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 (CSPs) constituyen un paradigma fundamental para modelar desafíos combinatorios en inteligencia artificial, investigación operativa y ciencias de la computación. Formalmente, un CSP se define mediante una tripleta (V, D, C), donde V es un conjunto de variables, D sus dominios discretos finitos, y C un conjunto de restricciones que limitan las combinaciones válidas de valores **39**. Por ejemplo, en un problema de asignación de horarios, V representaría cursos, D los horarios disponibles, y C las reglas que evitan superposiciones. La solución óptima no solo satisface todas las restricciones, sino que también optimiza criterios como la utilización de recursos **40**.

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

El Problema de la Satisfacibilidad Booleana (SAT) emerge como un CSP restringido donde los dominios son binarios $(0, 1)$ y las restricciones se expresan en Forma Normal Conjuntiva (FNC) **43**. Una fórmula en FNC es una conjunción de cláusulas, donde cada cláusula es una disyunción de literales (variables o sus negaciones) **6**. Determinar si existe una asignación de valores que satisfaga todas las cláusulas equivale a resolver un CSP binario con restricciones específicas.

La relevancia teórica de SAT radica en su condición de problema NP-completo, demostrada por Cook y Levin en 1971 **2,24**. Este estatus implica dos consecuencias cruciales: primero, cualquier problema en NP puede reducirse a SAT en tiempo polinomial **26**; segundo, la existencia de un algoritmo polinomial para SAT implicaría $P = NP$, colapsando la jerarquía de complejidad **2**. Aunque en la práctica los solvers modernos resuelven instancias con millones de variables **3**, en el peor caso SAT

conserva una complejidad exponencial inherente **27**.

1.1.2. Ineficiencias Fundamentales de SAT

El método de fuerza bruta para SAT —evaluar todas 2^n asignaciones posibles mediante tablas de verdad— ilustra su naturaleza intratable en el peor caso **19**. Esta explosión combinatoria se agrava en fórmulas sin estructura discernible, donde técnicas como la propagación unitaria o el aprendizaje de cláusulas tienen impacto limitado **30**. Por ejemplo, en instancias aleatorias de 3-SAT cerca del umbral de fase (aproximadamente 4.26 cláusulas por variable **30**), los algoritmos clásicos como DPLL exhiben un crecimiento exponencial en el tiempo de ejecución **27**.

Aun así, SAT destaca como herramienta práctica gracias a dos factores: (1) la capacidad de codificar CSPs genéricos en FNC mediante técnicas como *encodings* directos o Tseitin **44**, y (2) el desarrollo de solvers CDCL que explotan regularidades empíricas en instancias industriales **3**. Esta dualidad entre dificultad teórica y éxito práctico sitúa a SAT en el núcleo de aplicaciones como verificación de hardware, planificación autónoma y criptoanálisis **39**.

1.1.3. Relación Práctica entre CSPs y SAT

Aunque los CSPs permiten modelar problemas con dominios arbitrarios y restricciones globales —ventaja frente a la rigidez booleana de SAT—, su resolución nativa mediante métodos como *backtracking* o consistencia de arco sufre de limitaciones similares en escalabilidad **46**. Por ello, una estrategia común consiste en traducir CSPs a SAT, aprovechando décadas de optimizaciones en solvers CDCL **44**. Estudios empíricos demuestran que codificaciones eficientes (e.g., *order encoding* para restricciones de orden) reducen hasta un 60 % el tiempo de resolución frente a enfoques CSP nativos **45**.

No obstante, esta traducción no está exenta de trade-offs. Mientras SAT favorece restricciones locales y cláusulas pequeñas, los CSPs manejan eficientemente restricciones globales (e.g., *alldifferent*) mediante propagadores especializados **46**. Por ejemplo, en problemas de asignación de turnos hospitalarios, un modelo CSP con restricciones de recurso puede resolver instancias en minutos, mientras su traducción a SAT requiere horas debido a la explosión de cláusulas **46**. Esta dicotomía subraya la importancia de seleccionar el paradigma adecuado según la estructura del problema.

1.1.4. Enfoques de Solución: CP, MILP y SAT

La resolución de CSPs y SAT se enmarca en tres metodologías principales:

- **Programación con Restricciones (CP) 40**: Combina búsqueda sistemática con propagación de restricciones, ideal para dominios discretos y restricciones no lineales.
- **Programación Entera Mixta (MILP) 40**: Utiliza relajaciones lineales y técnicas branch-and-bound, óptima para problemas con estructura matemática explícita.
- **Satisfacibilidad Booleana (SAT) 44**: Emplea algoritmos CDCL con aprendizaje de cláusulas, eficaz para problemas binarios o altamente restringidos.

Cada enfoque tiene un nicho de aplicabilidad. Por ejemplo, CP domina en scheduling con restricciones complejas, MILP en optimización logística lineal, y SAT en verificación formal donde la traducibilidad a FNC es natural **39,40**. La elección depende críticamente de la capacidad para explotar la estructura subyacente del problema —factor que explica el éxito paradójico de SAT pese a su complejidad teórica **3**.

Esta sección sienta las bases para analizar en detalle la evolución de los SAT solvers (Sección 1.2), donde se explorará cómo técnicas como CDCL superaron las ineficiencias teóricas mediante innovaciones algorítmicas pragmáticas.

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

La historia de los SAT *solvers* representa un hito en la ciencia computacional, transformando un problema teóricamente intratable en una herramienta práctica de amplio uso industrial. Esta evolución se sustenta 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 no solo refleja avances técnicos, sino 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 inicialmente en el contexto de la lógica proposicional, constituye la base teórica de muchos algoritmos SAT. Este principio

[1]Problema NP-completo que busca determinar si existe una asignación de verdad que satisfaga una fórmula lógica dada.

permite derivar nuevas cláusulas a partir de un par de cláusulas existentes que contengan literales complementarios, reduciendo progresivamente la fórmula hasta detectar una contradicción o verificar su satisfacibilidad. Aunque teóricamente completo para fórmulas en Forma Normal Conjuntiva (FNC), su aplicación directa resulta impráctica debido al crecimiento explosivo en el número de cláusulas generadas **12**. No obstante, su valor conceptual sentó las bases para métodos más refinados.

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

Introducido en 1960 **5**, el algoritmo DP fue el primer esfuerzo sistemático para operacionalizar el PR en un marco computacional. Su estrategia se centraba en la eliminación iterativa de variables mediante dos reglas: (1) *eliminación de literales puros* (asignar valores a variables que aparecen con un único signo en todas las cláusulas) y (2) *resolución dirigida* para eliminar variables seleccionadas. Aunque evitaba el espacio exponencial de las tablas de verdad, su dependencia de la resolución lo hacía vulnerable a explosiones combinatorias en el número de cláusulas intermedias **12**. Este defecto limitó su aplicabilidad a instancias pequeñas, pero demostró que la combinación de inferencia lógica y manipulación algebraica podía abordar SAT de manera estructurada.

1.2.3. Algoritmo DPLL: Búsqueda Inteligente con Retroceso

En 1962, Davis, Putnam, Logemann y Loveland refinaron DP mediante un giro paradigmático: reemplazaron la resolución explícita por una *búsqueda con retroceso* en el espacio de asignaciones, dando origen al algoritmo DPLL **12**. Este método integra tres componentes clave:

- **Propagación Unitaria**: Asignación forzada de literales en cláusulas unitarias, reduciendo la fórmula antes de tomar decisiones.
- **Eliminación de Literales Puros**: Optimización heredada de DP para simplificar la fórmula.
- **Bifurcación con Retroceso**: Elección heurística de valores para variables no determinadas, seguida de retroceso ante conflictos.

Al operar como un recorrido en profundidad del árbol de decisiones, DPLL evita el costo de memoria de DP y se convirtió en el núcleo de los SAT *solvers* clásicos **7**. Sin embargo, su eficiencia se veía mermada en problemas industriales debido a la exploración redundante de subespacios conflictivos y la ausencia de mecanismos

para capitalizar información de conflictos pasados **8**. Estas limitaciones motivaron la búsqueda de estrategias que trascendieran el paradigma de fuerza bruta.

1.2.4. CDCL: La Revolución de los *Solvers* Modernos

El algoritmo CDCL emergió como solución a las limitaciones estructurales de DPLL en instancias industriales. Dos innovaciones fueron fundamentales: el *aprendizaje de cláusulas* y el *retroceso no cronológico* **12**. El primero transforma los conflictos en conocimiento estructural mediante el análisis de implicaciones, generando cláusulas que encapsulan inconsistencias para podar futuras ramas de búsqueda **17**. El segundo permite saltar a niveles de decisión estratégicos, evitando la exploración redundante de subárboles irrelevantes **6**. Juntos, estos mecanismos resolvieron tres problemas críticos de DPLL: la explosión combinatoria por conflictos repetidos, la falta de adaptación heurística durante la búsqueda y la ineficiencia del retroceso cronológico **5,7,8**.

Mejoras Clave en CDCL: Heurísticas y Estrategias

Si bien CDCL superó teóricamente a DPLL, su eficacia práctica requirió mejoras adicionales. Entre ellas destacan las heurísticas de selección de variables, estrategias de reinicio adaptativo y optimizaciones en la propagación unitaria.

Las *heurísticas de selección de variables* representan un equilibrio entre complejidad teórica y pragmatismo. DLIS (*Dynamic Largest Individual Sum*), uno de los primeros enfoques dinámicos **17**, prioriza variables frecuentes en cláusulas insatisfechas mediante conteos estáticos **18**. Aunque intuitivo, su costo computacional lineal ($O(n)$) y su incapacidad para adaptarse a la dinámica de búsqueda lo limitaron en instancias industriales **22**. VSIDS (*Variable State Independent Decaying Sum*), en contraste, introdujo un modelo basado en actividades dinámicas con decaimiento exponencial **27**. Al incrementar puntuaciones de variables en conflictos recientes y aplicar factores de olvido periódicos **28**, VSIDS logró un balance empírico entre adaptación y costo computacional ($O(1)$ mediante colas de prioridad) **30**. Sin embargo, ambas heurísticas son aproximaciones: DLIS por su simplicidad estática, VSIDS por depender de correlaciones no garantizadas entre conflictos y tiempo de resolución **10,35**.

Las *estrategias de reinicio adaptativo* surgieron para contrarrestar el estancamiento en regiones locales del espacio de búsqueda, un efecto colateral de heurísticas como VSIDS **44**. Técnicas como los reinicios LBD-driven en Glucose **14** vinculan la frecuencia de reinicios al Literal Block Distance promedio de cláusulas aprendidas, indicador indirecto de estancamiento. Métodos más sofisticados, como MAB (*Multi-Armed Bandit*) en Kissat **37**, emplean aprendizaje en línea para seleccionar políticas de reinicio óptimas según el contexto de búsqueda. Estas estrategias reducen hasta un 70 % el tiempo de resolución en instancias modulares al reorientar la exploración sin perder cláusulas aprendidas críticas **21**.

En la *propagación unitaria*, optimizaciones como el esquema de Dos Literales Vigilados (TWL) **29** minimizaron el costo de detectar cláusulas unitarias, evitando inspeccionar el 90 % de las cláusulas en cada asignación **12**. Técnicas posteriores como CFUP (*Core First Unit Propagation*) **96** priorizaron cláusulas con LBD ≤ 7, basándose en la observación empírica de que el 80 % de los conflictos ocurren en estas estructuras. Si bien CFUP carece de fundamentación teórica, su efectividad práctica lo consolidó como estándar en solvers como CaDiCaL **30**.

Otras Contribuciones Relevantes

El ecosistema de mejoras en CDCL incluye avances complementarios. La gestión de cláusulas aprendidas mediante métricas como LBD permitió eliminar cláusulas redundantes y reciclar aquellas con alto valor predictivo (*glue clauses*) **14-17**. La integración con búsqueda local, mediante técnicas como *re-phasing* basado en frecuencias de conflicto **1**, mejoró hasta un 40 % el rendimiento en instancias satisfacibles. En paralelo, meta-heurísticas como AutoSAT **35** exploraron el uso de modelos de lenguaje grande (LLMs) para optimizar parámetros dinámicos, aunque su adopción práctica sigue siendo incipiente.

1.2.5. CDCL como Motor para Problemas con Restricciones

La efectividad de CDCL en CSPs depende críticamente de su arquitectura adaptativa. VSIDS explota patrones emergentes en restricciones codificadas, mientras los reinicios LBD-driven manejan la modularidad típica de estos problemas **15,21**. Técnicas como CFUP, al priorizar cláusulas çore", reflejan jerarquías implícitas en redes de restricciones **96**. Esta sinergia explica por qué solvers como Kissat y Glucose se han convertido en motores subyacentes para sistemas de verificación y planificación industrial **12**.

1.2.6. Conclusión: De la Teoría a la Revolución Práctica

La transición desde el PR y DP hasta CDCL ilustra cómo innovaciones algorítmicas pueden superar barreras teóricas aparentemente infranqueables. Mientras DPLL sentó las bases de la integración búsqueda-inferencia, CDCL introdujo un paradigma de *aprendizaje reflexivo*, donde cada conflicto alimenta la inteligencia del solver. Esta evolución, respaldada por mejoras en heurísticas y estructuras de datos, ha convertido a los SAT *solvers* en pilares de la computación moderna, demostrando que incluso problemas NP-completos pueden abordarse eficientemente en escenarios prácticos **5,7,11**.

1.3. Parámetros de Evaluación para Solvers CDCL: Taxonomía de *Benchmarks*

La evaluación rigurosa de solvers CDCL —particularmente en el análisis comparativo de heurísticas como VSIDS y DLIS, estrategias de reinicio y selección de cláusulas unitarias— requiere una taxonomía precisa de instancias SAT. Estas se clasifican no solo por su origen, sino por propiedades intrínsecas que revelan fortalezas y limitaciones algorítmicas **15,17**.

1.3.1. Clasificación por Origen y Estructura

Las competiciones anuales de SAT (*SAT Competition*) establecen cuatro categorías canónicas **17**:

Instancias de Aplicación: Derivadas de problemas industriales como verificación de circuitos o planificación logística **11**, exhiben estructuras modulares y *backdoors* pequeños —subconjuntos de variables cuya asignación correcta simplifica drásticamente la solución **21,23**. Estas características permiten a heurísticas dinámicas como VSIDS explotar patrones locales mediante el rastreo de conflictos recientes **20**.

Instancias Combinatorias Dificultosas: Construidas artificialmente para desafiar a los solvers (e.g., codificaciones del principio del palomar) **16**, carecen de la estructura implícita de las aplicaciones reales pero poseen simetrías y dependencias globales que prueban la capacidad de generalización de las heurísticas **19**.

Instancias Aleatorias: Generadas mediante modelos como Random k-SAT **17**, su falta de estructura las convierte en un desafío para CDCL, donde técnicas como el aprendizaje de cláusulas muestran eficacia limitada **21**. Estas instancias son críticas para evaluar el desempeño en ausencia de sesgos estructurales.

Instancias Ágiles: Resultantes de la conversión de problemas de lógicas superiores a SAT (*bit-blasting*) **19**, testean la capacidad de manejar fórmulas con alta densidad de cláusulas y variables auxiliares.

1.3.2. Clasificación por Satisfacibilidad y Propiedades Estructurales

La naturaleza SAT o UNSAT de una instancia influye significativamente en el comportamiento del solver. Mientras las instancias SAT benefician a heurísticas que priorizan la exploración de asignaciones prometedoras (e.g., VSIDS con *phase saving*), las UNSAT requieren estrategias que aceleren la refutación mediante cláusulas aprendidas de alto impacto **41**.

Propiedades estructurales como la modularidad, *treewidth* y presencia de *backbones* (variables con valor fijo en todas las soluciones) **20,22** ofrecen insights adicionales.

Por ejemplo, instancias con alta modularidad —típicas en aplicaciones industriales— permiten a los reinicios adaptativos reorientar la búsqueda hacia comunidades no exploradas **21**, mientras que un *treewidth* bajo correlaciona con tiempos de resolución reducidos debido a la eficiencia de la propagación unitaria **20**.

1.3.3. Densidad y Transición de Fase

La relación cláusulas/variables (densidad) determina la dificultad en instancias aleatorias. Para 3-SAT, la transición de fase alrededor de 4.27 cláusulas por variable **25** marca el pico de complejidad, donde métodos como DLIS —basados en frecuencias estáticas— fallan ante la ausencia de patrones explotables **17**. En contraste, VSIDS combinado con reinicios dinámicos (*LBD-driven*) logra navegar estas regiones mediante un balance entre explotación local y exploración global **21**.

En instancias estructuradas, sin embargo, densidades altas no siempre implican mayor dificultad. Codificaciones compactas de problemas como el coloreado de grafos pueden ser resueltas eficientemente si los *solvers* identifican *backdoors* mediante heurísticas sensibles al contexto **23**.

1.3.4. Relevancia para la Evaluación de Heurísticas

La elección de *benchmarks* es crucial al comparar técnicas como VSIDS y DLIS. Mientras DLIS —dependiente de conteos estáticos de aparición— rinde mejor en instancias aleatorias lejos de la transición de fase **18**, VSIDS domina en aplicaciones reales gracias a su adaptación dinámica **30**. Estrategias de reinicio como MAB **37** muestran ventajas en instancias modulares, donde reiniciar preservando cláusulas *glue* acelera la cobertura del espacio de búsqueda **21**.

Propiedades como la presencia de *backbones* o bajo *treewidth* permiten diseccionar el impacto de técnicas específicas. Por ejemplo, la selección de cláusulas unitarias mediante CFUP **96** reduce hasta un 40 % el tiempo en instancias con alta densidad de cláusulas *core* ($LBD \leq 7$), pero tiene efecto marginal en fórmulas sin estructura comunitaria **21**.

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

Capítulo 2

Algoritmos

2.1. DP/DPLL

2.1.1. Principio de Resolución (PR)

El Principio de Resolución (PR) es una de las primeras técnicas aplicadas para intentar resolver SAT.

Dada una fórmula de la Lógica Proposicional escrita en Forma Normal Conjuntiva (FNC), esta se representa en su forma conjuntual, donde cada cláusula constituye un conjunto de literales y la fórmula en general es un conjunto de conjuntos. Gracias al concepto de "conjunto" esta representación evita la repetición de literales en las cláusulas, así como aquellas que aparezcan más de una vez en la FNC.

Tomando esta representación como entrada, PR busca iterativamente pares de clásulas que contengan literales opuestos, para a partir de ellas generar una nueva clásula que constituya la unión de ambas, quitando eliminando el literal en cuestión y su opuesto. Es decir:

Sean $\mathbf{B}$ y $\mathbf{C}$ dos cláusulas de la FNC $\mathbf{A}$, tales que $l \in \mathbf{B}$ y $\neg l \in \mathbf{C}$, donde l es un literal y $\neg l$ su opuesto; entonces la cláusula resultante sería:

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

Aquí las cláusulas $\mathbf{B}$ y $\mathbf{C}$ se denominan "cláusulas padres" o "premisas" y $\mathbf{D}$ constituye el "solvente" o "conclusión" de $\mathbf{B}$ y $\mathbf{C}$.

Téngase en cuenta el siguiente ejemplo para una mejor comprensión.

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

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

El objetivo principal de PR es refutar $\mathbf{A}$ si esta es insatisfacible, derivando una cláusula vacía ($\{\}$). Es decir, si $\{\} \in \mathbf{A}$ entonces $\mathbf{A}$ es insatisfacible, y si $\{\} = \mathbf{A}$ entonces $\mathbf{A}$ es satisfacible.

Resolución Unitaria (RU)

Un caso particular de PR es Resolución Unitaria (RU), donde una de las "cláusulas padres" es una cláusula unitaria [1] Por ejemplo:

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

2.1.2. Davis-Putnam (DP)

Uno de los primeros algoritmos para resolver SAT fue Davis-Putnam (DP), donde PR constituye uno de sus pilares. DP realiza tres procedimientos fundamentales: Propagación Unitaria (PU), Eliminación de Literales Puros (ELP) y Resolución Basada en División (RD). PU busca en la FNC de entrada clásulas unitarias y procede a eliminarlas de la fórmula, además de eliminar el literal complemntario de cada cláusula donde aparezca. ELP busca los literales que tengan una única polaridad en toda la fórmula (no exista su complementario) y procede a eliminar aquellas cláusula que contengan literales puros. Ambas, PU y ELP, son preprocesamientos que buscan simplificar lo más posible la FNC. Una vez realizados, DP procede con RD donde asigna un valor (0 o 1) a una variable y continúa recursivamente aplicando DP hasta encontrar poder decidir si la FNC es satisfacible o no. Véase el ejemplo a continuación:

Sean las siguientes, fórmulas de la Lógica Proposicional

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

A paritr de la conjunción de estas, se obtiene la siguiente FNC:

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

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

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

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

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

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

[1] Cláusula que contiene un único literal, por ende su valor se ve forzado a ser 1.

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

Aplicando **PU** en $\{s\}$:

$$\{\}$$

Luego la fórmula es satisfacible.

DP requiere una memoria exponencial ya que evalúa todas las posibles asignaciones para las variables, generando un árbol de decisión como espacio de búsqueda que crece exponencialmente.

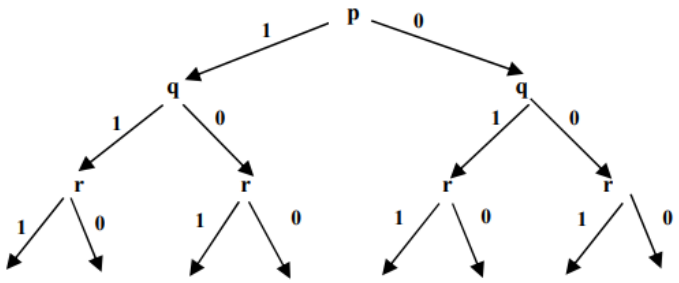

Figura 2.1: Posible espacio de búsqueda de una FNC

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

El algoritmo Davis-Putnam-Logemann-Lovelans (DPLL) aplica las mismas bases que DP, excepto que resuelve el problema de la memoria exponencial al realizar un *backtrack* cronológico (al nivel anterior de decisión) en el árbol de asignaciones una vez encontrada una "cláusula de conflicto"[2]. Con este método, DPLL utiliza una estrategia *"lazy"* para generar el árbol, dado que antes de ramificarse (otorgar un valor a una variable) verifica que no haya conflictos, garantizando que las asiganaciones válidas para la fórmula de entrada se encuentren en las hojas del árbol de decisión. Obsérvese que si un conflicto se produce en el nivel de decisión 0 y ambos valores

[2]Se denomina cláusula de conflicto a aquella cláusula cuyos literales evaluaron a 0 tras una asignación.

para la variable ya han sido examinados, entonces se concluye que la fórmula es insatisfacible. Este proceso realizado por DPLL se denomina ramificación + propagación unitaria + retroceso.

Cabe destacar que DPLL tiene una etapa de preprocesamiento de la FNC, sobre la cual se aplican leyes de la Lógica Proposicional, así como se eliminan cláusulas redundantes mediante la aplicación de la subsunción de cláusulas[3].

Obsérvese el siguiente ejemplo:

Sea la FNC:

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

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:

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

Aplicando PU:

$$\{\neg r\}$$

$$\emptyset$$

Luego la FNC es satisfacible.

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

En primer lugar la selección de la variable a asignar un valor influye en la "forma" que tomará el espacio de búsqueda, por lo que malas decisiones en este sentido conllevan a caminos más largos en la búsqueda de una solución. Análogamente se gana en eficiencia considerando heurísticas en la selección de variables. (poner ejemplo)

[3]Sean C y C', cláusulas de una FNC; si $C' \subseteq C$, entonces se elimina C' de la FNC.

En segundo lugar, el *backtrack* cronológico a partir de un conflicto obliga a explorar el resto de las posibles asignaciones de las variables en niveles anterirores, potencialmente, de forma innecesaria, sobre todo para aquellos casos donde la asignación causante del conflicto se encuentre a k niveles de distancia del nivel del conflicto. Además, DPLL no aprovecha las cláusulas que han resultado conflictos, es decir, no aprende de ellas, por lo que es vulnerable a cometer el mismo error (mismo patrón incorrecto de asignaciones para las variables involucradas en el conflicto). Esto lo hace susceptible a repetir errores de forma recurrente.

Finalmente, el problema de selección de cláusulas unitarias, influye también en la eficiencia del algoritmo. Se puede decir que guarda relacón 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 2 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 nulo 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. 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 1 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.

En caso de que el nivel del *backjump* sea el nivel 0, CDCL considera la FNC como insatisfacible.

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 de CDCL dada una FNC escrita en forma de *array* de *arrays* donde estos últimos son las cláusulas. Por su parte, cada variable está representada por un número entero positivo y sus literales serán el propio número y su opuesto.

El primer método inicializa las estructuras con las que trabajará:

- *assigments*: Mapea cada variable asignada con su valor.
- *levels*: Mapea cada variable aasignada con el nivel de decisión en el que se le dio su valor.
- *reasons*: Estos serán los "arcos" del grafo de decisión, pues mapea por cada variable asignada la cláusula que provocó su asignación, y *None* para el caso de variables asignadas por decisión.
- *decision_level*: guarda el actual nivel de decisión.
- *decision_stack*: registra el grafo al guardar en una pila tuplas de 3 elementos: variable asignada, valor que tomó y el nivel de decisión en el que se le asignón su valor.

El bucle principal del algoritmo se encuentra en el método *solve(self)* que mientras no haya conflicto asigna valores a las variables y actualiza las estructuras de la clase, y en caso de no existir más variables por asignar, entonces devuelve la solución (asignaciones para cada variable) declarando la fórmula como satisfacible. En cambio, si una cláusula conflicto es detectada y el nivel actual de decisión es 0, entonces devuelve *None*, declarando no existe una asignación válida para las variables, luego la fórmula es insatisfacible.

El método encargado de realizar la propagación unitaria por cada nivel de decisión es *unit_propagate(self)*. Este, cosiste en un algoritmo de punto fijo que se ejecuta mientras haya cambio, es decir, mientras existan cláusulas unitarias, y en caso de haber una cláusula conflicto detiene el bucle y devuelve dicha cláusula. En caso de no haber conflicto, retorna *None*. La salida de este algoritmo es tomada por el método

solve() explicado anteriormente. El status de una cláusula ('*satisfied*', '*conflict*', '*unit*', '*undefined*') es determinado por *check_clause(self, clause)*.

Una vez encontrada una cláusula de conflicto, se llama al método *conflict_analysis(self, conflict_clause)*, el cual primero analiza a cuántas variables en la cláusula conflicto se le asignaron valor en el nivel de decisión actual. Esto debido a que la cláusula aprendida solo debe contener un literal cuyo valor haya sido asignado en el nivel de decisión actual. En caso de haber más de 1, se busca el último de estos literales cuya variable fue asignada, se toma a partir del grafo de decisión la cláusula que causó su asignación, y se realiza Pincipio de Resolución con esta y la actual cláusula aprendida. La cláusula resultante para a ser la cláusula aprendida hasta el momento. Este prcedimiento se repite hasta que solo quede un literal asignado en el actual nivel de decisión.

Luego de obtener la cláusula aprendida a partir de un conflicto se determina el *backjump level*, el cual será el nivel de decisión más alto de los literales se la cláusula aprendida. Una vez hallado el nivel al cual retroceder, el método *backjump(self, level)* realiza las actualizaciones de todas las estructuras de la clase (*backjump*).

2.3. DLIS

La heurística Dynamic Largest Individual Sum (DLIS) para selección de variables, es uno de los métodos aproximados que pueden integrarse en un CDCL SAT *solver* con el objetivo de aumentar la eficiencia al asignar un valor a una variable en cada nivel de decisión.

DLIS lleva un contador por cada variable que indica el número máximo que cláusulas insatisfechas que pueden resolverse al asignar uno de los dos valores (0 o 1). Es decir, dada una variable x, se calcula la cantidad de cláusulas insatisfechas en las que aparece el literal x y su complementario $\neg x$. Sea $dlis(x)$ la mayor cantidad de cláusulas que x puede satisfacer, y $count_pos(x)$ y $count_neg(x)$ la cantidad de veces que aparece el literal positivo y negativo, repectivamente, en cláusulas aún sin resolver; luego:

$$dlis(x) = max(count_pos(x), count_neg(x))$$

Teniendo en cuenta este cálcula la próxima variable a asignar será:

$$x_k \mid dlis(x_k) = max(dlis(x_i)), i \in [1, n]$$

donde n es la cantidad de variables de la FNC.

Obsétvese que si $count_pos(x_k) > count_neg(x_k)$ entonces x_k tomará como valor 1, y 0 en caso contrario, puesto que el objetivo es satisfacer dichas cláusulas. Téngase en cuenta que el cálculo se le aplica a las variables que aún no han sido asignadas,

además de solo tenerse en cuenta aquellos valores que aún no han sido explorados para una misma variable en el actual nivel de decisión.

La estrategia de selección de variables se puede insertar en el anterior algoritmo de CDCL en el método *pick_branching_variable(self)* el cual se encarga de decidir la próxima variable a la cual se le asignará un valor (0 o 1). Tomando como base el código anterior y modificando el método *pick_branching_variable(self)*, una posible implementación de DLIS podría quedar de la siguiente forma:

```
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* almacenan la cantidad de veces que el literal positivo y el negativo, respectivamente, de una variable aparece en cláusulas aún sin satisfacer. Para actualizar estas estructuras primero se inspeccionan aquellas cláusulas que aún no han sido resueltas. Luego se completa la información con aquellas variables sin asignar que no se hayan incluido en el procedimiento anterior y se pone su contador en 0. Este puede ser el caso de variables sin asignar que solo se encuentren en cláusulas satisfechas (a otra variable de la misma cláusula se le asignó 1 como valor). Una vez actualizado el conteo por cada literal, se haya el *score* por vaiable (el máximo entre ambos conteos), se ordenan de mayor a menor y se decide asignar aquella variable con el *score* más alto. El valor a asignársele a esta variable es el de mayor conteo entre ambas polaridades.

Con esta estrategia DLIS busca satisfacer en una sola asignación la mayor cantidad de cláusulas posibles, sin embargo, esta estrategia resulta costosa en instancias grandes: $O(n)$ por cada nivel de decisión, donde n es la cantidad de literales.

2.4. VSIDS

Por su parte, la heurística Variable State Independent Decaying Sum (VSIDS) prioriza asignarle valores aquellas variables que hayan estado en conflictos recientes.

Para ello, VSIDS lleva un *score* por cada literal l (no por cada variable) que aumenta cada vez que aparezcan en clásulas aprendidas de conflictos. Además, para evitar que literales que hayan pertenecido a conflictos pasados y no recientes sean tenidos en cuenta por encima de los más actuales, cada cierta cantidad T de conflictos se multiplica los *scores* de cada literal por α, donde $0 < \alpha < 1$ (usualmente $\alpha = 0{,}95$). Integrado con CDCL, VSIDS se comportaría de la siguiente forma:

1. Procede el algoritmo CDCL.
2. Si ocurre un conflicto, se añade la cláusula aprendida $\mathbf{C_{learn}}$. Luego, por cada literal l tal que $l \in \mathbf{C_{learn}}$ se tiene que $score(l) += \delta$, donde δ es el incremento.
3. Si ocurre el conflicto T-ésimo, entonces $\delta = \delta \cdot \alpha$ con $0 < \alpha < 1$.
4. Si no ocurre un conflicto, se seleccionará según VSIDS la variable v si $score(l_v) = \text{máx}(score(l_i))$ para todo i tal que $1 \leq i \leq 2n$, con n cantidad de variables. Si l_v es positivo, entonces v tomará valor 1, y 0 en caso contrario.

Una posible implementación para VSIDS que se integre al código base anterior de CDCL es la siguiente.

```
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 solo selecciona entre las que no han sido asignadas, aquella variable con mayor *score* de acuerdo al criterio de este algoritmo. Para llevarlo a cabo, se

añadió a la clase *SATSolver* una estructura *activity* que mapea cada variable con su *score*. Esta estructura se inicializaría 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
```

Además, *activity* se pudiese actualizar dentro del método *solve(self)* de la siguiente forma:

```
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
```

2.5. Reinicio (*restart*)

Las estrategias de reinicio buscan no estancarse en espacios locales de búsqueda mediante un "reinicio" del árbol de decisión, es decir, eliminan todas las asignaciones realizadas hasta el momento y vuelven a empezar, pero manteniendo en la FNC las cláusulas aprendidas producto de CDCL, y los datos extras como *scores* de los literales que hayan aportado las heurísticas de selección de variables.

Existen varios criterios para realizar los *restarts*:

1. Fijo: se reinicia cada k conflictos fijos.

2. Geométrico: cada intervalo r_i crece como un secuencia geométrica de la siguiente forma: $r_0 = b; r_i = \alpha \cdot r_{i-1} \mid \alpha > 1$. Es importante definir bien el valor de α pues si este es muy grande los reinicios serán muy espaciados, y si es muy pequeño habrá una sobrecarga de *restarts*.

3. Luby: Este reinicio se basa en la secuencia de Luby (1,1,2,1,1,2,4,1,1,2,...), obteniéndose que $r_i = b \cdot Luby(i)$, donde r_i constituye el i-ésimo *restart*, y b es un parámetro de intervalo.

4. *Glucose-style (LBD-based)*: esta estrategia está basada en el cálculo Literal Block Distance (LBD) que consiste en, dada una cláusula aprendida, contar la cantidad de niveles de decisión diferentes a los que pertenecen cada uno de sus literales. Es decir, dada una cláusula aprendida C_{learm} se tiene que $LBD(C_{learn}) = \|\{level(l)\, l \in C_{learn}\}\|$. LBD busca medir la calidad de una cláusula a partir de la diversidad de niveles de decisión en una cláusula, planteando que mientras menor sea este número las variables implicadas en el conflicto estarán más cerca en el árbol de decisión, por ende más relacionadas, luego más útil la cláusula. Por tanto, a menor LBD, mayor calidad de cláusula. Ahora, la estrategia de reinicio *Glucose-style* haya dos promedios de LBD para una ventana "rápida" (usualmente 50 o 100 últimos conflictos) y una ventana "lenta" (usualmente 1000 últimos confilctos)de cantidad de conflictos. Estas dos cantidades se dividen de la siguiente forma: sean μ_r promedio de LBD en la ventana rápida de últimos conflictos, y μ_l su homólogo para la ventana lenta; sea, además, $T > 1$ umbral de decisión, entonces *Glucose-style* realiza la comparación $\dfrac{\mu_r}{\mu_l} > T$, y si esta resulta verdadera entonces procede con el reinicio. Esta estrategia sugiere que si el promedio de LBD en cláusulas aprendidas recientes supera significativamente al de cláusulas más antiguas, el algoritmo estaría estancado en espacios locales de búsqueda infructíferos.

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

```python
# 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))
```

Análogo a las heurísticas anteriores, se añaden nuevas estructuras a la clase

SATSolver, y en base a la estrategia de *restart* de Luby, se decide en el método *solve(self)* si es necesario un reinicio.

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 usadas actualmente es *Two Watched Literals (TWL)*. Esta tiene por objetivo evitar el recorrido de todas las cláusulas en el momento de realizar la propagación unitaria mediante la "vigilancia" de dos literales l_1 y l_2 por cada cláusula. Si durante el algoritmo tanto l_1 como l_2 no han sidp evaluados, pues no es necesario revisar dicha cláusula dado que no es unitaria ni de conflicto. En cambio, si alguno ha sido asignado a 0 se busca en cada cláusula otro literal para sustituirlo y, de no ser posible implicaría que la cláusula es unitaria (asumiendo que el otro literal vigilado no tiene valor). Si ambos literales son evaluados a 0, entonces la cláusula es de conflicto. Esta estrategia reduce grandemente el costo de recorrer cada cláusula en cada nivel de asignación.

Para insertar esta estrategia en el código base de CDCL, es necesario realizar cambios en casi todos los métodos. Véase el siguiente ejemplo de implementación:

```
import formulas as f
from collections import defaultdict, deque

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.clauses = [list(c) for c in 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
           vars).
        self.decision_level = 0  # Current decision level.
```

```
        self.decision_stack = [] # Stack of (variable,
            value, level).

        # Two-Watched Literals: map literal -> list of
            clause indices watching it
        self.watches = defaultdict(list)
        self._init_watches()

    def _init_watches(self):
        """Initialize two watched literals per clause."""
        for ci, clause in enumerate(self.clauses):
            # If clause has only one literal, watch it
                twice.
            w0 = clause[0]
            w1 = clause[1] if len(clause) > 1 else clause
                [0]
            self.watches[w0].append(ci)
            self.watches[w1].append(ci)

    def literal_value(self, literal):
        """
        Evaluate a literal under current partial
            assignment.
        Returns True, False, or None if unassigned.
        """
        var = abs(literal)
        if var not in self.assignments:
            return None
        return self.assignments[var] if literal > 0 else
            not self.assignments[var]

    def check_clause(self, clause):
        """
        Determine clause status: 'satisfied', 'conflict',
            'unit', or 'undefined'.
        If 'unit', also return the unit literal.
        """
        unassigned = 0
        last = None
        for lit in clause:
```

```
            val = self.literal_value(lit)
            if val is True:
                return ('satisfied', None)
            if val is None:
                unassigned += 1
                last = lit
        if unassigned == 0:
            return ('conflict', None)
        if unassigned == 1:
            return ('unit', last)
        return ('undefined', None)

    def _enqueue(self, var, value, level, reason):
        """
        Assign var=value at given level with reason and
           push onto decision stack.
        Returns the corresponding literal for propagation.
        """
        self.assignments[var] = value
        self.levels[var] = level
        self.reasons[var] = reason
        self.decision_stack.append((var, value, level))
        return var if value else -var

    def unit_propagate(self):
        """
        Perform unit propagation using two-watched
           literals.
        Returns a conflicting clause if conflict, else
           None.
        """
        queue = deque()
        # Enqueue all literals assigned at current level
        for var, val, lvl in self.decision_stack:
            if lvl == self.decision_level:
                queue.append(var if val else -var)

        while queue:
            lit = queue.popleft()
            lit_false = -lit
```

```
            # We iterate over a snapshot since watch list
               may change
            watchers = list(self.watches[lit_false])
            for ci in watchers:
                clause = self.clauses[ci]
                # Try to find a new literal to watch
                   instead of lit_false
                found_replacement = False
                for l in clause:
                    if l == lit_false:
                        continue
                    if self.literal_value(l) is not False:
                        # relocate watch from lit_false to
                            l
                        self.watches[l].append(ci)
                        self.watches[lit_false].remove(ci)
                        found_replacement = True
                        break
                if found_replacement:
                    continue

                # No replacement found: clause must be
                   unit or conflict
                status, unit_lit = self.check_clause(
                   clause)
                if status == 'conflict':
                    return clause
                elif status == 'unit':
                    v = abs(unit_lit)
                    if v not in self.assignments:
                        new_lit = self._enqueue(v,
                           unit_lit > 0, self.
                           decision_level, clause)
                        queue.append(new_lit)
        return None

    def pick_branching_variable(self):
        """
        Select next unassigned variable (naive).
        """
```

```
        all_vars = {abs(l) for c in self.clauses for l in
           c}
        for v in all_vars:
            if v not in self.assignments:
                return v
        return None

    def resolve(self, c1, c2, pivot):
        """
        Resolve two clauses on pivot literal.
        Returns the resolvent.
        """
        res = [l for l in c1 if l != pivot]
        for l in c2:
            if l != -pivot and l not in res:
                res.append(l)
        return res

    def conflict_analysis(self, conflict_clause):
        """
        First-UIP conflict analysis.
        Returns (learned_clause, backjump_level).
        """
        learned = conflict_clause.copy()
        cur_lvl = self.decision_level
        while True:
            # Count lits at current level
            lvl_lits = [l for l in learned if self.levels.
               get(abs(l), -1) == cur_lvl]
            if len(lvl_lits) <= 1:
                break
            # Find most recent one
            last = None
            for v,_,lvl in reversed(self.decision_stack):
                if lvl != cur_lvl:
                    continue
                for l in lvl_lits:
                    if abs(l) == v:
                        last = l
                        break
```

```
                if last:
                    break
            reason = self.reasons.get(abs(last))
            if not reason:
                break
            learned = self.resolve(learned, reason, last)

        # Compute backjump level
        back_lvl = 0
        for l in learned:
            lvl = self.levels.get(abs(l), 0)
            if lvl != cur_lvl and lvl > back_lvl:
                back_lvl = lvl
        return learned, back_lvl

    def backjump(self, level):
        """
        Undo assignments above given level.
        """
        new_stack = []
        for v, val, lvl in self.decision_stack:
            if lvl > level:
                self.assignments.pop(v, None)
                self.levels.pop(v, None)
                self.reasons.pop(v, None)
            else:
                new_stack.append((v, val, lvl))
        self.decision_stack = new_stack

    def solve(self):
        """
        Main CDCL loop.
        Returns a satisfying assignment or None if UNSAT.
        """
        while True:
            conflict = self.unit_propagate()
            if conflict:
                if self.decision_level == 0:
                    return None
                learned, bj = self.conflict_analysis(
```

```
                conflict)
            # add learned clause and set up its
                watches
            self.clauses.append(learned)
            ci = len(self.clauses) - 1
            w0 = learned[0]
            w1 = learned[1] if len(learned) > 1 else
                learned[0]
            self.watches[w0].append(ci)
            self.watches[w1].append(ci)
            # backjump and continue
            self.backjump(bj)
            self.decision_level = bj
        else:
            var = self.pick_branching_variable()
            if var is None:
                return self.assignments
            # make a new decision
            self.decision_level += 1
            lit = self._enqueue(var, True, self.
                decision_level, None)
```

En este código, de igual forma, se añaden las estrucutras necesarias y se realiza el procedimiento de acuerdo con la idea que plantea TWL.

Capítulo 3

Propuesta

Capítulo 4

Detalles de Implementación y Experimentos

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.

Recomendaciones

Recomendaciones