance of a heuristic for the satisfiability problem. *DAMATH: Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science*, 22, 1988.

[15] Alex S. Fukunaga. Variable-selection heuristics in local search for SAT. In *AAAI/IAAI*, pages 275–280, 1997.

[16] J. Gu, P. Purdom, J. Franco, and B. Wah. Algorithms for the satisfiability (SAT) problem: A survey, 1996.

[17] Marc Herbstritt. Improving propositional satisfiability algorithms by dynamic selection of branching rules, 2001.

[18] John N. Hooker and V. Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15(3):359–383, 1995.

[19] H. Hoos and T. Sttzle. SATLIB: An online resource for research on SAT, 2000.

[20] Paul Walton Purdom Jr. Average time for the full pure literal rule. *Information Sciences*, 78(3-4):269–291, 1994.

[21] Elias Koutsoupias and Christos H. Papadimitriou. On the greedy algorithm for satisfiability. *Information Processing Letters*, 43(1):53–55, 1992. 3.1

[22] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *IJCAI (1)*, pages 366–371, 1997. 5.5

[23] Raymundus Lullus. *Ars generalis ultima*. Barcelona: Pere Posa, 1501. 1

[24] Crawford J. M., Auton L. D. P., Boufkhad Y., and Carlier J. Experimental results on the crossover point in satisfiability problems. *DIMACS, Rutgers University, New Brunswick, NJ 08903*, 1993.

[25] Li C. M and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 366–371, 1997.

[26] David G. Mitchell, Bart Selman, and Hector J. Levesque. Hard and easy distributions for SAT problems. In Paul Rosenbloom and Peter Szolovits, editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 459–465, Menlo Park, California, 1992. AAAI Press. 3.1

[27] Guillermo Morales-Luna. Lógica matemática: Un enfoque computacional. CINVESTAV, diciembre 2002.

[28] Dubois O., Andre P., Boufkhad Y., and Carlier J. Can a very simple algorith be efficient for solving the SAT problem? *DIMACS, Rutgers University, New Brunswick, NJ 08903*, 1993.

[29] Dechter R. and Rish I. Directional resolution: The DavisPutnam procedure. In *Proceedings of KR-94*, 1994.

[30] J. W. Rosenthal, J. W. Plotkin, and J. Franco. The probability of pure literals. *Journal of Logic and Computation*, 9(4):501–513, 1999.

[31] Bart Selman, David G. Mitchell, and Hector J. Levesque. Generating hard satisfiability problems. *Artificial Intelligence*, 81(1-2):17–29, 1996.

[32] Carsten Sinz, Wolfgang Blochinger, and Wolfgang Küchlin. PaSAT - parallel SAT-checking with lemma exchange: Implementation and applications. In H. Kautz and B. Selman, editors, *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001)*, volume 9 of *Electronic Notes in Discrete Mathematics*, Boston, MA, USA, 2001. Elsevier Science Publishers. 1.3

[33] M. Stickel and H. Zhang. Implementing the Davis-Putnam method. *Journal of Automated Reasoning*, (24):277–296, 2000.

[34] Peter G. Stock. Solving non-boolean satisfiability problems with the Davis-Putnam method. Bsc dissertation, Department of Computer Science, University of York, marzo 2000. 5.5

[35] Moskewicz M. W., Madigan C. F, Zhao Y., Zhang L., and Malik S. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*, pages 530–535, 2001.

[36] Makoto Yokoo. Weak-commitment search for solving constraint satisfaction problems. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94); Vol. 1*, pages 313–318, Seattle, WA, USA, July 31 - August 4 1994. AAAI Press, 1994.