UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

PROGRAMA DE MAESTRÍA Y DOCTORADO EN CIENCIAS MATEMÁTICAS Y DE LA ESPECIALIZACIÓN EN ESTADÍSTICA APLICADA

HEURÍSTICAS PARA EL PROBLEMA DE SATISFACIBILIDAD BOOLEANA

TESIS
QUE PARA OPTAR POR EL GRADO DE:
MAESTRO EN CIENCIAS

PRESENTA:
GABRIEL CACHOA OCAMPO

TUTOR PRINCIPAL:
DRA. CLAUDIA O. LÓPEZ SOTO
FACULTAD DE CIENCIAS, UNAM

CIUDAD UNIVERSITARIA, CIUDAD DE MÉXICO 30 DE AGOSTO DE 2022.

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Índice general

1. Introducción **3**

2. Problemas de optimización combinatoria **7**
- 2.1. Problemas de optimización . . . 7
  - 2.1.1. Ejemplos de instancias de problemas de optimización combinatoria . . . 8
  - 2.1.2. Definición formal de problema de optimización combinatoria . 10
  - 2.1.3. Formulación del problema del SAT . . . 11
- 2.2. Formulación estándar del problema de satisfacibilidad booleana . . . 13
- 2.3. Implementación del problema de satisfacibilidad booleana . . . 15

3. Complejidad del problema de satisfacibilidad booleana **16**
- 3.1. Máquinas de Turing . . . 16
  - 3.1.1. Las máquinas de Turing como una descripción formal de los algoritmos . . . 19
- 3.2. La clase P . . . 21
- 3.3. Las clases *NP*, *NP*-completo y NP-duro . . . 22
- 3.4. Teorema de Cook-Levine . . . 25

4. Heurísticas **28**
- 4.1. Búsqueda local . . . 28
- 4.2. Metaheurísticas de trayectoria . . . 29
  - 4.2.1. Escalar las colinas . . . 30
  - 4.2.2. Búsqueda Tabú . . . 31
  - 4.2.3. Recocido Simulado . . . 33
  - 4.2.4. GRASP . . . 36
  - 4.2.5. Búsqueda por Vecindades Variables . . . 38
- 4.3. Metaheurísticas basadas en poblaciones . . . 40
  - 4.3.1. Algoritmos Genéticos . . . 41
  - 4.3.2. Optimización por Colonias de Hormigas . . . 43

5. Heurísticas específicas para el problema del SAT **46**
- 5.1. Algoritmo Davis-Putnam-Loveland-Logeman . . . 47
  - 5.1.1. Aprendizaje de cláusulas . . . 52
- 5.2. Algunas heurísticas modernas basadas en el DPLL . . . 57

6. Análisis computacional de las heurísticas **61**
- 6.1. Análisis usando instancias homogéneas . . . 61
  - 6.1.1. Detalles generales sobre las implementaciones . . . 62

6.1.2. Escalar colinas . . . 62
6.1.3. Búsqueda Tabú . . . 63
6.1.4. Recocido Simulado . . . 64
6.1.5. GRASP . . . 65
6.1.6. Búsqueda por vecindades variables . . . 66
6.1.7. Algoritmos genéticos . . . 67
6.1.8. Optimización por colonias de hormigas . . . 69
6.1.9. GLUCOSE . . . 69
6.2. Comparación de las heurísticas . . . 70
6.3. Conclusiones . . . 70

7. Conclusiones **72**

A. Implementaciones **74**
A.1. Implementaciones elementales . . . 74
A.2. Codificaciones . . . 75
A.3. Heurísticas . . . 79

Capítulo 1

Introducción

Esta tesis aborda una clase de problemas que teóricamente se pueden resolver pero que prácticamente son muy difíciles de resolver. Problemas que trazan los límites entre lo teóricamente posible y lo prácticamente imposible. Esta clase de problemas han sido estudiados ampliamente desde la teoría de la complejidad computacional y han sido clasificados como problemas *NP*-duros. Nosotros centraremos nuestra atención en estudiar sólo uno de esos problemas: el problema de satisfacibilidad booleana que está al centro del estudio de los problemas *NP*-duros. En la actualidad, la investigación de operaciones ha resuelto el problema de encontrar soluciones factibles y *buenas* a problemas *NP*-duros. Los procedimientos para resolver problemas *NP*-duros usando una cantidad factible de recursos son métodos no deterministas, que en ocasiones sacrifican la optimalidad o la factibilidad de las soluciones, que se basan en la exploración estocástica del espacio de soluciones, se les conoce de diversas maneras: “heurísticas” o “meta-heurísticas”.

Un ejemplo muy común donde podemos apreciar la importancia y utilidad de los métodos heurísticos en el contexto de un problema *NP*-duro, el **problema del agente viajero**. Una instancia del problema del agente viajero es la siguiente: Supongamos que una persona tiene que recorrer cuatro ubicaciones que vamos a enumerar como A, B, C y D, de manera que su recorrido inicie y termine en la ubicación A y que recorra todas las ubicaciones. Cada transición de una ubicación a otra, es decir cada viaje, tiene un costo. ¿Cómo escogemos el recorrido que tenga el menor costo? En la Figura 1.1 podemos ver una representación gráfica de esta instancia en la que cada nodo es una de las ubicaciones y el número asignado a cada arista es el costo de hacer ese viaje.

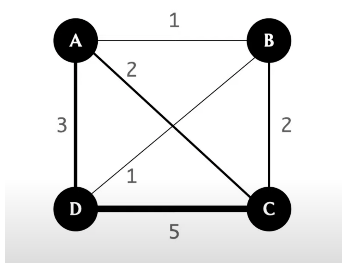

Figura 1.1: Instancia del problema del agente viajero [1].

Una manera de resolver este problema podría ser trazar la ruta a partir de la primera

ubicación y seleccionar la siguiente en función de dos criterios: escoger la arista (o viaje) que tenga menor costo y no seleccionar ubicaciones que ya hayan sido *visitadas*. A esta estrategia se le llama **glotona**.

Si seguimos esta estrategia podemos ver que la solución que obtendríamos sería la siguiente permutación de las letras A, B, C y D:

$$A \to B \to D \to C \to A$$

que tendría un costo de 1+1+5+2 = 9. Este recorrido está representado gráficamente en la Figura 1.2.

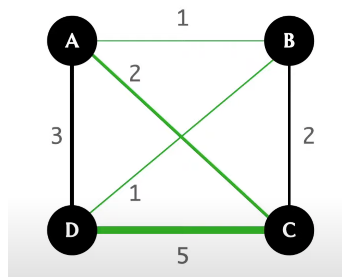

Figura 1.2: Solución glotona a una instancia del problema del agente viajero [1].

No obstante, es claro que ésta no es la mejor solución. Una solución menos costosa sería la siguiente permutación de las letras A, B, C y D:

$$A \to D \to B \to C \to A$$

que tendría un costo de $3 + 1 + 2 + 2 = 8$. Esta solución está representada en la Figura 1.3.

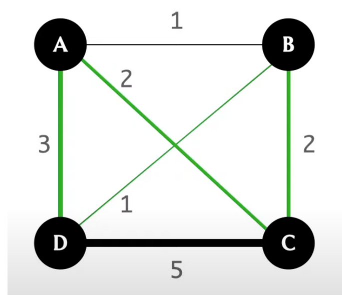

Figura 1.3: Solución óptima a una instancia del problema del agente viajero [1].

Aunque la diferencia entre la solución óptima (8) y la solución que se obtiene usando un algoritmo glotón (9) en este caso no difieren mucho, podrían hacerlo. De aquí podemos concluir que un algoritmo glotón no siempre encuentra el valor óptimo. En este ejemplo, también es muy sencillo encontrar la solución óptima por simple inspección, sin embargo cuando consideramos instancias más grandes esto se vuelve muy difícil. Por ejemplo, consideremos la instancia del problema del agente viajero representada en la Figura 1.4.

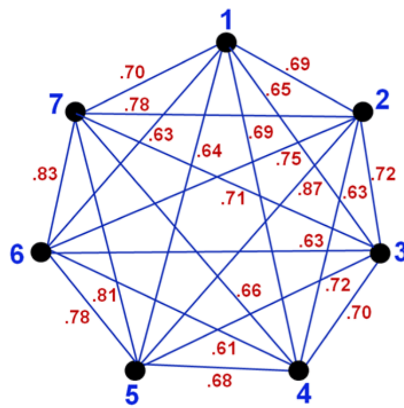

Figura 1.4: Otra instancia del problema del agente viajero [56].

La instancia de la Figura 1.4 luce mucho más compleja que la de la instancia en la Figura 1.1 esto se debe a que el número de posibles conexiones depende del número de ubicaciones según un polinomio de orden cuadrado. Aún así, esta instancia todavía es pequeña y se puede encontrar su solución óptima a mano.

Un algoritmo que nos permitiría encontrar de manera determinista la ruta menos costosa, es decir la ruta óptima, sería listar todas las posibles rutas y de cada una de ellas calcular su costo, para después escoger la de menor costo.

El número de posibles rutas a seguir en una instancia del problema del agente viajero depende factorialmente del número de ubicaciones. Por ejemplo, consideremos los 32 estados de México como las ubicaciones y el costo entre cada estado es el costo dado por alguna línea de camiones que conecta a todos los estados de la República. Supongamos que salimos y terminamos en la CDMX. En este caso, tendríamos 32! posibles rutas a considerar. Es decir:

$$32! = 263,130,836,933,693,530,167,218,012,160,000,000.$$

Si a una computadora le tomara 0.0004 segundos calcular el costo de una de esas rutas, entonces le tomaría aproximadamente

$$483,412,030,000,000,000,000,000 \text{ años}$$

evaluar todas las posibles rutas.

De la inmensidad del tamaño del espacio de búsqueda surge la imposibilidad de resolverlo por medio de *fuerza bruta*, es decir considerar todas las opciones y tomar la mejor. Y estrategias como los algoritmos glotones no ofrecen la solución óptima ni siquiera en instancias muy pequeñas.

Frente a este problema, usamos métodos heurísticos. Éste es sólo un ejemplo de los problemas de optimización combinatoria. Gracias a la diversidad de problemas combinatorios que existen y que día a día surgen a partir de problemas de la vida real, los métodos heurísticos tienen una amplia aplicación en contextos de la vida real. En este trabajo investigaremos los métodos heurísticos para resolver el problema de satisfacibilidad booleana.

Objetivos de la tesis

El objetivo de este trabajo es estudiar *métodos* no *determinista* que sean *factibles* y que nos ayuden a encontrar soluciones *buenas* para problemas combinatorios *difíciles*, en particular para problema de satisfacibilidad booleana. Desarrollamos la teoría necesaria para formular con precisión este objetivo, es decir la teoría de complejidad computacional. Investigamos las heurísticas clásicas para resolver problemas de optimización combinatoria en general e investigamos las heurísticas particulares diseñadas para el problema del satisfacción booleana de decisión y de optimización. Implementamos estos métodos y estudiamos el estado del arte a un nivel hemerográfico de los solucionadores del problemas del SAT actuales. Por último, analizamos la información que obtuvimos de la experimentación, concluimos y establecemos las posibles aplicaciones y el trabajo relacionado.

Estructura de la tesis

- **Problemas de optimización combinatoria:** En este capítulo desarrollamos intuitiva y formalmente las características de un problema de optimización combinatoria. Así como la necesidad de reolverlos y la variedad de contextos en los que aparecen. Se visita la lista de Karp de problemas NP-completos.

- **La complejidad del problema de satisfacibilidad booleana:** En este capítulo se demuestra el teorema de Cook-Levine, es decir que el problema del SAT es NP-completo. Para hacer esta demostración se desarrolla formal e intuitivamente la teoría de computación necesaria.

- **Heurísticas:** En este capítulo se hace un recuento bibliográfico de las heurísticas clásicas: Escalar las colinas, Búsqueda Tabú, Recocido Simulado, GRASP y Algoritmos Genéticos. Y de dos heurísticas más recientes: Búsqueda por Vecindades Variables y Optimización por Colonias de Hormigas. Estas heurísticas fueron implementadas y estudiadas. En este capítulo se presentan algunos resultados de calibrar los parámetros de dichas heurísticas.

- **Heurísticas específicas para el problema del SAT:** En este capítulo se hace una revisión bibliográfica de publicaciones recientes que revolucionaron los paradigmas de las heurísticas para resolver el problema del SAT.

- **Análisis de las heurísticas:** En este capítulo estudiamos el desempeño y limitaciones de las heurísticas que implementamos sobre un conjunto de instancias homogéneas.

Capítulo 2

Problemas de optimización combinatoria

> Mathematics can instruct us on how to optimize. But what to optimize—that remains a question for humans.
>
> *Ben Orlin*

En este capítulo vamos a describir informalmente qué es un problema de optimización combinatoria y mostraremos ejemplos de instancias de problemas de optimización combinatoria. Después definiremos formalmente qué es un problema de optimización combinatoria. Por último formulamos el problema de satisfacibilidad booleana, primero de una manera intuitiva y luego de manera estándar.

2.1. Problemas de optimización

Uno de los conceptos fundamentales de las matemáticas es el de optimización, es decir escoger algunas variables para obtener el *mejor* resultado de algún proceso. Desde que estamos en el bachillerato nos hablan de optimización, por ejemplo de encontrar la *mejor* manera de recortar un cartón para hacer una caja sin tapa. Este clásico problema que podemos ver en la Figura 2.1 se resuelve modelando el problema a través de una función diferenciable y usando herramientas de cálculo para encontrar las posibles soluciones. Los problemas de optimización que nos encontramos en la vida real no siempre se pueden modelar a través de funciones diferenciables.

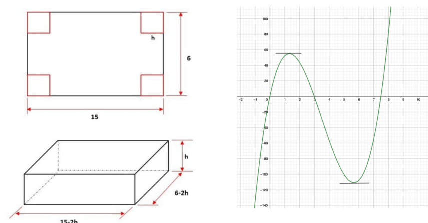

Figura 2.1: Instancia de un problema de optimización [53].

2.1.1. Ejemplos de instancias de problemas de optimización combinatoria

A continuación vamos a describir algunos problemas de optimización combinatoria y mostraremos una aplicación concreta de estos problemas, a esta aplicaciones les llamamos *instancias*.

Problema de la mochila

Un ejemplo muy claro en el que una función diferenciable no modela el problema es el **problema de la mochila**. En este problema, tenemos un conjunto de objetos en el que cada uno tiene un costo y un beneficio y también tenemos una cota del máximo costo a invertir. Esto se representa como una mochila con una máxima capacidad de carga y un conjunto de objetos cada uno con un peso y una ganancia económica. El problema de la mochila es: ¿cuál es el conjunto de objetos que representan la mejor ganancia y tiene un costo admisible de acuerdo con la cota? En la Figura 2.2 podemos ver una ilustración de una instancia del problema.

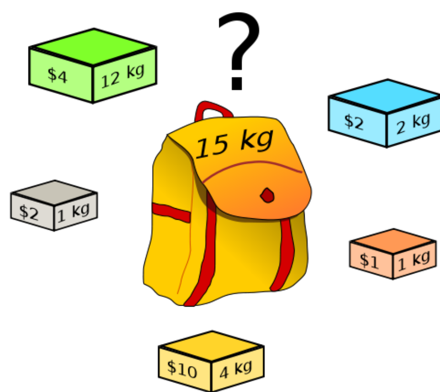

Figura 2.2: Ejemplo de instancia del problema de la mochila.[11]

En este caso los objetos son cinco cajas, cada una tiene una ganancia y un costo. La ganancia es el valor económico en pesos que tiene cada caja escrita y el costo es el peso en kilogramos que también tiene escrita cada caja. Queremos llevarnos el conjunto de objetos con mayor valor económico pero tenemos una restricción: en nuestra mochila sólo podemos cargar 15 kilogramos. ¿Cuál es el *mejor* conjunto de objetos que nos podemos llevar?

Es claro que, con una simple inspección podemos encontrar el mejor conjunto de objetos. No obstante, la complejidad de este problema se puede apreciar en instancias más complejas. Por ejemplo, supongamos que tenemos tres objetos y una mochila con capacidad de 4 unidades. Los objetos están descritos en la siguiente tabla.

Objeto	Objeto 1	Objeto 2	Objeto 3
Peso	3	2	2
Ganancia	1.8	1	0.9

Esta instancia también se puede resolver por simple inspección. Sin embargo, podemos ver que no se puede resolver por un algoritmo glóton. Siguiendo un algoritmo

glotón que escoja al objeto según la máxima ganancia o incluso según la máxima razón entre el peso y la ganancia, escogemos el Objeto 1. Sin embargo, la solución óptima es el conjunto que tiene a los objetos 1 y 2.

¿Cuál es el algoritmo para resolver las instancias del problema de la mochila? Otra opción sería identificar a cada conjunto de objetos con una secuencia de ceros y unos. Así, el conjunto de objetos que sólo tiene al Objeto 1 sería la secuencia: 100. Y la secuencia que tiene a los objetos 1 y 2 sería: 011. El espacio de búsqueda sería el conjunto de sucesiones de longitud 3 de ceros y unos. Este conjunto tiene $2^3 = 8$ elementos. Entre más elementos, el espacio de búsqueda es más grande. Un algoritmo que podría resolver eficazmente este tipo de problemas sería explorar cada una de las opciones, otra vez este algoritmo se le conoce como método de *fuerza bruta*.

El problema del clan

Otro ejemplo de problema de optimización combinatoria surge en un contexto de gráficas. Una gráfica es una tupla (N, E) donde N es un conjunto de nodos y $E \subseteq N \times N$ es un conjunto de aristas. Para una gráfica, decimos que $C \subseteq N$ es un **clan** si es un conjunto de nodos en el que todos son *adyacentes*, es decir que hay una arista que los une. Un k-clan es un clan de cardinalidad k.
En la Figura 2.3 podemos ver que en (A) el subconjunto de vértices en rojo no es un clan, en (B) el subconjunto de vértices rojo sí es un clan pero no es un clan maximal (es decir hay un clan más grande que contiene a esos nodos), y en (C) y (D) sí son clans maximales.

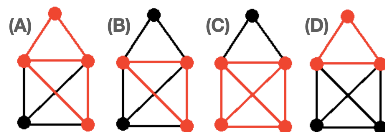

Figura 2.3: Ejemplo de gráfica y de posibles clans [39].

En la instancia anterior es bastante sencillo encontrar un clan. Sin embargo la complejidad de este problema aumenta con respecto al número de vértices. Por ejemplo, no es inmediato cómo encontrar un 10-clan en la gráfica de la Figura 2.4.

Figura 2.4: Ejemplo de gráfica que tiene un 10-clan [35]

¿Cómo podemos encontrar un algoritmo que dada una gráfica decida si tiene o no un k-clan? Podemos, de la misma manera que hicimos con la mochila a cada subconjunto de vértices identificarlo como una secuencia de ceros y unos. Dada una secuencia, evaluar que todos los vértices 1 de la secuencia sean adyacentes. De esta manera también podemos resolver este problema por medio de *fuerza bruta*, en el que el espacio de búsqueda también depende exponencialmente del número de vértices.

Programación 0-1

El último ejemplo de problema de optimización combinatoria que vamos a mostrar es el de programación 0-1. Dada una matriz A de $n \times n$ cuyas entradas sean: 0, 1 ó -1 y un vector B de n números enteros, existe un vector x de n ceros o unos tal que:

$$Ax \leq B.$$

Donde $\leq$ entre vectores significa que cada entrada de uno es menor o igual que la entrada del otro. Un ejemplo de instancia de este problema podría ser la siguiente.

$$\begin{bmatrix} 1 & 2 & 3 \\ -1 & -5 & -8 \\ 3 & -2 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \leq \begin{bmatrix} -3 \\ 2 \\ 0 \end{bmatrix}$$

¿Cómo podemos encontrar un algoritmo que determine si ese vector exista? Cada vector es una secuencia de ceros y unos. Podemos evaluar cada posible secuencia si es que satisface o no la desigualdad. De la misma manera que con los otros ejemplos, por un método de *fuerza bruta* podemos resolver este problema.

2.1.2. Definición formal de problema de optimización combinatoria

Ahora vamos a desarrollar formalmente la teoría de los problemas de **optimización combinatoria**. En general, los problemas de optimización combinatoria buscan la mejor configuración de variables para obtener algún objetivo. Estos problemas se pueden clasificar de acuerdo a si las variables son continuas o discretas. De acuerdo con Papadimitriou y Stegilgitz [46] podemos definir un problema de optimización combinatoria como sigue.

Definición 2.1.1. *Un problema de optimización combinatoria es una tupla* (I, f, m, g) *donde:*

- I *es un conjunto de instancias para el problema,*
- *dada una instancia* $x \in I$*,* $f(x)$ *es el conjunto de posibles soluciones para la instancia* x *(también lo llamamos el espacio de búsqueda de la instancia),*
- *dada una instancia* x *y una posible solución* $y \in f(x)$*,* $m(x, y)$ *es la medida de la solución (que a veces podemos representar como el costo de la solución o como la eficiencia de la solución), y*
- $g \in \{max, min\}$.

Así un problema de optimización combinatoria se traduce en que dada una instancia $x \in I$ encontrar la solución $y \in f(x)$ tal que:

$$m(x, y) = g\{m(x, y') \mid y' \in f(x)\}.$$

Resolver un problema de optimización combinatoria significa encontrar $s^* \in E$ tal que s^* sea un mínimo (máximo) global de la función f. Es decir,

$$\forall s \in S \quad f(s) > f(s^*) \quad (f(s) < f(s^*)) .$$

A la solución s^* le llamamos óptimo global y denotamos por $S^* \subseteq S$ al conjunto de soluciones que sean óptimos globales.

Dentro de los problemas de optimización combinatoria existe una clase que llamamos problemas *NP*-duros. En el siguiente capítulo de este trabajo desarrollamos la teoría formal de estos problemas, pero por ahora podemos decir que son la clase de problemas para los que "quizás no existe un algoritmo determinista y eficiente que los resuelva". Problemas como el del agente viajero y el de la mochila son *NP*-completos. La lista de problemas de optimización combinatoria que resultan ser *NP*-completos es inmensa y muchos problemas de optimización combinatoria que son necesarios en la práctica son *NP*-completos.

2.1.3. Formulación del problema del SAT

En 1971 los matemáticos Stephen Cook y Leonid Levin sentaron las bases de la teoría de complejidad computacional al demostrar que el *problema de satisfacibilidad booleana* es un problema *NP*-completo [12]. En 1972 el matemático Richard Karp publicó una lista [33] de problemas que también son *NP*-completos. El famoso libro de Michael Garey y David S. Johnson de 1979 [24] tiene muchos más ejemplos de problemas *NP*-completos, pero la lista está en continua construcción y en páginas como [60] se pueden encontrar más actualizaciones de la lista de problemas *NP*-completos, aunque ésta lista tampoco está completa. En la práctica, los problemas de optimización que son *tes* son problemas *NP*-duros que no están en *P*.

Un mismo problema de optimización combinatoria puede tener tres formulaciones equivalentes: **decisión**, **optimización** o **evaluación**. El problema de la satisfacibilidad booleana está enunciado originalmente como un problema de decisión. Para escribirlo como problema de optimización necesitamos saber que las fórmulas en lógica proposicional son sucesiones de símbolos que abstraen *proposiciones*, es decir, enunciados que pueden ser escritos en lenguaje natural y que pueden ser catalogados como verdaderos o falsos, y se combinan por medio de conectivos lógicos: "no ...", "... y ...", "... o ...", "si ... entonces", y "... si y sólo si... ". Mientras que es necesario establecer los detalles para definir las fórmulas en lógica proposicional, por ahora es intuitivamente claro que un ejemplo de fórmula es:

$$(P \to Q) \to P \wedge (\neg R \vee Q),$$

que en español se lee "P implica Q si P y no R o Q " donde P, Q y R son proposiciones. En este contexto, a cada proposición le podemos asignar un 0 si es falsa o un 1 si

es verdadera. También en este contexto interpretamos a cada uno de los conectivos lógicos como operaciones entre ceros y unos. Con esto, dados valores de verdad para las proposiciones podemos calcular el valor de verdad de la fórmula. Por ejemplo, la siguiente *tabla de verdad* nos ilustra que la asignación es verdadera cuando $P = 1$, $Q = 1$ y $R = 0$ pero es falsa cuando $P = 0$, $Q = 0$ y $R = 1$.

P	Q	R	$(P \to Q)$	$\neg R$	$(\neg R \vee Q)$	$P \wedge (\neg R \vee Q)$	$(P \to Q) \to P \wedge (\neg R \vee Q)$
1	1	0	1	1	1	1	1
0	0	1	1	0	0	0	0

Tabla 2.1: Ejemplo de tabla de verdad para una fórmula proposicional en general.

Podemos enunciar este problema de decisión como un problema de optimización ya que *dada una fórmula en lógica proposicional existe otra fórmula equivalente en forma normal conjuntiva.* Una fórmula está en forma normal conjuntiva si es la conjunción de fórmulas que son disyunciones de proposiciones o negaciones de proposiciones. A continuación mostramos el procedimiento para escribir esta fórmula en forma normal conjuntiva.

$$(P \to Q) \to P \wedge (\neg R \vee Q)$$ La fórmula original

$$\neg(P \to Q) \vee (P \wedge (\neg R \vee Q))$$ Usando la definición de $\to$.

$$\neg(\neg P \vee Q) \vee (P \wedge (\neg R \vee Q))$$ Usando la definición de $\to$.

$$(P \wedge \neg Q) \vee (P \wedge (\neg R \vee Q))$$ Usando leyes de DeMorgan

$$(P \vee (P \wedge (\neg R \vee Q))) \wedge (\neg Q \vee (P \wedge (\neg R \vee Q)))$$ Distributividad de $\vee$ en $\wedge$.

$$((P \vee P) \wedge$$
$$(P \vee (\neg R \vee Q))) \wedge$$
$$(\neg Q \vee (P \wedge (\neg R \vee Q)))$$ Separamos en renglones las cláusulas, quitamos paréntesis y distribuimos $\wedge$. sobre $\vee$.

$$(P \wedge$$
$$P \vee \neg R \vee Q \wedge$$
$$(\neg Q \vee (P \wedge (\neg R \vee Q)))$$ Usamos la idempotencia y, quitamos paréntesis.

$$(P) \wedge$$
$$(P \vee \neg R \vee Q) \wedge$$
$$(\neg Q \vee P) \wedge$$
$$(\neg Q \vee \neg R \vee Q)$$ Distribuimos $\vee$ sobre $\wedge$.

La fórmula anterior es equivalente a la fórmula del ejemplo en el sentido en que ésta es satisfacible si y sólo si la otra lo es. Se puede demostrar que cada fórmula en un lenguaje

de primer orden se puede escribir en su forma normal conjuntiva.

A cada una de las fórmulas en las conjunciones le llamamos **cláusula**. Es claro que esta fórmula es satisfacible si y sólo si existe una asignación que satisfaga a cada una de las cláusulas. Ahora, a cada asignación de valores de verdad le podemos asignar el número de cláusulas que satisface. Por lo tanto podemos escribir el problema de satisfacibilidad booleana en su versión de optimización como sigue:

> *Dada una fórmula en lógica proposicional en su forma normal conjuntiva encontrar la asignación que maximice el número de cláusulas verdaderas.*

Si una asignación satisface a una fórmula entonces cada cláusula es satisfacible, es decir que el máximo número de cláusulas son verdaderas. Por lo tanto, si se resuelve el problema de decisión entonces se resuelve el problema de optimización. Por otro lado, si se satisface el máximo número de cláusulas entonces todas las cláusulas son verdaderas, es decir que la fórmula es satisfacible. Por lo tanto, resolver el problema de optimización resuelve el problema de decisión.

Como para cualquier fórmula en lógica proposicional siempre se puede encontrar una fórmula equivalente en forma normal conjuntiva entonces son equivalentes la versión de optimización y la versión de decisión del problema de satisfacibilidad booleana.

Mientras otros problemas *NP*-duros tienen una aplicación más inmediata a problemas de la vida real, el problema de satisfacibilidad booleana tiene una importancia teórica, histórica y práctica frente a los demás problemas. Por un lado, el problema de satisfacibilidad booleana fue el primer problema que se demostró que es *NP*-duro, a este resultado se le conoce como *Teorema de Cook-Levine*. La importancia teórica radica en la relación que guarda este problema con los demás ya que para demostrar que algún problema es *NP*-duro se demuestra que éste problema se puede reducir al problema de satisfacibilidad booleana. En el siguiente capítulo definiremos formalmente qué quiere decir que un problema se reduzca a otro. El problema de satisfacibilidad booleana es un intermediario entre varios problemas combinatorios y la definición formal de *NP*-duro. La importancia práctica del problema de satisfacibilidad booleana radica en la gran variedad de estrategias que existen para resolverlo aproximadamente y en la diversidad de aplicaciones que se la ha dado al problema en el contexto de problemas combinatorios.

2.2. Formulación estándar del problema de satisfacibilidad booleana

Desde 1995, se han establecido formatos gráficos para una variedad de problemas de optimización combinatoria. Actualmente, estas estandarizaciones se usan ampliamente y nos permite estandarizar también a las heurísticas para recibir cualquier tipo de instancia en dicho formato gráfico.

La estandarización de las instancias del problema de satisfacibilidad booleana se las debemos al DIMACS (*Center for Discrete Mathematics and Theoretical Computer Science*) [18] de la Universidad de Rutgers.

El formato gráfico del problema de satisfacibilidad booleana se llama **formato CNF** está inspirado en las fórmulas en forma normal conjuntiva. Es decir, fórmulas que sólo utilizan conjunciones, negaciones y disyunciones.

Definición 2.2.1. *Una instancia del problema de satisfacibilidad booleana está en formato CNF si es un archivo de texto descrito de la siguiente manera.*

1. *Comentarios: Una línea de comentarios empieza con la letra c, son ignorados por el programa y son humano-legibles.*

   ```
   c Este es un ejemplo de línea de comentario.
   ```

2. *Línea de problema: La línea de problema empieza con la letra p, sólo hay una línea de problema por código, la linea de problema tiene el siguiente formato*

   ```
   p cnf VARIABLES CLÁUSULAS
   ```

   donde VARIABLES debe ser el número de variables en la instancia y CLÁUSULAS debe ser el número de cláusulas en la instancia.

3. *Cláusulas: Por cada cláusula hay una línea en el código, cada línea consiste de los números de las variables que aparecen en la cláusula separados por un espacio y si la variable tiene una negación se escribe un signo - antes del número.*

Por ejemplo, la fórmula $(x_1 \vee \overline{x_2} \vee x_4) \wedge (x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_3})$ se escribe en formato CNF como:

```
c Este es un ejemplo de instancia.
p cnf 4 3
1 -2 4
2 3
-1 -3
```

Nosotros en la línea del problema siempre escribiremos cnf, el formato de DIMACS acepta otros formatos para las instancias pero nosotros no los utilizaremos. Otros formatos admisibles para las instancias según DIMACS son: sat (las fórmulas se escriben explícitamente usando *, +, y - como los conectivos lógicos), satx (que utiliza el conectivo lógico xor), sate (que utiliza el conectivo lógico =) y satex (que utiliza ambos conectivos lógicos xor e =).

Ahora que hemos establecido la formulación de las instancias podemos establecer cuál es el espacio de búsqueda del problema de satisfacibilidad booleana.

Definición 2.2.2. *Para una instancia de n variables del problema de satisfacibilidad booleana, el espacio de búsqueda es el conjunto de vectores de longitud n de ceros y unos. Es decir, $\{0,1\}^n$.*

Una solución para una instancia del problema de satisfacibilidad booleana es un vector de n entradas, la n-ésima entrada representa el valor de verdad que se le asigna a esa variable.

La función objetivo cuenta cuántas cláusulas están satisfechas por la asignación. Cada cláusula es una conjunción de variables o de negación de variables. Es suficiente para que una cláusula esté satisfecha que una de sus variables, tal como aparece en la cláusula, aparezca en la asignación. Evaluar la asignación es un proceso que se divide en tres partes: comparar las entradas de la asignación y de la cláusula, registrar si la cláusula es verdadera y contar el número de cláusulas verdaderas.

En el siguiente pseudocódigo describimos la función objetivo a implementar en el problema de satisfacibilidad booleana. Las entradas de la función son la línea del problema, la instancia como matriz y la asignación de valores para las variables. Es decir que cada fila sea una cláusula y que en cada fila de la columna aparezca el número entero que representa a la variable. La asignación puede ser un vector de ceros (o menos unos) y unos donde la i-ésima entrada representa que i-ésima variable sea verdadera o falsa; o la asignación podría ser un conjuto con los primeros n números naturales con signo negativo o no, si el número i pertenece a este conjunto entonces la i-ésima variable es verdadera pero si el número $-i$ pertenece al conjunto entonces la variable es falsa. Esta función esencialmente representa un proceso de búsqueda de las variables de cada cláusula en la asignación. Si la variable que aparece en la cláusula aparece representada en la asignación entonces la cláusula es verdadera.

Algoritmo 1 Función objetivo para el problema de satisfacibilidad booleana

Require: $p = [n, m] \leftarrow$ [Variables, Cláusulas]
Require: $P \leftarrow$ Instancia como matriz
Require: $s \leftarrow$ Asignación
Require: $L \leftarrow zeros(n)$ ▷ Vector de ceros con longitud igual al número de variables.
  for $i \leq n$, $j \leq m$ **do**
    if signo(P[i,j]=s(i) **then**
      $L[j] \leftarrow 1$
    end if
  end for
  return Sumar todos los elementos de L

Claramente, para el problema de decisión, la única modificación que se hace al algoritmo es comparar si la suma de elementos de L es igual a n. Usar esta función nos permite calificar cuándo una asignación es mejor que otra en el sentido en el que satisface más cláusulas.

Si llamamos n al número de variables en la instancia, m al número de cláusulas y l a la máxima longitud de una cláusula. Entonces en el peor escenario, la función objetivo hace $m \times l \times n$ comparaciones, m registros y una suma de m términos. En este sentido, la función tiene un grado de complejidad cúbico.

2.3. Implementación del problema de satisfacibilidad booleana

A lo largo del trabajo vamos a usar Python 3.6 en un ambiente físico de Google Colab. Utilizando la Función A.1 convertimos el formato gráfico de una instancia del problema de satisfacibilidad booleana en un arreglo numérico de NumPy. En el que cada fila se corresponde con una cláusula y cada entrada de la fila con una variable. Además, usando esta función, la línea de problema se convierte en un vector con dos entradas donde la primera entrada es el número de variables y la segunda entrada es el número de cláusulas.

De acuerdo con el pseudocódigo 1, implementamos la Función A.2 que aparece en el apéndice A que toma las cláusulas, P, en el formato cargado por la Función A.1 y un vector, s, de NumPy de ceros y unos con longitud el número de variables que representa la asignación de valores de verdad para las variables.

Capítulo 3

Complejidad del problema de satisfacibilidad booleana

> The idea behind digital computers may be explained by saying that these machines are intended to carry out any operations which could be done by a human computer. The human computer is supposed to be following fixed rules; has no authority to deviate from them in any detail. We may suppose that these rules are supplied in a book, which is altered whenever is put on to a new job. Also, has an unlimited supply of paper on which does calculations.
>
> *Alan Turing, 1950*

El objetivo de este capítulo es desarrollar la teoría necesaria para definir el grado de complejidad de un problema de optimización combinatoria. Es importante aclarar, de inicio, que no se mide la complejidad de un problema *per se*. Más bien, se mide la complejidad de los algoritmos que lo resuelven. La teoría formal de los algoritmos surge en la teoría de máquinas de Turing. En este capítulo vamos a definir formalmente lo que es una máquina de Turing y desarrollaremos un ejemplo explícito. Después estudiaremos el orden de complejidad de las máquinas de Turing deterministas para definir la clase de complejidad *P*. De manera análoga, describiremos el orden de complejidad de máquinas de Turing no deterministas para introducir la clase de complejidad *NP*. Por último, vamos a definir la clase de complejidad *NP*-completo y vamos a demostrar que el problema de satisfacibilidad booleana pertenece a esta clase.

3.1. Máquinas de Turing

Después de leer la cita de Alan Turing sobre cómo concibe a las máquinas, lo primero que me viene a la mente es la película *Hidden Figures* dirigida por Theodore Melfi. Ésta es la historia de las matemáticas afroamericanas Katherine Johnson, Dorothy Vaughan y Mary Jackson cuyo trabajo permitió que John Glenn se convirtiera en el primer astronauta

estadounidense en hacer una órbita completa de la Tierra. En esta película podemos ver a las *calculadoras humanas* de la NASA. Personas encargadas de hacer y revisar cálculos aritméticos, en cierto sentido mecánicos. Las calculadoras humanas no tenían las mismas responsabilidades que los ingenieros espaciales, quienes utilizaban su creatividad y conocimientos matemáticos para resolver los problemas que la administración presentaba. El trabajo de las calculadoras humanas ya no existe, ha sido reemplazado por las modernas computadoras que pueden hacer de una manera eficiente e infalible las operaciones aritméticas.

Figura 3.1: Fotografía de calculadoras humanas en la NASA. [23]

Las máquinas de Turing son los modelos de la computación actual, las calculadoras humanas son personas que hacen una operación, anotan el resultado, vuelven a hacer una operación, y repiten esto de acuerdo a un conjunto de reglas y son un ejemplo muy claro de cómo funciona una máquina de Turing, es decir son un ejemplo muy claro del modelo de computación.

Podemos describir informalmente una máquina de Turing de la siguiente manera: Primero consideremos el conjunto de sucesiones finitas de ceros y unos, $\{0,1\}^*$ y una función $f : \{0,1\}^* \to \{0,1\}$. La máquina de Turing, o el algoritmo, que calcula a f es un conjunto de reglas mecánicas tal que al seguir ese conjunto de reglas podemos calcular $f(x)$ dado $x \in \{0,1\}^*$. Este conjunto de reglas es fijo, es decir que se usa el mismo conjunto de reglas para todas los posibles valores en el dominio de la función, pero cada regla en el conjunto se puede aplicar un número arbitrario de veces. Cada regla involucra una o más de las operaciones elementales. Éstas son:

1. Leer un bit de x, formalmente calcular la proyección de x en el alguna de sus entradas.
2. Leer un bit o un símbolo del bloc de notas, formalmente en proyectar alguna entrada de la tira donde la máquina escribe.

Y basándose en los valores leídos, la máquina también puede:

3. Escribir un símbolo en el bloc de notas.
4. Parar y escribir el resultado, 0 ó 1.

5. Escoger una nueva regla del conjunto para aplicar a continuación.

En el bloc de notas, la máquina de Turing puede anotar símbolos diferentes a 0 ó 1. Estos símbolos son elementos de un *alfabeto*. El tiempo de corrida de una máquina de Turing es una función $T : \mathbb{N} \to \mathbb{N}$ de manera que $T(n)$ cuenta cuántas operaciones se tienen que ejecutar para que la máquina calcule el valor de una entrada de longitud n.

Podemos enunciar de manera informal el siguiente hecho alrededor de este modelo: Este modelo es robusto frente a algunos cambios en la definición. Por ejemplo, cambiar la cardinalidad del alfabeto o aumentar el número de blocs de notas. La versión más sencilla de este modelo puede simular a la versión más complicada con una ralentización polinomial.

Ahora, vamos a definir formalmente a una máquina de Turing de k-cintas. Primero vamos a describir dos de sus elementos: el bloc de notas y el conjunto de reglas. Después daremos la definición precisa y mostraremos un ejemplo. La siguiente Figura es una ilustración de los elementos de una máquina de Turing de 3-cintas.

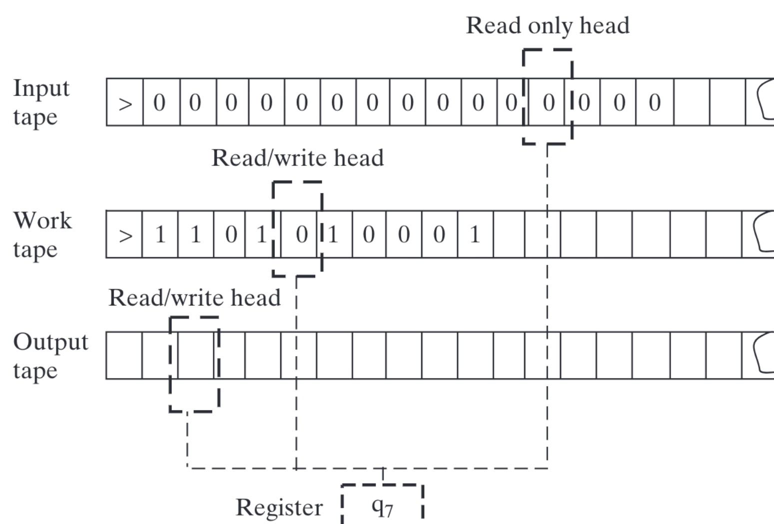

Figura 3.2: Imagen de una máquina de Turing de 3 cintas. [24]

Bloc de notas

El bloc de notas consiste en k-cintas. Una cinta es una línea infinita y unidireccional de celdas que pueden almacenar un símbolo de un conjunto finito Γ, que llamamos alfabeto de la máquina de Turing. Cada cinta está equipada con una *cabeza* que puede leer o escribir símbolos, uno a la vez, en la cinta. Los cálculos que puede hacer la máquina están en tiempo discreto, es decir en pasos, la cabeza de la cinta se puede mover a la izquierda o a la derecha en cada uno de esos pasos.

La primera cinta de la máquina es la cinta de entrada, en esta cinta la cabeza es de *sólo lectura*, es decir que no puede modificar ni escribir sobre la cinta de entrada. En las siguientes $k-1$ cintas la cabeza de la cinta es de lectura y escritura. A estas cintas les vamos a llamar espacio de trabajo. La última cinta es la cinta de salida, en ella la cabeza escribe la respuesta final antes de que la máquina de Turing se detenga.

Conjunto de operaciones

La máquina tiene un conjunto finito de estados $\mathcal{Q}$ y un registro que tiene a un único elemento de $\mathcal{Q}$, ese símbolo es el estado de la máquina en ese instante. Ese símbolo determina la siguiente acción en el siguiente paso computacional que puede consistir en alguno de los siguientes: (1) Leer un símbolo en una de las k-cintas, (2) leer o escribir un símbolo en el espacio de trabajo, (3) cambiar el estado por otro en $\mathcal{Q}$ y (4) mover las cabezas de cada cinta una celda a la izquierda, a la derecha o no moverlas.

3.1.1. Las máquinas de Turing como una descripción formal de los algoritmos

Aunque a veces los algoritmos se puedan describir mejor en lenguaje natural y simple, esta descripción de los algoritmos nos permite discutirlos a un nivel matemático y formal. De la misma manera que uno describe un algoritmo en un lenguaje de programación para que una computadora pueda ejecutarlo.

Definición 3.1.1. *Una máquina de Turing M está descrita por una tupla $(\Gamma, \mathcal{Q}, \delta)$ de manera que:*

1. Γ *es un conjunto finito de símbolos que se pueden escribir en las cintas de M. El símbolo vacío $\square$, el cero 0, el 1 y un símbolo de inicio $\triangleright$. A este conjunto le llamamos alfabeto.*
2. $\mathcal{Q}$ *es un conjunto finito de símbolos que representan los estados en los que M puede estar. El estado de inicio q_{inicio} y el estado de final q_{fin} son elementos de $\mathcal{Q}$.*
3. $\delta : \mathcal{Q} \times \Gamma^k \to \mathcal{Q} \times \Gamma^{k-1} \times \{L, S, R\}^k$ *es la función de transición y describe cómo actúa la máquina en cada paso.*

IF			THEN			
Input symbol read	Work/ output tape symbol read	Current state	Move input head	New work/ output tape symbol	Move work/ output tape	New state
⋮	⋮	⋮	⋮	⋮	⋮	⋮
a	b	q	**R**ight →	**b'**	**L**eft ←	**q'**
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Figura 3.3: Ejemplo de función de transición de una máquina de Turing de dos cintas. [24]

Si la máquina está en estado $q \in \mathcal{Q}$ y $(\sigma_0, ..., \sigma_k)$ son los símbolos que se están leyendo actualmente en cada una de las k cintas y $\delta(q, (\sigma_1, ..., \sigma_k)) = (q', (\sigma'_2, ..., \sigma'_k), z)$ donde

$z \in \{L, R, S\}^k$, entonces en el siguiente paso los símbolos en que se leyeron en las últimas $k-1$ cintas son reemplazados por los símbolos $(\sigma_2', ..., \sigma_k')$, cada una de las cintas se movió una de acuerdo a z, donde L es moverse a la izquierda, R a la derecha y S no moverse y el estado de la máquina es q'

Todas las cintas del espacio de trabajo inician en el símbolo de inicio $\triangleright$ y tienen el símbolo vacío $\square$ en todas sus demás celdas. La celda de inicio empieza con el símbolo de inicio $\triangleright$, una cantidad finita de celdas consecutivas no vacías y en el resto de las celdas el símbolo vacío. Cuando todas las celdas están en el símbolo de inicio y el estado de la máquina es q_{inicio} decimos que la máquina de Turing está en *configuración de inicio*. Cada paso de la máquina de Turing resulta de aplicar la función δ como se describió anteriormente. Cuando la función de transición selecciona el estado q_{fin} la máquina de Turing se detiene. Para la teoría de complejidad sólo nos interesan las máquinas de Turing que se detienen en un número finito de pasos.

Ejemplo

A continuación vamos a describir de manera precisa un ejemplo de máquina de Turing. La máquina decidirá si una sucesión $x \in \{0, 1\}^*$ es un palíndromo o no: $PAL(x) = 0$ si x no es un palíndromo y $PAL(x) = 1$ si x es un palíndromo. Vamos a mostrar que el tiempo de corrida para una sucesión de longitud n es de $3n$.

Esta máquina de Turing usará tres cintas: entrada, trabajo y salida. El alfabeto será $\Gamma = \{\triangleright, \square, 0, 1\}$. El conjunto de estados será $\mathcal{Q} = \{q_{inicio}, q_{copiar}, q_{izquierda}, q_{probar}, q_{fin}\}$. La función de transición está definida de la siguiente manera: [1]

1. Si la máquina de Turing tiene estado q_{inicio} entonces en la cinta de inicio se mueve a la izquierda y el estado se cambia a q_{copiar}.

2. Si la máquina de Turing tiene estado q_{copiar} entonces consideraremos dos casos. Que el símbolo leído en la cinta de entrada sea $\square$ o no.

   a) Si el símbolo leído en la cinta de entrada no es $\square$ entonces se mueven a la derecha las cabezas de la cinta de entrada y de la cinta de trabajo. Se escribe en la cinta de trabajo el símbolo que se leyó en la cinta de entrada. El estado de la máquina se queda como q_{copiar}

   b) Si el símbolo leído en la cinta de entrada es $\square$ entonces cambia el estado a $q_{izquierda}$.

3. Si la máquina de Turing tiene estado $q_{izquierda}$ entonces consideraremos dos casos. Que el símbolo leído en la cinta de entrada sea $\triangleright$ o no.

   a) Si el símbolo leído en la cinta de entrada no es $\triangleright$ entonces mueve a la izquierda la cinta de entrada. El estado de la máquina se queda como $q_{izquierda}$

   b) Si el símbolo leído en la cinta de entrada es $\triangleright$ entonces se mueve la cabeza de la cinta de entrada a la derecha y la cabeza de la cinta de trabajo a la izquierda. El estado cambia a q_{probar}

4. Si la máquina de Turing tiene estado q_{prueba} entonces consideraremos tres casos.

   a) Si el símbolo leído en la cabeza de la cinta de entrada es $\square$ y el leído en la cabeza de la cinta de trabajo es $\triangleright$ entonces la cabeza de la cinta de salida escribe uno y cambia el estado a q_{fin}.

[1] salvo que se indique lo contrario ninguna cabeza de cinta se mueve

b) Si el símbolo que lee la cabeza de trabajo y la cabeza de entrada son diferentes entonces la cabeza de salida escribe 0 y el estado cambia a q_{fin}

c) Si el símbolo que lee la cabeza de entrada y la cabeza de trabajo son iguales entonces la cabeza de la cinta de entrada se mueve a la derecha y la cabeza de la cinta de salida se mueve a la izquierda y el estado se queda como q_{prueba}.

Para ver que esta máquina de Turing tiene un tiempo de corrida es a lo más $3n$ donde n es la longitud de la sucesión de entrada es suficiente notar que: a la máquina de Turing le tomará n pasos copiar la cinta de entrada en la cinta de trabajo, luego le tomará n pasos en los que estará en estado $q_{izquierda}$ es decir cuando la cabeza de la cinta de entrada recorre toda la cinta hasta el inicio y por último a la cinta le toma a lo más n pasos en los que está en estado q_{prueba} es decir cuando las cabezas de las cintas de entrada y trabajo recorren sobre lo que han escrito y comparan las lecturas de cada cinta.

3.2. La clase P

La complejidad de un algoritmo se pude medir de acuerdo al número de pasos que tarda en detenerse en función de la longitud de la entrada.

Definición 3.2.1. *Sea M una máquina de Turing determinista. El tiempo de corrida de M es la función $T : \mathbb{N} \to \mathbb{N}$ tal que $T(n)$ es el máximo número de pasos para que M ejecute una entrada de longitud n.*
Si $T(n)$ es el tiempo de corrida de una máquina Turing M, entonces decimos que M corre con tiempo $T(n)$.

Calcular con precisión la complejidad de una máquina de Turing suele ser muy complicado por lo que sólo la aproximamos. Convencionalmente estimamos la complejidad de la máquina de Turing a partir de su comportamiento en entradas muy grandes, a esto se le llama análisis asintótico. Para definir esto con precisión necesitamos introducir la notación de la O-grande.

Definición 3.2.2. *Sea M una máquina de Turing con tiempo de corrida $T(n)$ y sea $g : \mathbb{N} \to \mathbb{R}$. Decimos que $T(n) = O(g(n))$ si existen $c, n_0 \in \mathbb{N}$ tal que para cada $n \geq n_0$ se cumple que*

$$T(n) \leq cg(n).$$

Cuando $T(n) = O(g(n))$ decimos que $g(n)$ es una cota superior asintótica para $f(n)$.

Por ejemplo, si el tiempo de corrida de una Máquina Turing M es una función $T(n) = 5n^3 + 2n^2 + 22n + 6$, entonces podemos considerar el término de orden superior: $5n^3$ y descartar los demás términos para encontrar una función que es cota superior asintótica de $T(n)$. En este caso tenemos que $T(n) = O(n^3)$, ya que también podemos descartar al coeficiente del término de orden superior.

Definición 3.2.3. *La clase $DTIME(T(n))$ son las máquinas de Turing para las que existe una cuyo tiempo de complejidad es $O(T(n))$.*

La D en $DTIME$ es por máquina de Turing determinista. Ahora, podemos hablar de la clase de complejidad P. Es la clase de máquinas de Turing que determinan un algoritmo eficiente. Es decir, un algoritmo cuyo tiempo de corrida es polinomial.

Definición 3.2.4. *P es la clase de máquinas de Turing deterministas con un tiempo de complejidad polinomial. Es decir,*

$$P = \bigcup_{c \in \mathbb{N}} DTIME(n^c).$$

Las diferencias polinomiales entre algoritmos en la vida real son profundamente significativas, un crecimiento de orden 1 es mucho menor que una de orden 3, sin embargo optamos por no distinguir entre estas diferencias para concentrarnos con distinguir a los algoritmos que corren en tiempo de complejidad polinomial y los que corren en tiempo exponencial.

La clase P juega un papel fundamental en la teoría de complejidad por dos razones centrales. La primera es que la definición es robusta con respecto al modelo de computación. Como mencionamos, existe un teorema que establece que el tiempo de complejidad que puede tener una máquina de Turing determinista en un modelo de computación cambia a lo más en orden cuadrado al cambiar el modelo de computación. Es decir si un lenguaje decidible tiene tiempo de complejidad polinomial de acuerdo a un modelo de computación determinista, entonces ese mismo lenguaje también tiene un tiempo de complejidad polinomial de acuerdo con cualquier otro modelo de computación determinista. Por otro lado, la clase P representa a todos los problemas que sí se pueden resolver de manera realista por una computadora.

Vamos a decir que un problema pertenece a P si existe un algoritmo (o máquina de Turing) que lo resuelva tal que ésta pertenezca a P. Por ejemplo, el problema que desarrollamos sobre si una sucesión finita de ceros y unos es o no un palíndromo está en P porque la máquina de Turing que calcula a la función PAL tiene tiempo de complejidad polinomial. Algunos ejemplos de otros problemas que se puede demostrar que están en P son:

1. Dada una gráfica dirigida y dos nodos, ¿existe un camino que inicie en uno de los nodos y termine en el otro? (PATH)
2. Dados dos números naturales, ¿los números son primos relativos? (REALPRIME)
3. Encontrar el máximo común divisor de dos números. (EUCLIDEAN)

Es importante notar que aunque siempre se puede resolver un problema a través de un algoritmo exhaustivo que corra a un tiempo de complejidad exponencial, para los problemas en P podemos encontrar máquinas de Turing que los resuelvan en tiempo de complejidad polinomial.

3.3. Las clases *NP*, *NP*-completo y NP-duro

Hasta ahora hemos establecido que un problema de optimización combinatoria pertenece a P si hay una máquina de Turing en P que lo resuelva. Ahora, un problema de optimización combinatoria pertenece a P si hay una máquina de Turing no determinista en *NP* que lo resuelve. O, equivalentemente, si el problema se puede verificar en tiempo polinomial. Vamos a describir los conceptos necesarios para formular precisamente esta equivalencia y la vamos a probar.

Las máquinas de Turing no deterministas son herramientas teóricas que se usan en la teoría de computación. Mientras que una máquina de Turing determinista formaliza el concepto de algoritmo, una máquina de Turing no determinista formaliza un árbol de posibles acciones que puede seguir un algoritmo.

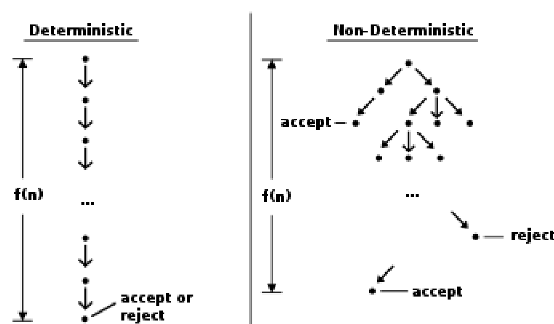

Figura 3.4: Interpretación gráfica de una máquina de Turing determinista y una no determinista. [24]

Formalmente, una máquina de Turing no determinista es una máquina de Turing determinista con dos diferencias:

1. Existen dos funciones de transición δ_0 y δ_1, tales que en cada paso computacional, la máquina de Turing escoge aleatoriamente usar a δ_0 o a δ_1 como función de transición.
2. Y tiene un estado adicional $q_{aceptar} \in \mathcal{Q}$. Decimos que una máquina de Turing no determinista acepta a una entrada $x \in \{0,1\}^*$ si para alguna corrida de la máquina no determinista de Turing, la máquina tiene estado $q_{aceptar}$. En otro caso, decimos que la máquina de Turing rechaza a la entrada $x \in \{0,1\}^*$.

Análogamente al tiempo de corrida de una máquina de Turing determinista, podemos definir el tiempo de corrida de una máquina de Turing no determinista.

Definición 3.3.1. *Sea N una máquina de Turing no determinista. El tiempo de complejidad de N es la función $T : \mathbb{N} \to \mathbb{N}$ tal que $T(n)$ es el máximo número de pasos en cualquier rama para que N ejecute una entrada de longitud n.*

Podemos interpretar gráficamente a una máquina de Turing como una línea recta, recordemos que una maquina de Turing es una formalización de un algoritmo. Por otro lado, una máquina de Turing no determinista es como un árbol de posibilidades. En cada paso computacional el árbol se bifurca porque existen dos funciones de transición que la máquina de Turing podría utilizar. Cada rama de ese árbol es una posible secuencia que la máquina podría seguir. Para la máquina de Turing determinista, el tiempo de complejidad es la máxima longitud que la secuencia de pasos puede tener, o el largo de la línea. Para una máquina no determinista, el tiempo de complejidad es la altura de ese árbol de posibilidades, es decir la máxima longitud de una de las ramas en el árbol de posibilidades.

Ahora análogamente a como definimos $DTIME$ podemos definir $NDTIME$ y con esto podemos definir a la clase *NP*.

Definición 3.3.2. *Sea $T : \mathbb{N} \to \mathbb{N}$ una función. La clase $NDTIME(T(n))$ son las máquinas no deterministas de Turing cuyo tiempo de complejidad es $O(T(n))$.*

Definición 3.3.3. *NP es la clase de lenguajes que se pueden decidir por una máquina de Turing no determinista en un tiempo de complejidad polinomial. Es decir,*

$$NP = \bigcup_{c \in \mathbb{N}} NDTIME(n^c).$$

Ahora vamos formular precisamente y probar que las máquinas de Turing no deterministas que pertenecen a *NP* son aquellos para los que pueden ser verificados en tiempo polinomial.

Definición 3.3.4. *Un verificador V para una máquina de Turing no determinista M es una máquina de Turing determinista tal que M acepta a $w \in \{0,1\}^*$ si y sólo si existe $c \in \{0,1\}^*$ tal que V acepta a (w, c).*

Por ejemplo, supongamos que M es una máquina no determinista que resuelve el problema de satisfacibilidad booleana. Es decir, es un máquina de Turing no determinista que acepta o rechaza a la instancia w del problema si la instancia es satisfacible o no. Un verificador para esta máquina sería otra máquina determinista que acepte a (w, c) si c es una asignación de valores de verdad para las variables que ocurren en w de manera que c satisface a w. Recordemos que $w, c \in \{0,1\}^*$ por lo que es necesario precisar cómo se codifican las instancias del problema y las asignaciones de valores de verdad en términos de elementos de $\{0,1\}^*$. Sin que todavía tengamos esta codificación explícita, es claro que podemos construir un algoritmo en el que dadas una instancia del problema y una asignación podamos determinar si esa asignación satisface a la instancia.

El tiempo de complejidad de un verificador V es el correspondiente tiempo de complejidad de V como una máquina de Turing determinista.

Teorema 3.3.1. *Una máquina de Turing no determinista está en NP si y sólo existe un verificador de ese máquina con tiempo de complejidad polinomial.*

Demostración. Sea M una máquina de Turing no determinista que pertenece a *NP*. Vamos a definir una máquina de Turing determinista V como sigue.

Para cada $w \in \{0,1\}^*$ tal que M acepta a w. Sea $n \in \mathbb{N}$ la máxima cantidad de pasos computacionales que le toma a M aceptar a w. Consideremos la secuencia de funciones de transición que se usan en la máxima cantidad de pasos computacionales tal que M acepta a w: $(\delta_{j_i})_{i \leq n}$, es decir $j_i \in \{0,1\}$. Para cada $i \leq n$ sea $c_{w,i} = j$ donde j es el índice de la función de transición que se usa en el paso computacional i. Sea $c_w = (c_{w,1}, ..., c_{w,n})$.

Sea V la máquina de Turing tal que en cada entrada (w, c_w) *imita* a la máquina M en cada paso computacional y en el que en el i-ésimo paso sigue la función de transición $\delta_{c_w(i)}$.

Dado que M está en *NP* y M acepta a w entonces el tiempo de corrida de M en w depende polinomialmente de la longitud de (w, c_w). Como la máquina V imita a M entonces V tiene el mismo tiempo de corrida que depende de la longitud de (w, c_w).

Por otro lado, sea V un verificador para una máquina de Turing M con un orden de complejidad polinomial. Definimos una máquina de Turing no determinista M de manera que las funciones de transición δ_0, y δ_1 determinan azarosamente un vector c. En w, M imita a V con entrada (w, c) donde c es el vector que generan las funciones de transición de M. □

Para cerrar esta sección introduciremos la clase $NP - completo$, la complejidad de los problemas $NP - completos$ está relacionada con la complejidad de toda la clase de problemas en *NP* en el sentido en el que si existe un algoritmo que resuelve en tiempo polinomial a un problema $NP - completo$ entonces para cada problema en *NP* existirá un algoritmo que corra en tiempo polinomial que resuelve el problema. Para describir con más detalle esta clase de problemas necesitamos las siguientes definiciones.

Definición 3.3.5. *Una función $f : \{0,1\}^* \to \{0,1\}^*$ es computable en tiempo de complejidad polinomial si existe un algoritmo en P que calcula a f.*

Definición 3.3.6. *Una máquina de Turing A se reduce en tiempo polinomial a una máquina de Turing B si existe una función computable f tal que A acepta a w si y sólo si B acepta a $f(w)$.*
??

Tenemos que la clase *NP* es *cerrada* bajo reducciones polinomiales. Formalmente:

Teorema 3.3.2. *Si A, B son máquinas de Turing, A se reduce en tiempo polinomial a B y $B \in P$ entonces $A \in P$.*

Ahora podemos definir con precisión el concepto de problema *NP*-duro.

Definición 3.3.7. *Un problema B es NP-duro si se cumplen que cualquier problema A en NP se reduce en tiempo polinomial a B.*

A partir de esto, podemos definir *NP*-completo.

Definición 3.3.8. *Un problema B es NP-completo si es NP y NP-duro.*

Si encontramos un problema *NP*-completo que sea P entonces tendríamos que $P = NP$. En la siguiente sección mostraremos que el problema de satisfacibilidad booleana es $NP-completo$.

3.4. Teorema de Cook-Levine

Ahora, demostraremos que el problema de satisfacibilidad booleana es $NP-completo$. Es claro que el problema pertenece a *NP* ya que al final del capítulo anterior mostramos un algoritmo en tiempo polinomial que verifica al problema.

Teorema 3.4.1. *El problema de satisfacibilidad booleana es NP-completo.*

Demostración. Sea $A \in NP$ y N una máquina de Turing no determinista que decide a A en un tiempo n^k para alguna k, de hecho por conveniencia y sin pérdida de generalidad supondremos que la máquina corre en un tiempo $n^k - 3$. La siguiente noción nos permitirá hacer una descripción más precisa para construir la reducción que buscamos.

Un **cuadro** para N es una tabla de $n^k \times n^k$ tal que las filas de la tabla son las configuraciones de una rama de N sobre un cálculo con entrada w. La siguiente Figura es el ejemplo de un cuadro.

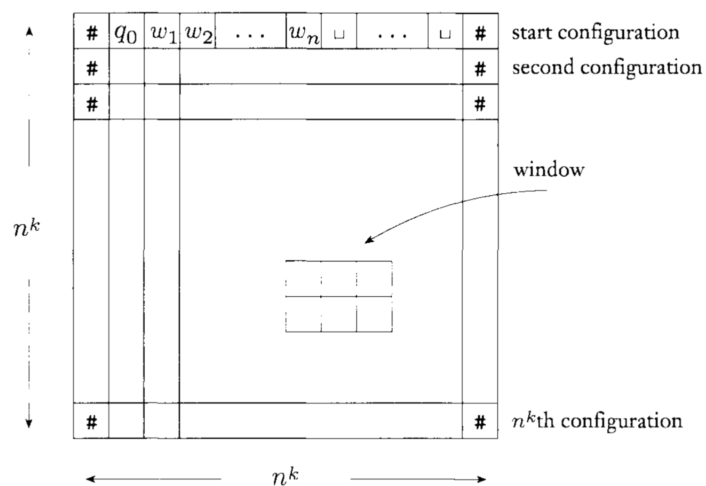

Figura 3.5: Ilustración de un cuadro. [51]

Por conveniencia, vamos a decir que cada configuración empieza y termina con el símbolo $\sharp$. La primera fila es la configuración inicial de N sobre w y cada fila precede a la siguiente de acuerdo alguna función de transición de N.

Un cuadro es de **aceptación** si alguna de sus filas es una configuración N con estado de aceptación. Con esta noción, el problema de que N acepte a w es equivalente a determinar si existe un cuadro de N que empiece con la configuración de w que sea de aceptación.

Nuestro objetivo es construir una fórmula ϕ que sea satisfacible si y sólo si un cierto cuadro de N es de aceptación para una entrada w.

Empezaremos describiendo sus variables. Sea Q el conjunto de estados y Γ el alfabeto de la máquina de Turing N. Llamemos $C = Q \cup \Gamma \cup \{\sharp\}$. Dado un cuadro de la máquina de Turing cada una de sus $(n^k)^2$ celdas tiene coordenadas $i, j \leq n^k$ y tiene escrito un elemento $s \in C$. Para cada $i, j < n^k$ y $s \in C$ vamos a construir las variables $x_{i,j,s}$ donde $x_{i,j,s} = 1$ si y sólo si en la celda (i, j) está escrito el símbolo s.

Ahora procedemos a construir la fórmula, ésta será la conjunción de cuatro subfórmulas:

$$\phi = \phi_{celda} \wedge \phi_{inicio} \wedge \phi_{movimiento} \wedge \phi_{aceptar}.$$

La fórmula ϕ_{celda} garantizará que todas las celdas del cuadro tengan un símbolo escrito.

$$\phi_{celda} = \bigvee_{i,j<n^k} \left[\left(\bigwedge_{s \in C} x_{i,j,s} \right) \wedge \left(\bigvee_{s \neq t} (\neg x_{i,j,s} \vee \neg x_{i,j,t}) \right) \right].$$

La conjunción establece que ninguna celda está vacía y la disyunción establece que no puede haber más de un símbolo escrito en la celda.

Las partes de la fórmula $\phi_{inicio}, \phi_{movimiento}$, y $\phi_{aceptar}$ garantizarán que los símbolos del cuadro correspondan a las variables de la fórmula.

La fórmula ϕ_{inicio} garantiza que la primera fila del cuadro sea el estado de la máquina N con la entrada w.

$$\phi_{inicio} = x_{1,1,\sharp} \wedge x_{1,2,q_0} \wedge \cdots \wedge x_{1,n+2,w_n} \wedge \left(\bigwedge_{j=n+3}^{n^k-1} x_{1,j,\sqcup} \right) \wedge x_{1,n^k,\sharp}.$$

La fórmula $\phi_{aceptar}$ garantiza que en alguno de las configuraciones la máquina tomó el estado de aceptación $q_{aceptar}$.

$$\phi_{aceptar} = \bigvee_{i,j<n^k} x_{i,j,q_{aceptar}}.$$

La fórmula $\phi_{movimiento}$ garantiza que las configuraciones estén escritas en el cuadro de acuerdo a la función de transición de N. Para poder hacer esto necesitamos recurrir a la noción de ventana de un cuadro. Una ventana de un cuadro es una subtabla de seis celdas de 2×3. La noción de ventana nos ayuda a reducir la cantidad de verificaciones que debemos de realizar para garantizar que las configuraciones que se escriben siguen la función de transición de N porque si la primera fila del cuadro es la configuración inicial y cada ventana de la tabla sigue la función de transición de N entonces cada fila de la tabla es la configuración correspondiente que precede a la configuración anterior. Este hecho nos permite escribir a la fórmula $\phi_{movimiento}$:

$$\phi_{movimiento} = \bigwedge_{i,j<n^k} \left(\text{la ventana } (i,j) \text{ sigue la función de transición de } N\right),$$

podemos reemplazar “la ventana (i,j) sigue la función de transición de N” con la siguiente fórmula, donde $a_1, \ldots, a_6$ son los arreglos que las ventanas legales podrían tener según la función de transición.

$$\bigvee_{a_1,\ldots,a_6} \left(x_{i,j,a_1} \wedge \ldots x_{i,j,a_6}\right).$$

Para terminar es suficiente mostrar que el algoritmo de construcción de la fórmula ϕ tiene orden de complejidad asintótico polinomial.

Primero observemos que el número de variables de ϕ es ℓn^{2k} donde ℓ es la cardinalidad de C. Ahora estimemos la longitud de cada una de las partes de la fórmula. La fórmula ϕ_{celda} tiene una variable por cada celda del cuadro, entonces tiene orden n^{2k}, las fórmulas ϕ_{inicio} y $\phi_{aceptar}$ tienen longitud n^k y por último la fórmula $\phi_{movimiento}$ tiene un orden n^{2k}. Por lo tanto el orden de complejidad asintótico es n^{2k}.

□

Ahora que sabemos que el problema de satisfacibilidad booleana es *NP*-completo, para ver que otro problema es *NP*-completo es suficiente verificar que éste se puede reducir al problema de satisfacibilidad booleana.

Capítulo 4

Heurísticas

Imaginemos el problema de encontrar un alfiler en un pajar. Con una diferencia, puede ser que haya más de un alfiler o que no lo haya. Un método determinista de encontrar la aguja en el pajar es ir una por una de las pajas y verificar si es o no la aguja. Además de requerir mucho tiempo requeriría dividir al pajar entre al menos dos montículos, lo revisado y lo no revisado. Resolver este problema por fuerza bruta requiere de mucho tiempo y espacio. Podemos llamar al pajar espacio de búsqueda, a cada paja le podemos llamar solución y a cada alfiler solución óptima. Una metodología para encontrar el alfiler sería una metodología para explorar el espacio de búsqueda. Los métodos heurísticos pueden ser pensados como metodologías de búsqueda. La metáfora se puede extender con sus limitaciones por ejemplo: una vecindad de una solución es puñado de paja que está cerca de esa solución y una búsqueda local es buscar al alfiler por puñados.

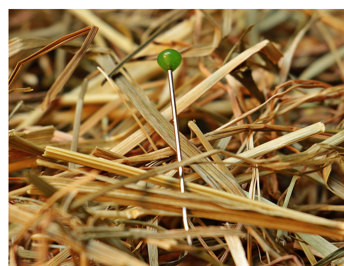

Figura 4.1: Alfiler en un pajar

En este capítulo vamos a introducir la teoría general de metaheurísticas como metodologías de búsqueda para resolver problemas de optimización combinatoria. Primero vamos a describir lo que significa un proceso de búsqueda local, que es un proceso subyacente en varias de las metaheurísticas que estudiaremos. Después describiremos siete metaheurísticas: Escalar colinas, Búsqueda Tabú, GRASP, Recocido Simulado, Búsqueda por vecindades variables, algoritmos genéticos y Optimización por colonias de hormigas. En cada uno de ellos describimos su contexto histórico, su pseudocódigo y su diagrama de flujo.

4.1. Búsqueda local

A las metodologías que usan las metaheurísticas las podemos distinguir entre: constructivas o de búsqueda local.De acuerdo con Blum y Roli [6] los algoritmos para resolver problemas combinatorios se pueden clasificar como: **completos** o **aproximados**. Dentro de los métodos aproximados, distinguen los métodos **constructivos** y los métodos de **búsqueda local**. Los métodos constructivos construyen una solución a partir del vacío en un proceso en el que le añaden componentes. Los métodos constructivos son más rápidos que los métodos de búsqueda local, aunque los métodos de búsqueda local tienden a dar mejores soluciones que los métodos constructivos. Los métodos de búsqueda local parten de una solución que reemplazan iterativamente por una mejor solución dentro de una estructura bien definida de soluciones llamada vecindad. Una vecindad de una solución $s \in S$ es un

conjunto $N(s) \subseteq S$ tal que cada solución $s' \in N(S)$ se obtiene a partir de un *movimiento* de s, es decir que algún o algunos elementos de s y s' pueden ser diferentes. Las vecindades de una solución pueden contener soluciones que no sean factibles.

En los últimos 20 años ha surgido una nueva clase de métodos de aproximación que combinan las estrategias básicas de búsqueda local y procesos constructivos con en rutinas de trabajo que exploran de una manera eficiente y eficaz el espacio de búsqueda. A estos métodos les llamamos **metaheurísticas**, el término lo acuñó Glover en [26]. También se les han llamado metaheurísticas modernas, por ejemplo [47].

Algunos ejemplos de metahurísticas son: la optimización por colonias de hormigas, algoritmos genéticos, búsqueda local iterativa, Recocido Simulado, búsqueda tabú, etc. Hasta ahora no hay una única definición de metaheurística. De acuerdo con [45] [59] [54] las propiedades fundamentales que caracterizan a las metaheurísticas son las siguientes.

1. Son estrategias que *guían* el proceso de búsqueda.
2. Exploran eficientemente el espacio de búsqueda para encontrar *buenas* soluciones.
3. Son estrategias que combinan desde una simple búsqueda local hasta un procedimiento complejo de aprendizaje.
4. No son deterministas.
5. Incorporan mecanismos para evitar que la búsqueda quede confinada a regiones del espacio de búsqueda.
6. Sus elementos básicos se pueden describir de una manera abstracta.
7. No son específicas a una sola instancia de un problema.
8. Las metaheurísticas más avanzadas usan la experiencia de búsqueda para guiar la búsqueda.

Las metaheurísticas las podemos clasificar de acuerdo a la filosofía que siguen para realizar su búsqueda. Por un lado tenemos las metaheurísticas de **trayectoria** y por el otro las metaheurísticas **basadas en poblaciones**. Las metaheurísticas de trayectoria son extensiones *inteligentes* del algoritmo de búsqueda local, algunos ejemplos de estas metaheurísticas son: búsqueda tabú, búsqueda por vecindades variables, GRASP y Recocido Simulado. Las metaheurísticas de poblaciones *aprenden* implícita o explícitamente cuáles son las variables decisivas que identifican regiones de alta calidad en el espacio de búsqueda. Esto lo logran haciendo un muestreo sesgado del espacio de búsqueda, un ejemplo son los algoritmos genéticos que recombinan soluciones o las colonias de hormigas que muestrean el espacio de acuerdo con una distribución de probabilidad.

4.2. Metaheurísticas de trayectoria

El término *trayectoria* se usa porque el proceso de búsqueda en el espacio determina una trayectoria en el espacio de búsqueda. Estos procesos inician en una solución inicial a partir de la cual describen una trayectoria en el espacio de búsqueda. A continuación haremos un bosquejo de las metaheurísticas de trayectoria que vamos a estudiar.

En el problema de la satisfacibilidad booleana con n variables, el espacio de búsqueda es el conjunto de vectores de ceros y unos de longitud n. El primer problema del trabajo es determinar cuál es la estructura de vecindad más apropiada para cada heurística. En la Función A.3 implementamos una estructura de vecindad a partir de cuatro parámetros: s es la solución a la que se le construye su vecindad, n es el número de cambios que se le hace a la solución c para construir a un vecino, n es el tamaño de la vecindad y *seed* es un número entero que fija la semilla de los números aleatorios. Considerar como vecinos sólo a las soluciones con un cambio es muy restrictivo y no acotar el tamaño de la vecindad de c cambios es muy exhaustivo.

4.2.1. Escalar las colinas

Un proceso de búsqueda local termina cuando no se encuentra una mejor solución en la vecindad. La efectividad de una búsqueda local depende de una variedad de aspectos como la estructura de la vecindad, la estrategia de búsqueda en la vecindad o la solución inicial. Se pueden seguir dos estrategias de búsqueda en la vecindad: *mejor-mejora* o *primera-mejora*. La primera consiste en evaluar a todos los elementos de la vecindad y escoger la mejor, la segunda consiste en evaluar a cada elemento de la vecindad y cuando una de ellas es mejor que la actual escogerla.

En el siguiente pseudocódigo describimos cómo hacer un proceso de búsqueda local. La entrada de este proceso es una solución inicial S y la salida una solución S'. Para llevar a cabo este proceso requerimos tener definida una estructura de vecindad para una asignación dada. Este proceso es exhaustivo, es decir que evalúa en la función objetivo a cada solución de la vecindad.

Algoritmo 2 Búsqueda local

Require: V un conjunto de soluciones
  for $S \in V$ **do**
    Evaluar $f(S)$
  end for
  Escoger S tal que $f(S)$ sea el máximo

El proceso de búsqueda local es una heurística que puede ser descrita a través del siguiente diagrama de flujo 4.2.

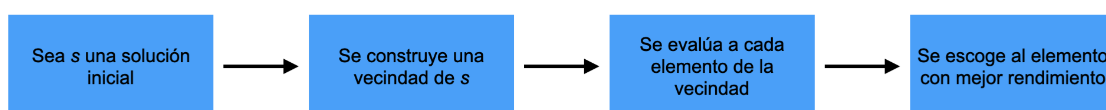

Figura 4.2: Algoritmo de búsqueda local básica.

La implementación de un proceso de búsqueda local está representada en la Función A.12 que aparece en el Apéndice A. Esta función tiene dos entradas, un conjunto de soluciones y una instancia del problema. Dicho conjunto de soluciones puede ser una vecindad de una solución específica, la implementación de la construcción de una vecindad dada una solución está representada en la Función A.3.

Una búsqueda local es un proceso que dada una solución en el espacio de búsqueda obtiene un óptimo local para alguna estructura de vecindad de la solución dada. Al proceso

iterativo de realizar búsquedas locales sobre los óptimos locales que resultan de búsquedas locales le llamamos **escalar las colinas** como una traducción de *Hill Climbing*.

En el siguiente pseudocódigo representamos el proceso de escalar las colinas. La entrada de este proceso es un solución inicial y la salida es una *probablemente* mejor solución. Para llevar a cabo este proceso tenemos que tener definido un criterio de paro que determine cuántas iteraciones de búsqueda local se van a realizar y una estructura de vecindad dada una solución.

El método escalar colinas se enfrenta con un grave problema cuando el óptimo local de una vecindad también es el óptimo local de su propia vecindad. En este caso decimos que la búsqueda ha quedado atrapada en un atractor. Las metaheurísticas de búsqueda que describimos más adelante se enfrentan con este problema utilizando una variedad de estrategias.

La implementación de una proceso de escalar las colinas está representada en la Función A.13 que aparece en el Apéndice A. Para hacer este proceso utilizamos la función de búsqueda local A.12, la función que construye vecindades A.3 y como criterio de paro el tiempo computacional que transcurre desde que inicia el proceso.

4.2.2. Búsqueda Tabú

La búsqueda tabú es una de las metaheurísticas más usadas y más citadas. Esta metaheurística fue propuesta por primera vez por [26]. La búsqueda tabú usa la historia de la búsqueda para escapar de óptimos locales y para implementar una estrategia de exploración.

El algoritmo más sencillo de la búsqueda tabú usa una estrategia de memoria a corto plazo para escapar de mínimos locales y para evitar ciclos. La memoria a corto plazo se implementa como una lista tabú que lleva un registro de las soluciones que se visitaron recientemente y a las que se le prohíbe al algoritmo volver a visitar. Así, las vecindades de las soluciones están restringidas a no intersectarse con la lista tabú. De la vecindad restringida en cada iteración se selecciona al óptimo local y se le incluye en la lista tabú. El algoritmo termina si se cumple la condición de paro o si la vecindad de una solución resulta vacía tras eliminar todos los elementos de ésta que pertenezcan a lista tabú.

El plazo de la memoria de esta estrategia está controlado por la longitud de la lista tabú. Si la longitud es pequeña, entonces la búsqueda se concentrará en regiones pequeñas del espacio de soluciones porque permite volver a visitar soluciones después de pocas iteraciones. En cambio, si la longitud es grande, entonces la búsqueda explorará regiones más amplias porque no puede volver a visitar las mismas soluciones. La longitud de la lista tabú puede ser dinámica, por ejemplo puede borrar elementos al azar o cambiar su longitud de acuerdo a la repetición de soluciones visitadas o al rendimiento de la estrategia.

Una lista tabú de soluciones que tenga una gran longitud es muy impráctica. En vez de soluciones una lista tabú puede ser de atributos de soluciones. Un atributo puede ser un componente de una solución o una diferencia entre dos soluciones. Dado que más de un atributo puede ser considerado, se introduce una lista tabú para cada uno. De esta manera definimos los **atributos tabú** que se utilizan para filtrar las vecindades de las soluciones. Mientras que utilizar atributos es mucho más eficiente conlleva perder información. Para sobrellevar este problema, se puede implementar un **criterio de aspiración** que permite

que una solución que cumple un criterio tabú no sea filtrada de las vecindades.

El principio básico de la búsqueda tabú es seguir con la búsqueda siempre que un óptimo local se encuentre permitiendo movimientos sin mejora. Esto se logra previniendo que se visiten cíclicamente soluciones que ya se han visitado a través de una estructura de memoria llamada **lista tabú**. Esta idea está inspirada en las metodologías de explotar la información para guiar una búsqueda que surgen en un contexto de inteligencia artificial [44]. Un comentario importante es que Glover no veía a esta heurística propiamente como una heurística, más bien como una estrategia para guiar y controlar otra metaheurísticas internamente.

Listas tabú

Las listas tabú son los elementos distintivos de esta heurística. La lista tabú registra actividades prohibidas para la metodología de búsqueda, por ejemplo volver a visitar soluciones que se hayan visitado recientemente. Las listas tabú se almacenan en estructuras de memoria de corto plazo de un tamaño fijo y adecuado.

Registrar soluciones completas en listas tabú puede ser muy costoso en términos de memoria y tiempo, tanto para registrar las soluciones como para revisar si una cierta solución es tabú o no. Dos posibles estrategias para resolver este problema pueden ser: solamente registrar las últimas soluciones que se han evaluado o solamente registrar las características importantes de las soluciones tabú.

En la implementación estándar, la lista tabú tiene un tamaño fijo. No obstante, algunos autores [52, 55] han sugerido que variar la longitud de la lista tabú durante el proceso de búsqueda puede mejorar el desempeño de la heurística. Otra solución es borrar aleatoriamente intervalos de registros en la lista en vez de siempre borrar en los últimos registros [25].

Otros criterios de aspiración, más complicados, consisten en aceptar soluciones que hayan sido visitadas si hay una garantía de que al aceptarlas se evitan ciclos, esto criterios han sido propuestos por [30].

El proceso básico de búsqueda tabú está descrito en el siguiente pseudocódigo. La entrada para este proceso es una solución inicial y la salida es una solución *probablemente* mejor. El proceso utiliza una lista tabú T, al inicializar el proceso construimos esta lista en blanco. Requerimos haber definido previamente un proceso para determinar cuáles son las vecindades admisibles.

Algoritmo 3 Búsqueda Tabú

Require: S solución inicial
  $T \leftarrow \emptyset$
  $f^* \leftarrow f(S)$
  $S^* \leftarrow S$
  while Criterio de paro no se cumpla **do**
    $V(S) \leftarrow$ Vecindad admisible de S
    $S \leftarrow$ BúsquedaLocal(S^*, V)
    if $f(S) < f^*$ **then**
      $f^* \leftarrow f(S)$
      $S^* \leftarrow S$
      Actualizar T
    end if
  end while
  Regresar S

En cada iteración del proceso, para la mejor solución se construye una vecindad admisible, es decir una vecindad de manera que ninguno de los elementos pertenezca a la lista tabú. Después de construir esta vecindad admisible, se hace un proceso de búsqueda local básico. El proceso de búsqueda tabú esté descrito en el siguiente diagrama de flujo.

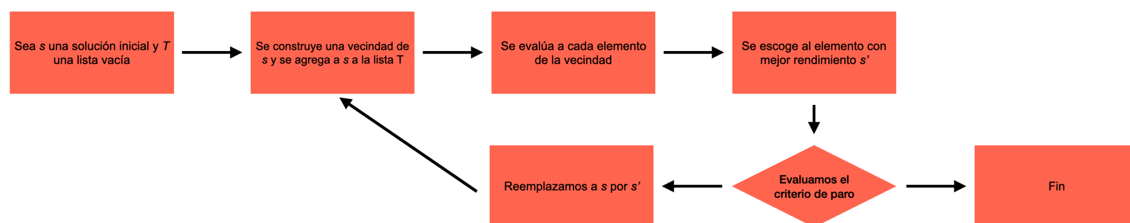

Figura 4.3: Diagrama de búsqueda tabú.

En la implementación de esta heurística desarrollamos una función que *limpia* a las vecindades que vamos a investigar usando búsqueda local. Es decir, que dada una vecindad y la lista tabú obtengamos la vecindad sin elementos de la lista tabú. Nuestra implementación convierte a todos los elementos de la vecindad en números naturales, compara a los números naturales obtenidos con la lista tabú, registra cada intersección en otra lista y en función de esa lista borra a los elementos de la vecindad. Esta función está descrita en A.14 del Apéndice A..

4.2.3. Recocido Simulado

Para la física de la materia condensada, el recocido es un proceso térmico para obtener estados de bajas energías de un sólido a través de un baño de calor, el proceso consiste en dos pasos.

1. Incrementar la temperatura del baño de calor hasta el máximo valor en el que el sólido se derrita.

2. Disminuir cuidadosamente la temperatura del baño de calor hasta que las partículas se organicen en un arreglo que adquiere el cuerpo sólido.

En la etapa líquida, todas las partículas se organizan aleatoriamente y durante el enfriamiento las partículas se organizan una retícula muy bien estructurada. Este fenómeno ocurre solamente si el valor máximo de la temperatura es suficientemente alto y el enfriamiento se hace lo suficientemente despacio.

El algoritmo para simular la evolución de un sólido en un proceso de recocido fue introducido por Metropolis [40]. Su algoritmo se basa en técnicas del método de Monte-Carlo y genera una secuencia de estados de un sólido de la siguiente manera: Dado el estado actual i de un sólido con energía E_i el siguiente estado j del sólido se obtiene de perturbar al estado actual del sólido en una pequeña distorsión, por ejemplo desplazar una de sus partículas. La energía del sólido en este estado es E_j. Si $E_j - E_i \leq 0$ entonces se acepta el estado j. Pero si la diferencia de energías es positiva entonces el estado j se acepta con probabilidad

$$e^{\frac{E_i - E_j}{k_B T}},$$

donde k_B es la constante de Boltzmann. A esta regla para transicionar entre estados se le conoce como criterio de Metropolis. Bajar la temperatura lo suficientemente lento permite que el sólido alcance un equilibrio térmico en cada temperatura. El equilibrio térmico está caracterizado por la distribución de Boltzmann que calcula la probabilidad de que el sólido en estado i tenga energía E_i a temperatura T.

La analogía entre un sistema físico de muchas partículas y un problema de optimización combinatoria radica en dos puntos: Las soluciones de un problema de optimización combinatoria son análogas a los estados de un sistema físico y el costo de una solución es equivalente a la energía de un estado. En los problemas de optimización combinatoria podemos introducir un parámetro que juega el papel de la temperatura.

Se dice que *Recocido Simulado* es la más antigua de las metaheurísticas, es uno de los primeros algoritmos que tiene una estrategia explícita que escapa de los mínimos locales. Este método proviene de un algoritmo estudiado en mecánica estadística, el algoritmo de Metropolis-Hastings [40]. La primera vez que se presentó como un algoritmo para resolver problemas de optimización combinatoria fue en [34].

El algoritmo empieza con una solución inicial s dada e inicializando un parámetro llamado "temperatura" T. En cada iteración, se muestrea una solución $s' \in N(s)$ y es aceptada como la nueva solución dependiendo de $f(s)$, $f(s')$ y T. Hay dos casos, si $f(s') < f(s)$ (o para un problema de maximización $f(s') > f(s)$) entonces se cambia a s por s'. En el otro caso $f(s) < f(s')$ (respectivamente $f(s) > f(s')$) entonces se intercambia a s por s' con una probabilidad que esté en función de T y $f(s') - f(s)$. Durante el proceso, T decrece de acuerdo con un esquema que se la llama "esquema de enfriamiento". Este proceso es análogo al de recocido de metales y vidrios que para tomar forma pasan de temperaturas muy altas a temperaturas muy bajas de acuerdo con un esquema de enfriamiento que permite que las moléculas se alineen.

Este proceso está dividido en dos etapas, una en la que se siguen trayectorias aleatorias y otra en la que se mejora el rendimiento iterativamente. En la primer etapa la probabilidad de mejora en el rendimiento es baja, pero permite la exploración en el espacio de búsqueda. En la medida que la probabilidad de que este proceso errático ocurra el algoritmo converge a un óptimo local. Dicha probabilidad está en función de dos variables, la temperatura y la

diferencia en el rendimiento de la nueva solución. La temperatura disminuye a lo largo del proceso de acuerdo con el esquema de enfriamiento y las diferencias en los rendimientos disminuyen en la medida en la que el algoritmo se acerca a óptimos locales. El desempeño del algoritmo está íntimamente ligado al esquema de enfriamiento. Se ha demostrado que en ciertos esquemas de enfriamiento la metaheurística converge a un óptimo global, no obstante estos esquemas no son prácticamente aplicables [2]. Uno de los esquemas de enfriamiento más usados sigue una ley geométrica que se corresponde con el decaimiento exponencial de la temperatura en un proceso de Recocido.

En los años 80 independientemente Kirckpatrick [34] y Černỳ [9] introdujeron el concepto de recocido en optimización combinatoria. Originalmente este concepto fue inspirado por la analogía entre el proceso físico de recocido de los sólidos y el problema de resolver grandes instancias de problemas combinatorios. Análogo al algoritmo de Metropolis, Recocido Simulado consta de tres etapas. La primera es la inicialización en donde definimos el parámetro de temperatura inicial, el factor del esquema de enfriamiento, la solución inicial y calculamos el valor de la solución inicial en la función objetivo. En la siguiente etapa, de acuerdo con un parámetro L se itera el proceso de búsqueda local en el que se acepta la solución si mejora su valor en la función objetivo o de acuerdo a una probabilidad. En la última etapa se actualiza el valor de la temperatura de acuerdo con el esquema de enfriamiento.

El valor del parámetro T_0 determina la temperatura inicial del *baño de temperatura* en el que el sólido se debe de fundir en un líquido. El parámetro $\alpha \in [0, 1)$ reduce el valor de la temperatura con iteración del ciclo, este valor es el que corresponde a disminuir la temperatura *suficientemente lento*. El parámetro L determina el tamaño de la búsqueda a una misma temperatura, éste valor correspondería al tiempo durante el que el sólido se encuentra a un equilibrio térmico. En general, los parámetros L y α pueden ser dinámicos. Observemos que si reemplazamos $e^{\frac{f(S')-f(S^*)}{T}}$ por 0 entonces el proceso de Recocido Simulado sería simplemente un procedimiento de escalar colinas usando vecindades más pequeñas, o quizás del mismo tamaño, que las vecindades usuales. Si se escogen adecuadamente los parámetros ser puede demostrar que el método de Recocido Simulado converge asintóticamente a un óptimo global. La formulación correspondiente así como la demostración de dicho teorema exceden los límites de este trabajo, pero se pueden ver en [41].

En el siguiente pseudocódigo representamos el proceso de Recocido Simulado. Para realizar este proceso como entrada requerimos una temperatura inicial y una solución inicial. Previamente tenemos que haber definido una estructura de vecindad y un valor α que representa al esquema de enfriamiento. Si tras hacer un proceso de búsqueda local, si es una mejora entonces se admite y si no lo es entonces se calcula $e^{\frac{f(S')-f(S^*)}{T}}$ y se encuentra un número aleatorio entre cero y uno. En caso de que el número sea mayor que el valor dado entonces se admite a la solución que se generó en el proceso de búsqueda local. Después de cada iteración, disminuimos el valor de la temperatura de acuerdo con el proceso de enfriamiento.

Algoritmo 4 Recocido Simulado

Require: S solución inicial, L longitud de búsqueda, T_0 temperatura inicial, α tasa de enfriamiento
  $T \leftarrow T_0$
  $f^* \leftarrow f(S)$
  $S^* \leftarrow S$
  while Criterio de paro no se cumpla **do**
    for $i = 1 : L$ **do**
      Generar solución $S' \in V(S)$
      if $f(S') < f^*$ **then**
        $f^* \leftarrow f(S')$
        $S^* \leftarrow S'$
      else
        if $e^{\frac{f(S')-f(S^*)}{T}}$ <rand[0,1) **then**
          $f^* \leftarrow f(S')$
          $S^* \leftarrow S'$
        end if
      end if
    end for
    $T \leftarrow \alpha T$

  end while
  Regresar S^*

Este proceso lo podemos representar a través de un diagrama de flujo.

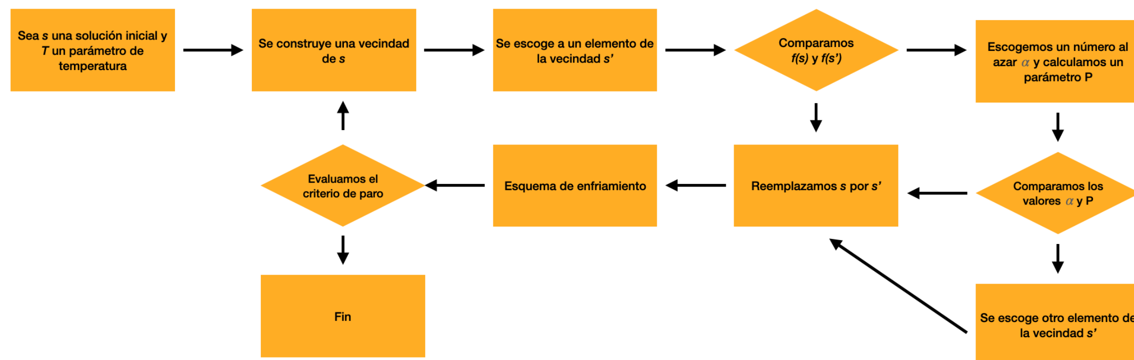

Figura 4.4: Diagrama de Recocido Simulado.

En la Función A.16 que aparece en el Apéndice A implementamos el proceso de Recocido Simulado.

4.2.4. GRASP

El acrónimo *GRASP* proviene de las siglas *Greedy Randomized Adaptive Search Procedure* que podría traducirse como: procedimiento de búsqueda glotón, aleatorizado y adaptativo. Fue propuesto por primera vez en [22]. Esta metaheurística tiene dos fases: construcción de una solución y mejora de la solución.

La fase de construcción se caracteriza por ser dinámica y aleatoria. La construcción de una solución se hace a través de un algoritmo que escoge aleatoriamente a un elemento de una lista de los mejores candidatos. En cada iteración, la lista de candidatos cambia y pueden cambiar los criterios para pertenecer a la lista de candidatos.

La longitud de la lista de candidatos tiene un papel fundamental en la naturaleza del proceso de búsqueda. Si la lista de candidatos permitiera sólo un elemento, entonces no habría una selección aleatoria y, por lo tanto, este proceso de búsqueda sería similar al de una búsqueda local básica. Por otro lado, si la lista de candidatos fuera muy grande, entonces la selección sería completamente aleatoria.

La segunda etapa es un proceso de búsqueda local. Esta etapa es una de las principales áreas de oportunidad de esta metaheurística ya que este proceso podría ser sustituido por otra heurística diferente al proceso de búsqueda local.

En la primera etapa se construye una solución, si esta solución no es factible se aplica un procedimiento de reparación hasta que ésta resulte factible. Por último, en la etapa de búsqueda local se mejora la solución examinando una de sus vecindades.

La aleatoriedad juega un papel muy importante en el diseño de algoritmos. GRASP y algoritmos genéticos dependen de la aleatoriedad del espacio de búsqueda. La aleatoriedad también se utiliza para romper empates, seguir diferentes trayectorias desde una misma solución inicial en un método multiarranque o tomar muestras de diferentes partes de una misma vecindad. Uno de los usos más importantes de algoritmos glotones aleatorios aparece en el contexto de GRASP.

Los algoritmos glotones aleatorios se basan en el mismo principio de construcción de soluciones guiado por los algoritmos glotones, con una diferencia. Los algoritmos glotones aleatorios construyen diferentes soluciones en diferentes corridas. Podemos describir un algoritmo glotón aleatorio de la siguiente manera. Los algoritmos glotones, en cada iteración, el conjunto de candidatos es cada elemento de E tal que al incorporarlo a la solución actual el resultado es una solución factible. Cada uno de los candidatos es evaluado a través de una función glotona que mida su costo incremental. A partir de la información que se obtiene de las evaluaciones, se construye una lista restrictiva de candidatos (RCL) formada por los *mejores* elementos de la lista de candidatos con respecto al aumento en el costo incremental. Seleccionamos un elemento de la lista restrictiva de candidatos aleatoriamente y actualizamos la lista de candidatos. Este algoritmo es glotón porque sólo toma en cuenta las mejores soluciones en cada iteración, pero es aleatorio por escoger un elemento al azar de los mejores candidatos.

El uso de listas restringidas de candidatos es esencial para reducir el número de soluciones examinadas en cada iteración cuando las vecindades sean muy grandes o las soluciones sean muy costosas de evaluar. El objetivo de estas listas es restringir las *regiones* de la vecindad que contienen características deseables para inspeccionarlas más de cerca. La eficiencia de la lista de candidatos se puede garantizar a través de estructuras de memoria que se puedan actualizar eficientemente. La eficiencia de la lista de candidatos se evalúa en términos de la mejor solución encontrada.

Se puede construir una lista restringida de candidatos en función de la cardinalidad que le permitamos tener a la lista. Consideramos c^{min} como el mínimo costo incremental de la lista de candidatos, y a c^{max} como el máximo costo incremental de la lista de candidatos. De acuerdo a un parámetro $\alpha \in [0, 1]$ podemos decidir cuántos elementos de la lista de

candidatos escoger según el conjunto $\{e \in C \mid c(e) \geq c^{min} + \alpha(c^{max} - c^{min})\}$. Si $\alpha = 0$, entonces este conjunto es igual a la lista de candidatos por lo que el procedimiento es completamente aleatorio si por otro lado $\alpha = 1$ entonces este procedimiento es completamente glotón en el sentido en el que la lista de restringida de candidatos es el conjunto de los candidatos que alcanzan el máximo en el costo incremental.

En este último pseudocódigo representamos al proceso de GRASP. Para hacer esto nos referimos a "Construcción glotona aleatoria", ésta podría ser la que representamos en el algoritmo anterior aunque las variantes de GRASP podrían utilizar otro proceso construcción glotona aleatoria.

Algoritmo 5 GRASP

```
Require: MaxIteraciones, α, seed
  f* ← ∞
  for k = 1, ..., MaxIteraciones do
      S ← ConstrucciónGlotonaAleatoria(α, seed)
      if S no es factible then
          S ← RepararSolución(S)
      end if
      S ←BúsquedaLocal(S)
      if f(S) < f* then
          S* ← S
          f* ← f(S)
      end if
  end for
```

En el siguiente diagrama de flujo representamos el proceso de GRASP.

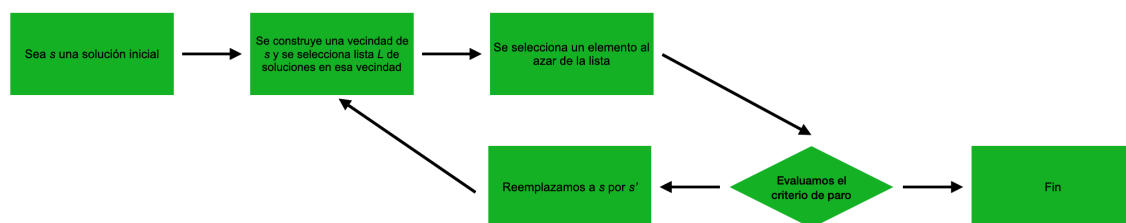

Figura 4.5: Diagrama de GRASP.

En la Función A.17 implementamos una función glotona para evaluar instancias del problema del SAT en soluciones parciales. Con la Función A.18 encontramos los candidatos para construir soluciones parciales. Con la Función A.19 garantizamos que la solución construida sea una solución factible. La Función A.20 nos permite hacer una construcción aleatoria. La Función A.21 construye la lista restringida de candidatos. La Función A.22 nos permite hacer un proceso de construción glotón aleatorio. Y por último, la implementación de GRASP es la Función A.23.

4.2.5. Búsqueda por Vecindades Variables

Esta metaheurística fue propuesta en [29] y su filosofía es cambiar dinámicamente la estructura de las vecindades. Este algoritmo es muy general y tiene varios grados de libertad

para diseñar muchas variantes para diferentes problemas e instancias.

En el paso de inicialización se define un conjunto de estructuras de vecindades. Estas se pueden definir de cualquier manera, pero usualmente se define una sucesión de vecindades de cardinalidad creciente. Luego, se genera una solución inicial y el algoritmo itera hasta un cierto criterio de paro.

El ciclo del algoritmo está compuesto por tres etapas: sacudida, búsqueda local y movimiento. En la etapa de sacudida se escoge a una solución vecina en la k-ésima vecindad de la solución actual. Esa solución es el punto de partida de una búsqueda local usando cualquier estructura de vecindad y de acuerdo con algún criterio de paro. Si la solución encontrada en la búsqueda local es mejor que la solución inicial, entonces es reemplazada. Si no hubo mejora, entonces se repite el procedimiento con la solución inicial dada pero cambiando el valor de k.

El objetivo de la etapa de sacudida es perturbar a la solución inicial de la búsqueda local, con esto se busca escapar del mínimo local inicial. Sin embargo, no es un proceso multi-arranque como vecindades variables, la perturbación no resulta en una solución *tan lejana* a la inicial porque pertenecen a la misma vecindad, para alguna estructura de vecindad. Si no hay mejora a través de este proceso, el cambiar el valor de k logra diversificar la búsqueda. El hecho de que las vecindades estén organizadas de acuerdo a su cardinalidad tiene como objetivo diversificar más la búsqueda. La filosofía detrás de esta metaheurística es que una misma solución puede tener una buena vecindad y una mala vecindad. Más aún, una solución podría ser un óptimo local en una vecindad y no serlo para otra vecindad de una misma solución. A partir de este metaheurística, se propone un proceso de búsqueda local de *mayor mejora* llamado: vecindades variables descendientes.

En el siguiente pseudocódigo representamos al proceso de búsqueda por vecindades variables. Para llevar a cabo este proceso debemos reconocer que hay tres funciones especiales que tenemos que definir previamente. La primera es una función que defina estructuras de vecindades, una que construya una solución inicial y una que realice el movimiento de la solución en la k-ésima vecindad. Una iteración en este proceso recorre hasta k_{max} estructuras de vecindades, con reinicios cada vez que se mejora el rendimiento de la solución encontrada.

Algoritmo 6 Búsqueda por Vecindades Variables

```
Require: Definir estructuras de vecindad N_0, ..., N_{k_max}
  S ← Solución inicial
  f* ← ∞
  while Criterio de paro no se cumpla do
    k ← 1
    while k ≤ k_max do
      S' ← Movimiento(S, k)
      S'' ←BúsquedaLocal(S')
      if f(S'') > f(S) then
        S ← S''
        k ← 1
      else
        k ← k + 1
      end if
    end while
  end while
```

En el siguiente diagrama de flujo representamos el proceso de búsqueda por vecindades variables.

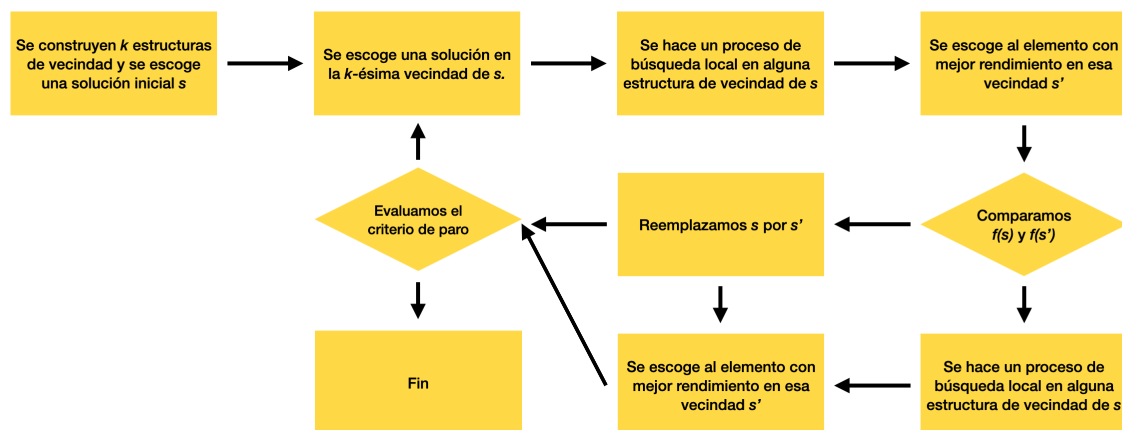

Figura 4.6: Diagrama de Búsqueda de Vecindad Variable.

Las siguientes funciones aparecen en el apéndice A: la Función A.24 provee a la heurística de una solución inicial, la Función A.25 nos permite perturbar soluciones dentro de una misma vecindad, y la Función A.26 es la implementación de la heurística de vecindades variables.

4.3. Metaheurísticas basadas en poblaciones

La metaheurísticas basadas en poblaciones lidian en cada iteración con un conjunto de soluciones en vez de con una sola solución. El desempeño de estas metodologías de búsqueda está determinado por la forma en la que se manipula a la población. Vamos a hablar de dos estrategias que engloban varias metaheurísticas y que son los métodos más estudiados en el contexto de metaheurísticas basadas en poblaciones: algoritmos genéticos y optimización por colonias de hormigas.

4.3.1. Algoritmos Genéticos

La computación evolutiva está inspirada en la forma en la que los seres vivos naturalmente evolucionan para adaptarse a su entorno. Los algoritmos en computación evolutiva se pueden caracterizar como modelos de procesos evolutivos. La población en cada iteración es una generación de la población, es decir en cada iteración se manipula a la población para que el resultado sea la descendencia de la población anterior. En el cada iteración, la población pasa por un ciclo de operaciones que pueden ser operaciones de recombinación, mutación o selección. De manera que el resultado de la iteración es la población mejor adaptada.

En cada iteración, la población se combina y muta para generar una descendencia. De esta descendencia se escoge a los más aptos, con ellos y quizás con elementos de la población se construye una nueva población. A continuación haremos un pequeño comentario de cada una de estas etapas.

Descripción de los individuos Los individuos de la población en un algoritmo genético no tienen que ser necesariamente soluciones del espacio de búsqueda, pueden ser soluciones parciales o subconjuntos de soluciones. Comúnmente los individuos de las poblaciones se representan como vectores de ceros y unos o permutaciones de números enteros. En el contexto de los algoritmos genéticos a los individuos se les llaman *genotipos* y las soluciones que son codificadas por los individuos se les llaman *fenotipos*. Determinar la descripción adecuada de los individuos es crucial para el éxito de esta heurística.

Tamaño de la población En cada iteración se reemplaza a la población, se puede escoger entre mantener algunos individuos de la población anterior o que la nueva población sea exclusivamente de nuevos individuos. El tamaño de la población puede ser fijo o dinámico aunque en general el tamaño de las poblaciones es fijo. Si es dinámico, una opción podría ser que la población se fuera reduciendo su tamaño hasta que sólo sobreviva un individuo.

Fuentes de información La principal fuente de información para crear descendencia es la combinación de dos individuos (o padres), sin embargo hay varias estrategias en las que se pueden combinar a más elementos.

Intensificación y diversificación El hecho de usar conjuntos de soluciones o conjuntos de componentes de soluciones garantiza ampliar las regiones de búsqueda de esta metaheurística. Sin embargo, podemos intensificar esta búsqueda utilizando estrategias de búsqueda local. El proceso de mutación en los algoritmos genéticos es esencial para diversificar la búsqueda.

Los algortimos genéticos son metodologías de búsqueda basados en los principios de selección natural. En esta heurística, como en las demás metaheurísticas de población hay una terminología asociada que permite hacer las analogías con los principios de selección natural más claras.

En algoritmos genéticos, las variables de decisión del problema se codifican como secuencias de longitud finita de un cierto alfabeto. Las cadenas que se refieren a soluciones del problema les llamamos **cromosomas**, a los elementos del alfabeto les llamamos **genes** y a los valores de los genes les llamamos **alelos**. Por ejemplo, en el problema del SAT un cromosoma es una solución del espacio de búsqueda y un gen es cada una de las coordenadas de la solución. En general, los algoritmos genéticos codifican a los parámetros en vez de usar a las mismas soluciones. Para evolucionar a mejores soluciones, es necesaria una **función objetivo** que distinga a las buenas soluciones de las malas. Uno de los conceptos

clave de los algoritmos genéticos es el de **población** que consiste un conjunto de soluciones candidatos. El tamaño de la población es uno de los parámetros que decide el usuario y es decisivo en el desempeño de los algoritmos genéticos. Una población muy pequeña conduce a una convergencia prematura y una población muy grande tiene un costo computacional significativo.

Una vez que se ha codificado el problema de una manera adecuada, el proceso de evolución de las soluciones sigue el siguiente procedimiento.

1. **Inicialización:** Se crea una población inicial aleatoriamente a lo largo de todo el espacio de búsqueda.
2. **Evaluación:** Se evalúan los valores en la función objetivo de cada individuo en la población.
3. **Selección:** Este paso implementa el mecanismo de la *supervivencia del más apto*. Hay varias estrategias para escoger a los *mejores* individuos de la población de acuerdo con su valor en la función objetivo. Algunas de estas estrategias son: selección por medio de la ruleta, selección universal estocástica, selección por torneo o selección por desempeño.
4. **Recombinación:** Este paso implementa la creación de descendencia para crear, posiblemente, mejores soluciones. Hay muchas maneras de lograr esto. Esencialmente es buscar estrategias para combinar los cromosomas de dos individuos en la población para obtener nuevo individuo.
5. **Mutación:** Este paso implementa la mutación genética que ocurre aleatoriamente en la descendencia. La mutación altera localmente y aleatoriamente a los cromosomas que se generaron como descendencia de individuos de la población.La mutación hace uno o varios cambios en los individuos que se obtuvieron por el proceso de recombinación.
6. **Reemplazo:** Se modifica a la población del algoritmo. Este paso puede ser implementado de varias maneras. Se puede reemplazar a toda la población por la descendencia o se puede sólo reemplazar a algunos. Algunas estrategias pueden ser: reemplazo elitista, reemplazo por la sabiduría de la generación o reemplazo por estados estables.

De acuerdo con Goldberg [27] individualmente cada una de las operaciones listadas anteriormente son ineficientes, pero al combinarlas tienen un buen desempeño. A continuación vamos a describir algunos métodos en de las operaciones básicas de los algoritmos genéticos. No hablaremos de los métodos de inicialización dados que estos de no ser aleatorios, se obtendrían por otra heurística.

En el siguiente pseudocódigo representamos el proceso de algoritmos genéticos. La siguiente representación es la versión más general de los algoritmos genéticos y como hemos discutido anteriormente hay muchas variantes para cada etapa de este proceso.

Algoritmo 7 Algoritmos genéticos

Require:
  Inicializar población P
  Evaluar a la población
  while Criterio de paro no se cumpla **do**
    Seleccionar a los individuos más aptos para ser cruzados
    Cruzar a los individuos más aptos
    Mutar a los descendientes
    Actualizar a la población
    Evaluar a la población
  end while

La siguiente Figura representa el proceso de algoritmos genéticos en un diagrama de flujo.

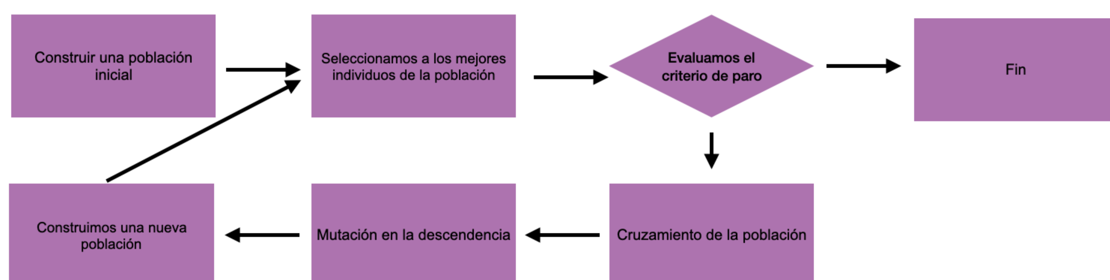

Figura 4.7: Diagrama de Algoritmos Genéticos.

En nuestra implementación utilizamos el método de la ruleta para el proceso de selección, el método de recombinación por un punto, para el método de mutación usamos una probabilidad de mutar de 0.2 en un solo punto y para el método de reemplazo intercambiamos a la peor mitad, con respecto a la función objetivo, de la población inicial por la descendencia.

Las siguientes funciones están representadas en el apéndice A: Para la inicializar a la población, usamos la Función A.27. Para seleccionar a los mejores individuos de la población, usamos la Función A.28. Para combinar a los elementos de la población, usamos la Función A.29. La función que nos permite mutar individuos es A.31. La función para reemplazar a los individuos de la población es A.32. Por último, la implementación de algoritmos genéticos es la Función A.33.

4.3.2. Optimización por Colonias de Hormigas

La metaheurística de optimización por colonias de hormigas fue propuesta por [17]. La inspiración de esta metaheurística son las colonias de hormigas reales, su comportamiento les permite encontrar un camino corto entre su fuente de alimento y su nido. Mientras las hormigas caminan entre su nido y la fuente de comida depositan una sustancia en el piso llamada feromona. Cuando una hormiga decide qué camino tomar escoge con mayor probabilidad un camino que ha sido marcado con mayores concentraciones de feromonas. Este fenómeno es la base de un comportamiento cooperativo que resulta en el surgimiento de caminos más cortos.

Los algoritmos de optimización por colonias de hormigas se basan en modelos probabilísticos parametrizados. Esta metaheurística es constructiva, las hormigas artificiales hacen

recorridos aleatorios (*random walks*) por una gráfica conexa cuyos vértices son componentes de soluciones y las aristas son las conexiones entre esas componentes, a esta gráfica se le llama gráfica de construcción. Las restricciones del problema se traducen en restricciones sobre los recorridos que las hormigas pueden hacer, es decir sólo pueden atravesar una arista entre dos componentes de solución solamente si al juntar esos dos componentes la solución es factible.

A cada componente de solución (vértice) y a cada arista se les puede asociar un parámetro que le corresponde al rastro de feromonas, T_i para el i-ésimo vértice y T_{ij} para la arista que una al i-ésimo y al j-ésimo. Además cada vértice y cada arista tiene un valor predeterminado η_i y η_{ij} respectivamente. Estos valores son usados por las hormigas para tomar decisiones probabilísticas sobre cómo moverse a lo largo de la gráfica.

A la primera implementación de esta heurística se le conoce como Sistema de Hormigas. En ésta, primero se inicializan los valores de feromonas y los valores heurísticos de la gráfica de construcción. Cada hormiga hace un recorrido en la gráfica de construcción de manera que la probabilidad que vaya de una componente a otro está dada por una probabilidad proporcional a la concentración de feromonas y de valores heurísticos. Después de que todas las hormigas han hecho su recorrido, se actualizan los valores de feromonas de cada componente y de cada arista, de manera proporcional al número de recorridos que las hormigas hicieron por esa componente o por esa arista, además es proporcional a un factor de *evaporación* menor que uno que hace que la concentración disminuya en caso de haber sido visitada.

Una implementación más general, a la que propiamente se le llama Optimización por Colonias de Hormigas, consiste esencialmente en el mismo procedimiento que el anterior con dos principales diferencias. La primera es que se implementa un esquema de actividades en el que la actualización de los valores de las feromonas puede ser dinámico, en vez de esperar a que todas las hormigas pasen y luego actualizar los valores. La otra diferencia consiste en implementar *actividades demonio* en el esquema de actividades. Estas actividades realizan acciones que no pueden ser ejecutadas por hormigas, por ejemplo búsquedas locales que cambien el valor de las feromonas.

Algoritmo 8 Optimización por Colonias de Hormigas

```
Inicializar los valores de las feromonas.
while Criterio de paro no se cumpla do
    for i = 1,...,k do
        La i-ésima hormiga construye una solución siguiendo el camino de las feromonas.
    end for
    Actualizar los valores de las feromonas.
    Evaporar la traza de las feromonas.
end while
```

En la siguiente Figura representamos el proceso de optimización por colonias de hormigas.

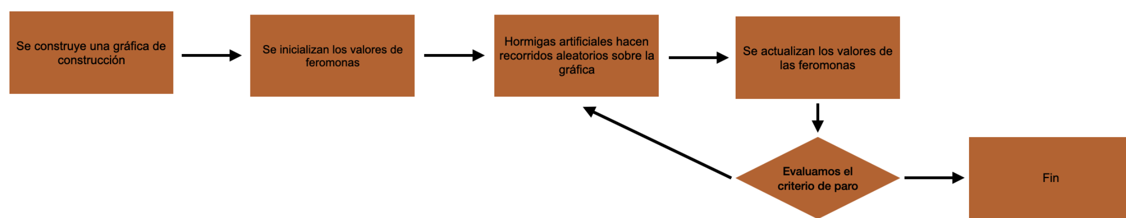

Figura 4.8: Diagrama de Sistemas de Hormigas.

Las funciones que implementamos aparecen representadas en el Apéndice A. Para inicializar la matriz de valores de feromonas usamos la Función A.34. Para inicializar la matriz de valores de probabilidades de transiciones usamos la Función A.35. Para que las hormigas escojan una siguiente solución usamos la Función A.36 Para intensificar la matriz de feromonas y para actualizar la matriz de probabilidades usamos A.37 y A.38 respectivamente. Por último, la implementación de la heurística de optimización por colonias de hormigas es la Función A.39.

Capítulo 5

Heurísticas específicas para el problema del SAT

En las últimas décadas ha habido un enorme progreso en la eficiencia de las heurísticas que resuelven al problema del SAT o también llamadas *SAT-solvers*. Desde el 2001 se celebra anualmente la competencia del problema del SAT que han llevado al desarrollo e inovación de docenas de implementaciones de SAT-solvers y han creado una extensa variedad de instancias del problema del SAT que provienen del mundo real. Actualmente estas heurísticas pueden resolver instancias de millones de variables y millones de cláusulas. Representar un problema combinatorio como un problema del SAT hace que aumente significativamente la longitud de la representación del problema. Sin embargo esto no representa una dificultad para las nuevas heurísticas. Recientemente se ha observado un fenómeno en el que es más sencillo resolver problemas combinatorios a través de sus codificaciones como problemas del SAT que como están instanciados originalmente. Los avances que se han logrado para las heurísticas del SAT dependen fuertemente de la implementación de estructuras eficientes que nos facilitan resolver el problema.

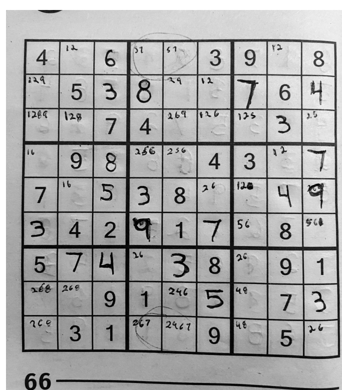

Figura 5.1: Sudoku a medio resolver [58]

En la Figura 5.1 se puede ver un ejemplo de un sudoku a medio resolver. En principio, en cada una de las casillas vacías podemos anotar cualquier número del 1 al 9. Sin embargo, al observar las restricciones que nos propone el juego y las restricciones que asumimos a partir de las decisiones que hemos tomado, entonces podemos descartar algunos números entre el 1 y el 9. La persona que estaba resolviendo este sudoku encontró útil escribir en número más pequeño en la parte superior qué valores podría anotar de acuerdo con las restricciones del problema y las decisiones que ha asumido. Anotar en la parte superior de las casillas vacías los posibles dígitos es una metodología eficiente que nos facilita resolver

este problema.

De la misma manera, podemos pensar en estructuras eficientes que nos permitan resolver instancias del problema de SAT. Esta estructuras surgieron a partir de dos algoritmos exactos que se desarrollaron a mediados del s. XX: el algoritmo de *Davis–Putnam–Logemann–Loveland* (1962) [13] y el algoritmo *Conflict-driven clause-learning* (1999) [38].

En este capítulo vamos a hacer una revisión hemerográfica de las heurísticas que constituyen el estado del arte de los solucionadores del problema de satisfacibilidad booleana.

5.1. Algoritmo Davis-Putnam-Loveland-Logeman

A finales de los años cincuenta, los matemáticos estadounidenses Martin Davis de la universidad de Princeton y Hilary Putnam del MIT diseñaron un algoritmo completo y correcto que resuelve el problema de satisfacibilidad booleana. El algoritmo de Davis-Putnam se basa en el teorema de Herbrand para lógica proposicional de primer orden. Un año más tarde, los matemáticos George Logeman y Donald Loveland propusieron una mejora para este algoritmo exacto.

La propuesta es un algoritmo constructivo que utiliza *vuelta hacia atrás* en español. En cada iteración se deciden valores de verdad para las variables, cuando se encuentra una contradicción el algoritmo *regresa* a la última decisión de asignación de valor de verdad y la cambia. Si, sin importar el valor de verdad de una variable, la fórmula no puede ser verdadera entonces la instancia no es satisfacible.

Además del proceso de asignación de valores de verdad a variables y de vuelta hacia atrás el algoritmo DPLL se basa en una estrategia revolucionaria en el contexto de heurísticas para el problema del SAT que resulta bastante natural: propagación unitaria de restricciones.

De la misma manera que cuando anotamos un número en el sudoku, ya no podemos anotar ese mismo número en ninguna fila o columna de esa casilla; cuando asignamos un valor de verdad a una variable eso tiene consecuencias sobre los valores de verdad, a este proceso le llamamos propagación de restricciones. En algunos casos, la propagación de restricciones puede implicar valores de verdad sobre otras variables. Sin embargo, el proceso de propagación unitaria de restricciones no asigna valores de verdad a otras variables.

En el siguiente pseudocódigo representamos al algoritmo DPLL. Este algoritmo es recursivo. Sus entradas son un conjunto de cláusulas C y una asignación ρ (parcial o no) de valores para las variables. Para establecer este procedimiento necesitamos de la siguiente definición.

Definición 5.1.1. *Para un conjunto de cláusulas* $C = \{[x_1, x_2, x_3], ..., [x_n, x_{n+1}, x_{n+2}]\}$ *y una asignación de valores* $\rho = \{x_i \leftarrow 1, ..., x_j \leftarrow 0\}$ *definimos* $C|\rho$ *como el conjunto de cláusulas que se obtiene cuando se reemplazan a los valores de verdad de* ρ *en* F*.*

- *Si la cláusula* $[x_r, ..., x_s] \in C$ *es verdadera gracias a la asignación* ρ*, entonces en* $C|\rho$ *eliminamos dicha cláusula.*
- *Si la cláusula* $[x_r, ..., x_s] \in C$ *no es verdadera y la variable* $\neg x_r \leftarrow T$ *aparece en* ρ *entonces se borra la variable* x_r *de dicha cláusula.*

El símbolo $C|\rho$ representa formalmente el resultado de evaluar a C en ρ. Si $C|\rho$ es un conjunto vacío entonces C es satisfacible bajo la asignación ρ. Si $C|\rho$ contiene una cláusula

vacía entonces la asignación no es satisfacible.

Algoritmo 9 DPLL recursivo

Require: (C, ρ)
  $(C, \rho) \leftarrow$ PropagaciónUnitara(C, ρ)
  if C contiene una cláusula vacía **then**
    C es UNSAT
  end if
  if $C = \emptyset$ **then**
    C es SAT bajo ρ
  end if
  $\ell \leftarrow$ una variable que no aparezca en ρ
  if DPLL$(C|\{\ell\}, \rho \cup \{\ell\})$=SAT **then**
    C es SAT bajo ρ
  else
    Ejecutar DPLL$(C|\{\neg\ell\}, \rho \cup \{\neg\ell\})$
  end if

Utilizar esta versión recursiva de $DPLL$ es más eficiente que usar una versión iterativa pues ésta implicaría recorrer cada una de las ramas del árbol de posibilidades. El proceso iterativo requiere que se hagan más operaciones y el proceso recursivo requiere más memoria.

A continuación vamos a desarrollar un ejemplo de resolver una instancia del problema del SAT usando el algoritmo de DPLL. Para ilustrar este algoritmo vamos a usar The SAT Game [50]. Las primeras cláusulas de la instancia son las siguientes.

$$
\begin{array}{llll}
(\neg x_3 \vee \neg x_5 \vee x_2) \wedge & (x_7 \vee x_9 \vee \neg x_6) \wedge & (\neg x_2 \vee x_3 \vee \neg x_4) \wedge \\
(\neg x_6 \vee x_5 \vee x_8) \wedge & (\neg x_9 \vee \neg x_5 \vee x_8) \wedge & (x_5 \vee x_4 \vee x_3) \wedge \\
(\neg x_6 \vee \neg x_2 \vee x_7) \wedge & (\neg x_7 \vee \neg x_6 \vee x_2) \wedge & (\neg x_3 \vee \neg x_5 \vee x_2) \wedge \\
(x_7 \vee \neg x_5 \vee x_2) \wedge & (\neg x_3 \vee \neg x_5 \vee x_2) \wedge & (\neg x_2 \vee \neg x_4 \vee x_7)
\end{array}
$$

En The SAT Game, estas cláusulas están representadas como en la Figura 5.2. En esta representación cada cláusula es una fila de longitud 3 y cada variable está representada por el entero que la enumera, los enteros son negativos si la variable está negada. La instancia aquí representada tiene 12 cláusulas.

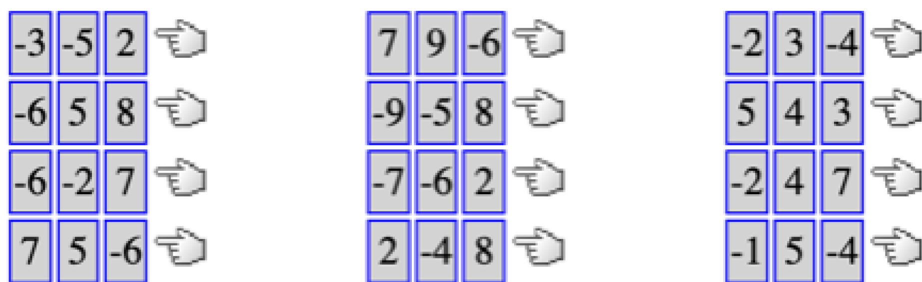

Figura 5.2: [50]

La instancia del problema que vamos a usar está representada en la Figura 5.3.Tiene 60 cláusulas y 9 variables.

-3	-5	2	7	9	-6	-2	3	-4
-6	5	8	-9	-5	8	5	4	3
-6	-2	7	-7	-6	2	-2	4	7
7	5	-6	2	-4	8	-1	5	-4
2	-9	7	1	5	-9	3	-6	-9
3	8	-6	7	-2	-4	-9	-2	5
-6	4	3	5	7	-3	7	2	-1
-9	2	1	9	2	3	-8	-7	-2
-2	-5	6	3	-2	-9	-6	3	-4
4	2	8	-2	-1	-6	-9	-6	-4
-9	-8	-2	5	-1	-2	2	-6	4
-4	7	-8	3	8	-4	8	7	-3
9	4	1	-9	-3	-2	-1	-9	-7
-9	-5	-7	-9	-7	-4	1	-9	-4
-3	-4	7	-8	-1	-3	-6	-2	5
-8	1	-6	8	2	3	-5	-9	3
-9	-2	7	-7	-3	5	7	-6	5
-2	-7	3	-5	-2	6	2	-5	1
-5	-8	-3	2	-6	7	-9	-2	-6
-8	-3	1	4	-8	5	6	-9	4

Figura 5.3: Instancia del problema del SAT [50]

En el primer paso, como podemos ver en la en la Figura 5.4, escogemos que la primera variable de la primera cláusula sea verdadera es decir $\neg x_3 \rightarrow T$ y aplicamos la propagación unitaria de restricciones. Es decir todas las cláusulas que en las que aparece $\neg x_3$ han sido satisfechas.

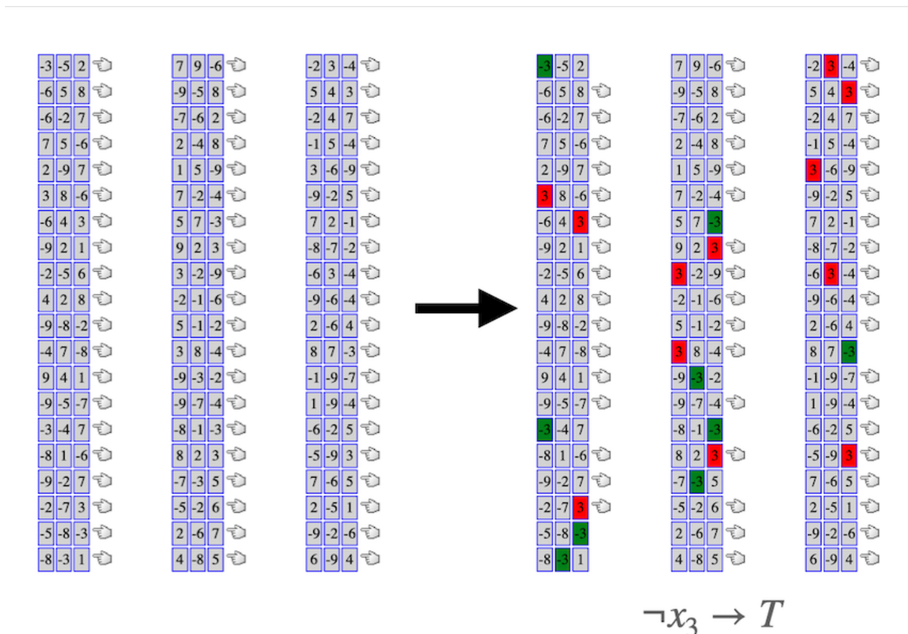

Figura 5.4: Primera iteración del DPLL.

En la Figura 5.4 las celdas que corresponden a la variable x_3 son rojas y las que corresponden a la variable $\neg x_3$ son verdes. Las cláusulas que tienen una celda verde ya no tienen una manita del lado derecho, porque esas cláusulas ya están satisfechas.

En los siguientes dos pasos hacemos lo mismo con las primeras variables de las siguientes dos cláusulas. Hasta ahora no hay ningún conflicto.

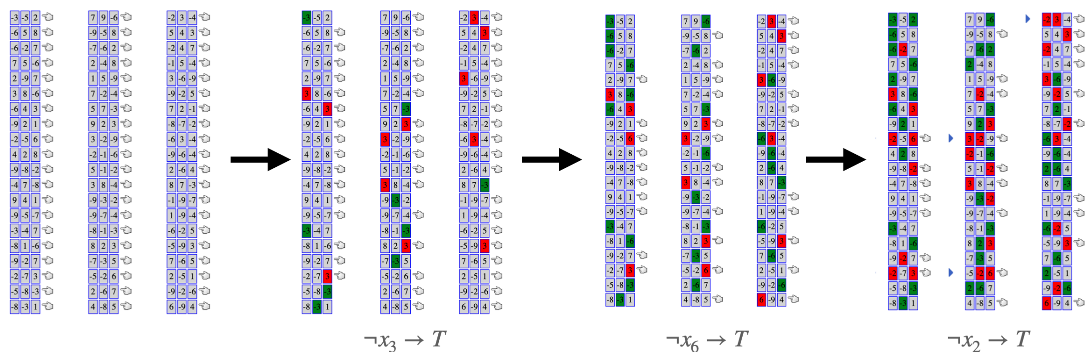

Figura 5.5: Primeras cuatro iteraciones del algoritmo DPLL.

En las siguientes tres iteraciones se hace lo mismo, es decir se asignan valores de verdad a las siguientes tres variables que aparecen en cláusulas no satisfechas. En el penúltimo paso de la Figura **??**, al asignarle un valor a x_4 encontramos que una de las cláusulas de la fórmula ya no se puede satisfacer, es la primer cláusula de la tercera columna. Utilizando el algoritmo de vuelta hacia atrás, recurrimos a la última asignación y cambiamos el valor de dicha variable. En el último paso de esta Figura en vez de asignar $x_4 \to T$, asignamos $\neg x_4 \to T$. Tras hacer la propagación unitaria de restricciones podemos ver que aquí también hay un conflicto. Por lo tanto, tenemos que volver a hacer bakctracking pero ahora al paso anterior al de asignarle valor a x_4.

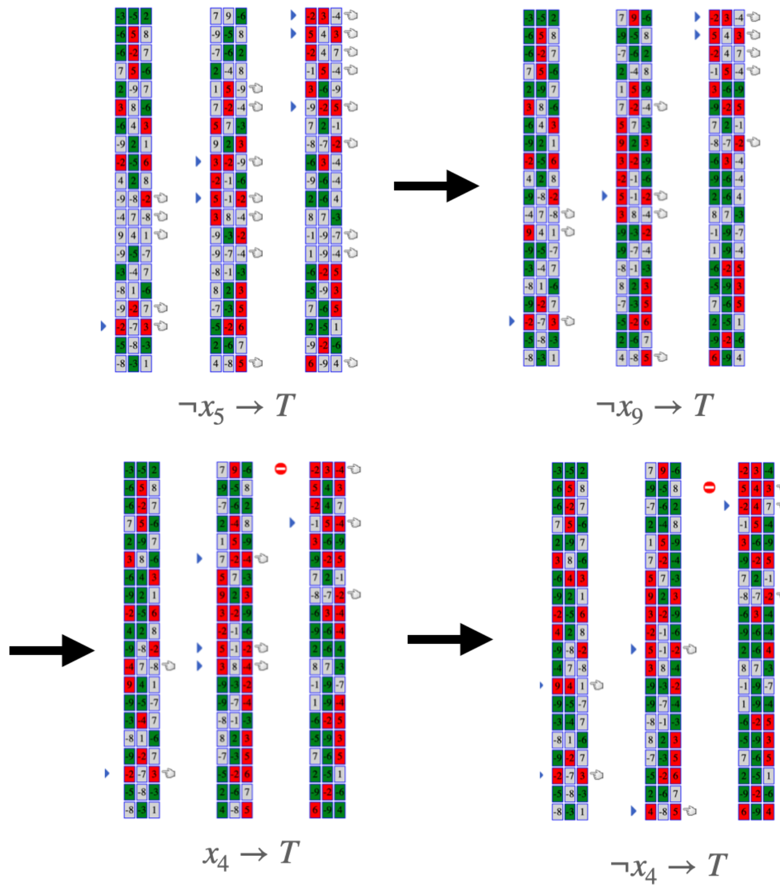

Figura 5.6: Siguientes cuatro iteraciones del algoritmo DPLL.

El paso que tenemos que seguir de acuerdo con el algoritmo de vuelta hacia atrás es cambiar el valor de verdad de la variable x_9. Cuando hacemos propagación unitaria de restricciones vemos que esto también conlleva un conflicto. Por lo tanto, el paso que tenemos que seguir de acuerdo con el algoritmo de vuelta hacia atrás es cambiar el valor de verdad de la variable x_5. Si seguimos este algoritmo, eventualmente encontraremos la solución final a la instancia, así como aparece en el último paso de la Figura 5.7. Como se puede ver en la solución, el algoritmo tiene que hacer un backtracing hasta cambiar el valor de la variable x_6, que fue la segunda decisión que hicimos desde el principio.

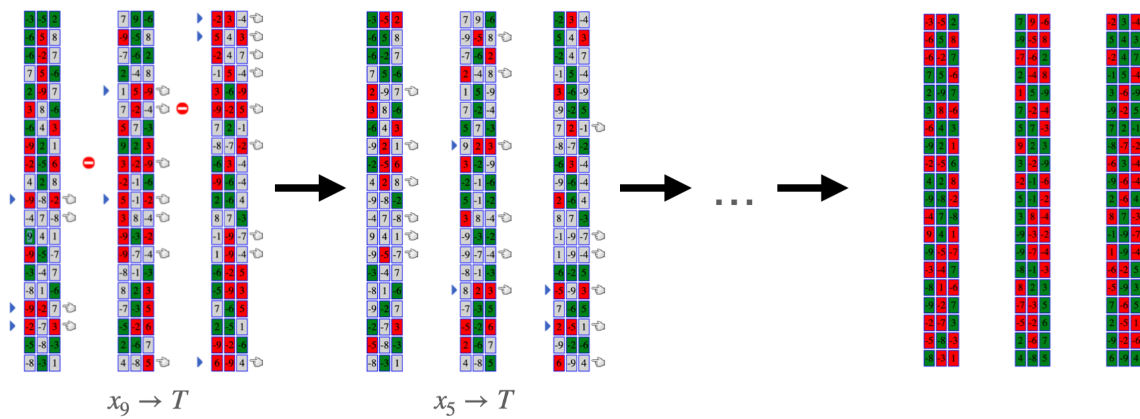

Figura 5.7: Siguientes tres iteraciones del algoritmo DPLL.

5.1.1. Aprendizaje de cláusulas

En el siguiente pseudocódigo representamos una de las estructuras de más alto nivel en el que se apoyan las SAT-solvers modernos: el aprendizaje de cláusulas. La siguiente versión del algoritmo DPLL es en su versión iterativa, que también es la versión más similar a la que utilizan los SAT-solvers actuales.

Para llevar a cabo este proceso tenemos que tener definidos los siguientes sub-procesos.

- *Decidir la siguiente rama:* En este proceso se decide la siguiente variable sobre la que se va a ramificar el procedimiento. Es decir, se decide a qué variable se le va a asignar un valor de verdad.
- *Deducción:* En este proceso se aplica propagación unitaria de restricciones. Si alguna cláusula queda vacía, se establece un estado de conflicto. Si el conjunto de cláusulas es vacío entonces se declara que la fórmula es satisfacible.
- *Análisis de conflictos:* Este procedimiento utiliza la estructura de implicaciones para determinar cuáles son los conflictos para calcular una *cláusula conflictiva.* En este proceso también se calcula el momento al que hay que regresar para aplicar el algoritmo de vuelta hacia atrás.

Algoritmo 10 DPLL iterativo con aprendizaje de cláusulas

```
Require: Fórmula CNF
  while TRUE do
     Decidir la siguiente rama
     estado ← Deducción
     if estado = CONFLICTO then
        nivel ← AnálisisDeConflicto
     else
        if estado = SAT then
           SAT
           break
        end if
     end if
  end while
```

En términos de notación, las variables a las que se le asignan valores de verdad a través del proceso de *Decidir la siguiente rama* se les llaman **variables de decisión**. Las variables a las que se les asignan valores de verdad a través del proceso de *Deducción* se les llaman **variables de implicación**. A lo largo del proceso de vuelta hacia atrás la última variable de decisión se convierte en una variable de implicación, su valor de verdad cambia de acuerdo con las claúsulas aprendidas. El **nivel de decisión** de una variable de decisión es el número de variables de decisión en la ramificación de dicha variable. El **nivel de implicación** de una variable de implicación y es el máximo de los valores de los niveles de decisión de las variables de decisión que implicaron a la y. Si la variable y fue implicada sin usar alguna variable de decisión entonces su nivel de implicación es cero. Por último, el **nivel de decisión de un paso del proceso DPLL** es el máximo nivel de decisión de todas las variables de decisión de ese nivel.

El aprendizaje de cláusulas se inspira en el trabajo de inteligencia artificial para encontrar explicaciones a los retornos que tienen lugar en el proceso de vuelta hacia atrás, estas explicaciones se añaden como cláusulas (o más generalmente como restricciones) al problema original. Utilizando estrategias de aprendizaje de cláusulas, el algoritmo DPLL sigue siendo un método exacto cuya dificultad es el enorme número de cláusulas que se agregan a la fórmula original. Este problema reduce la practicidad de este algoritmo. El éxito de las heurísticas basadas en esta estrategia dependen de explotar estructuras eficientes que manejen las cláusulas aprendidas. A lo largo de una gran variedad de artículos se ha mostrado que existen problemas que no se pueden resolver sin usar la estrategia de aprendizaje de cláusulas.

En general, el proceso de aprendizaje oculto detrás del análisis de conflicto se espera que nos proteja de hacer el mismo cálculo más de una vez. Hay muchas variantes del proceso de aprendizaje de cláusulas derivadas de conflictos, también llamados esquemas de aprendizaje, por ejemplo que se construyan varias cláusulas a partir de un mismo conflicto. A continuación vamos a describir con más detalle este proceso usando unas estructuras muy populares entre los SAT-solvers modernos llamadas gráficas de implicación y de conflicto.

Gráficas de implicación y de conflicto

La propagación unitaria de restricciones se puede asociar naturalmente con una **gráfica de implicación** que captura todas las posibles manera de deducir las cláusulas de implicación a partir de las cláusulas de decisión. En lo que sigue, nos referiremos como **cláusulas conocidas** a las cláusulas que son entrada de este proceso de aprendizaje.

Definición 5.1.2. *La gráfica de implicación G en un dado paso del proceso DPLL es una gráfica dirigida y acíclica cuyos vértices están rotulados según subconjuntos de cláusulas. Esta gráfica se construye de la siguiente manera.*

1. ***Paso 1:*** *Crear un nodo para cada variable de decisión. Estos nodos tendrán grado de entrada cero.*

2. ***Paso 2:*** *Mientras exista una cláusula $C = \ell_1 \vee ...\ell_k \vee \ell$ tal que $\neg\ell_1, ..., \neg\ell_k \in G$, entonces se sigue el siguiente procedimiento:*

   a) *Si el nodo $\ell \notin G$ entonces se añade.*

   b) *Se añaden las siguientes aristas (ℓ_i, ℓ) para $1 \leq i \leq k$.*

   c) *Añadir un nodo cuyo nombre sea C a la gráfica.*

   d) *Añadir un nodo $\bar{\Lambda}$ a la gráfica. Este nodo se llama nodo de conflicto. Para variable x que aparezca como nodo en la gráfica de manera que $\neg x$ también aparezca en la gráfica entonces agregamos las aristas: $(x, \bar{\Lambda})$ y $(\neg x, \bar{\Lambda})$.*

Mientras que la gráfica de implicación puede o no tener conflictos, la gráfica de conflicto siempre tiene un conflicto. La decisión de *qué* es un conflicto también es parte de la estrategia del SAT-solver. Un manera para definir la gráfica sería tomar una subgráfica de la gráfica de implicación que cumpla con las propiedades descritas en el paso 2 de la definición anterior. A esta manera de definir la gráfica de conflicto se le llama subgráfica únicamente de inferencia.

Cláusulas de conflicto

Para un subconjunto U de vértices, el **corte** de U es el conjunto de aristas que salen de vértices en U a vértices que no están en U.

Si consideramos la gráfica de implicación en un paso en el que tenga un conflicto y fijamos la subgráfica de conflicto de esa gráfica de implicación, entonces podemos definir un corte de manera que todas las variables de decisión estén de un lado y el nodo $\bar{\Lambda}$ y al menos un nodo conflictivo estén del otro lado. Para construir una cláusula de conflicto tomamos la disyunción de las negaciones de los nodos iniciales de las aristas del corte y el nodo conflictivo. A continuación vamos a desarrollar un ejemplo.

Ejemplo de DPLL con aprendizaje de cláusulas

- *Decidir la siguiente rama:* Si no hay una variable de implicación para convertir en una variable de decisión, entonces de la lista de cláusulas, vamos a tomar la primer cláusula no satisfecha y hacer verdadera a alguna de las variables de esa cláusula.
- *Deducción:* En cada iteración, agregamos todas las cláusulas de implicación a la gráfica de implicación.
- *Análisis de conflictos:* El corte de la gráfica de implicación es el *más cercano* al nodo de conflicto y sólo se añade el último nodo de conflicto dentro del área conflictiva. El vuelta hacia atrás se hace hasta el paso más antiguo de los nodos de decisión involucrados en el corte.

En la Figura 5.8 vamos a representar los primeros tres pasos sobre una instancia del problema del SAT. En el primer paso, escogimos hacer falsa a la variable x_7, en el segundo hicimos falsa a la variable x_3 y en el tercero falsa a la variable x_5. Las primeras dos decisiones las hicimos porque no había variables de implicación a las que asignarles un valor de verdad y en la tercera decisión la hicimos porque x_5 es una variable de decisión que surgió en el paso anterior.

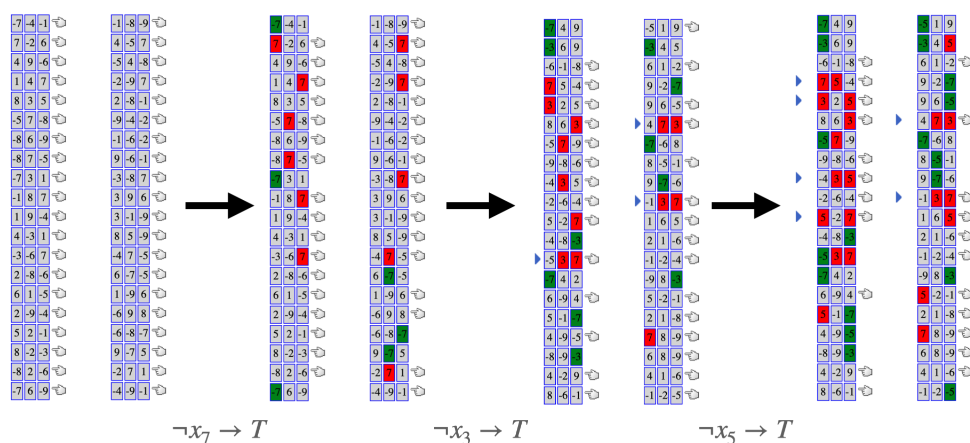

Figura 5.8: Primeras tres iteraciones del algoritmo DPLL con aprendizaje de cláusulas.

De sólo ver la instancia, podría parecer que para este punto no tenemos un conflicto. Sin embargo, si vemos la gráfica de implicación entonces podemos ver que sí tenemos un estado

de conflicto. Esto ya es una diferencia enorme con el algoritmo DPLL sin aprendizaje de cláusulas porque nos hemos ahorrado hacer varias iteraciones para llegar al conflicto.

En la primera iteración sólo aparece un nodo de variable de decisión, al igual que en la segunda iteración. Estos nodos los represento en la figura 5.9 con color morado.

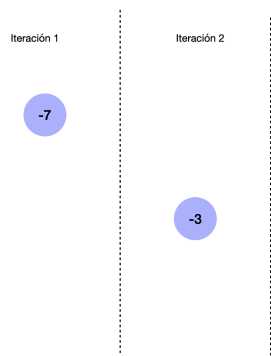

Figura 5.9: Primeras dos iteraciones de la construcción de la gráfica de implicación

En la tercera iteración obtenemos tres variables de implicación. Las flechas rojas son las implicaciones que surgieron de esa iteración. Los nodos los represento con color verde porque son nodos de implicación.

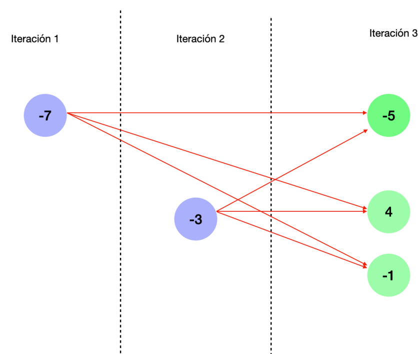

Figura 5.10: Representación de la primera parte de la tercera iteración de la construcción de la gráfica.

En esta misma iteración se escoge una variable de decisión. La siguiente variable de decisión que se toma es la primera variable de implicación que se dedujo, por eso en la iteración tres la variable $\neg x_5$ aparece en morado.

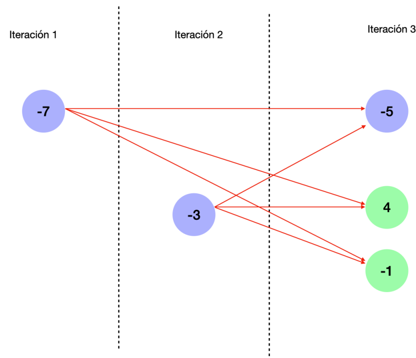

Figura 5.11: Representación de la segunda parte de la tercera iteración de la construcción de la gráfica.

En la siguiente iteración, obtuvimos otras tres variables de implicación. Las aristas de implicación de esta iteración las he colocado en color rojo.

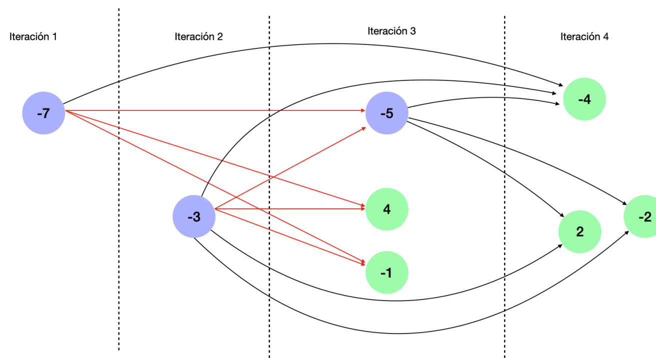

Figura 5.12: Cuarta iteración de la construcción de la gráfica.

En esta iteración obtenemos dos conflictos ya que aparecen los nodos de las variables x_2, $\neg x_2$, x_4 y $\neg x_4$.

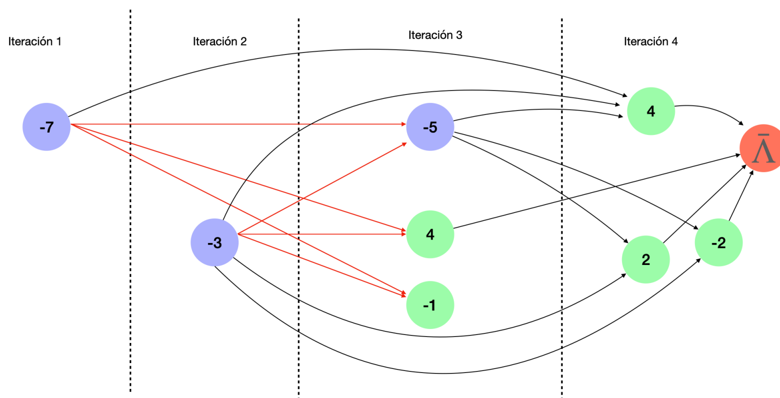

Figura 5.13: Gráfica de implicación para la cuarta iteración.

En la Figura 5.14 ejemplificamos un corte de conflicto. En esta Figura representamos de color rojo las aristas que implicaron el conflicto para poder rastrear a la variable de decisión involucrada más antigua. Como mínimo es esencial tener una variable de conflicto y el nodo

de conflicto de un lado. Los nodos de conflicto son: x_2, $\neg x_2$, x_4 y $\neg x_4$. Para el caso particular de la Figura 5.14 las aristas que llegan al nodo de conflicto seleccionado y que empiezan en un nodo de una variable de decisión son dos: $(\neg x_5, x_2)$ y $(\neg x_3, x_2)$. La cláusula que aprenderíamos a partir de este corte sería:

$$x_7 \vee \neg x_5 \vee x_3 \vee x_2.$$

El paso al que deberíamos de regresar de acuerdo con cómo definimos nuestro análisis de conflicto debería ser la iteración 1.

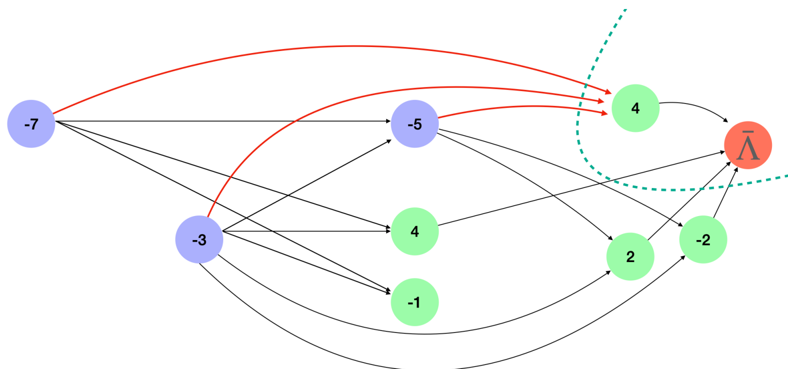

Figura 5.14: Ejemplo de corte para la gráfica de implicación.

En la Figura 5.15, representamos otro posible corte de conflicto, pero con el mismo esquema de aprendizaje. Es decir, la metodología con la que se construye el corte es la misma. Este esquema podría ser diferente, podría tener a todas las variables de conflicto, podría sólo tener a la última, podría incluir variables que no sean de conflicto, etc. En este caso, la cláusula que se aprendería sería:

$$x_5 \vee x_3 \vee x_2.$$

El paso al que deberíamos de regresar, sería la iteración 2.

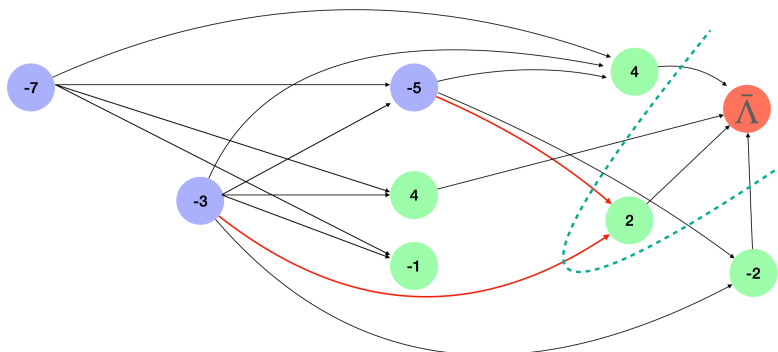

Figura 5.15: Otro ejemplo de corte para la gráfica de implicación.

5.2. Algunas heurísticas modernas basadas en el DPLL

Aunque el algoritmo DPLL se demostró como un método exacto, para resolver el problema del SAT las heurísticas que se basan en este algoritmo surgieron casi 40 años después. Cada

año en cada competencia del SAT surgen nuevas heurísticas que resuelven a una velocidad increíble, instancias del problema del SAT. Por ejemplo, la heurística GLUCOSE puede resolver una instancia de 15,000 cláusulas en un minuto. En esta última sección haremos una revisión bibliográfica de las características de algunas de las heurísticas que se basan en el algoritmo DPLL.

GRASP

La primera heurística que aprovecha al algoritmo DPLL se llama *GRASP: Generic seaRch Algorithm for the Satisfiability Problem* [37] fue propuesta en 1996. El nombre es una coincidencia con la otra heurística que investigamos que también se llama GRASP. Los autores de esta heurística son João P. Marques Silva que trabajaba en Cadence, una institución privada en Lisboa y Karem A. Sakallah de la Universidad de Michigan. A continuación en listamos las características principales de esta heurística.

- *Decidir la siguiente rama:* Para decidir a qué variable asignarle un valor de verdad se evalúa en cada una de las posibles variables cuántas cláusulas se satisfacen si se le asigna un valor de verdad a dicha variable. Se escoge la variable que satisface más cláusulas.
- *Gráfica de decisión:* En el algoritmo de DPLL ordinario se usan dos estructuras, una gráfica de implicación y una gráfica de conflicto donde la segunda es una subgráfica de la primera. En esta heurística sólo implementamos la gráfica de implicación y una estructura que lleva un registro del orden con el que se han ido asignando valores de verdad para las variables de decisión. Esta estructura es un árbol dirigido contenido en el árbol de decisión del problema, la cual nos permite hacer un vuelta hacia atrás eficiente.
- *Corte de conflicto:* Para cada variable conflictiva en la gráfica de implicación se hace un corte que separe a esa variable de todas las variable que le apunten. A partir de esas variables se construye una cláusula de conflicto.
- *Aseveraciones derivadas de conflictos:* A cada variable de conflicto en la gráfica de implicación se le asigna el valor de verdad opuesto.
- *vuelta hacia atrás dirigido por conflictos:* El diseño de esta heurística defiende que hacer un vuelta hacia atrás cronológico desperdicia mucho tiempo en regiones *inútiles* del árbol de búsqueda. Un vuelta hacia atrás cronológico regresa al nivel inmediato anterior en el árbol de decisión, Regresar a cualquier nivel anterior del árbol de decisión es un vuelta hacia atrás no cronológico. En la implementación de GRASP se decide regresar al nivel más antiguo de la primera variable de decisión que generó un conflicto.
- *Mecanismo de diagnóstico:* Agregar cláusulas con cada conflicto aumenta, en el peor escenario, el número de cláusulas en la instancia exponencialmente. Una posible solución para el número de cláusulas que se agregan es delimitar un número natural k, de manera que si la cláusula de conflicto tiene más de k variables entonces no se agrega. De esta manera, el posible número de cláusulas que se agregan aumenta, en el peor escenario, de manera polinomial.

Chaff

Después del éxito de GRASP, una de las heurísticas que causó una revolución a los SAT-solver fue Chaff [42]. Propuesto en 2001 por Matthew W. Moskewicz de UC Berkeley, Conor

F. Madigan de MIT y Ying Zhao, Lintao Zhang, y Sharad Malik de Princeton. Chaff se caracteriza por las siguientes cuatro características que implementa.

- *Propagación de cláusulas optimizada:* Del conjunto de cláusulas que aparecen en la instancia distinguimos aquellas que están *en riesgo* de no ser verdaderas como aquellas a las que se les han asignado valores de verdad a todas sus variables menos dos y el valor de la cláusula no es verdadero. A partir de este reconocimiento de cláusulas disminuimos el número de cláusulas a las que se les debe propagar las restricciones. A esto se le llama **esquema de dos variables vistas**. Este esquema disminuye significativamente el número de propagación de restricciones que se deben hacer, ahorrando recursos computacionales básicos.

- *VSIDS, Variable State Independent Decaying Sum Decision Heuristic:* Este proceso heurístico decide la siguiente rama y está inspirado la heurística DLIS (dynamic largest individual sum) en la que se escoge la variable que aparece con mayor frecuencia en las cláusulas no resueltas. En VSIDS a cada variable se le asigna un contador del número de cláusulas de conflicto en las que aparece, periódicamente los contadores disminuyen. La variable con el mayor contador es la que se escoge, los empates se rompen al azar.

- *Borrado de clausulas:* Esta heurística propone borrar cláusulas de conflicto que han sido agregadas para evitar la explosión de la memoria. Se establece una métrica de relevancia de las cláusulas con la que se determina en cuántas iteraciones en el futuro se eliminará la cláusula. Esta métrica cuenta cuántos valores de verdad no se han asignado a las variables de la cláusula.

- *Reinicios:* Esta heurística propone reinicios en el sentido en el que se borran todos los valores de verdad decididos y se asumen aleatoriamente algunas decisiones para explorar una nueva rama del árbol de decisiones. Lo que no se borra son las gráficas de implicación ni las cláusulas aprendidas. Estos reinicios ocurren de acuerdo a esquemas aleatorios.

GLUCOSE

Para cerrar esta sección y este capítulo describiremos uno de los SAT-Solvers que pertenece (2022) y ha pertenecido desde hace una década al estado del arte. Esta heurística fue propuesta por los matemáticos franceses Gilles Audemard y Laurent Simon en 2009, el nombre GLUCOSE [4] viene del juego de palabras *glue-clause* o *pegar cláusulas*. Esta heurística ha evolucionado y casi cada dos años ha presentado una nueva estrategia que mejora significativamente su eficiencia.

Desde la publicación de Chaff, se realizaron investigaciones que señalaron como indispensables tres de sus estrategias para resolver eficientemente problemas del SAT: Los reinicios, el esquema de dos variables vistas, y la heurística VSIDS. La principal aportación de GLUCOSE es una medida de calidad de las cláusulas que se aprenden llamada **LBD** o distancia de bloques de literales. Se ha probado que para resolver la mayoría de las instancias del problema del SAT que surgen en la industria es necesario eliminar agresivamente una cantidad importante de las cláusulas aprendidas. Hay dos factores que son muy importantes para que la heurística sea eficiente: mantener pequeña a la instancia para que la propagación de restricciones sea eficiente y mantener las cláusulas de mejor calidad.

Los SAT-solvers basados en el aprendizaje de cláusulas aprenden entre 5,000 y 15,000 cláusulas en un segundo. La primera estrategia que surgió para manejar la base de datos

de cláusulas aprendidas consistía en periódicamente eliminar elementos de la base. Aquí hay dos extremos, por un lado borrar muchas cláusulas elimina la eficiencia que se gana con el proceso de aprendizaje y, por otro lado, tener muchas cláusulas es muy poco eficiente. Las primeras investigaciones de GLUCOSE consistieron en determinar que existen cláusulas útiles y cláusulas inútiles. Es decir, cláusulas que aprenderlas o no, impacta en el proceso de resolver una instancia. La pregunta crucial es: ¿cómo distinguir entre cláusulas útiles y cláusulas inútiles durante el proceso de búsqueda?

GLUCOSE surge de una serie de investigaciones del desempeño de los SAT-solvers basados en aprendizaje de cláusulas. Una observación que fue crucial para el desarrollo de GLUCOSE fue que antes de encontrar un conflicto, el número de decisiones disminuye significativamente. Por lo tanto, para distinguir a las mejores cláusulas buscamos aquellas que disminuyan el número de decisiones que se deben hacer. La métrica LBD determina en el proceso de creación de la cláusula en cuántos niveles diferentes del árbol de decisión se le ha asignado valor de verdad a las variables de decisión que aparecen en esa cláusula. La métrica es estática y se mide en el momento en el que la cláusula surge. La implementación de esta medida para las cláusulas construidas permite que la heurística *borre agresivamente* a las cláusulas de baja calidad de acuerdo con la métrica.

Además de la métrica, una de las mejoras significativas del GLUCOSE fue un esquema dinámico de reinicios. Los esquemas anteriores de reinicios eran estáticos, ya que no tomaba en cuenta el desempeño de la heurística para hacer el reinicio. Gracias a la implementación de la métrica podemos hacer análisis estadísticos más profundos durante el proceso del SAT-solver. Por ejemplo, conocer el promedio global de los valores de la métrica y conocer el promedio actual. Si la diferencia es muy grande, se hace un reinicio.

Uno de las primeras mejoras que implementó la heurística GLUCOSE se llama SYRUP. Esta es una de las primeras heurísticas que satisfactoriamente resuelven el problema del SAT usando procesos paralelos. La idea central de esta heurística es usar varios procesos secuenciales de algoritmos de aprendizaje de cláusulas para que cooperativamente resuelvan el problema. La dificultad principal de trabajar en paralelo es inundar procesos de cláusulas. En cada proceso secuencial se generan las **mejores cláusulas**, si estas superan un criterio entonces se exportan a otros procesos. Existe un registro para las cláusulas que se exportan. Si estas se muestran eficientes en cada proceso entonces, su medida de acuerdo con la métrica aumenta.

Capítulo 6

Análisis computacional de las heurísticas

En este capítulo vamos a analizar computacionalmente las heurísticas que hemos implementado a través de un conjunto de instancias que se pueden encontrar en [31]. Primero vamos a describir el conjunto de instancias. Después, vamos a presentar algunos resultados que observamos al correr las heurísticas que implementamos en este conjunto de instancias. En este análisis incluimos a la heurística GLUCOSE. Por último comparamos los resultados que obtuvimos en cada una de las heurísticas. Para realizar cada uno de los resultados que aquí se presentan, se corrió cada heurística en cada instancia con 30 semillas aleatorias diferentes. Para calibrar los parámetros que obtuvimos en los mejores resultados de las heurísticas corrimos un conjunto de pruebas en los que los parámetros que utilizamos tuvieron una diferencia significativa en el desempeño de la heurística.

6.1. Análisis usando instancias homogéneas

El Dr. Holger Hoose de la *Universidad de Columbia Británica* ha creado una biblioteca de instancias para el problema de satisfacibilidad booleana [31]. En esta primera sección, vamos a estudiar 5 instancias *Uniform Random-3-SAT* de esta biblioteca. Estas instancias están descritas en la tabla 6.1.

Nombre de la instancia	Número de variables	Número de cláusulas	¿Satisfacible?
p20	20	91	Sí
p50	50	218	Sí
p75	75	325	Sí
p100	100	430	Sí
p125	125	538	Sí

Tabla 6.1: Descripción de un conjunto de cinco instancias de [31].

Una primera pregunta que podemos hacernos es cuánto tardaríamos en resolver estas instancias usando fuerza bruta. El espacio de búsqueda de cada una de las instancias es 2 elevado al número de variables de cada instancia. En cada elemento del espacio de búsqueda se evalúa la función objetivo y se compara con el número de cláusulas, que es el valor óptimo. En la tabla 6.2 registramos para cada instancia cuánto tiempo tarda en ser evaluada la función objetivo que implementamos y el tiempo que tardaríamos en hacer una búsqueda por fuerza bruta.

Nombre de la instancia	Duración de una evaluación	Duración de búsqueda por fuerza bruta
p20	0.002 seg	35 minutos
p50	0.009 seg	71404.1 años
p75	0.044 seg	1.75 veces la edad del universo
p100	0.045 seg	120,590,810.5 veces la edad del universo
p125	0.053 seg	4.7×10^{13} veces la edad del universo

Tabla 6.2: Tiempo que tardaría resolver cada instancia por fuerza bruta.

Como podemos ver en la tabla 6.2, cuatro de las cinco instancias que estudiamos no pueden ser resueltas de manera práctica por fuerza bruta. Tenemos que resaltar que evaluar cada instancia en una asignación toma tiempos distintos porque las instancias tienen diferentes tamaños. Por tamaño nos referimos al número de cláusulas que tiene. Sin embargo ésta no es la razón por la que el tiempo que tarda el proceso de fuerza bruta crece exponencialmente. La duración de la fuerza bruta depende exponencialmente del tamaño del espacio de búsqueda y linealmente del tiempo que tarda en hacerse una evaluación.

Para cada heurística que estudiamos, vamos a describir el desempeño de le heurística sobre este conjunto de instancias.

6.1.1. Detalles generales sobre las implementaciones

Para implementar estas heurísticas consideré como solución a para una instancia de n variables a una sucesión de ceros y unos de longitud n. Construí la vecindad de una solución como el conjunto con $n+1$ soluciones tal que cada solución en el conjunto difiere por exactamente una entrada de la solución original.

En el método de búsqueda local decidí buscar exhaustivamente en toda la vecindad para escoger la *mejor mejora*.

Como criterio de paro en todas las heurísticas implementadas usé el tiempo computacional de corrida de la heurística.

Todos los parámetros se calibraron tras hacer corridas de hasta 4 minutos.

Para cada corrida se fija una semilla aleatoria. Los resultados que se muestran en esta sección se obtuvieron a partir de 30 semillas aleatorias distintas.

6.1.2. Escalar colinas

La heurística de escalar colinas no alcanza el valor óptimo en ninguna de las instancias. En la tabla 6.3 mostramos para cada instancia el mejor valor obtenido por esta heurística.

Instancia	Mejor valor obtenido	Valor óptimo
p20	89	91
p50	215	218
p75	321	325
p100	412	430
p125	522	538

Tabla 6.3: Mejores valores obtenidos en la heurística de escalar colinas.

Aunque no obtuvimos el valor óptimo, podemos ver que la solución que obtenemos es casi 10 % peor que la solución óptima en los cinco casos.

El mal desempeño de esta heurística se debe al reducido espacio que ésta explora. Hicimos un experimento para mostrar esto. Corrimos esta heurística en cada una de las instancias y cada 6 segundos registramos cuántas soluciones diferentes se exploraron en ese intervalo de tiempo.

Siguiendo esta metodología encontramos que en menos de 20 segundos, el método de escalar colinas encuentra un óptimo local que es óptimo de su propia vecindad por lo que después de 20 segundos la heurística sólo explora una misma solución.

En cada una de las instancias que exploramos, en 20 segundos la heurística de escalar colinas encuentra un atractor del que no puede escapar. Aún así, la heurística ofrece soluciones que son aproximadamente 90 % tan buenas como el óptimo.

6.1.3. Búsqueda Tabú

La heurística de búsqueda local no alcanza el valor óptimo en ninguna de las instancias. De hecho, los mejores valores que obtuvimos al correr esta heurística no son mucho mejores que los que obtuvimos al correr la heurísticas de escalar colinas.

Ejecutamos la heurística y cada 0.20 segundos hicimos una copia de la lista tabú. Observamos que después de un tiempo determinado la heurística encontraba una solución que era óptimo local de su vecindad y de todas las vecindades de cada una de las soluciones en su vecindad. En la tabla 6.4 mostramos para cada instancia el mejor valor obtenido por esta heurística y el tiempo que tardó la heurística en encontrar una solución que ya no pudo mejorar.
Implementamos una lista Tabú de longitud ilimitada, ya que después de un cierto número de iteraciones la heurística no mejoraba no era importante acotar el tamaño de la lista.

Instancia	Mejor valor obtenido	Valor óptimo	Tiempo que tarda en agotar las vecindades
p20	89	91	< 0.20 seg
p50	215	218	7 seg
p75	317	325	22 seg
p100	419	430	30 seg
p125	524	538	37 seg

Tabla 6.4: Mejores valores obtenidos en la heurística de búsqueda tabú.

6.1.4. Recocido Simulado

Para esta heurística usamos una temperatura de $1000C$ y un esquema de enfriamiento de $\alpha = 0.9$.

La heurística de recocido simulado alcanzó el óptimo local en cada instancia usando el mismo par de parámetros.

En la gráfica 6.1 representamos el tiempo que tardó esta heurística en llegar al óptimo. Comparamos el tamaño de la instancia y el tiempo que la heurística tardó en llegar al óptimo.

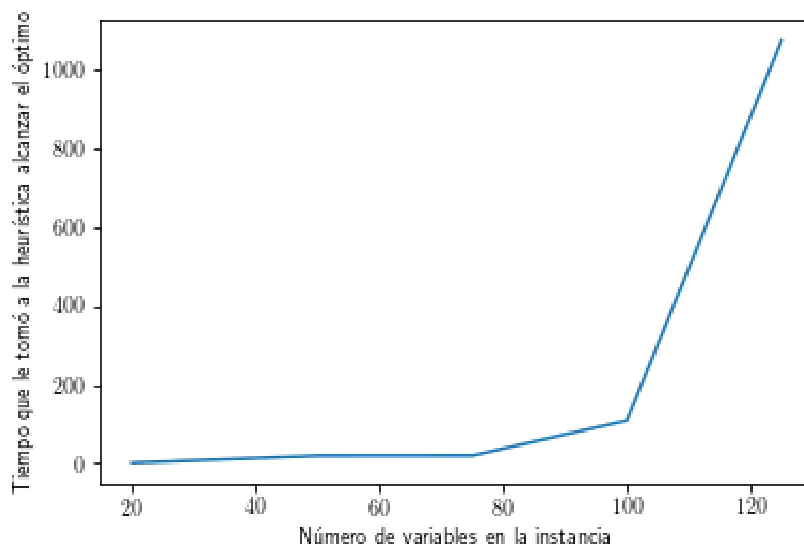

Figura 6.1: Tiempo que le tomó a recocido simulado alcanzar el óptimo.

En esta figura podemos ver cómo el tiempo crece exponencialmente con respecto al número de variables en la instancia. Ejecutamos esta heurística en varias ocasiones en el mismo conjunto de instancias. Hubo una semilla en la que recocido simulado encontró el óptimo de la instancia más grande en 42 segundos y hubo otra semilla en la que recocido simulado tardó más de veinte minutos.

A diferencia de las heurísticas de escalar colinas y de búsqueda tabú, en recocido simulado aceptamos soluciones que disminuyen el desempeño en la función objetivo. En la gráfica 6.2 registramos los valores en la función objetivo en cada iteración de recocido simulado al correrla sobre la instancia p75.

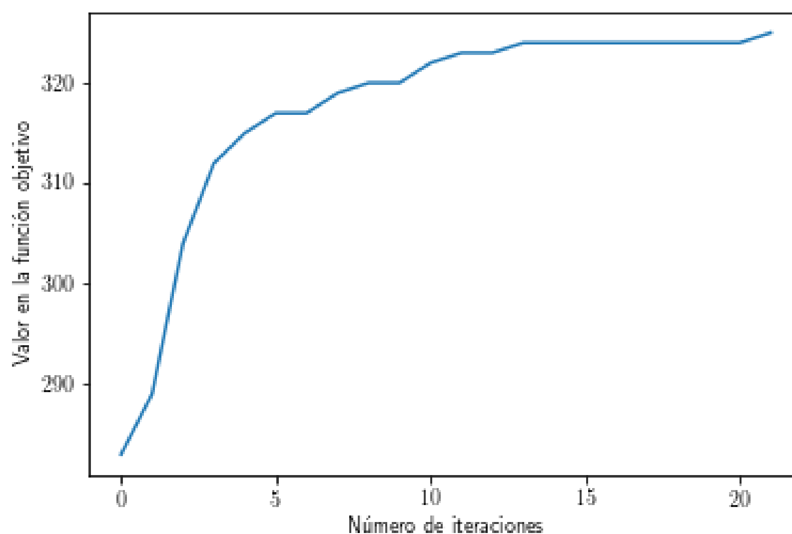

Figura 6.2: Valor en la función objetivo a lo largo de recocido simulado.

Podemos ver en esta gráfica que la función crece a un ritmo acelerado en las primeras iteraciones y en las siguientes el ritmo de crecimiento es mucho más lento. Se puede ver que esta función no es monótona y que cerca del óptimo (325) la función es constante.

6.1.5. GRASP

Para la heurística GRASP decidimos hacer una lista restringida de candidatos tomando al 20 % de los mejores candidatos. De esta lista restringida de candidatos tomamos un elemento al azar. Al tomar esta decisión, sólo tomamos en cuenta maximizar el número de cláusulas satisfechas.

Con la heurística de GRASP, alcanzamos el óptimo en la instancia p20 y p50. Para las instancias p75, p100 y p125 no alcanzamos el valor óptimo en menos tiempo que con la heurística de recocido simulado.

Esta es una heurística constructiva multiarranque, es decir que en cada iteración de esta heurística se construye una solución. Corrimos la etapa constructiva de esta heurística sobre la instancia p75 por 23 segundos, que es el tiempo que tardó recocido simulado en alanzar el óptimo. En la gráfica 6.3, mostramos los valores obtenidos en cada una de las soluciones que se obtienen de la etapa constructiva de la heurística. Corrimos esta heurística usando diferente valores para α, el parámetro que determina cuántas soluciones aceptamos en nuestra lista restringida de candidatos. En cada columna está representado el conjunto de valores que se obtuvieron en la función objetivo con respecto al valor de α.

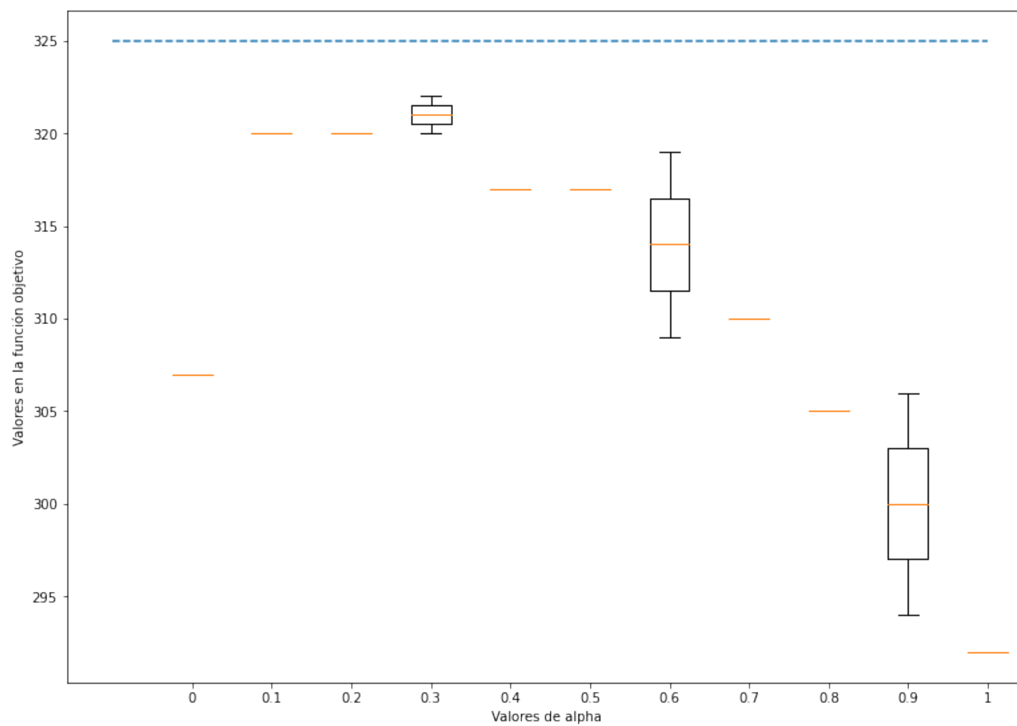

Figura 6.3: Valores en la función objetivo para las soluciones construidas en GRASP en una misma instancia.

En una gráfica de caja y bigote se muestra el valor máximo obtenido (la línea superior a la caja), el valor mínimo (la línea inferior), la mediana (la línea punteada dentro de la caja) y el primer y tercer cuartil que son los valores que limitan la caja. Con una línea azul y punteada representamos el valor óptimo de la función objetivo. Cuando sólo vemos una línea naranja es porque el conjunto de valores obtenidos sólo consta de un elemento.

Podemos ver que independientemente del valor que escojamos para α, las soluciones que se construyen con GRASP están muy por debajo del valor óptimo. Es interesante observar que sólo para tres valores de α los conjuntos tuvieron más de un elemento. Se puede observar que para valores de α cercanos a 0.2 GRASP tuvo el mejor rendimiento.

6.1.6. Búsqueda por vecindades variables

Nuestras estructuras de vecindad estuvieron definidas de la siguiente manera: La vecindad $V(n,c)$ de una solución s son n soluciones que difieran a lo más por c entradas de s. Si permitimos c cambios para la solución s, entonces tenemos que considerar $longitud(s)^c$ soluciones vecinas. Por eso acotamos el tamaño de la vecindad usando n. Para construir estas vecindades, escogemos aleatoriamente dónde hacer el cambio a la solución inicial s. Las estructuras de vecindad usuales que utilizamos son $V(long(s),1)$ donde realizamos todos los posibles cambios.

Para los experimentos que realizamos, usamos la siguiente sucesión de vecindades:

$$V(100,1), V(200,2), V(225,2), V(250,2),$$
$$, V(275,2), V(300,2), V(300,3), V(300,4).$$

De la misma manera que con GRASP, la heurística de búsqueda por vecindades variables

no encuentra el óptimo en las instancias de p75, p100 y p125 pero sí lo hace en p20 y p50.

Hicimos un análisis para las cinco instancias en el que registramos cuántas soluciones mejores fueron encontradas y cuántas veces se corrió la heurística sobre esa misma solución. En la gráfica 6.4 mostramos el caso de la instancia p100.

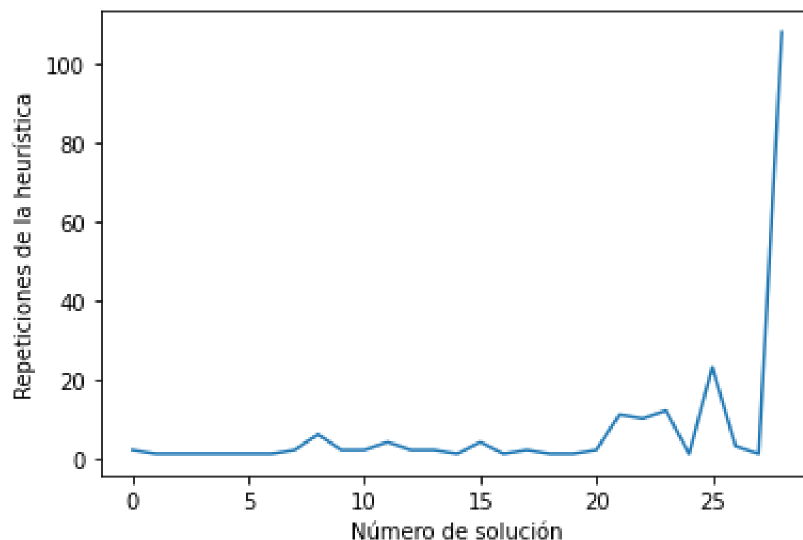

Figura 6.4: Número de veces que se corrió la heurística sobre una misma solución para la instancia p100.

En este experimento encontramos 29 soluciones que mejoraron el valor en la función objetivo. Estas soluciones se enumeran en el orden en el que fueron apareciendo y en la gráfica 6.4 están representadas en el eje horizontal. En el eje vertical contamos cuántas veces se corrieron la heurística de vecindades variables usando como solución inicial esa solución. Para este experimento usamos ocho estructuras de vecindad en orden creciente con respecto a su tamaño.

Podemos observar que la mayor parte del tiempo sobre el que corrió la heurística se utilizó para intentar escapar del óptimo local que se encontró. Se corrió más de cien veces la heurística sobre esa solución mientras que para las demás soluciones no se corrió más de 30 veces.

Esto muestra que hay óptimos locales para los que se necesita diversificar mucho más de lo que las estructuras de vecindad que utilizamos ofrece.

6.1.7. Algoritmos genéticos

Para esta heurística usamos el método de selección por ruleta, es decir escogemos aleatoriamente a los individuos que vamos a reproducir distribuyendo la probabilidad de ser escogido en función de su desempeño en la función objetivo. Escogemos a un 20 % Para el método de cruza escogemos al azar un punto de las dos soluciones en donde las recortamos y las pegamos para obtener una nueva solución. Para el método de mutación, cambiamos una entrada al azar de cada uno de los hijos con una probabilidad de 20 %. Reemplazamos al 20 % de la población de acuerdo a su desempeño en la función objetivo. Es decir quita-

mos a los *peores* y los reemplazamos por los hijos.

En la heurística de algoritmos genéticos sólo obtuvimos el valor óptimo en la instancia más pequeña. Los valores que obtuvimos en las demás instancias están representados en la tabla 6.5.

Instancia	Mejor valor obtenido	Valor óptimo
p20	91	91
p50	215	218
p75	321	325
p100	412	430
p125	513	538

Tabla 6.5: Mejores valores obtenidos en la heurística de algoritmos genéticos.

Los valores que obtuvimos son muy similares a los que obtuvimos en la heurística de escalar colinas. Sin embargo, para obtener estos valores dejamos correr la heurística por 240 segundos, mientras que la heurística de escalar colinas tardaba 20 segundos en obtener estos valores.

Podemos ver que la heurística se comporta como es esperado en el sentido en el que el rendimiento de la población aumenta a lo largo de las iteraciones de la heurística. Hicimos un experimento en el que registramos los valores de la función objetivo en cada elemento de la población para cada iteración. En cada iteración promediamos los valores obtenidos. En la Figura 6.5 se muestran los resultados para la instancia p75.

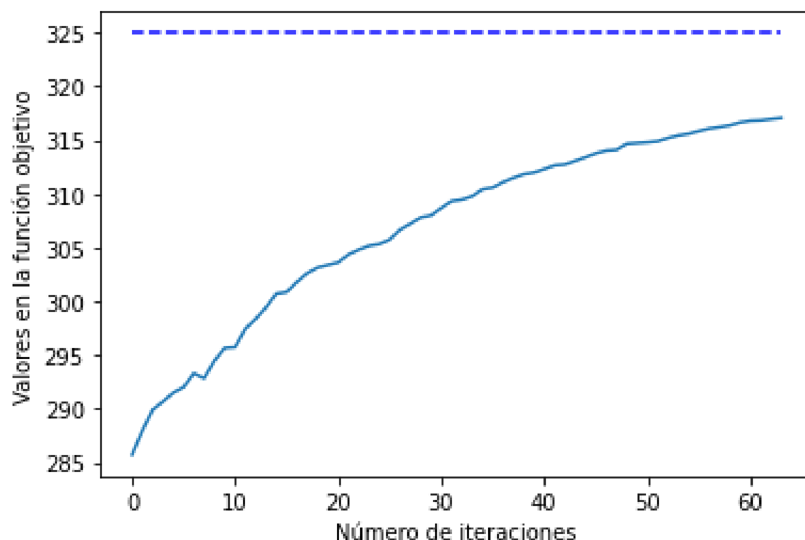

Figura 6.5: Valor promedio del rendimiento de la población a lo largo de la heurística de algoritmos genéticos.

En la Figura 6.5, el valor óptimo está representado por una línea azul y punteada, es decir 325. Podemos ver cómo el desempeño promedio está muy por debajo del valor óptimo. Sin embargo, la función crece a lo largo del número de iteraciones, este crecimiento no es monótono aunque los decrementos son muy pequeños o la función se mantiene constante.

6.1.8. Optimización por colonias de hormigas

Para esta heurística usamos como cada candidato a las $2n$ posibles asignaciones. En cada recorrido de las hormigas, sólo pueden escoger variables que hagan que la solución sea factible. Es decir, no permitimos que una hormiga camine sobre x_2 y sobre $\neg x_2$. Los valores de la matriz de probabilidades son todos iguales en la primera iteración y cambian de acuerdo con los recorridos de las hormigas.

Al igual que con la heurística de algoritmos genéticos, en la heurística de optimización por colonias de hormigas sólo obtuvimos el valor óptimo para la instancia de p20 en el mismo tiempo que recocido simulado.

Instancia	Mejor valor obtenido	Valor óptimo
p20	91	91
p50	209	218
p75	306	325
p100	401	430
p125	500	538

Tabla 6.6: Mejores valores obtenidos en la heurística de optimización por colonias de hormigas.

En la tabla 6.6 podemos ver que los valores que se obtuvieron estuvieron muy por debajo del valor óptimo. Esto sugiere que las heurísticas constructivas no funcionan adecuadamente como las hemos planteado para resolver el problema de satisfacibilidad booleana.

6.1.9. GLUCOSE

En esta última parte, vamos a mostrar cómo se desempeña la heurística de GLUCOSE en este conjunto de instancias. Esta heurística no la implementamos nosotros. La implementación original se hizo en C, pero los investigadores Alexey Ignatiev, Joao Marques-Silva, y Antonio Morgado desarrollaron en 2018 una biblioteca en Python en la que se pueden correr una gran variedad de heurísticas específicas para el problema de satisfacibilidad booleana. Más información sobre esta biblioteca se puede encontrar en [32].

En la tabla 6.7 registramos el tiempo que le tomó a GLUCOSE encontrar el óptimo global.

Instancia	Tiempo en alcanzar el óptimo global
p20	0.003 s
p50	0.004 s
p75	0.008 s
p125	0.011 s

Tabla 6.7: Tiempo que tardó GLUCOSE en encontrar el óptimo global.

Es interesante observar que esta heurística tarda menos en encontrar el óptimo global que hacer una sola evaluación de la función objetivo que hemos estado trabajando. Se puede observar un crecimiento lineal en el tiempo que tarda la heurística con respecto al tamaño de la instancia ya que las instancias que estamos usando son pequeñas para esta heurística. Sin embargo, tenemos que recordar que esta heurística ha pertenecido al podio

de ganadores de la competencia del SAT en varias de sus ediciones. Se ha observado que, como es de esperarse, el tiempo que tarda esta heurística en resolver el problema crece exponencialmente con respecto al número de variables [43].

6.2. Comparación de las heurísticas

En esta última sección vamos a comparar los resultados que obtuvimos a través del análisis computacional. Corriendo cada heurística en 30 semillas diferentes para cada instancia, consideramos la mejor solución que obtuvimos. Recordemos que como cada una de las instancias que utilizamos es satisfacible, entonces la solución óptima debe satisfacer a la totalidad de cláusulas.

Tomando en cuenta el mejor valor que obtuvimos de la heurística y el valor óptimo podemos definir la **desviación porcentual** como sigue:

$$\text{Desviación Porcentual} = \frac{\text{Valor Óptimo} - \text{Mejor Valor}}{\text{Valor Óptimo}} \times 100.$$

Es claro que la desviación porcentual es cero, cuando el mejor valor es el valor óptimo. También es claro que valores cercanos a cero representan buenas soluciones.

En la tabla 6.8, calculamos la desviación porcentual de los mejores valores que obtuvimos en cada una de las heurísticas para cada una de las instancias.

Heurística	Desviación porcentual en la instancia p20	Desviación porcentual en la instancia p50	Desviación porcentual en la instancia p75	Desviación porcentual en la instancia p100	Desviación porcentual en la instancia p125
Escalar Colinas	2.19	1.39	1.23	4.18	2.97
Búsqueda Tabú	2.19	1.39	0.67	2.55	2.60
Recocido Simulado	0	0	0	0	0
GRASP	0	0	4.71	3.48	4.34
Búsqueda por Vecindades Variables	0	0	0.56	1.56	2.03
Algoritmos Genéticos	0	1.39	1.84	4.18	4.64
Optimización por Colonias de Hormigas	0	2.19	3.52	4.21	3.91

Tabla 6.8: Desviación porcentual de los mejores valores obtenidos usando todas las heurísticas.

Podemos ver que la única heurística que encontró el óptimo en todas las instancias fue recocido simulado, que escalar colinas y búsqueda tabú no encontraron el valor óptimo en ninguna. Algoritmos genéticos y optimización por colonia de hormigas sólo encontraron el óptimo en la instancia más pequeña, a diferencia de GRASP que encontró el óptimo en la dos instancias más pequeñas. Podemos ordenar de acuerdo al promedio de desviación porcentual a las heurísticas como: Recocido Simulado (0 %),Búsqueda Tabú (1.88 %), Escalar Colinas (2.39 %), Búsqueda por Vecindades Variables (2.41 %), GRASP (2.506 %), y Optimización por Colonias de Hormigas (2.766 %).

6.3. Conclusiones

A lo largo del último capítulo analizamos las limitaciones que tiene cada heurística que implementamos y las complicaciones que presentan. La única heurística que con certeza encontró los óptimos en cada instancia fue **recocido simulado**. En las demás heurísticas encontramos un conjunto de dificultades que nos permitieron encontrar el valor óptimo.

En esta última sección vamos a sugerir un conjunto de posibles soluciones.

Tuvimos un inesperado mal desempeño en las heurísticas constructivas: GRASP y Optimización por Colonias de Hormigas. Esto puede deberse a que la función objetivo cuando evalúa soluciones parciales tiene dos salidas. La función objetivo para soluciones parciales S cuenta cuántas cláusulas se satisfacen con S en la instancia, pero hay otra cantidad a tomar en cuenta: el número de cláusulas de conflicto inducidas por S, es decir cuántas cláusulas son falsas dada la asignación S. Siguiendo una heurísitica constructiva debemos buscar maximizar el número de cláusulas satisfechas y minimizar el número de cláusulas de conflicto. En nuestra implementación sólo buscábamos maximizar el número de cláusulas satisfechas. Esta modificación impactaría en cómo definimos a la lista restringida de candidatos para GRASP y cómo definiríamos la matriz de probabilidades para optimización por colonias de hormigas.

Para estas dos heurísticas constructivas podemos inspirarnos en las heurísticas específicas para el problema de satisfacibilidad booleana. Por ejemplo, podríamos permitir reinicios del proceso multiarranque de GRASP si es que el número de cláusulas de conflicto es muy alto o si el número de cláusulas satisfechas es bajo. El objetivo de esto sería seguir un algoritmo de vuelta hacia atrás.

En este mismo sentido podemos implementar sobre las heurísticas específicas para el problema de satisfacibilidad booleana estrategias heurísticas como GRASP para escoger la siguiente variable sobre la que se hace propagación unitaria o cuál cláusula de conflicto borrar o en qué momento hacer un reinicio.

La heurística de búsqueda local se enfrentó con agotar vecindades de óptimos locales, lo que le impidió mejorar el rendimiento de la función objetivo. Podemos diversificar la búsqueda introduciendo sistemas de vecindades dinámicas como lo hacemos en vecindades variables. O podemos inspirarnos en las listas tabú que en vez de enlistar soluciones enlistan movimientos. Es decir, no permitir que en la vecindad a examinar haya soluciones que tengan ciertas características. Para identificar las características podemos seleccionar segmentos de la mejor solución y filtrar que en la vecindad no haya ninguna solución que comparta ese segmento.

En la heurística de búsqueda por vecindades variables nos enfrentamos a búsquedas muy profundas y muy poco diversas. Al igual que en búsqueda tabú, sin importar la estructura de vecindad que utilicemos, la heurística no lograba escapar del atractor. Observamos experimentalmente que recocido simulado escogía consecutivamente malas soluciones a esto le atribuimos el lograr escapar de los atractores. En búsqueda por vecindades variables hacemos una selección aleatoria y luego una búsqueda local. Sugerimos implementar un contador que rastree cuántas veces se ha corrido la heurística sobre la misma solución inicial y que con éste se determine hacer una secuencia de malas decisiones para diversificar la búsqueda.

Es claro que las heurísticas diseñadas para resolver problema del SAT frente a las heurísticas que implementamos tienen un mejor desempeño, sin embargo debemos tomar en cuenta que nuestro enfoque fue el de resolver un problema de optimización. Mientras que las heurísticas asociadas al problema del SAT están enfocadas en resolverlo como un problema de decisión. Esta diferencia hubiera sido más evidente si hubiéramos trabajado con instancias no satisfacibles. Ya que con las heurísticas específicas al problema del SAT sólo hubiéramos recibido un resultado: no satisfacible. Pero con la heurísticas que implementamos hubiéramos obtenido soluciones *buenas* aunque no óptimas.

Capítulo 7

Conclusiones

En este trabajo estudiamos métodos no deterministas para resolver el problema de la satisfacibilidad booleana. Hicimos una introducción teórica y accesible a los problemas *NP*-completos. En la que describimos de una manera intuitiva por qué son más complicados usando una variedad de ejemplos. En particular, describimos el problema de la satisfacibilidad booleana. Este problema es, a diferencia de los demás problemas NP completos, muy difícil de enunciar y parece no tener aplicaciones a la vida real. No obstante, mostramos que tiene una importancia teórica enorme. Además, describimos algunas metodologías modernas que han logrado resolver enormes instancias del problema del SAT en tiempos muy pequeños. Lo que hace que los solucionadores del problema del SAT estén a la vanguardia de la investigación de operaciones y que haya un gran interés por traducir otros problemas NP completos en instancias del problema de la satisfacibilidad booleana de manera eficiente.

Hicimos una descripción formal de la teoría de computación necesaria para entender y describir qué son los problemas *NP*-completos. Definimos las máquinas de Turing deterministas y no deterministas y mostramos que cualquier problema *NP*-completo se puede reducir en un tiempo polinomial al problema de la satisfacibilidad booleana.

Hicimos un recorrido teórico por las heurísticas clásicas. Investigamos su origen, describimos su pseudocódigo y realizamos su diagrama de flujo. Las clasificamos como heurísticas de trayectoria y como heurísticas de poblaciones. Además implementamos desde cero cada una de estas heurísticas.

En el recorrido teórico, estudiamos las heurísticas del estado del arte que resuelven el problema de la satisfacibilidad booleana. Describimos dos de los principales componentes que subyacen a estas metodologías: el agoritmo de Davis-Putman-Loveland-Logeman y el aprendizaje de cláusulas por medios de una gráfica de implicación. Estas heurísticas del estado del arte no las implementamos pero sí las corrimos.

Hicimos un estudio computacional de las heurísticas que implementamos usando un conjunto de instancias que satisfacen ser homogéneas en el sentido que todas las variables aparecen el mismo número de veces. En este estudio también corrimos una heurística del estado del arte. Los resultados que obtuvimos con inequiparables con los del estado del arte. Sin embargo, implementar estas heurísticas nos permitió entender cada una de sus partes y elementos más importantes.

A partir de este trabajo podemos plantear implementar las estrategias que usan las heurísticas del estado del arte junto las heurísticas clásicas. Es decir, implementar una nueva heurística. También queda pendiente estudiar instancias de la vida real con las heurísticas

que implementamos, tratando de rediseñar la heurística para que funcione de manera óptima según la instancia y no tratando de atacar a todas las posibles instancias del problema.

Apéndice A

Implementaciones

A.1. Implementaciones elementales

```
def load(fn):
  f = open(fn,    )
  data = f.read()
  f.close()
  clauses = []
  lines = data.split(    )
  p=[]

  for line in lines:
    if len(line) == 0 or line[0] in [   ,    ,    ]:
      continue

    if line[0]==   :
      problemline=line.split()
      p.extend([int(problemline[2]),int(problemline[3])])
    else:
      clause = [int(x) for x in line.split()[0:-1]]
      clauses.append([x for x in clause])

  P=np.array(clauses)

  return (p,P)
```

Implementación A.1: Cargar instancias en formato cnf

```
def f(P,s):
  (n,m)=P.shape
  values=np.zeros(n)
  Q=np.copy(P)
  for i in range(n):
    for j in range(m):
      if s[np.abs(Q[i,j])-1]==0:
        Q[i,j]=Q[i,j]*(-1)
    values[i]=np.sign(np.max(Q[i,:]))
  values[values==-1]=0
  return sum(values)
```

Implementación A.2: Función objetivo

```
def Neighborhood(p,s,n,c,seed):
  m=p[0]
```

```
  V=np.zeros((n+1,m))
  V[0,:]=s
  random.seed(seed)
  for i in range(1,n+1):
    V[i,:] = s
    for j in range(c):
      c1=random.randint(0,m)
      V[i,c1]=1-V[i,c1]
  return V
```

Implementación A.3: Estructura de vecindad

A.2. Codificaciones

*

Número cromático

```

def load(fn):
  f = open(fn,    )
  data = f.read()
  f.close()
  clauses = []
  lines = data.split(    )
  p=[]

  for line in lines:
    if len(line) == 0:
      continue

    if line[0]==   :
      problemline=line.split()
      p.extend([int(problemline[2]),int(problemline[3])])
      break

  P = np.zeros((p[0],p[0]))

  for i in range(len(lines)):
    line = lines[i].split()
    if len(line) == 0:
      continue
    if line[0] ==    :
      if line[1] <= lines[2]:
        P[int(line[1])-1,int(line[2])-1] = 1
      else:
        P[int(line[2])-1,int(line[1])-1] = 1

  return (p,P)
```

Implementación A.4: Cargar instancias en el formato gráfico col y hal

```
def encode_graph_coloring(edges,n,k):
    m=len(edges)

    n_clauses = 3*m+n
    n_vars = 3*n
    clauses = []
```

```
    for u, v in edges:
      for i in range(k):
        clauses.append([-(i*n+u), -(i*n+v)])
    for v in range(1, n+1):
      clauses.append([i*n+v for i in range(k)])
    return clauses
```

Implementación A.5: Codificador del problema del número cromático

```
import seaborn as sns
import numpy as np
import networkx as nx

def ListOfColors(k):
  palette = sns.color_palette(None, k)
  color = list(palette.as_hex())
  return color

def get_colors(assignments,n,k):
    all_colors = np.array(list(range(k)))
    colors = []
    for v in range(n):
      colors.append(all_colors[[assignments[v]>0, assignments[n+v
    ]>0, assignments[2*n+v]>0]])
      colors = [list(colors[i]) for i in range(len(colors))]
    return colors

def feasable_coloring(colors):
  return [color[0] for color in colors]
```

Implementación A.6: Funciones adicionales para obtener una paleta de colores

```
def GraphColoring(P,s,color):
  s = list(s)
  G = nx.Graph(P)
  color_map = []
  for node in G:
    color_map.append(color[s[node-1]])
  posCirc=nx.circular_layout(G)
  nx.draw(G,node_color=color_map,with_labels=True, pos=posCirc)
  return
```

Implementación A.7: Decodificador del problema del número cromático

*

Camino Hamiltoniano

```
def reduce_Hamiltonian_Path_to_SAT(edges,n):

    def index(i, j):
      return n*i + j + 1

    m = len(edges)
    n_clauses = 2*n + (2*n*n-n-m)*(n-1)
    n_vars = n*n
    clauses = []

    for j in range(n):
```

```
      clause = []
      for i in range(n):
        clause.append(index(i,j))
      clauses.append(tuple(clause))


    for i in range(1,n+1):
      clause = []
      for j in range(n*(i-1), n*i):
        clause.append(j+1)
      clauses.append(tuple(clause))


    for j in range(n):
      for i in range(n):
        for k in range(i+1, n):
          clauses.append((-index(i,j), -index(k,j)))

    for i in range(n):
      for j in range(n):
        for k in range(j+1, n):
            clauses.append((-index(i,j), -index(i,k)))

    for k in range(n-1):
      for i in range(n):
        for j in range(n):
          if i == j: continue
          if not [i+1, j+1] in edges:
              clauses.append((-index(k,i), -index(k+1,j)))

    clauses = [list(clause) for clause in clauses]
    return clauses
```

Implementación A.8: Codificador del problema del ciclo Hamiltoniano

```

import networkx as nx

def get_hamiltonian_path(assignments,n):
    path = [None]*n
    for i in range(n):
      for j in range(n):
       if assignments[i*n+j] > 0:
        path[i] = j+1
    return path



def GraphHamiltonian(P,path):

  G = nx.Graph()
  for edge in P:
    if edge in path or [edge[1],edge[0]] in path:
      G.add_edge(edge[0],edge[1],color=    ,weight=4)
    else:
      G.add_edge(edge[0],edge[1],color=    ,weight=2)

  pos = nx.circular_layout(G)

```

```
  edges = G.edges()
  colors = [G[u][v][       ] for u,v in edges]
  weights = [G[u][v][        ] for u,v in edges]

  nx.draw(G, pos, edge_color=colors, width=weights,with_labels=True)

  return
```

Implementación A.9: Decodificador del problema del ciclo Hamiltoniano

*

N-Reinas

```
def EncodeNQueens(n):

    count=0
    file1 = []
    for i in range(0,n):
     str0=  ;
     for j in range(1,n+1):
      number=str(j+n*i);
      str0=str0+number+
     str0=str0
     file1.append(str0);
     count+=1

    for i in range(0,n):
     for j in range(1,n):
       number=j+n*i;
       for l in range(1,n-j+1):
        str0=   +str(number)+    +str(number+l)
        file1.append(str0);
        count+=1

    for j in range(1,n+1):
     for i in range(0,n):
       number=j+n*i;
       for l in range(1,n-i):
        str0=   +str(number)+    +str(number+n*l)
        file1.append(str0);
        count+=1

    for i in range(0,n-1):
     for j in range(i,n-1):
      number=j+1+n*i;
      for l in range(1,n-j):
        str0=   +str(number)+    +str(number+l*(n+1))
        file1.append(str0);
        count+=1

    for i in range(0,n-1):
     for j in range(0,i):
      number=j+1+n*i;
      for l in range(1,n-i):
       str0=   +str(number)+    +str(number+l*(n+1))
       file1.append(str0);
       count+=1

```

```
    for i in range(0,n):
     for j in range(0,n-i):
      number=j+1+n*i;
      for l in range(1,j+1):
       str0=   +str(number)+    +str(number+l*(n-1))
       file1.append(str0);
       count+=1

    for i in range(0,n):
     for j in range(n-i,n):
      number=j+1+n*i;
      if (number != n*n ):
       for l in range(1,n-i):
        str0=   +str(number)+    +str(number+l*(n-1))
        file1.append(str0);
        count+=1

    return [list(map(int,file1[i].split())) for i in range(len(file1
   ))]
```

Implementación A.10: Codificador del problema de las n-reinas

```

from mlxtend.plotting import checkerboard_plot
import matplotlib.pyplot as plt
import numpy as np


def get_chessboard(assignment,n):
  board = []
  for i in range(n):
    row = []
    for j in range(n):
      row.append(assignment[n*i +j])
    board.append(row)

  ary = np.array(board)
  brd = checkerboard_plot(ary,col_labels=[   ]*3,row_labels=[   ]*3,
   cell_colors=[             ,             ],font_colors=[       ,
         ],figsize=(5, 5))
  plt.show()

  return
```

Implementación A.11: Decodificador del problema de las n-reinas

A.3. Heurísticas

*

Escalar colinas

```

def LocalSearch(P,V):
  v=[]
  for i in range(V.shape[0]):
    v.append(f(P,V[i,:]))
  f_star=max(v)
```

```
  s_star=V[v.index(max(v))]
  return (f_star,s_star)
```

Implementación A.12: Búsqueda local

```

def HillClimbing(P,p,s,n,c,seed,TIME):
  start_time = time.time()
  now_time = time.time()

  s_star = np.copy(s)

  while now_time - start_time < TIME:
    (f_star,s_star) = LocalSearch(P,Neighborhood(p,s_star,n,c,seed))
    s = np.copy(s_star)
    now_time = time.time()

  return (f_star,s_star)
```

Implementación A.13: Escalar las colinas

Búsqueda Tabú

```
def CleanTheNeighborhood(V,TabuList):

  CleanV = np.copy(V)

  EraseList = []

  for i in range(V.shape[0]):
    for j in range(len(TabuList)):
      if np.array_equal(V[i,:],TabuList[j]):
        EraseList.append(i)


  for i in range(len(EraseList)):
    if CleanV.shape[0] != 0:
      EraseList[i] = EraseList[i]-i
      CleanV = np.delete(CleanV,EraseList[i],0)
    else:
      break

  return CleanV
```

Implementación A.14: Limpiar la vecindad

```
def TabuSearch(P,s,n,c,seed,TIME,l):
  start_time=time.time()
  now_time=time.time()

  TabuList=[]
  s_star = np.copy(s)
  f_star = f(P,s)

  while now_time - start_time < TIME:

    V = Neighborhood(p,s_star,n,c,seed)
    V = CleanTheNeighborhood(V,TabuList)
```

```
    if V.shape[0]!=0:

      (f_s,s) = LocalSearch(P,V)
      TabuList.append(s)
      if len(TabuList) > 1:
        TabuList.pop(0)
      if f_s > f_star:
        s_star = np.copy(s)
        f_star = f_s

      now_time = time.time()

    else:
      now_time = time.time()

  return (f_star,s_star)
```

Implementación A.15: Búsqueda Tabú

Recocido Simulado

```
def SimulatedAnnealing(P,s,t,alpha,n,c,seed,TIME):

  s_star = np.copy(s)
  f_star = f(P,s_star)

  start_time = time.time()
  now_time = time.time()

  while now_time - start_time < TIME:
    V = Neighborhood(p,s_star,n,c,seed)
    (f_s,s) = LocalSearch(P,V)
    if f_s >= f_star:
      f_star = f_s
      s_star = np.copy(s)

    elif np.exp((f_star - f_star)/t) > random.uniform():
      f_star = f_s
      s_star = np.copy(s)


    t = t* alpha
    now_time=time.time()


  return (f_star,s_star)
```

Implementación A.16: Recocido Simulado

GRASP

```
def greedy_f(P,S):
  (n,m)=P.shape
  v=np.zeros(n)
  Q=np.copy(P)
  for i in range(n):
```

```
    for j in range(m):
      if Q[i,j] in S:
        v[i]=1
        break
  values = sum(v)
  return values
```

Implementación A.17: Función de evaluación glotona.

```
def Candidates(p,S):
  id_plus = lambda x : x
  id_minus = lambda x : -x

  Range = list(range(1,p[0]+1))

  for i in range(len(S)):
    Range.remove(np.abs(S[i]))

  L = [i(x) for x in Range for i in (id_plus, id_minus)]

  return L
```

Implementación A.18: Lista de candidatos

```
def SolutionFormat(p,S):

  s = np.array([1 if i+1 in S else 0 for i in range(p[0])])

  return s
```

Implementación A.19: Reparar la solución

```
def GreedyConstruction(P,p):
  S = []
  L = Candidates(p,S)
  GreedyValue = [greedy_f(P,[L[i]]) for i in range(len(L))]

  while len(L) != 0:
    Choosen = L[np.argmax(GreedyValue)]
    S.append(Choosen)
    L = Candidates(p,S)
    GreedyValue = [greedy_f(P,[L[i]]) for i in range(len(L))]

  s_star = SolutionFormat(p,S)

  return s_star
```

Implementación A.20: Construcción glotona

```
def RCL(P,L,S,alpha):

  S_copy = np.copy(S)
  GreedyValue = [greedy_f(P,[L[i]]) for i in range(len(L))]

  Max = np.max(GreedyValue)
  Min = np.min(GreedyValue)
  value = Max - alpha * (Max - Min)

  CandidatesRestrictedList = []
  for i in range(len(L)):
```

```
      if GreedyValue[i] > value:
        CandidatesRestrictedList.append(L[i])

  return CandidatesRestrictedList
```

Implementación A.21: Lista restringida de candidatos

```
def GreedyRandomizedConstruction(P,p,alpha):
  S = []
  L = Candidates(p,S)
  GreedyValue = [greedy_f(P,[L[i]]) for i in range(len(L))]

  while len(L) != 0:

    RCL_list = RCL(P,L,S,alpha)

    if len(RCL_list) != 0:
      index = random.randint(0,len(RCL_list))
      Choosen = RCL_list[index]
      S.append(Choosen)
      L = Candidates(p,S)

    else:
      GreedyValue = [greedy_f(P,[L[i]]) for i in range(len(L))]
      Choosen = L[np.argmax(GreedyValue)]
      S.append(Choosen)
      L = Candidates(p,S)

  s_star = SolutionFormat(p,S)

  return s_star
```

Implementación A.22: Construcción glotona aleatoreizada

```
def GRASP(P,p,alpha,seed,n,c,TIME):
  start_time = time.time()
  now_time = time.time()

  f_star = 0

  random.seed(seed)

  while now_time - start_time < TIME:

    s0 = GreedyRandomizedConstruction(P,p,alpha)

    V = Neighborhood(p,s0,n,c,seed)
    (f_s1,s1) = LocalSearch(P,V)

    if f_s1 > f_star:
      s_star = np.copy(s1)
      f_star = f_s1

    now_time = time.time()

  return (f_star,s_star)
```

Implementación A.23: GRASP

Vecindades Variables

```
def InitialSolution(p):
  s = random.randint(0,2,p[0])

  return s
```

Implementación A.24: Solución inicial

```
def Shaking(P,p,s_star,k,Sizes,seed):

  V = Neighborhood(p,s_star,Sizes[k][0],Sizes[k][1],seed)
  i = random.randint(0,V.shape[0])

  s_0 = V[i,:]

  return s_0
```

Implementación A.25: Perturbación

```
def VariableNeighborhoodSearch(P,p,Sizes,seed,TIME):
  start_time = time.time()
  now_time = time.time()


  random.seed(seed)

  s_star = InitialSolution(p)
  f_star = f(P,s_star)

  k_max = len(Sizes)
  k = 0

  while (k < k_max) and (now_time - start_time < TIME):
    s_0 = Shaking(P,p,s_star,k,Sizes,seed)
    (f_s1,s1) = LocalSearch(P,Neighborhood(p,s_star,Sizes[k][0],
   Sizes[k][1],seed))

    if f_s1 > f_star:
      s_star = np.copy(s1)
      f_star = f_s1

      k = 0
    else:

      k = k + 1


    now_time = time.time()

  return (f_star,s_star)
```

Implementación A.26: Vecindades variables

Algoritmos genéticos

```
def Initialization(P,p,size):

```

```
  Population = []
  for i in range(size):
    Population.append(random.randint(0,2,p[0]))
  Population=np.array(Population)

  return Population
```

Implementación A.27: Inicialización

```
def Selection(P,p,Population,size):

  a=np.arange(Population.shape[0])
  MatingPool=[]

  Values = [f(P,Population[i,:]) for i in range(Population.shape[0])
   ]
  sum = np.sum(Values)
  P = [value/sum for value in Values]

  for i in range(int(size/2)):
    MatingPool.append(random.choice(a,2,p))

  return MatingPool
```

Implementación A.28: Selección

```
  def Recombination(P,p,MatingPool,Population):
    Offspring=[]

    for pair in MatingPool:
      n=random.randint(p[0])
      son=np.append(Population[pair[0],0:n],Population[pair[1],n:p
   [0]])
      Offspring.append(son)

    return Offspring
```

Implementación A.29: Recombinación 0

```
def Recombination(P,p,MatingPool,Population):
  Offspring=[]
  for pair in MatingPool:
    n=random.randint(p[0])
    m=random.randint(n,p[0])
    son=np.append(Population[pair[0],0:n],Population[pair[1],n:m])
    son=np.append(son,Population[pair[0],m:p[0]])
    Offspring.append(son)
  return Offspring
```

Implementación A.30: Recombinación 1

```
def Mutation(P,p,Offspring):

  M=random.uniform(0,1,len(Offspring))
  M[M > 0.2]=0
  M[M != 0]=1

  for i in range(len(Offspring)):
    if M[i]==1:
      l=Offspring[i].shape[0]
```

```
10        n=random.randint(l)
11        Offspring[i][n]=1-Offspring[i][n]
12
13    return Offspring
```

Implementación A.31: Mutación

```
1  def Replacement(P,p,Population,Offspring,size):
2    new_population=[]
3    for i in range(size):
4      new_population.append(np.append(Population[i,:],f(P,Population[i
     ,:])))
5
6    new_population=np.array(new_population)
7    new_population = new_population[np.argsort(new_population[:, -1])]
8    new_population=np.delete(new_population,-1,1)
9    new_population=np.delete(new_population,range(int(size/2)),0)
10
11   for son in Offspring:
12     new_population=tuple(new_population)
13     new_population=np.vstack((new_population,son))
14
15   new_population=np.array(new_population)
16   return Population
```

Implementación A.32: Reemplazo

```
1  def GeneticAlgorithm(P,p,size,seed,TIME):
2
3    start_time = time.time()
4    now_time = time.time()
5
6    random.seed(seed)
7
8    Population = Initialization(P,p,size)
9
10
11   while now_time - start_time < TIME:
12
13
14     MatingPool = Selection(P,p,Population)
15
16     Offspring = Recombination(P,p,MatingPool,Population)
17
18     Offspring = Mutation(P,p,Offspring)
19
20     Population = Replacement(P,p,Population,Offspring)
21
22     now_time = time.time()
23
24   (f_star,s_star) = LocalSearch(P,Population)
25
26   return (f_star,s_star)
```

Implementación A.33: Algoritmos genéticos

Optimización por colonias de hormigas

```
def PheromonesInitialization(p):

  Pheromones = np.full((2*p[0]+1,2*p[0]+1),0.5)

  L = Candidates(p,[])

  for i in range(2*p[0]+1):
    for j in range(2*p[0]+1):

      if j > 0:
        Pheromones[0,j] = L[j-1]

      if i == j:
        Pheromones[i,j] = 0
      elif (i%2) == 0 and j == i-1:
        Pheromones[i,j] = 0
      elif (i%2) != 0 and j == i+1:
        Pheromones[i,j] = 0

    if i > 0:
      Pheromones[i,0] = L[i-1]


  return Pheromones
```

Implementación A.34: Inicialización de la matriz de feromonas.

```
def ProbabilitiesInitialization(p):
  Probabilities = np.full((2*p[0]+1,2*p[0]+1),1 / (p[0]-2))

  L = Candidates(p,[])

  for i in range(2*p[0]+1):
    for j in range(2*p[0]+1):

      if j > 0:
        Probabilities[0,j] = L[j-1]

      if i == j:
        Probabilities[i,j] = 0
      elif (i%2) == 0 and j == i-1:
        Probabilities[i,j] = 0
      elif (i%2) != 0 and j == i+1:
        Probabilities[i,j] = 0

    if i > 0:
      Probabilities[i,0] = L[i-1]
  return Probabilities
```

Implementación A.35: Inicialización de la matriz de probabilidades

```
def Choose(p,S,L,Probabilities):

  ALLCandidates = Probabilities[:,0]
  ActualPosition = np.where(ALLCandidates == S[-1])[0]
  RowOfProbabilities = Probabilities[ActualPosition,:]

  ProbabilitiesL = []

```

```
  for i in range(len(L)):
    index = np.where(ALLCandidates == L[i])[0][0]
    ProbabilitiesL.append(RowOfProbabilities[0,index])

  Sum = np.sum(ProbabilitiesL)

  NormalizedProbabilities = [p/Sum for p in ProbabilitiesL]

  Choosen = random.choice(L,p = NormalizedProbabilities, size = 1)
   [0]

  return Choosen
```

Implementación A.36: Selección de candidatos

```
def PheromonesIntensification(Pheromones,BestWalk,delta):
  ALLCandidates = Pheromones[:,0]
  for i in range(len(BestWalk)-1):
    index0 = np.where(ALLCandidates == BestWalk[i])[0][0]
    index1 = np.where(ALLCandidates == BestWalk[i+1])[0][0]

    Pheromones[index0,index1] = Pheromones[index0,index1] + delta

  return Pheromones
```

Implementación A.37: Intensificación de feromonas

```
def ProbabilitiesMatrix(Probabilities,Pheromones):
  for i in range(1,Probabilities.shape[0]):
    Sum = np.sum(Pheromones[i,1:])
    for j in range(1,Probabilities.shape[1]):
      Probabilities[i,j] = Pheromones[i,j] / Sum

  return Probabilities
```

Implementación A.38: Actualizar matriz de feromonas

```
def AntColonyOptimization(P,p,k,rho,delta,seed,TIME):

  start_time = time.time()
  now_time = time.time()

  random.seed(seed)

  Pheromones = PheromonesInitialization(p)

  Probabilities = ProbabilitiesInitialization(p)

  random.seed(seed)

  while now_time - start_time < TIME:

    Solutions = []
    Walks = []
    for i in range(5):
      S = []
      L = Candidates(p,S)

      S.append(L[random.randint(len(L))])
```

```
23
24      while len(L) != 0:
25        Choosen = Choose(p,S,L,Probabilities)
26        S.append(Choosen)
27        L = Candidates(p,S)
28
29      s = SolutionFormat(p,S)
30      Solutions.append(s)
31      Walks.append(S)
32
33    Values = [f(P,s) for s in Solutions]
34
35    BestWalk = Walks[Values.index(max(Values))]
36
37    s_star = Solutions[Values.index(max(Values))]
38    f_star = max(Values)
39
40    for i in range(1,Pheromones.shape[0]):
41      for j in range(1,Pheromones.shape[1]):
42        Pheromones[i,j] = Pheromones[i,j] * (1-rho)
43
44
45    Pheromones = PheromonesIntensification(Pheromones,BestWalk,delta
   )
46
47    Probabilities = ProbabilitiesMatrix(Probabilities,Pheromones)
48
49    now_time = time.time()
50
51
52  return (f_star,s_star)
```

Implementación A.39: Optimización por colonias de hormigas

Índice de figuras

1.1. Instancia del problema del agente viajero [1]. 3
1.2. Solución glotona a una instancia del problema del agente viajero [1]. 4
1.3. Solución óptima a una instancia del problema del agente viajero [1]. 4
1.4. Otra instancia del problema del agente viajero [56]. 5

2.1. Instancia de un problema de optimización [53]. 7
2.2. Ejemplo de instancia del problema de la mochila.[11] 8
2.3. Ejemplo de gráfica y de posibles clans [39]. 9
2.4. Ejemplo de gráfica que tiene un 10-clan [35] 9

3.1. Fotografía de calculadoras humanas en la NASA. [23] 17
3.2. Imagen de una máquina de Turing de 3 cintas. [24] 18
3.3. Ejemplo de función de transición de una máquina de Turing de dos cintas. [24] 19
3.4. Interpretación gráfica de una máquina de Turing determinista y una no determinista. [24] 23
3.5. Ilustración de un cuadro. [51] 26

4.1. Alfiler en un pajar 28
4.2. Algoritmo de búsqueda local básica. 30
4.3. Diagrama de búsqueda tabú. 33
4.4. Diagrama de Recocido Simulado. 36
4.5. Diagrama de GRASP. 38
4.6. Diagrama de Búsqueda de Vecindad Variable. 40
4.7. Diagrama de Algoritmos Genéticos. 43
4.8. Diagrama de Sistemas de Hormigas. 45

5.1. Sudoku a medio resolver [58] 46
5.2. [50] 48
5.3. Instancia del problema del SAT [50] 49
5.4. Primera iteración del DPLL. 50
5.5. Primeras cuatro iteraciones del algoritmo DPLL. 50
5.6. Siguientes cuatro iteraciones del algoritmo DPLL. 51
5.7. Siguientes tres iteraciones del algoritmo DPLL. 51
5.8. Primeras tres iteraciones del algoritmo DPLL con aprendizaje de cláusulas. 54
5.9. Primeras dos iteraciones de la construcción de la gráfica de implicación . . . 55
5.10. Representación de la primera parte de la tercera iteración de la construcción de la gráfica. 55
5.11. Representación de la segunda parte de la tercera iteración de la construcción de la gráfica. 56
5.12. Cuarta iteración de la construcción de la gráfica. 56
5.13. Gráfica de implicación para la cuarta iteración. 56

5.14. Ejemplo de corte para la gráfica de implicación. 57
5.15. Otro ejemplo de corte para la gráfica de implicación. 57

6.1. Tiempo que le tomó a recocido simulado alcanzar el óptimo. 64
6.2. Valor en la función objetivo a lo largo de recocido simulado. 65
6.3. Valores en la función objetivo para las soluciones construidas en GRASP en una misma instancia. 66
6.4. Número de veces que se corrió la heurística sobre una misma solución para la instancia p100. 67
6.5. Valor promedio del rendimiento de la población a lo largo de la heurística de algoritmos genéticos. 68

Lista de implementaciones

A.1. Cargar instancias en formato cnf 74
A.2. Función objetivo 74
A.3. Estructura de vecindad 74
A.4. Cargar instancias en el formato gráfico col y hal 75
A.5. Codificador del problema del número cromático 75
A.6. Funciones adicionales para obtener una paleta de colores 76
A.7. Decodificador del problema del número cromático 76
A.8. Codificador del problema del ciclo Hamiltoniano 76
A.9. Decodificador del problema del ciclo Hamiltoniano 77
A.10. Codificador del problema de las n-reinas 78
A.11. Decodificador del problema de las n-reinas 79
A.12. Búsqueda local 79
A.13. Escalar las colinas 80
A.14. Limpiar la vecindad 80
A.15. Búsqueda Tabú 80
A.16. Recocido Simulado 81
A.17. Función de evaluación glotona. 81
A.18. Lista de candidatos 82
A.19. Reparar la solución 82
A.20. Construcción glotona 82
A.21. Lista restringida de candidatos 82
A.22. Construcción glotona aleatoreizada 83
A.23. GRASP 83
A.24. Solución inicial 84
A.25. Perturbación 84
A.26. Vecindades variables 84
A.27. Inicialización 84
A.28. Selección 85
A.29. Recombinación 0 85
A.30. Recombinación 1 85
A.31. Mutación 85
A.32. Reemplazo 86
A.33. Algoritmos genéticos 86
A.34. Inicialización de la matriz de feromonas. 87
A.35. Inicialización de la matriz de probabilidades 87
A.36. Selección de candidatos 87
A.37. Intensificación de feromonas 88
A.38. Actualizar matriz de feromonas 88
A.39. Optimización por colonias de hormigas 88

Índice de tablas

2.1. Ejemplo de tabla de verdad para una fórmula proposicional en general. . . . 12

6.1. Descripción de un conjunto de cinco instancias de [31]. 61
6.2. Tiempo que tardaría resolver cada instancia por fuerza bruta. 62
6.3. Mejores valores obtenidos en la heurística de escalar colinas. 63
6.4. Mejores valores obtenidos en la heurística de búsqueda tabú. 63
6.5. Mejores valores obtenidos en la heurística de algoritmos genéticos. 68
6.6. Mejores valores obtenidos en la heurística de optimización por colonias de hormigas. 69
6.7. Tiempo que tardó GLUCOSE en encontrar el óptimo global. 69
6.8. Desviación porcentual de los mejores valores obtenidos usando todas las heurísticas. 70

Bibliografía

[1] YouTube, ago. de 2021. URL: https://www.youtube.com/watch?v=FfN3xZIms5c.

[2] Emile Aarts, Emile HL Aarts y Jan Karel Lenstra. *Local search in combinatorial optimization*. Princeton University Press, 2003.

[3] Israel Agbehadji. «SOLUTION TO THE TRAVEL SALESMAN PROBLEM, USING OMICRON GENETIC ALGORITHM. CASE STUDY: TOUR OF NATIONAL HEALTH INSURANCE SCHEMES IN THE BRONG AHAFO REGION OF GHANA». Tesis doct. Jun. de 2011. DOI: 10.13140/RG.2.1.2322.7281.

[4] Gilles Audemard y Laurent Simon. «Glucose: a solver that predicts learnt clauses quality». En: *SAT Competition* (2009), págs. 7-8.

[5] Gilles Audemard y Laurent Simon. «On the glucose SAT solver». En: *International Journal on Artificial Intelligence Tools* 27.01 (2018), pág. 1840001.

[6] Christian Blum y Andrea Roli. «Metaheuristics in combinatorial optimization: Overview and conceptual comparison». En: *ACM computing surveys (CSUR)* 35.3 (2003), págs. 268-308.

[7] Erick Cantu Paz. *Efficient and accurate parallel genetic algorithms*. Vol. 1. Springer Science & Business Media, 2000.

[8] James A Carlson, Arthur Jaffe y Andrew Wiles. *The millennium prize problems*. Citeseer, 2006.

[9] Vladimir Černy. «Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm». En: *Journal of optimization theory and applications* 45.1 (1985), págs. 41-51.

[10] Alex Chitu. *Google Maps Shows Funny Directions*. http://googlesystem.blogspot.com/2007/03/gc maps-shows-funny-directions.html. [Online; 7-nov-2021]. 2007.

[11] Wikimedia Commons. *Knapsack.svg*. 2007. URL: https://commons.wikimedia.org/wiki/File:Knapsack.svg.

[12] Stephen A Cook. «The complexity of theorem-proving procedures». En: *Proceedings of the third annual ACM symposium on Theory of computing*. 1971, págs. 151-158.

[13] Martin Davis, George Logemann y Donald Loveland. «A Machine Program for Theorem-Proving». En: *Commun. ACM* 5.7 (jul. de 1962), págs. 394-397. ISSN: 0001-0782. DOI: 10.1145/368273.368557. URL: https://doi.org/10.1145/368273.368557.

[14] Martin Davis e Hilary Putnam. «A computing procedure for quantification theory». En: *Journal of the ACM (JACM)* 7.3 (1960), págs. 201-215.

[15] DIMACS. *Clique and Coloring Problems Graph Format*. 1993. URL: https://mat.tepper.cmu.edu/COLOR/instances.html.

[16] DIMACS. *Satisfiability Suggested Format*. 1993. URL: https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html.

[17] Marco Dorigo. «Optimization, learning and natural algorithms». En: *Ph. D. Thesis, Politecnico di Milano* (1992).

[18] Dingzhu Du, Jun Gu, Panos M Pardalos y col. *Satisfiability problem: theory and applications: DIMACS Workshop, March 11-13, 1996*. Vol. 35. American Mathematical Soc., 1997.

[19] Niklas Eén y Niklas Sörensson. «An extensible SAT-solver». En: *International conference on theory and applications of satisfiability testing*. Springer. 2003, págs. 502-518.

[20] Herbert B Enderton. *A mathematical introduction to logic*. Elsevier, 2001.

[21] Thomas A Feo y Mauricio GC Resende. «A probabilistic heuristic for a computationally difficult set covering problem». En: *Operations research letters* 8.2 (1989), págs. 67-71.

[22] Thomas A Feo y Mauricio GC Resende. «Greedy randomized adaptive search procedures». En: *Journal of global optimization* 6.2 (1995), págs. 109-133.

[23] Margalit Fox. *Katherine Johnson dies at 101; mathematician broke barriers at NASA*. Feb. de 2020. URL: https://www.nytimes.com/2020/02/24/science/katherine-johnson-dead.html.

[24] Michael R Garey y David S Johnson. *Computers and intractability*. Vol. 174. freeman San Francisco, 1979.

[25] Michel Gendreau, Alain Hertz y Gilbert Laporte. «A tabu search heuristic for the vehicle routing problem». En: *Management science* 40.10 (1994), págs. 1276-1290.

[26] Fred Glover. «Future paths for integer programming and links to artificial intelligence». En: *Computers & operations research* 13.5 (1986), págs. 533-549.

[27] David E Goldberg y col. *The Design of Innovation: Lessons from and for Competent Genetic Algorithmsby David E. Goldberg*. Vol. 7. Springer Science & Business Media, 2002.

[28] Vincent Granville, Mirko Krivánek y J-P Rasson. «Simulated annealing: A proof of convergence». En: *IEEE transactions on pattern analysis and machine intelligence* 16.6 (1994), págs. 652-656.

[29] Pierre Hansen y Nenad Mladenović. «An introduction to variable neighborhood search». En: *Meta-heuristics*. Springer, 1999, págs. 433-458.

[30] Alain Hertz y Dominique de Werra. «The tabu search metaheuristic: how we used it». En: *Annals of mathematics and artificial intelligence* 1.1 (1990), págs. 111-121.

[31] Holger H. Hoos. *SAT Benchmarks*. URL: https://www.cs.ubc.ca/hoos/SATLIB/benchm.html.

[32] Alexey Ignatiev, Antonio Morgado y Joao Marques-Silva. «PySAT: A Python Toolkit for Prototyping with SAT Oracles». En: *SAT*. 2018, págs. 428-437. DOI: 10.1007/978-3-319-94144-8_26. URL: https://doi.org/10.1007/978-3-319-94144-8_26.

[33] Richard M Karp. «Reducibility among combinatorial problems». En: *Complexity of computer computations*. Springer, 1972, págs. 85-103.

[34] JB Kirkpatrick. «An iterative method for establishing priorities for the selection of nature reserves: an example from Tasmania». En: *Biological Conservation* 25.2 (1983), págs. 127-134.

[35] A. Lagos y G.P. Papavassilopoulos. «Network topology design to influence the effects of manipulative behaviors in a social choice procedure». Tesis doct. Mar. de 2019.

[36] Sofianto Lee y Raymond Lister. «Experiments in the dynamics of phase coupled oscillators when applied to graph colouring.» En: ene. de 2008, págs. 83-89. DOI: 10.1145/1378279.1378295.

[37] João P Marques Silva y Karem A Sakallah. «GRASP—a new search algorithm for satisfiability». En: *The Best of ICCAD*. Springer, 2003, págs. 73-89.

[38] Joao P Marques-Silva y Karem A Sakallah. «GRASP: A search algorithm for propositional satisfiability». En: *IEEE Transactions on Computers* 48.5 (1999), págs. 506-521.

[39] *Maximum clique problem: Linear Programming Approach*. Feb. de 2019. URL: https://kmutya.github.io/maxclique/.

[40] Nicholas Metropolis y col. «Equation of state calculations by fast computing machines». En: *The journal of chemical physics* 21.6 (1953), págs. 1087-1092.

[41] Wil Michiels, Emile Aarts y Jan Korst. *Theoretical aspects of local search*. Springer Science & Business Media, 2007.

[42] Matthew W Moskewicz y col. «Chaff: Engineering an efficient SAT solver». En: *Proceedings of the 38th annual Design Automation Conference*. 2001, págs. 530-535.

[43] Artur Niewiadomski y col. «Applying modern sat-solvers to solving hard problems». En: *Fundamenta Informaticae* 165.3-4 (2019), págs. 321-344.

[44] Nils J Nilsson. «Principles of artificial intelligence, reprint». En: *Palo Alto: Morgan Kaufmann* (1980).

[45] Ibrahim H Osman y Gilbert Laporte. *Metaheuristics: A bibliography*. 1996.

[46] Christos H Papadimitriou y Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.

[47] Colin Reeves. «Modern heuristics techniques for combinatorial problems». En: *Nikkan Kogyo Shimbun* (1997).

[48] Gerhard Reinelt. *TSPLIB95*. 1995. URL: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/.

[49] Ryan A. Rossi y col. «Fast Maximum Clique Algorithms for Large Graphs». En: *Proceedings of the 23rd International Conference on World Wide Web*. WWW '14 Companion. Seoul, Korea: Association for Computing Machinery, 2014, págs. 365-366. ISBN: 9781450327459. DOI: 10.1145/2567948.2577283. URL: https://doi.org/10.1145/2567948.2577283.

[50] Olivier ROUSSEL. *The Sat Game*. URL: http://www.cril.univ-artois.fr/~roussel/satgame/satgame.php?level=5&lang=eng.

[51] M Sipser. «Introduction To Theory Of Computation by Micheal Sipser MIT». En: *Boston, Massachusetts: Thomson Course Technology* (2004).

[52] Jadranka Skorin-Kapov. «Tabu search applied to the quadratic assignment problem». En: *ORSA Journal on computing* 2.1 (1990), págs. 33-45.

[53] Study.com. *By cutting away identical squares from each corner of a rectangular piece of cardboard*. 2020. URL: https://study.com/academy/answer/by-cutting-away-identical-squares-from-each-corner-of-a-rectangular-piece-of-cardboard-and-folding-up-the-resulting-flaps-an-open-box-may-be-made-if-the-cardboard-is-15-rm-in-long-and-6-rm-in-wide-find-the-dimensions-of-the-box-that-will-yield-th.html.

[54] Thomas Stützle. «Local search algorithms for combinatorial problems: analysis, improvements, and new applications». En: (1999).

[55] Eric Taillard. «Some efficient heuristic methods for the flow shop sequencing problem». En: *European journal of Operational research* 47.1 (1990), págs. 65-74.

[56] M Thilaga y col. «Shortest path based network analysis to characterize cognitive load states of human brain using EEG based functional brain networks». En: *Journal of integrative neuroscience* 17.2 (2018), págs. 253-275.

[57] Michael Trick. *Graph based on Mycielski transformation. Triangle free (clique number 2) but increasing coloring number.* URL: https://mat.tepper.cmu.edu/COLOR02/.

[58] Vampipe. Abr. de 2022. URL: https://twitter.com/vampipe/status/1514697445917265924?s=20&t=3PrL5byVIUVpEVoVstEXQQ.

[59] Stefan Voß y col. *Meta-heuristics: Advances and trends in local search paradigms for optimization.* Springer Science & Business Media, 2012.

[60] Wikipedia. *List of NP-complete problems.* 2021. URL: https://en.wikipedia.org/wiki/List_of_NP-complete_problems.