Universidad de La Habana
Facultad de Matemática y Computación

Análisis comparativo de métodos aproximados en solucionadores CDCL SAT

Autor:
Massiel Paz Otaño

Tutores:
Dr. Luciano García Garrido

Trabajo de Diploma
presentado en opción al título de
Licenciada en Ciencia de la Computación

Fecha

github.com/NinaSayers/Application-of-Computational-Logic-in-Problem-Solving.git

Dedicación

Agradecimientos

Agradecimientos

Opinión del tutor

Opiniones de los tutores

Resumen

Resumen en español

Abstract

Resumen en inglés

Índice general

Índice de figuras

Ejemplos de código

Introducción

El desarrollo de la lógica computacional como disciplina se enmarca en la revolución tecnológica del siglo XX, impulsada por la necesidad de resolver problemas complejos en ámbitos como la inteligencia artificial, la verificación de hardware y software, y la optimización industrial. La creciente demanda de sistemas automatizados capaces de procesar restricciones y tomar decisiones eficientes llevó a la comunidad científica a explorar métodos formales para modelar y resolver problemas combinatorios. En este escenario, la teoría de la complejidad computacional emergió como un pilar fundamental, especialmente tras la identificación de la clase NP-Completo por Cook en 1971, que transformó la comprensión de los límites de la computación.

Los problemas con restricciones —aquellos que requieren satisfacer un conjunto de condiciones lógicas— han sido centrales en áreas como la planificación, la criptografía y el diseño de circuitos. El problema de satisfacibilidad booleana (SAT), demostrado por Cook como el primer problema NP-Completo, se convirtió en la piedra angular para estudiar la viabilidad de soluciones eficientes. Aunque los primeros algoritmos para SAT, como el método de Davis-Putnam (DP) y su evolución, Davis-Putnam-Logemann-Loveland (DPLL), sentaron las bases de los resolvedores (solvers), su eficiencia se veía limitada por la explosión combinatoria en instancias complejas. La presencia de cláusulas unitarias, la selección subóptima de variables y el retroceso (backtrack) cronológico exponían claras debilidades, especialmente en problemas con miles de variables.

A pesar de los avances, los SAT resolvedores (solvers) clásicos enfrentaban un desafío crítico: escalar sin sacrificar completitud. Esto motivó la búsqueda de mejoras heurísticas y estratégicas, como el aprendizaje de cláusulas y el retroceso (backtrack) no cronológico, que culminaron en el surgimiento del paradigma Conflict-Driven Clause Learning (CDCL). CDCL no solo optimizó la exploración del espacio de soluciones, sino que introdujo mecanismos para evitar repeticiones de conflictos, marcando un hito en la resolución práctica de problemas NP-Completos.

El núcleo de la eficiencia de los SAT solvers modernos reside, sin lugar a dudas, en su capacidad para reducir el espacio de búsqueda de forma inteligente. Sin embargo, incluso con técnicas como CDCL, un desafío persiste: la selección óptima de variables. Esta elección determina la dirección en la que el algoritmo explora el árbol de decisiones, y una estrategia subóptima puede llevar a ciclos de conflicto-reparación redundantes, incrementando exponencialmente el tiempo de ejecución. En problemas NP-Completos, donde el número de posibles asignaciones crece como 2^n (con n variables), una heurística de selección inadecuada convierte instancias resolubles en minutos en problemas intratables.

En CDCL, tras cada conflicto, el resolvedor aprende una cláusula nueva para evitar repeticiones. No obstante, la eficacia de este aprendizaje depende de qué variables se eligieron para bifurcar el espacio de soluciones. Si se seleccionan variables irrelevantes o poco conectadas a los conflictos, las cláusulas aprendidas serán débiles o redundantes, limitando su utilidad. Así, la selección de variables no es solo una cuestión de orden, sino de calidad de la exploración.

Dos de las heurísticas de selección de variables son VSIDS (Variable State Independent Decaying Sum) y DLIS (Dynamic Largest Individual Sum). Ambas, son aproximaciones greedy, dado que optimizan localmente (paso a paso) sin garantizar una solución global óptima. Su eficacia depende de cómo la estructura del problema se alinee con sus criterios. Por una parte, VSIDS asigna un puntaje a cada variable, incrementándolo cada vez que aparece en una cláusula involucrada en un conflicto. Periódicamente, estos puntajes se reducen (*decaimiento* exponencial), priorizando variables activas recientemente. Por otra parte, DLIS calcula, para cada literal (variable o su negación), el número de cláusulas no satisfechas donde aparece. Selecciona el literal con mayor frecuencia y asigna su variable correspondiente.

Hoy, aunque los SAT resolvedores basados en CDCL dominan aplicaciones críticas, desde la verificación formal de chips hasta la síntesis de programas, su rendimiento varía significativamente según el tipo de problema (p. ej., aleatorios vs. estructurados) y las heurísticas empleadas. Mientras VSIDS prioriza variables recientemente involucradas en conflictos —útil en problemas con alta estructura local—, DLIS enfatiza la frecuencia de aparición de literales, mostrando ventajas en dominios con distribución uniforme de restricciones. Esta dualidad plantea preguntas clave: ¿bajo qué métricas (tiempo de ejecución, memoria, escalabilidad) una estrategia supera a la otra? ¿Cómo influye la naturaleza del problema en su eficiencia?

Esta tesis aporta una comparación sistemática entre VSIDS y DLIS dentro de entornos CDCL, evaluando su desempeño en problemas heterogéneos (industriales, aleatorios y académicos). A diferencia de estudios previos, se integran métricas adaptativas que consideran no solo el tiempo de resolución, sino también el impacto de las cláusulas aprendidas y la distribución de conflictos. Además, se propone un marco teórico para clasificar problemas según su afinidad heurística, contribuyendo a la selección informada de algoritmos en aplicaciones reales.

Teóricamente, este trabajo profundiza en la relación entre estructura de problemas y heurísticas, enriqueciendo la comprensión de CDCL. Prácticamente, ofrece directrices para ingenieros y desarrolladores de resolvedores, optimizando recursos en áreas como la verificación de software o la logística, donde minutos de mejora equivalen a ahorros millonarios.

Diseño teórico

- **Problema científico:** Ineficiencia de los SAT resolvedores ante problemas con distintas estructuras, asociada a la selección subóptima de variables.
- **Objeto de estudio:** Algoritmos CDCL con estrategias VSIDS y DLIS.
- **Objetivos:**
  - Analizar el impacto de VSIDS y DLIS en el rendimiento de CDCL.
  - Establecer correlaciones entre tipos de problemas y heurísticas eficaces.
- **Campo de acción:** Lógica computacional aplicada a la rersolución de problemas con restricciones.
- **Hipótesis:** El rendimiento de VSIDS y DLIS varía significativamente según la densidad de restricciones, la presencia de patrones locales y el balance entre cláusulas aprendidas y originales.

El documento se organiza en cinco capítulos:

- **Fundamentos de SAT y NP-Completitud:** Revisión teórica de problemas con restricciones y complejidad.
- **Evolución de los SAT resolvedores (solvers):** Desde DP/DPLL hasta CDCL.
- **Heurísticas en CDCL:** VSIDS vs. DLIS, ventajas y limitaciones.
- **Metodología experimental:** Diseño de pruebas, métricas y casos de estudio.
- **Resultados y conclusiones:** Análisis comparativo y recomendaciones prácticas.

Esta investigación busca no solo esclarecer el debate entre VSIDS y DLIS, sino también sentar bases para el diseño de heurísticas adaptativas, impulsando la próxima generación de resolvedores.

Capítulo 1

Marco Teórico

1.1. Fundamentos de los Problemas de Satisfacción de Restricciones y SAT

Los Problemas de Satisfacción de Restricciones (CSP, por sus siglas en inglés) constituyen un paradigma esencial para la modelación de problemas combinatorios en inteligencia artificial, investigación operativa y ciencias de la computación. Formalmente, un CSP se define como una tripleta (V, D, C), donde V representa un conjunto de variables, D sus dominios discretos finitos, y C un conjunto de restricciones que determinan las combinaciones válidas de valores **39**. Por ejemplo, en un problema de asignación de horarios, V corresponde a los cursos, D a los horarios disponibles, y C a las reglas que impiden superposiciones. La solución del problema consiste en una asignación de valores a las variables que satisface todas las restricciones, y en algunos casos, adicionalmente, optimiza ciertos criterios como la utilización de recursos **40**.

1.1.1. SAT como Caso Especial de CSP y su NP-Completitud

El Problema de Satisfacibilidad Booleana (SAT, por sus siglas en inglés) se considera un caso particular de CSP, en el cual los dominios de las variables son binarios $(0, 1)$ y las restricciones se expresan mediante fórmulas en Forma Normal Conjuntiva (FNC) **43**. Una fórmula en FNC se compone de una conjunción de cláusulas, donde cada cláusula es una disyunción de literales, es decir, variables o sus negaciones **6**. Resolver un problema SAT implica determinar si existe una asignación de valores que satisfaga simultáneamente todas las cláusulas, lo cual equivale a resolver un CSP binario con restricciones específicas.

La importancia teórica de SAT se fundamenta en su clasificación como problema NP-completo, establecida por Cook y Levin en 1971 **2,24**. Esta clasificación conlleva dos implicaciones fundamentales: en primer lugar, cualquier problema perteneciente a la clase NP puede reducirse a una instancia de SAT en tiempo polinomial **26**; en segundo lugar, la existencia de un algoritmo de tiempo polinomial que resuelva SAT implicaría que $P = NP$, lo cual provocaría un colapso en la jerarquía de complejidad computacional **2**. Si bien en la práctica los solucionadores (solvers) actuales logran resolver instancias que contienen millones de variables **3**, en el peor de los casos SAT presenta una complejidad exponencial intrínseca **27**.

1.1.2. Ineficiencias Fundamentales de SAT

El enfoque de fuerza bruta para resolver SAT —consistente en evaluar las 2^n asignaciones posibles mediante tablas de verdad— evidencia su carácter intratable en el peor de los casos **19**. Esta explosión combinatoria se intensifica en fórmulas que carecen de estructura discernible, en las cuales técnicas como la propagación unitaria o el aprendizaje de cláusulas presentan un impacto limitado **30**. En particular, las instancias aleatorias del problema 3-SAT próximas al umbral de fase (aproximadamente 4.26 cláusulas por variable **30**) provocan que algoritmos clásicos como DPLL experimenten un crecimiento exponencial en el tiempo de ejecución **27**.

No obstante, SAT se consolida como una herramienta práctica debido a dos factores principales: (1) la posibilidad de codificar CSPs genéricos en FNC mediante técnicas como codificaciones directas o el método de Tseitin **44**; y (2) la existencia de solucionadores modernos basados en CDCL, los cuales aprovechan regularidades empíricas observadas en instancias industriales **3**. Esta dualidad entre complejidad teórica y eficiencia práctica posiciona a SAT como un componente central en aplicaciones como verificación de hardware, planificación autónoma y criptoanálisis **39**.

1.1.3. Relación Práctica entre CSPs y SAT

Si bien los CSPs permiten modelar problemas con dominios arbitrarios y restricciones globales —lo cual representa una ventaja frente a la rigidez booleana de SAT—, su resolución directa mediante técnicas como *backtracking* o la aplicación de consistencia de arco presenta limitaciones similares en cuanto a escalabilidad **46**. Como respuesta a estas limitaciones, se recurre frecuentemente a la traducción de CSPs a SAT, con el objetivo de aprovechar las décadas de avances en la optimización de solucionadores CDCL **44**. Estudios empíricos muestran que ciertas codificaciones eficientes —por ejemplo, el *order encoding* para restricciones de orden— pueden reducir hasta en un 60 % el tiempo de resolución en comparación con enfoques nativos basados en CSPs **45**.

Sin embargo, dicha traducción conlleva compromisos. Mientras SAT se adapta mejor a restricciones locales y cláusulas pequeñas, los CSPs ofrecen mecanismos más adecuados para el tratamiento de restricciones globales —como la restricción *alldif-*

ferent— mediante propagadores especializados **46**. En contextos como la asignación de turnos hospitalarios, los modelos CSP con restricciones de recursos logran resolver instancias en minutos, mientras que sus equivalentes codificados en SAT pueden requerir varias horas debido a la proliferación de cláusulas **46**. Esta dicotomía resalta la necesidad de seleccionar el paradigma de resolución en función de la estructura del problema.

Cada enfoque posee un nicho de aplicabilidad particular. Por ejemplo, la Programación con Restricciones (CP) resulta más efectiva en tareas de planificación con restricciones complejas; la Programación Lineal Entera Mixta (MILP) es preferida en problemas de optimización logística con restricciones lineales; y SAT destaca en verificación formal, donde la traducción a FNC se realiza de manera natural **39,40**. La elección del paradigma adecuado depende fundamentalmente de la capacidad para explotar la estructura subyacente del problema, lo cual explica el éxito práctico de SAT a pesar de su complejidad teórica **3**.

Esta sección establece las bases para el análisis detallado de la evolución de los *solvers* SAT (Sección 1.2), donde se examina cómo técnicas como CDCL superan las ineficiencias teóricas mediante innovaciones algorítmicas pragmáticas.

1.2. Evolución de los SAT *solvers*: Principio de Resolución, DP, DPLL y CDCL

La evolución de los solucionadores SAT constituye un hito en la ciencia computacional, pues convierte un problema teóricamente intratable en una herramienta práctica de amplio uso industrial. Este desarrollo se apoya en tres pilares algorítmicos: el Principio de Resolución, los métodos Davis–Putnam (DP) y Davis–Putnam–Logemann–Loveland (DPLL), y la revolución moderna impulsada por los solucionadores de aprendizaje de cláusulas dirigido por conflictos (CDCL). Este recorrido refleja no solo avances técnicos, sino también una comprensión profunda de cómo combinar teoría y pragmatismo para superar las barreras de complejidad inherentes al problema de la satisfacibilidad booleana (SAT)[1].

1.2.1. Principio de Resolución: Fundamento Teórico

El Principio de Resolución (PR), propuesto originalmente en el contexto de la lógica proposicional, constituye la base teórica de numerosos algoritmos SAT. Este principio permite derivar nuevas cláusulas a partir de pares de cláusulas que contienen literales complementarios, reduciendo progresivamente la fórmula hasta detectar una

[1] Problema NP-completo que determina si existe una asignación de valores verdaderos y falsos que satisfaga una fórmula lógica dada.

contradicción o verificar su satisfacibilidad. Aunque es completo para fórmulas en Forma Normal Conjuntiva (FNC), su aplicación directa resulta impráctica debido al crecimiento exponencial en el número de cláusulas generadas **12**. Sin embargo, su relevancia conceptual fundamenta y guía el diseño de métodos más eficientes.

1.2.2. Algoritmo Davis–Putnam (DP): Primer Intento Práctico

El algoritmo Davis–Putnam (DP) propone una aplicación sistemática del Principio de Resolución en un entorno computacional **5**. Su estrategia se basa en dos operaciones principales: (1) la eliminación de literales puros, que asigna valores a variables que aparecen con un único signo en todas las cláusulas; y (2) la resolución dirigida, que elimina variables seleccionadas mediante la generación de cláusulas resolventes.

Aunque DP evita la exploración exhaustiva de las 2^n asignaciones, su dependencia de la resolución genera un crecimiento exponencial en el número de cláusulas intermedias, lo que limita su eficacia a instancias de tamaño reducido **12**.

1.2.3. Algoritmo DPLL: Búsqueda Inteligente con Retroceso

El algoritmo DPLL refina DP sustituyendo la resolución explícita por una búsqueda con retroceso en el espacio de asignaciones **12**. Incorpora tres mecanismos fundamentales:

- **Propagación unitaria**: asigna valores a literales en cláusulas unitarias, simplificando la fórmula antes de cada decisión.
- **Eliminación de literales puros**: mantiene la optimización heredada de DP para reducir el tamaño de la fórmula.
- **Bifurcación con retroceso**: selecciona heurísticamente valores para variables no asignadas y retrocede al detectar conflictos.

Este método explora en profundidad el árbol de decisiones y reduce el uso de memoria respecto a DP. Sin embargo, su eficiencia disminuye en instancias industriales debido a la repetición de exploraciones conflictivas y a la falta de mecanismos que aprovechen información de conflictos previos **8**.

1.2.4. CDCL: Solucionadores de Aprendizaje de Cláusulas Dirigido por Conflictos

El enfoque CDCL introduce dos innovaciones clave sobre DPLL: el aprendizaje de cláusulas y el retroceso no cronológico **12**.

El aprendizaje de cláusulas analiza los conflictos para generar cláusulas que evitan la repetición de subárboles conflictivos. El retroceso no cronológico permite regresar a niveles de decisión relevantes sin deshacer todos los pasos intermedios.

Estos mecanismos reducen la exploración redundante de subespacios, mejoran la adaptabilidad heurística durante la búsqueda y superan la ineficiencia del retroceso cronológico de DPLL **6,17**.

1.2.5. Algoritmo Davis–Putnam (DP): Primer Intento Práctico

El algoritmo Davis–Putnam (DP) propone una aplicación sistemática del Principio de Resolución en un entorno computacional **5**. Su estrategia se basa en dos operaciones principales: (1) la eliminación de literales puros, que asigna valores a variables que aparecen con un único signo en todas las cláusulas; y (2) la resolución dirigida, que elimina variables seleccionadas mediante la generación de cláusulas resolventes.

Aunque DP evita la exploración exhaustiva de las 2^n asignaciones, su dependencia de la resolución genera un crecimiento exponencial en el número de cláusulas intermedias, lo que limita su eficacia a instancias de tamaño reducido **12**.

1.2.6. Algoritmo DPLL: Búsqueda Inteligente con Retroceso

El algoritmo DPLL refina DP sustituyendo la resolución explícita por una búsqueda con retroceso en el espacio de asignaciones **12**. Incorpora tres mecanismos fundamentales:

- **Propagación unitaria**: asigna valores a literales en cláusulas unitarias, simplificando la fórmula antes de cada decisión.
- **Eliminación de literales puros**: mantiene la optimización heredada de DP para reducir el tamaño de la fórmula.
- **Bifurcación con retroceso**: selecciona heurísticamente valores para variables no asignadas y retrocede al detectar conflictos.

Este método explora en profundidad el árbol de decisiones y reduce el uso de memoria respecto a DP. Sin embargo, su eficiencia disminuye en instancias industriales debido a la repetición de exploraciones conflictivas y a la falta de mecanismos que aprovechen información de conflictos previos **8**.

1.2.7. CDCL: Solucionadores de Aprendizaje de Cláusulas Dirigido por Conflictos

El enfoque CDCL introduce dos innovaciones clave sobre DPLL: el aprendizaje de cláusulas y el retroceso no cronológico **12**.

El aprendizaje de cláusulas analiza los conflictos para generar cláusulas que evitan la repetición de subárboles conflictivos. El retroceso no cronológico permite regresar a niveles de decisión relevantes sin deshacer todos los pasos intermedios.

Estos mecanismos reducen la exploración redundante de subespacios, mejoran la adaptabilidad heurística durante la búsqueda y superan la ineficiencia del retroceso cronológico de DPLL **6,17**.

Otras Contribuciones Relevantes

El ecosistema de optimizaciones para CDCL incorpora varias líneas de mejora. En primer lugar, la gestión de cláusulas aprendidas emplea métricas como la LBD para identificar y eliminar cláusulas redundantes, al tiempo que conserva aquellas con alto valor predictivo (denominadas *glue clauses*) **14–17**. A continuación, la combinación con búsqueda local, mediante técnicas de *re-phasing* basadas en frecuencias de conflicto, incrementa el rendimiento en instancias satisfacibles en hasta un 40

1.2.8. CDCL como Motor para Problemas con Restricciones

La aplicación de CDCL a problemas de satisfacción de restricciones (CSP) se apoya en su arquitectura adaptable. Por un lado, la heurística VSIDS detecta patrones emergentes en la codificación de restricciones, favoreciendo variables con alta incidencia en conflictos **15,21**. Por otro lado, las políticas de reinicio basadas en LBD permiten gestionar la modularidad de los subproblemas generados por la traducción a FNC. Asimismo, técnicas como CFUP priorizan cláusulas centrales que reflejan jerarquías implícitas en la red de restricciones **96**.

No obstante, la traducción de restricciones globales a cláusulas booleanas puede provocar un aumento significativo en el número de cláusulas, lo que penaliza el rendimiento en problemas con propagadores especializados de alto nivel **46**.

1.2.9. Conclusión: De la Teoría a la Aplicación Práctica

La evolución desde el Principio de Resolución y DP hasta los solucionadores CDCL demuestra que la integración de mecanismos de inferencia e información de conflictos permite abordar problemas NP-completos con eficiencia razonable en la práctica. DPLL sienta las bases de la búsqueda con retroceso, mientras que CDCL introduce el aprendizaje reflexivo y el retroceso no cronológico para reducir exploraciones redundantes **5,7,11**. Estas innovaciones, junto con mejoras en heurísticas y estructuras de datos, establecen a los solucionadores SAT como herramientas esenciales en verificación, planificación y análisis automático de restricciones.

1.3. Solucionadores actuales

1.3.1. CaDiCaL

CaDiCaL es un solver SAT moderno implementado en C++ que combina claridad de diseño y eficiencia práctica. La arquitectura se organiza en un módulo interno, responsable de la búsqueda CDCL y de técnicas de simplificación de fórmulas —como eliminación acotada de variables (BVE), vivificación e instanciación—; y en un módulo externo que actúa como fachada, gestiona la API y, en modo incremental, revierte pasos de *inprocessing* para mantener limpia la pila de reconstrucción.

El solver se ofrece tanto como biblioteca en C++ y C, como aplicación independiente, y permite la integración de *propagadores de usuario* para sustituir motores basados en MiniSat. También incluye mecanismos de inversión de eliminación de cláusulas y soporte para SAT incremental.

Las decisiones de ramificación se guían por heurísticas de actividad basadas en VSIDS, alternando entre fases estables y enfocadas para mejorar el rendimiento en instancias satisfacibles. Además, se incorpora la noción de *Literal Stability*, que prioriza aquellos literales que permanecen satisfechos durante periodos prolongados.

Los reinicios dinámicos permiten al solver escapar de subespacios improductivos sin perder las cláusulas aprendidas. Para ello, CaDiCaL emplea políticas adaptativas que utilizan contadores de eventos —por ejemplo, accesos a memoria— en lugar de mediciones temporales directas, lo que garantiza determinismo entre ejecuciones. Estas políticas alternan modos estable y enfocado, y bloquean reinicios cuando la búsqueda se encuentra próxima a una solución potencial, siguiendo estrategias propuestas en la literatura.

Aunque estas técnicas mejoran notablemente el rendimiento, la complejidad de las políticas adaptativas y la sensibilidad de sus parámetros pueden introducir sobrecarga computacional y requerir ajustes cuidadosos para evitar reinicios excesivos o insuficientes.

Las fuentes no especifican directamente que DLIS (Dynamic Largest Individual Sum) sea una heurística de decisión implementada por defecto en CaDiCaL. Sin embargo, se indica que CaDiCaL y su reimplementación Kissat emplean la heurística VSIDS (Variable State Independent Decaying Sum). En Kissat, el solver alterna entre modos estable y enfocado; VSIDS se aplica en las fases estables, dirigidas principalmente a instancias satisfacibles.

Aunque no se documenta la inclusión de DLIS en CaDiCaL, puede inferirse que este solver moderno se basa en esquemas de actividad de variables como VSIDS, dada su prevalencia y extensa bibliografía.

Además, la selección de literales en CaDiCaL puede incorporar el concepto de Literal Stability, que prioriza aquellos literales que permanecen satisfechos durante intervalos prolongados al determinar los watched literals.

1.3.2. Políticas de reinicio

Los reinicios constituyen un mecanismo esencial en los solvers CDCL, pues facilitan la salida de regiones improductivas sin descartar las cláusulas aprendidas.

CaDiCaL implementa políticas de reinicio dinámicas que se ajustan al comportamiento interno del solver. Por ejemplo, Kissat alterna entre modos estable y enfocado, determinando la duración de cada fase mediante el recuento de accesos a memoria en lugar de medidas temporales, lo que garantiza determinismo entre ejecuciones. En el modo orientado a instancias satisfacibles, Kissat utiliza permanentemente la estrategia de target phasing.

Las políticas de rephasing de CaDiCaL y Kissat se inspiran en Glucose y alternan entre distintos modos de fase ("B"para best phasing, "W"para walk, .O"para original, Ï"para inverted y "#"para random).

Todas las restart policies pueden ajustarse mediante múltiples parámetros para adaptarse a las características de cada instancia.

1.3.3. Modularidad y Extensibilidad del Código de CaDiCaL

La modularidad y la extensibilidad constituyen objetivos esenciales en el diseño de CaDiCaL **Cadical2.0**. Este solver se ha convertido en plantilla para el "hack track"de la competencia SAT desde 2021, evidenciando su facilidad de adaptación. Los mecanismos principales para modificar su comportamiento incluyen:

- **API rica**: proporciona una interfaz en C++ (y limitada en C) que permite extender funcionalidades y personalizar la interacción con el solver.
- **Propagadores de usuario (ExternalPropagator)**: habilita la implementación de propagadores externos capaces de importar y exportar cláusulas aprendidas o sugerir decisiones al solver, otorgando control directo sobre la búsqueda.
- **Estructura del código fuente**: el código, organizado de forma clara y modular, facilita su uso como modelo para portar e integrar técnicas de última generación en otros solvers.

Diversas investigaciones han extendido CaDiCaL para incorporar nuevas características, algunas de las cuales se han integrado en la versión oficial.

1.3.4. Ventajas de CaDiCaL para Experimentación

CaDiCaL resulta adecuado para estudios académicos sobre heurísticas y políticas de reinicio en SAT solving por las siguientes razones:

- **Diseño limpio y modular**: la arquitectura está orientada a la comprensibilidad y facilidad de modificación, lo que simplifica la implementación de nuevas estrategias sin la complejidad de otros solvers avanzados.
- **Flexibilidad en heurísticas y reinicios**: aunque dispone de configuraciones predeterminadas, permite alternar entre modos estable y enfocado, y ajustar esquemas de rephasing para probar variaciones en políticas de reinicio **BetterDecisionHeuristic**
- **Rendimiento competitivo**: mantiene un desempeño de última generación, garantizando que los resultados experimentales sean representativos del estado del arte.
- **Adopción en la comunidad**: su uso extendido en investigación genera un entorno colaborativo y recursos para investigadores.
- **Documentación exhaustiva**: el código fuente cuenta con comentarios detallados, facilitando la comprensión y manipulación del solver por parte de nuevos usuarios.

1.3.5. Comparación con Otros Solvers

MiniSAT representa un solver CDCL clásico que implementa propagación unitaria, heurística de actividad y retrocesos cronológicos. No obstante, carece de optimizaciones avanzadas, como gestión adaptativa de cláusulas aprendidas, políticas de reinicio dinámicas y una API extensible, lo cual limita su desempeño frente a solvers más recientes

En contraste, Glucose incorpora la métrica LBD (Literal Block Distance) para evaluar la relevancia de las cláusulas aprendidas y optimizar la eliminación de las de menor impacto. Asimismo, emplea políticas de reinicio basadas en LBD que favorecen la resolución en escenarios incrementales. Sin embargo, su conjunto de características y el grado de modularidad de su código son inferiores al de CaDiCaL, lo que dificulta su adaptación en contextos de investigación

Por último, Kissat constituye una reimplementación optimizada de CaDiCaL en lenguaje C que combina heurísticas de actividad y reinicios adaptativos para maximizar la velocidad y reducir el uso de memoria. En consecuencia, lidera competiciones SAT recientes gracias a su equilibrio entre rendimiento y eficiencia de recursos. No obstante, su API es menos completa y carece de soporte integral para técnicas incrementales, lo cual restringe su versatilidad como plataforma de experimentación. En comparación, CaDiCaL mantiene una arquitectura modular que facilita la integración de nuevas técnicas y su utilización como referente en el desarrollo de solvers

1.4. Parámetros de Evaluación para Solvers CDCL: Taxonomía de extitBenchmarks

La evaluación sistemática de solvers CDCL exige una taxonomía clara de instancias SAT que permita comparar heurísticas de actividad, estrategias de reinicio y métodos de propagación unitaria. Para este propósito, las instancias se clasifican según su origen y propiedades estructurales, de modo que cada categoría revele puntos fuertes y limitaciones algorítmicas extbf15,17.

En primer lugar, las extitinstancias de aplicación proceden de problemas industriales, como la verificación de circuitos o la planificación logística. Estas fórmulas presentan estructuras modulares y extitbackdoors pequeños —subconjuntos de variables cuya asignación simplifica significativamente la búsqueda—, lo que permite a heurísticas dinámicas como VSIDS explotar patrones locales mediante el registro de conflictos recientes extbf20,21,23.

A continuación, las extitinstancias combinatorias dificultosas, diseñadas para desafiar la generalidad de los solvers (por ejemplo, codificaciones del principio del palomar), carecen de la estructura implícita de los casos reales y exhiben simetrías y dependencias globales que ponen a prueba la adaptabilidad de las heurísticas extbf16,19.

Además, las extitinstancias aleatorias, generadas según modelos Random k-SAT, muestran una distribución uniforme de cláusulas sin sesgos estructurales. En estas condiciones, técnicas como el aprendizaje de cláusulas ofrecen beneficios limitados, lo que resulta clave para evaluar el rendimiento en entornos carentes de regularidades extbf17,21.

Por último, las extitinstancias ágiles surgen de la transformación de problemas de lógica de orden superior a SAT mediante extitbit-blasting. Estas fórmulas presentan alta densidad de cláusulas y literales auxiliares, lo que prueba la capacidad del solver para manejar estructuras complejas y voluminosas extbf19.

1.4.1. Clasificación por Satisfacibilidad y Propiedades Estructurales

La distinción entre instancias SAT e UNSAT determina la orientación de las estrategias de resolución. En las primeras, las heurísticas que exploran asignaciones prometedoras —como VSIDS con preservación de fase *(phase saving)*— potencian la localización rápida de soluciones. Por su parte, las instancias UNSAT requieren la generación de cláusulas aprendidas de alto impacto, con el fin de acotar el espacio de búsqueda y acelerar la refutación **41**.

Del mismo modo, las propiedades estructurales aportan criterios de evaluación adicionales. La modularidad de la fórmula y la existencia de *backbones* —variables que mantienen el mismo valor en todas las soluciones— facilitan la identificación de subespacios críticos. De igual modo, un *treewidth* reducido correlaciona con una mayor eficiencia de la propagación unitaria, lo que contribuye a disminuir los tiempos de resolución **20,22**.

1.4.2. Densidad y Transición de Fase

La densidad de una instancia, definida como el cociente entre cláusulas y variables, condiciona su nivel de dificultad. En los modelos aleatorios de 3-SAT, la complejidad alcanza su punto máximo alrededor de 4.27 cláusulas por variable, conocido como umbral de transición de fase **25**. En esta zona, heurísticas estáticas como DLIS pierden eficacia ante la ausencia de patrones explotables **17**. En cambio, la combinación de VSIDS con políticas de reinicio basadas en LBD equilibra la exploración global y la explotación local, permitiendo al solver superar con eficiencia estas regiones críticas **21**.

1.4.3. Clasificación por Satisfacibilidad y Propiedades Estructurales

La distinción entre instancias SAT e UNSAT determina la orientación de las estrategias de resolución. En las primeras, las heurísticas que exploran asignaciones prometedoras —como VSIDS con preservación de fase *(phase saving)*— potencian la localización rápida de soluciones. Por su parte, las instancias UNSAT requieren la generación de cláusulas aprendidas de alto impacto, con el fin de acotar el espacio de búsqueda y acelerar la refutación **41**.

Del mismo modo, las propiedades estructurales aportan criterios de evaluación adicionales. La modularidad de la fórmula y la existencia de *backbones* —variables que mantienen el mismo valor en todas las soluciones— facilitan la identificación de subespacios críticos. Asimismo, un *treewidth* reducido correlaciona con una mayor eficiencia de la propagación unitaria, lo que contribuye a disminuir los tiempos de resolución **20,22**.

En instancias estructuradas, sin embargo, densidades elevadas no siempre implican mayor dificultad. Codificaciones compactas de problemas como el coloreado de grafos se resuelven con eficiencia cuando los solvers identifican *backdoors* mediante heurísticas sensibles al contexto **23**.

1.4.4. Densidad y Transición de Fase

La densidad de una instancia, definida como el cociente entre cláusulas y variables, condiciona su nivel de dificultad. En los modelos aleatorios de 3-SAT, la complejidad alcanza su punto máximo alrededor de 4.27 cláusulas por variable, conocido como umbral de transición de fase **25**. En esta zona, heurísticas estáticas como DLIS pierden eficacia ante la ausencia de patrones explotables **17**. En cambio, la combinación de VSIDS con políticas de reinicio basadas en LBD equilibra la exploración global y la explotación local, permitiendo al solver superar con eficiencia estas regiones críticas **21**.

1.4.5. Relevancia para la Evaluación de Heurísticas

La selección de benchmarks resulta crucial al contrastar técnicas como VSIDS y DLIS. Mientras DLIS, basada en conteos estáticos de aparición de literales, ofrece un rendimiento óptimo en instancias aleatorias alejadas del umbral de transición de fase **18**, VSIDS prevalece en aplicaciones reales gracias a su adaptación dinámica a patrones de conflicto **30**. Asimismo, en instancias modulares, las estrategias de reinicio basadas en bandits (MAB) demuestran ventajas al preservar cláusulas *glue*, acelerando la exploración del espacio de búsqueda **37,21**.

Propiedades estructurales como la presencia de *backbones* o un *treewidth* reducido permiten desglosar el impacto de técnicas específicas: por ejemplo, la selección de cláusulas unitarias mediante CFUP reduce hasta un 40 % el tiempo de resolución en instancias con alta proporción de cláusulas centrales ($LBD \leq 7$), mientras su efecto es marginal en fórmulas sin comunidad definida **96,21**.

Este marco taxonómico no solo sistematiza la evaluación de solvers, sino que orienta el diseño de heurísticas híbridas. Por ejemplo, la integración de VSIDS con métricas de centralidad de grafos **22** mejora la cobertura en instancias combinatorias, mientras el acoplamiento con búsqueda local **1** beneficia a instancias ágiles. De este modo, la elección estratégica de benchmarks establece un puente entre la complejidad teórica de SAT y su aplicabilidad práctica.

¿Qué son las 'assumptions'? • En esencia, las 'assumptions' permiten invocar un SAT solver con un conjunto de valores ya asignados a ciertas variables. Esto es particularmente útil en escenarios donde el solver se utiliza iterativamente. • Por ejemplo, si se necesita agregar o eliminar una cláusula ci entre llamadas al solver, se puede añadir a la fórmula como (ci $\vee$ si), donde si es una variable de activación, selección o indicador. Al establecer si = 1, la cláusula ci se "considera" en la fórmula, y al establecer si = 0, no se considera. • Esta técnica fue propuesta inicialmente en el solver MiniSat.

Además de su rol en la resolución incremental, las variables de activación también se emplean para cláusulas temporales. Por ejemplo, para añadir una cláusula temporal c, se introduce una variable de activación a y se añade (c ∨ a) a la fórmula, asumiendo ¬a. Para .^el^iminarç, simplemente se añade la cláusula unitaria (a). • Un aumento en el número de variables de activación puede impactar negativamente el rendimiento del solver, lo que tradicionalmente se ha abordado mediante reinicios periódicos del solver. Sin embargo, algunos enfoques, como GipSAT, permiten la reutilización de una única variable de activación al eliminar proactivamente las cláusulas aprendidas que la contienen después de cada proceso de resolución. CaDiCaL, en sus versiones recientes, utiliza 'constraints' que pueden simularse con 'activation literals' en versiones anteriores

¿Por qué se comprueban antes de la heurística de decisión de variables? Las 'assumptions' se asignan y se propagan antes de que comience la búsqueda efectiva mediante la heurística de decisión de variables. Esto se debe a la forma en que los solvers CDCL (Conflict-Driven Clause Learning) operan:

Propagación Unitaria (BCP) Inicial: El algoritmo CDCL inicia con una fase de Propagación Unitaria (BCP, por sus siglas en inglés, Boolean Constraint Propagation). En esta fase, se asignan valores a las variables que son forzadas por las cláusulas unitarias existentes o por las 'assumptions'. • Detección de Conflictos Temprana: Si la Propagación Unitaria (BCP) detecta un conflicto al nivel de decisión 0 (es decir, antes de que el solver haya tomado cualquier decisión propia), significa que la fórmula (junto con las 'assumptions' actuales) es insatisfacible.

- Eficiencia: Esta comprobación temprana es crucial para la eficiencia. Permite que el solver determine la insatisfacibilidad rápidamente, sin necesidad de explorar el espacio de búsqueda haciendo decisiones de ramificación que, en última instancia, llevarían a un conflicto inevitable debido a los supuestos. Solo si BCP no detecta conflictos y no todas las variables están asignadas, el solver procede a seleccionar una variable para ramificar utilizando la heurística de decisión.

En resumen, las 'assumptions' son como un punto de partida para el solver. Se procesan inicialmente a través de la propagación unitaria para asegurarse de que la configuración inicial (incluyendo estos supuestos) no genere un conflicto inmediato. Esto optimiza el proceso, ya que evita búsquedas infructuosas en partes del espacio que ya se sabe que son inconsistentes con los supuestos dados.

Capítulo 2

Algoritmos

2.1. DP/DPLL

2.1.1. Principio de Resolución (PR)

El Principio de Resolución (PR) constituye una técnica fundamental para abordar el problema SAT. Dada una fórmula en Forma Normal Conjuntiva (FNC), cada cláusula se representa como un conjunto de literales, evitando repeticiones tanto dentro de una cláusula como en la fórmula completa. A partir de esta representación, PR identifica pares de cláusulas que contienen literales complementarios y genera una nueva cláusula al unir ambas y eliminar el literal y su negación.

Concretamente, si **B** y **C** son cláusulas de la FNC **A** tales que $l \in \mathbf{B}$ y $\neg l \in \mathbf{C}$, entonces la cláusula resultante se define como:

$$\mathbf{D} = (\mathbf{B} \setminus \{l\}) \cup (\mathbf{C} \setminus \{\neg l\})$$

En este proceso, **B** y **C** actúan como cláusulas padres o premisas, mientras que **D** corresponde al solvente o conclusión.

Por ejemplo, la resolución de las cláusulas

$$\{\neg p, \neg q, \neg r\} \quad y \quad \{\neg p, q, \neg r\}$$

produce

$$\{\neg p, \neg r\}$$

Asimismo, la combinación de

$$\{\neg q\} \quad y \quad \{q\}$$

conduce a la cláusula vacía, lo que evidencia la insatisfacibilidad de la fórmula.

Resolución Unitaria (RU)

Una instancia particular del Principio de Resolución (PR) es la Resolución Unitaria (RU), en la cual una de las premisas es una cláusula unitaria[1]. Este caso especial adquiere relevancia por su simplicidad y eficiencia, ya que permite deducciones inmediatas a partir de asignaciones forzadas.

El proceso consiste en identificar una cláusula unitaria y aplicarla sobre otra cláusula que contenga el literal complementario, eliminándolo y generando una nueva cláusula más restringida. Por ejemplo:

$$\frac{\{\neg q, p, \neg r\}, \{r\}}{\{\neg q, p\}}$$

En este caso, la cláusula unitaria $\{r\}$ permite simplificar la cláusula $\{\neg q, p, \neg r\}$, eliminando el literal $\neg r$ y generando una nueva cláusula $\{\neg q, p\}$. Esta operación puede aplicarse iterativamente, facilitando la propagación de valores lógicos en la fórmula original, y constituye un componente fundamental en algoritmos como DPLL y CDCL, donde se utiliza para propagar restricciones a lo largo del proceso de asignación.

2.1.2. Davis-Putnam (DP)

Uno de los primeros algoritmos propuestos para la resolución del problema SAT fue el de Davis-Putnam (DP), cuyo funcionamiento se basa en gran medida en el Principio de Resolución (PR). Este algoritmo implementa tres procedimientos fundamentales: la Propagación Unitaria (PU), la Eliminación de Literales Puros (ELP) y la Resolución Basada en División (RD).

La Propagación Unitaria (PU) identifica cláusulas unitarias dentro de la fórmula en forma normal conjuntiva (FNC) y procede a asignar forzosamente el valor correspondiente al literal involucrado. A continuación, elimina de la fórmula todas las cláusulas satisfechas por dicha asignación, y también suprime el literal complementario en aquellas cláusulas donde aparezca. Por su parte, la Eliminación de Literales Puros (ELP) detecta literales cuya polaridad es única en toda la fórmula (es decir, cuyo complemento no se encuentra presente) y elimina las cláusulas en las que aparezcan, pues estas pueden satisfacerse directamente sin afectar la satisfacibilidad de la fórmula. Tanto PU como ELP se consideran técnicas de preprocesamiento, destinadas a simplificar la FNC antes de aplicar los pasos recursivos del algoritmo.

Una vez agotadas estas simplificaciones, DP procede con la Resolución Basada en División (RD), que consiste en seleccionar una variable, asignarle un valor (0 o 1) y

[1]Una cláusula que contiene un único literal, y por tanto, fuerza su valor a ser verdadero bajo una interpretación determinada.

continuar la resolución de forma recursiva sobre la fórmula resultante. Esta estrategia permite explorar sistemáticamente el espacio de soluciones posibles hasta determinar si la fórmula es satisfacible o no.

Sean las siguientes fórmulas de la Lógica Proposicional:

$$r, \quad [q \wedge r] \implies p, \quad [q \vee r] \implies \neg p, \quad [\neg q \wedge r] \implies \neg p, \quad \neg s \implies p$$

A partir de la conjunción de estas proposiciones, se obtiene la siguiente forma normal conjuntiva (FNC):

$$\{\{r\}, \{p, \neg q, \neg r\}, \{\neg p, \neg q\}, \{\neg p, \neg r\}, \{\neg p, q, \neg r\}, \{p, s\}\}$$

Aplicando **Propagación Unitaria (PU)** sobre la cláusula unitaria $\{r\}$, se eliminan todas las cláusulas que contienen r y se suprime $\neg r$ de las restantes:

$$\{\{p, \neg q\}, \{\neg p, \neg q\}, \{\neg p\}, \{\neg p, q\}, \{p, s\}\}$$

Posteriormente, al aplicar **PU** sobre la cláusula unitaria $\{\neg p\}$, se elimina toda cláusula que contenga $\neg p$ y se remueve p de las demás:

$$\{\{\neg q\}, \{s\}\}$$

Aplicando nuevamente **PU** sobre $\{\neg q\}$:

$$\{\{s\}\}$$

Y finalmente, aplicando **PU** sobre $\{s\}$:

$$\{\}$$

Dado que la fórmula ha sido completamente reducida sin generar contradicciones, se concluye que la instancia es satisfacible.

Cabe señalar que el algoritmo Davis-Putnam requiere memoria exponencial en el peor de los casos, ya que explora todas las asignaciones posibles para las variables. Esto se traduce en un árbol de decisión cuyo tamaño crece exponencialmente con el número de variables involucradas.

2.1.3. Davis-Putnam-Logemann-Loveland (DPLL)

El algoritmo Davis-Putnam-Logemann-Loveland (DPLL) constituye una mejora significativa del método Davis-Putnam (DP), al preservar sus fundamentos teóricos y superar una de sus principales limitaciones: el consumo exponencial de memoria. Para ello, DPLL incorpora un mecanismo de retroceso (extitbacktracking) cronológico que

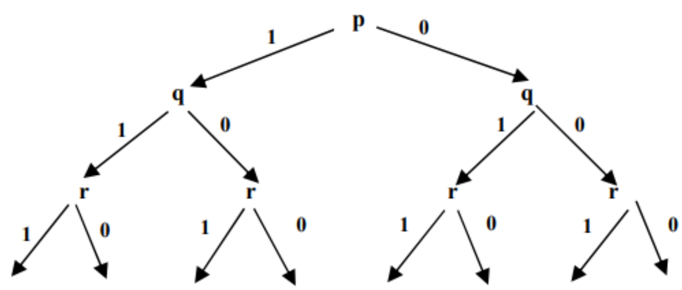

Figura 2.1: Posible espacio de búsqueda de una FNC

le permite deshacer asignaciones al regresar al último nivel de decisión anterior una vez detectada una "cláusula de conflicto"

ootnoteSe denomina cláusula de conflicto a aquella en la que todos sus literales son evaluados como falsos bajo una asignación parcial.. Este enfoque permite explorar el árbol de búsqueda de forma más eficiente, reduciendo la necesidad de almacenar todas las ramas posibles.

DPLL adopta una estrategia de generación "lazy" del árbol de asignaciones: antes de realizar una nueva ramificación, verifica mediante propagación unitaria si existen conflictos que invaliden dicha extensión. Esta verificación garantiza que las asignaciones parciales se mantengan consistentes, y que las soluciones (cuando existen) se ubiquen en las hojas del árbol de decisión. En caso de que el conflicto se produzca en el nivel de decisión cero, y ambas asignaciones posibles para una variable hayan sido consideradas, se concluye que la fórmula es insatisfacible. En conjunto, el procedimiento de DPLL puede resumirse como una combinación de ramificación, propagación unitaria y retroceso sistemático.

Adicionalmente, DPLL incluye una etapa de preprocesamiento sobre la fórmula en Forma Normal Conjuntiva (FNC), en la cual se aplican simplificaciones basadas en leyes de la Lógica Proposicional. Entre estas se encuentra la eliminación de cláusulas redundantes mediante el principio de subsunción[2]. Estas técnicas contribuyen a reducir el tamaño de la instancia antes de la búsqueda propiamente dicha, mejorando la eficiencia del algoritmo sin comprometer su completitud. Obsérvese el siguiente ejemplo:

Sea la FNC:

[2]Sean C y C' dos cláusulas de una FNC; si $C' \subseteq C$, entonces C' se considera subsumida por C y puede eliminarse sin alterar la satisfacibilidad de la fórmula.

$$\{\{\neg p, \neg q\}, \{\neg p, \neg q\}, \{\neg p, q, \neg r\}, \{\neg p, r, s\}, \{p, s\}\}$$

Simplificando mediante la ley de absorción:

$$\{\{\neg p, \neg q\}, \{\neg p, q, \neg r\}, \{\neg p, r, s\}, \{p, s\}\}$$

Eliminando el literal puro s:

$$\{\{\neg p, \neg q\}, \{\neg p, q, \neg r\}\}$$

Ramificando: $p = 1$

$$\{\{\neg q\}, \{q, \neg r\}\}$$

Aplicando **PU** en $\{\neg q\}$:

$$\{\{\neg r\}\}$$

Aplicando **PU** en $\{\neg r\}$:

$$\emptyset$$

Luego, la FNC es satisfacible.

No obstante la reducción del espacio de memoria de DPLL respecto a DP, aún persisten problemas fundamentales: la selección de variables, el *backtrack* cronológico y la elección de cláusulas unitarias.

En primer lugar, la selección de la variable a la que se asignará un valor influye directamente en la "forma" del espacio de búsqueda. Una decisión inapropiada puede derivar en caminos significativamente más largos hacia una solución. Por tanto, resulta crucial emplear heurísticas que optimicen esta elección. (poner ejemplo)

En segundo lugar, el *backtrack* cronológico ante un conflicto obliga a explorar, posiblemente de forma innecesaria, las asignaciones alternativas en niveles anteriores. Esta ineficiencia se acentúa cuando la causa real del conflicto se encuentra a k niveles de distancia del punto donde se detectó. Además, DPLL no capitaliza las cláusulas que originaron los conflictos; es decir, no "aprende" de ellos. En consecuencia, es susceptible de incurrir reiteradamente en los mismos patrones erróneos de asignación.

Finalmente, el problema de la selección de cláusulas unitarias también repercute en la eficiencia del algoritmo, estando íntimamente relacionado con la estrategia de selección de variables.

2.2. CDCL

CDCL es una mejora que se le añadió al algoritmo DPLL con el objetivo de erradicar el problema del retroceso (*backtrack*) cronológico, una vez encontrada una cláusula de conflicto (todos sus literales evalúan 0).

El retroceso cronológico consiste en recorrer el árbol de decisión (estructura propia del algoritmo DPLL que se forma al asignarle valores a las variables) retrocediendo de a 1 por cada nivel, probando todos los valores aún sin explorar de cada variable hasta encontrar la asignación causante del conflicto. Esta búsqueda es ineficiente dado que, además de analizar casos innecesarios, se vuelve susceptible a cometer el mismo error en el futuro (potencialmente realiza la misma combinación de asignaciones) generando búsquedas redundantes.

Para solucionar este problema, CDCL crea un grafo dirigido y acíclico que permite guardar el historial de asignaciones de cada variable. En dicho grafo, los nodos son las variables y los arcos constituyen la causa de la asignación de dicha variable: la cláusula a la que pertenece, si fue asignada por propagación unitaria, y `null` para variables asignadas por decisión. El grafo también contiene dos metadatos: el valor asignado a cada variable (0 o 1) y el nivel de decisión en el que se asignó (los diferentes niveles de decisión están marcados por la asignación de valores por decisión). Cabe destacar que la dirección de los arcos en el grafo va desde las variables de decisión hacia aquellas que, en el mismo nivel, tuvieron que forzar su valor por propagación unitaria. En el caso de una nueva variable de decisión, se crea un nuevo arco con valor `null` desde la variable asignada por decisión en el nivel anterior hasta la nueva variable.

Cuando una cláusula resulta ser de conflicto (sus literales evaluaron 0), CDCL crea un nuevo nodo en el grafo que representa dicho conflicto, para comenzar con su análisis. Este análisis busca en el grafo la asignación causante del conflicto, para retroceder justo hacia ese punto y realizar un *backjump* en lugar de un retroceso cronológico, como en DPLL. En caso de que el nivel del *backjump* sea el nivel 0, CDCL considera la FNC como insatisfacible. Asimismo, con este análisis CDCL busca conformar una cláusula (cláusula aprendida) que represente la combinación de asignación de valores que condujo a dicho conflicto, para incluirla en la base de datos de las cláusulas de la FNC y evitar cometer el mismo error en iteraciones futuras. El punto escogido para realizar el *backjump* es conocido como primer punto de implicación único (*First-UIP* por sus siglas en inglés). Este punto será aquel literal que, en la cláusula aprendida, posea el más alto nivel de decisión diferente del actual.

Es necesario enfatizar en el hecho de que la cláusula aprendida debe contener **únicamente** un literal cuyo valor haya sido asignado en el nivel de decisión actual. En caso de haber más de uno, CDCL recorre el grafo en busca de la cláusula que causó la asignación de una de estas variables y aplica el Principio de Resolución entre esta y la cláusula aprendida hasta el momento. La cláusula resultante pasará a ser la nueva cláusula aprendida. El proceso se repetirá hasta que la cláusula aprendida contenga solo un literal cuyo valor fue asignado en el nivel de decisión actual.

A continuación, se muestra como ejemplo la implementación en Python de un CDCL SAT *solver*

```
class SATSolver:
    def __init__(self, formula):
        """
        Initializes the SAT solver.

        Parameters:
          formula: A list of clauses, where each clause is
              represented as a list of integers.
                   A positive integer i represents the
                      variable x_i, and a negative integer
                       -i represents -x_i.
        """
        # Copy the formula so that learned clauses can be
           appended.
        self.formula = formula[:]
        self.assignments = {}    # Maps variable -> True/
           False assignment.
        self.levels = {}         # Maps variable ->
           decision level at which it was assigned.
        self.reasons = {}        # Maps variable -> clause
            that forced the assignment (None for decision
           variables).
        self.decision_level = 0  # Current decision level.
        self.decision_stack = [] # Stack storing tuples (
           variable, assigned value, decision level).

    def literal_value(self, literal):
        """
        Evaluates a literal given the current partial
           assignment.

        Returns:
          True if the literal is assigned True,
          False if the literal is assigned False,
          None if the variable is unassigned.
```

```
        """
        var = abs(literal)
        if var not in self.assignments:
            return None
        # For a positive literal, the assignment is the
           value; for a negative literal, invert the
           assignment.
        return self.assignments[var] if literal > 0 else
           not self.assignments[var]

    def check_clause(self, clause):
        """
        Determines the status of a clause with respect to
           current assignments.

        Returns a tuple (status, literal) where status is
           one of:
          - 'satisfied': Clause is already True under the
             assignment.
          - 'conflict': All literals are assigned False (
             the clause is unsatisfied).
          - 'unit': Exactly one literal is unassigned
             while all others are False (this literal must
              be True).
          - 'undefined': The clause is neither satisfied,
             conflicting, nor unit.
        """
        satisfied = False
        unassigned_count = 0
        unit_literal = None
        for literal in clause:
            val = self.literal_value(literal)
            if val is True:
                return ('satisfied', None)
            if val is None:
                unassigned_count += 1
                unit_literal = literal  # Last seen
                   unassigned literal.
        if unassigned_count == 0:
            return ('conflict', None)
```

```
        if unassigned_count == 1:
            return ('unit', unit_literal)
        return ('undefined', None)

    def unit_propagate(self):
        """
        Repeatedly applies unit propagation.

        Returns:
          A conflicting clause if a conflict is found
              during propagation; otherwise, returns None.
        """
        changed = True
        while changed:
            changed = False
            for clause in self.formula:
                status, unit_literal = self.check_clause(
                    clause)
                if status == 'conflict':
                    # A clause is unsatisfied => conflict!
                    return clause
                elif status == 'unit':
                    var = abs(unit_literal)
                    if var not in self.assignments:
                        # Determine the value needed to
                            satisfy the unit clause.
                        value = (unit_literal > 0)
                        self.assignments[var] = value
                        self.levels[var] = self.
                            decision_level
                        self.reasons[var] = clause  #
                            Store the clause as the reason
                            for this assignment.
                        self.decision_stack.append((var,
                            value, self.decision_level))
                        changed = True
        return None

    def pick_branching_variable(self):
        """
```

```
        Selects the next unassigned variable found in the
           formula.

        (In a production solver, better heuristics like
           VSIDS are used.)
        """
        variables = set()
        for clause in self.formula:
            for literal in clause:
                variables.add(abs(literal))
        for var in variables:
            if var not in self.assignments:
                return var
        return None

    def resolve(self, clause1, clause2, pivot):
        """
        Performs the resolution on two clauses over the
           pivot literal.

        Specifically, it returns:
          (clause1 \ {pivot}) U (clause2 \ {-pivot})
        """
        new_clause = []
        for lit in clause1:
            if lit == pivot:
                continue
            if lit not in new_clause:
                new_clause.append(lit)
        for lit in clause2:
            if lit == -pivot:
                continue
            if lit not in new_clause:
                new_clause.append(lit)
        return new_clause

    def conflict_analysis(self, conflict_clause):
        """
        Conducts conflict analysis to find the First UIP.
        """
```

```
learned_clause = conflict_clause.copy()
current_level = self.decision_level

while True:
    # Collect literals in the learned clause
       assigned at the current level
    current_level_lits = [
        lit for lit in learned_clause
        if self.levels.get(abs(lit), -1) ==
           current_level
    ]
    if len(current_level_lits) <= 1:
        break

    # Find the most recently assigned literal in
       current_level_lits
    last_literal = None
    # Iterate through assignments in reverse order
        (most recent first)
    for var, _, lvl in reversed(self.
       decision_stack):
        if lvl != current_level:
            continue
        # Check if this variable is in
           current_level_lits
        for lit in current_level_lits:
            if abs(lit) == var:
                last_literal = lit
                break
        if last_literal is not None:
            break

    if last_literal is None:
        break  # No resolvable literals (should
           not happen)

    # Resolve with the reason clause of
       last_literal
    reason_clause = self.reasons.get(abs(
       last_literal))
```

```
            if reason_clause is None:
                break  # Decision literal; cannot resolve
                    further

            learned_clause = self.resolve(learned_clause,
                reason_clause, last_literal)

        # Determine the backjump level
        backjump_level = 0
        for lit in learned_clause:
            lvl = self.levels.get(abs(lit), 0)
            if lvl != current_level and lvl >
                backjump_level:
                backjump_level = lvl

        return learned_clause, backjump_level

    def backjump(self, level):
        """
        Backtracks the search to the given decision level
            by undoing assignments above that level.
        """
        new_stack = []
        for var, value, lvl in self.decision_stack:
            if lvl > level:
                if var in self.assignments:
                    del self.assignments[var]
                if var in self.levels:
                    del self.levels[var]
                if var in self.reasons:
                    del self.reasons[var]
            else:
                new_stack.append((var, value, lvl))
        self.decision_stack = new_stack

    def solve(self):
        """
        The main solving loop which alternates between
            unit propagation, conflict analysis, and
            branching.
```

```
        Returns:
          A satisfying assignment as a dictionary mapping
             variables to Boolean values if the formula is
              SAT;
          Otherwise, returns None indicating the formula
             is UNSAT.
        """
        while True:
            conflict = self.unit_propagate()
            if conflict:
                if self.decision_level == 0:
                    # Conflict at level 0 indicates an
                       unsolvable (UNSAT) condition.
                    return None
                learned_clause, backjump_level = self.
                   conflict_analysis(conflict)
                # Learn the clause by adding it to the
                   formula.
                self.formula.append(learned_clause)
                # Backjump to the appropriate decision
                   level.
                self.backjump(backjump_level)
                self.decision_level = backjump_level
            else:
                var = self.pick_branching_variable()
                if var is None:
                    return self.assignments
                self.decision_level += 1
                # For this example, we simply decide that
                   the variable is True.
                self.assignments[var] = True
                self.levels[var] = self.decision_level
                self.reasons[var] = None  # Decision
                   assignments have no reason clause.
                self.decision_stack.append((var, True,
                   self.decision_level))
```

La implementación anterior consiste en una clase `SATSolver` que realiza los pasos del algoritmo CDCL, dada una FNC escrita como un *array* de *arrays*, donde estos últimos representan las cláusulas. Cada variable está representada por un número entero positivo, y sus literales son ese número o su opuesto (negado).

El método `__init__(self, formula)` inicializa las estructuras internas:

- `assignments`: Mapea cada variable con su valor asignado (`True` o `False`).
- `levels`: Mapea cada variable con el nivel de decisión en que fue asignada.
- `reasons`: Mapea cada variable con la cláusula que forzó su asignación, o `None` si fue por decisión. Estos funcionan como los "arcos" del grafo de decisión.
- `decision_level`: Guarda el nivel actual de decisión.
- `decision_stack`: Pila que registra el historial de asignaciones como tuplas: (variable, valor, nivel de decisión).

El bucle principal del algoritmo se encuentra en el método `solve(self)`. Mientras no se detecte un conflicto, asigna valores a variables no asignadas, actualizando las estructuras del solver. Si no quedan variables por asignar, devuelve la asignación completa y declara la fórmula como satisfacible. Si se detecta una cláusula conflicto y el nivel actual de decisión es 0, entonces devuelve `None`, declarando que la fórmula es insatisfacible.

El método `unit_propagate(self)` realiza propagación unitaria dentro de un ciclo de punto fijo que se repite mientras haya cambios (es decir, mientras existan cláusulas unitarias). Si se encuentra una cláusula conflicto, se detiene y la devuelve. Si no hay conflicto, retorna `None`. Para determinar el estado de una cláusula (`'satisfied'`, `'conflict'`, `'unit'`, `'undefined'`), se utiliza el método `check_clause(self, clause)`.

Cuando se detecta una cláusula conflicto, se invoca `conflict_analysis(self, conflict_clause)`. Este analiza cuántos literales de la cláusula conflicto fueron asignados en el nivel de decisión actual. La cláusula aprendida debe contener exactamente un literal asignado en ese nivel. Si hay más de uno, se identifica el último literal asignado (según el grafo de decisión), se busca la cláusula que causó su asignación y se aplica el Principio de Resolución entre ella y la cláusula aprendida. Este proceso se repite hasta obtener una cláusula con un solo literal en el nivel actual.

Luego, se determina el *backjump level*, es decir, el nivel de decisión más alto presente en los literales de la cláusula aprendida (distinto del actual). Si este nivel es 0, CDCL considera la FNC como insatisfacible. Finalmente, se invoca el método `backjump(self, level)` que deshace las asignaciones por encima del nivel objetivo y actualiza las estructuras correspondientes.

2.3. DLIS

La heurística *Dynamic Largest Individual Sum* (DLIS) es un método aproximado de selección de variables que puede integrarse en un SAT *solver* con CDCL, con el objetivo de mejorar la eficiencia al decidir qué variable asignar en cada nivel de decisión.

DLIS mantiene un contador por cada variable que indica la máxima cantidad de cláusulas actualmente insatisfechas que podrían satisfacerse al asignarle uno de sus dos posibles valores (0 o 1). Es decir, para una variable x, se calcula la cantidad de cláusulas no satisfechas en las que aparece el literal x (forma positiva) y su complemento $\neg x$ (forma negativa). Denotamos:

- $count\_pos(x)$: cantidad de veces que x aparece positivamente en cláusulas insatisfechas.
- $count\_neg(x)$: cantidad de veces que $\neg x$ aparece en cláusulas insatisfechas.
- $dlis(x) = \text{máx}(count\_pos(x), count\_neg(x))$.

La próxima variable a seleccionar será aquella que maximice el valor de $dlis(x)$ entre todas las variables no asignadas:

$$x_k \mid dlis(x_k) = \text{máx}(dlis(x_i)), \quad i \in [1, n]$$

donde n es el total de variables de la FNC.

Obsérvese que si $count\_pos(x_k) > count\_neg(x_k)$, entonces se asigna a x_k el valor 1; en caso contrario, se asigna 0. Esto con el objetivo de satisfacer la mayor cantidad de cláusulas posibles en esa decisión. El cálculo debe aplicarse **solo a las variables no asignadas** y **solo considerar valores aún no explorados en el nivel actual de decisión**.

Esta estrategia de selección de variables puede integrarse en el algoritmo CDCL previamente presentado, sustituyendo la función `pick_branching_variable(self)` por una versión que implemente la heurística DLIS.

A continuación, se presenta una posible implementación de esta heurística, considerando como base el código del `SATSolver` previamente desarrollado.

```
def pick_branching_variable(self):
    """
    Selects the next unassigned variable using the DLIS
       heuristic.
    Returns (variable, value) to assign, or None if all
       variables are assigned.
    """
    pos_counts = {}
    neg_counts = {}
    # Count occurrences in unsatisfied clauses
```

```
for clause in self.formula:
    status, _ = self.check_clause(clause)
    if status == 'satisfied':
        continue
    for lit in clause:
        var = abs(lit)
        if var not in self.assignments:
            if lit > 0:
                pos_counts[var] = pos_counts.get(var, 0) + 1
            else:
                neg_counts[var] = neg_counts.get(var, 0) + 1
# Collect all variables in the formula to find unassigned ones not in any clause
all_vars = set()
for clause in self.formula:
    for lit in clause:
        all_vars.add(abs(lit))
unassigned_vars = [var for var in all_vars if var not in self.assignments]
if not unassigned_vars:
    return None
# For variables not in pos/neg counts, set counts to 0
for var in unassigned_vars:
    if var not in pos_counts:
        pos_counts[var] = 0
    if var not in neg_counts:
        neg_counts[var] = 0
# Score variables based on DLIS heuristic
scores = []
for var in unassigned_vars:
    pos = pos_counts[var]
    neg = neg_counts[var]
    max_count = max(pos, neg)
    total = pos + neg
    scores.append((-max_count, -total, var))  # Negative for ascending sort
scores.sort()  # Sorts by max_count (desc), then total (desc), then var (asc)
```

```
    var = scores[0][2]
    value = pos_counts[var] > neg_counts[var]
    return (var, value)
```

Las estructuras `pos_counts` y `neg_counts` se utilizan para almacenar, respectivamente, la cantidad de veces que el literal positivo y el literal negativo de cada variable no asignada aparece en las cláusulas actualmente insatisfechas. Este conteo es esencial para aplicar la heurística DLIS (*Dynamic Largest Individual Sum*) de forma efectiva.

El procedimiento comienza recorriendo todas las cláusulas de la fórmula. Por cada cláusula que no esté satisfecha, se inspeccionan sus literales, y se incrementa el contador correspondiente de cada literal no asignado. Por ejemplo, si un literal positivo x aparece en una cláusula insatisfecha, se incrementa `pos_counts[`x`]`, y si aparece su complemento $\neg x$, se incrementa `neg_counts[`x`]`.

Luego de este recorrido, se identifican todas las variables no asignadas que podrían no haber sido contadas, ya que podrían aparecer solamente en cláusulas ya satisfechas. A estas variables se les inicializa ambos contadores (positivo y negativo) en cero para garantizar una evaluación completa.

A continuación, se calcula el *score* para cada variable no asignada, definido como el máximo entre `pos_counts[`x`]` y `neg_counts[`x`]`. Este valor representa la cantidad máxima de cláusulas que podrían satisfacerse al asignarle a la variable el valor correspondiente. Las variables se ordenan según su *score* en orden descendente, y se elige aquella con el *score* más alto como la próxima variable a asignar. En caso de empate, se utilizan criterios secundarios como la suma total de apariciones y el índice de la variable.

El valor que se asigna a la variable seleccionada depende de cuál de los dos conteos es mayor. Si `pos_counts[`x`]` es mayor que `neg_counts[`x`]`, entonces se asigna el valor `True` (es decir, x), de lo contrario se asigna `False` (es decir, $\neg x$).

Esta heurística DLIS tiene como objetivo maximizar el número de cláusulas satisfechas con cada asignación, lo que en teoría podría reducir la cantidad total de decisiones requeridas para encontrar una solución o para detectar una contradicción. Sin embargo, su aplicación resulta costosa para instancias de gran tamaño, debido a que implica una revisión completa de todas las cláusulas insatisfechas en cada nivel de decisión. Esto conduce a una complejidad computacional de $O(n)$ por nivel, donde n es el número total de literales en la fórmula.

2.4. VSIDS

Por su parte, la heurística *Variable State Independent Decaying Sum* (VSIDS) prioriza asignar valores a aquellas variables cuyos literales hayan estado involucrados en conflictos recientes. Para ello, VSIDS mantiene un *score* para cada literal l (no por variable) que se incrementa cada vez que dicho literal aparece en cláusulas aprendidas a partir de conflictos.

Además, para evitar que literales involucrados en conflictos antiguos tengan mayor prioridad que los relacionados con conflictos recientes, cada cierta cantidad T de conflictos se actualizan los *scores* de todos los literales multiplicándolos por un factor de decaimiento α, con $0 < \alpha < 1$ (usualmente $\alpha = 0{,}95$).

Integrado en el algoritmo CDCL, VSIDS funciona del siguiente modo:

1. Se ejecuta el algoritmo CDCL.

2. Si ocurre un conflicto, se añade la cláusula aprendida $\mathbf{C_{learn}}$. Luego, para cada literal l en $\mathbf{C_{learn}}$, se actualiza su *score* con $score(l) = score(l) + \delta$, donde δ es un incremento positivo.

3. Si se alcanza el conflicto número T, se actualiza el incremento con $\delta = \delta \times \alpha$, con $0 < \alpha < 1$.

4. Si no ocurre un conflicto, se selecciona la variable v cuyo literal l_v cumple que su *score* es máximo entre todos los literales l_i, con $i = 1, 2, \ldots, 2n$, siendo n la cantidad de variables. Si l_v es positivo, se asigna el valor 1 a v, y 0 en caso contrario.

```python
def pick_branching_variable(self):
    """
    Selects the next unassigned variable using the VSIDS
       heuristic (highest activity).
    """
    candidates = []
    for var in self.activity:
        if var not in self.assignments:
            candidates.append(var)
    if not candidates:
        return None
    # Select the candidate with the highest activity; in
       case of tie, choose the smallest variable.
    max_activity = max(self.activity[var] for var in
       candidates)
    best_vars = [var for var in candidates if self.
       activity[var] == max_activity]
    best_vars.sort()  # Deterministic tie-breaking by
       choosing the smallest variable
    return best_vars[0]
```

Este método selecciona, de entre las variables aún no asignadas, aquella con el mayor *score* de acuerdo con la estrategia VSIDS. Para implementar este comportamiento, se añade a la clase *SATSolver* una estructura llamada *activity*, la cual asocia a cada variable un valor numérico que representa su actividad o relevancia. Esta estructura se inicializa al comienzo de la ejecución inspeccionando todas las variables presentes en las cláusulas de la fórmula, como se muestra a continuación:

```
# previus code
    for clause in self.formula:
        for lit in clause:
            var = abs(lit)
            if var not in self.activity:
                self.activity[var] = 0.0
```

Durante la ejecución del algoritmo CDCL, la estructura *activity* se actualiza dentro del método *solve(self)*. En particular, cada vez que se detecta un conflicto y se aprende una nueva cláusula, se incrementa la actividad de todas las variables que aparecen en dicha cláusula. Además, para priorizar los conflictos más recientes, se aplica un factor de decaimiento a todas las variables. A continuación se muestra el fragmento correspondiente:

```
decay_factor = 0.95
while True:
    conflict = self.unit_propagate()
    if conflict:
        if self.decision_level == 0:
            # Conflict at level 0 indicates an unsolvable
               (UNSAT) condition.
            return None
        learned_clause, backjump_level = self.
           conflict_analysis(conflict)
        # Learn the clause by adding it to the formula.
        self.formula.append(learned_clause)
        # Update activities for variables in the learned
           clause
        for lit in learned_clause:
            var = abs(lit)
            self.activity[var] += 1.0
        # Decay all activities
        for var in self.activity:
            self.activity[var] *= decay_factor
        # Backjump to the appropriate decision level.
```

```
        self.backjump(backjump_level)
        self.decision_level = backjump_level
# rest of code
```

De este modo, la heurística VSIDS logra priorizar variables relevantes en conflictos recientes, mejorando la toma de decisiones del solucionador sin necesidad de reexaminar toda la fórmula.

2.5. Reinicio (*restart*)

Las estrategias de reinicio tienen como objetivo evitar que el algoritmo de CDCL se estanque en regiones locales del espacio de búsqueda. Para ello, se permite reiniciar el árbol de decisiones, es decir, eliminar todas las asignaciones realizadas hasta el momento y comenzar nuevamente desde el nivel de decisión cero. Sin embargo, se conservan las cláusulas aprendidas durante el proceso, así como la información acumulada por las heurísticas de selección de variables (por ejemplo, las actividades de los literales en VSIDS o los conteos en DLIS).

Estas estrategias buscan que, al conservar el conocimiento adquirido (cláusulas aprendidas), el solucionador pueda explorar regiones más prometedoras del espacio de soluciones sin tener que recorrer nuevamente caminos improductivos.

Existen diversos criterios para decidir cuándo realizar un reinicio:

1. **Fijo**: se realiza un reinicio después de un número fijo k de conflictos. Esta es una estrategia sencilla pero poco adaptativa.

2. **Geométrico**: el número de conflictos entre reinicios crece de forma geométrica según la relación $r_0 = b$, $r_i = \alpha \cdot r_{i-1}$ con $\alpha > 1$. Esta estrategia permite reinicios más frecuentes al principio, reduciéndose con el tiempo. Un valor muy grande de α puede hacer que los reinicios sean demasiado esporádicos, mientras que un valor muy pequeño puede provocar una sobrecarga de reinicios.

3. **Luby**: utiliza la secuencia de Luby $(1,1,2,1,1,2,4,1,1,2,\dots)$ para definir los intervalos entre reinicios, según la fórmula $r_i = b \cdot \text{Luby}(i)$, donde b es un parámetro base que define el tamaño mínimo del intervalo. Esta estrategia tiene fundamentos teóricos que justifican su uso en entornos donde no se conoce a priori una buena política de reinicio.

4. **Glucose-style (basada en LBD)**: esta estrategia se basa en el cómputo de la medida Literal Block Distance (LBD) de las cláusulas aprendidas. Dada una cláusula C_{learn}, su LBD se define como el número de niveles de decisión distintos a los que pertenecen sus literales:

$$\text{LBD}(C_{learn}) = |\{\text{level}(l) \mid l \in C_{learn}\}|$$

La idea es que una cláusula con menor LBD involucra decisiones más cercanas entre sí, lo que la hace más relevante para guiar el proceso de resolución.

Para implementar esta estrategia, se mantienen dos promedios móviles de los LBD: uno para una ventana rápida de conflictos recientes (por ejemplo, los últimos 50 o 100) y otro para una ventana más larga (por ejemplo, los últimos 1000). Denotando estos promedios como μ_r (rápido) y μ_l (lento), se define un umbral $T > 1$. Si se cumple la condición:

$$\frac{\mu_r}{\mu_l} > T$$

entonces se considera que el solucionador está atrapado en una región poco productiva y se procede con un reinicio.

La incorporación de estrategias de reinicio, particularmente aquellas adaptativas como Luby o Glucose-style, ha demostrado ser fundamental para el éxito de solucionadores modernos de SAT, ya que permiten alternar entre exploración y explotación de manera eficiente en espacios de búsqueda complejos.

Ussando como base el código anterior de CDCL, una posible implementación para la estrategia Luby puede ser la siguiente:

```
# restart_luby.py

from collections import deque

def luby(u, k):
    """
    Generates the k-th value of the Luby sequence
       multiplied by u (unit run).
    """
    def _luby(i):
        # Encuentra el mayor j tal que i = 2^j - 1
        j = 1
        while (1 << j) - 1 < i:
            j += 1
        if i == (1 << j) - 1:
            return 1 << (j - 1)
        return _luby(i - (1 << (j - 1)) + 1)
    return u * _luby(k)

class SATSolverLuby:
    def __init__(self, formula, unit_run=100):
```

```
        from restart_luby import luby  # si ejecutas desde
            fuera
        self.formula = formula[:]
        self.assignments = {}
        self.levels = {}
        self.reasons = {}
        self.decision_level = 0
        self.decision_stack = []
        # Luby restart parameters
        self.unit_run = unit_run
        self.luby_idx = 1
        self.conflicts_since_restart = 0
        self.next_restart = luby(self.unit_run, self.
           luby_idx)

    # the same functions (literal_value, check_clause,
       unit_propagate,
    # pick_branching_variable, resolve, conflict_analysis,
        backjump)

    def solve(self):
        while True:
            conflict = self.unit_propagate()
            if conflict:
                self.conflicts_since_restart += 1
                if self.decision_level == 0:
                    return None
                learned_clause, backjump_level = self.
                   conflict_analysis(conflict)
                self.formula.append(learned_clause)
                self.backjump(backjump_level)
                self.decision_level = backjump_level

                # restart?
                if self.conflicts_since_restart >= self.
                   next_restart:
                    # Restart: clear assignments, preserve
                        learned clauses
                    self.assignments.clear()
                    self.levels.clear()
```

```
                self.reasons.clear()
                self.decision_stack.clear()
                self.decision_level = 0
                # Prepare next umbral
                self.luby_idx += 1
                self.next_restart = luby(self.unit_run
                    , self.luby_idx)
                self.conflicts_since_restart = 0
        else:
            var = self.pick_branching_variable()
            if var is None:
                return self.assignments
            self.decision_level += 1
            self.assignments[var] = True
            self.levels[var] = self.decision_level
            self.reasons[var] = None
            self.decision_stack.append((var, True,
                self.decision_level))
```

En el caso de la estrategia basada en la secuencia de Luby, esta se ha integrado al algoritmo CDCL modificando su estructura de control de conflictos. Para ello, se define una función `luby(u, k)` que genera el k-ésimo valor de la secuencia de Luby multiplicado por una unidad base `u`, la cual determina el número de conflictos que deben ocurrir antes de realizar un reinicio.

Dentro de la clase principal del solucionador, se inicializan los siguientes parámetros:

- `unit_run`: intervalo base de conflictos entre reinicios.
- `luby_idx`: índice actual dentro de la secuencia de Luby.
- `conflicts_since_restart`: contador de conflictos desde el último reinicio.
- `next_restart`: umbral de conflictos para el siguiente reinicio, calculado como `luby(unit_run, luby_idx)`.

Durante la ejecución del método `solve()`, cada vez que se detecta un conflicto y se realiza el correspondiente análisis y retroceso (“*backjump*”), se incrementa el contador de conflictos. Luego, se verifica si el número acumulado de conflictos ha alcanzado el umbral definido por la secuencia de Luby. En caso afirmativo, se ejecuta un reinicio que implica:

- Vaciar las estructuras de asignaciones, niveles de decisión, razones de propagación y la pila de decisiones.
- Reiniciar el nivel de decisión a cero.
- Calcular el nuevo umbral de reinicio actualizando el índice de Luby y recalculando `next_restart`.
- Reiniciar el contador de conflictos desde el reinicio.

Cabe destacar que durante el reinicio no se eliminan las cláusulas aprendidas, lo cual permite conservar información valiosa obtenida en decisiones anteriores. Esta estrategia favorece la salida de áreas del espacio de búsqueda poco prometedoras, maximizando la posibilidad de encontrar una solución en regiones más relevantes del espacio de soluciones.

De igual modo, una implementación para la estrategia *Glucose-Style (LBD-based)* sería como la que se muestra a continuación:

```
# restart_glucose.py

class SATSolverGlucose:
    def __init__(self, formula, lbd_window=50):
        self.formula = formula[:]
        self.assignments = {}
        self.levels = {}
        self.reasons = {}
        self.decision_level = 0
        self.decision_stack = []
        # Glucose-style parameters
        self.lbd_history = []
        self.window_size = lbd_window
        self.prev_avg_lbd = float('inf')

    #same base functions: literal_value, check_clause,
       unit_propagate, pick_branching_variable, resolve,
       backjump

    def conflict_analysis(self, conflict_clause):
        learned_clause, backjump_level = super().
           conflict_analysis(conflict_clause)
        # Calcular LBD (Literal Block Distance)
```

```
        levels = { self.levels.get(abs(l), 0) for l in
            learned_clause }
        lbd = len(levels)
        # Mantener ventana de LBDs
        self.lbd_history.append(lbd)
        if len(self.lbd_history) > self.window_size:
            self.lbd_history.pop(0)
        return learned_clause, backjump_level

    def should_restart(self):
        if len(self.lbd_history) < self.window_size:
            return False
        curr_avg = sum(self.lbd_history) / len(self.
            lbd_history)
        # Reiniciar si la media de LBD sube respecto al
            ciclo anterior
        if curr_avg > self.prev_avg_lbd:
            self.prev_avg_lbd = curr_avg
            return True
        self.prev_avg_lbd = curr_avg
        return False

    def solve(self):
        while True:
            conflict = self.unit_propagate()
            if conflict:
                if self.decision_level == 0:
                    return None
                learned_clause, backjump_level = self.
                    conflict_analysis(conflict)
                self.formula.append(learned_clause)
                self.backjump(backjump_level)
                self.decision_level = backjump_level

                # Glucose-style restart
                if self.should_restart():
                    self.assignments.clear()
                    self.levels.clear()
                    self.reasons.clear()
                    self.decision_stack.clear()
```

```
                self.decision_level = 0

        else:
            var = self.pick_branching_variable()
            if var is None:
                return self.assignments
            self.decision_level += 1
            self.assignments[var] = True
            self.levels[var] = self.decision_level
            self.reasons[var] = None
            self.decision_stack.append((var, True,
               self.decision_level))
```

2.6. Selección de cláusulas unitarias

Una de las estrategias más empleadas en solucionadores modernos de SAT para optimizar la propagación unitaria es
textitTwo Watched Literals (TWL). Esta técnica tiene como objetivo evitar el recorrido exhaustivo de todas las cláusulas en cada paso de propagación, mediante la vigilancia de dos literales por cláusula, denominados l_1 y l_2.

Durante la ejecución del algoritmo, si ambos literales vigilados en una cláusula no han sido evaluados, o al menos uno de ellos tiene valor verdadero, entonces no es necesario revisar dicha cláusula, ya que no es unitaria ni entra en conflicto. En caso contrario, si uno de los literales es asignado a falso, se intenta buscar dentro de la cláusula otro literal que no haya sido evaluado o que tenga valor verdadero, con el fin de reemplazar al literal que fue asignado a falso. Si no se encuentra un reemplazo válido y el otro literal vigilado aún no tiene valor, entonces la cláusula se vuelve unitaria, forzando la asignación del segundo literal. Si ambos literales vigilados son evaluados a falso, la cláusula se considera en conflicto.

Esta estrategia resulta altamente eficiente, ya que disminuye considerablemente el número de cláusulas que deben revisarse en cada nivel de decisión, optimizando así el rendimiento del algoritmo CDCL.

Para integrar la estrategia TWL en el código base de CDCL, es necesario modificar la estructura interna del solucionador, incluyendo los métodos de propagación, asignación y retroceso, entre otros. A continuación se presentará un ejemplo de implementación.

La implementación del algoritmo CDCL puede enriquecerse significativamente mediante la integración de estrategias que optimicen sus componentes críticos. Una de estas mejoras es la adopción del esquema Two Watched Literals (TWL), el cual transforma de manera sustancial la manera en que se gestiona la propagación unitaria.

En la versión básica de CDCL, el procedimiento de propagación unitaria examina exhaustivamente todas las cláusulas de la fórmula para identificar aquellas que se reducen a una sola literal no asignada bajo la interpretación parcial actual. Este enfoque, si bien correcto, resulta ineficiente, ya que el costo computacional de revisar cláusula por cláusula crece linealmente con el tamaño de la fórmula. En contraste, el enfoque basado en TWL introduce una estructura auxiliar que asigna a cada cláusula exactamente dos literales que actúan como "vigilantes". Mientras al menos uno de ellos no esté asignado a falso, se garantiza que la cláusula no puede causar conflicto ni ser unitaria, permitiendo así omitir su revisión durante la propagación.

El impacto de esta estrategia se observa de inmediato en la estructura del código. En la implementación enriquecida, se incorpora un diccionario que mapea cada literal a una lista de índices de cláusulas que lo están "observando". Esta organización permite que, ante la asignación de un literal, solo se revisen aquellas cláusulas que lo tienen como vigilante, reduciendo de forma drástica la cantidad de cláusulas analizadas en cada nivel de decisión. Además, se redefine el procedimiento de propagación unitaria para que, al detectar que un literal vigilado ha sido evaluado a falso, se intente encontrar otro literal en la cláusula que pueda ocupar su lugar como nuevo vigilante. Si esto no es posible y el otro literal vigilado tampoco satisface la cláusula, se deduce que esta es unitaria o de conflicto, y se actúa en consecuencia.

Además de modificar la propagación unitaria, la estrategia TWL afecta también la gestión de cláusulas aprendidas. Cada vez que se genera una nueva cláusula como resultado del análisis de conflictos, se seleccionan dos de sus literales para ser vigilados y se actualizan las estructuras de observación en consecuencia. Esto asegura que las optimizaciones ofrecidas por TWL se mantengan vigentes incluso cuando la fórmula evoluciona durante la ejecución del algoritmo.

En síntesis, la transición desde un esquema de propagación unitaria tradicional hacia un enfoque basado en Two Watched Literals representa una mejora sustancial en términos de eficiencia algorítmica. Al limitar el conjunto de cláusulas a inspeccionar en cada paso, se logra una aceleración significativa del proceso de inferencia, lo cual resulta especialmente beneficioso en instancias de gran escala, donde los costos de propagación pueden dominar el tiempo total de ejecución. Esta adaptación no solo evidencia una comprensión profunda de los mecanismos internos de CDCL, sino que también refleja el compromiso con la incorporación de estrategias de vanguardia para la resolución eficiente del problema de satisfacibilidad booleana.

Capítulo 3

Propuesta

Capítulo 4

Detalles de Implementación y Experimentos

El objetivo de esta tesis es comparar dos de las heuristicas integradas a CDCL, en este caso, dos de las que intentan dar soluci/'on al problema de selecci/'on de variables: Dichas estrategias, VSIDS y DLIS, son comparadas usando *restart* y no.

En el trabajo se usaron los lenguajes de programaci/'on C++, Python y bash. CaDiCaL est/'a programado en C++, por lo que es el lenguaje en el que est/'an implementadas las heur/'isticas. Python fue usado para automatizar los procesos de pruebas, guardar los resultados obtenidos y realizar los an/'lisis estad/'sticos. Finalemnte, bash se usa como parte de la çompilaci/'on"del solver, cuyos archivos como *configure* y *makefile* que vienen integrados al *solver*, son los encargados de hacerlo funcionar.

4.1. CaDiCaL

CaDiCaL fue el *solver* escogido para esta comparaci/'on, ya que cuenta con la posibilidad de activar y desactivar VSIDS (ya viene integrada), al igual que la estrategia *restart*, tambi/'en integrada en el solucionador.[1] Como ya se mencion/'o con anterioridad en este documento, pr/'acticamente todos los CDCL SAT *solvers* modernos no incluyen DLIS como heur/'istica de selecci/'on de variables, y CaDiCaL es uno de ellos. Por esta raz/'on, se decidi/'o integrar una implementaci/'on de DLIS a este solucionador, aprovechando la pol/'itica *open source* de su c/'odigo fuente.

[1]Cabe destacar que CaDiCaL ofrece esta misma posibilidad para much/'isimas heur/'sticas, adem/'as de estrategias para adaptar la v/'a de soluci/'on al tipo de problemas (citar documentaci/'n de CaDiCaL)

4.1.1. Integraci/'on de DLIS

Para integrar DLIS en CaDiCaL se modificaron los siguientes archivos:

- *internal.hpp*
- *internal.cpp*
- *decide.cpp*
- *options.hpp*

internal.hpp

En este archivo se declaran los m/'todos que ser/'an implementados como parte de la clase *Internal*:

```
// DLIS
int next_decision_variable_with_dlis ();
int count_literal_in_unsatisfied_binary_clauses(int lit)
    ;
```

internal.cpp

En *internal.cpp* se a/ nadi/'o el siguiente c/'odigo que contiene la l/'ogica de funcionamiento de DLIS. CaDiCaL almacena las variables de la FNC en una lista doblemente enlazada, en la que aquellas que ya han sido asignadas y las que a/'un no tienen un valor definido se encuentran separadas en dos grupos (dentro de la misma lista) separadas por un /'indice que marca el l/'mite entre ellas. Este valor va cambiando con cada asignaci/'on. Haciendo uso de esta estructura, el siguiente m/'etodo se encarga de comparar la ocurrencia, en cl/'ausulas insatisfechas, de los literales correspondientes a las variables sin asignar, y elige el de valor m/'aximo.

```
int Internal::next_decision_variable_with_dlis () {
  int best_lit = 0;
  int best_score = -1;

  //Empezamos en el primer nodo ``no asignado'' de la cola
  int idx = queue.unassigned;

  // Recorremos la lista de variables sin asignar (link(
     idx).prev nos lleva
  //al siguiente ``sin asignar''), igual que en
     next_decision_variable_on_queue().
```

```
  while (idx) {
    // Si idx ya tiene val(idx) != 0, saltemos (aunque en
       teor\'ia queue.unassigned
    // siempre apunta a un idx tal que val(idx)==0; no
       obstante, por seguridad lo comprobamos).
    if (val(idx) == 0) {
      // Calcular la puntuaci'\on DLIS para +idx y para -
         idx
      int pos_score =
         count_literal_in_unsatisfied_binary_clauses(idx);
      int neg_score =
         count_literal_in_unsatisfied_binary_clauses(-idx)
         ;

      // Comparar con el mejor hasta ahora
      if (pos_score > best_score) {
        best_score = pos_score;
        best_lit   = idx;     // literal positivo
      }
      if (neg_score > best_score) {
        best_score = neg_score;
        best_lit   = -idx;    // literal negativo
      }
    }
    // Avanzamos al siguiente \'indice ``no asignado'':
    idx = link(idx).prev;
  }

  LOG ("next DLIS decision literal %d with score %d",
     best_lit, best_score);
  return abs(best_lit);
}
```

Como puede observarse, este m/'etodo hace un llamado a

```
count_literal_in_unsatisfied_binary_clauses(idx)
```

que es el encargado de realizar el conteo de ocurrencias de literales por variable sin asignar, por cláusula insatisfecha.

```
/// Cuenta ocurrencias de un literal 'lit' en cl\'ausulas
   binarias
```

```
/// no basura y no satisfechas, optimizando la llamada a
   val(lit).
int Internal::count_literal_in_unsatisfied_binary_clauses
   (int lit) {
  int count = 0;
  // Cacheamos una sola vez el valor de 'lit'.
  const signed char val_lit = val (lit);
  // Recorremos su lista de watchers
  const Watches &ws = watches (lit);
  for (const Watch &w : ws) {
    if (!w.binary ()) continue;          // Solo cl\'
       ausulas binarias
    Clause *c = w.clause;
    if (!c || c->garbage) continue;      // Saltar nulos y
        garbage
    int other = w.blit;                  // El otro
       literal de la cl\'ausula
    // Si 'lit' o 'other' ya son verdaderos, la cl\'ausula
        est\'a satisfecha
    if (val_lit > 0 || val (other) > 0) continue;
    ++count;
  }
  return count;
}
```

Es importante aclarar que el c/'alculo del *score* de cada literal no se hace tal cual dicta DLIS, pues en aras de mantener la consistencia del código con la implementaci/'on de CaDiCaL se usaron estrategias y estructuras que ya vienen implementadas, y a las cuales se acoplan las heur/'isticas de decisi/'on de variables que iya incorpora el solucionador. Por ejemplo, en el c/'odigo anterior puede verse que el recorrido para efectuar el conteo de ocurrencias de un literal no se realiza sobre todas las cláusulas insatisfechas, sino que, haciendo uso de la estrategia *Two Watched Literals* (TWL) (hacer referencia al marco teórico)

(assumptions–>hacer referencia a donde se explican en el marco teorico)

4.1.2. Empleo de *flags* en la l/'inea de comandos

Los *flags* usados en la l/'inea de comandos para combinar las heur/'isticas fueron:

4.2. Problemas

4.3. Generador de problemas

4.4. Estad/'isticas

Conclusiones

El dilema en CDCL CDCL mitiga parcialmente este problema mediante el aprendizaje de cláusulas, pero no elimina la dependencia de la selección inicial de variables. Por ejemplo:

Si VSIDS elige variables periféricas en un problema con núcleos críticos (ej: PHP), el solver gastará recursos en regiones irrelevantes.

Si DLIS prioriza literales frecuentes en problemas con restricciones jerárquicas (ej: scheduling), perderá la capacidad de explotar correlaciones locales.

Esta interdependencia entre heurísticas y estructura del problema explica por qué, a pesar de los avances en CDCL, no existe una estrategia universalmente óptima. La selección de variables sigue siendo un cuello de botella teórico y práctico, especialmente al escalar a miles de variables con relaciones complejas.

La introducción de CDCL marcó un avance al reemplazar el retroceso (backtrack) cronológico con uno dirigido por conflictos, pero su éxito está ligado a la sinergia entre aprendizaje y selección de variables. Mientras las cláusulas aprendidas reducen el espacio de búsqueda, las heurísticas de selección determinan cómo se navega en él. Un desbalance entre estos componentes condena al solver a un rendimiento subóptimo, perpetuando la necesidad de estudios comparativos como el propuesto en esta tesis.

La efectividad de los solucionadores modernos de satisfacibilidad booleana (SAT) descansa en gran medida en la capacidad de integrar de manera sinérgica estrategias de optimización y heurísticas dentro del esquema general del algoritmo Conflict-Driven Clause Learning (CDCL). Lejos de constituir componentes aislados, estas mejoras operan como módulos interdependientes que refinan y potencian el comportamiento global del algoritmo, permitiéndole escalar de manera eficiente frente a instancias de gran complejidad.

Estrategias como los reinicios adaptativos, ejemplificados por los esquemas de Luby o Glucose-style, permiten una regulación dinámica del balance entre exploración y explotación dentro del espacio de búsqueda. Al reconocer patrones de estancamiento mediante métricas como el LBD, estos mecanismos inducen reinicios calculados que promueven la reorientación del proceso de deducción, sin perder el conocimiento acumulado en forma de cláusulas aprendidas.

Simultáneamente, estructuras como Two Watched Literals (TWL) reformulan la propagación unitaria, eliminando redundancias computacionales y concentrando la atenci'on en un subconjunto relevante de cláusulas. Esto reduce de forma considerable el costo por iteración, acelerando la inferencia sin comprometer la corrección de las deducciones.

Estas estrategias, cuando se integran con coherencia en la arquitectura del solucionador, transforman el comportamiento de CDCL en una maquinaria altamente adaptativa. Cada componente contribuye a un objetivo común: reducir el tiempo necesario para alcanzar una solución, ya sea satisfactible o insatisfactible, mediante una navegación informada y eficiente del espacio de búsqueda. Esta modularidad permite, además, la extensibilidad del algoritmo, favoreciendo la experimentación y la incorporación de nuevas heurísticas en función de las demandas de cada dominio de aplicación.

En suma, la integración de heurísticas avanzadas y estrategias de optimización en el marco de CDCL no sólo representa una mejora en rendimiento, sino que constituye una evolución conceptual en el diseño de algoritmos de búsqueda, marcando un camino claro hacia solucionadores más inteligentes, robustos y generalizables.

Recomendaciones

Recomendaciones