Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN COMPUTACIÓN

Resolución de problemas MaxSAT a través de Evolución Diferencial

Estudante: Manuel Framil de Amorín
Dirección: José Pedro Cabalar Fernández
José Santos Reyes

A Coruña, xuño de 2021.

Sólo podemos ver un poco del futuro, pero lo suficiente para darnos cuenta de que hay mucho por hacer.

Alan M. Turing

Agradecimientos

En primer lugar, me gustaría agradecer a mis tutores, Pedro Cabalar y José Santos, por el esfuerzo que han dedicado y lo mucho que me han ayudado durante la realización de este trabajo. Quisiera dar las gracias también a los organizadores de la Evaluación MaxSAT, en especial a Fahiem Bacchus, por atenderme y preocuparse por mi trabajo, pese a no tener la obligación de hacerlo. Finalmente, agradecer a mi familia, amigos y compañeros que han estado apoyándome durante toda la carrera, y sobre todo en este último año.

Resumen

En este proyecto se ha desarrollado un algoritmo capaz de resolver el problema MaxSAT empleando un algoritmo evolutivo híbrido o memético, que combina el algoritmo evolutivo de Evolución Diferencial con GSAT y RandomWalk, dos heurísticas de búsqueda local específicas de MaxSAT. El algoritmo desarrollado ha sido empleado para resolver benchmarks recientes de la Evaluación MaxSAT 2020. Se ha comparado el funcionamiento del algoritmo con el de los mejores solvers presentados en la Evaluación MaxSAT 2020, alcanzando el estado del arte tanto en la calidad de las soluciones como en el tiempo de cómputo requerido para obtenerlas.

Abstract

In this project, an algorithm capable of solving the MaxSAT problem has been developed using a hybrid evolutionary or memetic algorithm, which combines the evolutionary algorithm of Differential Evolution with GSAT and RandomWalk, two MaxSAT-specific local search heuristics. The algorithm developed has been used to solve recent benchmarks of the MaxSAT Evaluation 2020. The performance of the algorithm has been compared with that of the best solvers presented in the MaxSAT Evaluation 2020, reaching the state of the art both in the quality of the solutions and in the computing time required to obtain them.

Palabras clave:

- MaxSAT
- Evolución Diferencial
- Algoritmo Memético
- GSAT
- RandomWalk
- Evaluación MaxSAT

Keywords:

- MaxSAT
- Differential Evolution
- Memetic Algorithm
- GSAT
- RandomWalk
- MaxSAT Evaluation

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

El problema de satisfactibilidad booleana (SAT) es un problema de decisión central en Inteligencia Artificial y Complejidad Computacional, y consiste en determinar si una fórmula proposicional es cierta o falsa. No obstante, en muchas ocasiones no sólo interesa saber si es satisfactible o no, sino cómo de cerca está de serlo. A esta pregunta intenta responder el problema MaxSAT, una extensión del problema SAT que combina los campos de la lógica proposicional con la optimización matemática.

La resolución del problema SAT (y su extensión a MaxSAT) es uno de los campos de investigación más antiguos y longevos de la informática. Los primeros intentos se remontan a la década de 1960, cuando Davis, Logeman y Loveland [1] propusieron el algoritmo DPLL, un método de *backtracking* para determinar la satisfactibilidad de una fórmula de lógica proposicional, basándose en el que previamente habían desarrollado Davis y Putman [2]. Desde entonces, numerosos algoritmos se han ido desarrollando para resolver instancias de SAT, pero no fue hasta 1992 cuando se realizó la primera competición de herramientas de resolución de SAT, que tuvo lugar en la Universidad de Paderborn [3]. Desde 2002 esta competición se realiza bianualmente [4], y desde 2006 acoge las evaluaciones de *solvers* de MaxSAT [5]. Aunque en un principio no se contemplaban, a partir de la edición de 2012 se incluye una categoría para resolutores incompletos en la Evaluación MaxSAT, es decir, para aquellos algoritmos que no proporcionan necesariamente una solución óptima, pero sí una buena solución en un lapso de tiempo corto.

Una de las razones de que se hayan dedicado tantos esfuerzos a resolver los problemas SAT y MaxSAT es la cantidad de problemas que se pueden codificar directamente como fórmulas proposicionales. Aparte, existe también una gran cantidad de problemas que pueden ser transformados (o reducidos) a problemas de decisión de manera eficiente. Por ejemplo, el Problema del Viajero (TSP, de sus siglas en inglés) es un problema de optimización que consiste en: dada una lista ciudades, determinar la ruta más corta para recorrerlas todas exactamente una vez. Este problema se puede reformular para tratarlo como una secuencia de problemas

de decisión, de manera que en cada paso se debe decidir si existe una ruta más corta o no. En general, cualquier problema que pertenezca a la clase NP (*nondeterministic polynomial time*) puede ser expresado como una instancia de SAT o MaxSAT, a la que pertenecen ambos. Por tanto, cualquier avance que se realice en la dirección de solucionar estos problemas de una manera eficiente y rápida está directamente relacionado con el problema *P vs NP*, considerado por muchos como el problema más importante de la informática teórica, y que fue nombrado como uno de los siete problemas del milenio por el *Clay Mathematics Institute* (CMI).

La mayoría de problemas del mundo real involucran una componente de optimización, lo que ha provocado que en los últimos años haya aumentando enormemente la demanda de aproximaciones automáticas para encontrar buenas soluciones a problemas difíciles de aproximar (desde un punto de vista computacional). Muchos de esos problemas del mundo real pueden ser codificados como un problema SAT o MaxSAT, y a pesar de que estos problemas son NP, existen algoritmos para resolverlos de manera muy rápida (además de un gran interés por seguir mejorando estos algoritmos). Esto ha motivado el desarrollo de numerosas herramientas que codifican problemas industriales como instancias de SAT o MaxSAT. Por ejemplo, SAT se ha empleado para resolver problemas de verificación formal mediante comprobación por modelos [6], planificación en Inteligencia Artificial [7] y gestión de paquetes software [8], entre otras muchas aplicaciones [9]. En cuanto a MaxSAT, se encuentran aplicaciones en inferencia probabilística [10], depuración de diseños [11], visualización [12], ciberseguridad [13], localización de fallos [14] o agrupación de datos correlacionados (correlation clustering) [15], entre otras muchas.

El algoritmo de Evolución Diferencial (ED) es un método evolutivo que trata de optimizar un problema iterativamente, manteniendo una población de individuos que representan soluciones candidatas. ED trata de imitar la evolución de las especies, codificando el problema a resolver en los "genes" de los individuos, y aplicando sobre ellos operaciones de cruce y mutación. Una de las principales ventajas de ED frente a otros algoritmos evolutivos es su relativa sencillez: es un algoritmo fácil de entender e implementar, con pocos parámetros definitorios y en el que resulta simple introducir modificaciones para adaptarlo al dominio del problema que se pretende resolver y permite realizar hibridaciones con otros algoritmos para refinar las soluciones. Por otro lado, ED emplea una cantidad constante de memoria, directamente proporcional al tamaño de la población de individuos que mantenga. Esto resulta ventajoso, ya que los métodos tradicionales de resolución de MaxSAT (como *Branch & Bound*) suelen estar limitados a espacios de búsqueda pequeños debido a que requieren una cantidad de memoria exponencial. Otra gran diferencia con otros métodos de búsqueda local en MaxSAT es que ED realiza un proceso de búsqueda global, gracias a que cada individuo de la población de soluciones se encuentra en un punto distinto del espacio de búsqueda. Es decir, ED incorpora (implícitamente) mecanismos que le permiten escapar de los máximos locales y las mesetas,

dos de los problemas centrales en los algoritmos de optimización.

1.1 Objetivos

Teniendo el cuenta la importancia del problema MaxSAT y las ventajas que ofrece el algoritmo de Evolución Diferencial, se ha establecido como objetivo principal de este trabajo construir una herramienta de resolución aproximada de MaxSAT basada en ED, para lo que se deberán cumplir una serie de objetivos específicos:

- Adaptación de ED al problema MaxSAT, usando una versión de ED que utiliza genotipos binarios.
- Aplicación en benchmarks MaxSAT.
- Comparación con otras herramientas de resolución de MaxSAT.
- Estudio de posibles hibridaciones entre ED y las anteriores herramientas.

1.2 Organización del trabajo

El trabajo se organiza de la siguiente manera:

En el Capítulo 2 se describen los antecedentes y fundamentos sobre los que se basa el trabajo. La sección 2.3 trata los fundamentos de la lógica proposicional y el problema SAT. En las secciones 2.4 y 2.5 se detalla el problema MaxSAT y el estado del arte de los resolutores de dicho problema. Por último, en la 2.6 se expone el algoritmo de Evolución Diferencial y su adaptación para trabajar en un dominio discreto.

El Capítulo 3 proporciona una descripción precisa del funcionamiento del resolutor MaxSAT desarrollado, bautizado como DEMaxSATSolver, así como los detalles de implementación más relevantes.

La aplicación del DEMaxSATSolver a benchmarks MaxSAT se describe en el Capítulo 5. Concretamente, en la Sección 5.1 se discuten los diferentes métodos propuestos, mientras que en la Sección 5.2 se muestran los resultados obtenidos por el resolutor desarrollado en comparación con otros resolutores presentados en la Evaluación MaxSAT de 2020 (MSE).

Finalmente, en el capítulo 6 se abordan las conclusiones finales del trabajo, así como las limitaciones encontradas y posibles mejoras a incorporar en un futuro.

Capítulo 2

Antecedentes

En este capítulo se describen los fundamentos (definiciones, métodos y técnicas) necesarios para la comprensión del trabajo desarrollado.

2.1 Complejidad Computacional

En 1928, David Hilbert y Wilhelm Ackermann propusieron el Entscheidungsproblem ("Problema de decisión"), un reto a la comunidad científica de la época, que consistía en encontrar un algoritmo general que, dada una fórmula de primer orden, determine si es universalmente válida (es decir, si es un teorema[1]).

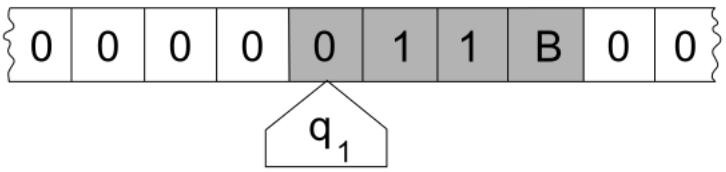

Figura 2.1: Representación de una Máquina de Turing

A pesar de que se realizaron muchos esfuerzos en encontrar una solución positiva a esta pregunta, Alonzo Church [16] y Alan Turing [17] demostraron de manera independiente que este problema era irresoluble. Para demostrarlo, ambos necesitaron definir modelos de computación universales: Church lo hizo introduciendo su λ-Calculus, mientras que Turing empleó la llamada Máquina de Turing. Ambos modelos son equivalentes (i.e. pueden resolver exactamente los mismos problemas), ya que podemos representar cualquier Máquina de Turing (MT) en λ-Calculus, por lo que cualquier problema que se pueda computar en una MT también se puede en λ-Calculus, y viceversa.

[1] Según el Teorema de Completitud de Gödel, un teorema debe ser probado a partir de los axiomas de la lógica correspondiente.

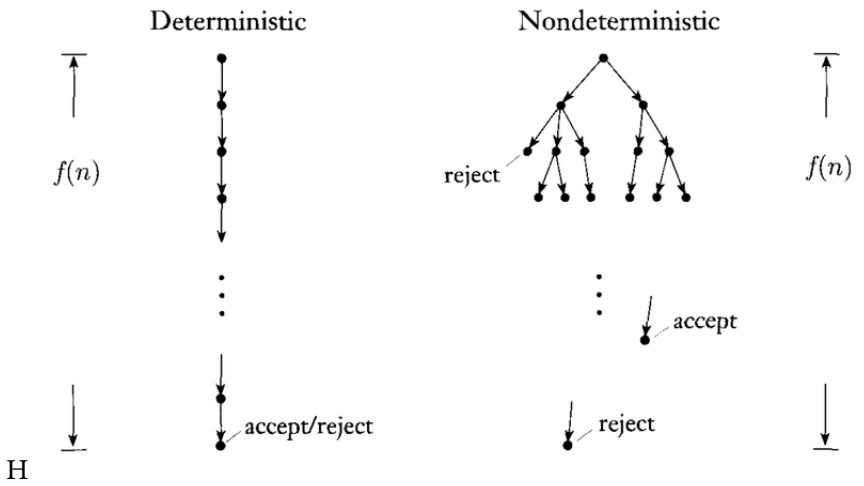

Figura 2.2: MT Determinista VS MT no determinista

En la Figura 2.1 se muestra una MT tal y como fue descrita por Turing. La máquina consiste en una cinta infinita divida en "celdas" discretas, que guardan un símbolo, y una cabecera que se posiciona sobre de esas celdas. El funcionamiento es el siguiente: (i) la cabecera lee (o escanea) el símbolo de la celda sobre la que está situado, (ii) escribe un símbolo en la celda y (iii) desplaza la cinta hacia la derecha o la izquierda. Finalmente (iv) la máquina decide si procesa otra instrucción o se detiene ("halts").

Este modelo se puede extender de muchas formas, por ejemplo, permitiendo que la máquina no realice ningún movimiento tras escribir el símbolo, introduciendo múltiples cintas (*multi-tape MT*), múltiples cabeceras en la misma cinta, o múltiples cabeceras que se mueven independientemente en múltiples cintas (*multi-track MT*). No obstante, todas estas modificaciones son equivalentes a la Máquina de Turing original, en cuanto a los problemas que pueden resolver. Se dice que un lenguaje es *Turing-Completo* si es teóricamente capaz de expresar cualquier problema que pueda ser simulado en una Máquina de Turing[2].

Un caso especial son las Máquinas de Turing No Deterministas (NDTM, de sus siglas en inglés) que pueden realizar más de una acción en cada paso. Es decir, en una Máquina de Turing determinista cada acción está completamente determinada por el estado en el que se encuentra y la entrada que recibe, mientras que en una NDTM, esto no es cierto. Igual que sucedía con el resto de modificaciones, cualquier problema que pueda ser solucionado por una NDTM puede ser resuelto también por una MT determinista. Sin embargo, a pesar de que ambas tengan la misma capacidad computacional, es posible que no tengan la misma complejidad temporal. En la Figura 2.2 se puede ver el proceso que realiza una MT Determinista y una NDMT para resolver un problema de decisión. En cada paso, la NDTM puede generar

[2] Casi todos los lenguajes de programación modernos son Turing-Completos si se ignoran las limitaciones de memoria (pues es finita)

distintas ejecuciones (tantas como quiera) para la misma entrada, mientras que la determinista sólo tiene un "camino" de ejecución. Se cree, por tanto, que las Máquinas de Turing no deterministas son capaces de resolver en tiempo polinomial[3] problemas que requieren tiempo no polinomial a una MT Determinista. No obstante, esto es una conjetura, ya que demostrarlo resultaría equivalente a demostrar que $P \neq NP$.

Una máquina oráculo (*oracle machine*) es máquina abstracta que sirve para el estudio de problemas de decisión. Se puede ver como una Máquina de Turing con una caja negra, denominada *oráculo*, que es capaz de resolver un cierto problema de decisión en un sólo paso. Las máquinas oráculo permiten definir nuevas clases de complejidad para problemas de decisión: los problemas que se pueden resolver por algoritmos en la clase A con un oráculo para el lenguaje L pertenecen a la clase A^L. Por ejemplo, a la clase P^{SAT} pertenecen aquellos problemas que se pueden resolver en tiempo polinomial por una MT Determinista con un oráculo para SAT.

Un problema de decisión pertenece a la clase NP (*nondeterministic polynomial time*) si, dada una instancia cuya respuesta sea "Sí", puede ser verificada por una MT Determinisita en tiempo polinomial. Una definición alternativa: pertenecen a la clase NP aquellos problemas que pueden ser resueltos por una Máquina de Turing No Determinista en tiempo polinomial. Informalmente, los problemas en NP son muy complejos de resolver, pero una vez obtenida una solución, esta es muy fácil de verificar.

Por contra, en la clase P se encuentran todos los problemas resolubles (deterministicamente) en tiempo polinomial. Teniendo en cuenta esto, es fácil ver que $P \subseteq NP$, ya que si un problema es resoluble por una MT Determinista, también lo será por una no determinista.

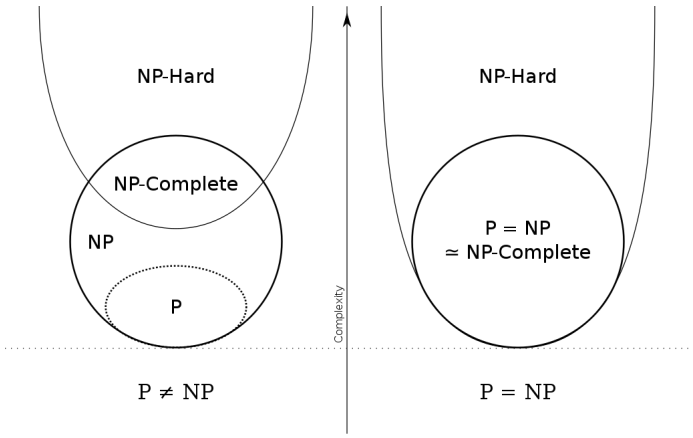

Figura 2.3: Diagrama sobre P, NP, NP-Completo y NP-Hard

Un problema A es *NP-Hard* (o NP-Difícil) si todos los problemas en NP pueden ser reduci-

[3] "Tiempo polinomial" se refiere a que el número de operaciones que necesita un algoritmo crece de manera lineal respecto al tamaño del problema.

dos[4] a A en tiempo polinomial. Si además de ser NP-Hard, el problema A pertenece a la clase NP, se dice que dicho problema es *NP-Completo*.

Cualquier problema de optimización puede expresarse como un problema de decisión: dada una instancia x y un entero k, decidir cuándo la solución óptima es mayor (menor, en los problemas de minimización) o igual que k. Por tanto, se puede resolver un problema de optimización mediante n problemas de decisión. Teniendo en cuenta esto, los problemas de optimización que se pueden reducir a problemas de decisión NP se clasifican como NP-Optimization problems (NPO, de aquí en adelante) y cumplen las siguientes propiedades [18]:

1. Dada una entrada, se puede verificar rápidamente que es una instancia válida del problema.

2. Dado una solución, se puede verificar eficientemente que es una solución válida.

3. La función objetivo se puede computar en tiempo polinomial (dada una instancia y una solución del problema). Es decir, el valor de una solución es fácil de calcular.

4. El tamaño de las posibles soluciones está limitado polinomialmente al tamaño de las entradas.

En relación con la complejidad de los problemas de optimización, se define la clase *APX* como el conjunto de problemas NPO que admiten algoritmos que aproximan la solución óptima a un factor constante, en tiempo polinomial [18]. Por otro lado, los algoritmos PTAS son aquellos que, dado un parámetro fijo ϵ, permiten generar en tiempo polinomial una solución que se acerque al óptimo con una desviación del orden de $1 + \epsilon$. Un problema B se considera APX-Hard si existe una redución PTAS[5] de todos los problemas APX a B. Asimismo, un problema es APX-Completo si es APX-Hard y además pertenece a APX. Esto conlleva que $PTAS \neq APX$, por lo que ningún problema APX-Hard tiene PTAS (a no ser que $P = NP$).

Finalmente, igual que existen las clases P y NP para problemas de decisión, se pueden definir las clases análogas para relaciones binarias: FP y FNP, respectivamente.

2.2 Lógica Proposicional

La *Lógica Clásica Proposicional* es posiblemente la lógica existente más sencilla y conocida. Su sintaxis parte de un conjunto llamado *signatura* At que se compone de *átomos* o *proposiciones* cuyo valor puede ser únicamente *cierto* (1) o *falso* (0). Una *fórmula bien formada* φ se

[4] Una *reducción* es un algoritmo para transformar un problema en otro.

[5] Una reducción PTAS es una reducción entre soluciones de problemas de optimización que mantiene la distancia al óptimo y preserva la propiedad PTAS del problema.

construye siguiendo la gramática:

$$\varphi ::= p \mid \bot \mid \top \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid (\varphi)$$

donde $p \in At$ representa cualquier átomo y las conectivas $\neg$, $\wedge$ y $\vee$ representan la negación, conjunción y disyunción, respectivamente. Las conectivas binarias son asociativas por la izquierda, la disyunción tiene menor prioridad que la conjunción, y esta, a su vez, menor prioridad que la negación. Las constantes $\bot$ y $\top$ representan los valores de certeza *falso* y *cierto*, respectivamente. Otros operadores comunes son la implicación $\varphi \to \psi$, definida como $\neg\varphi \vee \psi$, y la equivalencia $\varphi \leftrightarrow \psi$, que se define como $(\varphi \to \psi) \wedge (\psi \to \varphi)$. Una *teoría* Γ se define como un conjunto de fórmulas bien formadas. Llamamos *literal* a cualquier fórmula de las formas p (literal *positivo*) o $\neg p$ (literal negativo) para cualquier átomo $p \in At$.

La semántica de lógica proposicional se define en función del concepto de *interpretación*. Una *interpretación* I es una función $I : At \longrightarrow \{0, 1\}$ que asigna, a cada proposición $p \in At$ un valor de certeza $I(p) \in \{0, 1\}$. Otra forma habitual de representar una interpretación es usando un conjunto de átomos $I \subseteq At$ que son aquellos a los que la interpretación asigna un 1, es decir, los hace ciertos.

La función de interpretación I se puede ampliar a cualquier fórmula φ, de modo que se asigna un valor $I(\varphi) \in \{0, 1\}$ siguiendo el siguiente método: primero sustituimos en φ cada átomo p por su valor $I(p)$ en la interpretación, y luego aplicamos las siguientes tablas bien conocidas del álgebra de Boole

φ	$\neg\varphi$
0	1
1	0

φ	ψ	$\varphi \wedge \psi$
0	0	0
0	1	0
1	0	0
1	1	1

φ	ψ	$\varphi \vee \psi$
0	0	0
0	1	1
1	0	1
1	1	1

además de los valores constantes $I(\bot) = 0$ e $I(\top) = 1$. Se dice que una interpretación I *satisface* una fórmula φ, escrito $I \models \varphi$, cuando $I(\varphi) = 1$. Una interpretación I es un *modelo* de una teoría Γ si satisface todas las fórmulas de Γ, esto es, $I \models \varphi$ para toda $\varphi \in \Gamma$. Una fórmula φ es *satisfactible* cuando existe al menos una interpretración I que la satisface, es decir, cuando tiene al menos un modelo. Decimos que una fórmula φ es *válida* o es una *tautología* cuando $I \models \varphi$ para cualquier posible interpretación $I \subseteq At$. Habitualmente, escribimos $\models \varphi$ para indicar que φ es una tautología. Por ejemplo, es fácil ver que $\models p \vee \neg p$, ya que cualquier interpretación hará cierta una de las dos partes de la disyunción. Por el contrario, decimos que una fórmula φ es *inconsistente* o *insatisfactible* cuando toda interpretación la hace falsa, es decir, $I(\varphi) = 0$ para toda interpretación $I \subseteq At$. En otras palabras, φ no tiene modelos.

Por ejemplo, $p \wedge \neg p$ es insatisfactible ya que toda interpretación hará falsa una de sus dos subfórmulas. Dos fórmulas φ y ψ son *equivalentes*, y lo escribimos $\varphi \equiv \psi$, cuando tienen los mismos modelos, es decir, $I \models \varphi$ si y sólo si $I \models \psi$, para toda interpretación I. En lógica proposicional, esto es lo mismo que comprobar que $\models \varphi \leftrightarrow \psi$, es decir, que la doble implicación sea una tautología.

Denominamos *cláusula* (disyuntiva) a una disyunción de literales, es decir, a una fórmula del estilo $L_1 \vee \cdots \vee L_n$ con $n \geq 0$ donde cada L_i es un literal. Por ejemplo, la fórmula $p \vee \neg q \vee r$ es una cláusula con dos literales positivos (p, r) y un literal negativo ($\neg q$). Cuando $n = 1$ la cláusula contiene un único literal (no contiene disyunción) y se dice que es *unitaria*. En ocasiones se admite también $n = 0$ y la cláusula se denomina *vacía* siendo equivalente a la fórmula $\bot$, es decir, una inconsistencia.

Decimos que una fórmula está en *forma normal conjuntiva* (en inglés, *Conjunctive Normal Form* o CNF) cuando tiene la forma de una conjunción de cláusulas. Por ejemplo, la fórmula

$$(p \vee \neg q \vee r) \wedge (p \vee \neg r) \wedge (\neg p \vee \neg q) \wedge q \tag{2.1}$$

tiene cuatro cláusulas: nótese que la de más a la derecha es la cláusula unitaria q.

Cualquier fórmula proposicional puede convertirse a una fórmula equivalente en CNF mediante la aplicación de las siguientes transformaciones. En un primer paso, se reemplazan los operadores derivados $\rightarrow$ y $\leftrightarrow$ por sus definiciones. Por ejemplo, en la siguiente fórmula tenemos:

$$\begin{aligned}
& (p \leftrightarrow \neg q) \rightarrow \neg(r \wedge \neg s) \\
\equiv\ & (p \rightarrow \neg q) \wedge (\neg q \rightarrow p) \rightarrow \neg(r \wedge \neg s) \\
\equiv\ & (\neg p \vee \neg q) \wedge (\neg\neg q \vee p) \rightarrow \neg(r \wedge \neg s) \\
\equiv\ & \neg((\neg p \vee \neg q) \wedge (q \vee p)) \vee \neg(r \wedge \neg s)
\end{aligned}$$

A continuación, se aplican de forma exhaustiva las leyes de De Morgan:

$$\begin{aligned}
\neg(\varphi \wedge \psi) &\equiv \neg\varphi \vee \neg\psi \\
\neg(\varphi \vee \psi) &\equiv \neg\varphi \wedge \neg\psi
\end{aligned}$$

y se eliminan las dobles negaciones $\neg\neg\varphi \equiv \varphi$ hasta que la negación quede únicamente aplicada a átomos. En el ejemplo, continuaríamos con:

$$\begin{aligned}
\equiv\ & \neg(\neg p \vee \neg q) \vee \neg(q \vee p) \vee \neg(r \wedge \neg s) \\
\equiv\ & (\neg\neg p \wedge \neg\neg q) \vee (\neg q \wedge \neg p) \vee (\neg r \vee \neg\neg s) \\
\equiv\ & (p \wedge q) \vee (\neg q \wedge \neg p) \vee \neg r \vee s
\end{aligned}$$

Por último, se aplica la ley de distributividad:

$$\varphi \vee (\varphi \wedge \psi) \quad \equiv \quad (\varphi \vee \psi) \wedge (\varphi \vee \psi)$$

hasta obtener una conjunción de disyunciones de literales. Continuando con el ejemplo, obtendríamos:

$$\equiv \quad (p \vee \neg q \vee \neg r \vee s) \wedge (p \vee \neg p \vee \neg r \vee s) \wedge (q \vee \neg q \vee \neg r \vee s) \wedge (q \vee \neg p \vee \neg r \vee s)$$

De estas cuatro cláusulas, en realidad, podemos observar que las dos de en medio contienen un literal y su contrario ($p \vee \neg p$ forma parte de la segunda cláusula y $q \vee \neg q$ de la tercera) por lo que ambas son tautológicas y se pueden eliminar de la conjunción. Esto reduce el conjunto final de cláusulas a las dos siguientes:

$$(p \vee \neg q \vee \neg r \vee s) \wedge (q \vee \neg p \vee \neg r \vee s) \tag{2.2}$$

En general, este último paso de aplicación de distributividad puede producir una explosión combinatoria que hace que la reducción de una fórmula a CNF tenga complejidad al menos NP (NP-hard). Sin embargo, hoy en día existen técnicas [19] que permiten reducir cualquier fórmula a CNF en tiempo polinomial, a costa de introducir nuevos átomos auxiliares. En el resto de la memoria, supondremos que partimos siempre de una fórmula proposicional que ya está expresada en CNF.

2.3 Satisfactibilidad Proposicional (SAT)

El problema SAT, o problema de satisfactibilidad proposicional, es un problema de decisión que consiste en determinar si una fórmula proposicional (habitualmente en CNF) es satisfactible, esto es, tiene al menos un modelo. Este fue el primer problema en ser categorizado como NP-Completo, en 1971 por Stephen Cook [20], e independientemente por Leonid Levin en 1973 [21], dando paso al conocido como Teorema Cook-Levin. Esto conlleva que, a no ser que $P = NP$, cualquier algoritmo SAT que se construya requerirá tiempo exponencial en el peor caso.

Existe además una competición anual (*SAT competition*[6]) donde se comparan distintos resolutores participantes sobre un conjunto de casos de prueba de referencia (en inglés *benchmarks*). Estos casos de prueba están clasificados en distintas categorías y siguen un formato de entrada estándar denominado DIMACS.

El formato DIMACS es un estándar para representar fórmulas booleanas en CNF como

[6] http://www.satcompetition.org/

texto plano ASCII, de forma que puedan ser leidas y procesadas por un programa informático. Los ficheros comienzan con una línea de parámetros (*p-line*) con la forma *p cnf <variables> <clauses>*, donde *<variables>* y *<clauses>* denotan, respectivamente, el número de variables y cláusulas del problema. Después de la *p-line* se encuentran las cláusulas de la fórmula. Los literales de las cláusulas se representan con enteros (distintos de cero) en función de su índice en la fórmula (el primer literal recibe el 1, el segundo el 2, ...), de forma que los positivos representan los literales positivos, y los negativos, el literal negado. El final de una cláusula se marca con un cero ("0"). Cada cláusula acostumbra a aparecer en una línea separada.

Además, se puede añadir líneas de comentarios, que comienzan con 'c', y suelen ir al principio, antes de la línea de parámetros.

Por ejemplo, la fórmula (2.1) se representaría de la siguiente forma:

```
c
c Primer ejemplo formato DIMACS
c
p cnf 3 4
1 -2 3 0
1 -3 0
-1 -2 0
2 0
```

Mientras que la fórmula (2.2) tendría el siguiente aspecto:

```
c
c Segundo ejemplo formato DIMACS
c
p cnf 4 2
1 -2 -3 4 0
-2 -1 -3 4 0
```

2.4 MaxSAT

El problema de *máxima satisfactibilidad booleana* (MaxSAT de aquí en adelante) combina la lógica proposicional con la optimización matemática. MaxSAT es una generalización del problema SAT, en la que ya no se busca una interpretación que satisfaga todas las cláusulas de una fórmula CNF (que habitualmente es insatisfactible, es decir, no tiene modelos), sino que se pretende maximizar el número de cláusulas satisfechas en dicha fórmula (o minimizar las insatisfechas).

MaxSAT pertenece a la clase de problemas $FP^{NP} - Completo$, es decir, pertenece a la clase de relaciones binarias $f(x, y)$ donde dado x se puede calcular y en tiempo polinomial con una máquina oráculo que resuelva problemas NP. Esto es, puede ser resuelto mediante

una cantidad polinomial de llamadas al oráculo[7] (esta característica es la que explotan los resolutores basados en SAT, véase 2.5.2). Además, MaxSAT es muy difícil de aproximar: es un problema APX-Completo, lo que significa que admite una aproximación con factor constante, pero no tiene PTAS.

Variantes o ampliaciones de MaxSAT

Al problema MaxSAT se le pueden añadir modificaciones. Las extensiones más relevantes son las que se describen a continuación:

- **Weighted-MaxSAT**: Se añade un peso a cada cláusula, de manera que ahora el objetivo es maximizar la suma de los pesos de las cláusulas ciertas (o minimizar la suma de las falsas). Cabe destacar que si se fijan los pesos de todas las cláusulas a 1, el problema se reduce al original.

- **Partial-MaxSAT (PMS)**: Dado un problema MaxSAT, se establece la restricción de que algunas de las cláusulas deban ser satisfechas obligatoriamente. Estas cláusulas se denominan *hard*, mientras que las que no son obligatorias se denominan *soft*. El objetivo es encontrar una asignación de verdad que satisfaga todas las clásulas *hard* y maximice el número de *soft*. El problema MaxSAT original puede considerarse como una instancia de este problema donde todas las cláusulas son *soft*, mientras que si consideramos todas las cláusulas como *hard*, estaríamos en un problema SAT.

Ambas extensiones se pueden combinar, de modo que cada cláusula *soft* tienen un peso asociado, y las *hard* tendrán un peso $+\infty$.

2.4.1 La Evaluación MaxSAT

Asociada a las competiciones SAT, desde el año 2006 se celebra anualmente la Evaluación MaxSAT (MSE) [5], el mayor evento de estilo competitivo en el paradigma de resolutores MaxSAT. Su principal objetivo es evaluar el estado del arte de las herramientas *open-source* de resolución del problema MaxSAT, y mostar cómo MaxSAT es una opción viable para resolver una gran variedad de problemas de optimización. Además, con esta evaluación también se pretende coleccionar y redistribuir un conjunto de benchmarks MaxSAT, que estén disponibles para cualquiera y sirvan para el desarrollo de estudios científicos y futuras evaluaciones. En la Figura 2.4 se pueden ver los resultados de la MSE 2020 en el conjunto de benchmarks unweighted.

La eficiencia de los resolutores MaxSAT no ha parado de crecer: mientras que a principios de los 90 eran capaces de lidiar con apenas 100 variables y 200 cláusulas, en la MSE 2020 se

[7] En la práctica, los resolutores SAT suelen actuar como oráculos NP.

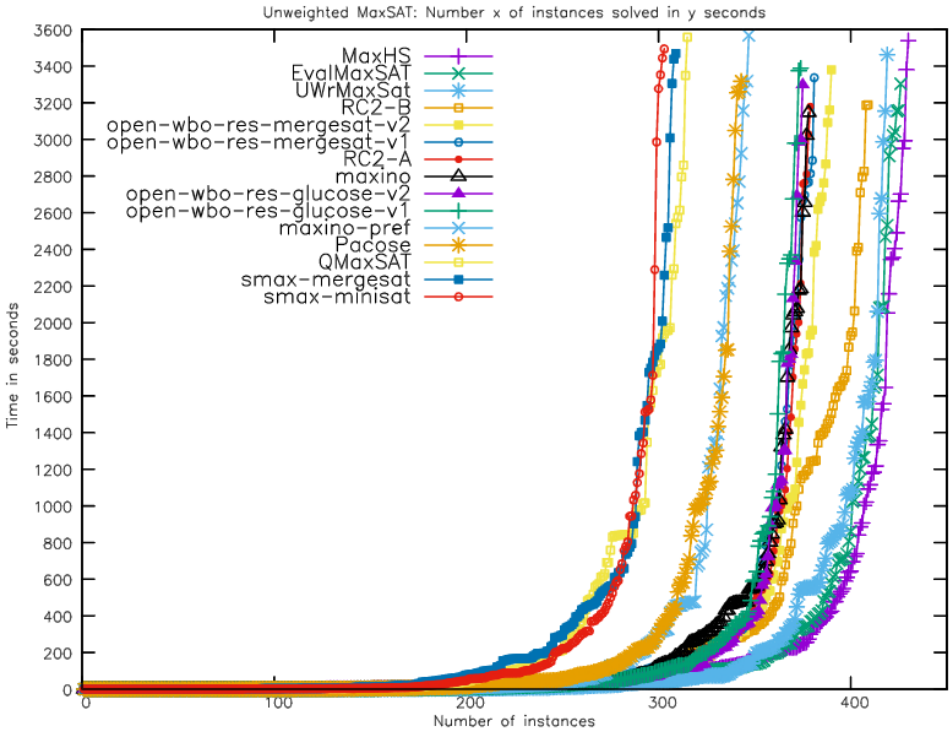

Figura 2.4: Resumen de los resultados en el conjunto unweighted de benchmarks completos.

han conseguido resolver (de manera exacta) benchmarks con más de 10 millones de variables y 70 millones de cláusulas.

Por otro lado, la MSE ha conseguido reunir, a lo largo de los años, un gran conjunto de benchmarks de MaxSAT. Este conjunto incluye casi 2000 instancias de MaxSAT, y ocupa (después de ser comprimido) más de 80GB, que además son heterogéneas tanto en tamaño como en forma y contenido, ya que codifican problemas de 15 dominios de aplicación distintos.

2.4.2 Formato DIMACS modificado

Dado que DIMACS es un formato genérico para representar fórmulas proposionales en CNF, se puede emplear tanto para SAT como para MaxSAT. No obstante, es habitual querer representar las extensiones de MaxSAT también, para lo que es necesario modificar el formato DIMACS para que admita pesos y PMS. Todos los benchmarks empleados en este trabajo pertenecen a la MSE, que propone la siguiente modificación de DIMACS.

Los ficheros DIMACS comienzan con la línea de parámetros (*p-line*) modificada. Ahora tiene la forma *p wcnf <variables> <clauses> <top>*, donde *<variables>* y *<clauses>* denotan, respectivamente, el número de variables y cláusulas del problema, mientras que *<top>* representa el límite a partir del cual una cláusula es considerada *hard*. Esto es, las cláusulas *soft* tendrán

un peso menor que *top*, y las *hard* tendrán un peso igual a *top*. Además, *top* siempre debe ser mayor que la suma de todos los pesos de las cláusulas *soft*.

Después de la *p-line* se encuentran las cláusulas de la fórmula. El primer número corresponde al peso de dicha cláusula, que siempre debe ser mayor o igual que 1 y menor que 2^{63} (que es el valor máximo para un entero con signo de 64 bits, *long int* en C). A continuación del peso se encuentra la fórmula, codificada igual que en SAT. Los comentarios tampoco cambian respecto a lo explicado en el apartado anterior.

Por ejemplo, la fórmula

$$(P \vee Q) \wedge (\neg P \vee R) \wedge (\neg Q \vee \neg S)$$

donde se quiere que la primera cláusula sea *hard*, y las otras dos tengan un peso de 1 y 2, se representaría de la siguiente forma:

```
c
c Ejemplo formato DIMACS para
c una instancia de Weighted PMS
c
p wcnf 4 3 4
4 1 2 0
1 -1 3 0
2 -2 -4 0
```

2.5 Algoritmos para MaxSAT

En esta sección se describen los algoritmos para la resolución del problema MaxSAT que han sido utilizados en este trabajo, así como otros que se consideran relevantes para detallar el estado del arte en esta materia.

2.5.1 Algoritmos de búsqueda local

El espacio de búsqueda en los problemas MaxSAT está formado por el conjunto de todas las posibles asignaciones de verdad para las variables del problema, lo que significa que el espacio de búsqueda crece de manera exponencial con el número de variables. La idea general de los algoritmos de búsqueda local es ir haciendo pasos aleatorios en el espacio de búsqueda invirtiendo el valor de una variable en cada paso. Cada algoritmo utiliza su propia estrategia para decidir qué variable invertir en cada paso, y es lo que supone la principal diferencia entre los diferentes algoritmos planteados.

Normalmente, los algoritmos de búsqueda local son *incompletos*. Esto es, no pueden garantizar que se encuentre una solución óptima en un tiempo finito. Esto se debe a que pueden

quedarse atrapados en máximos locales o mesetas, en las que el algoritmo ya no sepa hacia dónde avanzar. Para esto, muchos algoritmos incorporan alguna estrategia para escapar de los máximos locales. Por ejemplo, muchos de estos algoritmos realizan una reinicialización aleatoria después de un número fijo de pasos (o un timeout) en los que no se ha encontrado una solución.

A continuación se describe la arquitectura GSAT, la familia de algoritmos sobre la que se basan la mayoría de técnicas de búsqueda local para MaxSAT. En la actualidad, versiones más sofisticadas de esta arquitectura siguen vigentes, especialmente en el campo de los resolutores incompletos. Por ejemplo, los resolutores *SATLike* [22], *StableResolver* [23] y *sls-mcs* [24] fueron presentados en la evaluación de MaxSAT 2020 (MaxSAT Evaluation, MSE) [5] y emplean algoritmos de búsqueda local inspirados en esta heurística.

Arquitectura GSAT

El algoritmo GSAT fue introduciddo por Selman, Levesque y Mitchell en 1992 [25]. Es un algoritmo voraz ("greedy"), lo que significa que la heurística toma la decisión óptima localmente en cada paso, pero no asegura obtener una solución global óptima. Por ejemplo, en el problema del viajero, una estrategia voraz es visitar la ciudad más cercana en cada paso del camino. No intenta obtener la mejor solución, pero sí una buena solución en un tiempo razonable.

El funcionamiento del algoritmo GSAT, detallado en el Pseudocódigo 1, es el siguiente: comienza con una asignación de verdad aleatoria, y en cada paso invierte ("flip") la variable que conduzca al mayor incremento del número total de cláusulas satisfechas. Este proceso se repite hasta que se alcance el máximo número de flips (MAX_FLIPS) preestablecido. Una vez alcanzado, se reinicia el algoritmo y vuelve a comenzar el proceso partiendo de una nueva asignación aleatoria. Este procedimiento de realizar la búsqueda voraz y reiniciar se realiza un número de veces preestablecido (MAX_TRIES), y se considera como solución el mejor resultado a lo largo de todas las ejecuciones.

El problema de este algoritmo es que puede quedarse atrapado en máximos locales con facilidad, y el único mecanismo que posee para escapar es el reinicio aleatorio. Para solucionar esto, a lo largo de los años se han propuesta numerosas modificaciones para dotarlo de más herramientas para escapar de los máximos locales, y acercarse al óptimo tanto como sea posible.

Una de las modificaciones más destacables es **GSAT con Random Walk (GWSAT)**. Fue propuesta en 1993 por Selman y Kautz [26], donde la presentan como una estrategia útil para ayudar a GSAT a escapar de los máximos locales, gracias a la introducción de una nueva heurística, el "paseo aleatorio" (*Random Walk, RW*), que funciona de la siguiente forma: en cada paso de la búsqueda local, se elige aleatoriamente una cláusula del conjunto de cláusulas

Algorithm 1 GSAT

```
1: SolucionGlobal ← 0
2: for i ∈ 1 : MAX_TRIES do
3:     T ← Asignación de verdad generada aleatoriamente
4:     for j ∈ 1 : MAX_FLIPS do
5:         p ← la variable que más incremente el número de cláusulas satisfechas
6:         InvertirVariable(p)
7:         T ← T con la variable p invertida
8:         SolucionActual ← Numero de cláusulas satisfechas por T
9:         if SolucionActual > SolucionGlobal then
10:            SolucionGlobal ← SolucionActual
11:        end if
12:    end for
13: end for
14: return SolucionGlobal
```

insatisfechas, y se invierte el valor de una de las variables que aparecen en esa cláusula (forzando dicha cláusula a ser cierta). Esta estrategia se combina con GSAT decidiendo, en cada paso, qué algoritmo se ejecuta en función de una probabilidad *p*:

$$heuristica = \begin{cases} RandomWalk & \text{con probabilidad } p \\ GSAT & \text{con probabilidad } 1-p \end{cases}$$

Hoos probó en 1998 [27] que este algoritmo tiene la propiedad PAC (*Probabilistically Approximately Complete*) siempre que $p > 0$. Esto significa que la probilidad de que encuentre la solución óptima tiende a 1 conforme el tiempo tiende a infinito, o lo que es lo mismo, cuanto más trabajo realice el algoritmo, mayor es la probabilidad de que encuentre el óptimo.

Otras modificaciones que merece la pena mencionar, aunque no serán desarrolladas en este trabajo, son GSAT con Tabu Search [28], que prohíbe escoger variables que han sido elegidas recientemente; y HSAT (Historic SAT) [29] que siempre elige la variable que más tiempo lleva sin ser invertida, de aquellas con la misma puntuación.

2.5.2 Algoritmos basados en SAT

El uso de resolutores SAT para resolver problemas MaxSAT se ha establecido en los últimos años como uno de los paradigmas más eficaces, especialmente en la resolución de benchmarks industriales. Prueba de ello es que en la evaluación MaxSAT 2020 se presentaron numerosos resolutores que utilizan resolutores SAT como base, sobre todo en la categoría de resolutores completos, como EvalMaxSAT [30], MaxHS [31], Open-WBO [32] y UWrMaxSat [33].

La idea principal de los resolutores basados en SAT es intentar resolver el problema MaxSAT mediante la resolución de una secuencia de problemas SAT [34]. Cada instancia de esta secuencia se genera a partir de la anterior, mediante la adición de nuevas variables (*blocking variables*) y nuevas cláusulas que representan restriciones de cardinalidad codificadas en CNF. Las restricciones de cardinalidad (en relación a las fórmulas proposicionales) son restricciones en el número de variables que pueden ser verdaderas dentro de un conjunto dado. Estas restricciones aparecen en muchos problemas del mundo real, y se pueden traducir (codificar) como fórmulas en CNF.

Los resolutores basados en SAT se dividen principalmente en dos grupos [35], cuyo funcionamiento es el siguiente:

- Los basados en satisfactibilidad (**satisfiability-based**): dada una instancia MaxSat $\phi = C_1, ..., C_n$, a cada cláusula suave (es decir, que se puede volver falsa) C_i se le añade nueva variable b_i, denominada de *bloqueo*. Resolver el problema MaxSAT se reduce a minimizar el número de variables de bloqueo verdaderas en $\phi' = C_1 \vee b_1, ..., C_n \vee b_n$. Se comienza resolviendo ϕ' sin restricciones con un resolutor SAT, para obtener un modelo inicial con k variables de bloqueo verdaderas. Después se van añadiendo progresivamente restricciones de cardinalidad a las variables de bloqueo $\sum_1^n b_i \leq k$, y ejecutando el resolutor SAT de nuevo, hasta que el problema sea insatisfactible. Cuando se llegue a ese punto, k es la solución óptima. Es decir, el coste de la asignación óptima de ϕ corresponde al punto exacto de la transición entre las fórmulas SAT satisfactibles e insatisfactibles.

- Los basados en insatisfactibilidad (**unsatisfiability-based**): dada una instancia MaxSat ϕ, realizan el mismo proceso iterativamente hasta que ϕ es vuelve satisfactible. Este proceso consiste en ejecutar un resolutor SAT sobre ϕ, y si es insatisfactible, obtener un subconjunto (o *núcleo*) de cláusulas que sea insatisfactible (*unsatisfiable core*). A cada cláusula del núcleo se le añade una variable de bloqueo b_i, y ϕ se amplía añadiendo una restricción de cardinalidad para que exactamente una de las nuevas variables sea cierta $\sum_1^n b_i = 1$. Este proceso se realiza hasta que la instancia es satisfactible, y el coste óptimo del problema MaxSAT original corresponde con el número de iteraciones realizadas. Este procedimiento fue presentado originalmente por Fu y Malik en 2007 [36].

La eficiencia de los resolutores basados en SAT depende críticamente del resolutor SAT que utilicen, y de cómo codifiquen las restricciones de cardinalidad.

2.5.3 Algoritmos evolutivos para MaxSAT

Los Algoritmos Evolutivos (EA, de sus siglas en inglés) son una familia de algoritmos de optimización que se inspiran en la evolución natural. Estos algoritmos mantienen una población de individuos que representan un conjunto de soluciones candidatas. Las dos características más importantes a la hora de resolver un problema mediante EAs es la definición de los individuos y la función objetivo a emplear, que debe ser eficiente y lo más precisa posible (ajustarse al objetivo real del problema). Para MaxSAT, los individuos pueden ser modelados como vectores de variables booleanas, tantas como tenga la instancia. En relación a la función objetivo, suele emplearse una función que dada una solución (es decir, una interpretación) devuelva el coste (es decir, el número de cláusulas satisfechas) de la instancia MaxSAT que se está evaluando, que es exactamente el objetivo a maximizar. Además, atendiendo a las propiedades que debe cumplir un problema para pertenecer a la clase NPO (a la que pertenece MaxSAT), debemos recordar que las funciones objetivo de estos problemas deben ser computables en tiempo polinomial.

Por estas razones, el problema MaxSAT es un buen candidato para resolver mediante EAs. No obstante, la mayoría de algoritmos descritos en la literatura están desarrollados para SAT, aunque son fácilmente extensibles para resolver MaxSAT. Existen aproximaciones, como *GA-SAT*[37][38] o *pEvoSAT*[39], que implementan algoritmos genéticos, un método clásico de EAs que se inspira en operadores biológicos como la mutación, el cruce o la selección natural. También existen intentos de emplear otros algoritmos evolutivos, como el algoritmo de la Colonia Artificial de Abejas (ABC), algoritmos evolutivos inspirados en la cuántica (QIEA), o algoritmos de estimación de distribución (EDA) [40].

2.6 Evolución Diferencial

Desde la llegada de los primeros ordenadores a mediados del siglo pasado, la optimización matemática se ha convertido en uno de los campos de investigación más importantes, pues tiene aplicaciones en casi todas las áreas del conocimiento humano - economía, biología molecular, aeronáutica, física, investigación operativa, y un largo etcétera - permitiendo resolver problemas del mundo real grandes y complejos. No suele ser posible dar una solución óptima en un tiempo razonable (i.e. polinomial) en la mayoría de problemas, por lo que en los últimos años se ha desarrollado una gran cantidad de algoritmos incompletos, es decir, que no aseguran una solución óptima, pero sí una buena aproximación en un intervalo relativamente corto de tiempo.

La *Evolución Diferencial* (ED) es un algoritmo evolutivo que se inspira en la evolución biológica y cuyo objetivo es aproximar una buena solución a través de un proceso iterativo. La ED optimiza el problema manteniendo una población de soluciones candidatas y creando

nuevas soluciones mediante la combinación de las existentes. Fue introducido en la década de 1990 por Storn y Price [41], y desde entonces se han propuesto múltiples variantes y mejoras.

2.6.1 Metaheurísticas

La Evolución Diferencial se enmarca dentro de una familia de algoritmos denominados metaheurísticos, que están diseñados para encontrar una solución suficientemente buena a un problema de optimización en espacios de búsqueda muy grandes, pero sin garantías de alcanzar una solución óptima. En general, las *metaheurísticas* son procesos (o heurísticas) de alto nivel que realizan muy pocas o ninguna suposición sobre el problema a optimizar, por lo que se pueden aplicar a una gran variedad de problemas.

2.6.2 Descripción del algoritmo

Se parte de una población inicial de soluciones candidatas (*individuos*, en la nomenclatura usual en algoritmos evolutivos) que codifican en su *genotipo* (asignación de valores para los distintos parámetros del individuo) una posible solución a un problema, y las soluciones posteriores se generan mediante la combinación de las anteriores, utilizando las operaciones de cruce vectorial y mutación. A cada individuo se le asigna un valor de *calidad* (en inglés, *fitness*), que representa la calidad de dicha solución y que corresponde a la función objetivo $f_{obj}(x)$ a optimizar. En el Pseudocódigo 2 se muestra el algoritmo básico de Evolución Diferencial.

2.6.3 Parámetros

En este algoritmo es necesario ajustar los siguientes parámetros:

- **CR (Crossover)**: Controla la probabilidad de cruce.
- **F**: Controla la mutación ponderando la importancia de los dos miembros seleccionados para calcular el vector denominado donador.
- **NP**: Tamaño de la población [8].
- **GENS**: Número de generaciones.

2.6.4 Etapas

El algoritmo ED posee cuatro etapas bien definidas:

[8] No confundir el parámetro NP con la clase de complejidad computacional NP, con la que no guarda ninguna relación. Se ha elegido esta notación porque es la se emplea normalmente para designar el tamaño de la población en ED (y cualquier otro algoritmo evolutivo) [41].

Algorithm 2 Evolución diferencial

for `Individuo x ∈ Población` **do**
  `Individuo x ← InicializarAleatoriamente()`
end for
while not CriterioParada() **do**
  for $i \in 1 : NP$ **do**
    $x_{r1}, x_{r2}, x_{r3} \leftarrow$ `IndividuosAleatorios(Población)` ▷ $x_i \neq x_{r1} \neq x_{r2} \neq x_{r3}$
    $R \leftarrow$ `NumeroAleatorio ∈ [1,D)`
    for $j \in 1 : D$ **do**
      $rand_j \leftarrow$ `NumeroAleatorio ∈ [0,1]`

$$y_{i,j} = \begin{cases} x_{r1,j} + F * (x_{r2,j} - x_{r3,j}) & \text{if } R == j \text{ or } rand < CR \\ x_{i,j} & \text{en otro caso} \end{cases}$$

    end for
    if $f_{obj}(y_i) <= f_{obj}(x_i)$ **then**
      $x_i = y_i$ ▷ Update the target
    end if
  end for
end while

- **Inicialización**: Se ejecuta una única vez al comienzo del algoritmo, y su objetivo es realizar una primera asignación de los parámetros que codifican en cada individuo el problema a optimizar. Típicamente esta asignación de valores es aleatoria, pero acotada al rango del problema.

- **Mutación**: Se encarga de generar los vectores donadores. Un *vector donador* se forma eligiendo tres individuos de modo aleatorio entre la población actual, diferentes entre sí y al vector principal (al que se le va a generar un *candidato* partiendo de este donador). El donador se forma sumando el vector del primer individuo (denominado *vector base*) con la diferencia ponderada de los otros dos.

$$v_{donador} = x_1 + F(x_2 - x_3)$$

- **Recombinación** (*crossover*): En esta etapa se genera un vector o individuo candidato $y^g_{p,m}$, combinando los parámetros del vector inicial $x^g_{p,m}$ (el individuo *objetivo* o *target*) con los del vector donador $v^g_{p,m}$ (el *mutante*), en función del factor CR.

$$y^g_{p,m} = \begin{cases} v^g_{p,m} & \text{if } R = j \parallel r < CR \\ x^g_{p,m} & \text{en otro caso} \end{cases}$$

Siendo D el número de variables del vector (es decir, la dimensionalidad del problema), $g \in [0, GENS)$ la generación actual, $r \in [0.0, 1.0]$ un número aleatorio entre 0 y 1, $p \in [0, NP)]$ el índice del individuo en la población, $m \in [0, D)$ el índice de la variable en el vector y $R \in [0, D)$ un número aleatorio entero. El parámetro R garantiza que, durante la generación del vector candidato, siempre se va a considerar el material genético del donador (ya que al menos un parámetro proviene de él).

- **Selección**: En esta etapa se genera la siguiente generación, de forma que si un individuo candidato mejora al individuo *target (x)*, el candidato (y) reemplaza al *target* en la siguiente generación.

$$v_p^{g+1} = \begin{cases} y_p^g & \text{if } f_{obj}(y_p^g) <= f_{obj}(x_p^g) \\ x_p^g & \text{en otro caso} \end{cases}$$

Las tres últimas etapas se repiten durante todas las generaciones hasta satisfacer un criterio de parada. La idea principal detrás del operador de mutación es adaptar la magnitud de la diferencia de vectores $F * (x_2 - x_3)$ a lo largo del proceso evolutivo. Esa diferencia es la que marca la magnitud de la perturbación en el vector base (x_1) a la hora de buscar (explorar) nuevas soluciones en el espacio de búsqueda. En las primeras generaciones es de esperar que esta diferencia sea muy alta, pues los individuos serán muy distintos unos de otros. Pero, a medida que avancen las generaciones, la población se irá estabilizando y homogeneizando, con lo que esta diferencia se irá reduciendo progresivamente. Este concepto puede apreciarse facilmente en la figura 2.5. De este modo, ED presenta un control automático entre exploración y explotación. La exploración es mayor en las primeras generaciones y va tendiendo a mayor explotación a lo largo de las generaciones.

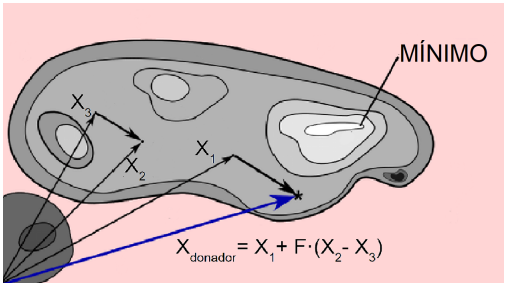

Figura 2.5: La mutación en la definición del individuo "donador"

2.6.5 Evolución Diferencial Binaria

El algoritmo EDB (*Evolución Diferencial Binaria*) [42] pretende replicar la naturaleza continua de ED, pero en un dominio discreto. El principal cambio se introduce en la operación de

mutación. El mutante $v_{p,m}^{g}$ pasa a generarse de la siguiente forma: de la población se escogen tres individuos x_1, x_2 y x_3 diferentes entre sí, y diferentes también al *target*. El individuo x_1 será el *vector base*, y de los otros dos y de un número aleatorio $r_j \in [0.0, 1.0]$ dependerá si se invierte o no el valor del j-ésimo bit de x_1. El pseudocódigo 3 muestra cómo se puede adaptar ED para trabajar directamente con variables binarias.

Como se puede ver en el Pseudocódigo 3, al trabajar con variables Booleanas se cambia la forma de construir el *vector donador*. Los pasos de inicialización y selección funcionan de forma análoga, cambian los de mutación y cruce, que ahora están bien diferenciados:

- Primero, en el paso de *mutación* se construye el vector donador v_i. En el algoritmo continuo se ponderaba la diferencia de dos vectores con el parámetro F, esto permitía que aquellas variables que tenían más impacto sobre la calidad de los individuos (las más importantes) tendieran a valores similares a lo largo de las generaciones, y la diferencia se acercase cada vez más a cero, de manera que esa variable fuese más homogénea y estable. Para reproducir este comportamiento en el dominio discreto se deben hacer varios cambios:

  - Para empezar, dado que solo hay dos valores posibles (1 y 0), mutar una variable consiste en invertir su valor.
  - El parámetro F ahora es una probabilidad de mutación, en vez de una medida de la importancia de la diferencia.
  - Antes de invertir una variable (mutarla), se comprueba que los vectores x_{r2} y x_{r3} tienen valores distintos en la variable actual (sobre la que se está iterando). Conforme avanzan las generaciones y el algoritmo identifica las variables más importantes, es más probable que estas tengan el mismo valor en varios individuos, y este mecanismo evita hacer cambios en esas variables.

  Además, igual que en ED, el vector donador se forma (principalmente) a partir de los valores del vector base x_{r1}, aplicando las mutaciones pertinantes sobre sus variables.

- Durante el paso de *cruce* se combinan los materiales genéticos del vector donador y el individuo x_i en función del parámetro CR. Este paso es muy parecido a ED original, la principal diferencia es que no se asegura que algún material genético del vector donador se encuentre en el candidato.

Algorithm 3 Evolución diferencial binaria

1: **for** `Individuo x ∈ Población` **do**
2: `Individuo x ← InicializarAleatoriamente()`
3: **end for**
4: **while not** CriterioParada() **do**
5: **for** $i \in 1 : NP$ **do**
6: $x_{r1}, x_{r2}, x_{r3} \leftarrow$ `IndividuosAleatorios(Población)`
▷ $x_i \neq x_{r1} \neq x_{r2} \neq x_{r3}$
7: **for** $j \in 1 : D$ **do**
8: $mrand_j \leftarrow$ `NumeroAleatorio ∈ [0,1]`
9: Paso de mutación:

$$v_{i,j} = \begin{cases} 1 - x_{r1,j} & \text{if } x_{r2,j} \neq x_{r3,j} \text{ and } mrand_j < F \\ x_{r1,j} & \text{en otro caso} \end{cases}$$

10: $crand_j \leftarrow$ `NumeroAleatorio ∈ [0,1]`
11: Paso de cruce:

$$y_{i,j} = \begin{cases} v_{i,j} & \text{if } crand_j <= CR \\ x_{i,j} & \text{en otro caso} \end{cases}$$

12: **end for**
13: **if** $f_{obj}(y_i) <= f_{obj}(x_i)$ **then**
14: $x_i = y_i$ ▷ Update the target
15: **end if**
16: **end for**
17: **end while**

Capítulo 3

Trabajo desarrollado

3.1 Funcionamiento

El trabajo principal de este proyecto ha consistido en desarrollar un resolutor de MaxSAT basado en ED, que hemos denominado DEMaxSATSolver. El resolutor desarrollado funciona, a grandes rasgos, de la siguiente forma:

Tal y como se ha explicado antes, ED mantiene una población de soluciones candidadas del problema que intenta resolver. En este caso, obviamente, dichas soluciones se corresponde a posibles soluciones del problema MaxSAT. De esta forma, cada individuo mantendrá en su genotipo una interpretación, es decir, una asignación de verdad para todas las variables de la fórmula proposional que se está resolviendo (i.e la instancia MaxSAT). En cuanto a la función de fitness, se emplea la suma de los pesos de las cláusulas satisfechas por una determinada interpretación (o de modo equivalente, si consideramos el problema como minimización, el fitness sería la suma de los pesos de las cláusulas insatisfechas). ED intentará mejorar a los individuos mediante los operadores de mutación, cruce y selección, para maximizar esta función objetivo. Para guiar el proceso evolutivo y refinar las soluciones, se han integrado dos heurísticas clásicas de búsqueda local para el problema MaxSAT, GSAT y Random Walk. El resolutor está disponible de manera libre en [43].

3.2 Parámetros

Antes de explicar los módulos con los que cuenta el resolutor, conviene comentar los diferentes parámetros que se pueden ajustar, su significado y cómo afectan al funcionamiento del algoritmo:

- **GEN**: El número máximo de generaciones del algoritmo evolutivo.
- **NP**: El tamaño de la población.

- **CR**: Probabilidad de cruce.

- **F**: Probabilidad de mutación.

- **LSS** (Local Search Step): Los algoritmos de búsqueda local (GSAT y RandomWalk) se aplican N veces en cada generación sobre cada individuo. Este parámetro ajusta el número de veces que se ejecutan dichas heurísticas, definiendo N como un tanto por ciento (determinado por LSS) de las variables del problema. Es decir, si se fija LSS=0.1 y el problema tiene 100 variables, se ejecutan 10 pasos de búsqueda local en cada generación en cada individuo seleccionado para la aplicación del LSS. Esto se hace para que el número de pasos locales dependa del tamaño del problema (en general, conforme crece el tamaño del problema crece el número de variables, no sólo el número de cláusulas).

- **maxLSS**: Fija el máximo número de pasos de búsqueda local que se pueden realizar en cada generación en cada individuo.

- **RW**: Probabilidad de emplear GSAT o RandomWalk. Durante cada paso de búsqueda local se genera un número aleatorio $r \in [0.0, 1.0]$, y la heurística a ejecutar se elige de la siguiente forma:

$$heuristica = \begin{cases} GSAT & \text{si } r > RW \\ RandomWalk & \text{en otro caso} \end{cases}$$

- **SEED**: Semilla de los números aleatorios. Puede fijarse a -1 para usar la hora del sistema como semilla.

- **HSCOPE** (Heuristic Search Scope): Define el alcance de las heurísticas de búsqueda local, es decir, a qué individuos se aplica. Puede fijarse a *all*, para que se apliquen a todos los individuos; o al valor *better_than_mean*, para que se apliquen solamente a aquellos individuos cuya calidad es mejor que la calidad media de la población.

Estos parámetros se encuentran definidos en el fichero de configuración llamado ”maxsat.cfg”, y el fichero encargado de procesarlos es *settingParser.c*, del módulo *SettingParser*.

3.3 Parsing de los ficheros

El módulo WCNFParser procesa los ficheros e implementa las estructuras de datos necesarias para almacenar una instancia del problema MaxSAT.

3.3.1 Cláusulas

Como se ha explicado anteriormente, un problema MaxSAT se compone de un conjunto de cláusulas, y cada cláusula es un conjunto de literales. Por tanto, lo primero es definir cómo se guardan las cláusulas.

La estructura de datos empleada para almacenar las cláusulas consta de cuatro campos: *size* es el tamaño actual de la cláusula (i.e. el número de literales que contiene), *capacity* es el número máximo de literales que puede almacenar actualmente, *weight* es el peso de la cláusula, y finalmente *literals* es un array donde se guardan los literales de dicha cláusula. Los literales se almacenan como enteros.

En la mayoría de problemas no se conoce a priori el número de literales que tendrá una cláusula. Una opción para solucionar esto es inicializar el vector de literales con un tamaño muy grande, por ejemplo, el número de variables del problema. Pero esta aproximación estaría desperdiciando mucha memoria, por lo que se ha optado por implementar un array dinámico. El array de literales se inicializa con una capacidad arbitraria pequeña (4 literales, en este caso), que va aumentando conforme se necesita, multiplicando el tamaño del array por un factor de crecimiento, y reservando memoria acorde a esto [1].

3.3.2 Entradas

Esta estructura guarda, para cada variable, las cláusulas en las que aparece, es decir, funciona como un índice invertido. De esta forma, permite tener una correspondencia bidireccional entre cláusulas y variables. Para cada entrada del índice (es decir, para cada variable) se guardan tres campos análogos al módulo de las cláusulas (eliminando el campo *weight*) pero, en vez de un array de literales, se guarda un vector que almacena los índices de las cláusulas en las que aparece la variable.

WCNF

Finalmente, se ha creado una estructura que representa el problema MaxSAT en formato WCNF. Almacena una lista de las cláusulas del problema y una lista de entradas. Aparte, los parámetros de la *p-line* se almacenan en tres campos: *variable_count* , *clause_count* y *top*, junto con una variable adicional, *max_cost*, que guarda la suma de los pesos de todas las cláusulas soft.

[1] En este trabajo se utiliza un factor de crecimiento de 2.

3.4 Implementación de Evolución Diferencial Binaria

El módulo principal del resolutor que se ha desarrollado en este trabajo es el que implementa la Evolución Diferencial Binaria (EDB).

3.4.1 Los individuos

Como se ha explicado en 2.6, los individuos son vectores que almacenan los parámetros de la función que deseamos optimizar. En el problema MaxSAT, cada uno de los elementos del vector será una variable de la fórmula proposicional, y la función de calidad o *fitness* recibirá una fórmula y una interpretación, y devolverá la suma de los pesos de las cláusulas satisfechas.

Por tanto, en nuestro resolutor cada individuo codifica una asignación de verdad de la fórmula, es decir, una interpretación. Teniendo una asignación para cada variable, podemos determinar el valor de las cláusulas (si son ciertas o falsas). La calidad de individuo (su valor de *fitness*) se define como la suma de los pesos de las cláusulas satisfechas por esa interpretación.

Entre los datos de cada individuo, aparte de la asignación de verdad y su calidad, se almacena más información relevante. Se guarda un array *clValues* que indica, para cada cláusula, si es verdadera o falsa, con la intepretación del individuo. También se mantiene otro vector denominado *supports*, que guarda el número de soportes que tiene cada cláusula. Definimos los *soportes* de una cláusula como los literales que la hacen cierta. Por ejemplo, para un individuo que tenga como genotipo la asignación de variables correspondiente a la interpretación $I = \{r\}$ para las variables $At = \{p, q, r\}$, en la cláusula $p \vee \neg q \vee r$ tendríamos 2 soportes, que son los literales $\neg q$ y r, ya que ambos hacen cierta la cláusula, pues q es falso en I mientras que r es cierto. Por último, en los datos del individuo, también almacenamos un array *unsatCl* con los índices de las cláusulas insatisfechas.

Algunos de estos atributos (la asignación y la calidad) son imprescindibles, mientras que el resto permiten acelerar la computación o son necesarios para las heurísticas. En concreto, clValues se usa para implementar la función de re-evaluación (que se explica detalladamente en el apartado 3.4.3) y la heurística GSAT. Esta misma heurística necesita también el número de soportes de cada cláusula. Los índices de las cláusulas insatisfechas son utilizados para la implementación de RandomWalk.

3.4.2 Vectores binarios

Cada individuo mantiene varios vectores de elementos binarios, es decir, un array donde cada posición puede ser verdadera o falsa (1 o 0). Una aproximación trivial para implementar esto sería utilizar un array de enteros (*int*), donde en cada posición guardamos un 1 o un 0. La desventaja de esta implementación es que, para representar verdadero/falso, solo se necesita 1 bit, pero los *ints* utilizan normalmente 32 bits (4 bytes) para representar cada entero. Por tanto,

estaríamos desperdiciando 31 de cada 32 bits del array, esto es, casi un 97% de la memoria reservada.

El lenguaje C no proporciona una implementación de un array de bits, pero sí las operaciones a nivel de bit necesarias para implementarlo, por lo que se ha optado por esta opción. Es decir, definimos los vectores binarios como arrays de enteros normales, pero accedemos a sus posiciones a nivel de bit.

Supongamos que tenemos un array A de 4 elementos (y, para que sea más manejable, los enteros tendrán un tamaño de 4 bits), por lo que podemos almacenar 4*4 = 16 valores binarios. Para acceder a un bit concreto, por ejemplo, al 7, primero tenemos que saber cuál es el índice del entero al que pertenece ese bit. Para ello, primero calculamos el cociente entero entre el bit al que queremos acceder y el tamaño de los enteros $7//4 = 2$. Después, necesitamos saber la posición de nuestro bit dentro de ese entero (el offset), para lo que aplicamos el módulo $7\%4 = 3$. Para acceder a un valor concreto, se aplica un AND binario y se desplaza a la izquierda un 1 hasta el offset donde se encuentre el bit. En resumen:

```
A[2] = 1100
0001 << 3 = 0100
1100 & 0100 = 0001
```

De forma parecida, trabajando con los operadores lógicos binarios, se implementan las opereciones que permiten asignarle 1 o 0 a un bit, y la operación para invertirlo. Estas funciones están basadas en las que se pueden encontrar en [44].

En la implementación real, el tamaño de los *ints* suele ser 32 bits, aunque en sistemas antiguos era de 16 bits y en los modernos puede llegar hasta los 64 bits.

3.4.3 Función de *fitness*

La función de *fitness* recibe la fórmula proposicional y una asignación de verdad, y calcula la suma de los pesos de las cláusulas satisfechas. Para ello, recorre todas las cláusulas de la fórmula y para cada una determina si es verdadera o falsa, comprobando para cada literal el valor que posee en la interpretación del individuo que se está evaluando.

Además, se aprovecha que esta función recorre todas las cláusulas para actualizar el resto de campos de los individuos:

- El numero de soportes de una cláusula. Esta medida es interesante, pues si cambiamos el valor de una variable que es el único soporte de una cláusula, esta pasará a ser falsa, mientras que si una cláusula tiene varios soportes, sabemos que podemos cambiar el valor de verdad de alguna de sus variables sin tener que preocuparnos de que vaya a ser falsa. Este valor se guarda en el vector *supports* de cada individuo.

- Se construye un índice que indica, para cada cláusula, si es verdadera o falsa. De nuevo, solo necesitamos 1 bit para cada posición, así que se vuelve a hacer uso de los vectores de bits explicados anteriormente. Corresponde al atributo *clValues* de los individuos.
- Los índices de las cláusulas insatisfechas en el atributo *unsatCl* de los individuos.

Esta función es muy costosa, ya que itera sobre todas las cláusulas y sobre todos los literales de cada cláusula. En ocasiones, si se han modificado pocas variables, el proceso de evaluación resulta muy ineficiente, pues recalcula el valor de muchas cláusulas que no han sido alteradas. Para solventar este problema, se ha implementado una función de re-evaluación.

Re-evaluación

Esta función hace uso del índice invertido construído durante la lectura del fichero para acelerar el cálculo de la calidad de un individuo. Para ello, dada una variable cuyo valor ha sido modificado, itera únicamente sobre las cláusulas en las que está presente dicha variable. De esta forma se evita recorrer cláusulas cuyo valor no ha cambiado en absoluto, y se ahorra mucho tiempo de computación.

Hay que tener en cuenta que la eficiencia de esta función disminuye conforme aumenta el número de variables que han sido modificadas. Por ejemplo, si hubiera dos variables que estuvieran en todas las cláusulas, estaríamos calculando dos veces el valor de todas las cláusulas, mientras que con la función normal de evaluación, solo una. Este es un caso extremo, pero, en general, es común que las variables se encuentren en múltiples cláusulas, y cuanto mayor sea la co-ocurrencia de las variables, más ineficiente será este método.

3.4.4 Mutación, cruce y selección

El algoritmo básico de ED se puede modificar para adaptarse al problema que se intenta resolver. Dado que el objeto de este trabajo es resolver el problema MaxSAT, se debe ajustar el algoritmo evolutivo para que se pueda aplicar directamente a variables proposicionales. Al adaptar ED para el problema MaxSAT, el genotipo de un individuo (es decir, su vector de parámetros), será una interpretación proposicional, es decir, una asignación de verdad para los átomos de la signatura At usados en la fórmula CNF. Por tanto, si la fórmula que se intenta resolver tiene N variables diferentes, los individuos tendrán un genotipo de N elementos binarios (bits), denotando con un 1 cuando una variable es cierta, y 0 en caso contrario.

Las operaciones de mutación y cruce se implementan tal y como se detalla en el punto 2.6.5. Se emplean los operadores binarios explicados anteriormente para trabajar con los vectores de asignación, que son los que van a determinar la calidad de la solución.

En cada generación, se itera sobre todos los individuos. Para cada individuo, se itera sobre todas las variables de su asignación de verdad, pues son los atributos que determinan su

calidad y, por tanto, sobre los que se aplican los operadores genéticos.

Tras realizar las operaciones de mutación y cruce, se evalúa el nuevo individuo *candidato*. Si su calidad es mayor que la del *target*, se hace una copia profunda (*deep clone*) del *candidato* en la posición de memoria del *target*.

3.5 Heurísticas de búsqueda local

A diferencia de ED, que es un método general adaptado para este problema, estas heurísticas son propias del problema MaxSAT. Se aplican sobre cada individuo justo antes del bucle que itera sobre sus variables (es decir, antes de los operadores genéticos), y se decide cuándo aplicarla o no a un individuo en función del parámetro HSCOPE, explicado en el punto 3.2.

Las dos heurísticas se combinan entre sí mediante una probabilidad, denominada factor de RW, y un número aleatorio $r \in [0.0, 1.0]$, de forma que, en cada iteración, se decide la heurística a aplicar de la siguiente forma:

$$heuristica = \begin{cases} GSAT & \text{si } r > RW \\ RandomWalk & \text{en otro caso} \end{cases}$$

3.5.1 Implementación de GSAT

Mientras que los operadores de cruce y mutación son aleatorios, esta búsqueda local permite guiar al proceso evolutivo, modificando los individuos para que puedan mejorar su calidad. Esta heurística se aplica sobre cada individuo, y consta de dos etapas diferenciadas:

- En la primera fase se le asigna una *puntuación* a cada variable. Esta puntuación se obtiene a partir del número de cláusulas que se harían verdaderas si se invierte el valor de esa variable. Para implementarlo, para cada variable se calculan dos valores: *break*, que representa el número de cláusulas que se romperían (i.e. se harían falsas) si se invierte la variable; y el valor *make*, que denota el número de cláusulas que se harían ciertas. La puntuación de la variable *i* se calcula como:

  $$score_i = make_i - break_i$$

  Estos valores (*break y make*) se calculan de la siguiente forma: si una variable es el único soporte de una cláusula, se aumenta el valor de *break*, ya que si cambiamos el valor de la variable, pasará a ser falsa en esa cláusula y la romperá; si una cláusula es falsa, aumentamos el *make* de todas sus variables, pues si se inviertiese cualquiera de esas variables, la cláusula pasaría a ser cierta. Por tanto, el *score* de una variable representa

el *cambio neto* de la calidad al invertir esa variable. Si es positivo, el individuo mejorará su calidad, y empeorará si el *score* es negativo.

Tras asignar los valores de todas las variables, se selecciona aquella variable con mejor *score*.

- Una vez obtenida la mejor variable, simplemente se invierte el valor de esa variable, y se calcula la nueva calidad del individuo (utilizando la función de reevaluación, pues sólo se modifica una variable). Dado que el *score* es el cambio neto que se produce en la calidad del individuo al invertir una variable, la nueva calidad puede calcularse como la calidad antigua (antes de invertir la variable) sumada con el score. No obstante, se deben hacer más cálculos para actualizar el resto de atributos del individuo.

En resumen, el proceso es el siguiente: primero se elige la variable con mayor puntuación, luego se invierte su valor y finalmente se evalúa el individuo.

Esta heurística funciona muy bien porque explota la estructura del problema, pero tiene la desventaja de que es fácil que el proceso se quede atascado en máximos locales. Para corregir esto es común utilizar esta heurística combinada con otras que le ayuden a escapar de los mínimos locales. En este trabajo se ha optado por RandomWalk.

3.5.2 Implementación RandomWalk

Se ha elegido esta heurística con el objetivo de ayudar al algoritmo a escapar de mínimos locales en los que pueda caer. Además, este método tiene la ventaja de que requiere muy poca computación.

Se elige una cláusula aleatoria del conjunto de cláusulas insatisfechas, para lo cual se hace uso del atributo *unsatCl* de los individuos. Como se ha explicado anteriormente, este atributo es un vector que almacena los índices de las cláusulas insatisfechas. De esta cláusula, se elige una literal aleatorio, y se cambia su valor.

Cambiar el valor de un literal de una cláusula insatisfecha hará que inmediatamente esta cláusula pase a ser cierta, debido a que los literales de las cláusulas están unidos por disyunciones, con lo que es suficiente con que uno sea cierto. Pero también modificar el valor de una variable afectará a otras cláusulas, pudiendo romperlas. Por tanto, esta heurística puede empeorar a los individuos, pues no controla de ninguna forma que las variables que cambia no rompan más cláusulas de las que satisfacen. De todas formas, el objetivo principal de este método no es mejorar el individuo, sino ayudarlo a escapar de máximos locales y explorar el espacio de búsqueda.

3.6 Salida

La salida del resolutor sigue las normas establecidas por la MaxSAT Evaluation[2] 2020, y está inspirada en el formato DIMACS. Según esta especificación, el programa no puede escribir a ningún fichero, solo a la salida estándar y a la salida de error estándar, aunque solo se tendrá en cuenta la salida estándar. Los posibles tipos de líneas de esa salida son los siguientes:

- **Comentarios ('c' lines)**: estas líneas empiezan por dos caracteres, la letra minúscula 'c' seguida por un espacio (código ASCII 32). Son opcionales y pueden aparecer en cualquier lugar de la salida del resolutor. En este programa se utilizan para mostrar los parámetros de configuración con los que se está ejecutando el resolutor, así como datos sobre el problema MaxSAT sobre el que se está ejecutando (número de cláusulas, literales y coste máximo).

- **Estado (*status*) de la solución ('s' lines)**: estas líneas empiezan por dos caracteres, la letra minúscula 's' seguida por un espacio (código ASCII 32). Sólo se permite una única línea de este tipo, y debe ser una de las siguientes:

  - *s OPTIMUM FOUND*: Cuando las últimas líneas 'o' y 'v' (explicadas a continuación) corresponden a una solución óptima.
  - *s UNSATISFIABLE*: Cuando el conjunto de cláusulas *hard* es insatisfactible, y el resolutor lo ha comprobado.
  - *s UNKNOWN*: En cualquier otro caso.

  Si no se devuelve ninguna *s-line*, se supone que es *UNKNOWN*. Los resolutores completos están obligados a devolver *OPTIMUM FOUND* para puntuar, mientras que en el caso de los incompletos, lo más normal es devolver *UNKNOWN*. Podrían devolver la solución óptima, pero ello no conlleva puntuación extra.

  Dado que el programa desarrollado en este trabajo es un resolutor incompleto en el que no se pretende probar que una solución dada es óptima o no, siempre se devuelve *UNKNOWN*.

- **Coste de la solución ('o' lines)**: estas líneas empiezan por dos caracteres, la letra minúscula 'o' seguida por un espacio (código ASCII 32). Después continúan con un entero que representa el coste de la mejor solución encontrada, es decir, la suma de los pesos de las cláusulas falsas.

- **Asignación de verdad ('v' lines)**: estas líneas empiezan por dos caracteres, la letra minúscula 'v' seguida por un espacio (código ASCII 32). Después continúan con una

[2] https://maxsat-evaluations.github.io/2020/

secuencia de 0s y 1s, uno por cada variable del problema, que representan una solución con el coste dado por la línea 'o' correspondiente.

Un ejemplo de la salida del resolutor:

```
c ------------------------------------------
c BINARY DIFFERENTIAL EVOLUTION MAXSAT resolutor
c Generations   = 250
c Population    = 100
c Crossover     = 0.50
c Mutation      = 0.50
c Repetitions   = 1
c RW            = 0.50
c Seed          = -1
c LS Step       = 2 (0.01%)
c Max LSS       = 50
c Clauses       = 6120
c Literals      = 153
c Total cost    = 3064081
c h - scope     = all
c ------------------------------------------

s UNKNOWN
o 2131
v 10001010111100101101000010111001001100111
01000011010011110111010101010010011001100001
11101010101011001100011001010010001111001111
00001011010101110111010
```

Para evaluar los resolutores incompletos, se establece un tiempo límite (*timeout*) para dar una solución, y gana el resolutor que mejor solución alcance. El comando *timeout* de Unix envía una señal SIGTERM cuando finaliza el tiempo establecido. Esta señal debe ser recogida por el programa para imprimir las lineas 'o' y 'v' justo antes de que finalice la ejecución.

3.7 Implementación

Todo el DEMaxSATSolver está codificado en lenguaje C, empleando únicamente librerías estándar de Unix. Consta de, aproximadamente, 1200 líneas de código, repartidas entre 10 ficheros fuente. El proyecto está estructurado de forma que existe un fichero principal *"main.c"* desde el cual se inicia el programa, y tres módulos en los que se implementan diferentes funcionalidades del programa:

- *SettingParser*: se encarga de leer y procesar el fichero de configuración.

- *WCNFParser*: es el módulo que procesa los ficheros en formato DIMACS que contienen instancias de MaxSAT. Asimismo, también incluye las estructuras de datos necesarias para almacenar dichas instancias y poder trabajar con ellas.
- *DESolver*: es el módulo principal. Dentro se encuentra la codificación de los individuos y el algoritmo ED adaptado para MaxSAT, así como las heurísticas de búsqueda local empleadas para refinar las soluciones.

Además, el proyecto incluye una serie de scripts escritos en Python para generar las gráficas del Capítulo 5. Para la generación de estas gráficas se han empleado además las librerías Numpy y Matplotlib. El código fuente está disponible en el repositorio del proyecto [43].

En el Anexo A se puede encontrar una descripción más detallada de todos los ficheros que componen el resolutor desarrollado.

Capítulo 4

Proceso de ingeniería

En este capítulo se describen los aspectos relacionados con la gestión del proyecto, cómo ha sido el ciclo de desarrollo del mismo y el equipo y herramientas empleadas.

4.1 Metodología

Es difícil hablar de una metodología concreta en este trabajo pues no nos hemos ajustado a ninguna metodología de la literatura en particular, si bien siempre hemos tenido claro que queríamos un desarrollo basado en *métodos ágiles*[1] y enfocado a la creación de prototipos funcionales en cada fase. Es por ello que se ha optado por una aproximación a medio camino entre SCRUM y un ciclo de desarrollo en espiral.

SCRUM es una metodología muy enfocada al desarrollo de un producto para un cliente (el propietario del producto). En este sentido, durante el desarrollo de este proyecto se han realizado una serie de iteraciones, de forma que al final de cada una, el alumno presentaba un prototipo funcional al "cliente" (los tutores del trabajo). Esto se realizaba con una periodicidad preestablecida, acordando llevar a cabo reuniones semanales, en las cuales se revisaba el trabajo realizado y se planteaban los objetivos para la próxima semana. Otros aspectos de la metodología SCRUM que se han tenido en cuenta es el desarrollo de *sprints* (iteraciones), en promedio de unas dos semanas, el mantenimiento de una pequeña agenda de producto (*product backlog*) con las funcionalidades a completar, así como una agenda de iteración (*sprint backlog*) con los objetivos de la iteración correspondiente. Dado que los requerimientos eran habitualmente claros (el problema a resolver estaba netamente definido) estas agendas se mantenían como simples ficheros de texto con puntos a completar.

No obstante, no se puede decir que se haya utilizado la metodología SCRUM al uso, pues una de las partes importantes de esta metodología es el trabajo en equipo y la repartición de tareas, y este proyecto ha sido llevado a cabo por un único alumno junto con los directores

[1] https://agilemanifesto.org/

(que actuaban más como propietarios de producto). Por ejemplo, no existía la figura de *Scrum Master* dado que el equipo de desarrollo era unipersonal.

Respecto al ciclo de desarrollo en espiral, la similitudes con esta metodología residen en el desarrollo incremental que se ha llevado a cabo. Se han realizado una serie de iteraciones, y al final de cada una se obtenía una implementación funcional de la herramienta. Después, se realizaban pruebas sobre este prototipo, identificando los aspectos a mejorar y estableciendo los objetivos para la siguiente iteración o incluso diseñando nuevos experimentos (por ejemplo, variantes de heurísticas).

Las pruebas han consistido en evaluar la calidad de las soluciones obtenidas por el resolutor en un conjunto de benchmarks.

4.2 Planificación

En la figura 4.1 se pueden ver las diferentes fases que se han seguido durante el desarrollo del proyecto. Como se puede apreciar, han sido implementadas varias versiones del resolutor, tras las cuales se realizaron las correspondientes pruebas. Estas pruebas consistieron principalmente en ejecutar el DEMaxSATSolver sobre un conjunto de benchmarks, y comparar los resultados con los de la versión anterior, o empleando diferentes valores para los parámetros.

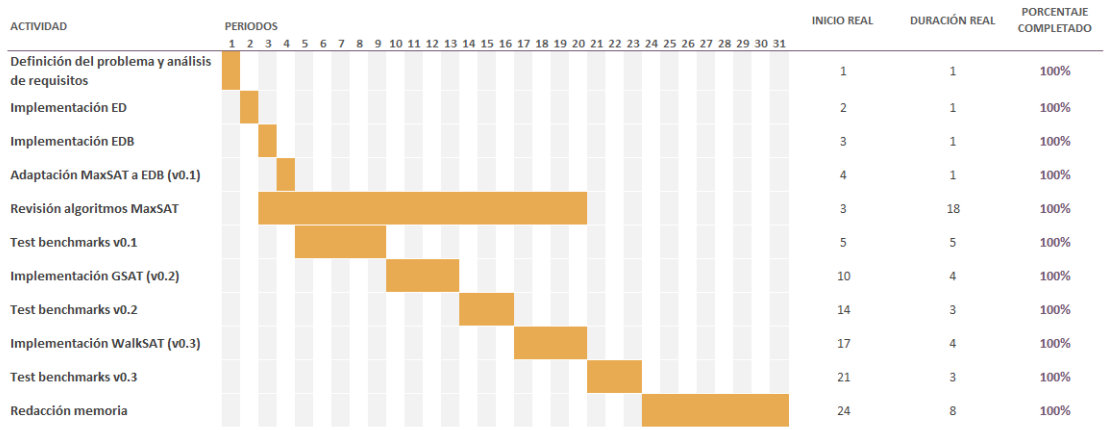

Figura 4.1: Planificación del trabajo

La mayoría de tareas se suceden de forma lineal, al finalizar una empieza la siguiente, con la excepción de la tarea de revisión de los algoritmos ya existentes de MaxSAT, que se ha prolongado a lo largo de gran parte del ciclo de desarrollo del proyecto.

Cada periodo en el diagrama corresponde a una semana de trabajo, por lo que el proyecto ha abarcado casi 8 meses (31 semanas), en los que se ha trabajo una media de 10 horas semanales. Como se ha mencionado anteriormente, las reuniones con los directores tuvieron lugar semanalmente, con una duración aproximada de una hora. El coste por hora está extraído de

la *Calculadora de Contratos de la UDC*[2]. Según la calculadora y teniendo en cuenta la titulación de cada perfil, al desarrollador (con un contrato de "Ayudante de apoyo a la investigación") le corresponde un sueldo de 1125€ brutos al mes y 37.5 horas semanales (repartidos en 12 pagas), lo que supone un coste por hora de

$$\frac{1125€}{37.5 \text{ horas} * 4 \text{ semanas}} = 7.5€/hora$$

mientras que para los directores del proyecto hemos usado el sueldo de "Investigador asociado" que serían 1714.29€, esto es:

$$\frac{1714.29€}{37.5 \text{ horas} * 4 \text{ semanas}} = 11.43€/hora$$

En la Tabla 4.1 se puede observar una estimación del coste total de los recursos humanos del proyecto.

Recurso	**Horas (h)**	**Coste/hora (€/h)**	**Coste total (€)**
Desarrollador	310	7.5	2325.00
Director 1	31	11.43	354.33
Director 2	31	11.43	354.33
Coste total de los recursos (€)			3033.66

Tabla 4.1: Coste estimado de los recursos humanos del proyecto

Por otro lado, todas las herramientas software que se han usado para desarrollar el proyecto son gratuitas, y no se ha requirido ningún hardware adicional, por lo que el coste de los recursos materiales es 0€. Pueden verse en detalle las herramientas utilizadas y el coste calculado en la tabla 4.2.

Recurso	**Coste (€)**
gcc	0
Intérprete de python	0
Git	0
Matplotlib	0
Coste total de los recursos (€)	0

Tabla 4.2: Coste estimado de los recursos materiales del proyecto

[2] https://rede.udc.es/calculadoraContratos/

4.3 Equipo y herramientas empleadas

El solver DEMaxSATSolver ha sido desarrollado enteramente en lenguaje C, compilado con *gcc* y empleando únicamente librerías estándar. No obstante, para generar las gráficas comparativas se ha empleado Python 3.8.5 con un *virtualenv*, donde se han instalado las librerías Numpy y Matplotlib. Como herramienta de control de versiones se ha utilizado Git y un repositorio de Github.

Las especificaciones del sistema en las que se ha desarrollado el trabajo son las siguientes:

- **SO:** Kubuntu 20.04
- **Tipo de SO:** 64 bits
- **CPU:** 8 × Intel® Core™ i5-9300H CPU @ 2.40GHz
- **Memoria:** 15,5 GiB de RAM
- **GPU:** Nvidia GeForce GTX 1650

Capítulo 5

Evaluación

En este capítulo se describe cómo se comporta el DEMaxSATsolver frente a problemas MaxSAT reales, presentados en la MaxSAT Evaluation 2020 [5].

Dado que el resolutor no está adaptado para trabajar con cláusulas *hard*, se ha elegido un conjunto de benchmarks que no contienen cláusulas de este tipo. Todos los benchmarks escogidos forman parte del conjunto de benchmarks incompletos, lo que significa que la solución óptima para estos casos de prueba se desconoce. En muchas ocasiones se consigue alcanzar una solución que podría ser la óptima, pero no se puede demostrar o no se ha conseguido todavía. No obstante, se utiliza como *ground truth* el coste de la mejor solución encontrada por cualquier resolutor, que nos ha sido proporcionada personalmente por la organización de la Evaluación MaxSAT.

Los bechmarks elegidos están divididos en dos conjuntos, uno de instancias *weighted*, es decir, donde las cláusulas tienen pesos, y otro conjunto *unweighted* donde todas las cláusulas tienen peso 1, y cada conjunto se evalúa por separado.

Para evaluar los diferentes métodos desarrollados en nuestro resolutor y compararlo con otros resolutores incompletos presentados en la evaluación MaxSAT 2020, hemos utilizado el mismo esquema de puntación que en dicha competición, de forma que a cada resolutor se le otorga una puntuacióon de acuerdo a la expresión:

$$\sum_{i \in \text{instancias resueltas}} \frac{(\text{coste de la mejor solución conocida para i}) + 1}{(\text{coste de la solución encontrada por el resolutor}) + 1} \tag{5.1}$$

donde el coste de la solución se corresponde con el peso de las cláusulas insatisfechas por dicha solución. Las mejores soluciones conocidas fueron solicitas a Fahiem Bacchus, uno de los organizadores de la MSE, ya que no estaban publicadas todavía[1].

Esta puntuación se promedia dividiéndola por el número de benchmarks utilizados. De esta forma, para cada resolutor se obtiene un valor entre 0 y 1 que indica cómo de cerca

[1] Actualmente las mejores soluciones conocidas ya están disponibles en [45]

se encuentra del mejor resultado encontrado para todos los casos de prueba. Cuánto más se aproxime la puntuación a 1, más cerca estarán las soluciones dadas por el resolutor de las mejores soluciones conocidas, es decir, mejor será ese resolutor.

5.1 Diferentes heurísticas empleadas

En este apartado se trata de determinar la importancia relativa de cada método utilizado, y decidir cuál es la versión del algoritmo que mejor funciona. Con este propósito se evalúan por separado las diferentes versiones del resolutor, denominadas:

- **Raw ED**: El algoritmo evolutivo aplicado a la resolución de problemas MaxSAT, sin incluir ninguna heurística adicional.
- **ED + RW**: El algoritmo evolutivo se combina con la heurística RandomWalk.
- **ED + GSAT**: El algoritmo evolutivo se combina con la heurística GSAT.
- **ED + GSAT + RW**: Se combina ED con las dos heurísticas. De aquí en adelante nos referiremos a este método como *ED + GWSAT*.

Además, en todos estos casos, las heurísticas se pueden aplicar a todos los individuos de la población (*all*) o sólo a los mejores individuos (*best*), lo que tendrá implicaciones en la calidad de las soluciones y en el tiempo necesario para alcanzarlas. Por tanto, el segundo objetivo de esta evaluación es decidir el alcance (*scope*) que tendrán las heurísticas de búsqueda local, es decir, sobre qué individuos se aplicarán.

5.1.1 Elección de los parámetros

Los parámetros empleados para realizar las pruebas han sido escogidos tras mucha experimentación a lo largo del desarrollo del proyecto. Todos los métodos se ejecutan durante 250 generaciones con una población de 100 individuos. Los parámetros de cruce (CR) y mutación (F) están fijados a 0.5.

El parámetro LSS que controla la cantidad de pasos de búsqueda local que se realizan en cada generación está fijado a 0.01 (es decir, un 1% del número de variables). Se ha comprobado[2] que valores más grandes provocan que se ejecuten demasiados pasos de búsqueda local, ralentizando el algoritmo y haciendo que estas heurísticas tengan un peso excesivo en las soluciones.

En la Figura 5.1 se puede observar la evolución de la calidad (peso de las cláusulas insatisfechas) a lo largo de 250 generaciones, variando el parámetro LSS entre 1%, 5% y 10%. El

[2] Todas las pruebas que se muestran en este apartado han sido ejecutadas con la variante ED+GWSAT aplicando las heurísticas a todos los individuos.

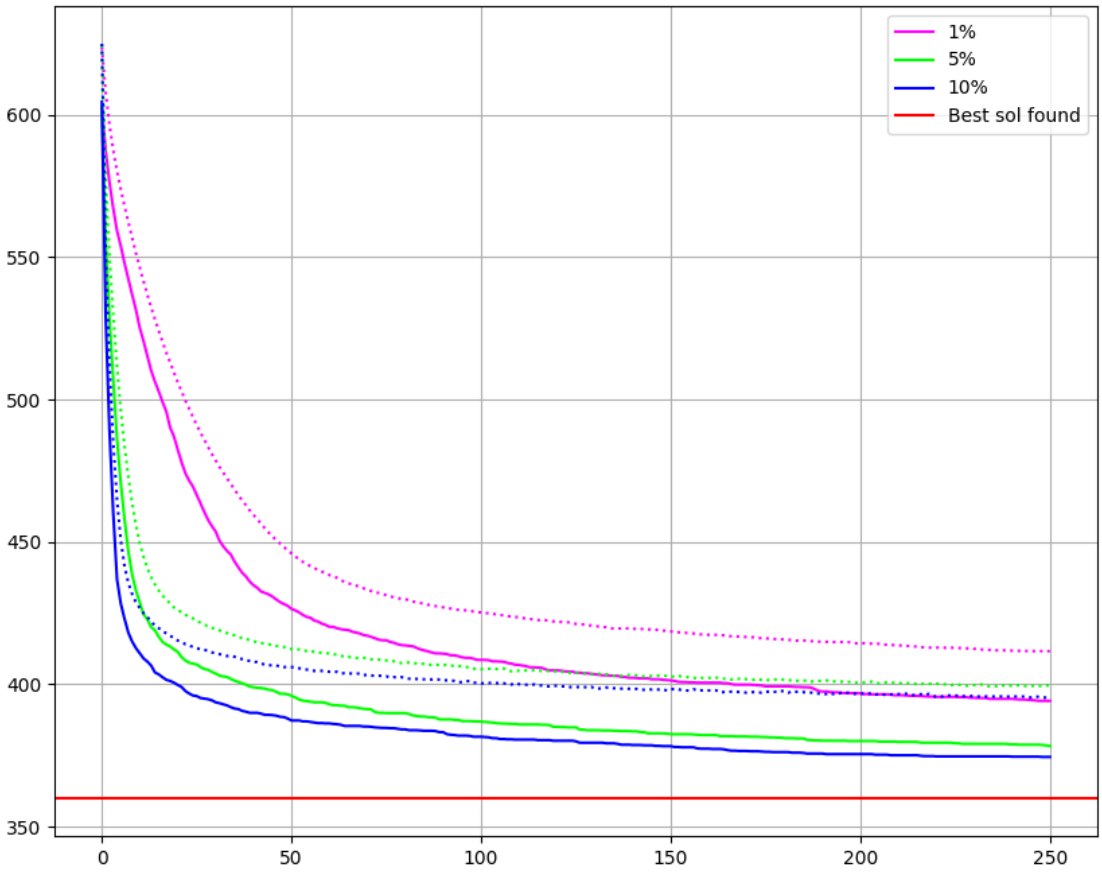

Figura 5.1: Evolución de la calidad (eje Y) a lo largo de 250 generaciones (eje X) con diferentes valores de LSS. Benchmark *scpcyc08_maxsat*.

benchmark *scpcyc08_maxsat* tiene 1024 variables y 2816 cláusulas, por lo que se ejecutan 10, 50 y 100 pasos de búsqueda local en cada generación, respectivamente. No obstante, esta comparación es injusta, pues no consume el mismo tiempo una generación en la que se realicen 10 pasos de búsqueda local que una en la que se realicen 100.

Tal y cómo refleja en la Figura 5.2, que muestra la evolución de la calidad respecto al tiempo durante 250 generaciones, el tiempo que consume cada método aumenta considerablemente al aumentar el valor de LSS. A pesar de que la calidad también parece aumentar conforme aumenta LSS, debemos tener en cuenta que uno de los objetivos del DEMaxSATsolver es alcanzar soluciones buenas en un periodo corto de tiempo, es decir, se tiene que asumir un compromiso entre la calidad de las soluciones y el tiempo necesario para alcanzarlas. Por otro lado, si fijamos un tiempo máximo (es decir, un timeout), los métodos con un LSS bajo ejecutarán más generaciones (pues cada generación consume menos tiempo) y por tanto llevarán a cabo una mayor exploración del espacio de búsqueda. Esto puede comprobarse en la Tabla 5.1, que muestra los resultados obtenidos en el benchmark *scpcyc08_maxsat* con un

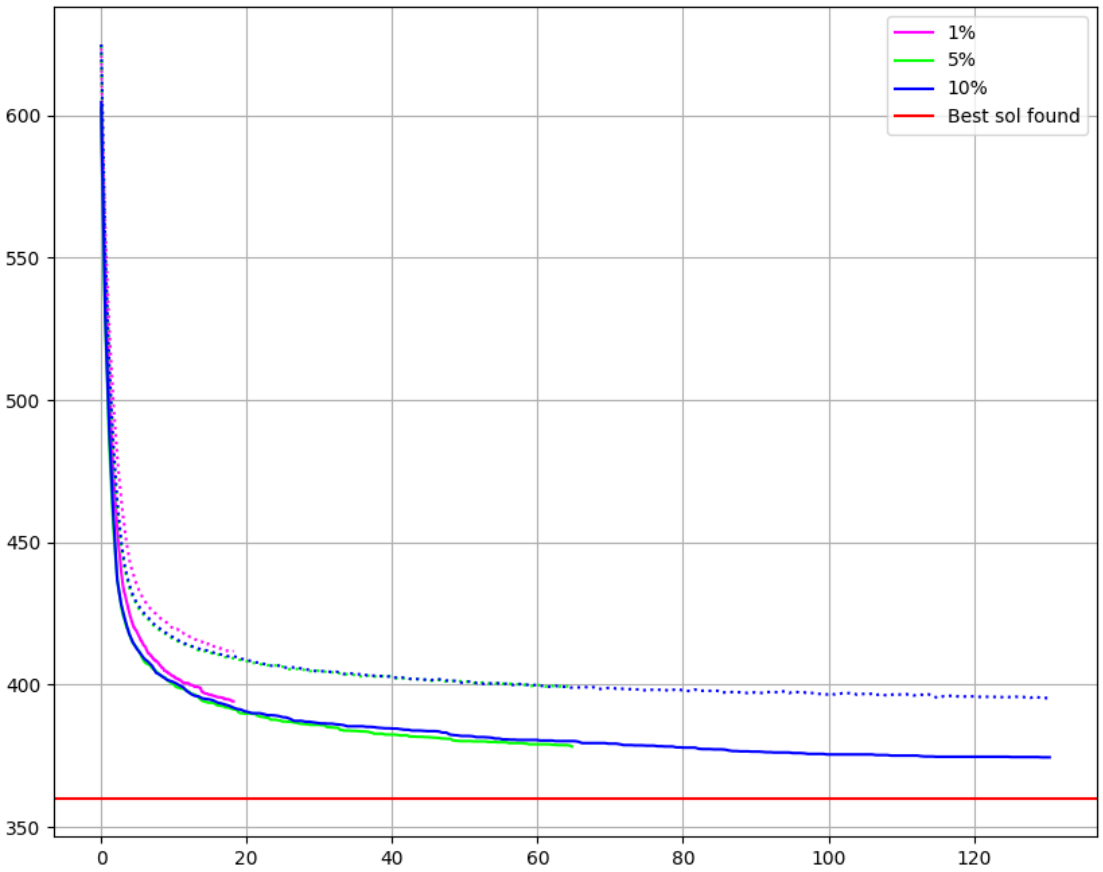

Figura 5.2: Evolución de la calidad (eje Y) a lo largo del tiempo (segundos) (eje X) con diferentes valores de LSS, durante 250 generaciones. Benchmark *scpcyc08_maxsat*.

timeout de 60 segundos, promediados entre 10 ejecuciones independientes. Se puede apreciar como todos los valores de LSS obtienen resultados similares (el mismo número de cláusulas insatisfechas), pero se realizan muchas más generaciones en los que tienen un valor LSS pequeño.

LSS	Media de la población	Mejor individuo	Nº Generaciones
1%	380.9	401.19	856.1
5%	379.7	400.12	226.2
10%	380.8	399.63	118.5

Tabla 5.1: Comparación de los resultados empleando diferentes valores de LSS con un timeout de 60s. Benchmark scpcyc08_maxsat.

Por la misma razón se limita el número máximo de pasos locales por generación (maxLSS) a 50, ya que en benchmarks muy grandes el 1% de las variables sigue siendo un número de-

masiado grande (por ejemplo, para un benchmark de 10000 variables, el 1% son 100).

En cuanto al parámetro RW, este dependerá del método: dado que representa la probabilidad de que se elija RW en cada paso, en los métodos que sólo emplean GSAT se fija a 0.0 (i.e 0% de probabilidad de ejecutar RW), mientras que en los que sólo emplean RW se fija a 1.0; los métodos que combinan GSAT y RW lo hacen con un 50% de probabilidad de ejecutar cada uno de ellos.

5.1.2 Resultados de las diferentes versiones

Los benchmarks usados en este apartado pertenecen a diferentes familias[3] y abarcan un rango de dificultades muy grande, desde instancias pequeñas con unos pocos cientos de cláusulas hasta instancias con más de 10.000 variables y 30.000 cláusulas. En total se han empleado 49 benchmarks (29 *unweighted* y 20 *weighted*), con características muy diferentes entre sí. En las Tablas 5.2 y 5.3 se resumen las características de todas las instancias evaluadas.

	Mínimo	**Máximo**	**Media**
Nº de variables	41	11264	822.10
Nº de cláusulas	432	39424	4874.62
Literales por cláusula	1	495	8.72

Tabla 5.2: Resumen de los benchmarks unweighted.

	Mínimo	**Máximo**	**Media**
Nº de variables	40	190	98.8
Nº de cláusulas	1001	9690	3160.4
Literales por cláusula	2	6	3.92
Peso de las cláusulas	1	1000	277.81

Tabla 5.3: Resumen de los benchmarks weighted.

Dado que DEMaxSATsolver es estocástico, se ejecutó 10 veces cada método sobre cada benchmark y se promediaron los resultados. A continuación se resumen los resultados de cada método al ejecutarlo durante 250 generaciones en los conjuntos de benchmarks unweighted (Tabla 5.5) y weighted (Tabla 5.4). Los valores corresponden a la medida previamente explicada en la Ecuación 5.1. No obstante, las comparaciones que se realizan en estas tablas son injustas, pues las versiones que emplean GSAT (ya sea total o parcialmente) consumen mucho más tiempo por generación que aquellas otras versiones que no lo hacen. Por tanto, es importante

[3] En la MSE los benchmarks se organizan por familias, de forma que cada familia codifica un problema real distinto como una instancia de MaxSAT.

Método	Media de la población	Mejor individuo	Tiempo medio
Raw ED	0.4465	0.4836	20m56s
ED+GWSAT:all	0.5478	**0.7197**	35m25s
ED+GWSAT:best	0.4565	0.6983	23m19s
ED+GSAT:all	**0.5764**	0.6680	45m02s
ED+GSAT:best	0.5048	0.6591	35m17s
ED+RW:all	0.3954	0.4801	19m36s
ED+RW:best	0.4002	0.4806	**15m58s**

Tabla 5.4: Comparación de los métodos en el conjunto weighted de benchmarks.

Método	Media de la población	Mejor individuo	Tiempo medio
Raw ED	0.6798	0.7127	**16m10s**
ED+GWSAT:all	0.8214	**0.9586**	43m19s
ED+GWSAT:best	0.7087	0.9502	26m18s
ED+GSAT:all	**0.8338**	0.9076	66m38s
ED+GSAT:best	0.7543	0.8992	41m12s
ED+RW:all	0.6452	0.7399	17m48s
ED+RW:best	0.6465	0.7383	16m40s

Tabla 5.5: Comparación de los métodos en el conjunto unweighted de benchmarks.

no fijarse sólo en la puntuación obtenida, sino también en el tiempo empleado por dicho método.

Como se puede apreciar en las tablas, las variantes que peor funcionan son las que emplean Raw ED y ED+RW, es decir, las que no implementan ninguna estrategia voraz (i.e. GSAT), mientras que las que sí las utilizan obtienen resultados significativamente mejores, siendo la que combina ED, GSAT y RW la que obtiene las mejores métricas en los dos conjuntos de benchmarks.

En relación a los tiempos de ejecución, los métodos más lentos son los que sólo emplean GSAT, mientras que los que combinan esta estrategia con RandomWalk, son más rápidos. Esto se debe a que la heurística GSAT requiere mucho más cómputo que RandomWalk, que tiene un coste computacional muy bajo[4]. Un resultado que llama la atención es que aquellos métodos que emplean RW sean más rápidos que Raw ED, que no emplea ningún método. La explicación parte del hecho de que al aplicar RW se suelen romper más cláusulas de las que se satisfacen, por lo que la calidad de los individuos empeora, y durante el paso de selección del evolutivo no es necesario actualizar tales candidatos con peor calidad, con lo que se ahorra la operación de copia profunda, que es costosa ya que debe iterar sobre todos los arrays que

[4] Recordemos que RW elige un literal aleatorio de una cláusula escogida aleatoriamente, dos operaciones con coste ínfimo.

mantiene un individuo.

Finalmente, respecto a los métodos que emplean las heurísticas sobre todos los individuos y aquellos que las aplican sobre los mejores, los primeros obtienen resultados ligeramente superiores, a cambio de tiempos de ejecución claramente más largos que los segundos. Por otro lado, la calidad media de la población es mayor en aquellos métodos que emplean GSAT sobre toda la población.

5.1.3 Ejemplo de ejecución sobre un benchmark

Las diferencias entre los métodos se pueden apreciar mejor si desglosamos los resultados por benchmarks[5]. Para ejemplificar el funcionamento de los diferentes métodos, se adjuntan las gráficas correspondientes al benchmark unweighted *scpcyc08_maxsat*, que tiene 1024 variables y 2816 cláusulas. Recordemos que cada individuo mantiene una asignación de verdad de todas las variables, y la calidad de ese individuo se corresponde con el peso de las cláusulas que satisface dicha asignación. No obstante, es importante tener en cuenta que maximizar este valor es equivalente a minimizar el peso de las cláusulas insatisfechas (y se puede traducir un valor al otro fácilmente). De esta última forma es cómo se representa la calidad de los individuos en todas las gráficas que se muestran en este capítulo. Además, las gráficas son el promedio de 10 ejecuciones independientes del correspondiente algoritmo evolutivo híbrido, debido al carácter estocástico de las evoluciones.

En la gráfica 5.3 se muestra la relación entre la calidad de las soluciones a lo largo de las generaciones[6]. Como se ha mencionado anteriormente, los métodos Raw ED y ED+RW obtienen resultados notablemente peores que los que emplean GSAT y GWSAT. Es interesante observar cómo los métodos ED+GSAT se quedan atascados en un mínimo local rápidamente (sobre la generación 25) y no consiguen escapar, mientras que los que incorporan GWSAT tardan unas pocas generaciones más en alcanzar dicho mínimo, pero logran superarlo.

[5] Las gráficas de todos los benchmarks se pueden encontrar en el repositorio del proyecto [43].

[6] Las líneas continuas representan la calidad del mejor individuo de cada generación, mientras que las líneas de puntos representan la calidad media de la población.

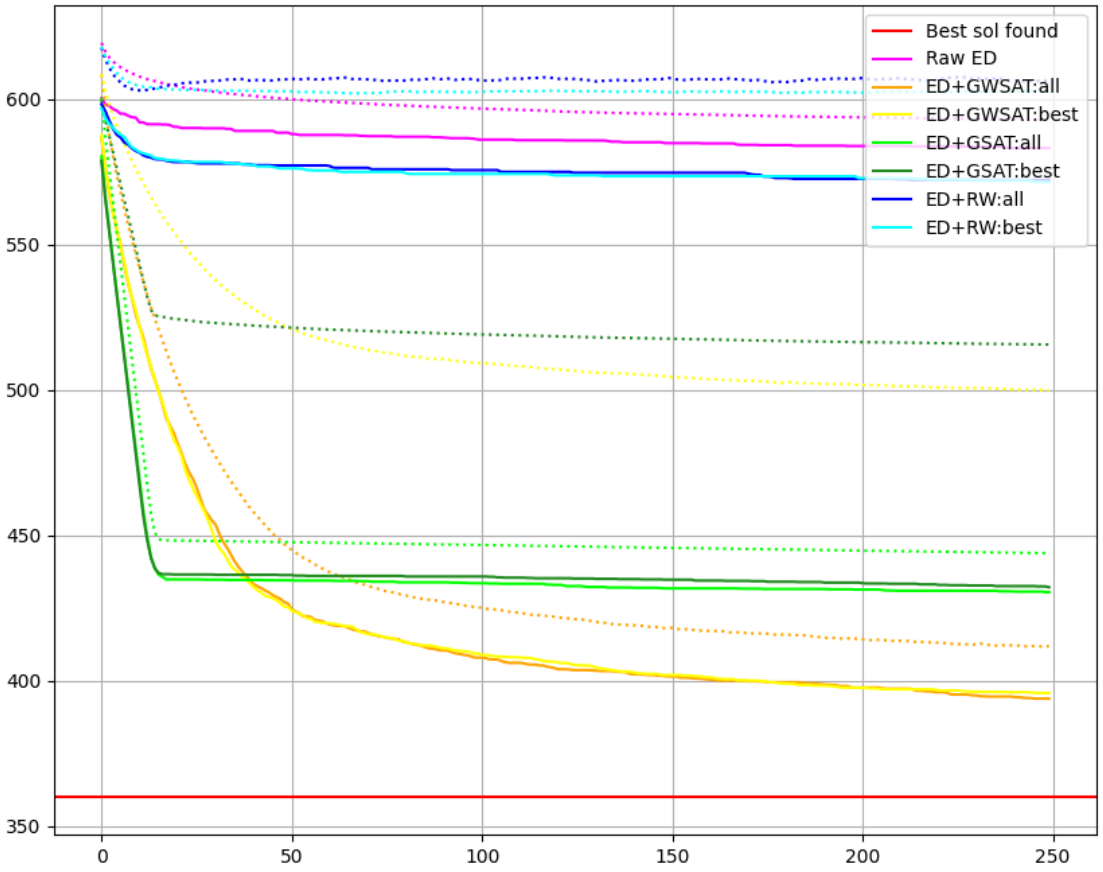

Figura 5.3: Evolución de la calidad (peso de las cláusulas insatisfechas, en el eje Y) a lo largo de las generaciones (eje X). Benchmark *scpcyc08_maxsat.wcnf*

Por contra, en la Figura 5.4 se muestra cómo evoluciona la calidad de las soluciones a lo largo del tiempo. De nuevo, la gráfica corresponde al promedio de 10 ejecuciones independientes de 250 generaciones de las diferentes versiones del algoritmo. Como se puede ver en esta gráfica, los métodos que emplean GSAT son considerablemente más lentos que aquellos que no lo hacen (o lo combinan con RW), aunque también consiguen mejores soluciones. También se puede apreciar cómo aquellos métodos que aplican GSAT a todos los individuos consumen más tiempo que los que sólo lo aplican sobre los mejores individuos.

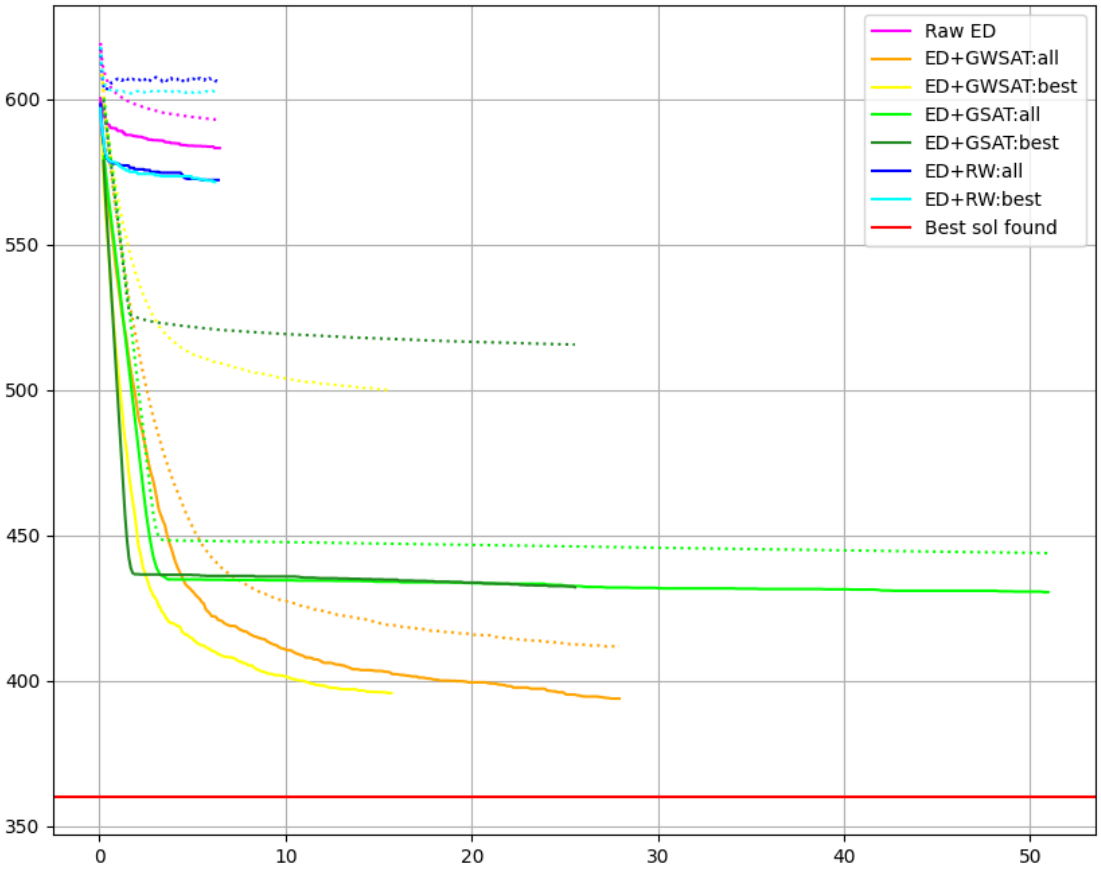

Figura 5.4: Evolución de la calidad (eje Y) a lo largo del tiempo de ejecución (segundos, eje X), cuando todas las diferentes alternativas se ejecutan durante 250 generaciones. Benchmark *scpcyc08_maxsat.wcnf*

En las gráficas también se puede apreciar cómo afecta a la población aplicar las heurísticas a todos los individuos o solo a los mejores. La calidad media de la población de los primeros métodos se acerca mucho más a la calidad de los mejores individuos, mientras que (como es esperable), en el resto de métodos la diferencia entre la media de la población y el mejor individuo es mucho mayor.

Finalmente, para poder comparar simultáneamente la calidad respecto a las generaciones y respecto al tiempo, se adjunta la Figura 5.5. Esta gráfica, a pesar de ser menos intuitiva que las gráficas anteriores, permite visualizar al mismo tiempo todos los datos de interés que estamos comentando. En el eje X se encuentran las generaciones (250), que son las mismas para todos

los métodos. El eje Y representa la calidad de las soluciones, es decir, el número de cláusulas insatisfechas: las mejores soluciones se encuentran más cerca, mientras que las peores (que se corresponden con los métodos Raw ED y ED+RW) están al fondo. Por último, el eje Z (la altura) representa el tiempo consumido por cada método. Se puede apreciar claramente que los métodos que emplean sólo GSAT (mostrados en verde) alcanzan mayor altura que los demás, es decir, requieren más tiempo.

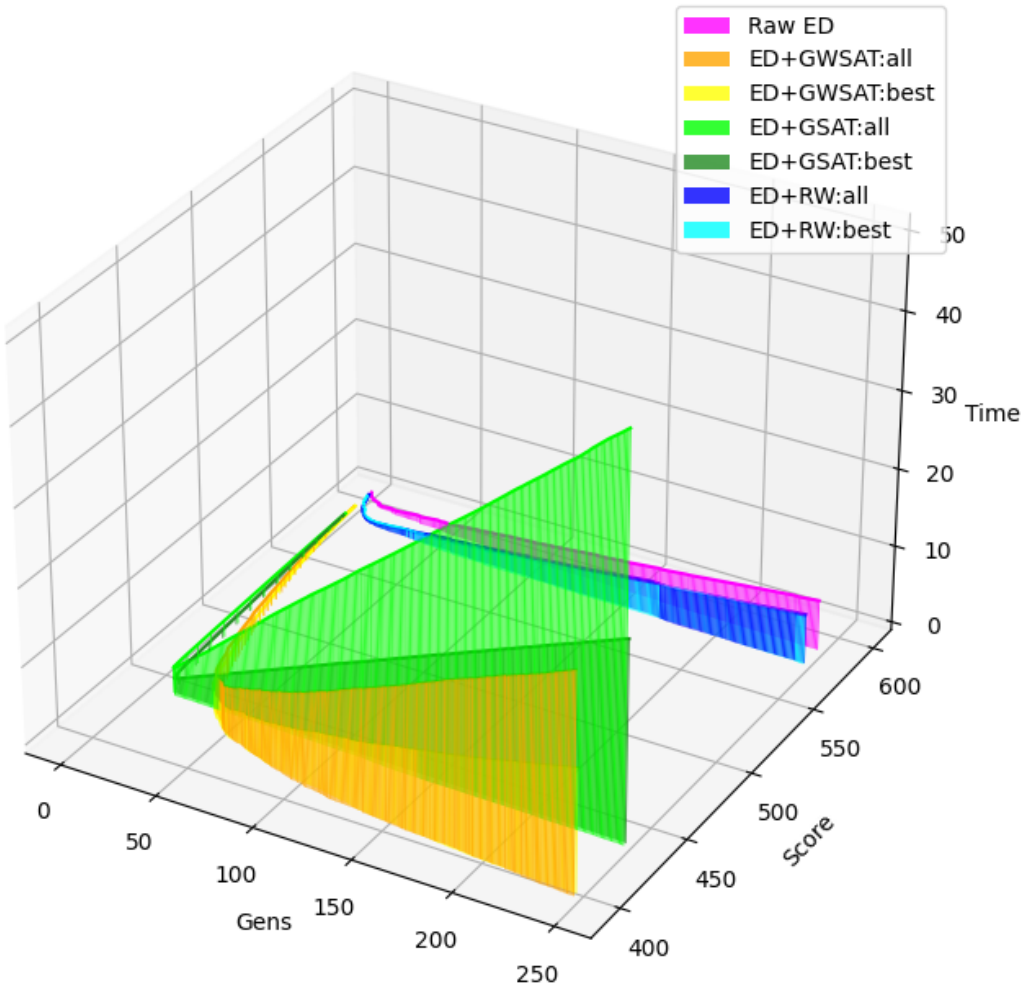

Figura 5.5: Evolución de la calidad (eje Y) a lo largo de las generaciones (eje X) y el tiempo (segundos) (altura, eje Z). Benchmark *scpcyc08_maxsat.wcnf*

A diferencia de los resolutores completos, cuyo objetivo es obtener el coste óptimo de una instancia MaxSAT, en los resolutores incompletos es necesario adoptar un compromiso entre la eficacia (cómo de buenos son los resultados) y el tiempo que se necesita para obtenerlos. Por tanto, consideramos que el método que emplea ED+GWSAT, aplicándolo solamente a los mejores individuos, es el que mejor cumple los requerimientos de calidad y velocidad, ya

que proporciona soluciones de buena calidad (gracias a GWSAT) en un lapso de tiempo corto (gracias al alcance restringido de las heurísticas). Este es, por tanto, el método que se emplea en el siguiente apartado para comparar el DEMaxSATsolver con otros resolutores incompletos.

5.2 Otros algoritmos incompletos

En esta sección se compara el funcionamiento del resolutor desarrollado DEMaxSATsolver con varios resolutores incompletos presentados en la Evaluación MaxSAT de 2020. Estos resolutores son *Satlike* [22], *StableReresolutor* [23], *Loandra-2020* [46] y *TT-Open-WBO-Inc-20* [47].

Las pruebas se han realizado con el mismo conjunto de benchmarks que en la sección previa. Además, siguiendo el formato de la MSE, en las pruebas se ha establecido un timeout de 60 segundos para cada benchmark, de forma que los resolutores tendrán que dar una solución en ese tiempo. De nuevo, y debido al carácter estocástico del DEMaxSATsolver, los resultados presentados están promediados con 10 ejecuciones.

resolutor	**Weight**	**Unweight**
satlike-cw	**0.9633**	**0.9752**
StableReresolutor	0.8868	0.9730
DEMaxSATsolver	0.8565	0.9300
Loandra-2020	0.7519	0.9332
TT-Open-WBO-Inc-20	0.7927	0.8802

Tabla 5.6: Puntuaciones de los resolutores

Gracias a que los organizadores de la MSE obligan a los participantes a liberar los resolutores presentados con una licencia libre, ha sido posible ejecutarlos sobre el conjunto de benchmarks seleccionado. De esta forma, se ha calculado la puntuación de todos los resolutores en cada benchmark, empleando la Ecuación 5.1 descrita en la sección anterior. Estas puntuaciones han sido promediadas sobre todos los benchmarks, obteniendo una medida entre 0 y 1 que indica cómo de bueno es un resolutor en el conjunto de benchmarks (de nuevo, los mejores resolutores son los que más se aproximen a 1) .

El rendimiento del DEMaxSATsolver se aproxima mucho al estado del arte de los resolutores incompletos de MaxSAT en las instancias sin cláusulas hard. En la Tabla 5.6 se muestran las puntuaciones de cada resolutor, y se puede apreciar cómo DEMaxSATsolver es competitivo en comparación con otros resolutores presentados en la evaluación MaxSAT de 2020, superando en el conjunto weight de benchmarks a los resolutores *Loandra-2020* y *TT-Open-WBO-Inc-20*, y a este último en el conjunto unweighted, a pesar de que los otros resolutores

han sido desarrollados a lo largo varios años por profesionales dedicados al estudio del problema MaxSAT.

Para obtener una visión más detallada del funcionamiento del DEMaxSATsolver, en las Figuras 5.6 y 5.7 se pueden ver los resultados de todos los resolutores, desglosados para cada benchmark de los conjuntos weighted y unweighted, respectivamente. En el eje Y se encuentra la puntuación obtenida por cada resolutor (obtenida mediante la fórmula 5.1) y en el eje X se encuentran todos los benchmarks de este conjunto. Estas gráficas permiten visualizar rápidamente qué benchmarks resultan más fáciles o complicados para cada resolutor.

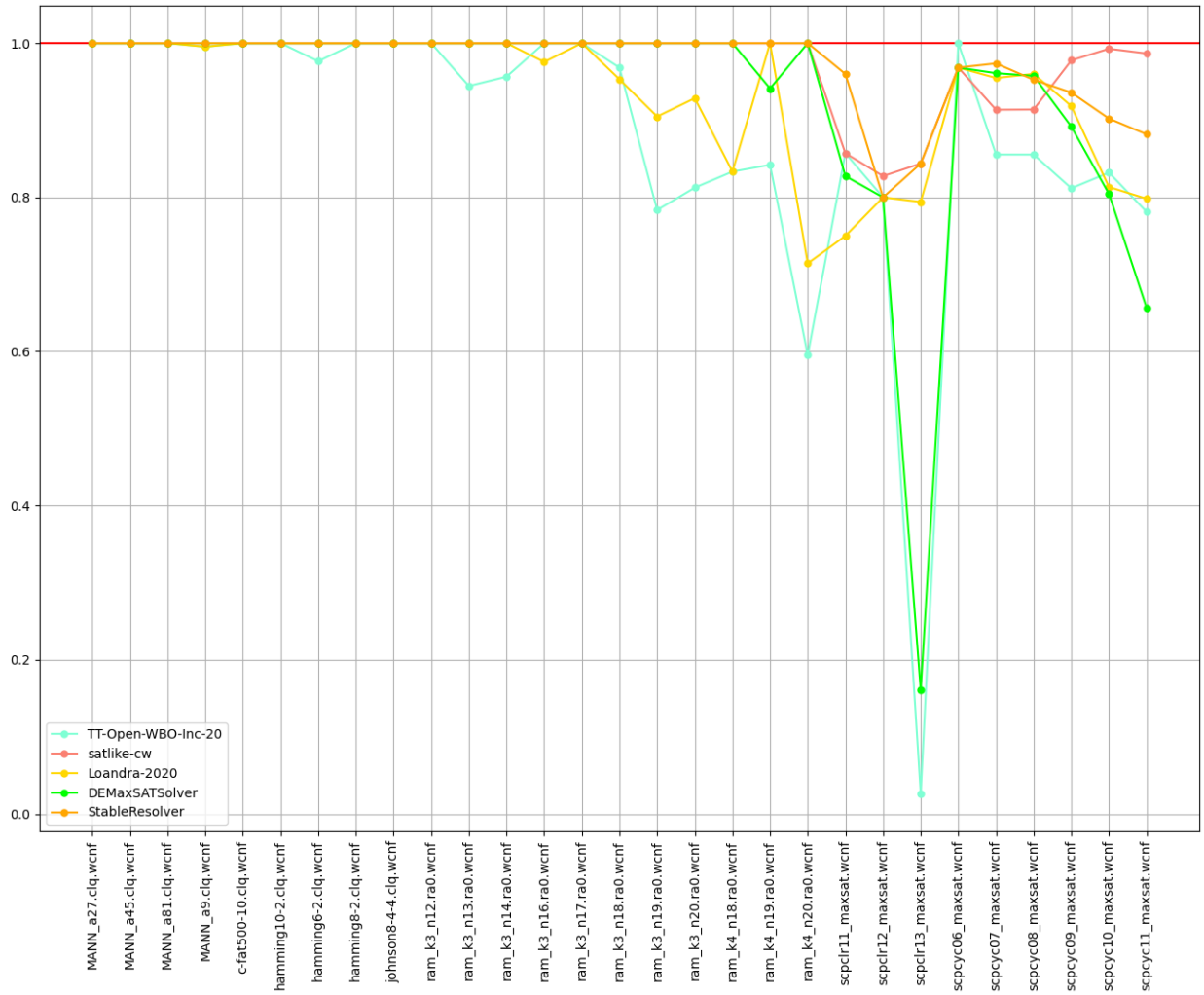

Figura 5.6: Comparación benchmarks unweighted (timeout 60s)

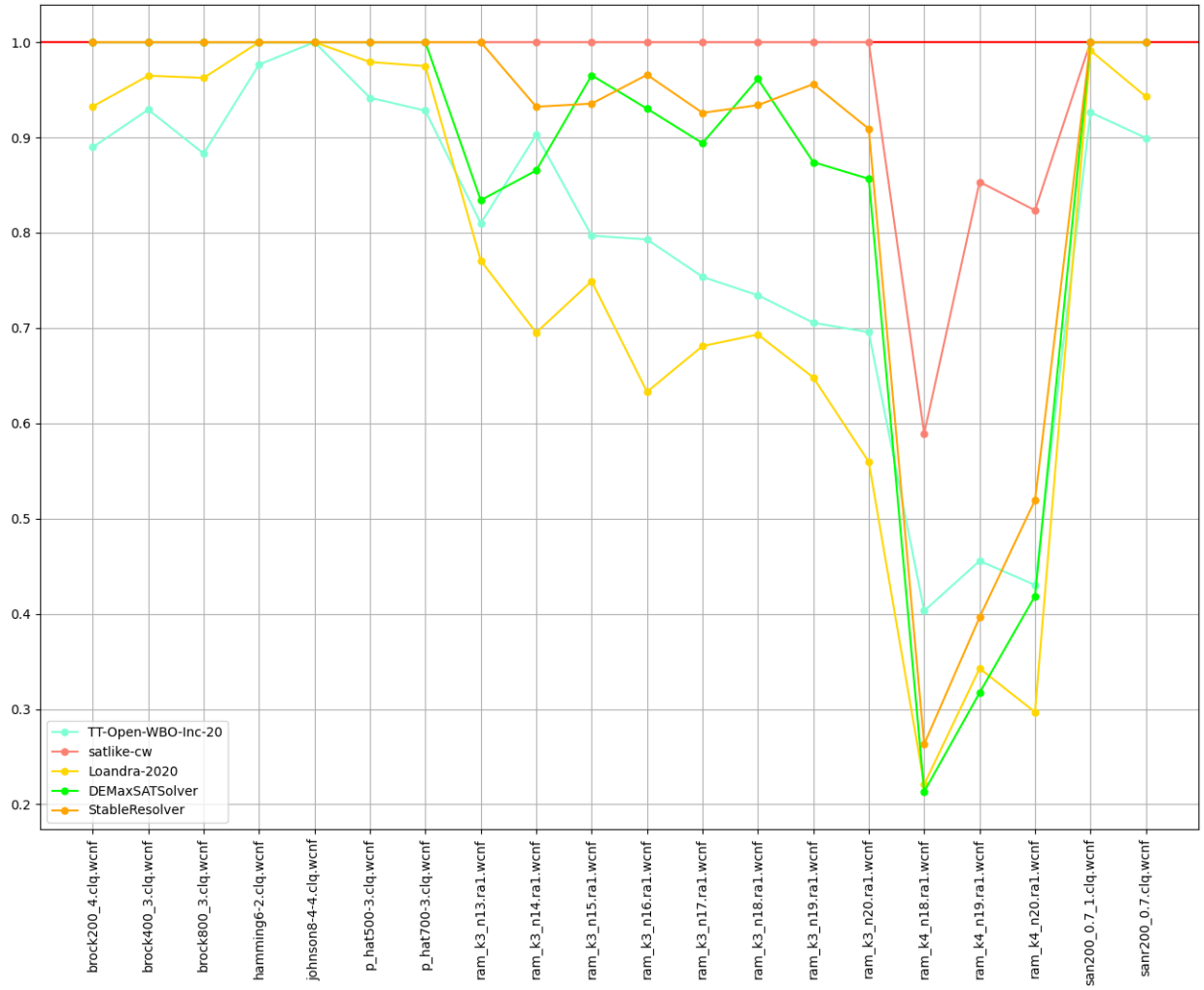

Figura 5.7: Comparación benchmarks weighted (timeout 60s)

Capítulo 6

Conclusiones

El problema MaxSAT es un problema complejo que combina la lógica proposicional con la optimización matemática, y tiene una gran cantidad de aplicaciones en el mundo real. En este proyecto se ha implementado un resolutor de MaxSAT capaz de encontrar soluciones aceptables en un tiempo limitado. El resolutor desarrollado, bautizado como DEMaxSATSolver, emplea un algoritmo evolutivo (Evolución Diferencial) integrado con dos heurísticas de búsqueda local desarrolladas específicamente para MaxSAT, GSAT y RandomWalk, para definir una combinación híbrida o memética *ad hoc* al problema. La búsqueda local permite refinar las soluciones de la población genética, mientras que la búsqueda global del algoritmo evolutivo permite realizar una búsqueda simultánea en varias zonas prometedoras del espacio de búsqueda.

En relación a los objetivos del proyecto, se ha cumplido el objetivo principal, que era desarrollar una herramienta de resolución de MaxSAT basada en ED. Para ello se ha empleado una adaptación de ED para trabajar en el dominio discreto, de forma que los individuos codifican soluciones del problema MaxSAT y se pueden aplicar los operadores genéticos de mutación y cruce sobre dichos individuos. Además, se ha integrado el algoritmo evolutivo con otros algoritmos de MaxSAT ya existentes. Finalemente, el resolutor desarrollado ha sido empleado para resolver instancias reales del problema MaxSAT (benchmarks) y los resultados han sido comparados con otras herramientas de resolución de MaxSAT.

Como conclusiones positivas, la contribución más destacable del presente trabajo es que se han alcanzado *resultados equiparables* al estado del arte de los resolutores incompletos de MaxSAT, superando incluso a algunos de ellos en determinados benchmarks. Este resultado es especialmente destacable teniendo en cuenta la sencillez del algoritmo base de evolución diferencial empleado y su fácil parametrización o adaptación a posibles variantes del problema. A la vista de la calidad de las soluciones obtenidas, el uso de ED ofrece una alternativa prometedora para su mejora futura: hay que tener en cuenta que la mayoría de las herramientas de la competición MaxSAT son el fruto del trabajo de grupos de investigación a lo largo de

varios años. Una ventaja importante del uso de ED es que el consumo de memoria depende del tamaño de la población elegido, pero no crece a lo largo del proceso de evolución, lo que puede suponer un factor de escalabilidad importante a la hora de resolución de problemas con aplicación práctica.

Por otro lado, la limitación de DEMaxSATSolver más evidente es sin duda la incapacidad para trabajar con cláusulas *hard*, es decir, no es capaz de resolver el problema MaxSAT parcial (PMS, Partial MaxSAT). Resolver este problema no es trivial, ya que se debe comprobar que existe un modelo para las cláusulas hard, se deben satisfacer todas ellas, a la vez que se maximiza el número de cláusulas soft satisfechas. Esta limitación además, es prohibitiva a la hora de participar en la Evaluación MaxSAT, dado que las herramientas deben resolver casos de PMS y se exige que las soluciones dadas por un resolutor satisfagan las cláusulas hard, o de lo contrario será descalificado. La dificultad de esta variante proviene del hecho de que, en el peor de los casos, un problema PMS puede derivar en un problema SAT (esto sucede si todas las cláusulas son hard). Esto complicaría las heurísticas para el algoritmo evolutivo, posiblemente requiriendo un análisis completamente distinto y que incluya la estructura de las cláusulas, lo que iría mucho más allá del alcance del presente trabajo fin de grado.

Para solventar el problema de las cláusulas hard se proponen varias aproximaciones que podrían funcionar. No han sido implementadas porque resolver el problema PMS se escapa del alcance del trabajo, pero se plantean para posibles mejoras futuras.

La primera (y la más "naive") es asignar una calidad muy baja (0, por ejemplo) a aquellos individuos que no satisfagan las cláusulas hard. Sin embargo, esto no resuelve el problema, ya que primero (y más importante) es que el algoritmo evolutivo encuentre una solución que satisfaga las claúsulas hard, lo que no está garantizado. Se puede asignar una calidad 0 si no se cumplen estas claúsulas o bien una penalización de fitness fuerte. Es, en definitiva, el problema de cómo pesar las restricciones en un problema mono-objetivo.

Por ello, se propone una segunda aproximación en la que se combine la primera con una inicialización a partir de un solver SAT, que dadas las cláusulas hard del problema, obtenga una asignación de verdad que las satisfaga.

Una tercera vía sería modificar el algoritmo evolutivo para que sea multiobjetivo, de forma que se intenten maximizar dos funciones de fitness, una para las cláusulas soft y otra para las hard. Al contrario que en la primera propuesta, con esta aproximación se optimizan de modo simultáneo ambos objetivos, tratando de encontrar progresivamente las soluciones "no dominadas" del Frente de Pareto [1].

Por último, los resultados muestran que los métodos implementados son efectivos y competitivos a la hora de resolver instancias de MaxSAT, por lo que, como trabajo futuro, se

[1] Una solución se denominada "no dominada" cuando no se puede mejorar el valor de ninguna de las funciones objetivo sin degradar alguna de las demás.

pretende seguir mejorando el DEMaxSATSolver para que sea capaz de lidiar con cláusulas hard y acelerar el procesamiento de las instancias más grandes, con el objetivo de presentarlo en la Evaluación MaxSAT de 2022 en la categoría de solvers incompletos.

Apéndices

Apéndice A

Descripción de los ficheros fuente

Todos los ficheros que se describen a continuación pueden encontrarse en el repositorio del proyecto [43]. En resolutor desarrollado se compone de un total de 10 ficheros fuente. El programa se inicia desde el *main.c*, y el resto de archivos se reparten en tres módulos, tal y cómo se explica a continuación:

- Módulo *DESolver*. Contiene tres ficheros que codifican el algoritmo ED y las heurísticas empleadas.
  - *BDE.c*: En este archivo se encuentra el algoritmo ED en sí, las heurísticas empleadas (GSAT y RandomWalk) y la función de fitness. Es el archivo más importante del programa.
  - *bitOperators.c*: contiene las operaciones a nivel de bit necesarias para trabajar con arrays binarios en C. Todas las operaciones están definidas como macros.
  - *individual.c*: en este fichero se define la estructura de datos que almacena los individuos, así como operaciones para crearlos (reservando la memoria necesaria), borrarlos, inicializarlos (aleatoriamente) y realizar un clonado profundo.
- El siguiente módulo es el *SettingParser*, que sólo contiene un fichero, *settingParser.c*, que es el encargado de leer y procesar el archivo de configuración.
- Finalmente, el módulo *WCNFParser* agrupa los ficheros encargados de leer y procesar los archivos en formato DIMACS. También se definen las estructuras de datos empleadas para almacenar las fórmulas proposicionales.
  - En *CNF_types.h* se definen los tipos para los literales, las cláusulas, las entradas (un índice invertido), y las fórmulas CNF. Los literales están definidos como enteros. Las cláusulas son arrays (dinámicos) de literales. En teoría, cada entrada debería mantener una lista de cláusulas, no obstante, sólo se almacenan sus índices. Finalmente, las fórmulas CNF se definen cómo una estructura que tiene una lista

de cláusulas, una lista de entradas, y variables para guardar los parámetros de la línea 'p' del formato DIMACS. Estas listas son arrays estáticos, ya que al leer un fichero se conocen tanto el número de cláusulas (que será el tamaño del array de cláusulas) cómo el número de variables (que será el tamaño del array de entradas).

- *Clause.c* implementa las operaciones necesarias para crear una cláusula (reservando la memoria necesaria), borrarla e introducir literales en ella.
- *Entry.c* implementa las operaciones necesarias para crear una entrada, borrarla e introducir los índices de las cláusulas.
- *CNF.c* implementa las operaciones para crear y borrar una fórmula CNF.
- Por último, el fichero *wcnf_parser.c* es el encargado de leer y procesar los ficheros DIMACS, y almacenar las fórmulas proposiciones en las estructuras de datos correspondientes.

Además, se ha subido al repositorio un directorio *Graphics*, dónde se encuentra el script empleado para generar las gráficas.

Apéndice B

README del repositorio

DEMaxSatSolver

DEMaxSATSolver is a incomplete solver for MaxSAT that combines a all-purposee genetic evolutionary algortihm like DE (Differential Evolution) with well-known local search heuristic for MaxSAT such GSAT and Random Walk.

The solver is capable of solve weighted and unweighted instances of MaxSAT, but it can't handle PMS (Partial MaxSAT) instances (yet).

Author: Manuel Framil de Amorín

Usage

Compile the program by typing

```
make
```

in the main folder. Call the solver using:

```
./main path/to/wcnf [gens_logs]
```

where the first argument "path to wcnf" is the path to the MaxSAT instance to be solved, in DIMACS format, as defined in the MSE (https://maxsat-evaluations.github.io/2020/). You can also give the path to a directory with several WCNF files, and they'll be solved one by one. For each generation, are saved: the best score, the best individual of this gen, the mean of population and the current computing time. These values are saved in "gens_logs". If no path is given, the program will generate a new one.

Alternatively, you can set a timeout, and the solver will return a solution within that time:

```
timeout TIME ./main path/to/wcnf [gens_logs]
```

Following the output format of the MSE, the solver will return several 'o' lines, containing the best cost founded so far (that is, the sum of weights of the unsatisfied clauses) and one 'v' with the assigment of the best solution found.

Results

The solver has been tested on 49 benchmarks submitted to the MSE'20, both weighted and unweighted, but all of them without hard clauses.

This solver has been developed in a very short time, it has achieved good results comparing with other solvers of the state of the art, tho. Here you can see a comparation between DEMaxSATSolver and other incomplete solvers presented on the MSE'20.

	Weight	Unweight
satlike-cw	**0.9633**	**0.9752**
StableReresolutor	0.8868	0.9730
DEMaxSATsolver	0.8565	0.9300
Loandra-2020	0.7519	0.9332
TT-Open-WBO-Inc-20	0.7927	0.8802

In the folder "imgs" you can see several graphics comparing differents versions proposed.

Contact

Contact me by email: m.framil.deamorin@udc.es

Lista de acrónimos

ABC Artificial Bee Colony algorithm. 19

CMI Clay Mathematics Institute. 2

CNF Conjuctive Normal Form. 18

DPLL Davis-Putman-Logeman-Loveland algorithm. 1

EA Evolutionary Algorithm. 19

ED Evolución Diferencial. 2, 19, 25

EDA Estimation of Distribution Algorithm. 19

EDB Evolución Diferencial Binaria. 22, 28

LSS Local Search Step. 26

MaxSAT Maximum SATisfiability problem. 1, 12

MSE MaxSAT Evaluation. 3, 13, 14, 16

NP Nondeterministic Polynomial time. 2, 7, 20

PMS Partial MaxSAT. 13

PTAS Polynomial-Time Approximation Scheme. 13

QIEA Quantum Inspired Evolutionary Algorithm. 19

RW RandomWalk. 16, 31

SAT Boolean SATisfiability problem. 1, 11

TSP Travelling Salesman Problem. 1

WCNF Weighted Conjuctive Normal Form. 27

Bibliografía

[1] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, no. 7, p. 394–397, Jul. 1962. [En línea]. Disponible en: https://doi.org/10.1145/368273.368557

[2] M. Davis and H. Putnam, "A computing procedure for quantification theory," *J. ACM*, vol. 7, no. 3, p. 201–215, Jul. 1960. [En línea]. Disponible en: https://doi.org/10.1145/321033.321034

[3] M. Buro and H. K. Büning, "Report on a sat competition," *Bulletin of the European Association for Theoretical Computer Science*, Nov. 1992.

[4] M. Heule, M. Järvisalo, M. Suda, M. Iser, T. Balyo, and N. Froleyks. The International SAT Competition. [En línea]. Disponible en: http://www.satcompetition.org/

[5] F. Bacchus, M. Järvisalo, J. Berg, and R. Martins, "MaxSAT Evaluation," 2020. [En línea]. Disponible en: https://maxsat-evaluations.github.io/2020/

[6] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," ser. TACAS '99. Berlin, Heidelberg: Springer-Verlag, 1999, p. 193–207.

[7] H. Kautz and B. Selman, "Planning as satisfiability." 01 1992, pp. 359–363.

[8] C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner, "OPIUM: Optimal Package Install/Uninstall Manager," in *29th International Conference on Software Engineering (ICSE'07)*, 2007, pp. 178–188.

[9] J. Marques-Silva, "Practical applications of Boolean Satisfiability," in *2008 9th International Workshop on Discrete Event Systems*, 2008, pp. 74–80.

[10] J. Park, "Using weighted MAX-SAT engines to solve MPE," 01 2002, pp. 682–687.

[11] Y. Chen, S. Safarpour, J. Marques-Silva, and A. Veneris, "Automated Design Debugging With Maximum Satisfiability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 11, pp. 1804–1817, 2010.

[12] K. Bunte, M. Järvisalo, J. Berg, P. Myllymäki, J. Peltonen, and S. Kaski, "Optimal neighborhood preserving visualization by Maximum satisfiability," *Proceedings of the National Conference on Artificial Intelligence*, vol. 3, pp. 1694–1700, 01 2014.

[13] P. Raiola, T. Paxian, and B. Becker, "Partial (Un-)Weighted MaxSAT Benchmarks: Minimizing Witnesses for Security Weaknesses in Reconfigurable Scan Networks," pp. 44–45, 2020. [En línea]. Disponible en: https://helda.helsinki.fi/bitstream/handle/10138/318451/mse20proc.pdf?sequence=1&isAllowed=y

[14] C. S. Zhu, G. Weissenbacher, and S. Malik, "Post-silicon fault localisation using maximum satisfiability and backbones," in *2011 Formal Methods in Computer-Aided Design (FMCAD)*, 2011, pp. 63–66.

[15] J. Berg and M. Järvisalo, "Optimal correlation clustering via maxsat," in *2013 IEEE 13th International Conference on Data Mining Workshops*, 2013, pp. 750–757.

[16] A. Church, "An unsolvable problem of elementary number theory," *Journal of Symbolic Logic*, vol. 1, no. 2, pp. 73–74, 1936.

[17] A. M. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. s2-42, no. 1, pp. 230–265, 01 1937. [En línea]. Disponible en: https://doi.org/10.1112/plms/s2-42.1.230

[18] G. Ausiello, P. Crescenzi, and M. Protasi, "Approximate solution of np optimization problems," *Theoretical Computer Science*, vol. 150, no. 1, pp. 1–55, 1995. [En línea]. Disponible en: https://www.sciencedirect.com/science/article/pii/030439759400291P

[19] G. Tseitin, "On the complexity of derivation in the propositional calculus," *Zapiski nauchnykh seminarov LOMI*, vol. 8, pp. 234–259, 1968.

[20] S. A. Cook, "The Complexity of Theorem-Proving Procedures," ser. STOC '71. New York, NY, USA: Association for Computing Machinery, 1971, p. 151–158. [En línea]. Disponible en: https://doi.org/10.1145/800157.805047

[21] L. Levin, "Universal sequential search problems," *Problems of Information Transmission*, 1973. [En línea]. Disponible en: http://www.mathnet.ru/links/a3c9ad7e65b5d7e7dcb2e26ba36d1f39/ppi914.pdf

[22] Z. Lei and S. Cai, "Solving (Weighted) Partial MaxSAT by Dynamic Local Search for SAT," 07 2018, pp. 1346–1352.

[23] J. Reisch and P. Großmann, "Stable Resolving - A Randomized Local Search Heuristic for MaxSAT," 2020, unpublished paper under review.

[24] A. P. Guerreiro, M. Terra-Neves, I. Lynce, J. R. Figueira, and V. Manquinho, "Constraint-Based Techniques in Stochastic Local Search MaxSAT Solving," in *Principles and Practice of Constraint Programming*, T. Schiex and S. de Givry, Eds. Cham: Springer International Publishing, 2019, pp. 232–250.

[25] B. Selman, H. Levesque, and D. Mitchell, "A New Method for Solving Hard Satisfiability Problems," in *Proceedings of the Tenth National Conference on Artificial Intelligence*, ser. AAAI'92. AAAI Press, 1992, p. 440–446.

[26] B. Selman and H. A. Kautz, "Domain-Independent Extensions to GSAT: Solving Large Structured Satisfiability Problems," in *IJCAI*, 1993.

[27] H. H. Hoss, "Stochastic Local Search — Methods, Models, Application," Master's thesis, TU Darmstadt, FB Informatik, 1998.

[28] W. Huang, D. Zhang, and W. Houxiang, "An Algorithm Based on Tabu Search for Satisfiability Problem." *J. Comput. Sci. Technol.*, vol. 17, pp. 340–346, 05 2002.

[29] I. P. Gent and T. Walsh, "Towards an Understanding of Hill-climbing Procedures for SAT." MIT Press, 1993, pp. 28–33.

[30] F. Avellaneda, "A short description of the solver EvalMaxSAT," 2020. [En línea]. Disponible en: http://florent.avellaneda.free.fr/dl/EvalMaxSAT.pdf

[31] F. Bacchus, "MaxHS in the 2020 MaxSat Evaluation," pp. 19–20, 2020. [En línea]. Disponible en: https://helda.helsinki.fi/bitstream/handle/10138/318451/mse20proc.pdf?sequence=1&isAllowed=y

[32] R. Martins, V. Manquinho, and I. Lynce, "Open-WBO: A Modular MaxSAT Solver," in *Theory and Applications of Satisfiability Testing – SAT 2014*, C. Sinz and U. Egly, Eds. Cham: Springer International Publishing, 2014, pp. 438–445.

[33] M. Piotrów, "UWrMaxSat: an Efficient Solver in MaxSAT Evaluation 2020," pp. 34–35, 2020. [En línea]. Disponible en: https://helda.helsinki.fi/bitstream/handle/10138/318451/mse20proc.pdf?sequence=1&isAllowed=y

[34] C. Ansótegui, M. L. Bonet, and J. Levy, "SAT-based MaxSAT algorithms," *Artificial Intelligence*, vol. 196, pp. 77–105, 2013. [En línea]. Disponible en: https://www.sciencedirect.com/science/article/pii/S000437021300012X

[35] M. Koshimura, T. Zhang, H. Fujita, and R. Hasegawa, "QMaxSAT: A Partial Max-SAT Solver system description," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 8, 01 2012.

[36] Z. Fu, "Extending the power of Boolean satisfiability solvers: Techniques and applications," 2007.

[37] J.-K. Hao, F. Lardeux, and F. Saubion, "Evolutionary Computing for the Satisfiability Problem," vol. 2611, 04 2003, pp. 258–267.

[38] F. Lardeux, F. Saubion, and J.-K. Hao, "GASAT: A Genetic Local Search Algorithm for the Satisfiability Problem," *Evolutionary computation*, vol. 14, pp. 223–53, 02 2006.

[39] B. Shabash and K. C. Wiese, "PEvoSAT: A Novel Permutation Based Genetic Algorithm for Solving the Boolean Satisfiability Problem," in *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 861–868. [En línea]. Disponible en: https://doi.org/10.1145/2463372.2463479

[40] H. M. Ali, D. Mitchell, and D. C. Lee, "MAX-SAT problem using evolutionary algorithms," in *2014 IEEE Symposium on Swarm Intelligence*, 2014, pp. 1–8.

[41] R. Storn and K. Price, "Differential Evolution: A Simple and Efficient Adaptive Scheme for Global Optimization Over Continuous Spaces," *Journal of Global Optimization*, vol. 23, 01 1995.

[42] B. Doerr and W. Zheng, "Working principles of binary differential evolution," *Theoretical Computer Science*, vol. 801, pp. 110–142, 2020. [En línea]. Disponible en: https://www.sciencedirect.com/science/article/pii/S0304397519305213

[43] M. Framil de Amorín. (2021) Differential Evolution MaxSat Solver. [En línea]. Disponible en: https://github.com/Manuframil/DEMaxSatSolver

[44] S. Y. Cheung. Array of bits. [En línea]. Disponible en: http://www.mathcs.emory.edu/~cheung/Courses/255/Syllabus/1-C-intro/bit-array.html

[45] F. Bacchus. MaxSat Lib. [En línea]. Disponible en: http://www.cs.toronto.edu/maxsat-lib/

[46] J. Berg, E. Demirovic, and P. Stuckey, "Loandra in the 2020 MaxSAT Evaluation," pp. 10–11, 2020. [En línea]. Disponible en: https://helda.helsinki.fi/bitstream/handle/10138/318451/mse20proc.pdf?sequence=1&isAllowed=y

[47] A. Nadel, "TT-Open-WBO-Inc-20: an Anytime MaxSAT Solver Entering MSE'20," pp. 32–33, 2020. [En línea]. Disponible en: https://helda.helsinki.fi/bitstream/handle/10138/318451/mse20proc.pdf?sequence=1&isAllowed=y