Universidad de La Habana
Facultad de Matemática y Computación

Análisis comparativo de métodos aproximados en solucionadores CDCL SAT

Autora:
Massiel Paz Otaño

Tutor:

Dr. Luciano García Garrido
Prof. Titular Consultante
Facultad de Matemática y Computación
Univarsidad de La Habana, Cuba

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

Esta tesis realiza un análisis comparativo de las heurísticas de decisión de variables VSIDS (Variable State Independent Decaying Sum) y DLIS (Dynamic Largest Individual Sum) en *solvers* SAT basados en *Conflict-Driven Clause Learning* (CDCL). El estudio examina ambas heurísticas con y sin estrategias de *restart*, utilizando como plataforma de evaluación el *solver* CaDiCaL.

Se implementó DLIS en CaDiCaL mediante modificaciones que permitieron su integración con la estructura existente del *solver* y sus otras heurísticas. Para la evaluación, se desarrolló un *benchmark* de 135 problemas SAT mediante un *script* Python, abarcando diversas categorías: *random* k-sat, *random biased* k-sat, *pingeonhole*, *graph coloring*, *parity* XOR y BCM flip flop. Estos problemas presentan variaciones en sus estructuras, incluyendo números de variables (desde decenas hasta miles), cantidad de cláusulas y densidad de restricciones.

Cada combinación heurística (VSIDS/DLIS, con/sin *restart*) se ejecutó sobre todos los problemas mediante *flags* específicos de CaDiCaL, estableciendo un *timeout* de 2 minutos por ejecución. Los resultados se almacenaron sistemáticamente en un CSV, registrando: nombre del problema, heurística empleada, resultado (SATISFIABLE/UNSATISFIABLE/TIMEOUT), tiempo de resolución, cantidad de variables, cantidad de cláusulas, densidad, entre otros.

El análisis estadístico reveló que el rendimiento depende más de la estructura intrínseca del problema que de la heurística específica o el uso de *restart*. No se observaron diferencias significativas entre DLIS con/sin *restart*, y VSIDS con/sin *restart*. Sin embargo, VSIDS exhibió mayor varianza en los tiempos de ejecución, mientras DLIS demostró ser más eficiente para problemas específicos como los de coloración de grafos y estructuras *XOR-parity*. Estos hallazgos sugieren que la selección óptima de heurísticas debería considerar las características estructurales de los problemas SAT a resolver.

Abstract

This Bachelor's Thesis conducts a comparative analysis of variable decision heuristics VSIDS (Variable State Independent Decaying Sum) and DLIS (Dynamic Largest Individual Sum) in Conflict-Driven Clause Learning (CDCL) SAT solvers. The study examines both heuristics with and without restart strategies, using the CaDiCaL solver as the evaluation platform.

DLIS was implemented in CaDiCaL through modifications that enabled its integration with the solver's existing structure and other heuristics. For evaluation, a benchmark of 135 SAT problems was developed using a Python script, covering diverse categories: random k-sat, random biased k-sat, pigeonhole principle, graph coloring, parity XOR, and BCM flip flop. These problems feature structural variations including variable counts (from tens to thousands), clause quantities, and constraint densities.

Each heuristic combination (VSIDS/DLIS, with/without restart) was executed on all problems using specific CaDiCaL flags, with a 2-minute timeout per execution. Results were systematically stored in a CSV file, recording: problem name, heuristic used, outcome (SATISFIABLE/UNSATISFIABLE/TIMEOUT), resolution time, variable count, clause count, density and others.

Statistical analysis revealed that performance depends more on the intrinsic structure of the problem than on the specific heuristic or restart usage. No significant differences were observed between DLIS with/without restart, and VSIDS with/without restart. However, VSIDS exhibited greater variance in execution times, while DLIS proved more efficient for specific problems such as graph coloring and XOR-parity structures. These findings suggest that optimal heuristic selection should consider the structural characteristics of target SAT problems.

Índice general

Introducción **1**

1. Marco Teórico **5**
1.1. Fundamentos de los Problemas de Satisfacción de Restricciones y SAT 5
1.1.1. SAT como Caso Especial de CSP y su NP-Completitud 5
1.2. Evolución de los SAT *solvers* 6
1.2.1. Principio de Resolución 6
1.2.2. Algoritmo Davis–Putnam (DP) 7
1.2.3. Algoritmo Davis–Putnam–Logemann–Loveland (DPLL) 9
1.2.4. *Conflict-Driven-Clause-Learning* (CDCL) 11
1.3. Heurísticas 14
1.3.1. *Dynamic Largest Individual Sum* (DLIS) 14
1.3.2. *Variable State Independent Decaying Sum* (VSIDS) 17
1.3.3. Reinicio (*restart*) 19
1.3.4. *Two Watched Literals (TWL)* 23
1.4. CaDiCaL 26
1.4.1. Ramificación (selección de variables) 26
1.4.2. Políticas de reinicio 26
1.4.3. *Assumptions* 27
1.4.4. *Phasing* 28
1.4.5. Modularidad y Extensibilidad del Código de CaDiCaL 29
1.4.6. Ventajas de CaDiCaL para Experimentación 29
1.5. Parámetros de Evaluación para *solvers* CDCL: Taxonomía de extit-Benchmarks 30
1.5.1. Categorías 30
1.5.2. Clasificación por Satisfacibilidad y Propiedades Estructurales . 30
1.5.3. Densidad y Transición de Fase 31
1.5.4. Relevancia para la Evaluación de Heurísticas 31
1.6. Problemas 31
1.6.1. Random k-SAT 31

1.6.2. Problema del palomar 32
1.6.3. Problema de coloreo de grafos 33
1.6.4. XOR 33
1.6.5. *Bounded Model Checking* (BCM) 33
1.7. Insuficiencias Fundamentales de SAT 34

2. Detalles de Implementación y Experimentos **36**
2.0.1. CaDiCaL 36
2.0.2. Empleo de *flags* en la l/'inea de comandos 41
2.0.3. Problemas 41
2.0.4. Generador de problemas 43
2.0.5. Estad/'isticas 43
2.1. Resultados 44
2.1.1. Análisis de los problemas con TIMEOUT 44
2.1.2. Análisis de problemas resueltos (SATISFIABLE/UNSATISFIABLE) 49
2.1.3. Porblemas RESUELTOS vs TIMEOUT 51
2.1.4. Rendimiento de las heurísticas en problemas resueltos 55

Conclusiones **61**

Recomendaciones **62**

Bibliografía **63**

Índice de figuras

1.1. Posible espacio de búsqueda de una FNC 9
1.2. Grafo de conflito 12
1.3. Corte que representa el esquema de aprendizaje 1-UIP 13
1.4. Fenómeno de cola pesada en DPLL SAT *solvers* 20
1.5. Efecto de aplicar *restarts* 21
1.6. Secuencia geométrica de reinicio interna-externa 22
1.7. Secuencia de reinicio de basada en Luby con 512 inicial. 22

2.1. Distribución de número de variables en problemas con timeout por heurística. 45
2.2. Distribución de número de cláusulas en problemas con timeout por heurística. 46
2.3. Distribución de densidad en problemas con timeout por heurística. . . 46
2.4. Distribución de tamaño promedio de cláusula en problemas con timeout por heurística. 47
2.5. Distribución de variables positivas en problemas con timeout por heurística. 47
2.6. Distribución de variables negativas en problemas con timeout por heurística. 48
2.7. Distribución de timeouts por heurística. 48
2.8. Distribución de timeouts por problema. 49
2.9. Distribución de características y tiempo por heurística. 50
2.10. Matriz de correlación - Problemas resueltos vs TIMEOUTS. 51
2.11. Matriz de correlación tras eliminar multicolinealidad. 52
2.12. Test de normalidad Shapiro-Wilk. 53
2.13. Prueba Mann-Whitney U. 53
2.14. Distribuciones de prueba Mann-Whitney U por resultados. 54
2.15. Regresión logística. 54
2.16. Correlación de Spearman. 55
2.17. Kruskal-Wallis. 56
2.18. Test Dunn por cuartiles para VSIDS con restart. 57

2.19. Test Dunn por cuartiles para VSIDS sin restart. 58
2.20. Test Dunn por cuartiles para DLIS con restart. 59
2.21. Test Dunn por cuartiles para DLIS csin restart. 60

Ejemplos de código

1.1. Algoritmo CDCL 13
1.2. Heurística DLIS 15
1.3. Función decide() con DLIS 15
1.4. Algoritmo VSIDS 17
1.5. Función decide() con VSIDS 18
1.6. Algoritmo reinicio geométrico 21
1.7. Algoritmo CDCL con reinicio 23
1.8. Algoritmo TWL 24
1.9. Algoritmo CDCL con TWL 25
2.1. Integración de DLIS en CaDiCaL. Internal.hpp 37
2.2. Integración de DLIS en CaDiCaL. Internal.cpp 37
2.3. Integración de DLIS en CaDiCaL. Calcular *score*. Internal.cpp 38
2.4. Integración de DLIS en CaDiCaL. decide.cpp 39
2.5. Integración de DLIS en CaDiCaL. options.hpp 41

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 *solvers*, su eficiencia se veía limitada por la explosión combinatoria en instancias complejas. La búsqueda y asignación secuencial de cláusulas unitarias, la falta de una estrategia de selección de variables y el retroceso (*backtrack*) cronológico exponían claras debilidades, especialmente en problemas con miles de variables.

A pesar de los avances, los SAT *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 *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 secuencial, o que no tenga en cuenta el posible impacto posterior a su asignación, 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 solucionador 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.

En los CDCL SAT *solvers* se ha observado que el tiempo de ejecución puede seguir una distribución de "cola pesada" (*heavy-tailed distribution*), lo que significa que el solucionador puede quedarse atascado en un camino de búsqueda improductivo por un tiempo prolongado. En aras de resolver este problema, surge la estrategia *restart*, la cual borra parte del estado del *solver* a intervalos determinados durante su ejecución. Su principal objetivo es reorientar la búsqueda y aprovechar el conocimiento acumulado mientras se evita profundizar en regiones improductivas del árbol de búsqueda. Al reiniciar, el solucionador puede escapar de una dirección de búsqueda desventajosa y tener una "segunda oportunidad" para encontrar una solución más rápidamente.

Hoy, aunque los SAT solucionadores 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, alternando entre el uso de *restart* dentro del entorno CDCL que ofrece el solucionador CaDiCaL, 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 características de los problemas. Además, se propone un marco teórico amplio para comprender la evolución algorítmica de los CDCL SAT *solvers*, entender algunas de las heurísticas que se emplean en los solucionadores modernos, específicamente en CaDiCaL, y 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 *solvers*, optimizando recursos en áreas como la verificación de *software* o la logística, donde minutos de mejora equivalen a ahorros millonarios.

Como problema científico se plantea la ineficiencia de los SAT solucionadores ante problemas con distintas estructuras, asociada a la selección subóptima de variables, influenciada o no por técnicas de reinicio. El objeto de estudio se centrará en algoritmos CDCL con estrategis VSIDS, DLIS y *restart*. Esta tesis tiene como objetivos:

- Analizar el impacto de VSIDS y DLIS, con y sin reinicio, en el rendimiento de CDCL.
- Establecer correlaciones entre tipos de problemas y heurísticas.
- Establecer correlaiones entre tipos de problemas y resultados de las heurísticas.

El campo de acción de esta tesis versa sobre la Lógica computacional aplicada a la resolución de problemas con restricciones. Como hipótesis se plantea que: El rendimiento de VSIDS y DLIS con y sin *restart* varía significativamente según la densidad de restricciones, el tamaño promedio de cláusula y la cantidad de variables. 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.

El documento se organiza en cinco capítulos:

- **Capítulo 1** Revisión teórica de los principales algoritmos usados en los solucionadores SAT, algunas heurísticas empleadas en los *solvers* modernos haciendo énfasis en CaDiCaL, y de las categorías de problemas usadas en los experimentos que se llevaron a cabo.
- **Capítulo 2** Detalles de implementación de las heurísticas en CaDiCaL, del generador de problemas y de los análisis estadísticos empleados. Explicación detallada de los algoritmos empleados y exposición de bibliotecas usadas.

- **Capítulo 3** Resultados de los experimentos.
- **Capítulo 4** Conclusiones y recomendaciones.

Capítulo 1

Marco Teórico

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

Los Problemas de Satisfacción de Restricciones (*Constraint Satisfaction Problems* CSP) constituyen un paradigma esencial para la modelación de problemas combinatorios en inteligencia artificial, investigación de operaciones y ciencia 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 Garrido 2024. 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 AlmaBetter 2025.

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). 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 Zulkoski 2018. 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 Marques-Silva et al. 2024. 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; 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 Guo 2024. Si bien en la práctica los solucionadores actuales logran resolver instancias que contienen millones de variables, en el peor de los casos SAT presenta una complejidad exponencial intrínseca Marques-Silva et al. 2024.

1.2. Evolución de los SAT *solvers*

La evolución de los solucionadores SAT constituye un hito en la ciencia de la computación, pues convierte un problema teóricamente intratable en una herramienta práctica de amplio uso industrial Fichte et al. 2023. Este desarrollo se apoya en tres pilares algorítmicos: el Principio de Resolución, los métodos Davis-Putnam y Davis-Putnam-Logemann-Loveland, y la revolución moderna impulsada por los solucionadores de aprendizaje de cláusulas dirigido por conflictos (*Conflict Driven-Clause Learning* CDCL). A continuación, se presenta un análisis detallado de cada uno.

1.2.1. Principio de Resolución

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

Dada una fórmula en Forma Normal Conjuntiva, cada una de sus cláusulas se representa como un conjunto de literales, y a su vez la fórmula se prepresenta como un conjunto de conjuntos Garrido 2025. Por esta razón no hay elementos repetidos dentro de una cláusula ni en la fórmula completa. Tomando como entrada esta representación, PR identifica pares de cláusulas que contienen literales complementarios (q y $\neg q$) y genera una nueva cláusula al realizar la operaciń de unión entre ellas y eliminando de cada una el literal y su opuesto Garrido 2025.

Concretamente, si $\mathbf{B}$ y $\mathbf{C}$ son cláusulas de la FNC $\mathbf{A}$ 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, $\mathbf{B}$ y $\mathbf{C}$ actúan como cláusulas padres o premisas, mientras que $\mathbf{D}$ 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 Garrido 2025.

Resolución Unitaria (RU)

Una instancia particular de 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 Garrido 2025.

El proceso consiste en identificar una cláusula unitaria y aplicar PR sobre otra cláusula que contenga el literal complementario, eliminándolo y generando una nueva cláusula más restringida. Por ejemplo Garrido 2025:

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

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 DPLL1.2.3 y CDCL1.2.4, donde se utiliza para propagar restricciones a lo largo del proceso de asignación.

El Principio de Resolución, aunque es completo para fórmulas en FNC, su aplicación directa resulta impráctica debido al crecimiento exponencial en el número de cláusulas generadas Garrido 2025. Sin embargo, su relevancia conceptual fundamenta y guía el diseño de métodos más eficientes.

1.2.2. Algoritmo 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

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

Principio de Resolución. Este algoritmo implementa tres procedimientos fundamentales: la Propagación Unitaria (PU), la Eliminación de Literales Puros (ELP) y la Resolución Basada en División (RD) Garrido 2025.

La Propagación Unitaria identifica cláusulas unitarias dentro de la FNC y procede a asignar forzosamente el valor correspondiente al literal involucrado. Seguidamente, elimina de la fórmula todas las cláusulas satisfechas por dicha asignación, y suprime el literal complementario en aquellas donde aparezca. Por su parte, la Eliminación de Literales Puros detecta literales cuya polaridad es única en toda la fórmula[2] y elimina las cláusulas en las que aparezcan. Este procedimiento persigue la idea de que las cláusulas con literales puros 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 realizadas estas simplificaciones, DP procede con la Resolución Basada en División, que consiste en seleccionar una variable, asignarle un valor (0 o 1) y continuar la resolución de forma recursiva a partir de la nueva fórmula. Esta estrategia permite explorar sistemáticamente el espacio de soluciones posibles hasta determinar si la fórmula es satisfacible o no Garrido 2025.

Véase el siguiente ejemplo Garrido 2025 para una mejor comprensión.

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 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 $\{s\}$:

[2]se dice que q es un literal puro en la FNC A si q ocurre en A y $\neg q$ no

{}

Dado que la fórmula ha sido completamente reducida sin generar contradicciones, se concluye que la instancia es satisfacible. (citar libro de Luciano de L'ogica Proposicional)

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.

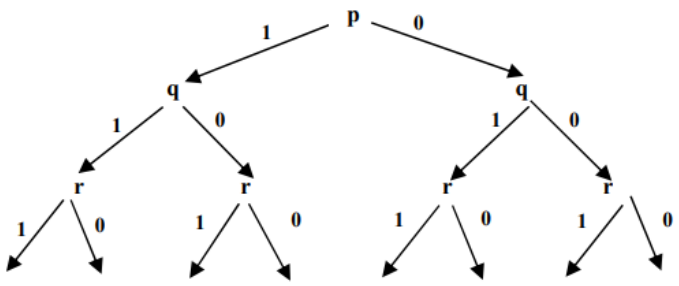

Figura 1.1: Posible espacio de búsqueda de una FNC
Garrido 2025

1.2.3. Algoritmo Davis–Putnam–Logemann–Loveland (DPLL)

El algoritmo Davis-Putnam-Logemann-Loveland (DPLL) constituye una mejora significativa del método 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 le permite deshacer asignaciones al regresar al nivel anterior de decisión una vez detectada una"cláusula de conflicto" [3]. Este enfoque permite explorar el árbol de búsqueda de forma más eficiente, reduciendo la necesidad de almacenar todas las ramas posibles Garrido 2025.

DPLL adopta una estrategia de generación "lazy" del árbol de asignaciones: antes de realizar una nueva ramificación, verifica mediante propagación unitaria si existen

[3]Se denomina cláusula de conflicto a aquella en la que todos sus literales fueron evaluados como falsos bajo una asignación parcial.

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 la variable de este nivel 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 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[4]. 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 Garrido 2025.

Obsérvese el siguiente ejemplo Garrido 2025:

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

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

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

$$\emptyset$$

Luego, la FNC es satisfacible Garrido 2025.

[4]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. En otras palabras, C es una cláusula redundante.

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.

1.2.4. *Conflict-Driven-Clause-Learning* (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.

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 pues además de analizar casos innecesarios, se vuelve susceptible a cometer el mismo error en el futuro dado que, potencialmente, realizará la misma combinación de asignaciones, lo cual genera 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. Igualmente, se almacenan los siguientes datos: 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. En la siguiente figura se muestra un ejemplo de un grafo de conflicto:

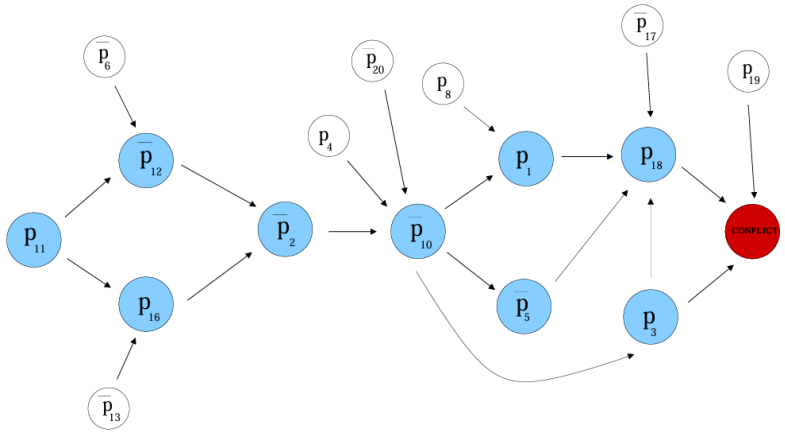

Figura 1.2: Grafo de conflito
Oliveras y Rodríguez-Carbonell 2009

Cuando una cláusula resulta ser de conflicto, CDCL crea un nuevo nodo en el grafo que representa dicho conflicto para comenzar con su análisis. Este, 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 aprendida" que represente la combinación de asignación de valores que condujo a dicho conflicto, y la añade al conjunto de cláusulas. De esta forma, evita 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. La figura 1.3 muestra el ejemplo del corte que representa el esquema de aprendizaje 1-UIP, en el grafo de aprendizaje:

En este caso, la cláusula aprendida es:

$$\neg p_{19} \vee p_{17} \vee \neg p_8 \vee p_{10}$$

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 PR 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.

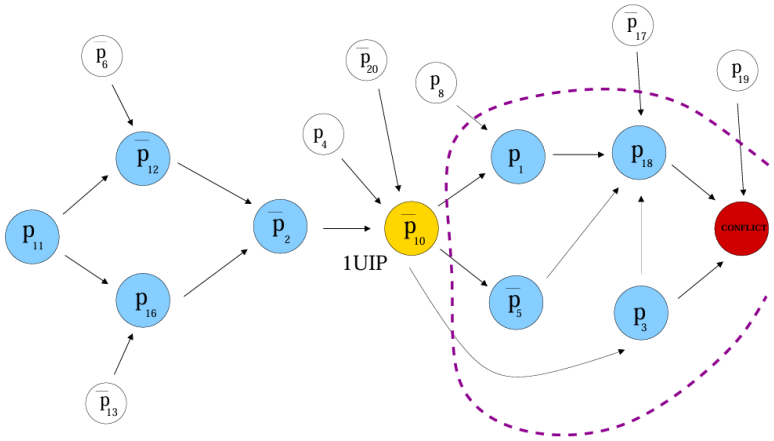

Figura 1.3: Corte que representa el esquema de aprendizaje 1-UIP
Oliveras y Rodríguez-Carbonell 2009

En el código 1.1 Oliveras y Rodríguez-Carbonell 2009 muestra un ejemplo de implementación CDCL en C++:

```
while(true){
        while (propagate_gives_conflict()){
                if (decision_level==0) return UNSAT;
                else analyze_conflict();
        }
        remove_lemmas_if_applicable();
        if (!decide()) returns SAT; // All vars assigned
}
```

Ejemplo de código 1.1: Algoritmo CDCL

El algoritmo 1.1 sigue el siguiente flujo:

1. Entra en un bucle principal infinito, que solo termina al encontrar una solución SAT o UNSAT.
2. Dentro del bucle, ejecuta la propagación de asignaciones de variables.
3. Si la propagación produce un conflicto, entra en un bucle interno:
   - Si el nivel de decisión es cero, devuelve UNSAT.
   - En caso contrario, analiza el conflicto para aprender una nueva cláusula y retroceder en el árbol de búsqueda (backtrack).

4. Después de resolver los conflictos, elimina lemmas (cláusulas aprendidas) si es necesario para gestionar la memoria.

5. Intenta tomar una nueva decisión (asignar una variable).

6. Si no quedan variables por decidir, devuelve SAT.

CDCL resuelve los problemas de búsquedas redundantes y *bactrack* cronológico de DPLL, sin embargo, aún deja pendientes por resolver problemas como la selección de variables.

1.3. Heurísticas

1.3.1. *Dynamic Largest Individual Sum* (DLIS)

La heurística *Dynamic Largest Individual Sum* es un método aproximado de selección de variables que busca hacer eficiente la selección de variables en un CDCL SAT *solver*. Para ello, DLIS considera que la próxima variable a asignar será aquella cuyo literal (el de mayor valor) tenga la mayor frecuencia de aparición en cláusulas insatisfechas. 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). Denotemos:

- $count\_pos(x)$: 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. El cálculo debe aplicarse solo a las variables no asignadas y solo considerar valores aún no explorados en el nivel actual de decisión.

Esta heurística busca satisfacer la mayor cantidad de cláusulas posibles en un mismo nivel de decisión.

Una posible implementación de DLIS sería como se muestra en 1.2

```
// Asumiendo que:
// - var_to_pos_clauses: map<var, int> con el n\'umero de
   cl\'ausulas no resueltas donde aparece la variable
   positiva
// - var_to_neg_clauses: map<var, int> con el n\'umero de
   cl\'ausulas no resueltas donde aparece la variable
   negativa

pair<var, bool> DLIS() {
    var best_var = 0;
    bool value = false;
    int max_pos = 0, max_neg = 0;

    // Encuentra la variable con mayor aparici\'on
       positiva y negativa
    for (const auto& [v, cnt] : var_to_pos_clauses) {
        if (cnt > max_pos) { max_pos = cnt; best_var = v;
           value = true; }
    }
    for (const auto& [v, cnt] : var_to_neg_clauses) {
        if (cnt > max_neg) { max_neg = cnt; }
        if (cnt > max_pos) { best_var = v; value = false;
           }
    }

    return {best_var, value};
}
```

Ejemplo de código 1.2: Heurística DLIS

La cual puede integrarse a la función `decide()` en 1.1 como se muestra en 1.3

```
bool decide() {
    auto [var, val] = DLIS();
    if (var == 0) return false; // No quedan variables por
        decidir
    take_decision(var, val); // Asigna var=val y aumenta
       decision_level
    return true;
}
```

Ejemplo de código 1.3: Función decide() con DLIS

En el código 1.2 se aprecia el siguiente flujo de decisiones:

- **Conteo de apariciones:** Para cada variable, se cuenta cuántas veces aparece positiva y negativamente en cláusulas no resueltas.
- **Selección:** Se elige la variable cuyo literal (positivo o negativo) aparece en más cláusulas no resueltas.
  - Si el literal positivo aparece en más cláusulas, se elige esa variable y se asigna verdadero.
  - Si el literal negativo aparece más, se elige esa variable y se asigna falso.
- **Ejemplo:** Si la variable x aparece positivamente en 5 cláusulas no resueltas y negativamente en 3, se asignará $x = \text{true}$; si la variable y aparece negativamente en 7 cláusulas y positivamente en 2, se asignará $y = \text{false}$.

Por su parte, en 1.3

- **Llama a DLIS:** Obtiene la variable y el valor sugeridos por la heurística DLIS.
- **Verifica si quedan variables:** Si no quedan variables por decidir, devuelve `false` (para indicar que todas están asignadas).
- **Asigna la variable:** Si hay variables, asigna el valor sugerido y aumenta el nivel de decisión.

Luego, el flujo completo con del código con la integración de DLIS sería:

- **DLIS** selecciona la variable y el valor que maximizan la aparición en cláusulas no resueltas.
- **CDCL** sigue su flujo normal, pero la función `decide()` ahora utiliza DLIS para tomar decisiones.
- **Integración:** Cada vez que se necesita una nueva decisión, se llama a DLIS para elegir la mejor opción, integrando así la heurística en el núcleo del algoritmo CDCL.

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.

1.3.2. *Variable State Independent Decaying Sum* (VSIDS)

La heurística de selección de variables que plantea *Variable State Independent Decaying Sum* (VSIDS), prioriza asignar valores a aquellas variables que hayan estado involucrados en conflictos recientes. Para ello, VSIDS mantiene un *score* por variable que se incrementa cada vez que esta aparece en cláusulas aprendidas a partir de conflictos. Además, para evitar que variables involucradas en conflictos antiguos tengan mayor prioridad que los relacionados con conflictos recientes, cada cierta cantidad T de conflictos se actualizan los *scores* de todas las variables multiplicándolas por un factor de "decaimiento" α, con $0 < \alpha < 1$ (usualmente $\alpha = 0{,}95$).

Una posible implementación de VSIDS sería como se muestra en 1.4:

```
vector<double> vsids_scores; // Inicializada al inicio con
    un valor para cada variable

void bump_vsids_score(var v) {
    vsids_scores[v] += 1.0;
}

void decay_vsids_scores() {
    for (auto &score : vsids_scores) {
        score *= 0.95; // Factor de decaimiento com\'un
    }
}

pair<var, bool> VSIDS() {
    var best_var = 0;
    double max_score = 0.0;
    for (var v = 1; v < vsids_scores.size(); ++v) {
        if (is_unassigned(v) && vsids_scores[v] >
           max_score) {
            max_score = vsids_scores[v];
            best_var = v;
        }
    }
    // VSIDS solo elige la variable, el valor puede
       elegirse aleatoriamente o con otra heur\'istica
    return {best_var, rand() % 2 == 0}; // Ejemplo: valor
       aleatorio
}
```

Ejemplo de código 1.4: Algoritmo VSIDS

Por su parte, para poder integrar esta heurística en 1.1, la función `decide()` se adaptaría como se muestra en :

```
bool decide() {
    auto [var, val] = VSIDS();
    if (var == 0) return false; // No quedan variables por
        decidir
    take_decision(var, val);    // Asigna var=val y
       aumenta decision_level
    return true;
}   return {best_var, rand() % 2 == 0}; // Ejemplo: valor
   aleatorio
```

Ejemplo de código 1.5: Función decide() con VSIDS

En 1.4, la heurística VSIDS mantiene un contador para cada variable, que se incrementa cada vez que la variable aparece en una cláusula aprendida (conflicto). Estos contadores se multiplican periódicamente por un factor de decaimiento (por ejemplo, 0.95) para dar más peso a los conflictos recientes.

- **Inicialización:** Cada variable tiene un contador de puntuación VSIDS.
- **Incremento de puntuación:** Cada vez que se aprende una cláusula, se incrementa la puntuación de las variables involucradas.
- **Decaimiento periódico:** Cada cierto número de decisiones o conflictos, todas las puntuaciones se multiplican por un factor menor que 1.
- **Selección de variable:** Cuando se necesita decidir, se elige la variable no asignada con mayor puntuación VSIDS.

Por su parte, en **??** la función `decide()` ahora utiliza la heurística VSIDS para elegir la próxima variable a decidir:

- Llama a `VSIDS()` para obtener la variable y el valor sugeridos.
- Si no quedan variables por decidir, devuelve `false`.
- Si hay variables, asigna el valor sugerido y aumenta el nivel de decisión.

Luego, el flujo completo quedaría:

- **VSIDS** selecciona la variable no asignada con mayor puntuación, priorizando variables implicadas en conflictos recientes.

- **La función `decide()`** ahora utiliza VSIDS para tomar decisiones.
- **Integración:** Cada vez que se aprende una cláusula, se actualizan las puntuaciones VSIDS, y periódicamente se aplica el decaimiento. El flujo principal del CDCL sigue igual, pero la heurística de decisión es VSIDS.

Esta heurística está entre las más usadas en los CDCL SAT *solvers* modernos.

1.3.3. Reinicio (*restart*)

Las estrategias de *restart* 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 las variables 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.

Los solucionadores de SAT basados en DPLL presentan un fenómeno de cola pesada Oliveras y Rodríguez-Carbonell 2009 1.4:

Estudios han demostrado que el uso de estrategias de reinicio ayudan significativamente a disminuir este fenómeno de cola pesada, como se muestra en 1.5:

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. En la figura 1.6 se muestra la secuencia geométrica interna-externa
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. En la figura 1.7 se muestra la secuencia de reinicio basada en Luby con 512 inicial:

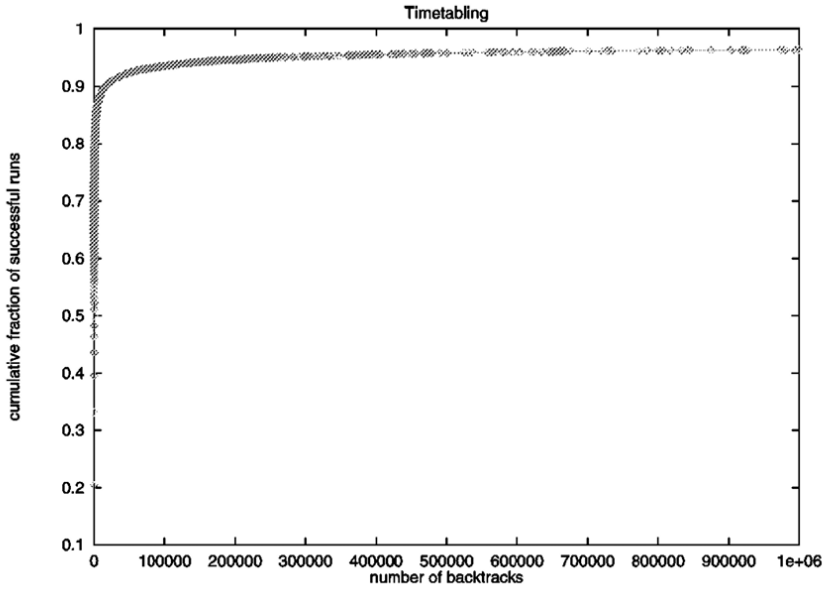

Figura 1.4: Fenómeno de cola pesada en DPLL SAT *solvers*
Oliveras y Rodríguez-Carbonell 2009

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 solucio-

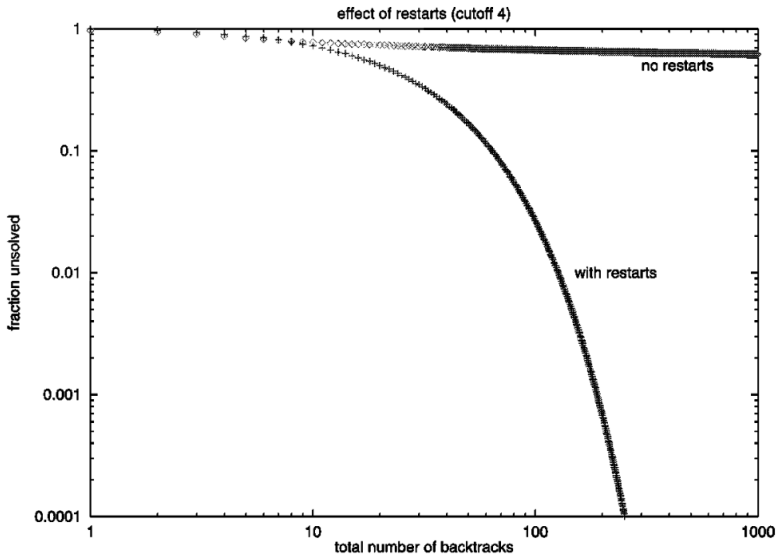

Figura 1.5: Efecto de aplicar *restarts*
Oliveras y Rodríguez-Carbonell 2009

nadores modernos de SAT, ya que permiten alternar entre exploración y explotación de manera eficiente en espacios de búsqueda complejos.

Una posible implementación de la estrategia de reinicio geométrico sería :

```
int inner = 100, outer = 100;
        for (;;){
        // Run SAT-solver for 'inner' conflicts
        if (inner >= outer){
                outer *= 1.1;
                inner = 100;
        }
        else
                inner*=1.1
        }
```

Ejemplo de código 1.6: Algoritmo reinicio geométrico

En 1.6 Oliveras y Rodríguez-Carbonell 2009:

- **Inicialización:** Se declaran e inicializan dos variables, `inner` y `outer`, ambas con valor 100.
- **Bucle infinito:** El bucle `for (;;)` se ejecuta indefinidamente.

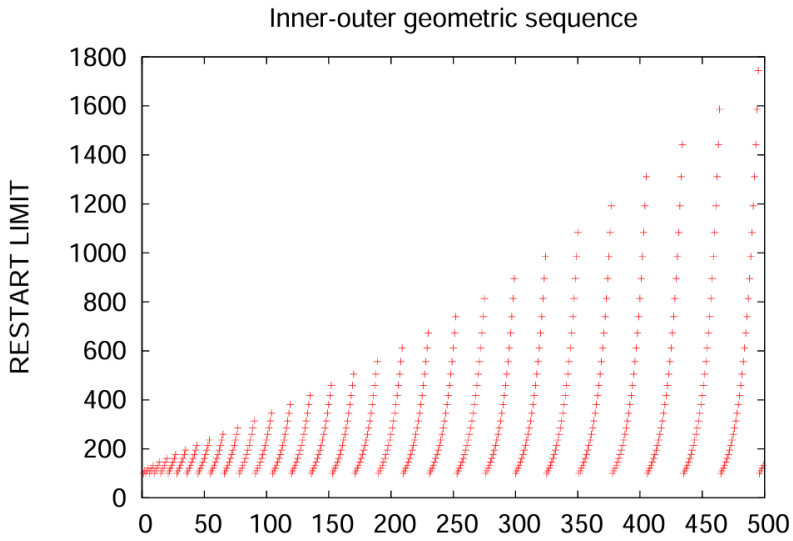

Figura 1.6: Secuencia geométrica de reinicio interna-externa
Oliveras y Rodríguez-Carbonell 2009

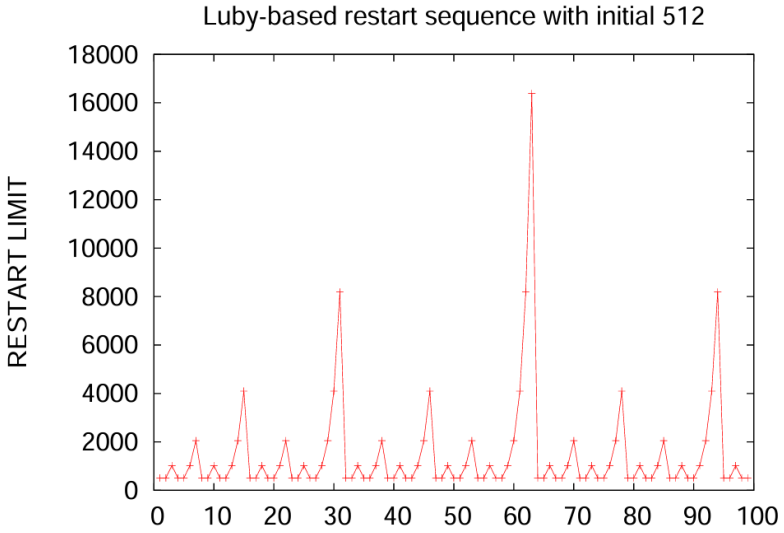

Figura 1.7: Secuencia de reinicio de basada en Luby con 512 inicial.
Oliveras y Rodríguez-Carbonell 2009

- **Ejecución del solucionador SAT:** En cada iteración, se ejecuta el solucionador SAT hasta que ocurran `inner` conflictos (el comentario indica la intención, aunque el código real del solucionador no está presente).

- **Comparación de `inner` y `outer`:**
  - Si `inner` es mayor o igual que `outer`, se actualiza `outer` multiplicándolo por 1.1 y se reinicia `inner` a 100.
  - Si `inner` es menor que `outer`, se multiplica `inner` por 1.1.
- **Objetivo:** El propósito de este esquema es permitir que el solucionador SAT ejecute bloques cada vez más grandes de conflictos (`inner`), pero si `inner` alcanza o supera a `outer`, se reinicia `inner` y se incrementa `outer`, haciendo que los bloques de conflictos permitidos crezcan de manera exponencial y controlada.

El código ajusta dinámicamente el tamaño del bloque de conflictos para el solucionador SAT. Cada vez que `inner` alcanza el valor de `outer`, este último se incrementa y `inner` se reinicia, permitiendo así que el solucionador explore bloques de conflictos cada vez más grandes, pero manteniendo un crecimiento controlado mediante el factor 1.1.

Para incorporar alguna estrategia de *restart* a 1.1 se puede proceder como muestra 1.7 Oliveras y Rodríguez-Carbonell 2009

```
while(true){
        while (propagate_gives_conflict()){
                if (decision_level==0) return UNSAT;
                else analyze_conflict();
        }
        restart_if_applicable();
        remove_lemmas_if_applicable();
        if (!decide()) returns SAT; // All vars assigned
}
```

Ejemplo de código 1.7: Algoritmo CDCL con reinicio

A pesar de los avances, la separación teórica entre *solvers* CDCL con y sin reinicios continúa siendo una pregunta abierta relevante, dado que muchos resultados dependen intrínsecamente de los reinicios Zulkoski 2018.

1.3.4. *Two Watched Literals (TWL)*

Una de las estrategias más empleadas en solucionadores modernos de SAT para optimizar la propagación unitaria es *Two 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.

Una posible implementación de esta estrategia sería la que se muestra en 1.8.

```
// Estructura para 2WL
struct Clause {
    vector<Lit> lits;
    Lit watch1, watch2;
    // ... otros campos
};

vector<Clause> clauses;
vector<vector<Clause*>> watch_list; // watch_list[var] =
   lista de cl\'ausulas donde var es watched

bool propagate_gives_conflict() {
    while (hay variables en la cola de propagaci\'on) {
        Lit lit = sacar de la cola;
        // Para cada cl\'ausula que vigila ~lit
        for (Clause* cl : watch_list[var(~lit)]) {
            // Intentar encontrar otro literal no asignado
                a falso para vigilar
            Lit other = (cl->watch1 == ~lit) ? cl->watch2
               : cl->watch1;
            if (valor(other) != FALSE) continue; // OK,
               sigue vigila otro literal
            // Buscar otro literal no asignado a falso
            bool found = false;
            for (Lit l : cl->lits) {
                if (l != cl->watch1 && l != cl->watch2 &&
                   valor(l) != FALSE) {
                    // Actualizar watched literal
                    if (cl->watch1 == ~lit) cl->watch1 = l
```

```
                        ;
                    else cl->watch2 = l;
                    watch_list[var(l)].push_back(cl);
                    found = true;
                    break;
                }
            }
            if (!found) {
                // Todos los literales excepto uno son
                   falsos: propagar el restante
                // Si ya est\'a asignado a falso, hay
                   conflicto
                // Si no, asignar el literal restante y
                   meter en la cola
                // ...
                if (/* conflicto */) return true;
            }
        }
    }
    return false;
}
```

Ejemplo de código 1.8: Algoritmo TWL

En la implementación 1.8:

- Mantiene, para cada cláusula, dos literales vigilados.
- Cuando un literal vigilado es asignado a falso, busca otro literal no asignado a falso para vigilar.
- Si no lo encuentra, propaga el literal restante o detecta un conflicto.
- Solo revisa las cláusulas afectadas por la asignación actual, lo que hace la propagación mucho más eficiente.

Luego, al integrar esta estrategia en 1.1, quedaría como se muestra en 1.9

```
while(true){
    while (propagate_gives_conflict()){ // Ahora con 2WL
        if (decision_level==0) return UNSAT;
        else analyze_conflict();
    }
    remove_lemmas_if_applicable();
```

```
    if (!decide()) return SAT;
}
```

Ejemplo de código 1.9: Algoritmo CDCL con TWL

Como se puede apreciar en 1.9, el algoritmo CDCL original se mantiene, pero la función de propagación `propagate_gives_conflict()` se implementa utilizando la estrategia *Two Watched Literals*, que permite propagar asignaciones de manera más eficiente.

La estrategia de los dos literales vigilados reduce notablemente la frecuencia con que se visita cada cláusula, ya que solo se considera una cláusula cuando uno de sus literales vigilados es asignado a falso. Durante el retroceso, no es necesario realizar ninguna acción adicional sobre las cláusulas, lo que simplifica el proceso. Además, los literales inactivos suelen permanecer como vigilados, lo que disminuye aún más el número de cláusulas que requieren revisión. Esta técnica resulta especialmente efectiva para cláusulas largas, como los lemas aprendidos. Para las cláusulas binarias, en cambio, se utilizan estructuras de datos especializadas que permiten un manejo aún más eficiente Oliveras y Rodríguez-Carbonell 2009.

1.4. 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; 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.

1.4.1. Ramificación (selección de variables)

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 Iser y Balyo 2021, Cherif et al. 2021.

1.4.2. Políticas de reinicio

CaDiCaL implementa políticas de reinicio dinámicas que se ajustan al comportamiento interno del *solver* Biere y Fröhlich 2019, Zhang et al. 2024.

Estos reinicios, permiten al solucionador 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 bloquean reinicios cuando la búsqueda se encuentra próxima a una solución potencial, siguiendo estrategias propuestas en la literatura. Cherif et al. 2021, Biere y Fröhlich 2019, Zhang et al. 2024.

Todas las *restart policies* pueden ajustarse mediante múltiples parámetros para adaptarse a las características de cada instancia Cai et al. 2022.

CaDiCaL alterna entre dos modos operativos principales: modo estable y modo enfocado. En el modo estable, se utilizan reinicios tipo Luby1.3.3 con un intervalo base elevado (1024 conflictos). Este modo promueve estabilidad, realiza retrocesos cronológicos y aplica la heurística VSIDS con bajo factor de incremento. Por otro lado, en el **modo enfocado**, se aplican reinicios como *Glucose-style*1.3.3 con intervalos cortos entre reinicios (2 conflictos).

1.4.3. *Assumptions*

Las *assumptions* permiten invocar un SAT *solver* con un conjunto de valores previamente asignados a ciertas variables. Esta funcionalidad resulta particularmente útil en contextos donde el *solver* se utiliza de manera iterativa.

Por ejemplo, si se requiere agregar o eliminar una cláusula c_i entre distintas invocaciones al *solver*, esta puede incorporarse a la fórmula como $(c_i \vee s_i)$, donde s_i es una variable de activación, selección o indicador. Al asignar $s_i = 1$, la cláusula c_i se incluye efectivamente en la fórmula; en cambio, al establecer $s_i = 0$, dicha cláusula se omite.

Esta técnica fue introducida originalmente en el *solver* MiniSat Marques-Silva et al. 2024.

Además de su uso en la resolución incremental, las variables de activación también se aplican para representar cláusulas temporales. Por ejemplo, para incorporar una cláusula temporal c, se introduce una variable de activación a y se añade $(c \vee a)$ a la fórmula, bajo la suposición de $\neg a$. Para eliminar c, basta con agregar la cláusula unitaria (a).

En versiones recientes, CaDiCaL ha incorporado el uso de *constraints*, que pueden simularse a través de *activation literals* en versiones anteriores Biere, Fleury et al. 2024.

El incremento en el número de variables de activación puede afectar negativamente el rendimiento del *solver*. Tradicionalmente, este inconveniente se ha abordado mediante reinicios periódicos Su et al. 2025.

Las *assumptions* se asignan y se propagan antes de que se inicie la búsqueda efectiva mediante la heurística de decisión de variables. Esta ordenación obedece a la arquitectura de los *solvers* CDCL (*Conflict-Driven Clause Learning*) Marques-Silva et al. 2024.

El algoritmo CDCL comienza con una fase de propagación unitaria. Si la PU

detecta un conflicto en el nivel de decisión 0 (es decir, antes de que el *solver* haya realizado alguna decisión propia), se concluye que la fórmula, junto con las *assumptions* actuales, es insatisfacible Sun et al. 2024.

Esta verificación temprana resulta fundamental para la eficiencia del proceso, ya que permite detectar la insatisfacibilidad sin necesidad de explorar el espacio de búsqueda a través de decisiones de ramificación que, eventualmente, conducirían a un conflicto inevitable debido a los supuestos Marques-Silva et al. 2024.

Solo si la PU no detecta conflictos y no todas las variables están asignadas, el *solver* procede a seleccionar una variable para ramificar, utilizando la heurística de decisión correspondiente Sun et al. 2024.

Las *assumptions* constituyen un punto de partida para el *solver*. Estas se procesan inicialmente mediante la propagación unitaria, lo que permite verificar que la configuración inicial, incluyendo los supuestos, no conduzca a un conflicto inmediato. Esta estrategia optimiza el procedimiento, al evitar búsquedas infructuosas en regiones del espacio que se sabe de antemano que son inconsistentes con los supuestos proporcionados.

1.4.4. *Phasing*

La estrategia de *phasing* constituye un componente fundamental en los SAT *solvers* basados en CDCL, ya que desempeña un papel crucial en la guía del proceso de búsqueda de soluciones.

En el contexto de los *solvers* CDCL, una vez seleccionada una variable para ramificación, se debe decidir qué valor de verdad (también denominado *fase* o *polaridad*) asignarle. Esta decisión influye significativamente en la eficiencia del *solver*, particularmente en fórmulas satisfacibles.

La técnica de *phase saving* (ahorro de fases) consiste en registrar el último valor asignado a una variable (ya sea mediante decisión o propagación unitaria) y reutilizar este valor cuando la variable sea seleccionada nuevamente como variable de decisión. Esta técnica, sencilla y de bajo costo computacional, ha demostrado mejorar el rendimiento de forma notable y se ha convertido en un estándar en la mayoría de los *solvers* CDCL modernos. Se clasifica como una estrategia de intensificación, al mantener la búsqueda dentro de regiones previamente exploradas y potencialmente prometedoras Cai et al. 2022.

Rephasing (refaseo) es una técnica de diversificación que complementa el ahorro de fases. Su propósito es reiniciar o ajustar la asignación parcial de las fases, permitiendo al *solver* explorar nuevas regiones del espacio de búsqueda. Dado que la selección de fase no compromete la corrección ni la terminación del algoritmo CDCL, las fases guardadas pueden modificarse arbitrariamente Cai et al. 2022.

CaDiCaL incorpora mecanismos avanzados tanto de ahorro de fases como de

rephasing Biere, Fleury et al. 2024. Ha sido pionero en la implementación de refaseo periódico, en el que los valores de fase se reinician en intervalos predefinidos durante la ejecución.

1.4.5. Modularidad y Extensibilidad del Código de CaDiCaL

La modularidad y la extensibilidad constituyen objetivos esenciales en el diseño de CaDiCaL Biere, Fleury et al. 2024. 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 Biere, Fleury et al. 2024.

1.4.6. 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 Cai et al. 2022.
- **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.

Biere, Fleury et al. 2024

1.5. Parámetros de Evaluación para *solvers* CDCL: Taxonomía de extitBenchmarks

1.5.1. Categorías

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.

En primer lugar, las instancias de aplicación proceden de problemas industriales, como la verificación de circuitos o la planificación logística. Estas fórmulas presentan estructuras modulares y *backdoors* 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 Zulkoski 2018.

Por otro lado, las instancias 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 Zulkoski 2018.

Además, las instancias 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.

1.5.2. 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.

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.

1.5.3. 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 4.26 cláusulas por variable, conocido como umbral de transición de fase. En esta zona, heurísticas estáticas como DLIS pierden eficacia ante la ausencia de patrones explotables. 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.

1.5.4. 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, VSIDS prevalece en aplicaciones reales gracias a su adaptación dinámica a patrones de conflicto.

1.6. Problemas

1.6.1. Random k-SAT

Los problemas Random k-SAT son un tipo de instancias generadas aleatoriamente que se utilizan ampliamente para evaluar y caracterizar el rendimiento de los algoritmos SAT. También se conocen como modelos de longitud fija de cláusula y constituyen un objeto de estudio prominente en la literatura Hoos 1998.

En un problema Random k-SAT, se define un número de variables n, un número de cláusulas m y una longitud fija de cláusula k. Para generar cada cláusula, se seleccionan k literales de forma independiente y uniforme al azar del conjunto de posibles literales, que incluye las variables proposicionales y sus negaciones. No se incluyen cláusulas que contengan múltiples copias del mismo literal. El proceso de

generación continúa hasta que la fórmula contiene el número total especificado de cláusulas m Hoos 1998.

Una propiedad destacada del Random k-SAT es la presencia de un fenómeno de transición de fase. Este fenómeno se manifiesta como un cambio abrupto en la solubilidad al variar sistemáticamente el número de cláusulas m para un número fijo de variables n. Cuando m es pequeño, casi todas las fórmulas son poco restringidas y, por ende, satisfacibles. Al alcanzar un valor crítico m_c, la probabilidad de que una instancia sea satisfacible disminuye abruptamente hasta casi cero. Más allá de este punto, la mayoría de las instancias se encuentran sobrerestringidas y son insatisfacibles. Para el caso de Random 3-SAT, esta transición ocurre aproximadamente cuando la razón cláusulas/variables (m/n) es cercana a 4.26 para valores grandes de n. La dificultad de los problemas SAT alcanza su punto máximo en esta región de transición Hoos 1998, Ganesh y Vardi s.f.

1.6.2. Problema del palomar

El "problema del palomar" (*pigeonhole problem*) es una instancia de problema de Satisfacibilidad Booleana (SAT) que se ha convertido en un punto de referencia en la investigación de solvers SAT. A pesar de ser un problema combinatorio de tamaño relativamente pequeño, presenta dificultades particulares para los solvers CDCL (Conflict-Driven Clause Learning).

Este problema es un ejemplo de fórmulas "artesanales" o generadas aleatoriamente que resultan difíciles para los solvers CDCL Zulkoski 2018. Se considera una de las clases de problemas para los cuales los solvers CDCL son "sin esperanza" en la demostración de insatisfacibilidad, dado que cualquier prueba de refutación por resolución requerirá un tamaño exponencial Oh 2016, enero.

Este comportamiento contrasta con la eficiencia que muestran los solvers CDCL para problemas industriales o del mundo real Zulkoski 2018. La dificultad radica en que los solvers CDCL no pueden generar pruebas de refutación sofisticadas de manera eficiente para este tipo de instancias, a diferencia de los problemas industriales donde sí pueden hacerlo Oh 2016, enero.

El problema del palomar se usa como ejemplo para entender por qué los solvers CDCL son eficientes en muchas clases de instancias del mundo real, pero tienen un rendimiento pobre en instancias generadas aleatoriamente o criptográficas Ganesh y Vardi s.f. La investigación se centra en comprender cómo las propiedades estructurales de las fórmulas se relacionan con la resolución SAT basada en CDCL Zulkoski 2018.

1.6.3. Problema de coloreo de grafos

El problema de coloreado de grafos (GCP) es un concepto fundamental en informática y matemáticas, estrechamente relacionado con los Problemas de Satisfacción de Restricciones (CSP) y los Problemas de Satisfacibilidad Booleana (SAT) AlmaBetter 2025.

GCP es una subclase importante dentro de los Problemas de Satisfacción de Restricciones (CSP). Su objetivo es asignar colores a los vértices de un grafo dado de modo que dos vértices conectados por una arista nunca compartan el mismo color. Un ejemplo sencillo de CSP es el coloreado de grafos, donde se busca que cada nodo adyacente tenga un color diferente. El problema de colorear la bandera canadiense es un ejemplo simple de coloreado de mapas, que a su vez es un caso particular del GCP Hoos 1998.

Las instancias de coloreado de grafos con tres colores se cuentan entre los benchmarks más comúnmente usados para CSP Hoos 1998.

La estrategia de "cold restart" FO (Forgetting Order), que consiste en reiniciar el solver olvidando el orden inicial de ramificación, ha demostrado ser especialmente adecuada para resolver instancias de la familia de problemas de coloreado **??**.

1.6.4. XOR

El problema de paridad XOR es un área importante dentro de la Satisfacibilidad Booleana (SAT) y campos relacionados, especialmente en criptografía y teoría de códigos. Se refiere a un caso especial de los problemas XOR-SAT Nandi et al. 2024. Surge cuando las fórmulas booleanas se expresan como una conjunción de cláusulas donde la suma en $\mathbb{F}_2$ (base 2) se convierte en la operación lógica XOR ($\oplus$) Trimoska et al. 2020. Estas son comúnmente llamadas cláusulas XOR o cláusulas de chequeo de paridad Nandi et al. 2024.

1.6.5. *Bounded Model Checking* (BCM)

Bounded Model Checking (BCM) es un método de verificación de modelos simbólicos utilizado en la verificación formal de *hardware* y *software*. Es una técnica cada vez más aceptada en la industria para detectar una gama más amplia de errores, incluyendo condiciones de error sutiles, en comparación con las técnicas de validación tradicionales basadas en simulación.

En un enfoque BMC, se utilizan codificaciones en Forma Normal Conjuntiva (FNC) y algoritmos SAT estándar para encontrar errores de manera más rápida y de tamaño mínimo Hoos 1998.

Instancias de problemas relacionados con BMC, como "hardware-bmc" y "hardware-bmc-ibm", se han utilizado en competiciones SAT para evaluar el rendimiento de los

solucionadores Biere y Fröhlich 2019.

1.7. Insuficiencias Fundamentales de SAT

Aunque los algoritmos SAT basados en Conflict-Driven Clause Learning (CDCL) han tenido un impacto dramático y son sorprendentemente eficientes para resolver instancias con millones de variables y cláusulas en aplicaciones del mundo real Li et al. 2024, presentan varias limitaciones e insuficiencias importantes, tanto teóricas como prácticas Marques-Silva et al. 2024.

El problema de Satisfacibilidad Booleana (SAT) es un problema NP-completo, lo que implica que, bajo la suposición de que $P \neq NP$, cualquier algoritmo completo para SAT operará en el peor de los casos en tiempo exponencial Marques-Silva et al. 2024. Por otro lado, hay clases de problemas para las cuales las refutaciones por resolución son de tamaño exponencial, lo que hace que los solucionadores CDCL sean inherentemente ineficientes para demostrar su insatisfacibilidad. Ejemplos notables incluyen: el Principio del Agujero del Palomar (*Pigeonhole Principle*) o fórmulas generadas aleatoriamente Ganesh y Vardi s.f.

Por otro lado, SAT presenta desafíos prácticos. Un desafío fundamental es que un conjunto de técnicas que funcionan bien para una clase de instancias puede fallar estrepitosamente para otra Li et al. 2024. Es muy difícil encontrar una heurística que tenga un alto rendimiento en cualquier instancia considerada Cherif et al. 2021. Asimismo, a menudo, las técnicas que mejoran el rendimiento en instancias satisfacibles (como la exploración de ramas prometedoras mediante búsqueda local o el reinicio frío que olvida fases) pueden degradar ligeramente el rendimiento en instancias insatisfacibles, o viceversa. Es muy difícil mejorar el rendimiento de forma simultánea en ambos tipos de instancias Cherif et al. 2024.

En cuanto a la gestión de cláusulas aprendidas, su acumulación excesiva puede ser perjudicial para el rendimiento del solucionador y puede "paralizar completamente un solucionador". Gran parte de las cláusulas aprendidas no son útiles para derivar una prueba de insatisfacibilidad en el sentido de rendimiento práctico del solucionador. Esto sugiere una ineficiencia en el proceso de aprendizaje, ya que solo una pequeña fracción de "lemas simples" de alta calidad son suficientes para resolver problemas del mundo real rápidamente Oh 2016, enero. La minimización de cláusulas aprendidas, aunque necesaria para el rendimiento, es una tarea compleja e, históricamente, se creía que era difícil de hacer efectiva con la propagación de unidades Luo et al. 2017.

Por lo tanto, si bien los solucionadores CDCL SAT son herramientas increíblemente potentes para una amplia gama de aplicaciones, sus insuficiencias se derivan de limitaciones teóricas inherentes a su modelo de prueba (resolución), desafíos prácticos en la gestión de heurísticas y cláusulas aprendidas para diversas instancias, y una

dificultad persistente para explotar eficazmente la estructura subyacente de ciertos problemas Ganesh y Vardi s.f.

Capítulo 2

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.

2.0.1. 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 Biere, Fleury et al. 2024.

[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)

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* 2.1:

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

Ejemplo de código 2.1: Integración de DLIS en CaDiCaL. Internal.hpp

internal.cpp En *internal.cpp* se a/ nadi/'o el c/'odigo 2.2 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_clauses_(idx);
      int neg_score =
         count_literal_in_unsatisfied_clauses_(-idx);

      float med_score = (pos_score + neg_score)/2; // Se
         toma el promedio por la estrategia de phase

      // Comparar con el mejor hasta ahora
      if (med_score > best_score) {
        best_score = med_score;
        best_lit   = idx;
      }
    }
    // 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);
```

Ejemplo de código 2.2: Integración de DLIS en CaDiCaL. Internal.cpp

Es importante aclarar que en 2.2 el cálculo del *score* de cada literal no se hace tal cual dicta DLIS, pues en aras de mantener la consistencia del código de CaDiCaL se usaron estrategias y estructuras que ya vienen implementadas, y a las cuales se acoplan las heurísticas de decisión de variables que ya incorpora el solucionador. En este caso, obsérvese que se promedian los *scores* de ambos literales. Esto, debido a que CaDiCaL implementa la estrategia *phase* 1.4.4 para elegir la polaridad de la variable. Por esta razón el método devuelve la variable y no el literal.

Obsérvese que este método hace un llamado a `int count_literal_in_unsatisfied_clauses_(i` que es el encargado de calcular el *score* para un literal 2.3.

```
int Internal::count_literal_in_unsatisfied_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) {
    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;
}
```

Ejemplo de código 2.3: Integración de DLIS en CaDiCaL. Calcular *score*. Internal.cpp

En el código 2.3 puede verse que para efectuar el conteo de ocurrencias de un literal no se revisan todas las cláusulas insatisfechas; en su lugar se aprovecha la estructura que usa CaDiCaL para aplicar la estrategia *Two Watched Literals* (TWL)1.3.4. Luego, se recorren solo las cláusulas insatisfechas donde el literal esté vigilado, y se procede con el cálculo de su *score*. Con esta implementación se busca disminuir para algunos problemas el costo de recorrer todas las clausulas.

decide.cpp

El flujo de decisión sobre la próxima variable a asignar se lleva a cabo en el archivo *decide.cpp*, específicamente en el siguiente método 2.4:

```
int Internal::decide () {
  assert (!satisfied ());
  START (decide);
  int res = 0;

  //... implementacion de assumptions

    else {
```

```
    int decision = 0;
    int idx = 0;

    if (opts.dlis /*&& stats.decisions < threshold*/) {
      //decision = pick_dlis_branch_literal();
      idx = next_decision_variable_with_dlis ();
      LOG ("DLIS decision literal %d", decision);
    } else {
      idx = next_decision_variable ();
    }
    const bool target = (opts.target > 1 || (stable &&
       opts.target));
    if (idx) decision = decide_phase (idx, target);


    if (decision) {
      stats.decisions++;
      LOG ("deciding literal %d", decision);
      search_assume_decision (decision);
    }
  }

  if (res) marked_failed = false;
  STOP (decide);
  return res;
}
```

Ejemplo de código 2.4: Integración de DLIS en CaDiCaL. decide.cpp

Como se puede observar en 2.4, se respeta el orden de decisión que sigue CaDiCaL, garantizando primero el análisis de los valores de las variables según *assumption* 1.4.3. Luego, antes de aplicar DLIS se verifica si su flag correspondiente está activada (DLIS no se usa por defecto), y en caso positivo se procede con la elección de la variable según lo explicado anteriormente. Obsérvese también que heurísticas las heurísticas de decisión de variables implementadas en el *solver* se analizan luego de *assumptions*.

Una vez escogida la variable, se llama al método *decide_phase* para elegir su polaridad en base a la estrategia *phasing* 1.4.4.

options.hpp

Finalmente, para habilitar la *flag* *–dlis=<bool>* para activar y desactivar la heurística incorporada, se añadió la siguiente línea al archivo *options.hpp* 2.5.

```
#define OPTIONS \
\
/*       NAME          DEFAULT, LO, HI,O,P,R, USAGE */ \
\
//... varias options...
OPTION( dlis,                0,  0,  1,0,0,1, "use DLIS
   decision heuristic") \
//...varias options...
```

Ejemplo de código 2.5: Integración de DLIS en CaDiCaL. options.hpp

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

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

- VSIDS + restart = `-score=false`
- VSIDS + no restart = `-score=false -restart=false`
- DLIS + restart = `-dlis=true -score=false`
- DLIS + no restart = `-dlis=true -score=false -restart=false`

La *flag* `-score=false` desactiva el empleo de la estrategia EVSIDS (citar en el marco teórico) para usar solo el *bump* característico de VSIDS. El resto de las que están son bastante descriptivas. Fueron usadas, además, dos *flags* generales para cada heurística: `-t 120` que pone un *timeout* de máximo 120 segundos, y `-stats` que muestra estadísticas extra para los problemas resueltos (resultado = *SATISFIABLE*/*UNSATISFIABLE*).

2.0.3. Problemas

Cada combinación de heurística fue probada en cada uno de los 135 problemas generados que abarcan las siguientes categorías:

- Random 3-SAT
  - Se generaron 15 instancias.
  - Tamaños de variables: 1000, 2000, 5000 (rotando cíclicamente).
  - Relación cláusulas/variables: 3.76, 4.26 y 4.76 (alrededor del umbral de fase para 3-SAT).

  - Ejemplo de archivo: random3sat_n1000_r4.26_0.cnf.

- *Pigeonhole Principle* (principio del palomar)
  - Se generaron 15 instancias.
  - Número de hoyos: 10, 20, 30, 40, 50 (rotando cíclicamente).
  - Número de palomas: siempre n_hoyos + 1.
  - Ejemplo de archivo: pigeon_11_into_10_0.cnf.

- Random 4-SAT
  - Se generaron 15 instancias.
  - Tamaños de variables: 500, 1000, 2000 (rotando cíclicamente).
  - Relación cláusulas/variables: 8.88, 9.88, 10.88 (alrededor del umbral de fase para 4-SAT).
  - Ejemplo de archivo: random4sat_n500_r9.88_0.cnf.

- *Graph Coloring* en Grafos Aleatorios
  - Se generaron 10 instancias.
  - Número de nodos: 50, 100, 200 (rotando cíclicamente).
  - Probabilidad de arista: 0.1, 0.3, 0.5 (rotando cíclicamente).
  - Número de colores: 3, 4, 5 (rotando cíclicamente).
  - Ejemplo de archivo: graphcol_n50_p0.10_k3_0.cnf.

- *Parity* (XOR) *Constraints*
  - Se generaron 10 instancias.
  - Número de variables: 10, 20, 30, 40, 50 (rotando cíclicamente).
  - Ejemplo de archivo: parity_n10_0.cnf.

- BMC de Flip-Flop Simple
  - Se generaron 10 instancias.
  - Profundidad del circuito: 3, 5, 7, 9, 11 (rotando cíclicamente).
  - Ejemplo de archivo: bmc_flipflop_d3_0.cnf.

- Problemas *DLIS-friendly*
  - Se generaron 30 instancias.

- Número de variables: 200, 300, 400 (seleccionado aleatoriamente).
- Tamaño de cláusula: 5 o 6 (aleatorio).
- Relación cláusulas/variables: valor real entre 2.0 y 3.5 (aleatorio).
- Sesgo positivo en literales: entre 0.7 y 0.8 (aleatorio).
- Ejemplo de archivo: biased_random5sat_n200_r2.45_b0.73_2000.cnf.

- Problemas *no DLIS-friendly*

  - Se generaron 30 instancias.
  - Número de variables: 1000, 2000, 5000 (aleatorio).
  - Tamaño de cláusula: 3.
  - Relación cláusulas/variables: valor real entre 4.0 y 5.0 (aleatorio).
  - Sin sesgo en literales: bias_pos = 0.5.
  - Ejemplo de archivo: biased_random3sat_n1000_r4.32_b0.50_3000.cnf.

El generador cubre exhaustivamente ocho familias de problemas: Random 3-SAT, *Pigeonhole*, Random 4-SAT, *Graph Coloring*, *Parity*, BMC Flip-Flop, *DLIS-friendly* y *no DLIS-friendly*. Cada familia tiene parámetros clave que varían (número de variables, relación cláusulas/variables, sesgo, tamaño de cláusula, profundidad, etc.), asegurando diversidad estructural y de dificultad en los *benchmarks* generados.

2.0.4. Generador de problemas

El generador de problemas (*benchmarks*) se programó en python y se usaron las bibliotecas *os*, *random* y *csv*. Los resultados se exportaron a una carpeta *"generated_benchmarks"* en el mismo directorio del *script* cuyos archivos se encuentran en formato DIMACS (referencia al marco teórico) en .cnf.

2.0.5. Estad/'isticas

El análisis estadístico de los resultados de la comparación de cada heurística por problema está implementado en el archivo *stats_analysis.ipynb*. Se comparó los problemas cuyo resultado fue *TIMEOUT* con los que sí resolvieron las heurísticas. Se hace un análisis de la distribución de cada característica en problemas *TIMEOUT* por heurística. Para el caso de los problemas resueltos (resultado = *SATISFIABLE/UNSATISFIABLE*) se hace de igual modo un análisis de la distribución y se incluye la característica *tiempo en segundos*. Posteriormente se hace un análisis comparativo entre los problemasa que fueron resueltos y los que no. Para ello se realiza un análisis entre variables y se eliminan aquellas que sean colineales. Esta comprobaciń se

realiza mediante el cálculo del Factor de Inflación de Varianza (VIF). Para realizar la comparación en cuanto a características entre los problemas resueltos y los que no, se realizan las pruebas de Mann-Whitney U, y para visualizar los resultados se emplean gráficas de caja y bigotes. Además, también se usa regresión logística ya que permite interpretar cómo cada variable afecta la probabilidad de que ocurra o no un *TIMEOUT*.

La otra parte de la evaluación consiste en un análisis del rendimiento de cada combinación de heurísticas en cuanto al tiempo de ejecución en los problemas resueltos. Para ello se hizo un análisis de la correlación de Spearman entre las características de los problemas y el tiempo en segundos. Se realizó, además, el test de Kruskal-Wallis para analizar si el tiempo difiere entre heurísticas para un mismo tipo de problema. Asimismo, se realizó el test de Dunn por cada heurística para realizar una comparación estadística post-hoc (prueba de Dunn) para analizar si existen diferencias significativas en los tiempos de resolución (log(tiempo)) de una heurística específica en los problemas resueltos, según los valores de una característica. Los resultados son graficados en *boxplot*. Finalmente, se realiza una regresión lineal múltiple para definir qué características predicen el tiempo de resolución.

Para la implementación de estos análisis se usaron las siguientes bibliotecas de python: `pandas`, `numpy`, `seaborn`, `matplotlib.pyplot`, `statsmodels.api`, `variance_inflation_fac` y variance_inflation_factor, ambas de `statsmodels.stats.outliers_influence`, `ols` de `statsmodels.formula.api`, `scipy.stats`, `LogisticRegression` de `sklearn.linear_model`, y `scikit_posthocs`.

2.1. Resultados

2.1.1. Análisis de los problemas con TIMEOUT

Tabla resumen de TIMEOUTs

En la tabla (insertar referencia) se muestran las caracteristicas de los problemas cuyo resultado fue timeout ademas del nombre de dicho problema y la heuristica. (visualizar estos datos mediante graficas)

Con base en el conjunto de resultados filtrados por TIMEOUT, se observa que las heurísticas evaluadas presentan diferencias claras en su comportamiento frente a instancias con distintas características estructurales. Por ejemplo, la heurística VSIDS combinada con reinicio tiende a experimentar TIMEOUT en instancias con menor número de variables pero con una densidad relativamente alta de cláusulas por variable, lo que sugiere que esta configuración puede ser menos efectiva en problemas más densos aunque de tamaño moderado.

En contraste, DLIS sin reinicio acumula TIMEOUT en instancias con un mayor

número de variables pero con densidad menor, indicando que esta heurística puede tener dificultades para escalar en problemas más grandes, aunque menos densos. Además, la presencia o ausencia de reinicios parece influir significativamente en el rendimiento, dado que las combinaciones con reinicio generalmente muestran un patrón distinto de TIMEOUT frente a las que no lo incorporan.

Estas observaciones permiten suponer que la elección de heurística y estrategia de reinicio debe adaptarse a la estructura específica del problema SAT para optimizar el rendimiento y evitar fallos por límite de tiempo.

Estadísticas descriptivas

El análisis estadístico descriptivo y las visualizaciones mediante diagramas de caja permiten observar con mayor detalle cómo se distribuyen las características de las instancias que resultaron en TIMEOUT según la heurística aplicada. En términos generales, las cuatro heurísticas (DLIS+no-restart, DLIS+restart, VSIDS+no-restart y VSIDS+restart) presentan distribuciones muy similares en cuanto a las métricas de número de variables, número de cláusulas, densidad, tamaño promedio de cláusula y cantidad de variables positivas y negativas.

Por ejemplo, en la figura 2.1 puede apreciarse que el número de variables en los casos con TIMEOUT oscila entre valores mínimos cercanos a 100 y máximos de 5000, con medianas alrededor de 2000 para todas las heurísticas, lo que indica que los problemas que causan TIMEOUT no se limitan a un rango estrecho de tamaño.

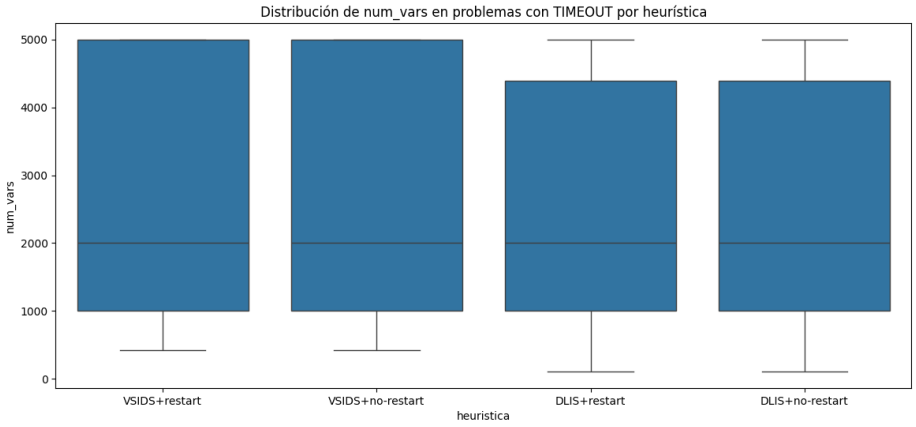

Figura 2.1: Distribución de número de variables en problemas con timeout por heurística.

Por otro lado, en la figura 2.2 puede observarse que el número de cláusulas de las instancias analizadas muestra valores elevados y gran variabilidad, con promedios

entre 15.758 y 16.875 cláusulas y máximos por encima de 63.000. No se observan diferencias significativas entre las estrategias con DLIS o VSIDS, ni tampoco al incorporar o no el mecanismo de restart.

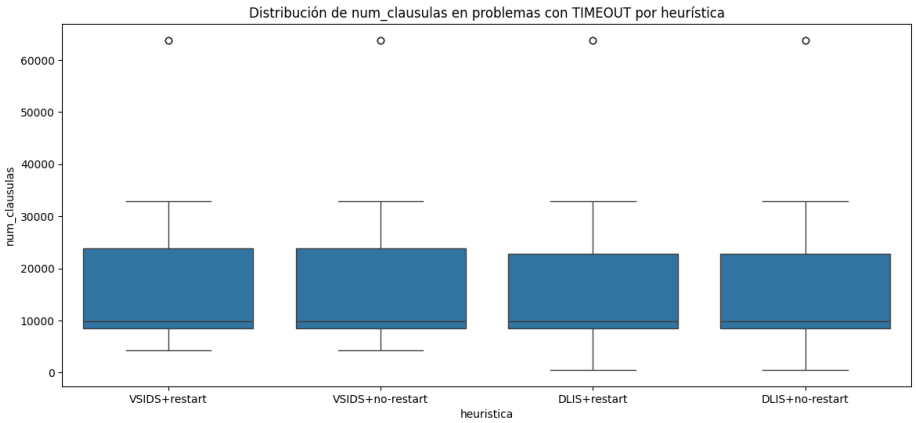

Figura 2.2: Distribución de número de cláusulas en problemas con timeout por heurística.

Como se muestra en 2.3, la densidad media se mantiene cercana a 7.4–7.7 para todas las heurísticas, con una dispersión considerable que alcanza hasta 25, lo que refleja que tanto problemas poco densos como muy densos pueden provocar fallos por tiempo.

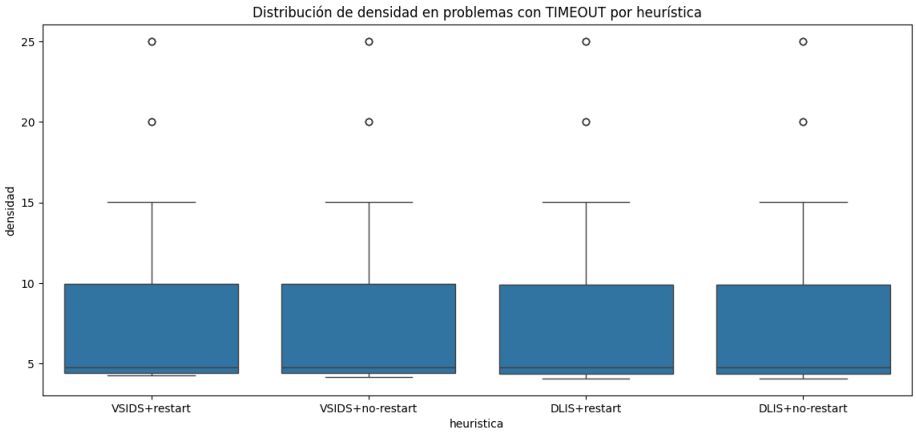

Figura 2.3: Distribución de densidad en problemas con timeout por heurística.

En la figura 2.4 se puede apreciar que el tamaño promedio de cláusula se mantiene estable alrededor de 3, con poca variabilidad, sugiriendo que esta característica no discrimina el comportamiento de las heurísticas en cuanto a TIMEOUT.

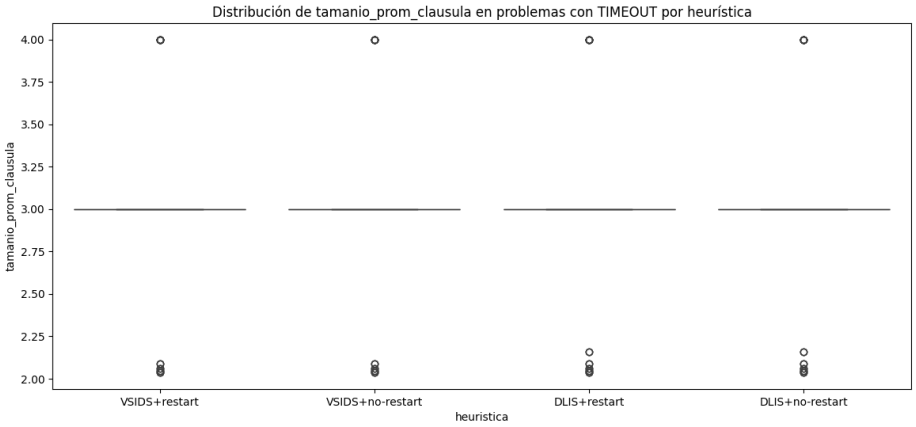

Figura 2.4: Distribución de tamaño promedio de cláusula en problemas con timeout por heurística.

Por su parte, en las gráficas 2.5 y 2.6, se puede ver que las variables positivas y negativas, respectivamente, también muestran simetría y valores medios similares entre heurísticas, lo que implica que la polaridad de las variables no es un factor determinante en la ocurrencia de TIMEOUT.

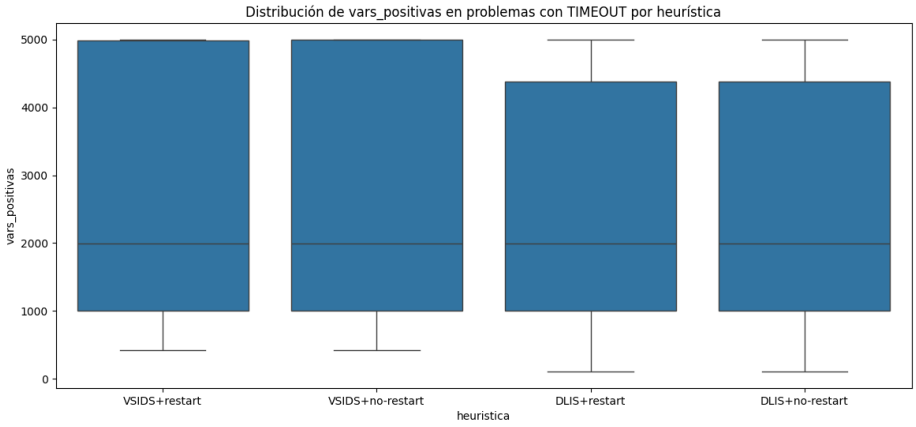

Figura 2.5: Distribución de variables positivas en problemas con timeout por heurística.

En conjunto, estos resultados sugieren que las diferencias en el rendimiento entre heurísticas no se explican fácilmente por las características básicas de las instancias con TIMEOUT.

Las visualizaciones de caja muestran solapamientos amplios en las distribuciones de cada característica por heurística.

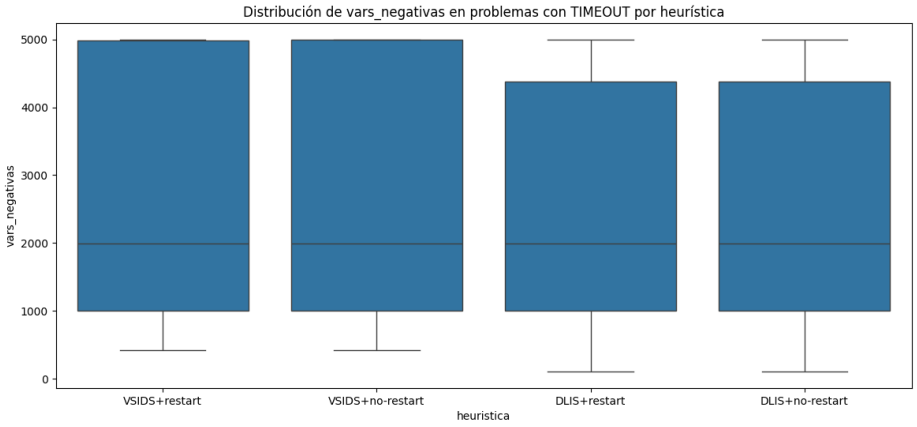

Figura 2.6: Distribución de variables negativas en problemas con timeout por heurística.

Conteo de TIMEOUT por heurística y por problema

En las gráficas 2.7 y 2.8 se puede observar que los problemas que tomaron mas tiempo en resolverse coinciden con los los planteados en la literatura que resultan mas dificiles de resolver a instancias SAT. Respecto a la cantidad de problemas con TIMEOUTS por heuristica, se puede apreciar que DLIS, independientemente del restart, se comporta mas lento que vsids para algunos problemas. Por su parte, vsids con restart se comporta mas lento que vsids sin restart para unas pocas instancias.

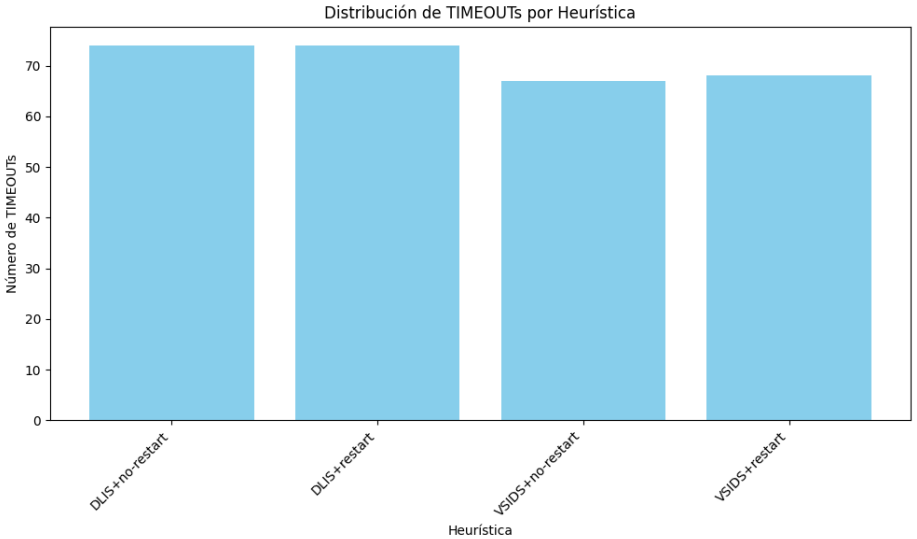

Figura 2.7: Distribución de timeouts por heurística.

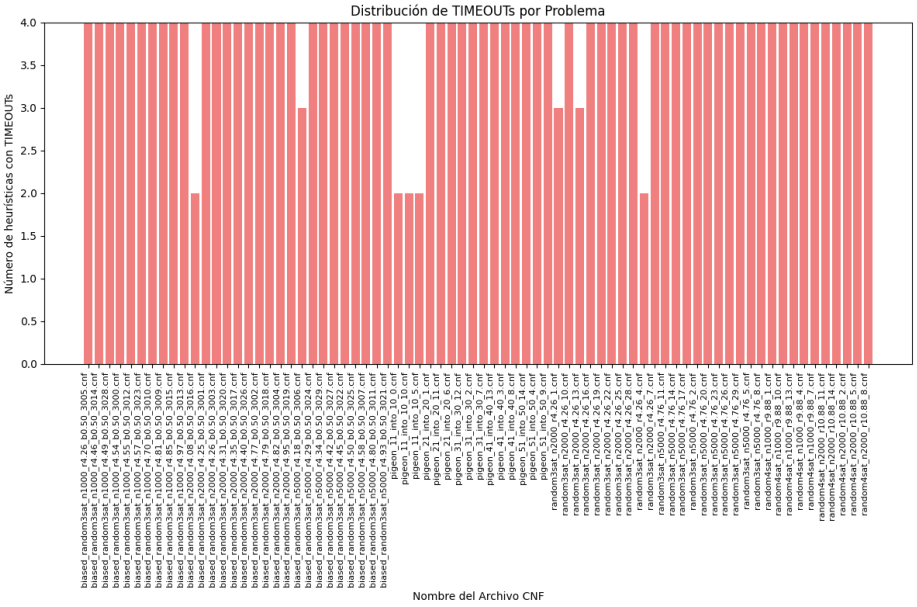

Figura 2.8: Distribución de timeouts por problema.

2.1.2. Análisis de problemas resueltos (SATISFIABLE/UNSATISFIABLE)

Tabla resumen de RESUELTOS

(insertar grafico que describa los resultados de la tabla filtrada por problemas resueltos)

El análisis de los problemas resueltos muestra que las instancias que concluyeron satisfactoriamente (ya sea SATISFIABLE o UNSATISFIABLE) abarcan un amplio rango de tamaños y características, desde problemas muy pequeños con 6 variables y 12 cláusulas hasta otros considerablemente más grandes con hasta 5000 variables y más de 23000 cláusulas.

Los tiempos de resolución registrados varían notablemente, desde fracciones de segundo en problemas pequeños (por ejemplo, alrededor de 0.0014 segundos en instancias con 6 variables) hasta varios segundos en problemas más complejos (por ejemplo, cerca de 9 segundos en instancias con 500 variables y alta densidad). Esto indica que, aunque el solver puede resolver eficientemente instancias pequeñas o medianas, el tiempo de cómputo crece conforme aumentan la cantidad de variables y cláusulas, así como la densidad del problema.

La presencia de las cuatro heurísticas en los resultados resueltos sugiere que todas son capaces de resolver ciertos conjuntos de problemas. Sin embargo, la diversidad en tiempos y tamaños sugiere que la heurística y la estrategia de reinicio podrían influir

en la eficiencia, especialmente en problemas de mayor escala.

Estadísticas descriptivas

El análisis estadístico descriptivo del tiempo de resolución y las características de las instancias resueltas por cada heurística revela patrones importantes sobre el comportamiento del *solver*. Estos resultados pueden apreciarse en conjunto en la gráfica 2.9.

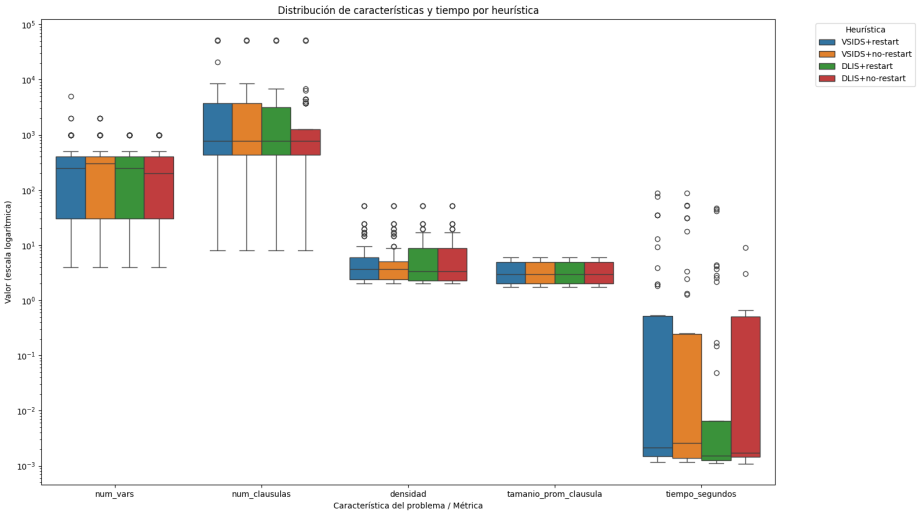

Figura 2.9: Distribución de características y tiempo por heurística.

En primer lugar, se observa que las instancias resueltas por las cuatro heurísticas (DLIS+no-restart, DLIS+restart, VSIDS+no-restart y VSIDS+restart) cubren un rango amplio en tamaño, con número de variables promedio entre 322 y 403, y máximos que alcanzan hasta 2000 o incluso 5000 variables en el caso de VSIDS+restart. La cantidad de cláusulas también presenta gran dispersión, con medias cercanas a 3100–3400 y máximos que superan las 50,000, lo que indica que el solver puede manejar problemas desde muy pequeños hasta instancias complejas. La densidad promedio se mantiene alrededor de 7,2 a 7,4, con valores máximos muy altos (hasta 51.9), reflejando diversidad en la estructura de los problemas.

En cuanto al tiempo de resolución, se evidencia una marcada diferencia entre heurísticas: DLIS+no-restart presenta el tiempo medio más bajo (0.37 s), seguido por DLIS+restart (3.13 s), mientras que VSIDS+no-restart y VSIDS+restart muestran tiempos medios más elevados, cerca de 4.98 y 4.98 segundos respectivamente, con alta variabilidad (desviaciones estándar muy grandes y máximos que superan los 80 segundos). Esta dispersión sugiere que aunque VSIDS puede ser eficiente en muchos casos, también enfrenta problemas que requieren tiempos significativamente mayores.

Además, la mediana del tiempo para todas las heurísticas es muy baja (en el orden de milisegundos), indicando que la mayoría de las instancias se resuelven rápidamente, pero existen casos atípicos que prolongan el tiempo promedio.

El tamaño promedio de cláusula es similar entre heurísticas, alrededor de 3.4, y las variables positivas y negativas mantienen valores simétricos, lo que indica que estas características no explican las diferencias en tiempo.

En conjunto, estos resultados sugieren que la elección de heurística impacta significativamente el tiempo de resolución en ciertos problemas, especialmente en instancias grandes o complejas, y que VSIDS puede tener mayor variabilidad en rendimiento.

2.1.3. Porblemas RESUELTOS vs TIMEOUT

Análisis de multicolinealidad

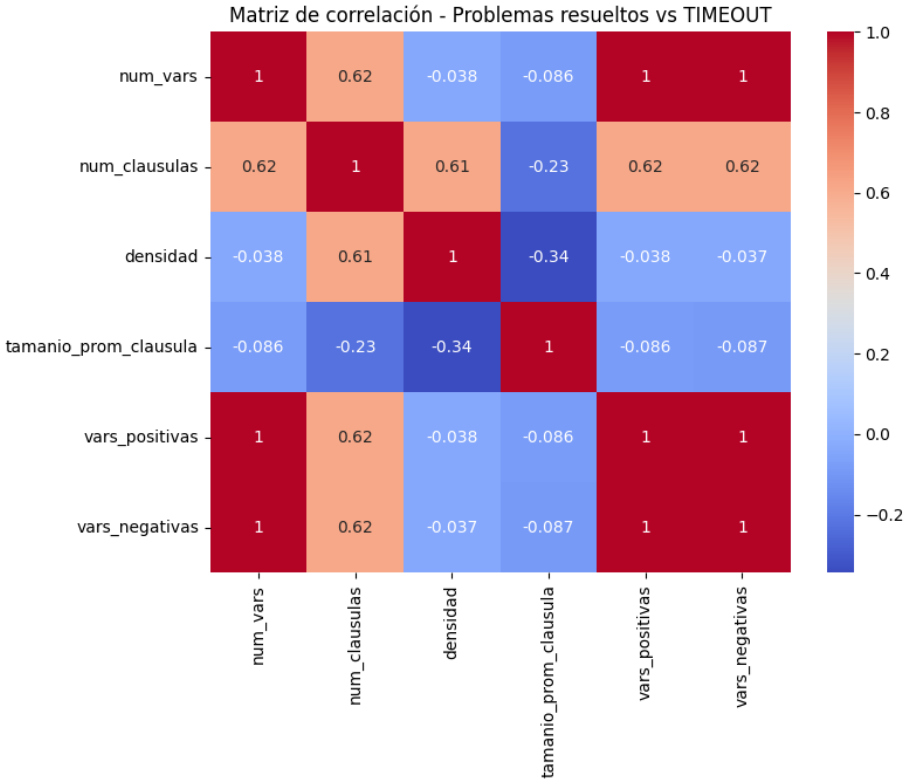

Figura 2.10: Matriz de correlación - Problemas resueltos vs TIMEOUTS.

Como se puede apreciar en la matriz de correlación 2.10, entre las variables num_vars y vars_positivas y vars_negativas, existe correlación perfecta ($r = 1$). Además, existe una correlación moderada entre las variables num_clausulas con

vars_negativas, vars_positivas, num_vars y densidad). Luego, es necesario reducir el conjunto de características, quedando el siguiente: `caracteristicas_reducidas = ['num_vars', 'densidad', 'tamanio_prom_clausula']`.

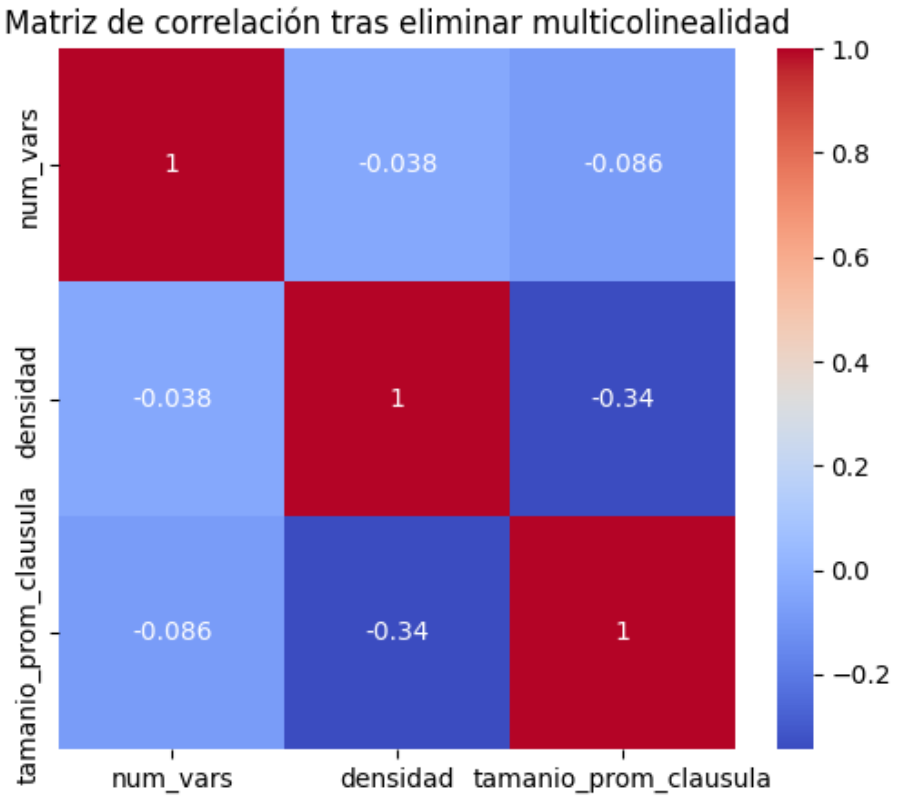

Figura 2.11: Matriz de correlación tras eliminar multicolinealidad.

Como se muestra en la matriz 2.11, se eliminó las correlaciones moderadas y severas.

Comparación de características entre RESUELTOS Y TIMEOUTs

Al realizar el test de normalidad de Shapiro-Wilk se obtuvo los resultados mostrados en 2.12:

Este análisis mostrado en la tabla 2.12 revela que ninguna de estas características sigue una distribución normal en los grupos de instancias que terminaron en TIMEOUT ni en las que fueron resueltas. Esto se evidencia en los valores de p muy bajos (todos menores a 0.05, incluso extremadamente cercanos a cero), lo que permite rechazar la hipótesis nula de normalidad para cada variable en ambos grupos. En

TEST DE NORMALIDAD SHAPIRO-WILK

	statistic	p-value
num_vars (TIMEOUT	0.7933191376191727	1.1794648547679767e-18
num_vars (NO TIMEOUT)	0.6698142994834743	9.428437191286474e-26
densidad (TIMEOUT)	0.6461253813664019	7.820027948180053e-24
densidad (NO TIMEOUT)	0.5441281055563653	2.297890446301959e-29
tamanio_prom_clausula (TIMEOUT)	0.7520422665672243	2.4804046463422804e-20
tamanio_prom_clausula (NO TIMEOUT)	0.8112114166434761	4.389991714803694e-20

Figura 2.12: Test de normalidad Shapiro-Wilk.

concreto, los estadísticos de Shapiro oscilan entre aproximadamente 0.54 y 0.81, y los p-valores indican una desviación significativa de la normalidad.

Luego, para comparar estas características entre instancias con y sin TIMEOUT no es apropiado utilizar pruebas paramétricas basadas en la normalidad, como el t-test, sino que se deben emplear métodos no paramétricos, por ejemplo la prueba de Mann-Whitney, cuyo resultado se presenta a continuación.

Prueba Mann-Whitney U

PRUEBA MANN-WHITNEY U

	p-value
num_vars	1.383e-83
densidad	3.168e-29
tamanio_prom_clausula	0.803

Figura 2.13: Prueba Mann-Whitney U.

Como se muestra en 2.13, la prueba muestra resultados concluyentes sobre las diferencias entre las instancias que terminaron en TIMEOUT y las que fueron resueltas. Estos, visualizarse de igual forma mediante las gráficas 2.14.

Para las variables número de variables y densidad, los valores p son extremadamente bajos (1.383e-83 y 3.168e-29 respectivamente), lo que indica diferencias estadísticamente significativas entre ambos grupos en estas características. Esto sugiere

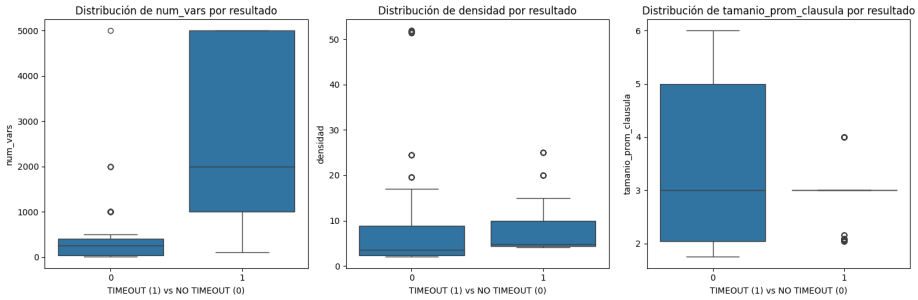

Figura 2.14: Distribuciones de prueba Mann-Whitney U por resultados.

que tanto el tamaño del problema como su densidad influyen fuertemente en la probabilidad de que un problema termine en TIMEOUT.

En contraste, para el tamaño promedio de cláusula el p-valor es alto (0.803), lo que implica que no hay evidencia suficiente para afirmar que esta característica difiera entre problemas con y sin TIMEOUT.

Estos resultados confirman que las variables relacionadas con la escala y complejidad estructural del problema (num_vars y densidad) son factores determinantes en el rendimiento del *solver*, mientras que el tamaño promedio de las cláusulas no parece ser un factor discriminante en la ocurrencia de TIMEOUT.

En consecuencia, para modelar o predecir la probabilidad de TIMEOUT conviene priorizar las variables num_vars y densidad, y descartar o tratar con menor peso el tamaño promedio de cláusula.

Regresión logística: ¿Qué características predicen TIMEOUT?

Trasa efectuar la regresión se obtuvieron los siguientes resultados 2.15:

REGRESIÓN LOGÍSTICA

	coeficiente logit
num_vars	0.0032
densidad	-0.0314
tamanio_prom_clausula	-0.5455

Figura 2.15: Regresión logística.

Con base en estos resultados de la tabla 2.15 , el modelo ofrece una interpretación clara sobre la influencia de cada característica en la probabilidad de que un problema no se resuelva en el tiempo límite.

El coeficiente positivo para num_vars (0.0032) indica que a medida que aumenta el número de variables, la probabilidad de TIMEOUT también incrementa, lo cual es consistente con la intuición y los análisis descriptivos previos que mostraron que problemas más grandes tienden a ser más difíciles de resolver.

Por otro lado, los coeficientes negativos para densidad (-0.0314) y especialmente para tamaño promedio de cláusula (-0.5455) sugieren que, manteniendo constante el número de variables, un aumento en estas características está asociado con una menor probabilidad de TIMEOUT. En particular, el tamaño promedio de cláusula tiene un efecto considerablemente más fuerte en sentido negativo, lo que podría indicar que cláusulas más largas o complejas, dentro del rango observado, facilitan la resolución o están asociadas a problemas menos propensos a agotar el tiempo.

Estos signos y magnitudes reflejan que la complejidad estructural del problema no se reduce solo al tamaño, sino que la forma en que las cláusulas están construidas también impacta el rendimiento del solver. En conjunto, el modelo confirma y cuantifica las tendencias observadas en análisis estadísticos previos, y ofrece una base para predecir y ajustar heurísticas en función de las características del problema para minimizar la ocurrencia de TIMEOUT.

2.1.4. Rendimiento de las heurísticas en problemas resueltos

Para los análisis estadísticos en esta sección se incluyó la variable tiempo, que e=indica en seg cuánto tardó el *solver* en dar una solución.

Correlación de Spearman

Se obtuvo los siguientes resultados 2.16:

CORRELACIÓN DE SPEARMAN

	rho	p-value
num_vars vs tiempo_segundos	0.845	1.116e-63
Densidad vs tiempo_segundos	0.318	8.968e-07
tamanio_prom_clausula vs tiempo_segundos	0.710	1.929e-36

Figura 2.16: Correlación de Spearman.

Los resultados que se muestran en la tabla 2.16 análisis revela asociaciones significativas y de distinta intensidad.

La variable número de variables muestra una correlación muy fuerte y positiva con el tiempo de resolución (rho = 0.845, p <0.001), indicando que a mayor tamaño del problema, mayor es el tiempo requerido para resolverlo, lo cual es coherente con la complejidad esperada en problemas SAT.

La densidad presenta una correlación positiva moderada (rho = 0.318, p <0.001), lo que sugiere que problemas con mayor densidad de cláusulas por variable tienden a requerir más tiempo, aunque este efecto es menos pronunciado que el del tamaño.

Por último, el tamaño promedio de cláusula también exhibe una correlación fuerte positiva (rho = 0.710, p <0.001), indicando que cláusulas más largas o complejas están asociadas con un aumento significativo en el tiempo de resolución.

Estos resultados confirman que tanto la escala del problema como su estructura influyen en el rendimiento del solver, siendo el número de variables el factor más determinante, seguido por la complejidad de las cláusulas y la densidad.

Kruskal-Wallis: ¿El tiempo difiere entre heurísticas para un mismo tipo de problema?

KRUSKAL-WALLIS

	VSIDS+restart	
	estadístico	p-value
num_vars	55.0704	1.738e-06
densidad	55.8711	8.922e-05
tamanio_prom_clausula	54.8239	0.0001258
	VSIDS+no-restart	
	estadístico	p-value
num_vars	54.7791	9.448e-07
densidad	55.2582	6.482e-05
tamanio_prom_clausula	55.6763	9.513e-05
	DLIS+restart	
	estadístico	p-value
num_vars	41.2553	4.441e-05
densidad	40.6674	0.001692
tamanio_prom_clausula	42.9055	0.003232
	DLIS+no-restart	
	estadístico	p-value
num_vars	46.9197	4.813e-06
densidad	47.8132	9.17e-05
tamanio_prom_clausula	48.7640	0.000332

Figura 2.17: Kruskal-Wallis.

Dados los siguientes resultados mostrados en la tabla 2.17, se tiene que el análisis mediante la prueba no paramétrica de Kruskal-Wallis aplicado a la variable transformada del tiempo de resolución (logaritmo natural del tiempo más uno) para cada heurística y característica del problema revela diferencias estadísticamente significativas en todos los casos evaluados.

Para las cuatro heurísticas (VSIDS+restart, VSIDS+no-restart, DLIS+restart y DLIS+no-restart), las variables número de variables, densidad y tamaño promedio de cláusula muestran valores de estadístico elevados y p-valores muy bajos (todos

menores a 0.005), lo que indica que la distribución del tiempo de resolución difiere significativamente entre los distintos grupos definidos por cada una de estas características. Esto implica que, independientemente de la heurística utilizada, las propiedades estructurales del problema impactan de forma relevante en el rendimiento temporal del *solver*.

La consistencia de estos resultados a través de todas las heurísticas sugiere que la influencia de estas variables es robusta y generalizable, reafirmando su importancia en el análisis y modelado del comportamiento del solver SAT. Además, el uso del logaritmo del tiempo como variable respuesta ayuda a mitigar la influencia de valores extremos y facilita la detección de diferencias significativas en la mediana y distribución del tiempo.

Pruebas post hoc con test de Dunn

Para estas pruebas se obtuvieron los resultados mostrados en las tablas 2.18, 2.19, 2.20 y 2.21:

TEST DE DUNN POR HEURÍSTICA

VSIDS+RESTART

	1	2	3	4
num_vars				
1	1.000000	0.703180	0.000015	1.06e-07
2	0.703180	1.000000	0.011745	0.000421
3	0.000015	0.011745	1.000000	1.00
4	1.06e-07	0.000421	1.00	1.000000
	1	2	3	4
densidad				
1	1.000000	0.000007	0.000503	0.070783
2	0.000007	1.000000	1.000000	0.103465
3	0.000503	1.000000	1.000000	0.872152
4	0.070783	0.103465	0.872152	1.000000
	1	2	3	4
tamanio_prom_clausula				
1	1.000000	0.123377	0.040784	7.50e-08
2	0.123377	1.000000	1.000000	0.006134
3	0.040784	1.000000	1.000000	0.022850
4	7.50e-08	0.006134	0.022850	1.000000

Figura 2.18: Test Dunn por cuartiles para VSIDS con restart.

El test post-hoc de Dunn aplicado por cuartiles (grupos definidos con ranking para evitar empates) para cada heurística y variable seleccionada permite identificar con

TEST DE DUNN POR HEURÍSTICA

VSIDS+NO-RESTART

	1	2	3	4
num_vars				
1	1.000000	0.937707	0.000237	2.45e-07
2	0.937707	1.000000	0.039659	0.000282
3	0.000237	0.039659	1.000000	1.00
4	2.45e-07	0.000282	1.00	1.000000
	1	2	3	4
densidad				
1	1.000000	0.000001	0.000050	0.108825
2	0.000001	1.000000	1.000000	0.027465
3	0.000050	1.000000	1.000000	0.197057
4	0.108825	0.027465	0.197057	1.000000
	1	2	3	4
tamanio_prom_clausula				
1	1.000000	0.189033	0.004745	3.63e-08
2	0.189033	1.000000	1.000000	0.001485
3	0.004745	1.000000	1.000000	0.108223
4	3.63e-08	0.001485	0.108223	1.000000

Figura 2.19: Test Dunn por cuartiles para VSIDS sin restart.

precisión entre qué grupos existen diferencias significativas en el tiempo de resolución (logaritmo del tiempo).

En general, para la variable "número de variables", se observan diferencias altamente significativas entre los cuartiles extremos (por ejemplo, grupo 1 vs. 4) en todas las heurísticas, con p-valores muy bajos ($p < 0.001$), lo que confirma que problemas más grandes requieren tiempos significativamente mayores para resolverse. En los grupos intermedios, las diferencias son menos consistentes pero también aparecen comparaciones significativas, evidenciando una relación gradual entre tamaño y tiempo.

Para la densidad, los resultados muestran diferencias significativas principalmente entre los cuartiles más bajos y más altos, aunque en algunos casos los grupos intermedios no difieren tanto, indicando que la densidad afecta el tiempo pero con menor fuerza y de manera menos lineal que el tamaño.

En cuanto al tamaño promedio de cláusula, también se detectan diferencias significativas entre los extremos, especialmente entre el primer y cuarto cuartil, y en varios casos entre grupos adyacentes, lo que sugiere que esta variable influye en el rendimiento, aunque su efecto puede ser más sutil o no monotónico.

La consistencia de estos patrones a lo largo de todas las heurísticas refuerza la importancia de estas características para explicar la variabilidad en el tiempo de

TEST DE DUNN POR HEURÍSTICA

DLIS+RESTART

	1	2	3	4
num_vars				
1	1.000000	1.000000	0.004494	1.18e-07
2	1.000000	1.000000	0.058306	1.02e-05
3	0.004494	0.058306	1.000000	0.166841
4	1.18e-07	1.02e-05	0.166841	1.000000
	1	2	3	4
densidad				
1	1.000000	0.024023	0.039453	0.005794
2	0.024023	1.000000	1.000000	1.000000
3	0.039453	1.000000	1.000000	1.000000
4	**0.005794**	1.000000	1.000000	**1.000000**
	1	2	3	4
tamanio_prom_clausula				
1	1.000000	0.000710	0.078823	0.000005
2	0.000710	1.000000	1.000000	1.000000
3	0.078823	1.000000	1.000000	0.098370
4	0.000005	1.000000	0.098370	1.000000

Figura 2.20: Test Dunn por cuartiles para DLIS con restart.

resolución.

TEST DE DUNN POR HEURÍSTICA

DLIS+NO-RESTART

	1	2	3	4
num_vars				
1	1.000000	0.563525	0.000105	1.98e-08
2	0.563525	1.000000	0.048111	0.000134
3	0.000105	0.048111	1.000000	0.785971
4	1.98e-08	0.000134	0.785971	1.000000

	1	2	3	4
densidad				
1	1.000000	6.86e-07	0.001024	0.010847
2	6.86e-07	1.000000	0.891933	0.174521
3	0.001024	0.891933	1.000000	1.000000
4	0.010847	0.174521	1.000000	1.000000

	1	2	3	4
tamanio_prom_clausula				
1	1.000000	0.016085	0.029254	5.33e-09
2	0.016085	1.000000	1.000000	0.010632
3	0.029254	1.000000	1.000000	0.008290
4	5.33e-09	0.010632	0.008290	1.000000

Figura 2.21: Test Dunn por cuartiles para DLIS csin restart.

Conclusiones

La presente tesis ha demostrado que, aunque el algoritmo Conflict-Driven Clause Learning (CDCL) representa un avance significativo en la resolución de problemas SAT al combinar aprendizaje de cláusulas y selección heurística de variables, persiste un dilema fundamental en la dependencia de la heurística inicial para la selección de variables. Este aspecto condiciona el rendimiento del solver, especialmente en problemas con estructuras complejas y gran número de variables, donde la probabilidad de TIMEOUT y los tiempos de resolución se incrementan notablemente, independientemente de la heurística aplicada.

Los resultados estadísticos obtenidos a través de pruebas no paramétricas, regresiones logísticas, análisis de varianza y modelos lineales confirman que el número de variables es el factor más determinante en la eficiencia del solver, seguido por la densidad de cláusulas, cuyo efecto es moderado y en algunos casos inverso, y el tamaño promedio de cláusula, cuya influencia varía según interacciones con otras características. En cuanto al desempeño de las heurísticas, no se identificó una estrategia universalmente óptima; sin embargo, la heurística DLIS sin reinicios mostró un mejor rendimiento promedio y menor tasa de TIMEOUT en problemas medianos y grandes con ciertas estructuras, mientras que las variantes basadas en VSIDS presentaron mayor variabilidad y peor desempeño en instancias densas y grandes, aunque con eficacia en problemas pequeños y medianos.

En conclusión, la investigación subraya la importancia crítica de adaptar la selección heurística y las estrategias de optimización a las características estructurales del problema para maximizar la eficiencia del solver CDCL. Los hallazgos estadísticos robustos orientan hacia el desarrollo de heurísticas adaptativas y modelos predictivos que consideren el tamaño y la estructura del problema, abriendo camino a solucionadores más inteligentes y generalizables. Este trabajo aporta una base sólida para futuras investigaciones que busquen superar el cuello de botella en la selección de variables y perfeccionar la sinergia entre aprendizaje y heurística en algoritmos de satisfacibilidad booleana.

Recomendaciones

Para futuros trabajos se recomienda lograr insertar una estructura en CaDiCaL que, de forma eficiente, almacene por cada literal las cláusulas insatisfechas en las que se encuentre. Se sugiere aumentar el *timeout* y volver a ejecutar los experimentos. Además, se recomienda continuar ampliar este estudio e incorporar nuevas heurísticas para comprobar su influencia.

Bibliografía

AlmaBetter. (2025). Constraint Satisfaction Problem in AI [Accedido: 2025-06-15]. (Vid. págs. 5, 33).

Biere, A., Fleury, M., Faller, T., Froleyks, N., Fazekas, K., & Pollitt, F. (2024). CaDiCaL 2.0 [University Freiburg, Johannes Kepler University Linz, TU Wien]. (Vid. págs. 27, 29, 30, 36).

Biere, A., & Fröhlich, A. (2019). Evaluating CDCL Restart Schemes [Partially supported by FWF, NFN Grant S11408-N23 (RiSE)]. En D. L. Berre & M. Järvisalo (Eds.), *Proceedings of Pragmatics of SAT 2015 and 2018 (POS-18)* (pp. 1-17, Vol. 59). EasyChair. (Vid. págs. 26, 27, 34).

Cai, S., Zhang, X., Fleury, M., & Biere, A. (2022). Better Decision Heuristics in CDCL through Local Search and Target Phases [Submitted 01/2022; published 08/2022]. *Journal of Artificial Intelligence Research*, *74*, 1515-1563 (vid. págs. 27-29).

Cherif, M. S., Habet, D., & Terrioux, C. (2021). Combining VSIDS and CHB Using Restarts in SAT [Article No.20; Licensed under Creative Commons License CC-BY 4.0]. En L. D. Michel (Ed.), *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)* (20:1-20:19). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany. (Vid. págs. 26, 27, 34).

Cherif, M. S., Habet, D., & Terrioux, C. (2024). Combining VSIDS and CHB Using Restarts in SAT. *Proceedings of a Conference or Workshop (details not specified)* (vid. pág. 34).

Fichte, J. K., Berre, D. L., Hecher, M., & Szeider, S. (2023). The Silent (R)evolution of SAT. *Communications of the ACM*, *66*(6), 54-63. https://doi.org/10.1145/3581777 (vid. pág. 6).

Ganesh, V., & Vardi, M. Y. (s.f.). On The Unreasonable Effectiveness of SAT Solvers [No se especifica año ni fuente en la información proporcionada] (vid. págs. 32, 34, 35).

Garrido, L. G. (2024). LC-CSP_Conferencia #1 [Profesor Titular Consultante]. (Vid. pág. 5).

Garrido, L. G. (2025). El problema de la Satisfacibilidad (SAT) [Capítulo educativo]. (Vid. págs. 6-10).

Guo, M. (2024). Progress in the Study of the Boolean Satisfiability Problem [© Content licensed under CC BY-NC 4.0]. *International Journal of Computer Science and Information Technology*, *2*(2). https://doi.org/10.62051/ijcsit.v2n2.25 (vid. pág. 6).

Hoos, H. H. (1998). SAT and Constraint Satisfaction [Available online: https://www.cs.ubc.ca/~hoos/SLS-Internal/ch6.pdf]. En H. H. Hoos & T. Stützle (Eds.), *Stochastic Local Search: Foundations and Applications* (pp. 203-260). Morgan Kaufmann. https://doi.org/10.1016/B978-1-55860-508-4.50012-4 (vid. págs. 31-33).

Iser, M., & Balyo, T. (2021). Unit Propagation with Stable Watches [Category: Short Paper; Supplementary Material: https://github.com/sat-clique/cadical_stability]. *Proceedings of the 2021 International Conference on Principles and Practice of Constraint Programming (CP 2021)*. https://doi.org/10.4230/LIPIcs.CP.2021.6 (vid. pág. 26).

Li, C., Liu, C., Chung, J., Lu, Z., Jha, P., & Ganesh, V. (2024). A Reinforcement Learning based Reset Policy for CDCL SAT Solvers. *42nd Conference on Very Important Topics (CVIT 2016)*. https://doi.org/10.4230/LIPIcs.CVIT.2016.23 (vid. pág. 34).

Luo, M., Li, C.-M., Xiao, F., Manyà, F., & Lü, Z. (2017). An Effective Learnt Clause Minimization Approach for CDCL SAT Solvers. *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI-17)*, 703-709 (vid. pág. 34).

Marques-Silva, J., Lynce, I., & Malik, S. (2024). Conflict-Driven Clause Learning SAT Solvers [CAV 2024, Montreal, QC, Canada]. En A. Gurfinkel & V. Ganesh (Eds.), *Computer Aided Verification* (pp. ---, Vol. 14681). Springer. https://doi.org/10.1007/978-3-031-65627-9 (vid. págs. 5, 6, 27, 28, 34).

Nandi, A., Chakrabartty, S., & Thakur, C. S. (2024). Margin Propagation based XOR-SAT Solvers for Decoding of LDPC Codes [arXiv:2402.04959v]. *arXiv preprint arXiv:2402.04959* (vid. pág. 33).

Oh, C. (2016, enero). *Improving SAT Solvers by Exploiting Empirical Characteristics of CDCL* [Ph.D. dissertation]. New York University. (Vid. págs. 32, 34).

Oliveras, A., & Rodríguez-Carbonell, E. (2009). From DPLL to CDCL SAT Solvers [Fall 2009]. (Vid. págs. 12, 13, 19-23, 26).

Su, Y., Yang, Q., Ci, Y., Li, Y., Bu, T., & Huang, Z. (2025). Deeply Optimizing the SAT Solver for the IC3 Algorithm. *arXiv preprint*, *arXiv:2501.18612*. https://arxiv.org/abs/2501.18612 (vid. pág. 27).

Sun, Y., Ye, F., Zhang, X., Huang, S., Zhang, B., Wei, K., & Cai, S. (2024). AutoSAT: Automatically Optimize SAT Solvers via Large Language Models [ar-

Xiv:2402.10705v3 [cs.AI]]. *Journal of Artificial Intelligence Research*, *1*, 1-37. https://arxiv.org/abs/2402.10705 (vid. pág. 28).

Trimoska, M., Ionica, S., & Dequen, G. (2020). Parity (XOR) Reasoning for the Index Calculus Attack [arXiv:2001.11229v1 [cs.CR], submitted January 30, 2020]. *arXiv preprint arXiv:2001.11229* (vid. pág. 33).

Zhang, X., Chen, Z., & Cai, S. (2024). Revisiting Restarts of CDCL: Should the Search Information be Preserved? [arXiv:2404.16387v2 [cs.LO], submitted May 28, 2024; Licensed under CC-BY 4.0]. *42nd Conference on Very Important Topics (CVIT 2016)*, 23:1-23:17 (vid. págs. 26, 27).

Zulkoski, E. (2018). *Understanding and Enhancing CDCL-based SAT Solvers* [Ph.D. dissertation]. University of Waterloo. (Vid. págs. 5, 23, 30, 32).