

# Evaluación probabilística de problemas #P-Completos mediante el método de Montecarlo y su implementación concurrente

Ramos López Luis Daniel<sup>1</sup>, Rodríguez Sánchez María Guadalupe<sup>2</sup>

<sup>1</sup>Departamento de Sistemas, UAM-Azcapotzalco, Ciudad de México, México

<sup>2</sup>Departamento de Ciencias Básicas, UAM-Azcapotzalco, Ciudad de México, México

E-mail: [ldr@azc.uam.mx](mailto:ldr@azc.uam.mx), [rsmg@azc.uam.mx](mailto:rsmg@azc.uam.mx).

**Resumen** — El presente artículo propone un método numérico probabilístico para estimar soluciones de problemas #P-Completos. Se discute una generalización del algoritmo, junto con una implementación concurrente en el lenguaje C++, que aprovecha bibliotecas estándar con funciones intrínsecas del procesador para generar instancias aleatorias de manera eficiente, así como el uso de hilos a nivel de usuario. Finalmente, se comparan los tiempos de ejecución y la precisión de las estimaciones obtenidas frente a algoritmos clásicos como búsqueda con retroceso o programación dinámica, y se evalúan la escalabilidad y las limitaciones del método.

**Palabras Clave** — probabilístico, método numérico, #P-Completo

**Abstract** — This paper proposes a probabilistic numerical method for estimating the number of solutions to #P-Complete problems. A generalization of the algorithm is discussed, along with a concurrent implementation in C++, which leverages standard libraries with intrinsic processor functions to efficiently generate random instances, as well as the use of user-level threads. Finally, the runtimes and accuracy of the estimates obtained are compared with classical algorithms like backtracking or dynamic programming, and the scalability and limitations of the method are evaluated.

**Keywords** — probabilistic, numerical method, #P-Complete

## I. INTRODUCCIÓN

La teoría de la complejidad computacional estudia la cantidad de recursos, como tiempo y espacio, necesarios para resolver un problema. En este contexto, un problema de clase #P aborda el conteo de soluciones válidas para un problema de decisión de la clase NP. Este tipo de problemas aparece en diversas disciplinas, como la teoría de gráficas, la física estadística o la combinatoria.

Dentro de esta clase, los problemas #P-Completos son aquellos cuya solución es particularmente costosa desde el punto de vista computacional. En consecuencia, a menudo es necesario recurrir a estrategias alternativas que permitan aproximarse a una solución utilizando recursos limitados.

Entre los problemas #P-Completos más representativos se encuentran el conteo de soluciones de problemas NP-Completos, como el número de asignaciones válidas del problema de satisfactibilidad booleana 3-SAT, el número de coloraciones válidas en el problema de coloración de gráficas, o el número de subconjuntos cuya suma es cero en el problema de la suma de subconjuntos, entre otros. Como la

resolución de estos problemas en su versión de decisión ya es considerada la más difícil de la clase NP, contar sus soluciones es, al menos, igual de complejo, lo que posiciona a los problemas #P-Completos como casos fundamentales dentro del estudio de la complejidad.

Este trabajo propone la generalización de un algoritmo basado en el método de Montecarlo para la estimación del número de soluciones en problemas de clase #P. El algoritmo fue implementado de manera concurrente en C++, utilizando tanto bibliotecas estándar para el uso de hilos a nivel usuario, como funciones intrínsecas del procesador para generar instancias aleatorias distribuidas uniformemente, requisito fundamental del método de Montecarlo. Se comparan los tiempos de ejecución y la precisión de las estimaciones obtenidas frente a algoritmos clásicos de búsqueda con retroceso o programación dinámica, y finalmente se analizan la escalabilidad y las limitaciones de ambos enfoques con respecto al tamaño de entrada.

## II. METODOLOGÍA

### A. Problemas NP-Completos

Un problema NP es, por definición, un problema de decisión (es decir, cuya respuesta solo puede ser “sí” o “no”) para el cual una solución candidata puede ser verificada en tiempo polinomial por una máquina de Turing determinista. Esta clase contiene problemas donde encontrar una solución puede ser difícil, pero verificar si una solución dada es válida es computacionalmente factible. La clase NP es probablemente una de las más estudiadas en la teoría de la complejidad computacional, dada su relevancia práctica y teórica.

Un problema NP-Completo es un problema de decisión que cumple dos condiciones fundamentales:

1. El problema está contenido en NP, y
2. Todo problema NP es reducible al problema dado mediante una transformación computable en tiempo polinomial

Una reducción polinómica de un problema  $L$  a un problema  $C$  es un algoritmo determinista que transforma una instancia  $l \in L$  en una instancia  $c \in C$ , de forma que  $l$  es una instancia afirmativa si y solo si  $c$  también lo es. Es decir, si  $C$  es NP-Completo, resolver  $C$  eficientemente implicaría poder resolver cualquier problema de NP de forma eficiente [1].

Hasta ahora, no se ha encontrado un algoritmo que resuelva un problema de la clase NP-Completo en tiempo polinomial, e incluso hay una discusión si en realidad los problemas de esta clase se pueden resolver en tiempo polinomial, abriendo la pregunta  $\#P = NP?$ . Entre los problemas más representativos de esta clase se encuentran el problema de la satisfactibilidad booleana (SAT), el problema del número cromático y el problema de la suma de subconjuntos, que más adelante se explicarán con más detalle.

### B. Problemas $\#P$ -Completos

Un problema de la clase  $\#P$  es un problema de conteo asociado a un problema de decisión de la clase NP. En lugar de preguntar si existe una solución válida (como ocurre en los problemas NP), un problema  $\#P$  pregunta cuántas soluciones válidas existen para una instancia dada.

Un problema se considera  $\#P$ -Completo si cumple dos condiciones análogas a las de los problemas NP-Complejos:

1. El problema pertenece a la clase  $\#P$ .
2. Todo problema en  $\#P$  es reducible al problema dado mediante una transformación computable en tiempo polinomial (es decir, el problema es  $\#P$ -Duro).

Muchos de los problemas NP-Complejos más conocidos tienen una versión de conteo que pertenece a la clase  $\#P$ -Completo. Por lo tanto, un problema  $\#P$ -Completo es al menos tan difícil como un problema NP-Completo [2].

Dado que los problemas  $\#P$ -Complejos son computacionalmente difíciles de resolver por medios deterministas clásicos como búsqueda con retroceso, se han desarrollado métodos probabilísticos que permiten estimar el número de soluciones con un margen de error controlado y en un tiempo mucho menor [3].

### C. Método de Montecarlo

Sea una muestra estadísticamente independiente de  $n$  observaciones  $x_1, \dots, x_n$  y  $f(x_i)$  el valor de una función que estamos evaluando en la muestra  $x_i$ , una estimación del valor esperado  $\mu$ , mediante el método de Montecarlo, es:

$$\mu \approx \frac{1}{n} \sum_{i=1}^N f(x_i)$$

El método de Montecarlo se basa en que las observaciones de la muestra sean completamente aleatorias y sin ningún sesgo [4].

Dada un nivel de confianza  $a\%$  asociado con una  $Z$ , un valor esperado  $\mu$  calculado a partir de una muestra de  $n$  observaciones y el error  $e_\sigma$  de la desviación estándar de las muestras, se puede estimar un intervalo de confianza del  $a\%$  dado el siguiente intervalo [5]:

$$[\mu - Z \cdot e_\sigma, \mu + Z \cdot e_\sigma]$$

Este intervalo proporciona una estimación del rango en el que se espera que se encuentre el valor verdadero del parámetro con una probabilidad del  $a\%$ .

### D. Estimación de problemas $\#P$ -Complejos utilizando el método de Montecarlo

Para aplicar el método de Montecarlo a un problema de la clase  $\#P$ , primero se define una función indicadora  $g(N_p, v)$  donde  $N_p$  es un problema de la clase NP y  $v$  es una configuración o asignación candidata para dicho problema. La función  $g$  se define como

```
g(Problema N_p, Configuración v)
  si v es una configuración válida para N_p
    regresa 1
  sino
    regresa 0
  fin
fin
```

Esta función permite evaluar si una configuración representa o no una solución válida para un problema de la clase NP.

La proporción  $\mu_v$  que representa la razón de configuraciones válidas con respecto al total de configuraciones posibles de un problema  $P_N$ , y  $K$  que es el número total de posibles configuraciones, está dada por:

$$\mu_v = \frac{1}{K} \sum_{i=1}^K g(P_N, v_i)$$

El problema  $\#P(N_p)$  asociado al problema  $N_p$  cuenta cuántas de las  $K$  configuraciones son válidas. Así, la proporción  $\mu_v$  cuenta qué fracción del total de configuraciones son válidas. El valor de  $\#P(N_p)$ , dada la proporción  $\mu_v$  y el total de configuraciones  $K$ , se define como:

$$\#P(N_p) = \mu_v \cdot K$$

En el contexto de los problemas  $\#P$ -Complejos, dada un problema  $\#P$  y su problema NP asociado  $N_p$ , una muestra estadísticamente independiente de  $n$  configuraciones  $v_1, \dots, v_n$  y la función  $g(P_N, v)$ , una estimación de  $\mu_v$  a partir del método de Montecarlo está dada por:

$$\mu_v \approx \frac{1}{n} \sum_{i=1}^n g(P_N, v_i)$$

El tamaño de la muestra  $n$  se propone calcularse con la fórmula del tamaño de muestra de una población infinita (puesto que, en general,  $K$  crece exponencialmente). Dado un error relativo deseado  $e_R$ , la proporción  $\mu_v$  de soluciones válidas del problema y un nivel de confianza  $Z$ , la fórmula se expresa como sigue:

$$n = \frac{Z^2 \cdot \mu_v \cdot (1 - \mu_v)}{e_R^2}$$

Como no conocemos el valor de  $\mu_v$ , primero suponemos que  $\mu_v = 0.5$ , para después recalcularlo iterativamente hasta que tenga un error estándar menor a  $e_R$  que está determinado por:

$$e_\mu = \sqrt{\frac{\mu_v \cdot (1 - \mu_v)}{n}}$$

En caso de que en alguna iteración se obtenga  $\mu = 0$ , el método de Montecarlo enfrenta una dificultad para producir una estimación significativa, ya que  $e_\mu = 0$  llevaría a concluir, en muchas ocasiones de manera apresurada, que el número total de soluciones es cero. Para evitar esta situación, se implementa un ajuste dinámico del tamaño de muestra, incrementando  $n$  mediante un factor de crecimiento  $f$ . En este caso, se recalcula  $n$  como:

$$n = f \cdot \frac{Z^2 \cdot \mu_\gamma \cdot (1 - \mu_\gamma)}{e_R^2}$$

El factor  $f$  se inicializa con el valor  $f = 1$  y se duplica cada vez que se obtenga  $\mu_v = 0$ , es decir:

$$f \leftarrow 2 \cdot f$$

Por otra parte, cuando el error estándar  $e_\mu$  es mayor que el error relativo deseado  $e_R$ , lo que puede ocurrir en espacios de soluciones muy grandes con tamaños de muestra pequeños, se requiere un incremento proporcional del número de muestras. Para ello, se ajusta  $f$  de acuerdo con la siguiente expresión:

$$f \leftarrow f \cdot \left(1 + \frac{e_\mu - e_R}{e_R}\right)$$

Este mecanismo asegura que el crecimiento del tamaño de la muestra sea más agresivo cuando el error es significativamente mayor al permitido, y más conservador cuando ya se está cerca del objetivo, evitando un aumento innecesario del costo computacional.

Finalmente, para evitar que el número de muestras crezca sin control, se impone un límite superior basado en el tamaño del espacio total de configuraciones. Si en alguna iteración se alcanza  $n > K$  y además se cumple  $\mu_v = 0$ , se concluye que es razonable asumir que el número de soluciones válidas es cero con alta probabilidad.

Por lo tanto, un algoritmo para calcular una estimación de un problema  $\#P$ , con un error relativo  $e_R$  y un nivel de confianza  $Z$  utilizando el método de Montecarlo será la siguiente:

```

Montecarlo(Problema P, Error rel. e_R, Nivel de confianza Z)
    flotante f ← 1, e_mu ← 1, mu_v ← 0.5
    hacer
        entero n ← (f · Z^2 · mu_v · (1 - mu_v)) / e_R^2
        entero val ← 0
        desde (i ← 0) hasta n
        hacer
            configuración v ← configuraciónAleatoria(P)
            val ← val + g(P, v)
            i ← i + 1
        flotante mu_act ← val / n
        si (mu_act = 0)
        hacer
            si (mu_act = 0)
            hacer
                sí (n > K) entonces
                    regresa 0
                fin sino
                    f ← 2 · f
                    continuar
                fin
            fin
            mu_v ← mu_act
            e_mu ← √(mu_v · (1 - mu_v)) / n
            si (e_mu > e_R)
            hacer
                f ← f · ((1 + (e_mu - e_R)) / e_R)
            fin
            mientras (e_mu > e_R)
            regresa mu_v · K
        fin
    fin

```

#### E. Intervalos de confianza de un problema $\#P(N_P)$

Debido a que el espacio de nuestra función  $g(N_P, v)$  es discreto, en vez de ser continuo como en el método de Montecarlo clásico, nuestra proporción  $\mu$  queda sesgada por el espacio que hay entre dos posibles valores. Para contrarrestar esta situación, nosotros en vez de construir el intervalo de confianza como se comúnmente se calcula, a partir de la proporción que nos generó nuestra función **Montecarlo**, se evaluará varias veces la función hasta que la media  $\bar{P}$  de los resultados de cada evaluación se estabilice dado un error absoluto y relativo menor a  $e_R$ .

Dada una media  $\bar{P}$  de simulaciones de Montecarlo y un error relativo máximo garantizado  $e_G$ , el intervalo de confianza utilizado en este proyecto se define como:

$$[\bar{P} - e_G \cdot \bar{P}, \bar{P} + e_G \cdot \bar{P}]$$

El valor de  $e_G = 0.05$  se calculó de manera empírica para el problema del polinomio cromático en un proyecto terminal de licenciatura [6], el uso de  $e_G$  como base del cálculo de los intervalos, permite que el intervalo se ajuste dinámicamente a la magnitud de las soluciones y al tamaño del problema, sin perder representatividad ni precisión.

El algoritmo para calcular los intervalos de confianza de la evaluación de un problema  $\#P$ , con un error estándar un error relativo  $e_R$  y un nivel de confianza  $Z$ , utilizando el método de Montecarlo es el siguiente:

```

Intervalos(Problema P, Error rel.  $e_R$ , Niv. Conf. Z)
vector<Flotante> v
flotante media ← Montecarlo( $P, e_R, Z$ ), cabs, crel
hacer
    flotante mant ← media(v)
    flotante r ← Montecarlo( $P, e_R, Z$ )
    insertar r en v
    media ← media(v)
    cabs ← |media - mant|
    crel ← cabs / mant
fin
mientras(cabs >  $e_R$  & crel >  $e_R$ )
    regresa [media - media *  $e_G$ , media + media *  $e_G$ ]
fin

```

### III. RESULTADOS

#### A. Implementación

La implementación del método de Montecarlo se realizó en C++ y se diseñó con un enfoque modular y concurrente utilizando únicamente la biblioteca estándar de C++. Para la generación aleatoria de configuraciones se utilizó una combinación de `std::uniform_int_distribution` y la instrucción `rdrand64_step`, de la biblioteca `immintrin.h`, que permite obtener números aleatorios de 64 bits directamente del procesador, garantizando aleatoriedad de alta calidad incluso en contextos paralelos. Cada configuración generada es verificada mediante una función indicadora que evalúa si cumple las condiciones del problema de decisión asociado.

Debido al crecimiento exponencial de los espacios de solución en problemas #P, se recurrió a tipos de datos extendidos como `_int128_t` [7] y `std::float128_t` [8] para evitar pérdidas de precisión y desbordamientos. Se implementaron funciones auxiliares específicas para operar con estos tipos.

La ejecución del método fue paralelizada mediante hilos estándar de C++, distribuyendo bloques de simulaciones entre todos los núcleos disponibles. El tamaño de muestra se ajusta dinámicamente según el error relativo deseado, y los resultados se utilizan para calcular intervalos de confianza a partir de estadísticas acumuladas durante la simulación.

Para este artículo, el proyecto se ejecutó en una computadora con un procesador Intel® Core™ i7-8700 que tiene un procesador de seis núcleos con una frecuencia máxima de 4.6Ghz, doce hilos lógicos y una memoria caché de 12MB. Para compilar el proyecto, se utilizó GNU 14.2.0 con la versión de C++23 [7] y se utilizaron las siguientes banderas: `-std=c++23`, `-march=native` y `-fquadmath`.

Los datos estadísticos que se utilizaron fue un error relativo  $e_R = 0.01$ , un intervalo de confianza del 99% con un  $Z = 2.745$  y un error garantizado  $e_G = 0.05$ .

#### B. Resultados con el problema de #3-SAT

El problema 3-SAT consiste en determinar si existe una asignación de valores de verdad que satisfaga una fórmula booleana en forma normal conjuntiva (CNF), donde cada cláusula contiene exactamente tres literales. Su versión de

conteo, que busca calcular el número total de asignaciones que satisfacen la fórmula, es un problema #P-Completo

Para evaluar la eficiencia y precisión del método, se comparó con un algoritmo exacto basado en búsqueda con retroceso. Se resolvieron 18 instancias diferentes generadas aleatoriamente, con un número de cláusulas  $m$  desde 1 hasta 18, y hasta  $1.5 \times m$  variables

En la Tabla I se resumen tiempos de ejecución y errores relativos para instancias seleccionadas, omitiendo el número exacto de soluciones para mantener la tabla compacta. Cabe destacar que, en la instancia más grande el problema tiene más de 26 mil millones de soluciones.

Tabla I.  
Resultados comparativos en #3-SAT

Cláusulas	Variables	Tiempo Backtracking (s)	Tiempo Montecarlo (s)	Error relativo (%)
1	3	0.000	0.112	0.10
5	11	0.001	0.321	0.06
9	19	0.046	0.568	0.18
14	27	7.257	0.796	0.54
18	38	8758.30	1.662	1.68

Los resultados muestran que el método de Montecarlo logra mantener errores relativos bajos, incluso en instancias grandes, con una reducción drástica en el tiempo de ejecución frente al método de búsqueda con retroceso. Esto confirma su utilidad para problemas #P-Completos donde el conteo exacto se vuelve intratable.

#### C. Resultados con el problema del Polinomio Cromático

Una coloración propia de una gráfica es una asignación de colores a sus vértices de forma que no existan vértices adyacentes con el mismo color. Contar el número total de coloraciones propias con  $k$  colores es un problema #P-completo, y está relacionado directamente con el polinomio cromático  $P(G, k)$ . Este polinomio se puede calcular mediante el método de borrado-contracción, cuya complejidad es  $P(G) = O(\phi^{|V|+|E|})$ , donde  $\phi = \frac{1+\sqrt{5}}{2}$  [1].

En el proyecto final anteriormente mencionado [6], se evaluaron polinomios cromáticos de diversas gráficas usando el método de Montecarlo y se compararon con el algoritmo de borrado-contracción. Se consideraron todas las gráficas no isomorfas hasta 8 vértices y muestras aleatorias para gráficas con 9, 10, 11 y 12 vértices. En hasta un 99.99 %, los valores exactos se encontraron dentro de los intervalos de confianza estimados.

Como evaluación destacada, se calculó el valor aproximado para un camino de 30 vértices y 13 colores, obteniéndose aproximadamente  $2.57 \times 10^{35}$ , con un error relativo cercano al 0.8 %.

En la Tabla II se presenta un resumen con instancias representativas:

Tabla II.

Resultados comparativos en la evaluación del polinomio cromático

Vértices	Gráficas evaluadas	Tiempo Montecarlo (s)	Tiempo Borrado-contracción (s)
5	34	0.340	0.0002
7	1044	0.432	0.0067
8	12,346	0.474	0.035
10	100	0.596	1.388
11	50	0.652	10.561
12	10	0.688	93.734

Se observa que el método exacto se vuelve rápidamente intractable conforme crece la gráfica, mientras que el método de Montecarlo mantiene tiempos estables y escalables, con errores bajos y controlados.

#### D. Resultados con el problema de #Subset sum y limitaciones del método

El problema de #Subset sum consiste en contar el número de subconjuntos de un arreglo de enteros que suman exactamente cero. Su versión de conteo pertenece a la clase #P-completo. A diferencia de otros problemas #P, existe un algoritmo exacto basado en programación dinámica (DP) con complejidad pseudopolinomial [1].

En este estudio se generaron instancias aleatorias con tamaños de 1 a 50 elementos, y valores enteros en el rango  $[-n^2, n^2]$  y se comparó el método del Montecarlo con el algoritmo de programación dinámica. Con este rango, se buscó realizar una comparación justa y crítica con el método, evitando diseñar instancias "trampa" que desacreditaran injustamente el enfoque exacto.

En la Tabla III se resumen casos seleccionados que ilustran este comportamiento, abarcando tamaños pequeños, medianos y grandes.

Tabla III.

Resultados comparativos en #Subset sum.

Tamaño	Tiempo DP (s)	Tiempo Montecarlo (s)	Error relativo (%)
10	0.001	0.722	6.10
15	0.009	1.512	3.55
20	0.041	1.435	2.66
30	0.283	5.988	16.00
40	1.030	8.558	37.60
50	3.300	48.806	61.09

Se observa que la programación dinámica resulta muy rápida en estas instancias, aprovechando la estructura repetitiva de las sumas parciales. Por otro lado, el método de Montecarlo mantiene tiempos bajos, pero el error relativo puede crecer de forma significativa debido a la alta proporción de subconjuntos solución: en esos casos, el criterio probabilístico puede cumplirse rápidamente y producir un error porcentual elevado.

## IV. DISCUSIÓN

Los resultados obtenidos muestran que el método de Montecarlo ofrece una herramienta flexible y general para estimar soluciones en problemas #P-completos, especialmente en instancias donde los métodos exactos resultan intractables. Su comportamiento depende de la estructura y la distribución de soluciones en cada problema, lo que puede influir en el error relativo y en la velocidad de convergencia.

Aun cuando en casos como #Subset sum existen algoritmos exactos altamente eficientes, Montecarlo sigue destacando por su sencillez, su capacidad de paralelización y su aplicabilidad universal sin requerir modificaciones específicas al problema.

## V. CONCLUSIONES

Se presentó un análisis generalizado del método de Montecarlo aplicado a problemas #P-completos, destacando tanto su versatilidad como sus limitaciones. Los resultados confirman que, además de ser una alternativa viable para problemas sin soluciones exactas eficientes, el método es capaz de adaptarse a distintas estructuras y tamaños de instancia. Esto reafirma su valor como herramienta complementaria en el estudio y aproximación de problemas complejos.

## APÉNDICE

Los programas utilizados, los resultados completos y las instancias de prueba se encuentran disponibles en el repositorio general de GitHub para este artículo: <https://github.com/danielramos/Aproximacion-sharp-P-Montecarlo>; y en el repositorio del proyecto terminal original: <https://github.com/danielramos/PIIC-2024O>.

## REFERENCIAS

- [1] H. S. Wilf, Algorithms and Complexity, Pennsylvania: University of Pennsylvania, 1994.
- [2] HSE University, Counting Problems (#P), Moscú: HSE University, 2019.
- [3] L. G. Valiant, «"The Complexity of Enumeration and Reliability Problems"», *SIAM Journal on Computing*, vol. 8, nº 3, p. 410–421, 1979.
- [4] P. Saavedra y V. H. Ibarra, El método Monte-Carlo y su aplicación a las finanzas, México: UAM Iztapalapa, 2008.
- [5] R. E. Walpole, S. L. Myers y Y. Keying, Probabilidad y estadística para ingeniería y ciencias, México: Pearson Education, 2012.
- [6] L. D. Ramos y M. G. Rodríguez, Intervalos de confianza de la evaluación del polinomio de una gráfica utilizando el método de Montecarlo, Ciudad de México: Universidad Autónoma Metropolitana Unidad Azcapotzalco, 2025.
- [7] Richard M. Stallman & GCC Developer Community, Using the GNU Compiler Collection (For GCC version 14.2.0), Massachusetts: GNU Press, 2024.
- [8] C++Reference, «Fixed width floating-point types (since C++23)», 3 Marzo 2024. [En línea]. Available: <https://en.cppreference.com/w/cpp/types/floating-point.html> [Último acceso: 2025 06 2025].