La NP-Completitud del Problema SAT

Un Análisis Profundo

Aday Cuesta Correa
Stephan Brommer Gutiérrez
Sofía De Fuentes Rosella
Gerard Antony Caramazza Vilá
Óscar García González

Índice

Introducción

La teoría de la complejidad computacional es una rama fundamental de la informática teórica que estudia los recursos necesarios para resolver problemas computacionales. Entre sus principales objetivos está la clasificación de problemas según la dificultad de su resolución, proporcionando un marco formal para analizar qué problemas pueden resolverse de manera eficiente y cuáles no. En este contexto, una de las clases más relevantes es NP, que agrupa los problemas cuya solución puede verificarse en tiempo polinómico.

Dentro de esta clase, el problema de la satisfacción booleana, conocido como SAT, ocupa un lugar central. SAT consiste en determinar si existe una asignación de valores de verdad a un conjunto de variables booleanas que satisfaga una fórmula lógica dada. Formalmente, dado un conjunto de cláusulas en forma normal conjuntiva (CNF), SAT responde a la pregunta: ¿Es posible hacer verdadera la fórmula asignando valores apropiados a las variables?

Breve historia del problema SAT

El problema SAT se introdujo formalmente en los años 50 como parte del desarrollo de la lógica matemática y el análisis de circuitos booleanos. Sin embargo, su verdadera relevancia emergió en 1971, cuando Stephen Cook, en su famoso artículo *"The Complexity of Theorem-Proving Procedures"*, demostró que SAT es NP-completo. Este resultado, conocido como el *Teorema de Cook-Levin*, marcó el inicio del estudio sistemático de los problemas NP-completos, que son aquellos más difíciles dentro de la clase NP y cuya solución eficiente implicaría la solución eficiente de todos los problemas en NP. Desde entonces, SAT se ha convertido en un problema central en la informática teórica y en un punto de partida para abordar la complejidad computacional de otros problemas.

Ejemplo introductorio

Para entender mejor el problema SAT, consideremos un ejemplo sencillo. Supongamos la fórmula lógica:

$$(A \lor \neg B) \land (\neg A \lor B)$$

Esta fórmula está en forma normal conjuntiva (CNF), donde:

- ***A*** y ***B*** son variables booleanas que pueden tomar los valores verdadero (***True***) o falso (***False***).

- ∨ representa la operación lógica "**o**".
- ∧ representa la operación lógica "**y**".
- ¬ representa la negación de una variable.

El objetivo es determinar si existe una asignación de valores a las variables ***A*** y ***B*** que haga que la fórmula completa sea verdadera.

1. Evaluemos las posibles combinaciones de valores para *A* y *B*:

- Si ***A* = *True*** y ***B* = *True***: La fórmula se evalúa como:

$$(True \vee \neg True) \wedge (\neg True \vee True) = (True \wedge True) = True$$

- Si ***A* = *True*** y ***B* = *False***: La fórmula se evalúa como:

$$(True \vee \neg False) \wedge (\neg True \vee False) = (True \wedge False) = False$$

- Realizando lo mismo para las otras combinaciones (***A* = *False*, *B* = *True*** y ***A* = *False*, *B* = *False***), se obtiene que solo la combinación ***A* = *True***, ***B* = *True*** satisface la fórmula.

Este ejemplo ilustra cómo funciona el problema SAT en la práctica: se trata de encontrar una asignación que haga verdadera la fórmula. En problemas más complejos, con miles de variables y cláusulas, este proceso es computacionalmente costoso, lo que convierte a SAT en un desafío interesante y fundamental en la teoría de la complejidad.

Importancia y objetivos del informe

El problema SAT es de gran importancia en la teoría de la complejidad computacional, ya que fue el primer problema demostrado como NP-completo. Este informe tiene como objetivo realizar un análisis profundo sobre la NP-completitud de SAT. Exploraremos su definición formal, demostraremos que pertenece a la clase NP, y examinaremos la reducción de problemas NP-duros a SAT, destacando su papel como problema canónico en la teoría de la complejidad. Además, incluiremos ejemplos y aplicaciones prácticas para ilustrar su relevancia.

A través de este análisis, buscamos no solo profundizar en la naturaleza de SAT, sino también enfatizar su impacto en la informática y en la resolución de problemas fundamentales. Este trabajo se enmarca en el esfuerzo por entender los límites de lo computable y las implicaciones que estos tienen en la práctica y la teoría.

Capítulo 1. La Clase de Complejidad NP

La teoría de la complejidad computacional analiza la dificultad de resolver problemas mediante procesos lógicos o algorítmicos y la facilidad de verificar soluciones propuestas. Dentro de este campo, la clase **NP** (*Nondeterministic Polynomial time*) agrupa problemas en los que es posible comprobar si una solución es correcta de manera eficiente (en tiempo polinómico), aunque encontrar esa solución puede no ser tan sencillo.

Definición de la clase NP

Un problema pertenece a la clase NP si cumple con las siguientes condiciones:

- **Certificado polinómico**: Existe una solución candidata al problema que puede verificarse en tiempo polinómico respecto al tamaño de la entrada.
- **Verificador eficiente**: Un algoritmo determinista es capaz de verificar la validez de dicha solución en un tiempo razonable

Desde un punto de vista formal, un problema en NP puede describirse como un lenguaje ***L*** tal que existe una máquina de Turing ***M*** y un polinomio ***p(n)*** con la propiedad que:

$$x \in L \leftrightarrow \exists y, |y| \leq p(|x|) \text{ y } M(x, y) \text{ acepta en tiempo polinómico}$$

Aquí:

- ***x****:* es la entrada
- ***y****:* el certificado o solución candidata
- ***M****:* el algoritmo determinista que verifica si *y* es válido para *x.*

Un problema NP no necesariamente puede resolverse en tiempo polinómico, pero siempre es posible verificar una solución propuesta de manera eficiente.

Importancia de los problemas NP-Completos

Dentro de NP, existe un subconjunto de problemas llamado **NP-Completos**. Estos problemas son especialmente importantes porque representan los más difíciles dentro de NP. Si se encontrara un algoritmo eficiente para resolver un problema NP-Completo, todos los problemas en NP podrían resolverse de manera eficiente.

Un problema es NP-Completo si cumple dos condiciones:

- **El problema pertenece a la clase NP**
- **NP-dureza**: Todo problema NP puede transformarse en él mediante una reducción polinómica. Es decir, existe un procedimiento eficiente que convierte cualquier problema de NP en una instancia del problema NP-Completo.

El teorema de *Cook-Levin* demostró que SAT es NP-Completo, lo que impulsó el análisis sistemático de esta clase de problemas.

Resolver un problema NP-Completo de manera eficiente significaría que todos los problemas NP también podrían resolverse rápidamente. Esto responde a la famosa pregunta de: ***¿P = NP?***. En otras palabras, ¿todos los problemas cuya solución puede verificarse rápidamente también pueden resolverse rápidamente?

Si la respuesta es afirmativa, se revolucionaría la informática, permitiendo resolver problemas complejos en campos como la criptografía, la planificación automática y el diseño de redes de manera eficiente. Sin embargo, hasta ahora, esta cuestión permanece abierta, y muchos expertos creen que ***P ≠ NP***.

Ejemplos de problemas NP

- **El problema SAT**: Plantea la pregunta de si existe una forma de asignar valores de verdad a las variables de una fórmula lógica para que esta sea verdadera. Este problema tiene gran relevancia histórica, ya que fue el primer caso reconocido como NP-Completo, marcando el comienzo del estudio formal de esta clase de problemas.
- **El problema de la clique**: Dado un grafo, este problema pregunta si existe un subconjunto de ***k*** vértices donde todos están conectados entre sí. Por ejemplo, en un grupo de personas, este problema puede modelar si existe un subconjunto de ***k*** individuos que se conocen mutuamente.
- **El problema del camino hamiltoniano**: Busca un recorrido que pase por todos los vértices de un grafo exactamente una vez. Por ejemplo, se podría aplicar para planificar la ruta de un cartero que debe visitar varias casas sin repetir ninguna.

La clase NP contiene problemas fundamentales cuya verificación es eficiente, aunque su resolución pueda no serlo. Los NP-Completos, en particular, representan un reto clave en la teoría de la complejidad computacional. Estudiarlos no solo permite explorar los límites de lo computable, sino que también fomenta el desarrollo de nuevas estrategias algorítmicas para problemas complejos.

Capítulo 2. El Problema SAT

Formulación lógica del SAT

Como ya se ha mencionado anteriormente, en términos sencillos, SAT consiste en determinar si existe una asignación de valores de verdad (verdadero o falso) a las variables de una fórmula lógica que haga que dicha fórmula sea verdadera. La fórmula está expresada en forma normal conjuntiva (CNF, por sus siglas en inglés), que es una conjunción de cláusulas.

Formalmente, dado un conjunto de variables booleanas $x_1, x_2, \ldots, x_n$ un **literal** es una variable o su negación $(x_i \text{ o } \neg x_i)$. Una **cláusula** es una disyunción de literales, es decir, una expresión de la forma $(x_1 \vee \neg x_2 \vee x_3)$, donde el símbolo $\vee$ denota la operación lógica ***OR***. Finalmente, una fórmula en **CNF** es una conjunción de estas cláusulas, es decir, una expresión de la forma:

$$(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee x_3)$$

El problema SAT pregunta: ¿Es posible asignar valores de verdad a las variables $x_1, x_2, \ldots, x_n$ de tal manera que la fórmula completa se evalúe como verdadera?

Representación gráfica de una instancia del SAT

Una representación gráfica de una instancia del problema de satisfacibilidad booleana (SAT) puede mostrar cómo las variables y las cláusulas se interrelacionan. Este tipo de representación se llama **grafo bipartito**, ya que los nodos se dividen en dos conjuntos disjuntos: uno que representa las variables (y sus literales) y otro que representa las cláusulas. En un grafo bipartito, las aristas solo existen entre nodos de conjuntos diferentes.

Cada nodo en el grafo representa una variable o un literal, y cada arista conecta un literal con una cláusula que lo contiene.

Por ejemplo, la fórmula:

$$(x_1 \vee \neg x_2) \wedge (x_2 \vee x_3) \wedge (x_1 \vee \neg x_3)$$

Podría representarse gráficamente como un grafo bipartito con tres cláusulas, donde cada cláusula tiene conexiones con los nodos que representan los literales involucrados, mostrando las relaciones entre las variables.

Algoritmos ingenuos para resolver el SAT (y su ineficiencia)

Aunque el problema SAT es de gran interés en la teoría de la complejidad, su resolución efectiva es un desafío debido a su naturaleza NP-completa. Los **algoritmos ingenuos** para resolver SAT suelen basarse en **fuerza bruta**, probando todas las posibles asignaciones de valores de verdad a las variables. Para una fórmula con n variables, existen $\mathbf{2^n}$ posibles combinaciones de valores de verdad que deben verificarse.

Un algoritmo ingenuo consistiría en generar todas las asignaciones posibles de valores a las variables $\boldsymbol{x_1, x_2, \ldots, x_n}$, y luego evaluar si la fórmula es verdadera para alguna de estas combinaciones. Si al menos una asignación satisface todas las cláusulas, la respuesta sería "sí"; de lo contrario, "no".

Este enfoque tiene una complejidad de tiempo exponencial, específicamente $\boldsymbol{O(2^n)}$, lo que lo hace inviable para instancias grandes, ya que el número de combinaciones crece exponencialmente con el número de variables.

Por ejemplo, si tenemos una fórmula con solo 20 variables, el algoritmo ingenuo tendría que evaluar 2^{20}=1,048,576 asignaciones posibles, lo cual es ineficiente incluso para máquinas modernas.

Capítulo 3. Demostración de que SAT ∈ NP

Demostración de que SAT ∈ NP

1. **Certificado (solución candidata)**: El certificado es una asignación de valores de verdad para las variables de la fórmula ϕ. Por ejemplo, para una fórmula con variables $\boldsymbol{(x_1, x_2, \ldots, x_n)}$, el certificado es una secuencia $\boldsymbol{(b_1, b_2, \ldots, b_n)}$, donde $\boldsymbol{b_i} \in \boldsymbol{\{True, False\}}$.
2. **Verificador**: El verificador es un algoritmo que, dada la fórmula ϕ y un certificado (la asignación de valores), verifica si el certificado satisface ϕ. Este algoritmo debe ejecutarse en tiempo polinomial respecto al tamaño de ϕ.

Resolución Completa del Problema SAT

El problema completo SAT no solo verifica una asignación, sino que busca **encontrar una asignación válida o demostrar que no existe ninguna**. Esto requiere explorar todas las posibles combinaciones de asignaciones para las variables, un proceso que en el peor caso tiene complejidad $O(2^n)$, donde n es el número de variables.

El código en C++ implementa este enfoque, dividiendo el problema en dos partes:

1. **SAT::Solve**: Busca de forma recursiva una solución satisfactoria para la fórmula, probando todas las combinaciones de asignaciones.
2. **SAT::IsSatisfiable**: Verifica si una asignación específica satisface la fórmula.

Algoritmo Verificador (IsSatisfiable)

Este método verifica si una **asignación candidata** satisface la fórmula en CNF:

```
C/C++
Input: Fórmula booleana ϕ en CNF, Asignación candidata (b1, b2, ..., bn)
Output: true si la fórmula es satisfacible bajo esta asignación, false en caso
contrario

1. Para cada cláusula Ci en ϕ:
    a. Inicializar satisfecha = False
    b. Para cada literal Lij en Ci:
        i. Si Lij es una variable xj y bj = True, entonces satisfecha = True
        ii. Si Lij es la negación de una variable ¬xj y bj = False, entonces
satisfecha = True
    c. Si satisfecha sigue siendo False:
        Retornar false
2. Retornar true
```

Algoritmo SAT completo (BackTracking)

Este método explora todas las posibles asignaciones de las variables para encontrar una solución satisfactoria:

```
C/C++
Input: Fórmula booleana ϕ en CNF, Asignación parcial solution[], Índice var_index
Output: true si existe una solución, false en caso contrario

1. Si var_index == número total de variables:
    a. Retornar SAT::IsSatisfiable(solution)
2. Asignar solution[var_index] = False
3. Si SAT::Solve(solution, var_index + 1) retorna true:
    a. Retornar true
4. Asignar solution[var_index] = True
5. Si SAT::Solve(solution, var_index + 1) retorna true:
```

```
    a. Retornar true
6. Retornar false
```

Relación entre el Verificador y el Código Completo

- **Verificador (SAT::IsSatisfiable)**: Este algoritmo se utiliza para comprobar si una asignación específica es válida. Es equivalente al proceso descrito en el apartado de "Verificador" original del informe y tiene complejidad polinómica respecto al tamaño de la fórmula.

- **Resolución Completa (SAT::Solve)**: Este procedimiento utiliza el verificador para explorar todas las posibles asignaciones hasta encontrar una válida o demostrar que no existe ninguna. Este enfoque es el que aborda el problema completo SAT y tiene un coste exponencial en el número de variables.

Este modelo muestra que SAT pertenece a NP porque:

1. Tiene un certificado (una asignación de valores de verdad).
2. Existe un verificador que corre en tiempo polinómico para validar dicho certificado.

Complejidad Temporal

- El verificador evalúa la fórmula en tiempo proporcional al número de cláusulas y la longitud de estas, es decir, O(tamaño de ϕ).
- La búsqueda exhaustiva genera 2^n combinaciones y aplica el verificador en cada una, lo que lleva a una complejidad total de O($2^n \cdot$tamaño de ϕ).

Por lo tanto, el verificador corre en tiempo polinomial respecto al tamaño de ϕ.

Capítulo 4. SAT es NP-duro

Para demostrar que SAT es **NP-duro**, usaremos el concepto de **reducción polinomial**.

Concepto de Reducción Polinomial

Una reducción polinomial es un método formal para transformar un problema ***A*** en otro problema ***B*** de manera eficiente (en tiempo polinomial). Esto implica que, si podemos resolver ***B*** eficientemente, entonces también podemos resolver ***A*** eficientemente.

En el contexto de la teoría de la complejidad, si un problema ***A*** se reduce a un problema ***B***, y ***A*** es **NP-completo**, entonces ***B*** debe ser al menos tan difícil como ***A***, lo que lo clasifica como **NP-duro**.

Para ilustrar esto, demostraremos que SAT es NP-duro reduciendo un problema NP-completo conocido a SAT.

Problema de la Clique

El Problema de la Clique se define de la siguiente manera:

Entrada: Un grafo $G = (V, E)$ y un número entero k.
Pregunta: ¿Existe un subconjunto de k vértices en G tal que todos los vértices en el subconjunto estén conectados entre sí (es decir, que formen una **clique**)?

Una **clique** en un grafo es un conjunto de vértices $A \subseteq V$ tal que para cada par de vértices $u, v \in C$, la arista (u, v) pertenece al conjunto de aristas E.

Reducción de Clique a SAT

La reducción consiste en transformar una instancia del Problema de la Clique en una fórmula booleana en forma normal conjuntiva (CNF) que sea satisfacible si y sólo si existe una clique de tamaño ***k*** en el grafo.

Paso 1: Definición de variables

Usamos variables booleanas para representar si un vértice $v_i \in V$ ocupa una posición en la clique de tamaño ***k***.

Sea x_{ij}, donde:

- x_{ij} = ***True*** si el vértice v_i está en la posición ***j*** dentro de la clique.
- x_{ij} = ***False*** si el vértice v_i no está en la posición ***j***.

Paso 2: Construcción de restricciones en CNF

1. **Cada posición de la clique debe ser ocupada por un vértice del grafo:**

Para cada posición ***j*** de la clique ($1 \leq j \leq k$), al menos uno de los vértices debe ocupar esa posición. Esto se asegura mediante la cláusula:

$$\bigvee_{i=1}^{n} x_{i,j}$$

Donde $n = |V|$ es el número de vértices en el grafo.

2. **Un vértice no puede ocupar más de una posición en la clique:**
   Si un vértice v_i ocupa la posición j, no puede ocupar ninguna otra posición j'. Para $j \neq j'$:

$$\neg x_{ij} \vee \neg x_{ij'}$$

3. **Dos vértices no pueden ocupar la misma posición:**
   Si v_i y v_h son vértices diferentes, no pueden ambos ocupar la posición j:

$$\neg x_{ij} \vee \neg x_{hj}$$

4. **Conectividad entre vértices en la clique:**
   Si los vértices v_i y v_h están en la clique, entonces debe existir una arista entre ellos. Para cada par (i, h) donde $i \neq h$, si no existe la arista $(v_i, v_h) \in E$, se asegura que no pueden ambos estar en la clique simultáneamente. Esto se representa con la cláusula:

$$\neg x_{ij} \vee \neg x_{hj'}$$

Paso 3: Fórmula final

La fórmula SAT es la conjunción de todas las restricciones anteriores:

1. Restricción de ocupación de posiciones:

$$\bigwedge_{j=1}^{k} \bigvee_{i=1}^{n} x_{i,j}$$

2. Restricción de exclusividad de posiciones para cada vértice:

$$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{k} \bigwedge_{j'>j} (\neg x_{i,j} \vee \neg x_{i,j'})$$

3. Restricción de exclusividad de vértices en una posición:

$$\bigwedge_{j=1}^{k} \bigwedge_{i=1}^{n} \bigwedge_{h>i} (\neg x_{i,j} \vee \neg x_{h,j})$$

4. Restricción de conectividad:

$$\bigwedge_{(v_i,v_h)\notin E} \bigwedge_{j=1}^{k} \bigwedge_{j'=1}^{k} (\neg x_{i,j} \vee \neg x_{h,j'})$$

Ejemplo de Reducción

Supongamos que el grafo ***G*** tiene los siguientes vértices y aristas:

- ***V* = {*v1*, *v2*, *v3*, *v4*}**
- **E = {(*v1*, *v2*), (*v2*, *v3*), (*v3*, *v4*)}**

Queremos determinar si existe una clique de tamaño ***k*** = 2.

1. **Definición de variables:**

   Creamos las variables x_{ij} para ***i* = *1, 2, 3, 4*** (los vértices) y ***j* = *1, 2*** (las posiciones en la clique).

2. **Restricción de ocupación de posiciones:**

   Cada posición debe estar ocupada por al menos un vértice:

   $$(x_{11} \vee x_{21} \vee x_{31} \vee x_{41}) \wedge (x_{12} \vee x_{22} \vee x_{32} \vee x_{42})$$

3. **Restricción de exclusividad de posiciones para cada vértice:**

   Si v_1 ocupa la posición 1, no puede ocupar la posición 2:

   $$(\neg x_{11} \vee \neg x_{12})$$

4. **Restricción de conectividad:**

   Dado que $(v_1, v_3) \notin E$, no pueden ambos vértices estar en la clique:

$$(\neg x_{11} \vee \neg x_{32}) \ \wedge \ (\neg x_{12} \vee \neg x_{31})$$

5. **Combinación de restricciones:**

   Este paso es crucial porque el objetivo final de una reducción es transformar el problema original (encontrar una clique) en un problema SAT que pueda resolverse con un **solver SAT**.

$$(x_{11} \vee x_{21} \vee x_{31} \vee x_{41}) \wedge (x_{12} \vee x_{22} \vee x_{32} \vee x_{42}) \wedge (\neg x_{11} \vee \neg x_{12}) \wedge (\neg x_{21} \vee \neg x_{22}) \wedge (\neg x_{31} \vee \neg x_{32}) \wedge (\neg x_{41} \vee \neg x_{42}) \wedge (\neg x_{11} \vee \neg x_{32}) \ \wedge \ (\neg x_{12} \vee \neg x_{31})$$

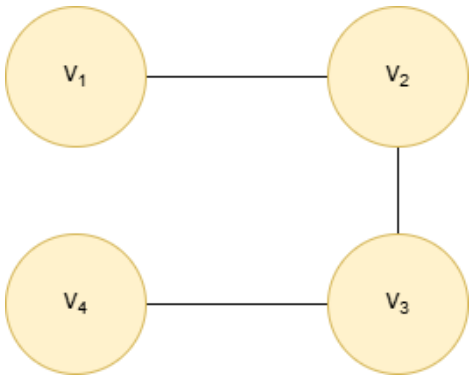

Figura 1. Ilustración del grafo de ejemplo de reducción de Clique a SAT

Justificación

- La reducción asegura que cualquier asignación satisfactoria de las variables corresponde a una clique válida en G.
- Es eficiente: el número de cláusulas generadas es polinomial en $|V|$ y k.
- Por lo tanto, si podemos resolver SAT en tiempo polinomial, también podemos resolver el Problema de la Clique en tiempo polinomial.

Esto demuestra que SAT es al menos tan difícil como el Problema de la Clique, estableciendo que SAT es **NP-duro**.

Capítulo 5. Conclusión

El problema SAT, al ser el primero demostrado como NP-completo mediante el Teorema de Cook-Levin, ocupa un lugar central en la teoría de la complejidad computacional. Este resultado implica que cualquier problema en la clase NP puede reducirse a SAT en tiempo polinomial, convirtiéndolo en un problema canónico para el estudio de la computación. Además, hemos demostrado cómo problemas NP-completos, como el Problema de la Clique, pueden transformarse eficientemente en instancias de SAT, evidenciando su equivalencia en términos de complejidad. SAT pertenece a NP porque sus soluciones pueden verificarse en tiempo polinomial y es NP-duro porque cualquier problema en NP puede transformarse en él.

La NP-completitud de SAT tiene implicaciones profundas. Primero, su centralidad en la teoría de la complejidad significa que resolver SAT eficientemente equivaldría a resolver eficientemente todos los problemas en NP, conectándolo directamente con la famosa pregunta de si P = NP. En la práctica, SAT es altamente relevante en áreas como la verificación de hardware y software, optimización, planificación e inteligencia artificial. Los avances en algoritmos, como los solvers SAT modernos, han permitido abordar problemas complejos en aplicaciones prácticas, a pesar de su dificultad teórica. Además, en el ámbito de la criptografía, la seguridad de muchos sistemas depende de la creencia de que P ≠ NP, lo que sería desafiado si SAT se resolviera de manera eficiente.

El estudio de SAT abre múltiples líneas de investigación futura. Una de ellas es el desarrollo de algoritmos más eficientes para resolver instancias específicas de SAT, incluyendo heurísticas y aproximaciones que reduzcan la complejidad en casos prácticos. También es fundamental explorar con mayor profundidad las relaciones entre SAT y otros problemas NP-completos, buscando nuevas formas de reducir problemas del mundo real a SAT. En el ámbito teórico, la pregunta sobre la relación P vs NP sigue siendo uno de los mayores desafíos, y SAT permanece en el núcleo de este debate. Por último, investigar variantes de SAT en clases de complejidad superior, como PSPACE o EXP, podría ampliar nuestra comprensión de los límites de lo computable en contextos más generales.

En resumen, el problema SAT no solo define los límites teóricos de la clase NP, sino que también impulsa el desarrollo de herramientas prácticas y conceptos fundamentales en la computación. Su estudio continúa siendo esencial tanto para la teoría de la complejidad como para la resolución de problemas computacionales en diversos campos.

Capítulo 6. Bibliografía

- Marin, S. R. (2023, septiembre 8). *Descubre el Teorema de Cook-Levin: La clave para resolver problemas complejos*. Teoremas de Cook-Levin
- Wikipedia contributors. (s/f). *Problema de satisfacibilidad booleana*. Wikipedia, The Free Encyclopedia

- P vs NP problems. (2024, enero 11). GeeksforGeeks
- Udofia, E. (2024, mayo 24). *¿Puede un problema SAT ser un problema NP completo?* EITCA Academy
- Wikipedia contributors. (s/f-b). *Problema del clique*. Wikipedia, The Free Encyclopedia