Benemérita Universidad Autónoma de Puebla

Facultad de Ciencias de la Computación

Maestría en Ciencias de la Computación

Determinación de Eficiencia y Factor de Garantía de un Algoritmo Basado en Búsqueda Local para el Tratamiento del Problema MaxSAT

Tesis que para obtener el grado de

Maestro en Ciencias de la Computación

presenta:

David Eduardo Pinto Avendaño

Asesor: Dr. Guillermo De Ita Luna

Con todo mi amor para mi querida esposa...
***Sofía*.**

Tú que en todo momento has apoyado cada uno
de mis pasos y me has hecho ser lo que ahora soy,
porque sin ti, definitivamente no lo hubiese logrado.

A mis padres y hermanos...
Celso Pinto, Isabel Avendaño y Celso, Alejandro y Mauricio

Ya que siempre me han alentado
a terminar mis metas.

A mi tía *Fanny*...

Fundamento principal
de mi carrera profesional.

A mi asesor...
Dr. Guillermo De Ita Luna

Por todo el apoyo recibido y
el conocimiento transmitido de
manera desinteresada.

Resumen

En esta tesis, estudiamos el diseño y análisis de algoritmos para un problema de satisfactibilidad boolena bajo un dominio discreto. El problema de *Máxima Satisfactibilidad* es un problema central en la demostración automática de teoremas y en la teoría de complejidad computacional, por lo que existe un gran interés en la búsqueda de algoritmos eficientes para la resolución de éste.

Ampliamente hablando, nuestro trabajo está compuesto de tres partes. En la primera parte, desarrollamos una serie de algoritmos y heurísticas (basados en búsqueda local), que combinados con las técnicas de búsqueda Tabú y una nueva propuesta para el control de reinicio de búsqueda, denominada antipodales, dan como resultado una nueva propuesta algorítmica a la que denominamos *Búsqueda Tabú No Obvia con Antipodales* (LS-NTA). Se propone una nueva función objetivo (resultado del análisis de la función objetivo no obvia propuesta por Khanna [KhS96]) para LS-NTA.

En la segunda parte, mostramos que la función objetivo obtiene un factor de garantía mejor que en la función objetivo no obvia propuesta por Khanna, esto para el tipo de instancias que cumplen que $W(S_1) \leq p \cdot W(S_0)$, $p \in \Re^{+}$.

Finalmente, en la última parte de este trabajo, corroboramos los resultados (obtenidos matemáticamente), mediante un análisis empírico documentado a través de tablas y gráficas.

Los resultados obtenidos se consideran satisfactorios, ya que el hecho de que MaxSAT sea un problema representativo de su clase de complejidad (APX), implica que las mejoras encontradas a los algoritmos que lo resuelven, pueden transformarse en mejoras de algoritmos que resuelvan los demás problemas de su clase de complejidad.

INDICE

Introducción — *1*

Capítulo I

Fundamentos teóricos — *3*

1.1 Máquinas de Turing — **3**

1.1.1 Máquina de Turing determinista — 3

1.1.2 Máquina de Turing no determinista — 7

1.1.3 Clases de complejidad definidas por las máquinas de Turing — 8

1.2 Teoría de los problemas NP-Completos — **12**

1.2.1 Caracterización sintáctica de los problemas en *NP* — 13

1.2.2 Problemas de optimización en NP — 14

1.2.3 Vista computacional de los problemas de optimización — 16

1.3 Problemas SAT y MaxSAT — **17**

Capítulo II

Propuestas algorítmicas para la resolución del problema MaxSAT — *20*

2.1 Técnica de búsqueda local — **20**

2.2 Búsqueda local aplicada al problema SAT y MaxSAT — **21**

2.3 Búsqueda local no obvia — **23**

2.4 Función objetivo no obvia propuesta — **24**

2.5 Búsqueda tabú — **25**

2.5.1 Búsqueda tabú no obvia con antipodales — 27

Capítulo III

Elementos para el análisis de algoritmos — *30*

3.1 Introducción — **30**

3.2 Métodos para determinar el factor de garantía de los algoritmos — **33**

3.3 Modelos para construir instancias de fórmulas booleanas — **35**

Capítulo IV

Determinación del factor de garantía y análisis experimental — *38*

4.1 Determinación del factor de garantía — **38**

4.2 Análisis experimental — **46**

Conclusiones — *62*

Bibliografía — *65*

Lista de acronismos — *68*

Introducción

La aparición de una gran cantidad de problemas en los últimos años que han demostrado estar dentro de la categoría de problemas denominados NP (los cuales son reconocidos por requerir de una gran cantidad de recursos y tiempo para que su resolución se cumpla) ha motivado el buscar algoritmos eficientes para su resolución; sin embargo, a menos que P = NP, no habrá algoritmos de tiempo polinomial para solucionar tales problemas.

En este trabajo se presentan diversas heurísticas para resolver eficientemente, aunque de manera aproximada dos problemas clásicos NP-Completos: SAT y MaxSAT.

SAT y MaxSAT son dos problemas cruciales para el desarrollo de la demostración automática de teoremas (DAT) en el cálculo proposicional, su planteamiento puede resumirse en la forma siguiente:

Dada una fórmula booleana F en forma normal conjuntiva (FNC), el problema SAT consiste en decidir si F es satisfactible. El problema de optimización MaxSAT consiste en determinar el número máximo de cláusulas que pueden satisfacerse simultáneamente.

A la fecha, como para todo problema NP-Completo, no se ha construido algoritmo eficiente que resuelva a SAT y/o MaxSAT. Una alternativa en la búsqueda de resolver eficientemente los problemas de optimización es el desarrollo de algoritmos de aproximación, es decir, algoritmos que trabajando dentro de cotas polinomiales de tiempo encuentren soluciones cercanas al valor óptimo, y especialmente que garanticen que las soluciones halladas se encuentran dentro de un factor multiplicativo de la solución óptima.

En este trabajo, se exploraron las técnicas de búsqueda local como algoritmos de aproximación para resolver MaxSAT. Es conocido que el factor de garantía de una búsqueda local para MaxSAT es de $k/k+1$, donde k es el número máximo de literales que hay en alguna cláusula de la fórmula booleana de entrada [HaP90]. Khanna propone un procedimiento de búsqueda local con una nueva función objetivo llamada función no obvia que obtiene un factor de garantía de $(2^k-1)/2^k$, siendo k, al igual que el caso anterior, el número máximo de literales que hay en alguna cláusula de la fórmula booleana de entrada [KhS96].

Como resultado de nuestras investigaciones, proponemos una nueva función objetivo que utilizada dentro del procedimiento de búsqueda local obtiene un mejor factor de garantía que el de la función objetivo propuesta por Khanna, esto para cierta clase de fórmulas booleanas. En este documento presentamos la demostración matemática para la obtención de este nuevo factor de garantía y corroboramos el resultado teórico a través de un análisis empírico sobre un universo de fórmulas booleanas generadas aleatoriamente.

Adicionalmente se diseñaron dos estrategias, una basada en una arreglo tabú para acelerar la búsqueda de óptimos locales y la otra basada en el uso de elementos antipodales como procedimiento para re-iniciar la búsqueda después de arribar a óptimos locales. Ambas estrategias se implementaron a la búsqueda local basada en la función objetivo que propusimos. El algoritmo resultante, según el análisis empírico, obtuvo mejores resultados dentro de tiempos comparables con los demás algoritmos probados.

La organización del documento es la siguiente: En el capítulo I se revisa una serie de conceptos básicos que son el preámbulo teórico del trabajo, estos conceptos incluyen: máquinas de Turing, clases de complejidad, teoría de los problemas NP-Completos, y una descripción acerca de los problemas SAT y MaxSAT. Posteriormente, en el capítulo II se presentan diversas técnicas de aproximación basadas principalmente en la búsqueda local, en el capítulo III se revisan los elementos necesarios para el análisis de algoritmos; en el capítulo IV se determina la eficiencia y factor de garantía del algoritmo propuesto, además, se realiza el análisis de los algoritmos mediante pruebas computacionales, tablas y gráficas comparativas. Por último se presentan las conclusiones obtenidas de la investigación y se proponen diferentes trabajos a futuro en esta línea del diseño y análisis de algoritmos de aproximación para MaxSAT.

Capítulo I

Fundamentos teóricos

1.1 Máquinas de Turing

Alan Turing [TuA36] desarrolló un modelo matemático simple conocido con el nombre de máquina de Turing; este modelo matemático ha servido como base para poder expresar en términos sencillos lo que un mecanismo de computación puede o no hacer. En la actualidad, es muy común decir que cualquier problema que es resoluble bajo una máquina de Turing será resoluble también por cualquier otro mecanismo de cómputo, e inversamente, problemas que no pueden resolverse por máquinas de Turing tampoco serán resueltos por cualquier otro mecanismo de cómputo. Es importante entonces revisar el funcionamiento de este modelo matemático que parece ser sencillo y a la vez muy poderoso.

Se definen diferentes máquinas de Turing de acuerdo a la manera en que trabajan, así, existen máquinas de Turing deterministas y máquinas de Turing no deterministas, cada una de éstas está orientada a resolver un tipo especial de problemas.

1.1.1 Máquina de Turing determinista

Las máquinas de Turing deterministas (MTD) constan de un control finito, una cabeza de lecto-escritura y una cinta dividida en un número infinito de celdas; dichas celdas son etiquetadas usando números enteros. Una representación esquemática de una máquina de Turing determinista puede observarse en la figura 1.1.

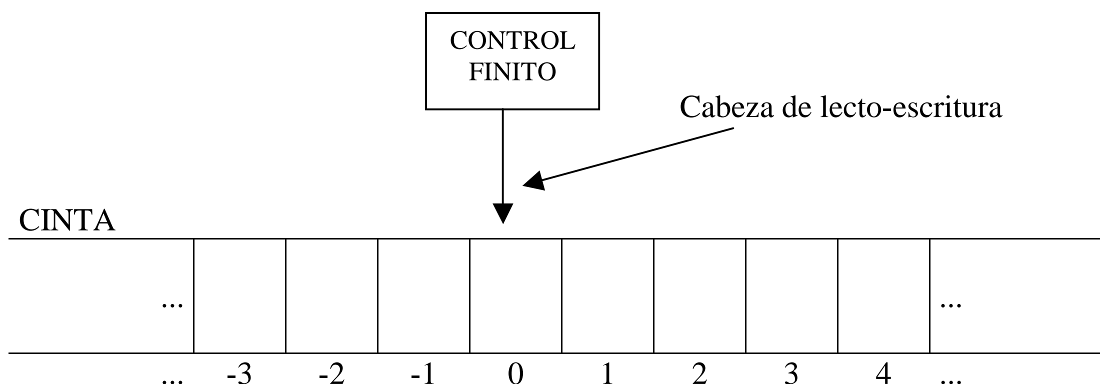

Figura 1.1 Representación esquemática de una máquina de Turing determinista

De manera formal, se puede definir una máquina de Turing determinista M por un sistema de la forma: M=<Q, Σ, t, q_0, F>, donde:

- Q: es un conjunto finito de estados; cada uno de tales elementos indica un estado en el que puede estar el control finito. Este conjunto de estados incluye dos estados distinguidos de parada: q_y y q_n y el estado de inicio q_0.
- q_0: es el estado distinguido en el que arranca la máquina de Turing.
- Σ: es el alfabeto de símbolos. Tales símbolos son usados en la codificación de la cadena de entrada así como para codificar los símbolos que pueden colocarse en la cinta de la máquina de Turing.
- F: subconjunto de Q, $F=\{q_y, q_n\}$ que denota a los estados finales o de paro de la máquina de Turing.
- t: es la función de transición, $t:(Q-F)\times \Sigma\rightarrow Q\times\Sigma\times\{-1, 0, 1\}$. La función de transición codifica al programa que va a ser ejecutado por la máquina de Turing.

Dado un alfabeto Σ, se define a una *cadena* x como una secuencia finita de símbolos, donde cada símbolo que conforma a la cadena es un elemento del alfabeto. Los términos *sentencia* y *palabra* son utilizados como sinónimos de cadena.

La longitud de una cadena x, $|x|$, es el número de símbolos que conforman a tal cadena. La cadena vacía, que denotaremos por λ, es una cadena especial de longitud cero. Sean x, y cadenas, la *concatenación* de x y y, que denotaremos por xy, es la cadena resultante de concatenar y a x. En este sentido, la cadena vacía es el elemento identidad bajo la operación de concatenación.

Sea Σ un alfabeto, se tiene que:
$\Sigma^0 = \{\lambda\}$ – es el conjunto que contiene sólo a la cadena vacía.
$\Sigma^1 = \Sigma$ – es el alfabeto mismo.
$\Sigma^i = \Sigma^{i-1}\Sigma$ – es Σ concatenado consigo mismo i-1 veces.
$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup\ldots$ – es el conjunto de cadenas de cualquier longitud que pueden formarse con elementos de Σ.

A Σ^* se le llama el lenguaje derivado (o diccionario) del alfabeto Σ. En este sentido, un lenguaje es un conjunto de cadenas definidas sobre el alfabeto.

El funcionamiento de una máquina de Turing determinista es simple y se describe a continuación:

Inicialmente la máquina de Turing M tiene al símbolo b en cada una de las celdas de su cinta, excepto para un cierto número de casillas en donde se encuentra escrita la cadena x, llamada la cadena de entrada, $x \in \Sigma^*$. Esta cadena x se coloca en las casillas numeradas de izquierda a derecha a partir de la celda 1 y hasta la longitud de la cadena de entrada $|x|$, habiendo un símbolo de la cadena de entrada por celda. La cabeza de lecto-escritura de la máquina de Turing se coloca inicialmente en el símbolo más a la izquierda de la cadena de entrada y el estado del control finito se encuentra en el estado inicial q_0.

En cada instante, el comportamiento de la máquina de Turing se rige por el programa codificado a través de la función de transición t, si M lee el símbolo $a_j \in \Sigma$, el control finito está en el estado $q_i \in Q$ y se tiene definida t para q_i y a_j, por ejemplo: $t(q_i,a_j)=(q_{ij}, a_{ij}, m_{ij})$, significa entonces que se sustituye en la cinta el símbolo a_i por el a_{ij}, el control finito pasa al estado q_{ij}, la cabeza de lecto-escritura pasa a examinar la celda que está a su izquierda o no se mueve o examina a la celda que está a su derecha, según el valor de m_{ij} (-1, 0,1) respectivamente.

Se define una descripción instantánea como una palabra XqY, donde X e Y están en Σ^*, y q es un estado de Q. XqY significa que en la cinta está escrita la palabra XY, el control de la máquina de Turing está en el estado q, y su cabeza de lecto-escritura se encuentra examinando al primer símbolo de Y. Un paso de cómputo de una máquina de Turing se denota por el par ordenado (d_1, d_2) y suele escribirse $d_1 \xrightarrow{\text{pasa}} d_2$, donde d_1 y d_2 son descripciones instantáneas de M. Tal paso de cómputo denota que en un solo paso, la máquina de Turing pasa de la descripción instantánea d_1 a la d_2. La relación *pasa* es llamada también un paso de cómputo. Formalmente significa que si $d_1 = XqY$, $d_2 = X'q'Y'$, y supóngase que $X = x_1x_2\ldots x_l$, $Y = y_1y_2\ldots y_k$ y que está definida la transición $t(q, y_1) = (p,\lambda,m)$.

Entonces se tiene $q' = p$ y tres posibles casos según el valor de m:
Si $m = -1$ entonces $X' = x_1\ldots x_{l-1}$, $Y'=x_l\lambda y_2\ldots y_k$.
Si $m = 0$ entonces $X' = X$, $Y'=\lambda y_2\ldots y_k$.
Si $m = 1$ entonces $X' = x_1\ldots x_l\lambda$, $Y'= y_2\ldots y_k$.

Una vez definida la relación $\xrightarrow{\text{pasa}}$ entre configuraciones, se define la relación $\xrightarrow{\text{sigue}}$ (sigue) como su cerradura transitiva. Entonces, $d_1 \xrightarrow{\text{sigue}} d_2$ denota que d_2 sigue de d_1 por uno o más pasos de cómputo.

A una sucesión de descripciones instantáneas relacionadas mediante pasos de cómputo se le llama una *computación*. Una computación que lleva a que la máquina de Turing acepte la entrada x, es una secuencia C_0, C_1, ..., C_t de pasos de cómputo, tal que $C_0=q_0x$ es la descripción instantánea de arranque, y para $0 \le i < t$, $C_i \xrightarrow{\text{pasa}} C_{i+1}$, q en C_i, q no es un estado final, y el estado q en C_t es el estado final q_y.

El *lenguaje aceptado* por una máquina de Turing, representado por L_M es el conjunto de palabras $x \in \Sigma^*$, que hacen que la máquina de Turing llegue al estado de aceptación q_y, cuando inicia con x en su cinta de entrada:

$$L_M = \{x \in \Sigma^* \mid q_0x \xrightarrow{\text{sigue}} uq_y v, \quad u, v \in \Sigma^*, q_y \in F\}$$

Todas las cadenas que pertenezcan al lenguaje aceptado por la máquina de Turing hacen suponer que harán detenerse a la máquina en algún momento, sin embargo, para aquellas cadenas que no pertenezcan al lenguaje, no es posible determinar si la máquina se detendrá en algún momento o no. Obviamente, para que un programa de la máquina de

Turing sea un algoritmo, es necesario que posea una condición de paro al procesar cualquier cadena del alfabeto de entrada.

Las definiciones anteriores están orientadas al reconocimiento de un lenguaje, sin embargo, se puede corresponder dichas definiciones a la resolución de problemas, especialmente hablando de problemas de decisión. Un problema de decisión (PD) se plantea como una pregunta general que acepta como respuesta sólo una de dos opciones: la respuesta '*SI*' o la respuesta '*NO*'. En forma abstracta, un problema de decisión es una estructura del tipo PD:$<D, S>$, donde D es el dominio de instancias del problema y $S \subseteq D$ es el conjunto de aquellas que lo resuelven afirmativamente.

El planteamiento del PD es:

$$\forall x \in D : \text{PD}(x) = \begin{cases} SI & \text{si } x \in S \\ NO & \text{en otro caso} \end{cases}$$

En estos términos, se dice que una máquina de Turing M resuelve el problema de decisión PD:$<D, S>$ si M se detiene para toda x en D y $L_M = S$ (L_M es el lenguaje reconocido por M). Usando el modelo de máquinas de Turing, se dice que el espacio de una computación de aceptación es el número de casillas utilizadas por la máquina de Turing durante el cómputo de aceptación y el tiempo es el número de pasos de computación que realiza la máquina de Turing durante el cómputo de aceptación.

Sea F:$\mathbf{N} \rightarrow \Re$ una función y M una máquina de Turing, se dice que M acepta su entrada en tiempo (espacio) $F(n)$ si para cada x en L_M hay una computación que lleva a aceptación de M sobre x tal que el tiempo (espacio) de la computación de aceptación no excede $F(n)$, con $n = long(x)$, donde $long(x)$ es la *longitud de la entrada x*.

En términos de problemas de decisión, la complejidad en tiempo de M se plantea de la siguiente manera: PD:$<D, S>$ tal que $L_M = S$.

La función de complejidad de tiempo f_M:$\mathbf{Z}^+ \rightarrow \mathbf{Z}^+$ se define como:

$$f_M(n) = \max\{m | \exists x \in S,\ n = \text{long}(x), \text{ y } m \text{ es el tiempo de la computación de aceptación de } x\}$$

En este sentido, se dice que el programa de una máquina de Turing tiene complejidad polinomial en tiempo, si existe un polinomio p tal que, para todo $n \in Z^+$, $f_M(n) \leq p(n)$.

En términos de los algoritmos, se dice que un algoritmo A es eficiente cuando su función de complejidad en tiempo se puede acotar por un polinomio. Es decir, si $f_A(n)$ es la función de complejidad en tiempo del algoritmo A y existe un polinomio p tal que, para todo $n \in \mathbf{Z}^+$ $f_A(n) \leq p(n)$, entonces se conviene en decir que A es un algoritmo eficiente.

1.1.2 Máquina de Turing no determinista

El modelo de máquina de Turing no determinista (MTND), definido en [GaM79], tiene la misma estructura que una máquina de Turing determinista, excepto que se le adiciona un módulo de adivinanza, el cual tiene su propia cabeza de sólo escritura y se usa con el solo propósito de que escriba la cadena que adivina. Una representación esquemática de este modelo se presenta en la figura 1.2.

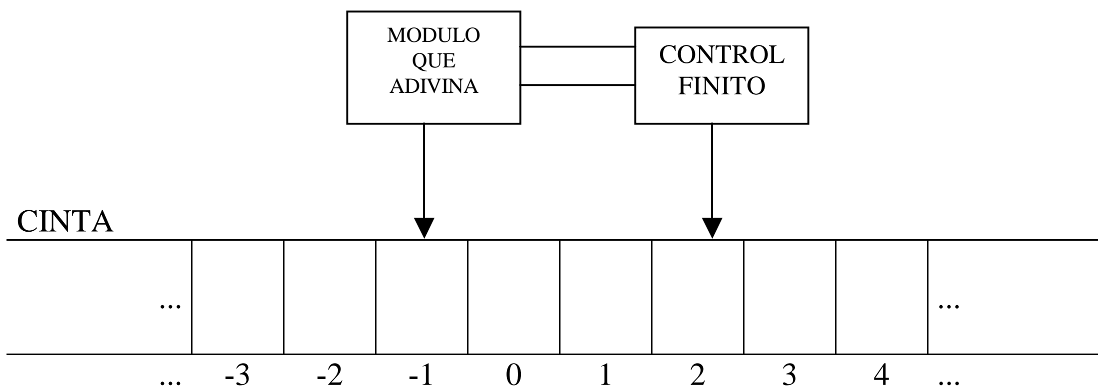

Figura 1.2 Representación esquemática de una máquina de Turing no determinista

El funcionamiento de la máquina de Turing no determinista se describe a continuación:

Al inicio, se escribe una cadena $x \in \Sigma^*$ en las celdas numeradas de 1 hasta $/x/$ de la cinta (y las demás celdas contienen el símbolo blanco), la cabeza del control finito está posicionada en la celda 1 y el control finito está inactivo. La principal diferencia de este modelo con el de la máquina de Turing determinista estriba en que el reconocimiento de la cadena x toma lugar en dos fases: la fase de adivinación y la fase de revisión.

Fase de adivinación: el módulo que adivina mueve su cabeza de sólo escritura una celda a la vez para dejar escrito a partir de las celdas etiquetadas desde -1 hasta $-/w|$, una cadena $w \in \Sigma^*$, pasando posteriormente a la fase de revisión en donde ahora el módulo que adivina permanecerá inactivo. La elección de la cadena w a escribir es realizada por el módulo que adivina de una forma totalmente arbitraria, y por tal razón, puede inclusive nunca parar al estar generando a w.

Fase de revisión: inicia cuando el control de estados finitos es activado en el estado q_0. A partir de este momento, la computación continúa en exactamente la misma forma en que trabaja la máquina de Turing determinista, puesto que no interviene en esta fase el

módulo que adivina. Aunque bien es posible que la cabeza del control finito visite la cadena w escrita por el módulo que adivina.

La computación de la máquina de Turing no determinista M se detiene cuando el control finito pasa a un estado final, y se dice que es una computación de aceptación si el control finito se detiene en el estado de aceptacion q_y. Nótese que para alguna combinación de la cadena wbx, M puede nunca parar. En general, M tiene un número infinito de posibles computaciones para una entrada x dada, una computación por cada cadena w generada por el módulo que adivina.

Se dice que M acepta a x si al menos existe una cadena w que hace que M llegue a un estado de aceptación. El lenguaje reconocido por M es entonces:

$$L_M = \{x \in \Sigma^* \mid \exists w \text{ generado por el módulo que adivina tal que } M \text{ acepta a } x\}$$

El tiempo requerido por una máquina de Turing no determinista M que acepta a una cadena $x \in L_M$, se define como el máximo (sobre todas las computaciones de aceptación de M sobre x), del número de pasos que ocurren tanto en la fase de adivinación como de revisión, hasta que esta última fase llega a un estado de aceptación. La función de complejidad de tiempo T_M:$\mathbf{Z}^+ \rightarrow \mathbf{Z}^+$ para M es:

$$T_M(n) = max(\{1\} \cup \{m \mid \exists x \in L_M, \text{long}(x) = n, M \text{ acepta a } x \text{ en } m \text{ pasos de computación}\})$$

Nótese que la función de complejidad de tiempo para M depende sólo del número de pasos que ocurren en una computación de aceptación, y que por convención, $T_M(n)$ es igual a 1 cuando ninguna entrada de longitud igual a n es aceptada por M.

Se dice que más que hallar una solución al problema de decisión PD, M comprueba soluciones para el PD, ya que dada una instancia x del PD, el módulo que adivina propone una propuesta de solución w, y M debe comprobar si w es realmente una solución En este sentido, la máquina de Turing no determinista M es usada como un mecanismo de verificación o comprobación de propuestas de solución.

Se dice que un programa corre en tiempo polinomial para la máquina de Turing no determinista M, si existe un polinomio p tal que $T_M(n) \leq p(n)$ para todo $n \geq 1$.

1.1.3 Clases de complejidad definidas por las máquinas de Turing

Con el fin de clasificar el esfuerzo computacional que se requiere al resolver problemas de decisión, originalmente se usaron las máquinas de Turing como el modelo formal de computación [HaJ65]. El concepto clave fue definir una clase de complejidad en términos de los lenguajes que reconoce una máquina de Turing con recursos acotados (se consideran aquí solo los recursos de tiempo y espacio). Las clases de complejidad permiten clasificar los lenguajes (problemas) según la complejidad intrínseca computacional

requerida para reconocerlos (resolverlos). De particular interés son las clases de complejidad definidas por recursos acotados logarítmicamente (salvo se indique lo contrario, se hablará de logaritmos base 2) o polinomios sobre la variable n, donde n es la longitud de las instancias de entrada.

Las primeras clases de complejidad definidas sobre MTD's fueron:

- *DLOG* $= \{L \subseteq \Sigma^* \mid \forall x \in L$, x es aceptada en espacio logarítmico$\}$
- *P* $= \{L \subseteq \Sigma^* \mid \forall x \in L$, x es aceptada en tiempo polinomial$\}$
- *PSPACE* $= \{L \subseteq \Sigma^* \mid \forall x \in L$, x es aceptada en espacio polinomial$\}$

También se pueden definir las clases de complejidad en términos de problemas de decisión, de la manera siguiente:

- *DLOG* $= \{PD \mid \exists M$ MTD que resuelve *PD* usando espacio logarítmico$\}$
- *P* $= \{PD \mid \exists M$ MTD que resuelve *PD* en tiempo polinomial$\}$
- *PSPACE* $= \{PD \mid \exists M$ MTD que resuelve *PD* usando espacio polinomial$\}$

A pesar de que estas clases de complejidad se definen en términos de máquinas de Turing, hasta ahora, todos los modelos reales de computación estudiados como máquinas de Turing de varias cintas, máquinas de acceso aleatorio (RAM), Intérpretes de lenguajes como Básic, Pascal, Lisp, etc., son equivalentes con respecto al comportamiento de complejidad polinomial en tiempo. Se cree entonces que cualquier otro modelo razonable de cálculo compartirá esta equivalencia. Por tanto, la clase de complejidad *P* definida anteriormente no será afectada si es cambiado el modelo de computación, así que se puede seleccionar uno u otro modelo de cómputo según convenga sin sacrificar la generalidad de los resultados.

La hipótesis de que el concepto intuitivo de computabilidad puede identificarse con lo que una máquina de Turing puede realizar, es conocida como la tesis de *Church-Turing*. Y aunque no podemos esperar una demostración de esta tesis, puesto que el concepto de calculable sigue siendo algo informal, se puede proporcionar evidencia de su carácter razonable [HoJ93].

Las clases de complejidad definidas usando máquinas de Turing no deterministas son:

- *NLOG*$=\{PD|\exists M$ MTND que comprueba soluciones de *PD* usando espacio logarítmico$\}$
- *NP* $= \{PD|\exists M$ MTND que comprueba soluciones de *PD* en tiempo polinomial$\}$
- *NPSPACE*$=\{PD|\exists M$ MTND que comprueba soluciones de *PD* usando espacio polinomial$\}$

A continuación se describe en términos de lenguajes solo a la clase *NP*, ya que la definición para las clases de complejidad *NLOG* y *NPSPACE* son equivalentes.

$NP = \{L \subseteq \Sigma^* \mid \exists M$ MTND: $(L = L_M)$ & $(\forall x \in L$, x es aceptada por M en tiempo polinomial$)\}$

Nótese que verificación o comprobación en tiempo polinomial, no implica resolubilidad en tiempo polinomial. Al decir que se puede comprobar que una cadena *w* es o no una solución al problema, no se está realmente considerando el tiempo invertido en encontrar dentro de un espacio potencialmente grande de posibilidades a la cadena *w* propuesta como solución al problema.

Correspondiendo a esta noción de modelos no deterministas, un algoritmo que se ejecuta en tales modelos, será no determinístico, y en este sentido, estará compuesto de dos fases separadas: la fase que adivina y la fase que revisa.

Dada una instancia *x* de un problema de decisión *PD*, la primera fase adivina una estructura *w* que se propone como solución. Entonces la instancia *x* y la cadena *w* serán la entrada a la segunda fase (fase de revisión) que procede en forma determinista para terminar con una respuesta '*SI*' o '*NO*' o, caer en un ciclo infinito.

Un algoritmo no determinista resuelve un problema de decisión *PD:<D, S>* si se cumplen las siguientes propiedades para todas las instancias *I* del problema de decisión:

1. Si *I* tiene una solución, entonces existe alguna cadena *w* que es adivinada y que conduce a la fase de revisión a que responda '*SI*'.
2. Si *I* es una instancia del *PD* sin solución, entonces no existe cadena *w* que pueda generar el módulo que adivina tal que conduzca a la fase de revisión a responder '*SI*'.

Por ejemplo, un algoritmo no determinista para el problema del agente viajero puede construirse usando un estado de adivinación, que simplemente proponga una secuencia arbitraria sobre las ciudades. El estado de revisión se encargará de verificar que la suma de las distancias de las ciudades en la secuencia dada cumpla la cota exigida.

Un algoritmo determinista que resuelve el problema de decisión *PD* se dice que es de tiempo polinomial si existe un polinomio *p* tal que para toda instancia *I* del *PD*, existe una cadena *w* que conduce a la fase de revisión a responder '*SI*', en tiempo acotado por $p(long(I))$. Nótese que esta cota impone también límites en la longitud de *w*.

Puesto que a cualquier modelo de cómputo se le puede adicionar el módulo que adivina con posibilidad de sólo escritura, tal y como se ha realizado en este apartado para las máquinas de Turing, se podría reproducir la definición de no determinismo con cualquier otro modelo de computación. Las versiones resultantes de definición de clases no deterministas serán equivalentes en los otros modelos de computación y particularmente,

todos estos modelos son equivalentes en lo que respecta al tiempo polinomial, y por tal, a la definición de la clase *NP*.

DTIME(*F*(*n*)) (*DSPACE*(*F*(*n*))) denota la clase de problemas que son aceptados por máquinas de Turing deterministas que reconocen su entrada dentro de un tiempo (espacio) *F*(*n*). En tanto que *NTIME*(*F*(*n*)) (*NSPACE*(*F*(*n*))) denota la clase de problemas aceptados por máquinas de Turing no deterministas que aceptan dentro de tiempo (espacio) *F*(*n*).

Al especificar las cotas de clases, muchas veces se usa la notación de la función de orden *O*. Si $G(n)$ es una función, $G{:}N \rightarrow \Re$, $O(G(n))$ es el conjunto de funciones *G'* que satisfacen $G'(n) \leq c \cdot G(n)$ para alguna constante real positiva c y para toda $n \geq n_0$, donde n_0 depende de *G*. Nótese que si n_0 es cero, entonces se cumple para todo $n \in \mathbf{N}$.

La notación *O* es usada dentro de las clases *NTIME*, *DTIME*, etc. Por ejemplo, $DTIME(2^{O(n)})$ denota la unión de $DTIME(2^{cn})$ tomadas sobre todas las constantes c. También se usa la notación $poly(G(n))$ que abrevia a $O(G(n)^{O(1)})$, esta es la clase de funciones acotadas superiormente por algún polinomio de función de *G*, ya que $O(1)$ denota a cualquier constante real positiva. Usando esta notación, podemos reescribir a las clases de complejidad como:

- $DLOG = DSPACE(log(n))$,
- $NLOG = NSPACE(log(n))$,
- $P = DTIME(poly(n))$,
- $NP = NTIME(poly(n))$ y
- $PSPACE = DSPACE(poly(n))$.

Las clases *P* y *NP* se pueden también identificarse de la siguiente manera:

- *P* es la clase de problemas que poseen algoritmos deterministas de resolución que tardan tiempo polinomial.
- *NP* es la clase de problemas que tienen un algoritmo determinista de resolución que corre en tiempo exponencial, pero para los cuales, también existe un algoritmo no determinista que corre en tiempo polinomial.

A los problemas de la clase *P* se les reconoce como problemas *tratables*, puesto que para cualquiera de sus instancias, éstas se resuelven por algoritmos que corren en un tiempo acotado polinomialmente. Se dice que un problema no ha sido bien resuelto hasta que es hallado un algoritmo determinista de tiempo polinomial que lo resuelve [GaM79]. Se le asigna a un problema el término de *intratable*, si este se ha mostrado tan difícil que no se ha encontrado algoritmo determinista con complejidad polinomial en tiempo que lo resuelva, es decir, no se ha hallado algoritmo eficiente que lo resuelva. Entonces una primera clase de complejidad que contiene problemas intratables es la clase *NP*.

Todo algoritmo cuya función de complejidad de tiempo no pueda acotarse por una función polinomial, se dice que es un *algoritmo de complejidad exponencial en tiempo*, aún cuando debe notarse que esta definición incluye ciertas funciones de complejidad de tiempo, tales como, $n^{\log(n)}$, las cuales normalmente no son consideradas como funciones exponenciales.

El término intratable, refleja el punto de vista de que los algoritmos de tiempo exponencial no son considerados “buenos” algoritmos y que, generalmente, tal clase de algoritmos reflejan variaciones de búsquedas exhaustivas, mientras que algoritmos de tiempo polinomial generalmente son construidos sólo a través de explotar las estructuras internas e inherentes del problema.

Existen algoritmos de tiempo exponencial que han sido muy útiles en la práctica. La complejidad de tiempo de un algoritmo tal y como se ha definido, es una medida para el peor de los casos, y el hecho de que un algoritmo tenga complejidad exponencial, significa que existe al menos una instancia del problema que requiere mucho tiempo, aún y cuando muchas de las instancias del mismo problema se resuelvan en la práctica en tiempo polinomial.

Aún y cuando ciertos algoritmos de complejidad exponencial para determinados problemas tienen un buen comportamiento en la práctica, esto no ha detenido el avance de las investigaciones por buscar algoritmos de tiempo polinomial que resuelvan los mismos problemas. Y de hecho, el comportamiento general de los algoritmos exponenciales, lleva a la sospecha de que hace falta capturar alguna propiedad crucial del problema cuyo refinamiento pueda llevarnos a mejores métodos. Un área de gran interés es sobre técnicas generales que permitan diseñar algoritmos deterministas de tiempo polinomial, ya sea para los problemas intratables o para variaciones de éstos.

La habilidad de algoritmos no deterministas, de poder revisar un número exponencial de posibilidades en tiempo polinomial, genera la sospecha de que este tipo de algoritmos son estrictamente más poderosos que los algoritmos deterministas de complejidad polinomial. Sin embargo, contra todos los esfuerzos realizados, no se ha podido demostrar esta especulación. La pregunta ¿$P = NP$? es uno de los problemas abiertos más importantes en la ciencia de la computación.

Aún más, sabiendo que se cumple la siguiente jerarquía entre las clases de complejidad: $DLOG \subseteq NLOG \subseteq P \subseteq NP \subseteq PSPACE$, sigue siendo una pregunta abierta decidir cuáles de estas contenciones son propias, aún cuando se sabe que $DLOG \neq PSPACE$ [StL87].

1.2 Teoría de los problemas NP-Completos

Iniciemos presentando una definición de la clase *NP* dada por Khanna [KhS96].

Definición 1.1 Un lenguaje $L \subseteq \Sigma^*$ está en *NP* si existe máquina de Turing determinista M que trabaja usando cotas polinomiales de tiempo y se tiene una constante c tal que para todo $x \in \Sigma^*$:

- $x \in L \Rightarrow \exists y \in \Sigma^*$ con $|y| = O(|x|^c)$ y tal que M reconoce al par (x, y) cuando se le da como entrada.
- $x \notin L \Rightarrow \forall y \in \Sigma^*$, M rechaza el par (x, y) cuando se le da como entrada.

Una interpretación natural de la definición anterior significa que la cadena y es la prueba para la aseveración de que $x \in L$ y la máquina M es el verificador de tal prueba.

La teoría sobre los problemas NP-Completos fue desarrollada originalmente para estudiar problemas de decisión; Cook [CoS71] y Levin [LeL73] establecieron que todos los lenguajes en *NP* son "reducibles" al problema 3-SAT (el problema de decidir si una formula en 3-FNC es o no satisfactible), un poco después, Karp [KaR72] mostró que el problema 3-SAT es "equivalente" a una gran variedad de problemas de decisión *NP*. A continuación se formalizarán las nociones de reducción y equivalencia.

Definición 1.2 Un lenguaje $L_1 \subseteq \Sigma_1^*$ es reducible en tiempo polinomial a otro lenguaje $L_2 \subseteq \Sigma_2^*$ (denotado como $L_1 \propto_p L_2$) si existe una función computable en tiempo polinomial $f{:}\Sigma_1^* \rightarrow \Sigma_2^*$ tal que para todo $x \in \Sigma_1^*$, $x \in L_1$ si y solo si $f(x) \in L_2$.

Esta noción de reducción es llamada reducción en tiempo polinomial de "muchos a uno" o simplemente "reducción Karp". Un lenguaje L_1 es equivalente en tiempo polinomial a otro lenguaje L_2 si: $L_1 \propto_p L_2$ y $L_2 \propto_p L_1$. A partir de ahora, obviaremos $\propto_p$ con $\propto$.

Definición 1.3 Un lenguaje L es NP-Difícil si para cualquier lenguaje $L' \in NP$: $L' \propto L$.

Definición 1.4 Un lenguaje L es NP-Completo si $L \in NP$ y L es NP-Difícil.

El resultado de Cook y Levin estableció que el problema 3-SAT es NP-Completo. Karp mostró que el problema 3-SAT es reducible a muchos otros problemas importantes de decisión en *NP*, tales como: determinar si un grafo es k-coloreable, verificar si hay ciclos hamiltonianos en un grafo, verificar si un grafo tiene un "clique" de al menos tamaño k, y de aquí, que estos problemas son también NP-Difíciles. Se tiene que, o bien todos estos problemas son difíciles (es decir, $P \neq NP$), o todos ellos son fáciles (es decir, $P = NP$).

La teoría de los problemas NP-Completos proporciona un ambiente de trabajo para el estudio de la complejidad de otros problemas que pueden plantearse como problemas de optimización.

1.2.1 Caracterización sintáctica de los problemas en *NP*

La noción de complejidad parece encontrarse inherentemente ligada a los modelos computacionales y recursos acotados en tiempo y espacio. Pero sorpresivamente, casi

paralelamente con el desarrollo del punto de vista de la teoría de los problemas NP-Completos orientada a la computación, se desarrolló un punto de vista descriptivo o sintáctico de las clases de complejidad. Tales esfuerzos están basados en una intuición simple, en la cual se dice que un problema que es "difícil de decidir" es probablemente también un problema "difícil de expresar", y viceversa. Un resultado clásico en esta dirección fue establecido por Fagin [FaR74] y se presenta en el teorema siguiente.

Teorema 1.1 *NP* se conforma precisamente por los problemas de decisión que pueden ser expresados en la lógica existencial de segundo orden y que se expresan como: $\exists S\ \Phi(I, S)$ donde Φ es una fórmula de primer orden, S es una estructura (es decir, una variable de segundo orden que recorre relaciones con aridad fija bajo un universo de entrada), y $I = (U; P)$ es la estructura de entrada comprendida de un universo U y un conjunto finito de predicados P de aridad constante.

Este es un teorema que vale la pena remarcar, ya que una clase de complejidad que fue definida en términos de la computación que realiza una máquina de Turing también puede ser descrita sin referencia tanto a la máquina como a los recursos. Esta caracterización expresa la estructura lógica de los lenguajes en *NP*. Usando esta caracterización de *NP* es una tarea relativamente intuitiva establecer 3-SAT como un problema NP-Completo. La facilidad de descubrir de manera natural problemas completos es un atributo muy deseable, el cual parece disponible con clases definidas sintácticamente, pero es una tarea difícil cuando una clase está definida en base a su comportamiento computacional.

En las últimas dos décadas, diversos pioneros (como Immerman y otros) han trabajados sobre caracterizaciones sintácticas de muchos otras clases, por ejemplo, *PSPACE* y *NSPACE* (ver [ImN95]).

1.2.2 Problemas de optimización en NP

La clase de problemas de optimización que se derivan de problemas de decisión en *NP* es llamada *NPO*, como un acrónimo para designar que son problemas de optimización derivados de problemas en *NP*.

Definición 1.5 Un problema Π en *NPO* se conforma por una *4*-tupla $\langle D, S, V, meta \rangle$ tal que:

- D es el conjunto de instancias de entrada que son reconocibles en tiempo polinomial.
- Dada cualquier entrada $x \in D$, $S(x)$ denota el conjunto de soluciones factibles, tales que, hay un polinomio p tal que para todo $y \in S(x)$ $|y| \leq p(|x|)$. Más aún, existe un predicado π computable en tiempo polinomial tal que $\pi(x, y)$ es verdadero, sí y solamente sí, $y \in S(x)$.
- Dada $y \in S(x)$, $V(x, y)$ denota el valor de la función objetivo, el cual es calculado en tiempo polinomial.

- Finalmente, $meta \in \{max, min\}$ y se usa para denotar cuando Π es un problema de maximización o minimización.

Los problemas en *NPO* pueden transformarse en problemas de decisión al utilizar cotas que satisfagan la función objetivo. El problema de decisión que resulta es al menos tan difícil como el problema de optimización original, y muchas veces, la complejidad de ambos problemas se relaciona solamente a través de un factor polinomial. Conforme pasan los años, se ha mostrado que más y más problemas de optimización tienen asociado un problema de decisión NP-Completo. Así que, la teoría de NP-Completos ha sido usada para establecer la intratabilidad de los problemas en *NPO*.

En la época de los 70's, se empezó a creer por una comunidad bastante amplia, que las dos clases *P* y *NP* son distintas y que por lo tanto, no hay algoritmos de tiempo polinomial que resuelvan exactamente a problemas en *NPO*. Sin embargo, dado que muchos de estos problemas tienen fuertes aplicaciones en la práctica, determinar la tratabilidad computacional de ellos es muy importante, por lo que fue natural que muchos investigadores comenzarán a considerar relajaciones de tales problemas para obtener versiones que fueran tratables. La primera relajación considerada fue la de la condición de "optimabilidad"; así que el enfoque fue puesto en la construcción de algoritmos que calcularan soluciones cercanas al óptimo, es decir, soluciones que garantizaran estar dentro de un factor multiplicativo de la solución óptima.

Definición 1.6 Un algoritmo *A* es un algoritmo de aproximación para un problema Π en *NPO* si dada una instancia de entrada *I*, éste calcula una solución factible *S* para la entrada *I*.

Por supuesto, el valor de la solución factible puede estar alejado del valor óptimo, así que el interés se encuentra en diseñar algoritmos que construyen soluciones con un "valor garantizado" de cercanía al óptimo, lo cual nos lleva a considerar algunas definiciones que nos ayudarán a caracterizar a los algoritmos de aproximación. Se usará la notación $OPT(I)$ para denotar el valor óptimo de la función objetivo en la instancia *I* y $V(I, A(I))$ para denotar el valor obtenido por el algoritmo *A* al evaluar la instancia *I* en la función objetivo.

Definición 1.7 Se define el cociente de aproximación de un algoritmo *A*, denotado por C_A, para una instancia *I* de un problema de optimización Π (con $|I| = n$) como:

$$C_A = min\left\{\frac{V(I, A(I))}{OPT(I)}, \frac{OPT(I)}{V(I, A(I))}\right\}$$

Nótese que C_A es: siempre menor o igual a 1 y el factor de garantía r_A del algoritmo *A* para Π será: $r_A = \min\{C_A \mid \text{para toda } I \text{ de } \Pi\}$

A una solución que está dentro de un factor multiplicativo r_A del valor óptimo se le conoce como una r_A-aproximación, y decimos que un problema NPO es aproximable dentro de un factor r_A si éste tiene un algoritmo de aproximación de tiempo polinomial con factor de garantía r_A.

A mediados de los 70's, Johnson [JoD74] estudió la propiedad de aproximabilidad en tiempo polinomial de problemas en *NPO* y descubrió que mientras es posible encontrar buenas aproximaciones en tiempo polinomial para diversos problemas de optimización NP-difíciles, hay muchos otros que resisten ser aproximados dentro de factores constantes. Los trabajos subsecuentes realizados por diversos investigadores solamente han confirmado lo observado por Johnson. En la actualidad, se tienen diversos resultados que muestran que muchos problemas de optimización NP-difíciles no pueden ser "bien aproximados", a menos que $P = NP$. La dificultad de hallar buenas aproximaciones podría deberse a que las reducciones usadas con los problemas de decisión no preservan la calidad de las soluciones aproximadas, por lo que se hace necesario desarrollar un ambiente de trabajo que permita estudiar los problemas de optimización, sin eliminar sus características esenciales.

1.2.3 Vista computacional de los problemas de optimización

Un esquema de clasificación basado en la aproximabilidad computacional de los problemas en *NPO* fue usado implícitamente en el trabajo de Johnson. Así, se tiene la conformación de clases como: *APX* y *PTAS*.

Definición 1.8 Un problema Π esta en la clase *APX* si existe un algoritmo de tiempo polinomial para Π cuyo factor de garantía está acotado por una constante.

Definición 1.9 Un problema Π está en la clase *PTAS* si para cualquier racional $\varepsilon > 0$, existe un algoritmo de aproximación de tiempo polinomial para Π cuyo factor de garantía esta acotado por $(1 + \varepsilon)$.

Sea f una función, f-APX denota la clase de problemas en *NPO* que son aproximables dentro de un factor f, así, se obtiene una jerarquía de clases de complejidad. Por ejemplo, poly-APX y log-APX son las clases de problemas en *NPO* los cuales tienen, respectivamente, algoritmos de aproximación con un factor de garantía acotado polinomialmente y logarítmicamente, con respecto a la longitud de la entrada.

La fortaleza de esta clasificación radica en la habilidad de capturar problemas que tienen un umbral específico de aproximabilidad. Podríamos estar interesados en restringir nuestra atención a subconjuntos polinomialmente acotados de estas clases.

Definición 1.10 El subconjunto polinomialmente acotado de la clase de problemas C, con $C \in$ APX, denotado como C-PB, es el conjunto de problemas $\Pi \in C$ tal que para toda instancia I, n=$|I|$, existe un polinomio $p(n)$ tal que $OPT(I) \leq p(|I|)$.

Entonces, APX-PB denota la clase de problemas en NPO que son aproximables dentro de factores constantes y que tienen valores óptimos acotados polinomialmente.

Los problemas SAT y MaxSAT son problemas NP-Completos, en particular MaxSAT es un problema de la clase APX-PB. Ambos problemas son centrales para la Demostración Automática de Teoremas (DAT) y para la teoría de la complejidad computacional, por lo que es importante realizar un análisis a detalle de estos dos problemas. En la siguiente sección se hará una descripción general de estos problemas y posteriormente, en el siguiente capítulo, se presentará un análisis extenso de algoritmos aplicados en la resolución de ellos.

1.3 Problemas SAT y MaxSAT

El problema de decisión asociado al problema de satisfactibilidad (SAT) de una fórmula proposicional en forma normal conjuntiva, puede expresarse como:

Instancia: Un conjunto $U = \{ u_1 ,..., u_n \}$ de variables booleanas y una colección $F = \{ C_1 ,..., C_m \}$ de cláusulas definidas sobre U.

Pregunta: ¿Existe una asignación de valores de verdad a los elementos de U, que haga que todas y cada una de las cláusulas de F tomen valor verdadero?

Notación:

Asignación: Es una función $t: U \rightarrow$ *{'Falso', 'Verdadero'}*.
Literal: Las constantes *'Falso', 'Verdadero'* o una expresión de la forma u o $\neg u$, donde u es elemento de U.
Frase: Es una conjunción de literales $F = l_1 \wedge ... \wedge l_k$. Una frase es verdadera si todas sus literales lo son.
Cláusula: Es una disyunción de literales, $C = l_1 \vee ... \vee l_k$. Una cláusula es verdadera si alguna de sus literales lo es.
Forma normal conjuntiva (FNC): Es una conjunción de cláusulas.
Una k-FNC, $k \in N$, es una FNC con exactamente k literales en cada cláusula.
Forma normal disyuntiva (FND): Es una disyunción de frases.
Una k-FND, $k \in N$, es una FND con exactamente k literales en cada frase.

Proposición 1.1: Toda fórmula proposicional es lógicamente equivalente a una FNC, y de hecho, la FNC equivalente es algorítmicamente calculable [GaJ87]

Una FNC F es *satisfactible* si y solo si existe una asignación de valores de verdad para U, que simultáneamente satisfaga a cada una de las cláusulas en F.

Por ejemplo, una instancia específica del problema SAT puede ser la siguiente fórmula *F,* sobre la que se cuestiona si es o no satisfactible:

$$F\ (x_1,..., x_7) = (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_3 \vee x_4) \wedge (x_5 \vee \neg x_3) \wedge (\neg x_5 \vee \neg x_6) \wedge (x_1 \vee x_6 \vee x_7) \wedge (\neg x_5 \vee \neg x_7) \wedge (x_2) \wedge (\neg x_3 \vee \neg x_7)$$

La fórmula anterior es satisfactible con la asignación:

$t(x_1) = t(x_3) = t(x_5) = t(x_6) =$ 'Falso'; $t(x_2) = t(x_4) = t(x_7) =$ 'Verdadero';

Toda asignación que satisface a una fórmula *F* se dice que es un modelo para *F*. El problema SAT consiste en decidir, si dada una fórmula *F* de entrada, que sin pérdida de generalidad puede suponerse en FNC (de acuerdo a la proposición 1.1), se determine si acaso existe una asignación de *U* que haga que *F* tome el valor verdadero.

$F = \{ C_1 ,..., C_n \}$ - Conjunto clausular

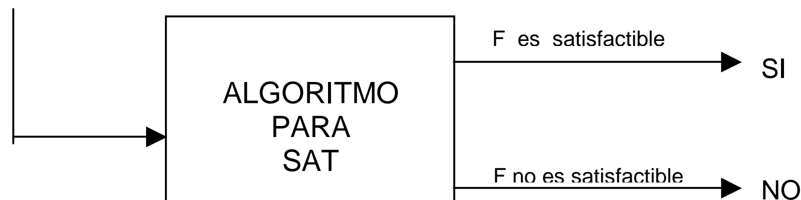

Figura 1.3 Planteamiento del problema SAT

El primer problema que se demostró ser NP-Completo fue el problema SAT [CoS71]. Cook utilizó el problema SAT para delimitar la clase de los problemas NP-Completos. SAT es el clásico problema *intratable*, lo que significa que todos los algoritmos hasta ahora presentados requieren de mucho tiempo de cómputo o de mucho espacio para su resolución en la práctica, excepto quizás, para algunos casos triviales. Es una pregunta abierta hasta ahora, demostrar si acaso SAT podría ser un problema *tratable*, es decir, saber si se podrán construir algoritmos que resuelvan SAT de manera eficiente (en tiempo polinomial).

Otro problema que se ha demostrado ser intratable, es la versión de optimización para el problema de satisfactibilidad, conocido como el problema de *máxima satisfactibilidad MaxSAT*, que consiste en encontrar una asignación a las variables que maximice el número de cláusulas que se satisfacen simultáneamente. Por ejemplo, si el conjunto original de cláusulas es insatisfactible, entonces MaxSAT busca conocer la asignación de valores veritativos que permitan que un número máximo de cláusulas se satisfagan simultáneamente. MaxSAT es NP-Completo aún cuando cada cláusula contenga a lo más dos literales (Max-2-SAT) [GaM76].

Uno de los puntos importantes al trabajar con los problemas SAT y MaxSAT, es el que, como representantes de clases de complejidad, el hallar mejoras a los algoritmos que los resuelvan, implica que tales mejoras pueden extenderse al tratar a los demás problemas de su clase de complejidad. Por ejemplo, si se hallase un algoritmo determinista de complejidad polinomial en tiempo para resolver SAT, entonces todo problema de la clase *NP* podría resolverse por algoritmos deterministas con complejidad de tiempo polinomial, y de hecho, entonces $P = NP$.

El problema a tratar en esta investigación es el problema MaxSAT, así, en el siguiente capítulo se proponen diversos algoritmos, todos ellos basados en la técnica de búsqueda local para el tratamiento de este problema y en el capítulo III se presentan diversos elementos para el análisis de las propuestas realizadas.

Capítulo II

Propuestas algorítmicas para la resolución del problema MaxSAT

2.1 Técnica de búsqueda local

Entre los procedimientos de aproximación, una técnica que ha resultado eficiente para hallar una asignación que satisfaga a una fórmula satisfactible, es la técnica de *búsqueda local* [GuJ93]. Para aplicar la búsqueda local a un problema particular es necesario especificar el criterio de vecindad a un punto, la función objetivo y un procedimiento que permita obtener los puntos iniciales de búsqueda. En general, todos los algoritmos a presentar aquí, se pueden analizar a partir de la estructura $<S_0, f, H>$, donde:

S_0 - procedimiento para generar el punto inicial de búsqueda,
f - función objetivo,
H - métrica que define la distancia entre puntos del dominio.

En cada uno de los algoritmos a presentar, el procedimiento de búsqueda opera sobre el espacio discreto de posibles asignaciones de la fórmula de entrada, se usa como punto inicial de búsqueda un punto del dominio generado aleatoriamente, y como métrica entre puntos del dominio se usa la distancia conocida con el nombre de "Hamming", que denotaremos con *H*.

Denotemos con N(x) la vecindad del punto x, N(x) se calcula como: N(x) = {y en el dominio / $H(x,y)=1$}. Es decir, dos puntos del dominio son vecinos si difieren sus cadenas de bits en a lo más un bit. Para una constante d, la d-vecindad a un punto x, es el conjunto de puntos del dominio que difieren de x (bajo la métrica H) en a lo sumo d-bits. Entonces, un d-vecino de una asignación dada es el conjunto de todas las asignaciones donde cambia el valor lógico de a lo más d variables.

Se presenta en la figura 2.1 un algoritmo de búsqueda local conocido como GSAT [GuJ94]. Este algoritmo muestra como se verifican todos los vecinos de una cierta asignación T (generada aleatoriamente) buscando un valor que satisfaga a una fórmula F, si después de repetir esta búsqueda durante un cierto número de iteraciones (determinado desde el inicio del proceso), tal asignación no se encuentra, entonces se envía un mensaje que indica que no se encontró una asignación de satisfactibilidad, y en cambio, regresa el número máximo de cláusulas que logró satisfacer de manera simultánea.

Para un punto x, denotamos por com(x,j) al complemento de x en la dirección j, a saber, a la asignación que coincide con x salvo en la j-ésima entrada, en donde tiene asociado el complemento de $(x)_j$.

Dada una d-vecindad a un punto x, el algoritmo de búsqueda local puede decidir verificar en "cada uno" de los d-vecinos del punto y de ellos escoger el que produzca un mejor valor de acuerdo a la función objetivo f, o solamente buscar hasta encontrar una

asignación que mejore el valor de *f*. En todos los algoritmos implementados, se eligió la estrategia de pasar inmediatamente a un vecino que mejora la función objetivo, en lugar de revisar por un óptimo de la vecindad, ya que en la práctica, se ha visto que ambas estrategias alcanzan similares aproximaciones, pero sin embargo, la primera de éstas estrategias, reduce la complejidad del tiempo promedio del algoritmo.

```
Procedure GSAT() begin
        m = 0;
        for (i=1; i<=MAX-INTENTOS; i++) begin
                S := Una asignación generada aleatoriamente
                for (j=1; j<=MAX-FLIPS; j++) begin
                        if S satisface F then return (S, |F|)
                        S := com(S, j)
                        m := max{m,  # de cláusulas satisfechas por S}
                end
        end
        return (“Número máximo de cláusulas que se logró satisfacer es: “, m)
end
```

Figura 2.1 Pseudocódigo del algoritmo de búsqueda local [GuJ94].

2.2 Búsqueda local aplicada al problema SAT y MaxSAT

Como función objetivo se puede usar el polinomio que se obtiene de *aritmetizar* la fórmula booleana *F*. Sea *F* la fórmula en FNC, entonces:

$$F = \bigwedge_{1 \le i \le m} \bigvee_{1 \le j \le lc} l_{ij}$$

El polinomio que codifica a *F*, y que es función objetivo en la búsqueda local es:

$$f = \sum_{i=1}^{m} \prod_{j=1}^{l_c} m(l_{ij})$$

con $m(l_{ij}) = (1 - l_{ij})$ si $l_{ij} = x_i$, y $m(l_{ij}) = l_{ij}$ si $l_{ij} = \neg x_i$.

Así por ejemplo, si $F(x_1, x_2, x_3) = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_3)$, el polinomio que codifica a *F* es: $fp(x_1, x_2, x_3) = (1 - x_1)x_2 + x_1(1 - x_2)(1 - x_3)$.

Para una asignación dada, el polinomio *f* cuenta el número de cláusulas que son insatisfechas por la asignación. Así pues, decidir la satisfactibilidad de *F* es propiamente equivalente a minimizar *f*. Entonces *F* es satisfactible si y sólo si el mínimo de *f* sobre el dominio de asignaciones de *F* es cero. Una explicación más amplia del proceso de aritmetización se puede ver en [DeG95].

Entrada: *F* - fórmula booleana, con *n* - No. de Variables, *m* - No. de cláusulas.
Salida: *xv* - asignación hallada, *Valor* = número de cláusulas que logró satisfacer.

Variables globales:
Iter = Número de iteraciones,
MAX_ITER = Cota que indica el número máximo de iteraciones a realizarse.

```
Procedure LS() begin
   x = AsignacionAleatoria();             /* Se obtiene una asignación aleatoria */
   Valor = f(x);                          /* Aplica función objetivo a la asignación x */
   Iter = 1;
   while (Valor < m) and (Iter < MAX_ITER) do begin
          Iter = Iter + 1;
          while (!SenaMin) begin                 /* Ciclo principal */
                 SenaMin = VERDADERO;   /* Supóngase haber llegado a un mínimo */
                 for (j=1; j<=k; j++) begin     /* k - posibles cambios de bits */
                        if (f(com(x,j)) > Valor) begin /* Se mejoró a MaxSAT ? */
                               x = xv = com(x, j);     /* Intercambio del j-ésimo bit de x */
                               Valor = f(x);   /* Cambiar valor de la mejor asignación */
                               SenaMin = FALSO;   /* Seguir buscando */
                        end
                 end
          end
   end
   return(xv, Valor);                            /* Regresa mejor asignación y su valor */
end
```

Figura 2.2 Búsqueda local obvia.

La búsqueda local implementada (LS) comienza a partir de un punto inicial x_0 construido de forma aleatoria, se determina la vecindad del punto actual de búsqueda $N(x_i)$, y aplica un proceso iterativo tal que en cada iteración se obtenga un nuevo punto x_{i+1} en $N(x_i)$ de manera que $f(x_{i+1}) < f(x_i)$ (o, alternativamente, $f(x_{i+1}) >= f(x_i)$ si se tratase de maximizar), si tal punto existiese, y en tal caso, éste se convierte en el nuevo punto "solución" y se reitera el proceso. En otro caso, se retiene a x_i como óptimo local. Un algoritmo con este mecanismo de búsqueda se observa en la figura 2.2.

Si bien la búsqueda local ha demostrado ser una estrategia eficiente para resolver SAT y MaxSAT cuando las fórmulas F tienen varias asignaciones que las satisfacen, tiene el grave problema de bloquearse en óptimos locales, pero aún más grave, es que los factores de aproximación que garantizan estas estrategias son aún lejanos del umbral al que teóricamente puede aproximarse, por ejemplo, MaxSAT.

2.3 Búsqueda local no obvia

La función objetivo en la búsqueda local ha expresado tradicionalmente y de forma directa, maximizar el número de cláusulas que se satisfacen en la fórmula. Pero diferentes tipos de búsquedas locales pueden obtenerse al usar diferentes funciones objetivos, inclusive al usar funciones que originalmente no muestran la opción natural de maximizar el número de cláusulas que se satisfacen por una asignación, a este tipo de búsqueda se le llama comúnmente, *búsqueda local no obvia* (LS-N).

Un caso de búsqueda local no obvia f_{NOB} para Max-k-SAT es la presentada por Khanna [KhS95], donde la función objetivo no obvia es de la forma siguiente:

$$f_{NOB} = \sum_{i=0}^{k} C_i W(S_i)$$

Ecuación 2.1 Función objetivo de la búsqueda local no obvia.

Donde S_i denota al conjunto de cláusulas, en las cuales, exactamente i literales son verdaderas bajo una asignación x. Es decir, S_i es el conjunto de todas las cláusulas en donde se satisfacen exactamente i literales, $W(S_i)$ es la cardinalidad de S_i; y los $C_{i's}$ $(i=0,...,k)$, son constantes reales seleccionadas adecuadamente para cumplir ciertas condiciones del proceso de búsqueda y que ayudarán en la determinación del factor de aproximación (más adelante se definirá la manera de obtener los $C_{i's}$).

Por ejemplo, la función objetivo propuesta por Khanna [KhS95] para fórmulas del tipo 2-FNC (FNC donde hay exactamente 2 literales por cláusula), es:

$$f_{NOB} = 3/2\ W(S_1) + 2\ W(S_2)$$

Los óptimos locales de la función objetivo "estándar" no son necesariamente los óptimos locales de esta nueva función objetivo. En este caso, la segunda función objetivo permite saltar sobre los óptimos locales de LS y de hecho, proporciona un comportamiento diferente de las direcciones en la trayectoria de búsqueda. Se espera que el comportamiento de LS-N sea mejor, debido a la manera en que trata cada asignación, por ejemplo, la función objetivo no obvia puede entre dos asignaciones que satisfacen el mismo número de cláusulas decidir que asignación es mejor para la búsqueda.

Los teoremas 7 y 8 en [KhS95] establecen que el factor de garantía para LS sobre una d-vecindad para Max-2-SAT es 2/3 para $d = o(n)$, mientras que LS-N sobre una 1-vecindad obtiene un factor de garantía de 3/4. Por lo tanto, LS-N proporciona un refinamiento del factor de garantía de las soluciones halladas, aún si la búsqueda es restringida a pocos vecinos.

LS-N obtiene un factor de garantía de $(1 - 1/2^k)$ para Max-k-SAT [KhS95] si la cantidades $\Delta_i = C_{i+1} - C_i$, satisfacen:

$$\Delta_i = \frac{1}{(k-i+1)\binom{k}{i-1}}\left[\sum_{j=0}^{k-i}\binom{k}{j}\right]$$

Nótese que el uso de esta fórmula permite definir los valores de cada C_i, para toda versión de Max-k-SAT.

En general, la forma en que opera la búsqueda local descansa en dos fases, en la primera, llamada de “descenso”, que ocurre más o menos rápidamente, se pueden encontrar vecinos del punto actual de búsqueda que mejoran la función objetivo. Esta fase es relativamente corta [GeI93] y es seguida posteriormente por una segunda fase en la cual es difícil hallar vecinos que mejoren la función objetivo, por lo que generalmente se opta por visitar puntos que mantienen el mismo valor de la función. Esta segunda fase denominada ‘Plateau’ (meseta) es donde generalmente se invierte la mayor cantidad de tiempo del proceso de búsqueda.

Muchas de las estrategias que tradicionalmente se aplicaron para mejorar el comportamiento de la búsqueda local clásica se pueden aplicar a LS-N.

2.4 Función objetivo no obvia propuesta

Analizando las soluciones obtenidas por programas que implementan la búsqueda local clásica así como la búsqueda local no obvia, se detectó que existen instancias de Max-k-SAT en las cuales la búsqueda local clásica obtiene soluciones mejores solución que la búsqueda local no obvia.

Un ejemplo sencillo de este caso, es la familia de fórmulas del tipo:

$$F = \{\{x_i, x_j\} \mid 1 \le i < j \le n,\ n \ge 5\} \cup \{\{\neg x_1, \neg x_2\}\}$$

Considerando n=5 y como solución actual a $x = (1,1,1,1,1)$, entonces

$$f_{NOB}(x) = \frac{3}{2}(0) + 2\binom{5}{2} = 20$$

Y en efecto, x es un óptimo local que no satisface a F, debido a que para todo 1-vecino x', $f_{NOB}(x) \ge f_{NOB}(x')$, por ejemplo, con $x_1 = (0,1,1,1,1)$, $f_{NOB}(x_1) = 3/2(5) + 2(6) = 19.5$.

Mientras x es un óptimo local bajo LS-N, su vecino x_1 es una mejor solución dado que x_1 satisface a la fórmula booleana F.

A pesar de que LS-N tiene un mejor factor de garantía que LS, para este tipo de fórmulas, LS obtiene una mejor solución que LS-N.

Para corregir esta situación, se determinó una nueva función objetivo no obvia. Por ejemplo, para Max-2-SAT se definió:

$$f'_{NOB} = 3/2W(S_1) + 2W(S_2) - W(S_0)$$

Se define esta nueva función objetivo para considerar el número de cláusulas insatisfechas que aparecen en cada asignación evaluada, evento que no es contemplado por la función objetivo no obvia f_{NOB} propuesta por Khanna. Los valores de los coeficientes C_i (i=1, ..., k) para f_{NOB} y para f'_{NOB}, pueden usarse para diferenciar entre las asignaciones que tienen el mismo número de cláusulas satisfechas, sin embargo, el coeficiente C_0 es crucial para determinar el número de cláusulas no satisfechas que aparecen en cada asignación evaluada y es por eso que se incluye en f'_{NOB}. A la búsqueda local no obvia que se obtiene al utilizar esta nueva función objetivo la denotaremos como LS-NC$_0$.

Dada la calidad de los resultados empíricos obtenidos de esta nueva función objetivo no obvia con respecto a las dos anteriores, se decidió utilizarla como el núcleo principal en la propuesta algorítmica final de este trabajo de investigación.

Tanto el análisis empírico realizado a los algoritmos como la determinación del factor de garantía del algoritmo que obtiene LS-NC$_0$ se realiza en el capítulo IV de esta tesis.

A continuación se presenta una de las heurísticas adicionadas a LS-NC$_0$.

2.5 Búsqueda tabú

Los orígenes de la búsqueda Tabú (TS) pueden situarse en diversos trabajos publicados alrededor de hace 20 años. Oficialmente, el nombre y la metodología fueron introducidos posteriormente por Fred Glover en 1989. Numerosas aplicaciones han aparecido en la literatura, así como artículos y libros para difundir el conocimiento teórico del procedimiento (ver [DiA96]).

TS es una técnica para resolver problemas combinatorios de gran dificultad que está basada en principios generales de la Inteligencia Artificial. En esencia es un metaheurístico que puede ser utilizado para guiar cualquier procedimiento basado en la búsqueda local. Se caracteriza por intentar evitar que la búsqueda quede "atrapada" en un óptimo local que no sea global. A tal efecto, TS toma de la Inteligencia Artificial el concepto de memoria y lo implementa mediante estructuras simples con el objetivo de dirigir la búsqueda teniendo en

cuenta la historia de ésta. Es decir, el procedimiento trata de extraer información de lo sucedido y actuar en consecuencia. En este sentido, puede decirse que hay un cierto aprendizaje y que la búsqueda es inteligente. El principio de TS podría resumirse como:

"Es mejor una mala decisión basada en información que una buena decisión al azar, ya que, en un sistema que emplea memoria, una mala elección basada en una estrategia proporcionará claves útiles para continuar la búsqueda. Una buena elección fruto del azar no proporcionará ninguna información para posteriores acciones" [DiA96].

TS comienza de la misma forma que cualquier procedimiento de búsqueda local, procede iterativamente de una solución x a otra dentro de su vecindad $N(x)$. Sin embargo, en lugar de considerar toda la vecindad de la solución, TS define una nueva vecindad $N'(x)$ tal que $N'(x) \subseteq N(x)$.

Existen muchas maneras de definir la vecindad reducida $N'(x)$de un punto x. La más sencilla consiste en etiquetar como tabú las soluciones previamente visitadas en un pasado cercano. Esta forma se conoce como memoria a corto plazo y está basada en guardar una lista tabú T de las soluciones visitadas recientemente. Así en una iteración determinada, $N'(x)$ del punto actual x se obtendría a partir de $N(x)$ eliminando las soluciones etiquetadas como tabú.

El objetivo principal de etiquetar las soluciones visitadas como tabú es reducir la búsqueda evitando pasa por caminos ya visitados y se espera también que esto reduzca la posibilidad de que la búsqueda se cicle. Por ello se considera que tras un cierto número de iteraciones la búsqueda está en una región distinta y puede liberarse del status tabú (pertenencia a T) de las soluciones antiguas. En los orígenes de TS se sugerían listas tabús de tamaño pequeño, actualmente se considera que las listas pueden ajustarse dinámicamente según la estrategia que se esté utilizando.

Es importante considerar que los métodos basados en búsqueda local requieren de la exploración de un gran número de soluciones en poco tiempo, por ello es crítico el reducir al mínimo el esfuerzo computacional de las operaciones que se realizan a menudo. En ese sentido, la memoria a corto plazo de TS está basada en atributos en lugar de ser explícita; esto es, en lugar de almacenar las soluciones completas (como ocurre en los procedimientos de búsqueda exhaustiva) se almacenan únicamente algunas características de éstas.

La memoria mediante atributos produce un efecto más sutil y beneficioso en la búsqueda, ya que un atributo o grupo de atributos identifica a un conjunto de soluciones.

Para diseñar una estrategia tabú al algoritmo de búsqueda local no obvia LS-NC_0 que hemos propuesto, combinamos la propuesta de Hansen [HaP90] con una que propusimos anteriormente en [DeG96].

Las ideas principales de esta estrategia tabú son:

1.- Llevar un recuento de la dirección de la búsqueda que llevó a un óptimo local, llamada *dirección descendente*, y entonces se prohiben movimientos inversos a tal dirección de descenso, al menos durante un número fijo de iteraciones.

2.- Aplicar la prohibición sólo durante un número fijo de iteraciones.

Con los elementos descritos puede diseñarse un algoritmo básico de TS para un problema de Optimización dado. En la propuesta de búsqueda tabú presentada en [DeG96], se introduce una estrategia que permite acelerar el proceso de búsqueda durante la fase de 'meseta', que como se ha dicho, es la fase en la cual es difícil hallar vecinos que mejoren la función objetivo y que es donde se invierte la mayor cantidad de tiempo del proceso de búsqueda.

Para llevar control de la dirección de descenso se introduce un arreglo *Ind* que indica las direcciones en las que ha habido cambios durante el proceso de búsqueda. Valores positivos en las posiciones j del arreglo *Ind* indican cambios locales (dirección de descenso) en la dirección j, por lo que al pasar a un nuevo punto de búsqueda después de arribar a un óptimo, significará invertir los valores de las variables que durante la búsqueda no hayan sufrido cambios, por ejemplo, que corresponden a valores 0 en la posición j del arreglo *Ind*. El uso del arreglo que indica la dirección de descenso es útil para reducir el tiempo de prueba de puntos en direcciones opuestas a la dirección de descenso, agilizando así la búsqueda durante la fase de 'meseta'. Este arreglo *Ind* es utilizado como una memoria a corto plazo basada en atributos que es la lista tabú.

Llamaremos a la búsqueda tabú anteriormente presentada y que utiliza como función objetivo la función no obvia presentada en la ecuación 2.1, búsqueda tabú no obvia.

2.5.1 Búsqueda tabú no obvia con antipodales

Otra mejora a los procedimientos basados en la búsqueda local, consiste en seleccionar una estrategia óptima, o al menos adecuada, para saltar al siguiente punto de la búsqueda después de arribar a óptimos. A diferencia del salto probabilístico que caracteriza a procedimientos de búsquedas con control probabilístico (como son por ejemplo las heurísticas de recocido simulado), en la propuesta de búsqueda tabú se ha realizado el salto a nuevos puntos de búsqueda sólo hasta llegar a configuraciones de óptimos locales, y así ir saltando aleatoriamente de un óptimo local a otro. Este salto se inicia mediante el uso de un nuevo punto de inicio x' para la búsqueda local; este punto de inicio se obtiene mediante la negación de cada uno de los bits que conforman al último local encontrado x, a este nuevo punto x' se le da el nombre de antipodal de x, en tal sentido, dos puntos x y x' son antipodales si y solo si x and $x' \cong 0$ y x or $x' \cong 1$ (vector 0 y vector 1).

Supongamos que x_0 resultara ser un óptimo local, entonces se le puede aplicar una perturbación para hacerlo "saltar" a un nuevo punto x_1 que difiera significativamente de x_0. Contrariamente a como lo evaluaría la estrategia de recocido simulado, no se le aplica la prueba de "aceptar/rechazar" al nuevo punto, sino que a partir de x_1 se llega a un nuevo óptimo local y sólo hasta ese momento se aplica la prueba de aceptación. En este esquema, es conveniente mantener un arreglo con los mínimos locales que se van visitando para evitar caer en ciclos.

Como resultado del análisis empírico realizado a LS, LS-N y LS-NC_0, al rastrear la lista de óptimos locales que se obtenían en cada corrida se observó que las trayectorias de búsqueda llegaban a pasar varias veces por regiones ya exploradas después de arribar a un óptimo local; esto depende principalmente de la definición del punto de re-inicio de búsqueda.

Así que, cuando se arriba a un óptimo local, es importante controlar el punto de re-inicio de la búsqueda, primero para evitar caer en caminos ya explorados así como para ampliar el área de búsqueda. La forma idónea encontrada para definir al nuevo punto de búsqueda, fue usando el punto antipodal del último óptimo local hallado, o bien, si este punto perteneciera a una trayectoria ya visitada, entonces se puede construir un nuevo punto que difiera significativamente de cada uno de los óptimos locales que se tienen en la lista de óptimos.

En la figura 2.4 se puede observar el algoritmo de búsqueda tabú no obvia con antipodales (LS-NTA) donde se utiliza el arreglo *Ind* como arreglo tabú de memoria corta basada en atributos.

El éxito de esta propuesta está determinado primero por su habilidad de moverse exitosamente a través de la fase de 'meseta' en la búsqueda, reduciendo el tiempo de ésta fase y, segundo, por el uso de elementos antipodales del último óptimo local hallado como nuevo punto de re-inicio de la búsqueda, dando así amplitud al dominio de búsqueda.

Después de haber definido la nueva propuesta algorítmica, es pertinente presentar los elementos para el análisis de algoritmos que permitan clarificar los métodos para el análisis del algoritmo propuesto y así determinar su factor de garantía. En el capítulo siguiente se presentan los elementos para el análisis de algoritmos.

```
Entrada: F - fórmula booleana, n - No. de Variables, m - No. de cláusulas.
Salida: xv - asignación hallada,  Valor  = número de cláusulas que logró satisfacer.

Variables globales:
    OptimosLocalesVisitados=0, NumeroDeIteraciones=0;
    Valor, Valor1 = Indican el valor devuelto por la función objetivo
    x, x1  = Valores para las asignaciones
    Ind    = arreglo para el control de la trayectoria
```

```
Procedure LS-NTA() begin
    x = AsignacionAleatoria();

    /* Se permite a lo más visitar 5 diferentes óptimos */
    while (OptimosLocalesVisitados < 5) begin
        Valor = f'_NOB (x);
        HayCambio = VERDADERO;
        for (j=0; j<n; j++) Ind[j] = 0;/* Inicializar arreglo que controla la trayectoria */
        while (HayCambio) begin
            HayCambio = FALSO;
            for (j=0; j<n; j++) begin
                if (Valor = m) then return(x); /* Fórmula Satisfactible */
                NumeroDeIteraciones++;       /* Una iteración por cada vecino */
                if (Ind[j] > 0) then Ind[j]--;   /* Trayectoria de descenso */
                else begin
                    x1 = com(x, j);
                    Valor1 = f'_NOB(x1);
                    if (Valor < Valor1) then begin
                        x = x1;              /* Se actualiza la asignación */
                        Valor = Valor1;
                        Ind[j] = Valor1 - Valor; /* Importancia del cambio */
                        HayCambio = VERDADERO;
                    end
                end
            end /* Fin de los flips */
        end
        if (EstaEnOptimos(x)) then Contador++;     /* ¿ Optimo Local ya visitado ? */
        else InsertaEnOptimos(x);                  /* Nuevo Optimo Local */
        x = Antipodal(x);                          /* Antipodal del Optimo Local */
    end
    xv = MejorOptimos();
    return(xv, Valor);
end
```

Figura 2.4 Algoritmo de búsqueda tabú no obvia con antipodales

Capítulo III

Elementos para el análisis de algoritmos

3.1 Introducción

Cuando se están diseñando propuestas algorítmicas para la resolución de problemas NP-difíciles (como es el caso de los problemas SAT y MaxSAT), es relevante hacer un estudio de la complejidad computacional, así como de la calidad de las soluciones que dan tales algoritmos**.** Este capítulo se enfoca a la presentación de algunos de los elementos importantes a considerar cuando se analiza el comportamiento de algoritmos en general, y posteriormente se pone especial énfasis en el análisis de algoritmos para el tratamiento de los problemas de satisfactibilidad.

La evaluación de un algoritmo está sujeta a la cobertura de un número de criterios que relacionan varios aspectos intrínsecos con la expresión y ejecución del algoritmo. Entre los criterios que se consideran para determinar lo que es un buen algoritmo, se incluyen cuestiones subjetivas tales como: sencillez, claridad, adaptabilidad a los datos a manejar, etc. La ingeniería de software intenta tratar con bases matemáticas estos términos, y así tener parámetros para evaluar lo que significa que un algoritmo sea sencillo y claro. Medidas más objetivas (lo que no significa que necesariamente tengan que ser más importantes), es la complejidad en tiempo y espacio del algoritmo, su propiedad de convergencia, la claridad y precisión de sus resultados, etc. Algunos de los criterios y propiedades de los algoritmos que se usan más comúnmente para evaluarlos, son:

- Sencillez y claridad
- Su facilidad de implementación
- Si son algoritmos de aplicación general o específicos
- La flexibilidad y robustez del algoritmo
- Su consistencia y completitud
- Su factor de convergencia
- La calidad de la solución que proporcionan y
- La complejidad en tiempo y espacio del algoritmo

Parte de la consecuencia de que un algoritmo tenga una representación sencilla y no haga uso de técnicas complicadas, será que facilitará tanto su análisis como su implementación a través de un lenguaje computacional. Aunque existen los casos en el que ideas sencillas exigen código complicado en la implementación. Un algoritmo que es comparativamente fácil de codificar tiene ventajas sobre implementaciones intrincadas, tanto para su descripción como para su mantenimiento.

También se distingue a los algoritmos de entre los específicos y los algoritmos generales. *Un algoritmo específico* usa información específica del problema y sólo puede ser aplicado al problema para el cual fue diseñado, si se quisiera aplicar este algoritmo a

otro tipo de problema se requiere generalmente rediseñarlo tanto que realmente resulta en un nuevo algoritmo. A la inversa, *algoritmos generales* son diseñados independientes al problema y para ser aplicados en un gran número de situaciones. Un algoritmo hecho a la medida en general se puede ejecutar comparativamente rápido, pero tiene una aplicabilidad limitada y, un algoritmo general es usualmente lento pero tiene un gran potencial de aplicación.

La flexibilidad de un algoritmo es una característica importante. *Un algoritmo flexible* debe ser capaz de actualizarse fácilmente y adaptarse para tratar problemas relacionados. *Un algoritmo robusto* es capaz de producir buenos resultados bajo muy diferentes circunstancias, tales como cuando se trabajan con instancias patológicas del problema. Tomando ambas características, un algoritmo flexible y robusto resulta ser un algoritmo de propósito general y es posible el que pueda aplicarse en la resolución de muchos tipos de problemas.

Un aspecto relevante de los algoritmos es referente a la consistencia de las soluciones que encuentra. Un *algoritmo consistente* no permite equivocaciones en sus respuestas. Por ejemplo, al tratar problemas de decisión, la solución que proporciona un algoritmo consistente debe ser siempre correcta y no permitir equivocaciones, como podría ser responder *'SI'* cuando la respuesta correcta es *'NO'* o a la inversa. Aunque ante la imposibilidad de hallar las respuestas correctas a un problema en un tiempo acotado de cómputo (generalmente acotado en un orden polinomial de tiempo) se han diseñado algoritmos con bases probabilísticas, es decir, que las soluciones encontradas tienen un factor de certeza (y a su vez de error) de ser la solución que se busca.

Otra división que se considera al analizar las propuestas algorítmicas, es precisar si se tratan de algoritmos *completos* o *incompletos*. Los *algoritmos completos* siempre encuentran la solución correcta para toda instancia del problema. Los *algoritmos incompletos* encuentran soluciones parciales al problema, es decir, sólo para ciertas instancias del problema hallarán la solución correcta, en tanto que para otras instancias no podrán determinar cuál es la solución correcta [GuJ93].

Por ejemplo, un algoritmo de resolución para el problema SAT se dice que es *completo,* si para toda fórmula de entrada siempre puede verificar si existe o no una asignación que la satisfaga. En tanto, que un algoritmo *incompleto* puede encontrar sólo la correspondiente respuesta para una cierta clase de fórmulas y, no poder determinar cual es la respuesta correcta para otro tipo de instancias.

Todos los algoritmos consistentes y completos hasta ahora conocidos para tratar el problema SAT, determinan la respuesta para muchas instancias del problema requiriendo sólo tiempo polinomial, pero también sucede que, por el tipo de búsqueda exhaustiva que en el peor de los casos deben realizar (aún y cuando incluyan estrategias que intentan acelerar el proceso de búsqueda), necesitan de un tiempo exponencial para llegar a la solución de las instancias.

La dificultad de diseñar algoritmos consistentes y completos, proviene de la cantidad de recursos computacionales que tales algoritmos requieren. Todos los algoritmos consistentes y completos diseñados para resolver problemas NP-difíciles son de complejidad exponencial en tiempo. Puesto que, de manera general, dada una instancia, no se sabe de antemano si un algoritmo completo lo podrá resolver en tiempo polinomial o bien tendrá que realizar una búsqueda exhaustiva, y como para el peor de los casos el algoritmo necesita tiempo exponencial para hallar la solución, esta clase de algoritmos son considerados algoritmos ineficientes.

Algunos algoritmos para el problema SAT que son completos, consistentes, y de complejidad exponencial en tiempo para los peores casos son:

- El Método de Davis y Putnam
- Los procedimientos que construyen modelos mínimos
- El procedimiento de resolución
- Procedimientos que usan árboles semánticos
- Búsqueda con retroceso
- Sistemas que usan tablas sintácticas

Al tratar problemas de optimización NP-difíciles (por ejemplo, problemas de la clase APX o PTAS) los algoritmos completos son aquellos que, para cualquier instancia, encuentran el punto del dominio que es óptimo global, pero nuevamente, son algoritmos que requieren tiempo exponencial en el peor de los casos. También para esta clase de problemas existen propuestas algorítmicas incompletas, en el sentido de que son algoritmos con complejidad polinomial en tiempo y que, sin importar la forma de las instancias, encuentran soluciones aproximadas a las óptimas, es decir, hallan valores cercanos a los óptimos globales.

En general, los algoritmos que encuentran con rapidez soluciones buenas, aunque no necesariamente óptimas, se denominan *algoritmos heurísticos,* puesto que estos algoritmos frecuentemente se basan en reglas derivadas de la experiencia o del sentido común. Aunque las heurísticas tienden a ser específicas al problema, se han encontrado principios generales en algunas heurísticas que permiten su aplicación a una gran cantidad de problemas. Nótese que los algoritmos heurísticos son casos especiales de algoritmos incompletos.

Entre los algoritmos eficientes o incompletos para los problemas SAT y MaxSAT, se tienen:

- Búsqueda local
- Recocido simulado
- Búsqueda tabú
- Algunos heurísticos para la programación entera
- Estrategias voraces
- Algoritmos genéticos

Una observación acerca de estos algoritmos incompletos es que la mayoría son de naturaleza iterativa, es decir, generalmente parten de uno o varios puntos del espacio de búsqueda y en cada iteración intentan irse acercando a los puntos solución.

Dos aspectos relevantes de los algoritmos iterativos son su convergencia global y el promedio de convergencia local.

La convergencia global de un algoritmo, es un análisis que determina si es posible que a partir de un punto inicial dado, la secuencia de puntos generados por el algoritmo (puntos intermedios) pueda converger a la solución final. *El promedio de convergencia local* de un algoritmo indica el promedio esperado de puntos (o el número promedio de pasos) en la secuencia que converge a una solución [GuJ93].

La calidad de la solución encontrada por un algoritmo puede medirse de diferentes maneras, tales como medir: el error absoluto, el error relativo, el factor de aproximación de la solución, etc. Pero todas estas medidas son esencialmente idénticas y tienen la intención de medir que tan cerca al óptimo es la solución dada por el algoritmo. Entre más corta sea la distancia entre la solución hallada y el óptimo, será mejor la calidad de la solución.

En lo que respecta a la precisión y calidad de los resultados obtenidos por algoritmos para SAT y MaxSAT, todo depende de la clase de algoritmos que se trate. En el caso de algoritmos completos, la solución encontrada para SAT corresponde a una decisión correcta de la satisfactibilidad de la fórmula y a un óptimo global para el caso MaxSAT. La calidad de la respuesta que dan los algoritmos incompletos será un elemento de análisis en la determinación de la calidad del algoritmo.

En la siguiente sección se tratará precisamente sobre un estudio en el análisis de la calidad de los resultados obtenidos por una clase particular de algoritmos incompletos, conocidos como algoritmos de aproximación y en el capítulo cuatro, se presenta el análisis de los algoritmos de interés, los cuales son algoritmos iterativos que usan heurísticas valiosas en la resolución del problema MaxSAT, y que en general, puede usarse en la resolución de una gran cantidad de problemas.

3.2 Métodos para determinar el factor de garantía de los algoritmos

Existen diferentes métodos formales y probabilísticos usados en el análisis de los algoritmos, estas técnicas generalmente se utilizan para determinar la complejidad en tiempo y espacio de los algoritmos, así como en la determinación del factor de garantía de los algoritmos de aproximación. Reeves [ReC93] enuncia tres métodos generales comúnmente utilizados para la determinación del factor de garantía de los algoritmos de aproximación, los cuales son:

- Métodos Analíticos.
- Métodos Empíricos.
- Análisis Estadístico.

El método analítico se enfoca principalmente en considerar un análisis en el peor de los casos, y tal vez, un análisis en el caso promedio.

El análisis en el peor de los casos, como su nombre lo sugiere, supone encontrar una garantía teórica que permita medir el límite de la eficiencia del algoritmo sobre las peores instancias posibles. Esta análisis proporciona garantías fuertes sobre el comportamiento del algoritmo que en general pueden no llegar a alcanzarse, además de que exige intentar caracterizar el tipo de instancias que para el algoritmo serán más difíciles de resolver. Pero este análisis, que en un sentido es "pesimista", intenta encontrar una cota de lo peor que pueden ser los resultados proporcionados por el algoritmo.

El análisis en el caso promedio busca determinar el comportamiento de la ejecución del algoritmo en términos de una esperanza. Sin embargo, como con cualquier análisis probabilístico, un valor sin medida de la varianza no es de mucho uso. También, esta forma de análisis frecuentemente se asocia a un espacio probabilístico que puede no concordar con el espacio de instancias reales y entonces dará resultados sin sentido, además de que el análisis probabilístico tiende a ser extremadamente difícil.

En términos probabilísticos, el análisis en el caso promedio puede decir mucho, especialmente cuando se aplica el algoritmo a muchas instancias con características similares. Por su naturaleza, este tipo de análisis debe hacer suposiciones acerca de la distribución probabilística que tiene la clase de instancias, y si estas suposiciones no son apropiadas entonces los resultados del análisis pueden no ser exactos para las instancias sobre las que se trabajará. Más aún, este tipo de análisis puede no decir algo definitivo acerca de la ejecución del algoritmo para una instancia particular, puesto que cualquier instancia particular puede ser atípica.

La inferencia estadística es el método de estimación de parámetros de una población estadística a partir de una muestra y se ha aplicado sólo recientemente en la estimación de la calidad de la solución de las heurísticas [ReC93], en general, se conocen algunos detalles de cómo estimar una aproximación usando resultados conocidos de la teoría estadística para valores extremos. El método es prometedor, pero requiere que se ponga especial atención al algoritmo durante su ejecución.

El más común de los métodos para evaluar el comportamiento de una heurística es el análisis empírico. En general, el análisis empírico empieza por ejecutar la heurística sobre un gran número de instancias del problema, tales instancias generalmente son generadas de manera aleatoria, e intenta hacer estimaciones globales del comportamiento de la heurística sobre estas instancias. El análisis empírico puede ser el más apropiado si se trata de un problema prueba que proporciona instancias reales, pero puede ser completamente engañoso si no se tiene cuidado en la elección de las instancias de prueba, ya que estas pueden tener características muy diferentes de aquellas que se quieren resolver en la práctica.

Las cotas obtenidas de los factores de garantía del algoritmo por ambas técnicas (analítica y empírica), son cotas muy fuertes, que al momento de compararlas con respuestas del algoritmo sobre instancias reales, pueden quedar lejanas, pero en general, estas cotas conducen a un mejor entendimiento tanto de la heurística como del problema que se está tratando.

De cualquier forma, analizar una heurística es una tarea matemática desafiante. Para realizar el análisis del algoritmo es bueno tener una idea sobre el tipo de instancias que se quieren resolver en la práctica. Sin embargo, en la mayoría de los casos, las instancias a resolver pueden ser de cualquier estilo o naturaleza, por lo que es común hacer uso de instancias generadas aleatoriamente y de manera uniforme con el fin de probar el comportamiento de la heurística.

En la siguiente sección, se presentan algunos modelos comúnmente usados en la construcción de instancias de prueba, tales instancias son útiles para analizar la convergencia y precisión de los resultados que proporcionan los algoritmos diseñados para la resolución de los problemas SAT y MaxSAT.

3.3 Modelos para construir instancias de fórmulas booleanas

Como se ha mencionado, es difícil hallar un algoritmo para SAT y/o MaxSAT que tenga un excelente comportamiento para todas las instancias del problema. Es común por tanto, elegir instancias específicas para estudiar la complejidad y comportamiento de los algoritmos propuestos, por ejemplo, es útil probar nuevas propuestas algorítmicas considerando como entrada a aquellas instancias que se sabe fueron difíciles de resolver por algoritmos tradicionales (ya sea por el tiempo que se requirió o, por la precisión de los resultados obtenidos).

Algunos de los modelos más comúnmente usados en la construcción de instancias específicas para los problemas SAT y MaxSAT, son:

1) Modelo para construcción de instancias aleatorias: El uso de la aleatoriedad para la generación de instancias especificas es una técnica estándar en el diseño de algoritmos [GuJ93]. Dentro de este modelo se tienen las opciones siguientes:

    a) Modelo exacto *k*-SAT: al construir las formulas en FNC se forma cada cláusula de manera independiente a las demás. Cada cláusula contiene exactamente k literales. Cada literal l es uniformemente elegida sin reemplazo del conjunto de variables, por ejemplo para $u \in U$, $\mathrm{Prob}(l=u) = 1 / |U|$, y la probabilidad de que la variable elegida tenga una ocurrencia negativa es p (parámetro de entrada, que usualmente se usa con valor de ½).
    b) Modelo promedio *k*-SAT: Cada cláusula se construye aleatoriamente y de manera independiente de las demás cláusulas de la fórmula. Cada una de las n variables aparece positivamente con probabilidad $p/2$, negativamente con

probabilidad $p/2$ y esta ausente con probabilidad $1-p$. El número promedio de literales en cada cláusula es de k.

2) Modelo práctico: Se usan como instancias de fórmulas en FNC, las formulas generadas a partir de problemas típicos, tales como: el problema de colorear una gráfica, el problema de colocar n-reinas, etc... o bien, instancias generadas a partir de aplicaciones reales.

3) Modelo regular: Como fórmulas a procesar se usan instancias que son reconocidas o reportadas en la literatura como casos difíciles de resolver por algún algoritmo en particular.

Para entender la dificultad que cada instancia específica del problema de satisfactibilidad tiene, se definen los conceptos de dificultad de una instancia y el tipo de distribución de las instancias.

La *dificultad* de una instancia para el problema SAT es una propiedad intrínseca a la fórmula F en FNC. La dificultad de una instancia es un indicador del grado de dificultad que tendrán los algoritmos para determinar su satisfactibilidad.

Por ejemplo, una formula F con m cláusulas y n variables es más difícil de satisfacer cuando se incrementa el cociente m/n, (muchas cláusulas definidas sobre pocas variables incrementa la posibilidad de tener conjuntos insatisfactibles de cláusulas, por ejemplo: $\{x, \neg x\}$). O bien, cuando el número promedio k de literales por cláusula decrece, ya que pocas literales y un gran número de cláusulas reducen la posibilidad de hacer a todas las cláusulas satisfactibles [ChV92]. Esta es una de las razones por lo que el problema 3SAT es tan complejo como el SAT general, o el Max-2-SAT es tan difícil como el caso general MaxSAT.

Evidencias empíricas [ChV92] sugieren que existe un umbral fino que denotaremos con β tal que cuando $c<\beta$ entonces una fórmula generada aleatoriamente con $m = c \cdot n$ cláusulas es casi seguramente satisfactible y cuando $c>\beta$ tal fórmula es casi seguramente insatisfactible. En [KaA94] se conjetura que tal umbral es alrededor de 4.2, y muestran además que para $c>4.758$ una fórmula 3-FNC es insatisfactible con alta probabilidad. Tal umbral no ha sido probado de manera rigurosa, pero se conjetura que existe.

Se define la *dificultad-algorítmica* de una serie de instancias de fórmulas en FNC, como la forma que deben tener las instancias para que la determinación de la satisfactibilidad se pueda decidir fácilmente por un determinado algoritmo, así como la forma que deben tener las instancias donde el algoritmo tiene más problemas (lo que generalmente significa que requiere de más tiempo o espacio de lo normal) para decidir si una fórmula es o no satisfactible.

Entonces, la dificultad-algorítmica de las instancias depende no solo de las instancias, sino también del algoritmo. Por ejemplo con instancias que son claramente insatisfactibles

en donde existe un gran número de inconsistencias (casos frecuentes de ocurrencias del conjunto {(x), (¬x)}), un algoritmo completo para el problema SAT (como podría ser el algoritmo de Davis-Putnam) puede decidir la inconsistencia de estas instancias rápidamente, pero para algoritmos incompletos tales como las búsquedas locales, éstos podrían no darse cuenta de la inconsistencia de tales instancias. Y por el contrario, algoritmos incompletos para el problema SAT podrían decidir más rápidamente la satisfactibilidad de ciertas instancias de lo que podría hacerlo el algoritmo de Davis-Putnam.

Los elementos presentados en este capítulo son útiles en el análisis de los algoritmos implementados en este trabajo. Para el análisis empírico de las propuestas algorítmicas (para el tratamiento del problema MaxSAT), se hace uso del modelo que se conoce con el nombre de “Modelo exacto *k*-SAT” y la técnica consiste en construir la formula FNC, formando cada cláusula de manera independiente a las demás y donde cada cláusula tiene exactamente *k* literales.

Se realizó el análisis analítico para determinar el factor de garantía de uno de los algoritmos que proponemos (LS-NC_0), y se realizó el análisis empírico a cada una de las propuestas algorítmicas presentadas LS, LS-N, LS-NC_0 y LS-NTA.

La información detallada acerca del análisis empírico y analítico se encuentra desarrollada en el capítulo siguiente.

Capítulo IV

Determinación del factor de garantía y análisis experimental

En el capítulo anterior se presentaron las técnicas más comunes para el análisis de algoritmos. En este capítulo, utilizaremos los conceptos presentados para aplicarse al análisis del algoritmo propuesto. Primeramente, en la sección 4.1 se realiza un análisis teórico del factor de garantía y, posteriormente, en la sección 4.2 se realiza un análisis empírico, el cual corrobora los resultados obtenidos en la sección 4.1.

4.1 Determinación del factor de garantía

En esta sección se mostrará el poder de la búsqueda local no obvia, y se demostrará que para la función objetivo no obvia propuesta por Khanna se obtiene un factor de garantía de ¾ para Max-2-SAT y de $(2^k - 1) / 2^k$ para Max-k-SAT. Así también, se demostrará que el algoritmo asegura un factor de garantía de 4/5 para Max-2-SAT sobre aquellas fórmulas FNC que cumplen la condición expresada posteriormente en la ecuación 8 de este capítulo.

Para entender la estrategia general para la obtención del factor de garantía, recordemos que dentro de la función objetivo no obvia: S_i indica el número de cláusulas que bajo una cierta asignación I, satisfacen exactamente i literales. $W(S_i)$ representa el peso asociado al conjunto de cláusulas S_i. $f_{NOB}(z)$ es la evaluación de la función objetivo sobre la asignación z. Y además, si z_1 es el vecino de z, entonces z_1 es $(z)_j$ cuyo valor es igual a z excepto en el bit de la posición j y $f_{NOB}(z_1)$ es la evaluación de la función objetivo f sobre z_1.

Denotemos como df/dz al diferencial de la función objetivo cuando es evaluada en puntos de la misma vecindad, en este caso: $df/dz = f(z') - f(z)$, donde $z' \in$ N(z). Y cuando explícitamente se quiera indicar al vecino de x, se expresará de la manera siguiente: df/dz_j, donde $df/dz_j = f((z)_j) - f(z)$ y $(z)_j$ es z excepto en el bit de la posición j.

La estrategia seguida en el análisis del factor de garantía consiste en calcular df/dz y de esta manera se establecen los cambios que sufre la función objetivo cuando una literal cambia de un valor a otro (presupondremos que será de uno a cero binario), después se analiza que sucede cuando este cambio se realiza considerando un óptimo local. Los resultados se expresan en términos del número de cláusulas que no se satisfacen (S_0), el cual puede ser fácilmente interpretado en función del número de cláusulas que se satisfacen. A continuación se presenta la determinación del factor de garantía para la búsqueda local no obvia.

Teorema 4.1 La búsqueda local no obvia sobre una 1-vecindad tiene un factor de garantía de ¾ para Max-2-SAT.

Prueba. Consideremos la función no obvia:

$$f_{NOB}(z) = C_0 W(S_0) + C_1 W(S_1) + C_2 W(S_2)$$

Considere cualquier asignación z que sea un óptimo local con respecto a la función no obvia. Sin pérdida de generalidad, asumimos que las variables se han renombrado de tal manera que a cada literal no negada se le asigna el valor verdadero. Denotemos como $P_{i,j}$ y $N_{i,j}$ el número de cláusulas en S_i que contienen a las literales z_j y $\neg z_j$ respectivamente.

Suponga que la variable z_j cambia su valor de asignación de: verdadero a falso, este cambio se analiza a través de la tabla 4.1, según los coeficientes C_i que ponderan a los valores $P_{i,j}$ y $N_{i,j}$ respectivamente.

$z_j = 1$	$z_j = 0$	$\partial f / \partial z_j$
C_1-$P_{1,j}$	$P_{1,j} \rightarrow P_{0,j}$ – C_0	$(C_0\text{-}C_1)P_{1,j} = -\Delta_1 P_{1,j}$
C_2-$P_{2,j}$	$P_{2,j} \rightarrow P_{1,j}$ – C_1	$(C_1\text{-}C_2)P_{2,j} = -\Delta_2 P_{2,j}$
C_0-$N_{0,j}$	$N_{0,j} \rightarrow N_{1,j}$ – C_1	$(C_1\text{-}C_0)N_{0,j} = \Delta_1 N_{0,j}$
C_1-$N_{1,j}$	$N_{1,j} \rightarrow N_{2,j}$ – C_2	$(C_2\text{-}C_1)N_{1,j} = \Delta_2 N_{1,j}$

Tabla 4.1 Análisis sobre el cambio de valor de una variable z_j perteneciente a una asignación z.

De la tabla 4.1 se tiene que:

$$\frac{\partial f}{\partial z_j} = -\Delta_1 P_{1,j} - \Delta_2 P_{2,j} + \Delta_1 N_{0,j} + \Delta_2 N_{1,j}$$

Ecuación 4.1 Diferencial considerando cambio de valor lógico sobre la posición j del punto z.

Además, si hacemos la suposición de encontrarnos bajo un óptimo local se cumple que:

$$\frac{\partial f}{\partial z_j} \leq 0$$

Esto, para todo j entre 1 y n ($\forall j$=1,...,n). Por lo que se tendría, que:

$$-\Delta_1 P_{1,j} - \Delta_2 P_{2,j} + \Delta_1 N_{0,j} + \Delta_2 N_{1,j} \leq 0$$

Sumando esta desigualdad bajo todas las variables z_j (j = 1,2,...,n) y usando las igualdades:

$$\sum_{j=1}^{n} P_{1,j} = \sum_{j=1}^{n} N_{1,j} = W(S_1)$$

$$\sum_{j=1}^{n} P_{2,j} = 2W(S_2)$$

$$\sum_{j=1}^{n} N_{0,j} = 2W(S_0)$$

Se obtiene que:

$$\sum_{j=1}^{n} (-\Delta_1 P_{1,j} - \Delta_2 P_{2,j} + \Delta_1 N_{0,j} + \Delta_2 N_{1,j}) = \Delta_1 W(S_1) - \Delta_2 2W(S_2) + \Delta_1 2W(S_0) + \Delta_2 W(S_1) \leq 0$$

Y asociando según $W(S_0)$, $W(S_1)$ y $W(S_2)$,

$$-2\Delta_2 W(S_2) - (\Delta_1\text{-}\Delta_2)W(S_1) + 2\Delta_1 W(S_0) \leq 0$$

queda finalmente que:

$$2\Delta_2 W(S_2) + (\Delta_1\text{-}\Delta_2)W(S_1) \geq 2\Delta_1 W(S_0)$$

Ecuación 4.2 Relación entre el número de cláusulas satisfechas e insatisfechas.

Nótese que el número total de cláusulas satisfechas, que denotaremos por SAT, es:

$$\text{SAT} = W(S_2) + W(S_1)$$

Por lo que quisiéramos que los coeficientes del lado izquierdo de la ecuación 4.2 se igualaran a uno, es decir, que:

$$2\Delta_2 = 1 \text{ y } (\Delta_1 - \Delta_2) = 1$$

De aquí se sigue que:

$$\Delta_2 = 1/2 = C_2 - C_1$$

$$\Delta_1 = 3/2 = C_1 - C_0$$

Asumiendo C_0=0 [KhS96], se infiere que C_2=2 y C_1=3/2 y la función objetivo no obvia propuesta por Khanna será:

$$f_{NOB}(z) = C_0 W(S_0) + C_1 W(S_1) + C_2 W(S_2) = \frac{3}{2} W(S_1) + 2W(S_2)$$

además, el lado derecho de la desigualdad de la ecuación 4.2 será:

$$2\Delta_1 W(S_0) = 2\cdot(3/2)\ W(S_0) = 3\ W(S_0)$$

Quedando la ecuación 4.2 como:

$$W(S_2) + W(S_1) \geq 3W(S_0)$$

Dado que se había planteado la utilización de una función de peso simple, significando esto que $W(S_i)$ es simplemente igual al número de cláusulas pertenecientes al conjunto S_i, se cumple que:

$$W(S_0) + W(S_1) + W(S_2) = m$$

Nótese que el total de cláusulas insatisfechas, que denotaremos por INSAT, es:

$$\text{INSAT} = W(S_0)$$

Y por tal, se tiene que al sumar $W(S_0)$ en ambos lados de la ecuación 4.2, nos genera la siguiente desigualdad.

$$4W(S_0) \leq m$$

Esto inmediatamente implica que el total de cláusulas insatisfechas en un óptimo local no es mayor que ¼ del total de todas las cláusulas; esto es, el algoritmo asegura un factor de garantía de ¾ .

A continuación se presenta la versión generalizada (para Max-k-SAT, donde k es el número máximo de literales que hay en alguna cláusula de la fórmula booleana de entrada) de la obtención del factor de garantía para la búsqueda local no obvia con la función objetivo propuesta por Khanna [KhS96].

Teorema 4.2 La búsqueda no obvia sobre 1-vecindad tiene un factor de garantía de $(2^k\text{-}1) / 2^k$ para Max-k-SAT.

Prueba: De nuevo, sin pérdida de generalidad, asumimos que las variables han sido renombradas de tal manera que cada literal no negada bajo la asignación actual se le asigna al valor verdadero. Así el conjunto S_i es el conjunto de cláusulas con exactamente 'i' literales verdaderas.

Sea :

$$\Delta_i = C_i - C_{i-1}$$

Y denotemos con:

$$\frac{\partial f}{\partial z_j}$$

al cambio del valor de la función cuando cambiamos el valor de z_j cuando esta variable pasa del valor uno a cero.

Se obtiene que:

$$\frac{\partial f}{\partial z_j} = -\Delta_k P_{k,j} + \sum_{i=2}^{k} (\Delta_i N_{i-1,j} - \Delta_{i-1} P_{i-1,j}) + \Delta_1 N_{0,j}$$

Esto a partir del análisis en el cambio de valor de z_j mostrado en la tabla 4.2.

$z_j = 1$	$z_j = 0$	$\partial f / \partial z_j$
C_1-$P_{1,j}$	$P_{1,j} \rightarrow P_{0,j}$ - C_0	$(C_0 - C_1)P_{1,j} = -\Delta_1 P_{1,j}$
C_2-$P_{2,j}$	$P_{2,j} \rightarrow P_{1,j}$ - C_1	$(C_1 - C_2)P_{2,j} = -\Delta_2 P_{2,j}$
:	:	:
:	:	:
C_{k-1}-$P_{k-1,j}$	$P_{k-1,j} \rightarrow P_{k-2,j}$ - C_{k-2}	$(C_{k-2} - C_{k-1})P_{k-1,j} = -\Delta_{k-1} P_{k-1,j}$
C_k-$P_{k,j}$	$P_{k,j} \rightarrow P_{k-1,j}$ - C_{k-1}	$(C_{k-1} - C_k)P_{k,j} = -\Delta_k P_{k,j}$
C_0-$N_{0,j}$	$N_{0,j} \rightarrow N_{1,j}$ - C_1	$(C_1 - C_0)N_{0,j} = \Delta_1 N_{0,j}$
C_1-$N_{1,j}$	$N_{1,j} \rightarrow N_{2,j}$ - C_2	$(C_2 - C_1)N_{1,j} = \Delta_2 N_{1,j}$
:	:	:
:	:	:
C_{k-2}-$N_{k-2,j}$	$N_{k-2-,j} \rightarrow N_{k-1,j}$ - C_{k-1}	$(C_{k-1} - C_{k-2})N_{k-2,j} = \Delta_{k-1} N_{k-2,j}$
C_{k-1}-$N_{k-1,j}$	$N_{k-1,j} \rightarrow N_{k,j}$ - C_k	$(C_k - C_{k-1})N_{k-1,j} = \Delta_k N_{k-1,j}$

Tabla 4.2 Análisis sobre el cambio de valor de una variable z_j perteneciente a una asignación z.

Y dado que sabemos que cuando el algoritmo termina, como z es un óptimo local, entonces:

$$\frac{\partial f}{\partial z_j} \leq 0 \quad , \forall j \in \{1, ..., n\}$$

Entonces se cumple que:

$$-\Delta_k P_{k,j} + \sum_{i=2}^{k} (\Delta_i N_{i-1,j} - \Delta_{i-1} P_{i-1,j}) + \Delta_1 N_{0,j} \leq 0$$

La cual puede ser escrita como:

$$-\Delta_k P_{k,j} + \sum_{i=1}^{(k-1)} (\Delta_{i+1} N_{i,j} - \Delta_i P_{i,j}) + \Delta_1 N_{0,j} \leq 0$$

Ecuación 4.3 Efecto del cambio de valor de la variable z_j bajo un óptimo local.

Sumando bajo todas las variables z_j ($j = 1,2,...,n$) la desigualdad de la ecuación 4.3 se transforma en:

$$\sum_{j=1}^{n} (-\Delta_k P_{k,j} + \sum_{i=1}^{(k-1)} (\Delta_{i+1} N_{i,j} - \Delta_i P_{i,j}) + \Delta_1 N_{0,j}) \leq 0$$

ó

$$-\Delta_k \sum_{j=1}^{n} P_{k,j} + \sum_{i=1}^{(k-1)} (\Delta_{i+1} \sum_{j=1}^{n} N_{i,j} - \Delta_i \sum_{j=1}^{n} P_{i,j}) + \Delta_1 \sum_{j=1}^{n} N_{0,j} \leq 0$$

y usando las igualdades,

$$\sum_{j=1}^{n} P_{i,j} = iW(S_i)$$

$$\sum_{j=1}^{n} N_{i,j} = (k-i)W(S_i)$$

Se obtiene que:

$$-k\Delta_k W(S_k) + \sum_{i=1}^{(k-1)} ((k-i)\Delta_{i+1} W(S_i) - i\Delta_i W(S_i)) + k\Delta_1 W(S_0) \leq 0$$

Despejando y factorizando, obtenemos la siguiente desigualdad:

$$k\Delta_1 W(S_0) \leq k\Delta_k W(S_k) + \sum_{i=1}^{(k-1)} (i\Delta_i - (k-i)\Delta_{i+1}) W(S_i)$$

Ecuación 4.4 Relación entre el número de cláusulas satisfechas e insatisfechas.

Ahora, nótese que el número total de cláusulas satisfechas, SAT, es:

$$\text{SAT} = \sum_{i=1}^{k} W(S_i)$$

Y para obtener el valor SAT del lado derecho de la ecuación 4.4, tendremos que igualar a uno los coeficientes:

$$(i\Delta_i - (k - i)\Delta_{i+1}) = 1 \qquad , \forall i \in [1, k\text{-}1]$$

y

$$k\Delta_k = 1$$

De aquí se sigue que:

$$\Delta_i = (1 + (k - i)\Delta_{i+1}) / i \qquad , \forall i \in [1, k\text{-}1] \text{ y}$$

y

$$\Delta_k = 1/ k$$

Por lo que:

$$\Delta_i = \frac{1}{(k-i+1)\binom{k}{i-1}} \sum_{i=1}^{k-i} \binom{k}{j}$$

Quedándonos la siguiente desigualdad:

$$(2^k - 1)W(S_0) \leq W(S_k) + \sum_{i=1}^{k-1} W(S_i)$$

Por lo que el adicionar $W(S_0)$ en ambos lados de la desigualdad, nos genera la siguiente ecuación:

$$2^k W(S_0) \leq W(S_k) + \sum_{i=1}^{k-1} W(S_i) + W(S_0) = \sum_{i=0}^{k} W(S_i) = m$$

Es decir,

$$2^k W(S_0) \leq m$$

Esto inmediatamente implica que el total de cláusulas insatisfechas al terminar el algoritmo no es mayor a $1/2^k$ del total de todas las cláusulas; esto es, el algoritmo asegura un factor de garantía de $(2^k\text{-}1) / 2^k$, para cualquier fórmula en *k*-FNC.

Analicemos ahora el efecto de considerar la nueva función objetivo no obvia que proponemos para Max-2-SAT,

$$f'_{NOB}(z) = \frac{3}{2}W(S_1) + 2W(S_2) - W(S_0)$$

De nuevo, sin pérdida de generalidad, asumimos que las variables han sido renombradas de tal manera que cada literal no negada bajo la asignación actual se le asigna al valor verdadero. Así el conjunto S_i es el conjunto de cláusulas con exactamente 'i' literales verdaderas, y sea: $\Delta_i = C_i - C_{i-1}$.

Del teorema 4.1, donde se obtuvo que se cumple para toda función objetivo no obvia la desigualdad :

$$2\Delta_1 W(S_0) \leq 2\Delta_2 W(S_2) + (\Delta_1 - \Delta_2) W(S_1)$$

Ecuación 4.5 Relación entre el número de cláusulas satisfechas e insatisfechas.

Así que, se quiere determinar valores para C_0, C_1 y C_2 de tal manera que Δ_1 sea un entero positivo grande, mientras que Δ_2 y (Δ_1-Δ_2) sean ambos, enteros positivos pequeños. Definiendo C_0 = -1, C_1 = 3/2 y C_2 = 2, se obtiene que Δ_1 = 5/2 y Δ_2 = 1/2. Entonces, la ecuación 4.5 se transforma en:

$$5W(S_0) \leq W(S_2) + 2W(S_1)$$

Dado que:

$$\text{SAT} = \sum_{i=1}^{k} W(S_i)$$

Para el caso Max-2-SAT, SAT = $W(S_1)$ + $W(S_2)$, y dado que el número total de cláusulas es $W(S_0) + W(S_1) + W(S_2) = m$, entonces el número de cláusulas que no se satisfacen que hemos denotado por INSAT es:

$$\text{INSAT} = W(S_0) \text{ y SAT} + \text{INSAT} = m$$

Ahora, para el tipo de instancias para los cuales se cumple la relación:

$$W(S_1) \leq W(S_0)$$

Ecuación 4.6 Condicional de la relación entre cláusulas insatisfechas y $W(S_1)$.

se puede reescribir la ecuación de la siguiente manera:

$$5 \cdot \text{INSAT} \leq \text{SAT}+W(S_1) \leq \text{SAT}+W(S_0) = \text{SAT}+\text{INSAT} = m$$

Obteniendo que,

$$5 \cdot \text{INSAT} \leq m$$

Esto inmediatamente implica que el total de cláusulas insatisfechas en éste óptimo local no es mayor que 1/5 del total de todas las cláusulas; esto es, el algoritmo asegura un factor de garantía de 4/5 para aquellas fórmulas que cumplen la condición expresada en la ecuación 4.6.

A partir de este momento se puede afirmar que la calidad de soluciones que encuentre un algoritmo, que utilice la nueva función objetivo no obvia, será mejor que la de los algoritmos que utilicen la función objetivo no obvia introducida por Khanna.

Hubo dos elementos que nos llevaron a pensar en que se puede definir una función objetivo no obvia diferente a la propuesta por Khanna, tales elementos fueron:

1.- El buscar que en el lado izquierdo de la ecuación 4.2 quede el valor SAT facilita la obtención del factor de garantía, además de que se cumple para toda fórmula booleana, sin embargo no explota totalmente la desigualdad ya que podemos encontrar diferentes relaciones entre $W(S_0)$ con $W(S_1)$ y/o $W(S_2)$ que permitirían obtener mejores factores de garantía.

2.- El análisis de los resultados que se obtenían por la búsqueda local basada en la función objetivo propuesta por Khanna, mostraba un decremento en el factor de aproximación con fórmulas donde había un gran número de cláusulas insatisfechas, esto mostraba que el coeficiente C_0 era importante para esta clase de fórmulas y el hecho de que Khanna minimizara su importancia indicaba que no habían sido capturados todos los parámetros que influyen en la técnica.

Los resultados teóricos obtenidos en esta sección han sido corroborados en la práctica. En la sección siguiente presentamos los resultados del análisis empírico realizado a los algoritmos: LS, LS-N, LS-NC$_0$ y LS-NTA, trabajando éstos sobre fórmulas booleanas en k-FNC generadas aleatoriamente.

4.2 Análisis experimental

El análisis experimental sobre los algoritmos: LS, LS-N, LS-NC$_0$ y LS-NTA consideró dos clases de instancias: por una parte, se retomaron un conjunto de instancias en 2-FNC generadas por Altamirano [AlL96] y que se presentan en la tabla 4.3 (de aquí en adelante las llamaremos ALT2SAT), y que resultaron ser útiles para mostrar que la calidad

de los resultados obtenidos son competitivos con los que se obtienen al aplicar técnicas basadas en programación semidefinida [GoM94],[GoM94a] y el algoritmo ávido de Johnson [JoD74]. El grupo de instancias ALT2SAT está formado por pequeños grupos de instancias distribuidos como se muestra en la tabla 4.3, los resultados obtenidos mostraban que estas instancias no eran lo suficientemente difíciles; lo que llevó a considerar otra clase de instancias para realizar el análisis empírico.

Se hizo uso de uno de los modelos para la construcción aleatoria de instancias de formulas booleanas presentado en el capítulo III; este modelo se conoce con el nombre de "Modelo exacto *k*-SAT" y la técnica consiste en construir una fórmula en FNC, formando cada cláusula de manera independiente a las demás. Cada cláusula debe contener exactamente *k* literales.

Cada literal *l* es uniformemente elegida sin reemplazo del conjunto de variables, por ejemplo para $u \in U$, $\text{Prob}(l=u) = 1 / |U|$, y la probabilidad de que la variable elegida tenga una ocurrencia negativa es *p* (parámetro de entrada, que en este caso, se utilizó con un valor de ½).

Mediante el modelo exacto *k*-SAT se generó un grupo de 160 instancias del tipo 2-FNC y un grupo de 160 instancias del tipo 3-FNC (de aquí en adelante las llamaremos INS2SAT e INS3SAT respectivamente). Las 160 instancias de INS2SAT y las de INS3SAT se agruparon de la manera que se muestra en la tabla 4.4.

Grupo	**Instancias (*F*)**	***n* – No. de variables**	***m* – No. de cláusulas**
1	De la instancia 1 a la instancia 10	15 variables	25 cláusulas
2	De la instancia 11 a la instancia 20	15 variables	50 cláusulas
3	De la instancia 21 a la instancia 30	15 variables	75 cláusulas
4	De la instancia 31 a la instancia 40	15 variables	100 cláusulas
5	De la instancia 41 a la instancia 50	20 variables	25 cláusulas
6	De la instancia 51 a la instancia 60	20 variables	50 cláusulas
7	De la instancia 61 a la instancia 70	20 variables	75 cláusulas
8	De la instancia 71 a la instancia 80	20 variables	100 cláusulas
9	De la instancia 81 a la instancia 90	25 variables	25 cláusulas
10	De la instancia 91 a la instancia 100	25 variables	50 cláusulas
11	De la instancia 101 a la instancia 110	25 variables	75 cláusulas
12	De la instancia 111 a la instancia 120	25 variables	100 cláusulas

Tabla 4.3 Distribución de variables y cláusulas para el grupo de instancias ALT2SAT.

Grupo	Instancias (*F*)	*n* – No. de variables	*m* – No. de cláusulas
1	De la instancia 1 a la instancia 10	25 variables	50 cláusulas
2	De la instancia 11 a la instancia 20	25 variables	75 cláusulas
3	De la instancia 21 a la instancia 30	25 variables	100 cláusulas
4	De la instancia 31 a la instancia 40	25 variables	125 cláusulas
5	De la instancia 41 a la instancia 50	50 variables	100 cláusulas
6	De la instancia 51 a la instancia 60	50 variables	150 cláusulas
7	De la instancia 61 a la instancia 70	50 variables	200 cláusulas
8	De la instancia 71 a la instancia 80	50 variables	250 cláusulas
9	De la instancia 81 a la instancia 90	75 variables	150 cláusulas
10	De la instancia 91 a la instancia 100	75 variables	225 cláusulas
11	De la instancia 101 a la instancia 110	75 variables	300 cláusulas
12	De la instancia 111 a la instancia 120	75 variables	375 cláusulas
13	De la instancia 121 a la instancia 130	100 variables	200 cláusulas
14	De la instancia 131 a la instancia 140	100 variables	300 cláusulas
15	De la instancia 141 a la instancia 150	100 variables	400 cláusulas
16	De la instancia 151 a la instancia 160	100 variables	500 cláusulas

Tabla 4.4 Tamaño de las instancias construidas tanto para INS2SAT como para INS3SAT.

Como se puede observar, cada renglón en las tablas 4.3 y 4.4 representa un grupo conformado por 10 instancias (fórmulas) diferentes con el mismo número de variables y el mismo número de cláusulas. Para realizar el análisis empírico, primeramente se obtuvo el valor exacto de MaxSAT para cada instancia (el número máximo de cláusulas que pueden satisfacerse para cada instancia), posteriormente se ejecutó 10 veces cada algoritmo sobre cada instancia, los resultados obtenidos de las 10 ejecuciones sobre una instancia dada se promediaron y posteriormente se dividió este promedio entre el valor exacto de la instancia, dándonos el cociente de aproximación para la instancia, el cociente de aproximación obtenido en cada instancia del grupo de 10 instancias del mismo tipo se promedia con los 10 cocientes de aproximación del grupo, y el resultado es el que se presenta como cociente de aproximación en la tabla 4.5, en este caso particular, para el grupo de instancias denominado ALT2SAT.

El mismo proceso se sigue para obtener los cocientes de aproximación de cada uno de los algoritmos probados sobre las instancias: INS2SAT e INS3SAT, con la diferencia de que el cociente de aproximación no puede ser obtenido sobre las instancias que poseen más de 75 variables por cláusula, debido a la dificultad de obtener el valor exacto, así que, para este tipo de instancias, el procedimiento seguido fue dividir el promedio del valor obtenido de ejecutar 10 veces cada instancia entre el número total de cláusulas en la formula; el resto del proceso es igual.

La igualdad de competencia para la ejecución de los algoritmos es esencial, dado que podría sugerirse una inclinación de los resultados para obtener mejores soluciones para un algoritmo específico. Por esto, se incluyen gráficas que indican el número de cambios en promedio realizados antes de llegar a un óptimo. El lector podría pensar que un buen criterio en la igualdad de competencia podría ser los tiempos de ejecución, sin embargo, desde el punto de vista del autor, los tiempos de ejecución pueden no señalar de manera directa una igualdad en la competencia de dos algoritmos. En nuestro caso, dado que se trata de algoritmos basados en búsqueda local, resulta conveniente comparar los cambios que realiza cada algoritmo. Por un *cambio* denotamos el hecho de saltar de una asignación z a otra asignación z_1, donde el valor de la función objetivo sobre la asignación z_1 sea mejor que sobre la asignación z (es decir, $f(z_1) > f(z)$). Las tablas 4.6, 4.8 y 4.10 muestran los *cambios* en promedio realizados por cada algoritmo sobre las instancias propuestas; enseguida de estas tablas se grafican los resultados que se presentaron de manera tabular.

A continuación se presenta tabular y gráficamente, los resultados obtenidos de los algoritmos LS, LS-N, LS-NC$_0$ y LS-NTA al ejecutarse primeramente sobre las instancias ALT2SAT y después sobre las instancias INS2SAT e INS3SAT.

			Cociente de Aproximación			
Grupo	**Variables**	**Cláusulas**	**LS**	**LS-N**	**LS-NCo**	**LS-NTA**
1	15	25	0.992	0.996	0.998	0.999
2	15	50	0.983	0.99	0.991	0.994
3	15	75	0.991	0.991	0.992	0.995
4	15	100	0.991	0.992	0.992	0.996
5	20	25	0.987	0.992	0.995	0.996
6	20	50	0.987	0.983	0.988	0.993
7	20	75	0.985	0.987	0.992	0.995
8	20	100	0.992	0.992	0.993	0.996
9	25	50	0.986	0.989	0.99	0.994
10	25	75	0.984	0.986	0.989	0.993
11	25	100	0.987	0.992	0.992	0.996
12	25	125	0.989	0.992	0.992	0.996

Tabla 4.5 Cocientes de aproximación de cada algoritmo ejecutado sobre las instancias: ALT2SAT.

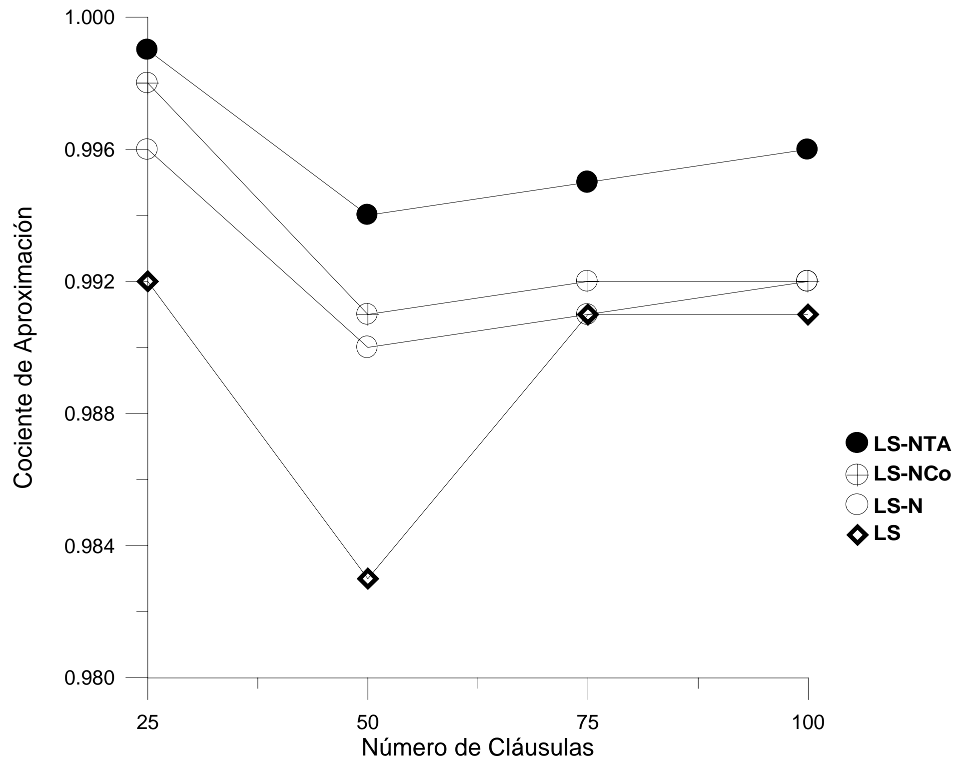

Figura 4.1 Cociente de aproximación de los algoritmos para 2-FNC con 15 variables.

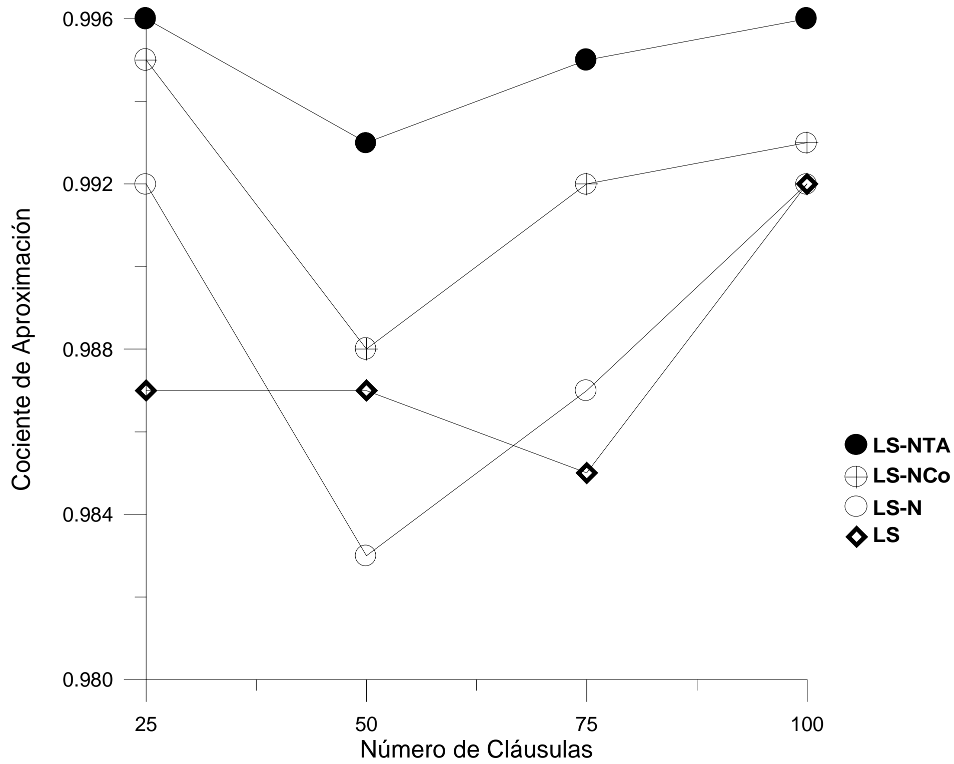

Figura 4.2 Cociente de aproximación de los algoritmos para 2-FNC con 20 variables.

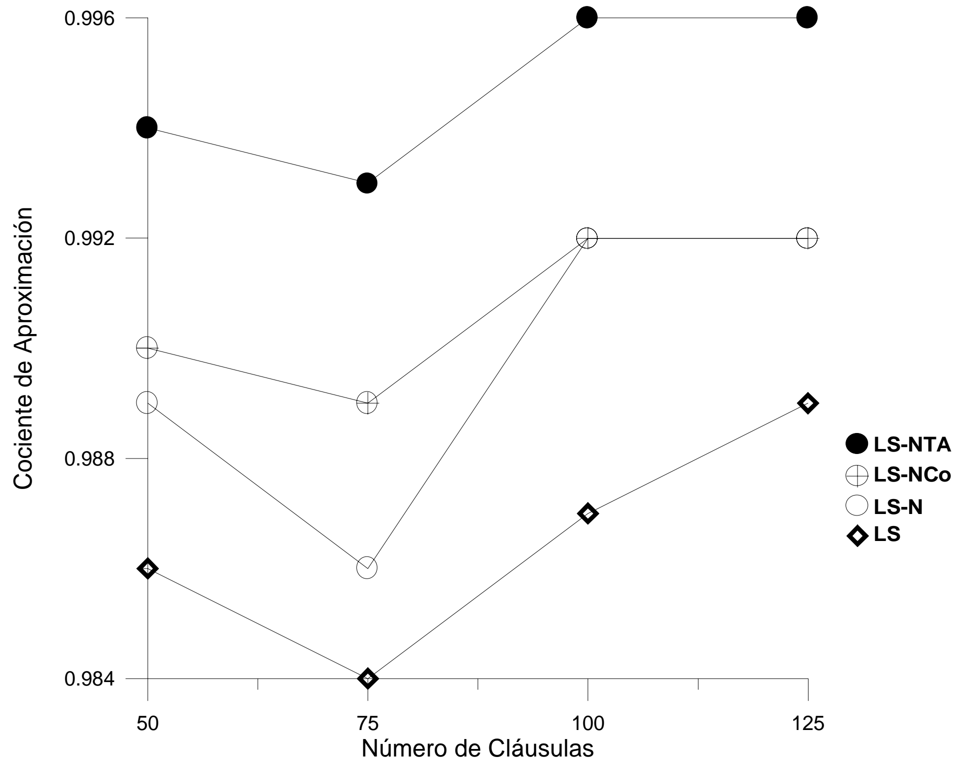

Figura 4.3 Cociente de aproximación de los algoritmos para 2-FNC con 25 variables.

			Promedio del número de cambios			
Grupo	**Variables**	**Cláusulas**	**LS**	**LS-N**	**LS-NCo**	**LS-NTA**
1	15	25	86.2	46.89	49.36	100.55
2	15	50	60.4	42.21	44.75	102.7
3	15	75	58.01	40.61	41.92	109.04
4	15	100	63.63	40.19	43.08	106.26
5	20	25	343.25	96.85	102.18	132.14
6	20	50	174.66	81.06	77.76	138.08
7	20	75	130.26	72.69	80.96	132.86
8	20	100	142.07	72.3	77.17	158.68
9	25	50	544.6	139.95	142.69	176.52
10	25	75	322.38	97.45	103.6	187.38
11	25	100	269.71	105.15	119.35	176.66
12	25	125	205.18	83.14	90.19	194.79

Tabla 4.6 Promedio del número de cambios realizados por cada algoritmo sobre las instancias: ALT2SAT.

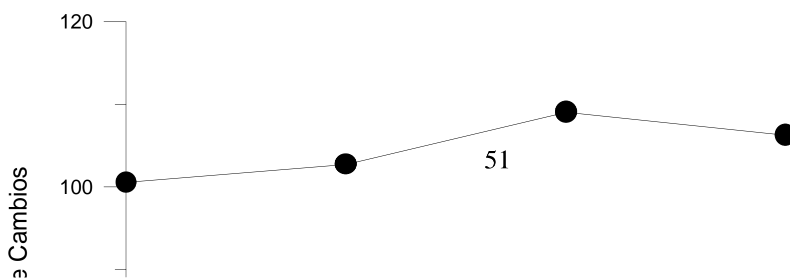

Figura 4.4 Promedio del número de cambios realizados por cada algoritmo para 2-FNC con 15 variables.

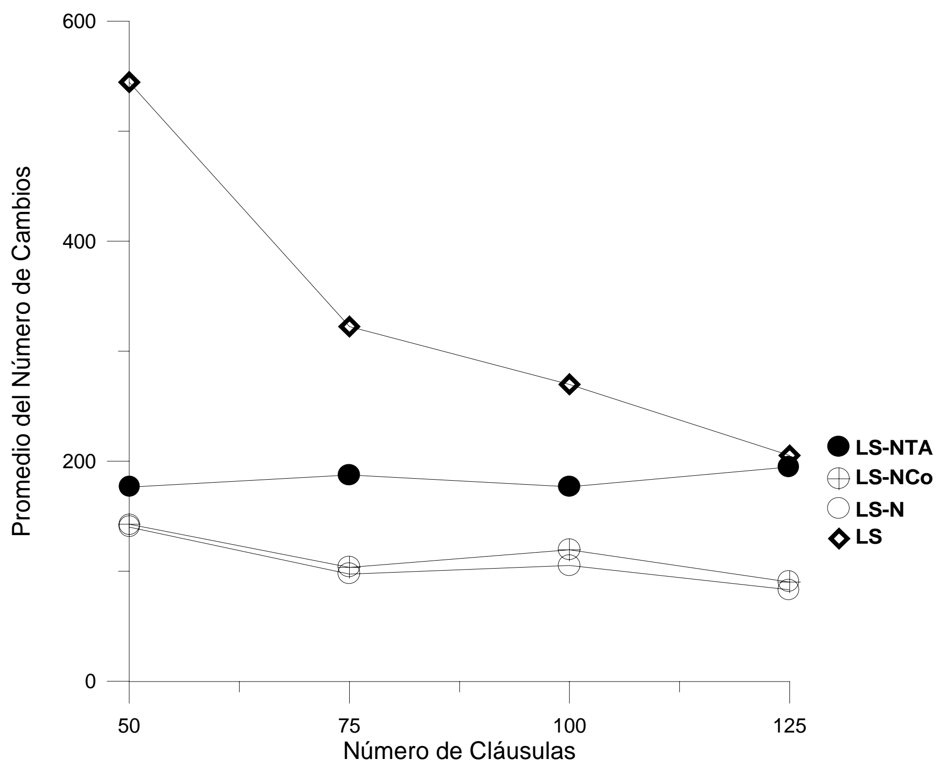

Figura 4.5 Promedio del número de cambios realizados por cada algoritmo para 2-FNC con 25 variables.

	Cociente de Aproximación

Grupo	Variables	Cláusulas	LS	LS-N	LS-NCo	LS-NTA
1	25	50	0.978	0.994	0.994	0.997
2	25	75	0.939	0.942	0.943	0.945
3	25	100	0.91	0.907	0.91	0.916
4	25	125	0.906	0.9	0.907	0.917
5	50	100	0.949	0.958	0.959	0.961
6	50	150	0.927	0.929	0.932	0.937
7	50	200	0.918	0.918	0.92	0.926
8	50	250	0.897	0.893	0.898	0.903
9	75	150	0.953	0.959	0.96	0.963
10	75	225	0.921	0.923	0.924	0.928
11	75	300	0.911	0.909	0.913	0.918
12	75	375	0.899	0.897	0.902	0.904
13	100	200	0.957	0.963	0.967	0.969
14	100	300	0.926	0.928	0.932	0.935
15	100	400	0.913	0.916	0.917	0.92
16	100	500	0.897	0.897	0.901	0.901

Tabla 4.7 Cocientes de aproximación de los algoritmos sobre las instancias INS2SAT.

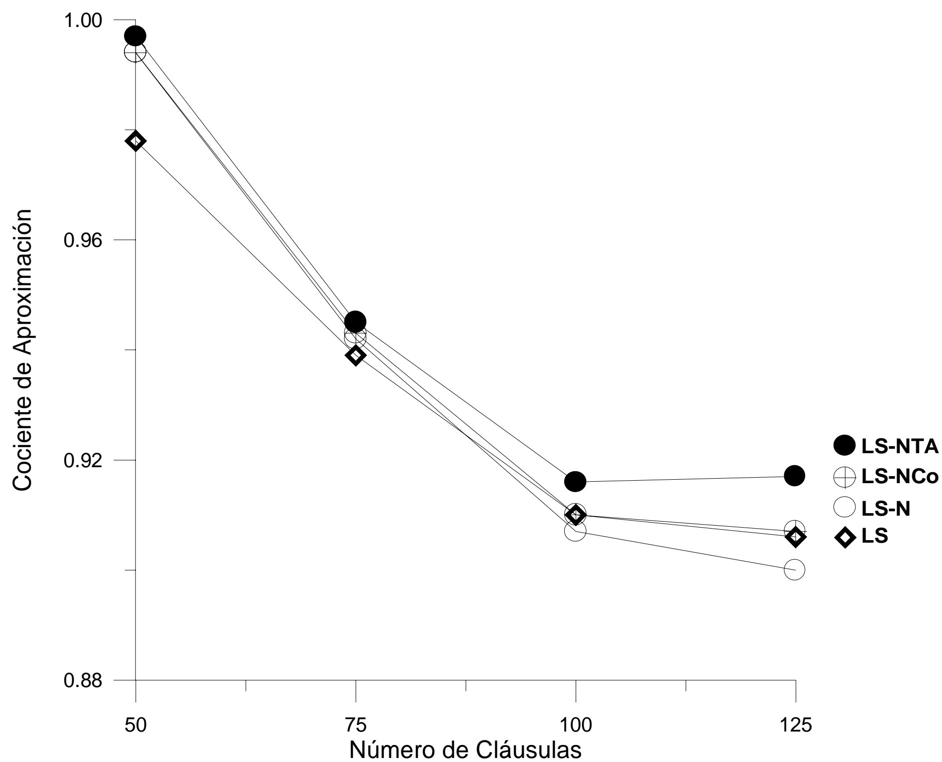

Figura 4.6 Cociente de aproximación de los algoritmos para 2-FNC con 25 variables.

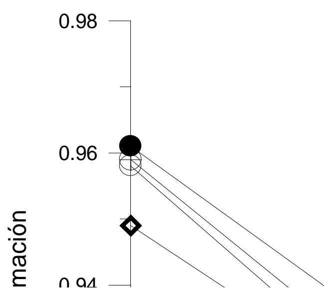

Figura 4.7 Cociente de aproximación de los algoritmos para 2-FNC con 50 variables.

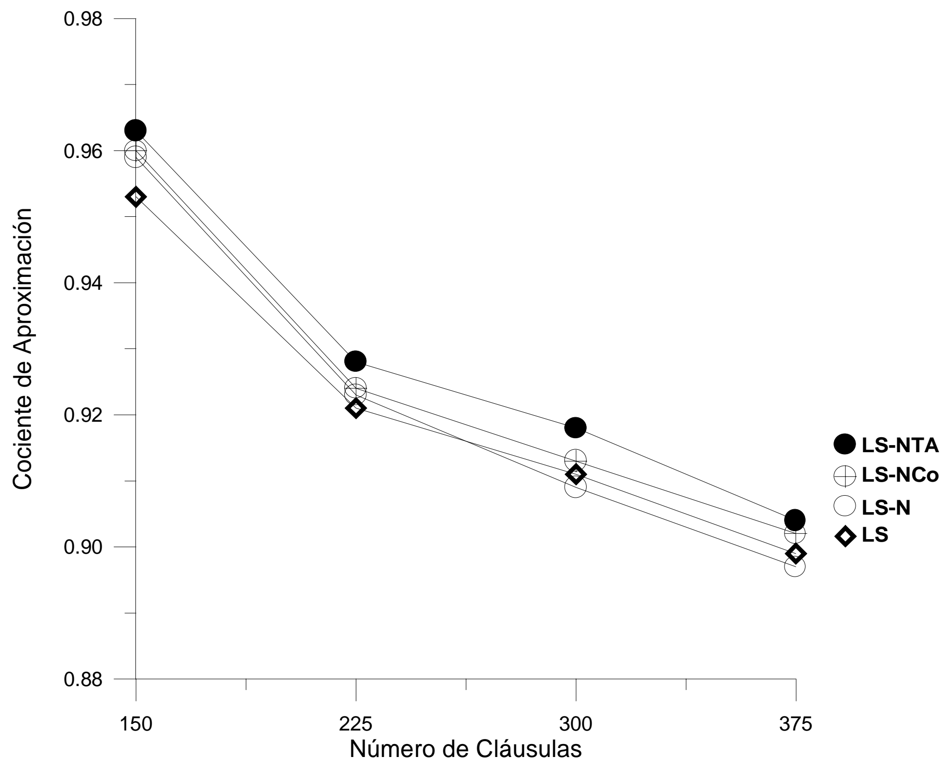

Figura 4.8 Cociente de aproximación de los algoritmos para 2-FNC con 75 variables.

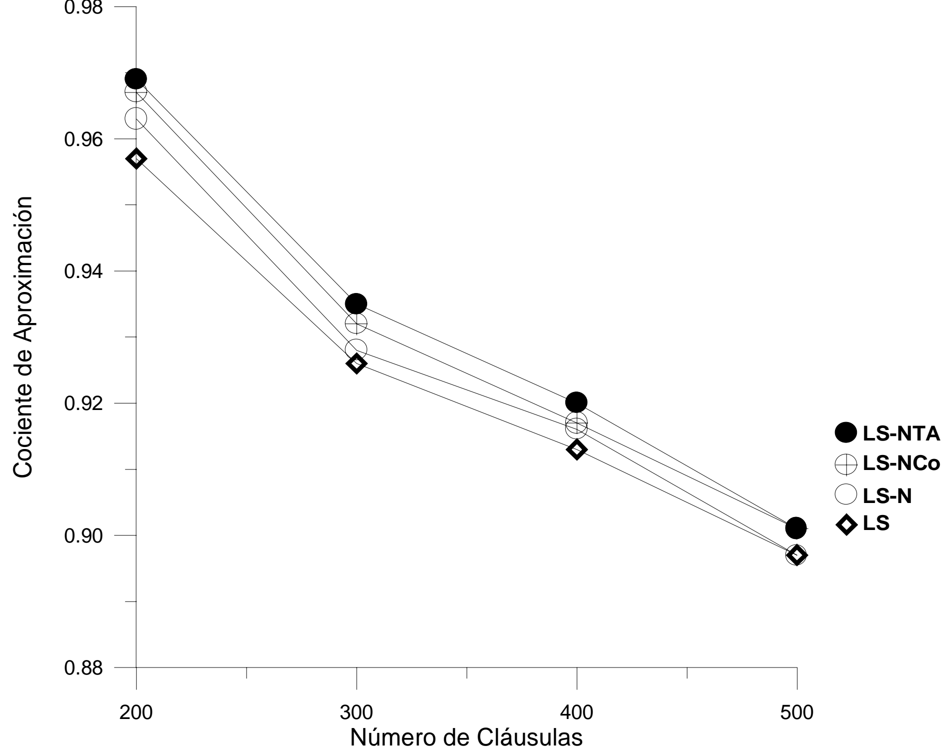

Figura 4.9 Cociente de aproximación de los algoritmos para 2-FNC con 100 variables.

			Promedio del número de cambios			
Grupo	**Variables**	**Cláusulas**	**LS**	**LS-N**	**LS-NCo**	**LS-NTA**
1	25	50	408.12	73.91	172.45	80.31
2	25	75	349.9	107.33	175.9	95.8
3	25	100	287.21	93.85	207.76	90.7
4	25	125	198.37	92.34	195.94	98.54
5	50	100	2844.98	849.73	369.92	812.04
6	50	150	3290.79	539.32	387.53	445.26
7	50	200	2965.81	403.25	410.59	417.64
8	50	250	2600.42	319.07	474.19	397.88
9	75	150	4194.46	5578.87	529.85	5247.66
10	75	225	4920.63	2046.88	581.78	2973.19
11	75	300	5584.94	2070.99	684.44	2279.02
12	75	375	5938.411	973.42	727.27	1194.09
13	100	200	5831.26	8038.34	819.67	8741.091
14	100	300	6592.28	6737.06	835.56	7131.56
15	100	400	7423.141	2915.01	994.88	3447.67
16	100	500	7750.49	2787.76	1164.56	3824.47

Tabla 4.8 Promedio del número de cambios realizados por cada algoritmo sobre las instancias: INS2SAT.

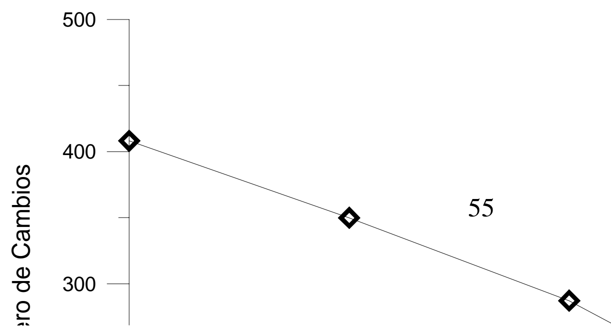

Figura 4.10 Promedio del número de cambios realizados por cada algoritmo para 2-FNC con 25 variables.

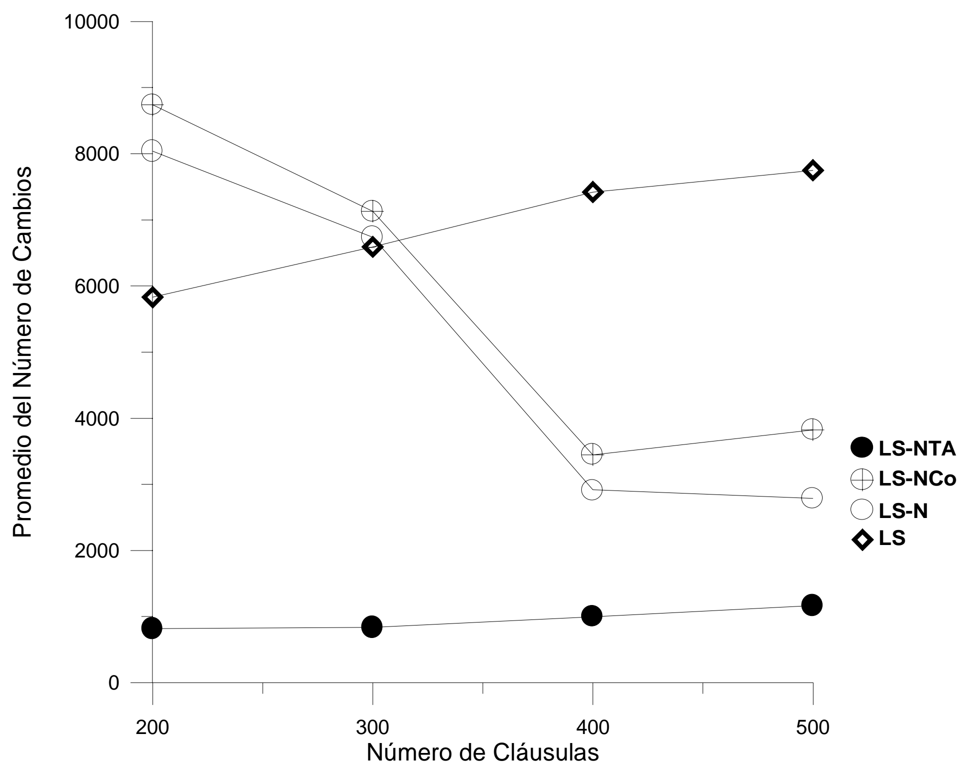

Figura 4.11 Promedio del número de cambios realizados por cada algoritmo para 2-FNC con 100 variables.

			Cociente de Aproximación			
Grupo	**Variables**	**Cláusulas**	**LS**	**LS-N**	**LS-NCo**	**LS-NTA**
1	25	50	0.999	1	1	1

2	25	75	0.997	0.999	0.999	1
3	25	100	0.982	0.985	0.986	0.988
4	25	125	0.975	0.978	0.979	0.981
5	50	100	0.995	0.997	0.999	1
6	50	150	0.989	0.995	0.996	0.997
7	50	200	0.982	0.986	0.987	0.991
8	50	250	0.975	0.979	0.98	0.982
9	75	150	0.996	0.999	1	1
10	75	225	0.988	0.992	0.993	0.994
11	75	300	0.985	0.989	0.989	0.993
12	75	375	0.975	0.978	0.979	0.981
13	100	200	0.996	1	1	1
14	100	300	0.989	0.995	0.996	0.997
15	100	400	0.983	0.982	0.986	0.99
16	100	500	0.976	0.979	0.981	0.983

Tabla 4.9 Cocientes de aproximación de los algoritmos sobre las instancias: INS3SAT.

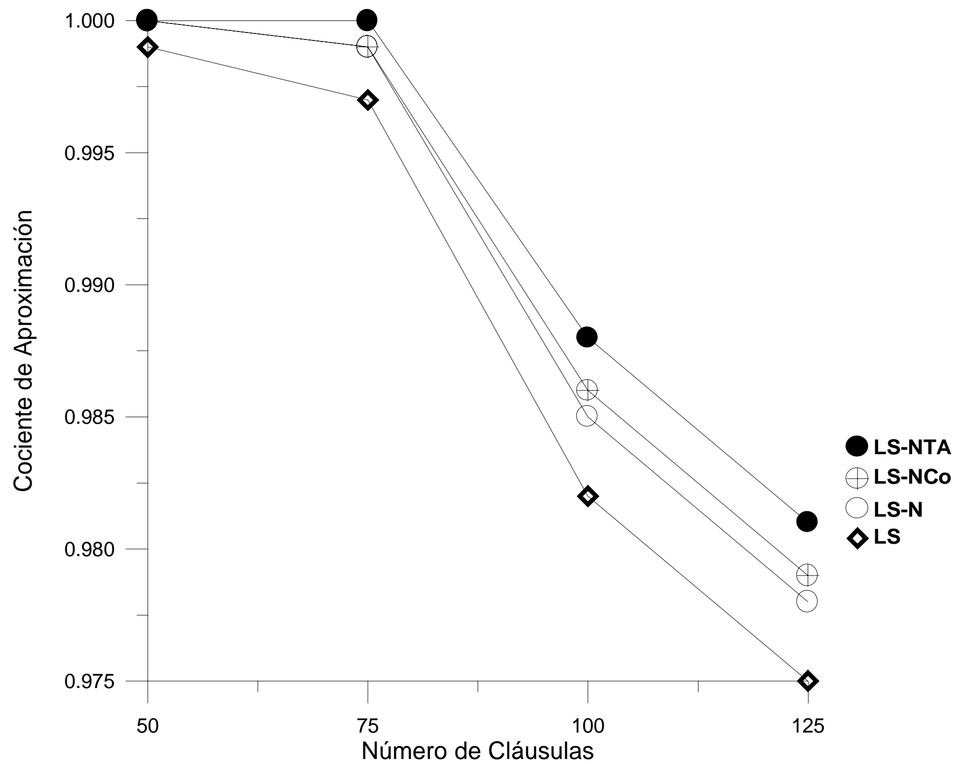

Figura 4.12 Cociente de aproximación de los algoritmos para 3-FNC con 25 variables.

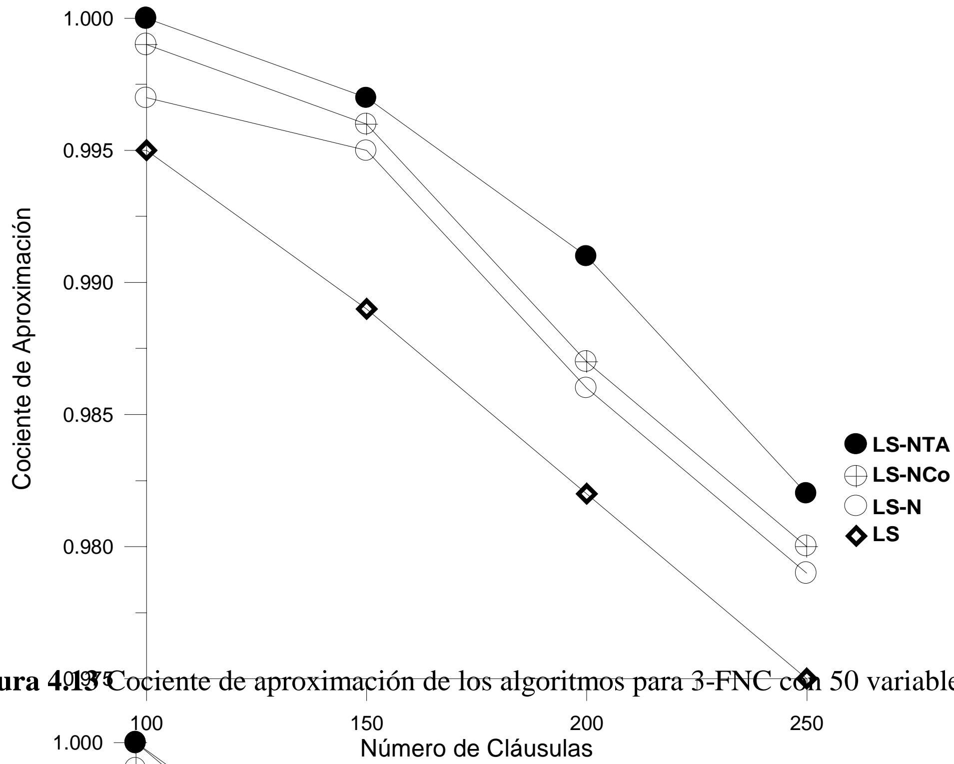

Figura 4.13 Cociente de aproximación de los algoritmos para 3-FNC con 50 variables.

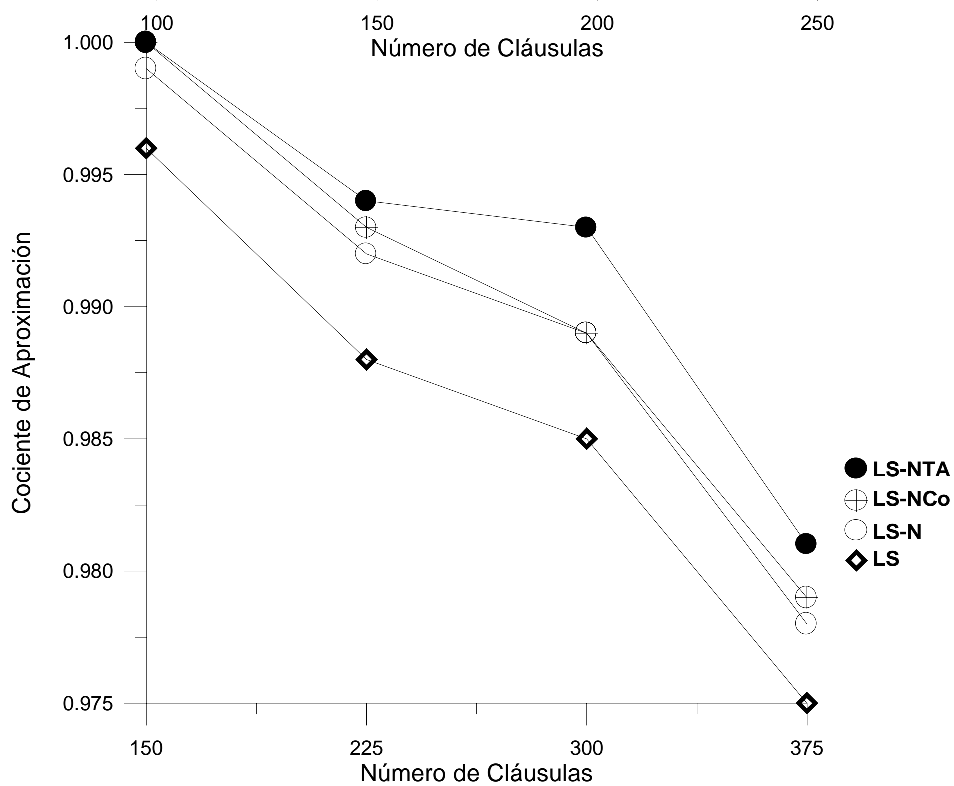

Figura 4.14 Cociente de aproximación de los algoritmos para 3-FNC con 75 variables.

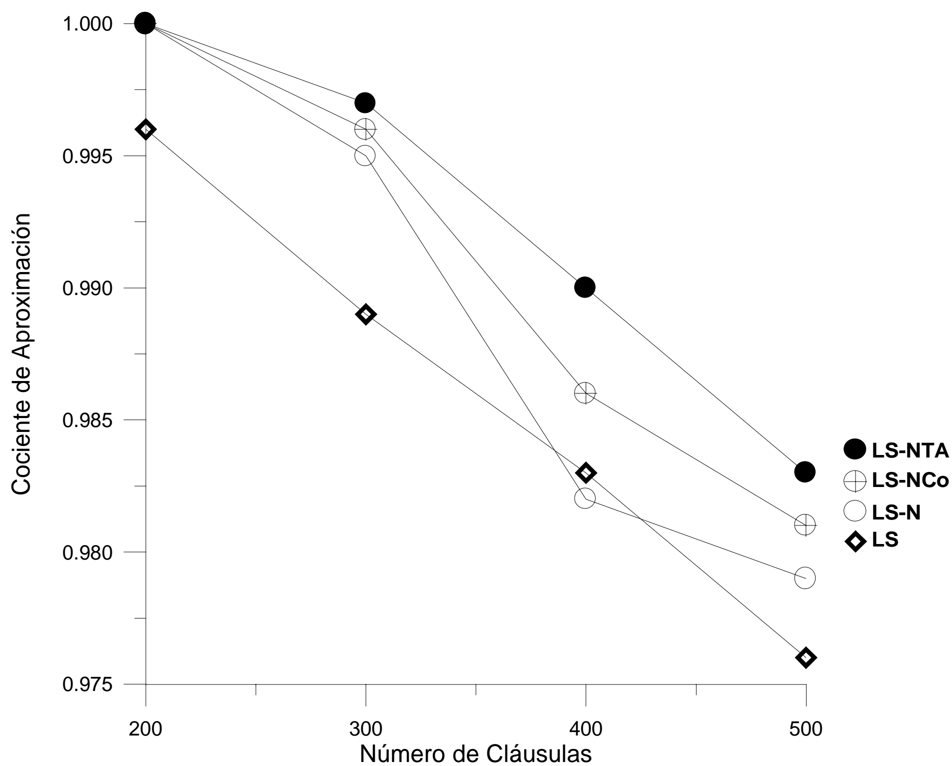

Figura 4.15 Cociente de aproximación de los algoritmos para 3-FNC con 100 variables.

			Promedio de Iteraciones			
Grupo	**Variables**	**Cláusulas**	**LS**	**LS-N**	**LS-NCo**	**LS-NTA**
1	25	50	937.23	149.68	196.23	158.35
2	25	75	874.62	124.29	227.6	148.25
3	25	100	624.74	126.41	231.16	141.31
4	25	125	543.74	134.3	200.47	149.77
5	50	100	1844.26	435.01	588.94	550.68
6	50	150	2325.8	559.07	580.79	652.63
7	50	200	2738.39	453.2	541.13	702.5
8	50	250	3024.51	511.57	696.77	860.16
9	75	150	2802.2	2367.18	1089.74	3334.97
10	75	225	3586.09	2909.27	1951.98	5180.05
11	75	300	4205.2	1436.74	1141.1	2297.7
12	75	375	4546.35	1394.08	1756.03	3192.86
13	100	200	3755.01	7196.39	1424.01	8380.84
14	100	300	4817.66	4434.78	3921.85	7612.75
15	100	400	5549.1	5988.149	5129.28	9826.341
16	100	500	6129.75	4502.08	4119.53	7689.091

Tabla 4.10 Promedio del número de cambios realizados por cada algoritmo sobre las instancias: INS3SAT.

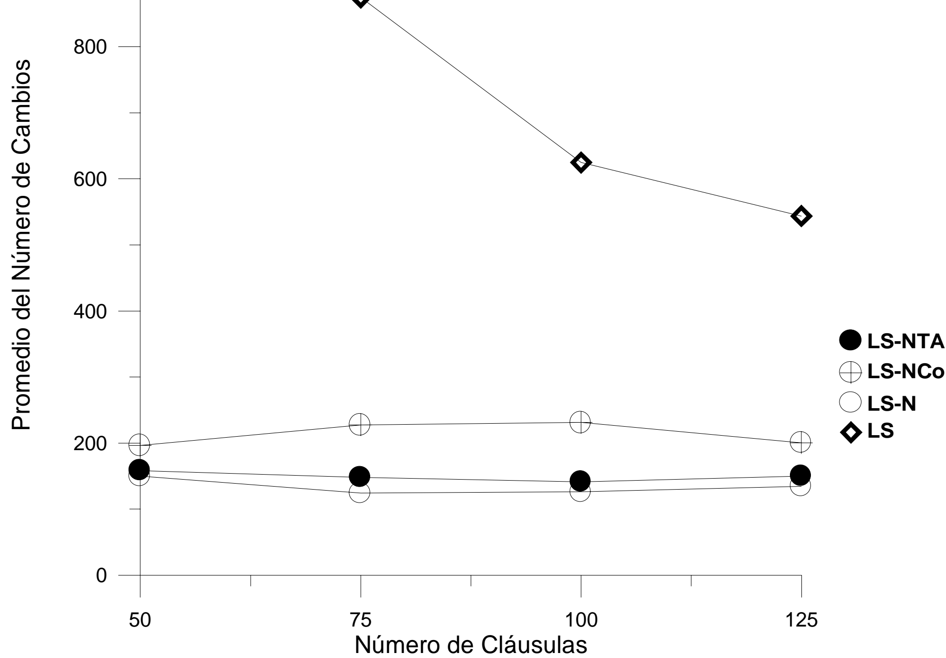

Figura 4.16 Promedio del número de cambios realizados por cada algoritmo para 3-FNC con 25 variables.

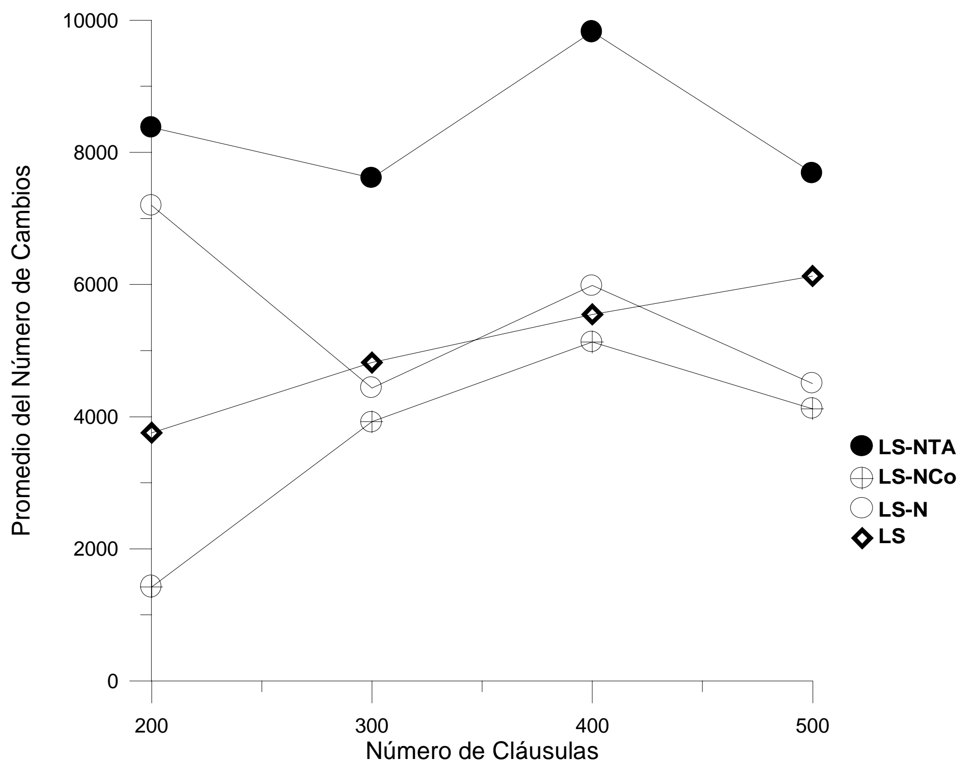

Figura 4.17 Promedio del número de cambios realizados por cada algoritmo para 3-FNC con 100 variables.

Las gráficas anteriores muestran todos los resultados obtenidos durante el análisis empírico. Se observa que la propuesta LS-NTA obtiene en todos los casos un mejor cociente de aproximación que los demás algoritmos.

Es de particular importancia mencionar el comportamiento del algoritmo LS-N (propuesto por Khanna [KhS96]), ya que se observa que el rendimiento decrece (en lo que respecta a su factor de aproximación) para instancias con muchas cláusulas insatisfechas.

En estos casos, el cociente de aproximación de LS-N está por debajo de los demás algoritmos, incluso de la búsqueda local sola (LS). En las gráficas se puede observar que este comportamiento puede ser corregido utilizando la nueva función objetivo que proponemos.

Como ya se ha comentado, el número de cláusulas insatisfechas (para cláusulas donde existen una gran cantidad de ellas) tiene un peso importante en el comportamiento de la búsqueda, y el hecho que la función objetivo propuesta por Khanna no considere este punto no le permite capturar dicha propiedad (al hacer el coeficiente C_0=0 en $f_{NOB}(z) = \sum_{i=0}^{k} C_i W(S_i)$).

El análisis de estos casos nos llevó a experimentar con diferentes valores para C_0, y nos permitió inferir que su signo debería ser opuesto al de los otros coeficientes, el análisis teórico nos confirmó tal conjetura.

El algoritmo resultante que denominamos LS-NC_0, mostró en la práctica un mejor comportamiento que LS y LS-N. Con el fin de acelerar el proceso de búsqueda, se le adicionó a LS-NC_0 dos heurísticas: una estrategia tabú y el uso de puntos antipodales como punto para re-iniciar la búsqueda después de arribar a óptimos locales, al algoritmo resultante se le llamó: LS-NTA.

El análisis sobre el comportamiento en la práctica de LS-NTA mostró que el uso de elementos antipodales para re-iniciar la búsqueda permite ampliar el dominio de ésta y en general, salta sobre caminos ya explorados. Si a esto añadimos una lista de los óptimos locales hallados para evitar ciclos (tabú), obtenemos una propuesta algorítmica que resultó ser robusta y, que de acuerdo al resultado del análisis empírico, mejora el comportamiento de los procedimientos basados en la búsqueda local.

El análisis experimental realizado, mostró que la aplicación de las estrategias: búsqueda tabú y uso de puntos antipodales, mejora en la práctica la calidad de las soluciones halladas. Así también, se corrobora los resultados obtenidos en la sección 4.1, en donde se determinó mediante el factor de garantía que la nueva propuesta LS-NC_0 tendría un mejor comportamiento que LS y LS-N.

Conclusiones

El objetivo de la demostración automática de teoremas (DAT) es diseñar algoritmos eficientes para demostrar la validez de una fórmula. Para el caso del cálculo proposicional, es bien sabido que la complejidad en tiempo de algoritmos dentro de la DAT para fórmulas booleanas generales es de orden exponencial y, consecuentemente, existen aún limitaciones para demostrar teoremas o probar satisfactibilidad en forma automática de manera eficiente.

El problema de *Máxima Satisfactibilidad* (MaxSAT) es un problema central en la DAT y en la teoría de complejidad computacional. Existe un gran interés en la búsqueda de algoritmos eficientes para la resolución de este problema. El hecho de que MaxSAT sea un problema representativo de su clase de complejidad (APX), implica que cualquier mejora que se encuentre a los algoritmos que resuelven MaxSAT, puede transformarse en mejoras a algoritmos que resuelvan los demás problemas de su clase de complejidad. Aun más, el hallar un algoritmo eficiente para este problema, repercute en la resolución de problemas abiertos trascendentales en las ciencias de la computación, tal como la pregunta abierta: ¿P = NP?

En esta investigación se analizaron diferentes propuestas algorítmicas basadas en la búsqueda local para solucionar el problema de MaxSAT, debido a que el objetivo principal del trabajo fue proponer un nuevo algoritmo *basado en búsqueda local* aplicado a la resolución del problema MaxSAT, así como determinar su factor de garantía, para mostrar que éste era competitivo con los algoritmos ya conocidos.

En este trabajo se construyeron dos algoritmos clásicos, de los cuales se ha determinado su factor de garantía. Para una fórmula booleana F (donde k es el número máximo de literales por cláusula), se tiene que la búsqueda local obvia (LS) obtiene un factor de garantía de $k/k+1$, mientras que la búsqueda local no obvia (LS-N) obtiene un factor de garantía de $(2^k-1)/2^k$.

Partiendo de estos dos algoritmos, proponemos uno nuevo al que denominamos: LS -NC_0, basado fundamentalmente en una modificación al coeficiente C_0 de la función objetivo propuesta por Khanna[KhS96] (sobre LS-N).

Se demostró analíticamente que nuestra propuesta obtiene un factor de garantía de 4/5 para Max2SAT, para ciertas clases de fórmulas, según se especificó en el capítulo II, mejorando el factor de ¾ del algoritmo propuesto por Khanna.

Posteriormente, con el fin de agilizar el proceso de búsqueda se adicionaron dos estrategias: una heurística tabú y el uso de elementos antipodales para re-iniciar la búsqueda después de arribar a óptimos locales. Esta nueva propuesta, que denominamos LS-NTA, mostró según el análisis empírico realizado y mostrado en el capítulo IV, que obtiene los mejores resultados de todos los algoritmos desarrollados.

Todos los algoritmos presentados preservan la complejidad polinomial en tiempo y mejoran el comportamiento general de la búsqueda local clásica. De estos algoritmos analizados, LS-NC$_0$ y LS-NTA son propuestas originales de este trabajo.

El éxito del algoritmo LS-NTA está determinado primero por su habilidad para moverse a través de la fase de 'meseta' de la búsqueda, reduciendo el tiempo de ésta fase y, segundo, por el uso de elementos antipodales del último óptimo local hallado como nuevo punto de re-inicio de la búsqueda, dando así amplitud al dominio de ésta.

Es de particular importancia mencionar el comportamiento del algoritmo LS-N (propuesto por Khanna[KhS96]), ya que se observa que el rendimiento decrece sobre cierto tipo de instancias donde hay un gran número de cláusulas insatisfechas. En estos casos, el cociente de aproximación de LS-N está por debajo de los demás algoritmos, incluso de la búsqueda local sola (LS). En las gráficas se puede observar que este comportamiento se corrigió utilizando la nueva función objetivo no obvia (aunque estos resultados son empíricos).

Parte de la explicación de este comportamiento es que LS-N no contempla en su función objetivo al conjunto de cláusulas S_0, es decir, el conjunto de cláusulas que no se satisfacen, así que cuando las instancias son grandes (tienen un mayor número de cláusulas), es evidente que se incrementa el número de cláusulas que no se satisfacen, en este caso, la función objetivo de LS-N empieza a carecer de elementos para discriminar correctamente entre dos asignaciones similares y decidir cual asignación es mejor en la búsqueda de la solución del problema. En nuestro caso, fue suficiente incluir una nueva función objetivo donde el coeficiente C_0=-1, sin embargo, aún queda pendiente determinar en detalle la razón de este comportamiento.

Se demostró que en la nueva función objetivo no obvia se obtiene un factor de garantía mejor que en la función objetivo no obvia propuesta por Khanna, esto para el tipo de instancias que cumplen que $W(S_1) \leq p \cdot W(S_0)$, $p \in \Re^+$. El análisis empírico realizado a LS-NC$_0$ verifica los resultados obtenidos a través del análisis teórico.

Las gráficas y las tablas presentadas en la segunda sección del capítulo IV muestran los resultados obtenidos del análisis empírico, se observa que la propuesta LS-NTA obtiene en todos los casos un mejor cociente de aproximación que los demás algoritmos. Esto debido principalmente al control de la trayectoria de búsqueda a través de los antipodales. Se puede observar además que el uso de elementos antipodales para re-iniciar la búsqueda permite ampliar el dominio de ésta y en general, saltar sobre caminos ya explorados. Si a esto añadimos una lista de los óptimos locales hallados para evitar caer en ciclos, obtenemos una propuesta algorítmica que resulta ser robusta y, que de acuerdo al resultado del análisis empírico mejora el comportamiento de los procedimientos basados en la búsqueda local.

Aunque, el análisis empírico resulta ser una herramienta valiosa para determinar el comportamiento de un algoritmo, el análisis teórico determina con exactitud éste, es por

eso, que en la sección 4.1 del capítulo IV se determinó de manera analítica el factor de garantía del algoritmo propuesto.

Personalmente, consideramos haber logrado desarrollar habilidades para la creación de nuevas propuestas algorítmicas. Es importante mencionar, que la línea de creación de nuevas propuestas algorítmicas resulta ser una área interesante, que puede ser frustrante si el pretenso investigador carece de habilidades tales como: creatividad y paciencia y de conocimientos de algorítmia, ya que la creación de nuevas propuestas algorítmicas no resulta una tarea sencilla, especialmente si se debe determinar teóricamente el factor de garantía.

Se concluye este trabajo de investigación, proponiendo un nuevo algoritmo para el tratamiento del problema MaxSAT, y determinando de manera empírica y analítica su factor de garantía.

Las aportaciones principales de este trabajo son las siguientes:

1. Propuesta de una nueva función objetivo:

$$f'_{NOB}(z) = \frac{3}{2}W(S_1) + 2W(S_2) - W(S_0)$$

2. Determinación de un nuevo factor de garantía para las propuestas algorítmicas. El factor de garantía determinado es de 4/5 para el caso Max2SAT, esto para el tipo de instancias que cumplen con: $W(S_1) \leq p \cdot W(S_0)$, $p \in \Re^+$.

3. Adición de heurísticas (tabú y antipodales) para acelerar el proceso de búsqueda.

4. Comprobación experimental de los resultados teóricos.

Una serie de trabajos a futuro se listan a continuación:

a) Determinar la razón del comportamiento extraño de LS-N sobre las fórmulas que poseen cantidades significativas de cláusulas insatisfechas, con lo que se podrá mejorar la calidad de las soluciones halladas. De aquí, se desprende una línea de investigación para futuros trabajos basados en nuestros resultados.

b) Experimentar con otras heurísticas para acelerar el proceso de búsqueda local.

c) Experimentar diferentes mecanismos para no bloquearse al hallar óptimos locales.

d) Proponer nuevas funciones objetivo no obvias y poder demostrar que obtienen mejores factores de garantía, o en su defecto, caracterizar el tipo de fórmulas sobre las que puede obtener mejores resultados.

Bibliografía

[AlL96] Altamirano L, Uso de la programación semidefinida para resolver Max2SAT, Tesis de maestría, Facultad de Ciencias Físico Matemáticas, BUAP, 1996.

[ChV92] Chvátal V., Reed B., Micks Gets Some (the Odds Are on His Side), Proc. 33rd. Annual Symp. On Foundations of Computer Science, 1992.

[CoS71] Cook S. A., The complexity of theorem proving procedures. In Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (1971), pp. 151-158.

[DeG95] De Ita G., Morales G., Algoritmos genéticos para el problema SAT, Memorias de la XII Reunión Nacional de Inteligencia Artificial, Noriega Edits., México, Sep. 1995.

[DeG96] De Ita G., Morales G., Heurísticas para mejorar la búsqueda local en el tratamiento del problema de máxima satisfactibilidad, Memorias del V congreso Iberoamericano de Inteligencia Artificial (IBERAMIA96), pág. 38-47, Edit. Limusa Noriega Editores, Nov. 1996.

[DiA96] Díaz A, Glover F., Ghaziri H., González J., Laguna M., Moscato P., Tseng F., Optimización Heurística y Redes Neuronales, Editorial Paraninfo, 105-142, 1996.

[FaR74] Fagin R., Generalized First-Order Spectra and Polynomial-time Recognizable Sets. In Richard Karp (ed.), Complexity of Computer Computations, AMS 1974.

[GaM79] Garey M. R. And Johnson D. S., Computers and Intractability: A Guide to the theory of NP-Completeness, W. H. Freeman and Co., New York, 1979.

[GaJ87] Galliere Jean H., Logic for Computer Science: Foundation of Automatic Theorem Proving, Wiley 1987.

[GaM76] Garey M., Johnson D., Stockmeyer L., Some Simplified NP-Complete graph problems. Theoret. Comput. Sci. 1, pp. 237-267, 1976.

[GeI93] Gent I.P., Walsh T., An empirical analysis of search in GSAT, Jour. of Artificial Intelligence Research 1, pp.47-59, 1993.

[GoM94] Goemans M. X., Williamson D. P., Improved Approximation Algorithms for Maximum Cut and Satisfiability Problems Using Semidefinite Programming, Proc. 26th Annual ACM Symp., on Theory of Computing, 422-431, 1994.

[GoM94a] Goemans M. X., Williamson D. P., ¾-Approximation Algorithms for the Maximum Satisfiability Problem, SIAM Journal of Discrete Mathematics, Vol. 7, No. 4, 1994.

[GuJ93] Gu J., Local Search for Satisfactibility (SAT) Problem, IEEE Transactions on Systems, Man, and Cybernetics, Vol. 23, No. 4, July / Autugst 1993.

[GuJ94] Gu J., *Global optimization for Satisfactibility (SAT) Problem*, IEEE Transaction on Knowledge and Data Engineering, Vol. 6, No.3, 361-381, June 1994.

[HaJ65] Hartmanis J., Stearns R., On the computational complexity of algorithms, Trns. Am. Math. Soc., 177 285-306, 1965.

[HaP90] Hansen P., B Jaumard, Algorithms for the Maximum SATisfiability Problem, Computing 44, 279-303, 1990.

[HoJ93] Hopcroft John E., Ullman Jeffrey D., Introducción a la Teoría de Autómatas, Lenguajes y Computación, CECSA 1993.

[ImN95] Immerman N., Descriptive Complexity: a Logician's Approach to Computation. In Notices of the American Mathematical Society, 42(10): 1127-1133, 1995.

[JoD74] Johnson D. S., Approximation Algorithms for Combinatorial Problems. Journal of Computer and System Sciences, vol. 9, pp. 256-278, 1974.

[KaA94] Kamath A., Motwani R., Khisna P., Spirakis P., Tail Bounds for Occupancy and the Satisfiability Threshold Conjecture, Proc. 35th Annual Symp. On Foundations of Computer Science, 1994.

[KaR72] Karp R. M., Reducibility among combinatorial problems. In Complexity of Computer Computations (R.E. Miller and J.W. Thatcher, Eds.), pp. 85-103, Plenum, New York, 1972.

[KhS95] Khanna Sanjeev, R. Motwani, M. Sudan and U. Vazirani, On Syntactic versus Computational Views of Approximability, TR95-023 ECCC Electronic Colloquium on Computational Complexity, Series 1995, pp. 13.

[KhS96] Khanna S., A structural view of approximation. Dissertation submitted to the department of computer science and the committee on graduate studies of stanford university, 1996.

[LeL73] Levin L. A., Universal sorting problems. Problems of Information Transmission, 9, pp. 265-266, 1973.

[ReC93] Reeves C. R., Modern Heuristic Techniques for Combinatorial Problems, John Wiley & Sans, Inc., 1993.

[StL87] Stockmeyer L. J., Classifying the Computational Complexity of Problems, The Journal of Symbolic Logic, March 1987.

[TuA36] Turing A. On computable numbers, with an application to the entscheidungs problem. Proc. London Math. Soc. Ser. 2, 42 and 43, 1936.

Lista de acronismos

SAT	Problema de satisfactibilidad.
MaxSAT	Versión de optimización del problema de satisfactibilidad.
Max-*k*-SAT	MaxSAT, donde se utilizan fórmulas que poseen *k* literales por cláusula.
DAT	Demostración automática de teoremas.
FNC	Fórmula en forma normal conjuntiva.
FND	Fórmula en forma normal disyuntiva.
k-FNC	FNC donde hay exactamente *k* literales por cláusula.
MTD	Máquina de Turing determinista.
PD	Problema de decisión.
MTND	Máquina de Turing no determinista.
RAM	Máquinas de acceso aleatorio.
DLOG	Clase de complejidad para la que existe una MTD que resuelve un PD usando espacio logarítmico.
NLOG	Clase de complejidad para la que existe una MTND que comprueba soluciones de PD usando espacio logarítmico.
PSPACE	Clase de complejidad para la que existe una MTD que resuelve un PD usando espacio polinomial.
NPSPACE	Clase de complejidad para la que existe una MTND que comprueba soluciones de PD usando espacio polinomial.
P	Es la clase de problemas que poseen algoritmos deterministas de resolución que tardan tiempo polinomial.
NP	Es la clase de problemas que tienen un algoritmo determinista de resolución que corre en tiempo exponencial, pero para los cuales, también existe un algoritmo no determinista que corre en tiempo polinomial.
NPO	Problemas de optimización derivados de problemas en NP.
APX	Clase de complejidad que posee problemas cuyos algoritmos de resolución tienen un factor de garantía acotado por una constante.
PTAS	Clase de problemas con esquemas de aproximación de tiempo polinomial. Clase de complejidad que posee problemas cuyos algoritmos de resolución tienen un factor de garantía acotado por $(1+\varepsilon)$.
f-APX	Clase de problemas en NPO que son aproximables dentro de un factor *f*.
poly-APX	Clase de problemas en NPO que tienen algoritmos de aproximación con un factor de garantía acotado polinomialmente, con respecto a la longitud de entrada.
log-APX	Clase de problemas en NPO que tienen algoritmos de aproximación con un factor de garantía acotado logarítmicamente, con respecto a la longitud de entrada.

APX-PB	Clase de problemas en NPO que son aproximables dentro de factores constantes y que tienen valores óptimos acotados polinomialmente.
LS	Búsqueda local clásica.
LS-N	Búsqueda local no obvia.
LS-NC_0	LS-N en donde se utiliza la nueva función objetivo propuesta.
TS	Búsqueda Tabú.
LS-NTA	Búsqueda Tabú no obvia con antipodales.
ALT2SAT	Conjunto de instancias 2-FNC generadas por Altamirano [AlL96].
INS2SAT	Conjunto de instancias 2-FNC generadas para el análisis empírico.
INS3SAT	Conjunto de instancias 3-FNC generadas para el análisis empírico.