ARTÍCULO

An Introduction to Satisfiability Algorithms

Carlos Ansotegui, Felip Manyà

*Dpto. de Informática e Ing. Industrial
Universitat de Lleida
Jaume II, 69, E-25001 Lleida, España

e-mail: {carlos, felip}@eup.udl.es

In this paper we present an introduction to satisfiability algorithms. Since most complete SAT solvers (e.g. Satz, SATO, GRASP, and Chaff) are based on the Davis-Putnam procedure, we first describe in detail that procedure. Then, we present the improvements that can be incorporated into the Davis-Putnam Procedure in order to develop a competitive SAT solver: optimized data structures, variable selection heuristics, non-chronological backtracking, conflict-driven learning, and restarts. Finally, we describe GSAT and WalkSAT, which are the most widely used local search algorithms for solving SAT.

Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial. No.20 (2003)
ISSN: 1137-3601. © AEPIA (http://www.aepia.org/revista) pp.43-56

Una introducción a los algoritmos de satisfactibilidad

Carlos Ansótegui*
Dpto. de Informática e Ing. Industrial
Universitat de Lleida
Jaume II, 69, E-25001 Lleida, España
carlos@eup.udl.es

Felip Manyà
Dpto. de Informática e Ing. Industrial
Universitat de Lleida
Jaume II, 69, E-25001 Lleida, España
felip@eup.udl.es

Resumen

En este artículo se presenta una introducción a los algoritmos de satisfactibilidad. Primero, se describe el procedimiento de Davis-Putnam, que constituye la base de la mayoría de algoritmos completos (por ejemplo: *Satz*, *SATO*, *GRASP* y *Chaff*). Después, se presentan las mejoras que pueden incorporarse al procedimiento de Davis-Putnam para obtener un algoritmo competitivo: estructuras de datos optimizadas, heurísticas de selección de variable, backtracking no cronológico, aprendizaje de cláusulas, aleatorización y reinicios. Finalmente, se describen *GSAT* y *WalkSAT*, que son los algoritmos incompletos de búsqueda local más utilizados.

1. Introducción

La utilización de las fórmulas booleanas proposicionales como un lenguaje de programación con restricciones para solucionar problemas NP-completos es una área de investigación activa en Inteligencia Artificial. Este método genérico de resolución de problemas consiste en reducir un determinado problema al problema de la satisfactibilidad de las fórmulas booleanas proposicionales (SAT), solucionar la instancia obtenida con un algoritmo de satisfactibilidad y, a partir de la solución encontrada, generar una solución para el problema original.

Este nuevo enfoque de resolver problemas combinatorios ha demostrado ser muy competitivo en campos tan diversos como verificación de circuitos [24, 32], cuadrados latinos [17], planificación [18] y scheduling [4]. Ello, a su vez, ha supuesto un impulso para diseñar e implementar algoritmos altamente optimizados para resolver el problema SAT.

En este artículo se presenta una introducción a los algoritmos de satisfactibilidad. Primero, se describe el procedimiento de Davis-Putnam [7, 8], que constituye la base de la mayoría de algoritmos completos (por ejemplo: *Satz* [20], *SATO* [34], *GRASP* [25] y *Chaff* [27]). Después, se presentan las mejoras que pueden incorporarse al procedimiento de Davis-Putnam para obtener un algoritmo competitivo: estructuras de datos optimizadas, heurísticas de selección de variable, backtracking no cronológico, aprendizaje de cláusulas, aleatorización y reinicios. Finalmente, se describen *GSAT* [31] y *WalkSAT* [29], que son los algoritmos incompletos de búsqueda local más utilizados.

*Este trabajo ha sido financiado por el Ministerio de Ciencia y Tecnología, proyecto TIC2001-1577-C03-03.

A los lectores y a las lectoras que estén interesados en ampliar la información contenida en este artículo les remitimos a alguno de los *surveys* que se han publicado en los últimos años [14, 23, 36], y a dos artículos que describen algunos problemas abiertos en SAT [19, 30]. Para conocer los últimos avances, les remitimos a las actas del congreso anual sobre SAT (*International Conference on Theory and Applications of Satisfiability Testing*), que se publican en la serie *Lecture Notes in Artificial Intelligence* de Springer. En las actas de congresos y revistas generalistas de Inteligencia Artificial también se pueden encontrar regularmente trabajos sobre SAT.

El sitio web SATLIB [16][1] contiene enlaces a páginas relacionadas con SAT, bibliografía, benchmarks y el código fuente de varios algoritmos de satisfactibilidad que se han desarrollado en los últimos años.

2. Preliminares

En esta sección se define la sintaxis y la semántica de las fórmulas booleanas proposicionales, y el problema SAT, que es el primer problema que se demostró que era NP-completo [6]. A continuación, se muestra cómo se puede codificar el problema de coloración de grafos como una instancia de SAT. Finalmente, al incluirse este artículo en una monografía sobre programación con restricciones, se define el problema SAT como un problema de satisfacción de restricciones.

Definición 1 *Sea* $\texttt{Var} = \{x_1, x_2, \ldots, x_n\}$ *un conjunto de variables proposicionales. Un literal es una variable proposicional (literal con polaridad positiva) o una variable proposicional precedida de un símbolo de negación (literal con polaridad negativa). El complemento* $\overline{L}$ *de un literal* L *es* $\neg x$ *si* $L = x$ *y es* x *si* $L = \neg x$*. Una cláusula es una disyunción de literales. Una cláusula es unitaria si contiene un único literal. Una fórmula en forma normal conjuntiva es una conjunción de cláusulas.*

En este artículo, para referirse a una fórmula en forma normal conjuntiva se utiliza tanto el término *fórmula* como el término *base de datos de cláusulas*. La *cláusula vacía* se representa por $\square$ y la *fórmula vacía* por $\emptyset$.

[1] www.satlib.org

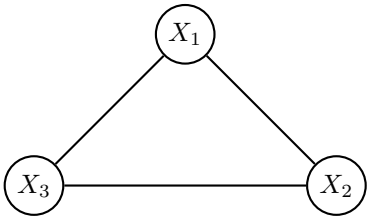

Figura 1: Ejemplo de grafo

Definición 2 *La longitud de una cláusula es el número total de literales que ocurren en la cláusula. La longitud de una fórmula es la suma de las longitudes de sus cláusulas.*

Definición 3 *Una interpretación es una función del conjunto de variables proposicionales* $\texttt{Var}$ *al conjunto de valores de verdad; i.e.,* $I : \texttt{Var} \to \{0, 1\}$*. Una interpretación* I *satisface un literal positivo* x *si* $I(x) = 1$*. Una interpretación* I *satisface un literal negativo* $\neg x$ *si* $I(x) = 0$*. Una interpretación satisface una cláusula si satisface al menos un literal de la cláusula. Una interpretación satisface una fórmula si satisface todas las cláusulas que ocurren en la fórmula.*

Definición 4 *El problema SAT es el problema de decidir si una fórmula es satisfactible.*

Ejemplo 1 *El problema de la* k*-coloración de un grafo consiste en decidir si los nodos del grafo pueden colorearse empleando* k *colores de manera que dos nodos adyacentes no tengan el mismo color. La codificación del problema de la 3-coloración para el grafo de la figura 1 como una instancia del problema SAT es la siguiente:*

- *El conjunto de variables proposicionales es*

  $$\{x_{11}, x_{12}, x_{13}, x_{21}, x_{22}, x_{23}, x_{31}, x_{32}, x_{33}\}.$$

  Una variable x_{ij} *se evalúa a 1 si, y sólo si, el vértice* X_i *tiene asignado el color* j*.*
- *La instancia SAT está formada por las siguientes cláusulas:*

Un conjunto de cláusulas que expresan que cada vértice tiene asignado un único color. Para ello, primero se añaden unas cláusulas que expresan que cada vértice tiene asignado al menos un color:

$$x_{11} \vee x_{12} \vee x_{13}$$
$$x_{21} \vee x_{22} \vee x_{23}$$
$$x_{31} \vee x_{32} \vee x_{33}$$

y, luego, se añaden unas cláusulas que expresan que cada vértice tiene asignado a lo sumo un color:

$$\begin{array}{lll} \neg x_{11} \vee \neg x_{12} & \neg x_{21} \vee \neg x_{22} & \neg x_{31} \vee \neg x_{32} \\ \neg x_{11} \vee \neg x_{13} & \neg x_{21} \vee \neg x_{23} & \neg x_{31} \vee \neg x_{33} \\ \neg x_{12} \vee \neg x_{13} & \neg x_{22} \vee \neg x_{23} & \neg x_{32} \vee \neg x_{33} \end{array}$$

Finalmente, se añade otro conjunto de cláusulas que expresan que vértices adyacentes tienen asignados colores diferentes:

$$\begin{array}{lll} \neg x_{11} \vee \neg x_{21} & \neg x_{12} \vee \neg x_{22} & \neg x_{13} \vee \neg x_{23} \\ \neg x_{11} \vee \neg x_{31} & \neg x_{12} \vee \neg x_{32} & \neg x_{13} \vee \neg x_{33} \\ \neg x_{21} \vee \neg x_{31} & \neg x_{22} \vee \neg x_{32} & \neg x_{23} \vee \neg x_{33} \end{array}$$

Obsérvese que, a partir de una interpretación que satisface la fórmula, se puede generar una coloración válida: si la variable x_{ij} tiene asignado el valor 1, se asigna el color j al vértice X_i.

El problema SAT puede definirse como un CSP[2] de la siguiente forma: cada variable proposicional se considera una variable CSP con dominio $\{0, 1\}$, y cada cláusula se representa como una restricción entre las variables que aparecen en la cláusula; las tuplas que pertenecen a la restricción corresponden a aquellas asignaciones de las variables que satisfacen la cláusula. Para conocer más sobre la relación entre SAT y CSP, remitimos a [33].

[2]Un *problema de satisfacción de restricciones (CSP)* se define por una terna (X, D, R), donde $X = \{X_1, X_2, \ldots, X_n\}$ es un conjunto finito de *variables*, $D = \{D_1, D_2, \ldots, D_n\}$ es un conjunto finito de *dominios*, y $R = \{R_1, R_2, \ldots, R_r\}$ es un conjunto finito de *restricciones*. Cada variable X_i toma valores en su correspondiente dominio D_i. Una restricción entre un subconjunto de variables $\{X_{i_1}, \ldots, X_{i_k}\}$ de X es un subconjunto del producto cartesiano $D_{i_1} \times \cdots \times D_{i_k}$ que está formado por las tuplas de valores que satisfacen la restricción.

3. Algoritmos de satisfactibilidad

Los algoritmos de satisfactibilidad pueden dividirse en dos clases: *algoritmos completos* y *algoritmos incompletos*. La mayoría de algoritmos completos están basados en el procedimiento de Davis-Putnam [7, 8], 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* (el algoritmo encuentra una interpretación que satisface la fórmula) o *insatisfactible* (el algoritmo explora todo el espacio de búsqueda y no encuentra ninguna interpretación que satisface la fórmula).

Los algoritmos de búsqueda local no exploran, en general, todo el espacio de búsqueda. Normalmente, empiezan generando una interpretación, de forma aleatoria, e intentan encontrar alguna interpretación que satisface la fórmula cambiando, en cada paso del algoritmo, el valor que tiene asignado una variable proposicional. Estos cambios se repiten hasta que se encuentra una interpretación que satisface la fórmula o hasta que se realiza un número de cambios prefijado. Este proceso se repite hasta que se encuentra una solución o hasta que se realiza 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 dificultad es que son incompletos y no pueden 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.

La ventaja de los algoritmos incompletos es que son capaces de resolver fórmulas satisfactibles que no se pueden resolver por ninguno de los algoritmos completos existentes. Por ejemplo, para ratios de número de cláusulas por número de variables de 4,25, se pueden solucionar ins-

tancias del problema 3-SAT aleatorio[3] que tengan como máximo 700 variables. Con el algoritmo *cnfs* [9], que es el algoritmo completo más eficiente para este tipo de instancias, se necesitan 25 días de CPU para resolver una instancia con 700 variables. Sin embargo, si se utiliza el algoritmo de búsqueda local *WalkSAT* [29], son suficientes dos horas para solucionar instancias con 100.000 variables. Recientemente, se ha desarrollado un algoritmo incompleto (*survey propagation* [5]), especialmente diseñado para resolver este tipo de instancias y basado en técnicas de Física Estadística, que sólo necesita dos horas para resolver instancias satisfactibles con un millón de variables.

3.1. Algoritmos completos

La mayoría de algoritmos completos para SAT están basados en la versión del procedimiento de Davis-Putnam [8] que describieron Davis, Logemann and Loveland en [7], y que se conoce como procedimiento *DPLL* en la bibliografía.

Entre los algoritmos completos para SAT que implementan el procedimiento *DPLL* están *POSIT* [10], *Satz* [20, 21], *RelSat* [3], *Satz-random* [12], *SATO* [34], *GRASP* [25], *cnfs* [9], *Chaff* [27] y *BerkMin* [11]. Estos algoritmos implementan el procedimiento *DPLL* e incorporan mejoras tales como estructuras de datos optimizadas para representar fórmulas, heurísticas de selección de variable, backtracking no cronológico, aprendizaje de cláusulas, aleatorización y reinicios.

3.1.1. El procedimiento *DPLL*

El procedimiento *DPLL* automatiza la aplicación de la *regla del literal unitario* y la *regla de ramificación*.

Dada una fórmula Γ que contiene una cláusula unitaria cuyo único literal es L, la regla del literal unitario (*one-literal rule*)[4] elimina de Γ todas las cláusulas que contienen L y elimina, del resto de cláusulas, todas las ocurrencias de $\overline{L}$. Es decir, se fija el valor del literal L a 1, puesto que si L se evaluase a 0, la fórmula sería insatisfactible.

La aplicación reiterada de la regla del literal unitario a una fórmula hasta derivar la cláusula vacía o alcanzar un estado de saturación se denomina *propagación unitaria* (*unit propagation*).[5] Al aplicar propagación unitaria a una fórmula Γ se obtiene una fórmula que es equisatisfactible con Γ, pero que es más sencilla de resolver porque tiene menos variables. Si se deriva la cláusula vacía, Γ es insatisfactible. En este caso, se dice que se ha identificado un *conflicto*. Si la propagación unitaria elimina todas las cláusulas, Γ es satisfactible. Si no se da ninguno de estos dos casos, no se puede decidir la satisfactibilidad de Γ. Obsérvese que, durante la aplicación de la regla del literal unitario, pueden derivarse nuevas cláusulas unitarias que ayudan a simplificar todavía más la fórmula original Γ.

Dada una fórmula Γ que contiene al menos una ocurrencia del literal L, la regla de ramificación (*branching rule*) reduce el problema de determinar la satisfactibilidad de Γ al problema de determinar si $\Gamma \cup \{L\}$ es satisfactible o $\Gamma \cup \{\overline{L}\}$ es satisfactible.

El procedimiento *DPLL* aplica la regla de ramificación a fórmulas que no contienen cláusulas unitarias; es decir, a fórmulas que no se pueden simplificar utilizando propagación unitaria. Puesto que la propagación unitaria preserva la satisfactibilidad, se puede reducir el problema de determinar la satisfactibilidad de Γ al problema de determinar si la fórmula que se obtiene al aplicar propagación unitaria a $\Gamma \cup \{L\}$ es satisfactible o la fórmula que se obtiene al aplicar propagación unitaria a $\Gamma \cup \{\overline{L}\}$ es satisfactible. De esta forma, dividimos un problema de satisfactibilidad en dos subproblemas más sencillos de resolver.

El procedimiento *DPLL* empieza aplicando propagación unitaria a la fórmula de entrada. Si no ha podido decidir su satisfactibilidad, aplica la regla de ramificación a la fórmula que ha obtenido al aplicar propagación unitaria y, a continuación, aplica propagación unitaria a ca-

[3] Dado un conjunto finito de variables proposicionales $\texttt{Var} = \{x_1, x_2, \ldots, x_n\}$, cada cláusula de una instancia del problema 3-SAT aleatorio (*Random 3-SAT*) se genera independientemente de las demás y se forma eligiendo uniformemente 3 literales, sin que haya dos iguales o complementarios, del conjunto de $2n$ literales que se pueden formar a partir de `Var`.

[4] También se conoce como *unit clause rule.*

[5] También se conoce como *Boolean Constraint Propagation (BCP).*

procedimiento DPLL
entrada: Γ
salida: verdadero si Γ es satisfactible,
falso en caso contrario

variable L: literal

Γ := propagación-unitaria(Γ);
si $\Gamma = \emptyset$ **entonces devolver** verdadero;
si $\square \in \Gamma$ **entonces devolver** falso;
L := selecciona-literal(Γ);
si DPLL($\Gamma \cup \{L\}$) **entonces**
devolver verdadero
sino
devolver DPLL($\Gamma \cup \{\overline{L}\}$);

procedimiento propagación-unitaria
entrada: Γ
salida: Γ tras aplicar propagación unitaria

variable L: literal
variable C: cláusula

mientras $\exists\{L\} \in \Gamma$ **y** $\square \notin \Gamma$ **hacer**
$\Gamma := \{C \,|\, L \notin C \in \Gamma\}$;
$\Gamma := \{C \setminus \{\overline{L}\} | C \in \Gamma\}$;
devolver Γ;

Figura 2: Procedimiento *DPLL*

da uno de los subproblemas generados. Estos pasos se repiten hasta que se deriva la fórmula vacía en algún subproblema (entonces, la fórmula de entrada es satisfactible) o hasta que se ha derivado la cláusula vacía en todos los subproblemas (entonces, la fórmula de entrada es insatisfactible).

El pseudocódigo del procedimiento *DPLL* se muestra en la figura 2. Este procedimiento puede verse como la construcción de un árbol de prueba. Los nodos del árbol están etiquetados con fórmulas, siendo la fórmula de entrada la etiqueta del nodo raíz. Cuando se aplica la regla de ramificación a un nodo, se generan dos descendientes. Dado un nodo etiquetado con una fórmula Γ, *selecciona-literal* selecciona un literal L de Γ utilizando algún tipo de *heurística de selección de variable*, y los descendientes de Γ son las fórmulas que se obtienen al aplicar propagación unitaria a $\Gamma \cup \{L\}$ y a $\Gamma \cup \{\overline{L}\}$.

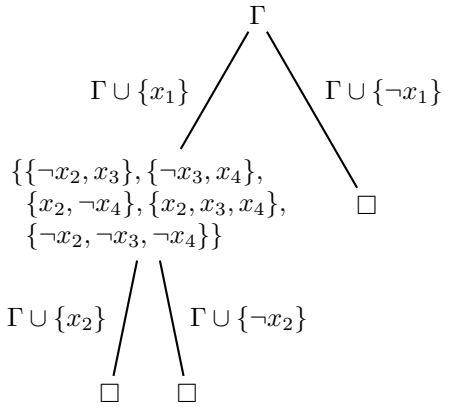

Figura 3: Un árbol de prueba *DPLL* para la fórmula Γ del ejemplo 2

El árbol de prueba se crea utilizando la estrategia de profundidad en prioridad, y el procedimiento hace backtracking cuando se detecta una cláusula vacía. La búsqueda termina cuando se deriva la fórmula vacía (en este caso, la fórmula de entrada es satisfactible) o cuando al hacer backtracking se visita el nodo raíz (en este caso, todas las hojas del árbol contienen la cláusula vacía y la fórmula de entrada es insatisfactible).

Ejemplo 2 *Sea la fórmula insatisfactible:*

$$\begin{aligned}\Gamma \;=\; & \{\{\neg x_1, \neg x_2, x_3\}, \{\neg x_1, \neg x_3, x_4\}, \{x_1, x_5\}, \\ & \{x_2, \neg x_4\}, \{x_2, x_3, x_4\}, \{\neg x_2, \neg x_3, \neg x_4\}, \\ & \{x_1, \neg x_5\}\}\end{aligned}$$

La figura 3 muestra el árbol de prueba creado por el procedimiento DPLL *para* Γ.

Una implementación del procedimiento *DPLL*, para ser competitiva, deberá tener en cuenta los siguientes aspectos: estructuras de datos, heurísticas de selección de variable, aprendizaje de cláusulas dirigido por conflictos, bracktracking no cronológico, reinicios y aleatorización.

Estructuras de datos

Las estructuras de datos que se utilizan para representar fórmulas son decisivas para realizar, eficientemente, las operaciones que constituyen un algoritmo de satisfactibilidad. En particular, una elección adecuada de las estructuras

de datos permite implementar la propagación unitaria con una complejidad lineal en el peor de los casos. Hay que tener en cuenta que las implementaciones existentes del procedimiento *DPLL* consumen gran parte del tiempo de ejecución aplicando propagación unitaria. Según los autores de *Chaff* [27], representa más del 90 % del tiempo de CPU en la mayoría de instancias.

A continuación, se describen dos tipos de estructuras de datos para algoritmos basados en el procedimiento *DPLL*: estructuras de datos basadas en contadores y estructuras de datos perezosas (*lazy*). Para ambos tipos de estructuras, se explica cómo identificar eficientemente cuándo una cláusula se satisface o no se satisface para cierta interpretación, así como identificar cuándo una cláusula es unitaria. Para ampliar la información sobre estructuras de datos para algoritmos completos, remitimos a [22].

En las estructuras de datos basadas en contadores, las cláusulas se representan como listas de literales y, para cada variable, se mantiene una lista de *todas las cláusulas* que contienen alguna ocurrencia de dicha variable. Para saber si una cláusula se satisface, no se satisface o es unitaria, se asocia, a cada cláusula, un contador de literales que se satisfacen y un contador de literales que no se satisfacen. Una cláusula no se satisface si el contador de literales que no se satisfacen es igual al número de literales de la cláusula. Una cláusula se satisface si el contador de literales que se satisfacen es mayor o igual que uno. La cláusula es unitaria si el contador de literales que no se satisfacen es igual al número de literales de la cláusula menos uno y queda un literal sin asignar. Cuando una cláusula se declara unitaria, se recorre la lista de sus literales para identificar el literal que todavía no tiene asignado ningún valor. Cuando se detecta un conflicto y se hace backtracking, se deben actualizar los contadores de todas las cláusulas en las que aparece alguna variable que pasa de estar asignada a estar no asignada. Este tipo de estructuras de datos son la base de las que incorporan *POSIT* [10], *Satz* [20, 21], *RelSat* [3] y *GRASP* [25].

Las estructuras de datos basadas en contadores presentan el inconveniente de que, cada vez que se asigna o desasigna un valor a una variable, hay que analizar todas las cláusulas en las que aparece dicha variable. Ello resulta particularmente costoso cuando se resuelven instancias que tienen muchas ocurrencias de una misma variable, y en algoritmos que aprenden cláusulas cada vez que se detecta algún conflicto. En este último caso, el número de ocurrencias aumenta a medida que avanza la búsqueda.

Es suficiente que, cuando se asigna un valor a una variable, se pueda identificar qué cláusulas pasan a ser unitarias y qué cláusulas pasan a no satisfacerse. Como veremos, este problema se soluciona incorporando estructuras de datos perezosas. Las estructuras de datos perezosas se caracterizan por mantener, para cada variable, una lista de *un número reducido de cláusulas* en las que aparece dicha variable. Estas estructuras de datos se basan en la observación de que una cláusula que tiene más de un literal sin asignar no puede ser declarada ni unitaria ni no satisfecha.

SATO [34] fue el primer algoritmo de satisfactibilidad con estructuras de datos perezosas. A continuación, se describen las estructuras de datos de Chaff, conocidas como estructuras con dos literales centinelas (*2-literal watching*). Son una mejora de las estructuras de SATO que tienen la ventaja de que hacer backtracking tiene un coste constante.

En el esquema de 2 literales centinelas, cada cláusula tiene dos apuntadores a dos literales de la cláusula, que se les conoce como *literales centinela*. No se impone ningún orden a la hora de escoger los literales centinela, y cada apuntador puede moverse en cualquier dirección. Inicialmente, las variables de los literales centinela no tienen asignado ningún valor. Además, cada variable x tiene una lista de apuntadores a las ocurrencias positivas de la variable que son centinelas (positive-watched(x)) y una lista de apuntadores a las ocurrencias negativas de la variable que son centinelas (negative-watched(x)).

Cuando se asigna el valor 1 a una variable x, para cada ocurrencia de $\neg x$ de la lista negative-watched(x), el algoritmo busca un literal l en la cláusula que ocurre $\neg x$ que no se evalúe a 0. Durante esta búsqueda, pueden presentarse cuatro casos:

1. Si existe un literal L que no es el otro literal centinela, entonces se elimina de negative-watched(x) el apuntador a $\neg x$ y se añade,

en la lista de literales centinela de la variable de L, un apuntador a L. En este caso, se dice que se ha movido el literal centinela.

2. Si el único literal L que existe es el otro literal centinela y su variable no tiene asignado ningún valor, entonces tenemos una cláusula unitaria cuyo único literal es el otro literal centinela.

3. Si el único literal L que existe es el otro literal centinela y se evalúa a 1, entonces no se hace nada.

4. Si todos los literales de la cláusula se evalúan a 0 y no existe ningún literal L, la cláusula es una cláusula vacía.

Cuando el algoritmo hace backtracking, el coste de deshacer la asignación de un valor a una variable es constante. Ello es debido a que los dos literales centinelas son los últimos a cuyas variables se les ha asignado un valor que ha llevado a que estos literales se evalúen a 0. Por tanto, al hacer backtracking, no hay que modificar los apuntadores de estos literales porque se evalúan a 1 o no tienen asignado ningún valor.

Heurísticas de selección de variable

La heurística utilizada para seleccionar la variable que interviene en la regla de ramificación es otro aspecto decisivo para desarrollar algoritmos de satisfactibilidad competitivos. El tamaño del árbol de prueba generado, así como el tiempo de resolución, son muy sensibles a esta heurística.

Una buena heurística de selección de variable debe mostrar un compromiso entre la habilidad que tiene de reducir el espacio de búsqueda y el coste computacional de calcularla.

Una heurística sencilla y bastante eficiente es MOMS.[6] MOMS escoge una variable entre aquellas que más aparecen en cláusulas de longitud mínima. Intuitivamente, estas variables permiten generar más cláusulas unitarias durante la aplicación de la propagación unitaria. Cuantas más cláusulas unitarias se generan, más se simplifica la fórmula y aumenta la probabilidad de detectar antes un conflicto o de derivar la fórmula vacía.

[6]El acrónimo MOMS viene de Most Often in Minimal Size clauses.

Otra heurística es UP.[7] UP explota la potencia de la propagación unitaria más que MOMS. Para cada variable proposicional x que ocurre en un fórmula Γ, UP aplica propagación unitaria a $\Gamma \cup \{x\}$ y a $\Gamma \cup \{\neg x\}$, y asigna un peso a la variable x en función de los resultados obtenidos al aplicar propagación unitaria; por ejemplo, considera como peso el número de cláusulas unitarias generadas. Como efecto secundario, UP permite detectar literales fallidos (*failed literals*): si al aplicar propagación unitaria a $\Gamma \cup \{x\}$ ($\Gamma \cup \{\neg x\}$) se detecta un conflicto, se fija el valor de x a 0 (1). Esta heurística suele generar árboles de prueba más pequeños que MOMS pero es más costosa de calcular.

Satz incorpora una heurística bastante sofisticada, que ha dado muy buenos resultados, en la que se combinan las ventajas de MOMS y UP. *Satz* aplica UP a un cierto número de variables que ocurren en cláusulas binarias. Para seleccionar las variables a las que aplica UP, utiliza un predicado, llamado $PROP_z$, que se define de la siguiente forma:

Definición 5 *Sea Γ una formula; sea $PROP(x, i)$ un predicado binario que es verdadero si, y sólo si, la variable proposicional x ocurre tanto negada como sin negar en cláusulas binarias de Γ, y hay al menos i ocurrencias de x en cláusulas binarias de Γ; y sea T un entero. $PROP_z(x)$ se define como el primero de los predicados $PROP(x, 4)$, $PROP(x, 3)$, verdadero (en este orden) cuya semántica denotacional contiene más de un número fijo de variables T. El valor de T en Satz es 10.*

La figura 4 muestra el pseudocódigo de la heurística de selección de variable de *Satz*. Después de aplicar propagación unitaria a un cierto número de variables y detectar literales fallidos, la heurística asigna un peso a cada literal teniendo en cuenta el número de cláusulas de longitud mínima que se han generado. La función $H(x)$, que fue primero incorporada en *POSIT* [10], se utiliza para conseguir un buen equilibrio entre pesos de literales positivos y negativos.

La aparición de *Chaff* [27] supuso un nuevo enfoque en el diseño de heurísticas de selección de

[7]UP es el acrónimo de Unit Propagation.

para cada variable no asignada x **tal que** $PROP_z(x)$ es verdadero **hacer**
(sean Γ' y Γ'' dos copias de la fórmula Γ que estamos considerando)

Γ' := propagación-unitaria($\Gamma' \cup \{x\}$);
Γ'' := propagación-unitaria($\Gamma'' \cup \{\neg x\}$);
si $\square \in \Gamma'$ **y** $\square \in \Gamma''$ **entonces devolver** "Γ es insatisfactible";
si $\square \in \Gamma'$ **entonces**
x:=0; $\Gamma := \Gamma''$;
sino si $\square \in \Gamma''$ **entonces**
x:=1, $\Gamma := \Gamma'$
si $\square \notin \Gamma'$ **y** $\square \notin \Gamma''$ **entonces**
(sea L un literal, y $w(L)$ el peso de L)
$w(x)$:= número de cláusulas de longitud mínima de Γ' que no aparecen en Γ;
$w(\neg x)$:= número de cláusulas de longitud mínima de Γ'' que no aparecen en Γ;

para cada variable no asignada x **hacer**
$H(x) := w(x) * w(\neg x) * 1024 + w(x) + w(\neg x)$;
seleccionar una variable x cuyo valor de $H(x)$ sea máximo;

Figura 4: Heurística de selección de variable de *Satz*

variable para algoritmos de satisfactibilidad. La motivación de los autores de *Chaff* era desarrollar una heurística que fuese rápida e independiente del estado de la variable, puesto que las heurísticas que se han descrito antes resultan poco eficientes para fórmulas que contienen un gran número de cláusulas.

La heurística de *Chaff* es VSIDS[8] y se define de la siguiente forma [27]:

1. Cada variable tiene un contador para cada polaridad, el cual está inicialmente a 0.
2. Cuando se añade una cláusula a la base de datos de cláusulas, se incrementa el contador asociado con cada uno de los literales de la cláusula. Hay que tener en cuenta que se añaden cláusulas tanto al principio como durante la ejecución (las que se aprenden al analizar conflictos).
3. Se escoge la variable (no asignada) y polaridad con mayor contador.
4. Los empates se resuelven aleatoriamente, aunque este punto es configurable.
5. Periódicamente, todos los contadores se dividen por una constante. Por defecto, esta constante es 2.

[8]VSIDS es el acrónimo de *Variable State Independent Decaying Sum.*

La idea es escoger un literal que aparezca mucho en cláusulas que se han añadido recientemente como consecuencia del análisis de conflictos.

En programación con restricciones, donde la cardinalidad del dominio de las variables es mayor o igual que dos, las heurísticas, generalmente, priorizan la selección de variables cuyo dominio es mínimo. En cambio, en SAT, puesto que todas las variables son booleanas, tales heurísticas no parecen, a priori, ventajosas. Sin embargo, en [1, 2], se muestra cómo identificar conjuntos de variables booleanas que representan una variable CSP, cómo identificar su dominio y, finalmente, cómo incorporar heurísticas de CSPs en SAT que mejoran notablemente el rendimiento de los algoritmos de satisfactibilidad *Satz* y *Chaff*.

Aprendizaje de cláusulas y backtracking no cronológico

Cuando se detecta un conflicto, el procedimiento *DPLL* hace backtracking y reinicia la búsqueda en la primera rama que encuentra que no ha sido todavía explorada; a este tipo de backtracking se le denomina backtracking cronológico.

Los algoritmos de satisfactibilidad más modernos (por ejemplo: *RelSat* [3], *SATO* [34], *GRASP* [25], *Chaff* [27] y *BerkMin* [11]) incor-

poran un procedimiento de análisis de conflictos que se ejecuta cada vez que se detecta un conflicto. Este procedimiento encuentra la razón que provocó el conflicto, y lo intenta resolver informando al algoritmo de satisfactibilidad de que no hay solución en cierta parte del espacio de búsqueda e indicándole dónde debe continuar explorando. Normalmente, el algoritmo continúa la búsqueda en un nodo anterior al nodo que regresaría con backtracking cronológico, podando de esta forma una parte del espacio de búsqueda en la que no hay ninguna solución.

La razón que provocó el conflicto se representa mediante cláusulas, que se añaden a la base de datos de cláusulas original. Estas cláusulas que se aprenden del análisis de conflictos son redundantes, pero evitan que se repita un conflicto por la misma razón que provocó un conflicto anterior. Sin ellas, se tardaría más en detectar conflictos en partes del espacio de búsqueda que todavía no han sido exploradas.

Para ampliar la información sobre backtracking no cronológico y aprendizaje de cláusulas dirigido por conflictos, remitimos a [25, 35, 36].

Reinicios y aleatorización

El tiempo de CPU que se necesita para resolver instancias similares con algoritmos completos, a menudo, varía mucho. Por ejemplo, dos instancias que sólo se diferencian en el orden de las variables pueden tardar tiempos muy diferentes; una puede resolverse en segundos mientras que la otra puede tardar varios días. Este comportamiento ha sido estudiado en [13], donde se propuso el uso de reinicios aleatorios para mitigarlo. De forma intuitiva, una mala elección al inicio de la búsqueda puede llevar a explorar partes muy grandes del espacio de búsqueda en las que no existe solución. Para evitar estas situaciones, lo mejor es reiniciar la búsqueda cada cierto tiempo y dirigirla en otra dirección.

Una manera de cambiar el rumbo de la búsqueda es introducir aleatorización en la heurística de selección de variable. De esta forma, cuando se reinicia la búsqueda, se exploran nuevas partes del espacio de búsqueda.

En los algoritmos que incorporan aprendizaje, las cláusulas que se han aprendido permanecen en la base de datos de cláusulas cuando se reinicia la búsqueda. En este caso, la situación de partida varía en cada reinicio.

3.2. Algoritmos incompletos

En esta sección se describen *GSAT* [31] y *WalkSAT* [26, 29], que son procedimientos incompletos de búsqueda local ampliamente utilizados por la comunidad de Inteligencia Artificial.

Las aproximaciones para resolver SAT basadas en algoritmos genéticos o redes neuronales no han demostrado ser, de momento, competitivas frente a *GSAT* y *WalkSAT*. El único método incompleto que ha superado *GSAT* y *WalkSAT* es *survey propagation* [5]. El inconveniente de *survey propagation* es que sólo sirve para resolver instancias del problema 3-SAT aleatorio. Para un análisis comparativo del comportamiento de *GSAT* y *WalkSAT* en una amplia colección de benchmarks, remitimos a [15].

3.2.1. Procedimiento *GSAT*

El procedimiento *GSAT* parte de una interpretación generada aleatoriamente, y la intenta mejorar incrementalmente buscando una mejor dentro de una vecindad. La vecindad la forman todas las interpretaciones que difieren de la actual en el valor que toma una variable.

En cada iteración, se cambia el valor de una de las variables de la interpretación actual y así se obtiene una nueva interpretación. La variable que se cambia es una de las que da el mayor decremento posible en el número total de cláusulas no satisfechas. Cada un cierto número prefijado de iteraciones (MAX_CAMBIOS), se reinicia la búsqueda para escapar de mínimos locales. Este proceso se repite un número prefijado de veces (MAX_INTENTOS). El pseudocódigo de *GSAT* se muestra en la figura 5.

El rendimiento de *GSAT* puede mejorarse considerablemente incorporando búsqueda tabú, para evitar visitar interpretaciones ya exploradas, y *random walks*, para escapar de mínimos locales [28].

procedimiento GSAT

entrada: Γ, MAX_INTENTOS y MAX_CAMBIOS
salida: una interpretación que satisface Γ, si la encuentra
variable i, j: entero
variable S: lista
variable x: variable
variable I: interpretación

para $i := 1$ **hasta** MAX_INTENTOS **hacer**
  $I :=$ una interpretación para Γ generada aleatoriamente;
  para $j := 1$ **hasta** MAX_CAMBIOS **hacer**
    si I satisface Γ **entonces**
      devolver I;
    sino
      $S :=$ conjunto de variables proposicionales tales que
          si se cambia el valor que les asigna I se obtiene
          el mayor decremento posible en el número total
          de cláusulas no satisfechas;
      $x :=$ una variable de S;
      $I := I$ con la asignación a x cambiada;
devolver "no se ha encontrado una interpretación que satisfaga Γ";

Figura 5: Procedimiento *GSAT*

procedimiento WalkSAT

entrada: Γ, MAX_INTENTOS, MAX_CAMBIOS y ω
salida: una interpretación que satisface Γ, si la encuentra

variable i, j, u: entero
variable S: lista
variable x: variable
variable C: cláusula
variable I: interpretación

para $i := 1$ **hasta** MAX_INTENTOS **hacer**
  $I :=$ una interpretación para Γ generada aleatoriamente;
  para $j := 1$ **hasta** MAX_CAMBIOS **hacer**
    si I satisface Γ **entonces**
      devolver I;
    sino
      $C :=$ una cláusula de Γ no satisfecha por I;
      $S :=$ conjunto de variables proposicionales que aparecen en C;
      $u := min\,(\ \{rotas(x, \Gamma, I) \mid x \in S\}\ )$;
      si $(u = 0)$ **entonces**
        $x :=$ una variable de S tal que $rotas(x, \Gamma, I) = 0$;
      si $(u > 0)$ **entonces**
        con probabilidad ω :
          $x :=$ una variable de S;
        con probabilidad $1 - \omega$:
          $x :=$ una variable de S tal que $rotas(x, \Gamma, I) = u$;
      $I := I$ con la asignación a x cambiada;
devolver "no se ha encontrado una interpretación que satisfaga Γ";

Figura 6: Procedimiento *WalkSAT*

3.2.2. Procedimiento *WalkSAT*

El procedimiento *WalkSAT* también parte de una interpretación generada aleatoriamente y la intenta mejorar incrementalmente, pero sigue otro criterio a la hora de seleccionar la variable a la que se le cambia el valor en cada iteración.

En *WalkSAT*, se selecciona una cláusula C que no está satisfecha por la interpretación actual y, luego, se cambia el valor de una de las variables de C para obtener una nueva interpretación. Si hay variables de C que no *rompen* cláusulas al cambiar su valor (es decir, que no provocan que cláusulas que antes se satisfacían ahora no se satisfacen), se cambia el valor de una de estas variables. Si todas las variables de C rompen cláusulas al cambiar su valor, entonces (i) con probabilidad ω ($\omega \in [0, 1]$), se cambia el valor de una variable de C y (ii) con probabilidad $1 - \omega$, se cambia el valor de una de las variables que rompen menos cláusulas. Como en *GSAT*, tenemos un número prefijado de cambios e intentos. El pseudocódigo de *WalkSAT* se muestra en la figura 6. La función $rotas(x, \Gamma, I)$ devuelve el número de cláusulas de Γ satisfechas por I que no se satisfacen si se cambia la asignación a la variable x.

En general, *WalkSAT* es superior a *GSAT*. El esquema básico de *WalkSAT* puede mejorarse introduciendo otras heurísticas de selección de variable más sofisticadas tales como *Novelty* y *R-Novelty* [26].

Referencias

[1] C. Ansótegui, J. Larrubia, C. M. Li, y F. Manyà. Mv-Satz: A SAT solver for many-valued clausal forms. En *4th International Conference Journées de L'Informatique Messine, JIM-2003, Metz, France*, 2003.

[2] C. Ansótegui, J. Larrubia, y F. Manyà. Boosting Chaff's performance by incorporating CSP heuristics. En *9th International Conference on Principles and Practice of Constraint Programming, CP-2003, Kinsale, Ireland*, páginas 96–107. Springer LNCS 2833, 2003.

[3] R. J. Bayardo y R. C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. En *Proceedings of the 14th National Conference on Artificial Intelligence, AAAI'97, Providence/RI, USA*, páginas 203–208. AAAI Press, 1997.

[4] R. Béjar y F. Manyà. Solving the round robin problem using propositional logic. En *Proceedings of the 17th National Conference on Artificial Intelligence, AAAI-2000, Austin/TX, USA*, páginas 262–266. AAAI Press, 2000.

[5] A. Braunstein, M. Mezard, y R. Zecchina. Survey propagation: An algorithm for satisfiability. Preprint, 2002.

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

[7] M. Davis, G. Logemann, y D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.

[8] M. Davis y H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[9] O. Dubois y G. Dequen. A backbone-search heuristic for efficient solving of hard 3-SAT formulae. En *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI'01, Seattle/WA, USA*, páginas 248–253, 2001.

[10] J. W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, PA, USA, 1995.

[11] E. Goldberg y Y.Ñovikov. BerkMin: A fast and robust SAT solver. En *Proceedings of Design, Automation and Test in Europe, DATE-2002, Paris, France*, páginas 142–149. IEEE Computer Society, 2001.

[12] C. Gomes, B. Selman, N. Crato, y H. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24(1/2):67–100, 2000.

[13] C. P. Gomes, B. Selman, y H. Kautz. Boosting combinatorial search through randomization. En *Proceedings of the 15th*

National Conference on Artificial Intelligence, AAAI'98, Madison/WI, USA, páginas 431–437. AAAI Press, 1998.

[14] J. Gu, P. Purdom, J. Franco, y B. Wah. Algorithms for the satisfiability (SAT) problem: A survey. En D. Du, J. Gu, y P. Pardalos, editores, *Satisfiability Problem: Theory and Applications*, volume 35 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, páginas 19–152. 1997.

[15] H. H. Hoos y T. Stützle. Local search algorithms for SAT: An empirical evaluation. *Journal of Automated Reasoning*, 24(4):421–481, 2000.

[16] H. H. Hoos y T. Stützle. SATLIB: An online resource for research on SAT. En I. Gent, H. van Maaren, y T. Walsh, editores, *SAT2000. Highlights of Satisfiability Research in the Year 2000*, volume 63 of *Frontiers in Artificial Intelligence and Applications*, páginas 283–292. IOS Press, 2000.

[17] H. A. Kautz, Y. Ruan, D. Achlioptas, C. P. Gomes, B. Selman, y M. Stickel. Balance and filtering in structured satisfiable problems. En *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI'01, Seattle/WA, USA*, páginas 351–358, 2001.

[18] H. A. Kautz y B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. En *Proceedings of the 14th National Conference on Artificial Intelligence, AAAI'96, Portland/OR, USA*, páginas 1194–1201. AAAI Press, 1996.

[19] H. A. Kautz y B. Selman. Ten challenges redux: Recent progress in propositional reasoning and search. En *9th International Conference on Principles and Practice of Constraint Programming, CP-2003, Kinsale, Ireland*, páginas 1–18. Springer LNCS 2833, 2003.

[20] C. M. Li y Anbulagan. Heuristics based on unit propagation for satisfiability problems. En *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI'97, Nagoya, Japan*, páginas 366–371. Morgan Kaufmann, 1997.

[21] C. M. Li y Anbulagan. Look-ahead versus look-back for satisfiability problems. En *Proceedings of the 3rd International Conference on Principles of Constraint Programming, CP'97, Linz, Austria*, páginas 341–355. Springer LNCS 1330, 1997.

[22] I. Lynce y J. P. Marques-Silva. Efficient data structures for backtrack search SAT solvers. En *Fifth International Symposium on the Theory and Applications of Satisfiability Testing, SAT-2002, Cincinnati, USA*, páginas 308–315, 2002.

[23] I. Lynce y J. P. Marques-Silva. An overview of backtrack search satisfiability algorithms. *Annals of Mathematics and Artificial Intelligence*, 37:307–326, 2003.

[24] J. P. Marques-Silva y L. Guerra. Algorithms for satisfiability in combinational circuits based on backtrack search and recursive learning. En *Proc. of XII Symposium on Integrated Circuits and Systems Design (SBCCI)*, 1999.

[25] J. P. Marques-Silva y K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[26] D. McAllester, B. Selman, y H. Kautz. Evidence for invariants in local search. En *Proceedings of the 14th National Conference on Artificial Intelligence, AAAI'97, Providence/RI, USA*, páginas 321–326. AAAI Press, 1997.

[27] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, y S. Malik. Chaff: Engineering an efficient sat solver. En *39th Design Automation Conference*, 2001.

[28] B. Selman y H. A. Kautz. Domain-independent extensions of GSAT: Solving large structured satisfiability problems. En *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI'93, Chambery, France*, páginas 290–295. Morgan Kaufmann, 1993.

[29] B. Selman, H. A. Kautz, y B. Cohen. Noise strategies for improving local search. En *Proceedings of the 12th National Conference on Artificial Intelligence, AAAI'94, Seattle/WA, USA*, páginas 337–343. AAAI Press, 1994.

[30] B. Selman, H. A. Kautz, y D. McAllester. Ten challenges in propositional reasoning and search. En *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI'97, Nagoya, Japan*, páginas 50–54. Morgan Kaufmann, 1997.

[31] B. Selman, H. Levesque, y D. Mitchell. A new method for solving hard satisfiability problems. En *Proceedings of the 10th National Conference on Artificial Intelligence, AAAI'92, San Jose/CA, USA*, páginas 440–446. AAAI Press, 1992.

[32] M. Velev y R. Bryant. Effective use of boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors. En *38th Design Automation Conference (DAC '01)*, 2001.

[33] T. Walsh. SAT v CSP. En *Proceedings of the 6th International Conference on Principles of Constraint Programming, CP-2000, Singapore*, páginas 441–456. Springer LNCS 1894, 2000.

[34] H. Zhang. SATO: An efficient propositional prover. En *Conference on Automated Deduction (CADE-97)*, páginas 272–275, 1997.

[35] L. Zhang, C. Madigan, M. Moskewicz, y S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. En *International Conference on Computer Aided Design, ICCAD-2001, San Jose/CA, USA*, páginas 279–285, 2001.

[36] L. Zhang y S. Malik. The quest for efficient Boolean satisfiability solvers. En *18th International Conference on Automated Deduction, CADE-18, Copenhagen, Denmark*, páginas 295–313. Springer, LNCS 2392, 2002.