CIENTIFICA
TechNoLogica
ISSN:3119-7906, Vol.1Núm.2(2026): Abril-Junio
https://doi.org/10.17268/tech1.2.2026.04

Fuerza Bruta: Análisis, Aplicaciones y Limitaciones de una Estrategia Algorítmica Fundamental

Brute Force: Analysis, Applications and Limitations of a Fundamental Algorithmic Strategy

Escobar Mendoza Eduardo David[1], Zapata Abanto Carmen Alicia del Rosario[1], Chavez Urcia Daniel Andree[1], Muñoz Gonzalez Gerardo Manuel[1], Vázquez Cabrera Ángel Igor[1], Toro Gallo Jorge Luis[1], Ramos Torres Danny Jhoel[1]

[1] Escuela de Ingeniería Informática, Universidad Nacional de Trujillo, La Libertad, Perú

*Autor correspondiente: eescobar@unitru.edu.pe (Eduardo. E)

RESUMEN

La fuerza bruta es una de las formas más simples y directas de diseñar y analizar algoritmos. En este artículo se revisan sus bases teóricas, su relación con la complejidad computacional y su uso en problemas clásicos de la informática, como la búsqueda lineal, la generación de permutaciones, el problema del viajero (TSP) y la búsqueda de patrones en textos. Mediante experimentos controlados desarrollados en Java, se evaluaron los tiempos de ejecución con entradas de tamaño creciente. Los resultados muestran cómo, en ciertos casos, el tiempo de procesamiento aumenta de manera muy rápida debido a la explosión combinatoria propia de algoritmos con complejidades como O(n!) y $O(2^n)$. Además, estos resultados se comparan con métodos más eficientes, como el algoritmo de Held-Karp para el TSP y los algoritmos KMP y Boyer-Moore para la búsqueda de patrones. Esta comparación permite evidenciar las principales limitaciones prácticas de la fuerza bruta cuando se trabaja con problemas de gran tamaño. En conclusión, aunque la fuerza bruta puede garantizar una solución óptima en problemas finitos, su capacidad de escalar es muy limitada. Por ello, resulta más útil en casos pequeños, en fases iniciales de prototipado o como punto de referencia para comprobar la validez de algoritmos más avanzados. Su estudio es fundamental porque ayuda a comprender por qué son necesarias estrategias más sofisticadas, como la programación dinámica, los algoritmos voraces y las metaheurísticas.

Palabras clave: Fuerza bruta, complejidad algorítmica, algoritmos de búsqueda, optimización combinatoria, diseño de algoritmos.

ABSTRACT

Brute force is one of the simplest and most direct approaches to designing and analyzing algorithms. This article reviews its theoretical foundations, its relationship with computational complexity, and its use in classic computer science problems such as linear search, permutation generation, the traveling salesman problem —TSP— and pattern matching in texts.

Through controlled experiments developed in Java, execution times were evaluated using increasingly large input sizes. The results show how, in certain cases, processing time grows very quickly due to the combinatorial explosion typical of algorithms with complexities such as O(n!) and $O(2^n)$.

In addition, these results are compared with more efficient methods, such as the Held-Karp algorithm for the TSP and the KMP and Boyer-Moore algorithms for pattern matching. This comparison highlights the main practical limitations of brute force when working with large-scale problems.
In conclusion, although brute force can guarantee an optimal solution for finite problems, its scalability is very limited. Therefore, it is most useful for small cases, early prototyping stages, or as a reference point to verify the validity of more advanced algorithms. Studying brute force is essential because it helps explain why more sophisticated strategies, such as dynamic programming, greedy algorithms, and metaheuristics, are necessary.
Keywords: *Brute force, algorithmic complexity, search algorithms, combinatorial optimization, algorithm design.*

I. INTRODUCCIÓN

El diseño de algoritmos es una parte muy clave en la ingeniería y la ciencia de computar. Desde los primeros pasos dados por Turing, quien mostró los límites de lo que se puede o no hacer con una máquina, hasta libros más nuevos de Cormen y Levitin, el aprender cómo resolver problemas con máquinas ha sido tarea central en este campo.

De todas las formas que hay, la fuerza bruta es especial porque es de las más simples y claras. Este tipo de método se basa en mirar todas las formas de arreglar un problema, una por una, sin buscar caminos cortos, trucos o usar datos del problema mismo que ayuden a reducir el trabajo

En pocas palabras, con fuerza bruta se debe poner en orden y probar todas las posibles respuestas, mirando si cumplen las reglas que pide el problema. Por lo simple que es, puede ser útil al inicio cuando uno aún no tiene otra forma de solucionar el tema, o cuando se necesita ver si un método más complejo da el resultado correcto. Pero su gran contra es que no sirve bien si el problema crece: las cosas a probar pueden ser muchas y subir muy rápido, lo que hace que tarde mucho tiempo.

En los años 50 y 60, cuando la idea de los algoritmos recién empezaba, la fuerza bruta era casi siempre la única forma de resolver algunos problemas. Justo por lo difícil y largo que era esto, se buscaron métodos que gastaran menos tiempo, como la programación dinámica de Bellman para el problema del viajero, y otros caminos que estudió Karp, quien ayudó a mejorar la teoría de los problemas NPcompletos, junto con Garey y Johnson.

En la actualidad, estudiar la fuerza bruta sigue siendo importante porque ayuda a tener una base para comparar con otros métodos más rápidos. Además, si hay problemas que no se pueden resolver de forma fácil o rápida, la fuerza bruta permite hallar la mejor solución en casos pequeños. Por ejemplo, en el campo de la criptografía, la fuerza bruta se usa para ver qué tan seguro es un sistema de cifrado: si un ataque de este tipo toma mucho tiempo y recursos, se dice que es seguro.

Este texto busca explicar las ideas principales de la fuerza bruta, ver cuán difícil es para una máquina, mostrar algunos usos en problemas comunes y compararlo con otros métodos mejores. Así, se podrá ver en qué momentos conviene usar fuerza bruta y cuándo no sirve tanto

II. MATERIALES Y METODOS

A. Revisión Bibliográfica

El trabajo se hizo leyendo libros y textos sobre algoritmos y el estudio de la complejidad. Se usaron los libros de Cormen y otros, Levitin, Sedgewick junto con Wayne, y Skiena, además de Kleinberg y Tardos. Para ver mejor la parte de matemáticas, se leyó Papadimitriou y también los tomos de Knuth, que son libros muy usados para ver cómo funcionan los métodos y algoritmos por dentro. Para aprender sobre encontrar patrones, se revisaron los trabajos de Knuth, Morris, Pratt y también de Boyer y Moore.

B. Problemas Seleccionados

De este modo, se buscó tener una base sólida para entender y explicar la fuerza bruta desde distintos puntos de vista.

- Búsqueda lineal: consiste en recorrer de manera secuencial un arreglo de n elementos hasta encontrar un valor específico. En el peor de los casos, su complejidad es O(n).
- Generación de permutaciones: se basa en listar todos los posibles ordenamientos de un conjunto de n elementos. Su complejidad es O(n!).
- Problema del viajero (TSP): implica evaluar todas las rutas hamiltonianas posibles en un grafo completo formado por n ciudades. Su complejidad es O(n!).
- Búsqueda de patrones ingenua: compara carácter por carácter un patrón de longitud m dentro de un texto de longitud n. Su complejidad es $O(n \cdot m)$.
- Problema de la mochila 0/1: consiste en revisar todos los subconjuntos posibles de n objetos para encontrar la combinación que maximice el valor total sin superar una capacidad W. Su complejidad es $O(2^n)$.

C. Implementación y Medición

Los algoritmos fueron desarrollados en Java utilizando el entorno de desarrollo NetBeans. Para medir los tiempos de ejecución se empleó el método System.nanoTime(). Con el fin de obtener resultados más estables y reducir posibles variaciones, se calculó el promedio de 10 ejecuciones por cada instancia evaluada.

Las pruebas se realizaron con tamaños de entrada $n \in \{5, 10, 12, 15, 18, 20\}$, en una computadora equipada con un procesador Intel Core i5 de décima generación y 8 GB de memoria RAM.

III. RESULTADOS

A. Tiempos de Ejecución

Los resultados obtenidos en los experimentos mostraron una diferencia clara entre los algoritmos con complejidad polinomial y aquellos cuyo crecimiento es exponencial o factorial. En la Tabla I se presentan los tiempos de ejecución registrados en el peor caso para cada uno de los problemas evaluados.

TABLA I

TIEMPO DE EJECUCIÓN POR PROBLEMA

Problema	Complejidad	n=10	n=15	n=20
Búsqueda lineal	O(n)	<1	<1	<1
String matching	O(n·m)	<1	<1	2
Mochila 0/1	$O(2^n)$	1	32	1 048
Permutaciones	O(n!)	3	1 307	$>10^6$
TSP fuerza bruta	O(n!)	4	1 520	$>10^6$

Para el caso de n = 20, la cantidad de permutaciones posibles llega aproximadamente a 2.43×10^{18}, lo que vuelve impracticable una exploración completa, incluso utilizando equipos modernos. En cambio, la búsqueda lineal y la búsqueda ingenua de patrones mantuvieron tiempos inferiores a los 2 ms en todas las pruebas realizadas. Esto confirma que los algoritmos de complejidad polinomial pueden escalar de manera mucho más adecuada en aplicaciones prácticas.

B. Problema del Agente Viajero

La Tabla II muestra una comparación entre los métodos basados en fuerza bruta y algunos algoritmos más eficientes que pueden utilizarse para resolver los mismos problemas o problemas similares.

TABLA II

FUERZA BRUTA VS ALGORITMOS EFICIENTES

Problema	Fuerza bruta	Algoritmo eficiente	Complejidad eficiente
TSP	O(n!)	Held-Karp [4]	$O(2^n \cdot n^2)$
String matching	O(n·m)	KMP [18]	O(n+m)
String matching	O(n·m)	Boyer-Moore [17]	O(n/m) mejor caso
Grafos	$O(V^3)$	Dijkstra [19]	O((V+E) log V)
Mochila 0/1	$O(2^n)$	Prog. dinámica [1]	O(n·W)

Esta comparación permite observar que, aunque la fuerza bruta puede ser sencilla de implementar y útil para entender el problema, existen alternativas mucho más eficientes cuando el tamaño de la entrada aumenta. Algoritmos como Held-Karp, KMP, Boyer-Moore, Dijkstra y la programación dinámica reducen considerablemente el tiempo de ejecución en comparación con la exploración exhaustiva.

IV. DISCUSIÓN

A. Complejidad y Escalabilidad

Los datos que se ven aquí van junto con lo que dicen Cormen et al. [1] y Levitin [2]. La diferencia entre tener una complejidad O(n) y una O(n!) no es solo de números; es una distancia muy grande si hablamos de lo que pasa al usarlo. Por ejemplo, cuando n es 20, una búsqueda línea solo hace 20 pasos o comparaciones, pero para resolver el TSP con fuerza bruta se deben mirar cerca de 2.43 x 10^18 casos. Ese número es tan grande que las máquinas rápidas de hoy no pueden trabajar con tantos datos, aunque corran casi a 10^11 pasos por segundo [14].

Garey y Johnson [11] hablaron de esta clase de problemas con la idea de NPcompletitud. El problema TSP está en ese grupo de problemas NPcompletos. Esto quiere decir que aún no hay nadie que muestre un

algoritmo que funcione bien y rápido para resolverlo si queremos una respuesta exacta. Si usamos la fuerza bruta, el tiempo se hace muy largo, ya que el número de pasos aumenta de forma muy fuerte cuando el tamaño del problema crece. Por esto, usar una mejora, aunque sea pequeña, puede cambiar por completo lo útil que es el método.

B. Alternativas para el TSP

El algoritmo de HeldKarp [4], que se creó en 1962 y que va junto con ideas de Bellman [10] sobre usar partes para resolver problemas grandes, ayudó a bajar la dificultad del TSP exacto de O(n!) a O(2^n n^2). Este tipo de dificultad aún es muy alta, pero es mucho mejor que la fuerza bruta y ayuda mucho. Para n igual a 20, la forma dura pide cerca de 2.43 x 10^18 pasos, pero HeldKarp hace que se necesiten menos pasos y así la diferencia es muy grande.

Para casos más grandes, hay otras ideas que no buscan la respuesta perfecta, pero sí una que esté cerca. Un ejemplo es el método por recocido simulado que dijeron Kirkpatrick et al. [5]. Esta clase de método ayuda a encontrar respuestas buenas en menos tiempo, lo cual es útil en muchos casos prácticos, cuando no podemos usar mucho tiempo ni recursos.

C. Fuerza Bruta en Búsqueda de Patrones

En la búsqueda de patrones dentro de cadenas de texto, el enfoque de fuerza bruta tiene una complejidad de O(n·m), ya que compara carácter por carácter el patrón con el texto. Sin embargo, este método es superado por algoritmos más eficientes, como el de Knuth, Morris y Pratt [18], cuya complejidad es O(n+m), y el de Boyer y Moore [17], que en el mejor caso puede alcanzar O(n/m).

Según Skiena [7], la ventaja de estos algoritmos está en que aprovechan la información obtenida durante las comparaciones fallidas para evitar repetir trabajo innecesario. Esta idea no está presente en la fuerza bruta, que reinicia la comparación de forma directa sin reutilizar información previa. En textos extensos, esta diferencia puede traducirse en mejoras muy notables en el tiempo real de ejecución.

D. Fuerza Bruta en Grafos y Criptografía

En el caso de los grafos, Dijkstra [19] demostró que, mediante el uso de una cola de prioridad, es posible encontrar el camino más corto desde un nodo origen con una complejidad de O((V+E) log V). Esto contrasta con enfoques de fuerza bruta que exploran múltiples caminos posibles y pueden alcanzar complejidades mucho menos eficientes, como $O(V^3)$ en el peor caso. Aho, Hopcroft y Ullman [16] ya señalaban que elegir adecuadamente las estructuras de datos puede ser tan importante como seleccionar una buena estrategia algorítmica.

En seguridad informática, la fuerza bruta cumple un papel distinto. Schneier [6] sostiene que la fortaleza de un sistema criptográfico se evalúa, en gran medida, por su resistencia ante este tipo de ataques. Por ejemplo, un cifrado como AES-256 requeriría teóricamente 2^{256} intentos para ser vulnerado mediante fuerza bruta, una cantidad tan grande que resultaría impracticable incluso con la capacidad computacional disponible actualmente. En este escenario, la fuerza bruta no se considera un método de ataque viable, sino una referencia para medir el nivel de seguridad que debe superar un sistema.

E. Branch and Bound: Fuerza Bruta Inteligente

Kleinberg y Tardos [8] describen la técnica de branch and bound como una mejora de la fuerza bruta tradicional. Esta estrategia también explora un espacio de soluciones, pero incorpora mecanismos de poda que permiten descartar ramas del árbol de búsqueda cuando se sabe que no podrán producir una solución mejor que la ya encontrada.

De esta manera, branch and bound conserva la garantía de encontrar una solución óptima, pero reduce de forma considerable la cantidad de combinaciones que deben evaluarse. Knuth [12] analiza estas técnicas en problemas de ordenamiento y búsqueda, mostrando que la poda puede disminuir de manera importante el trabajo necesario, pasando de crecimientos factoriales a comportamientos mucho más manejables en ciertos tipos de instancias.

V. CONCLUSIÓN

A partir de lo revisado en la teoría y en las pruebas realizadas, es posible extraer varias conclusiones sobre el uso de la fuerza bruta.

Este método resulta útil cuando el espacio de soluciones es reducido, ya que garantiza encontrar la respuesta óptima. Sin embargo, su utilidad se pierde rápidamente a medida que aumenta el tamaño del problema, debido al crecimiento acelerado del tiempo de ejecución. Un ejemplo claro se observa en problemas de orden O(n!). Con un valor de n igual a 20, el tiempo requerido se vuelve impracticable, mientras que enfoques como la programación dinámica logran resolver el mismo problema en tiempos considerablemente menores.

A pesar de estas limitaciones, la fuerza bruta mantiene un valor importante como herramienta de referencia. Aplicada sobre instancias pequeñas, permite verificar si algoritmos más sofisticados están dando resultados correctos, y suele emplearse también como punto de partida al momento de diseñar nuevas soluciones.

El análisis de otras alternativas, entre ellas la programación dinámica a través del algoritmo de Held-Karp, los métodos de búsqueda de patrones KMP y Boyer-Moore, el algoritmo de Dijkstra para grafos y la técnica de branch and bound, muestra que aprovechar la estructura particular de cada problema puede reducir de forma considerable el tiempo y los recursos necesarios para resolverlo, sin dejar de garantizar una solución óptima.

En el campo de la criptografía moderna, la fuerza bruta también cumple un rol relevante como medida mínima de seguridad. Un sistema criptográfico se considera seguro cuando probar todas las combinaciones posibles requiere más recursos de los que existen actualmente.

Finalmente, como línea de investigación futura se propone explorar enfoques híbridos que combinen la fuerza bruta con técnicas de poda como branch and bound, así como con métodos como los algoritmos genéticos o las colonias de hormigas. Esto permitiría entender mejor el equilibrio entre la calidad de las soluciones obtenidas y la eficiencia práctica al resolver problemas complejos.

VI. RECONOCIMIENTOS

Los autores agradecen a la Universidad y al docente de Estrategias Algorítmicas por el apoyo académico que orientaron la realización de este trabajo.

VI. REFERENCIAS BIBLIOGRÁFICAS

[1]. *T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 4th ed. Cambridge, MA: MIT Press, 2022.*

[2]. *A. Levitin, Introduction to the Design and Analysis of Algorithms, 3rd ed. Upper Saddle River, NJ: Pearson, 2012.*

[3]. *R. Sedgewick and K. Wayne, Algorithms, 4th ed. Upper Saddle River, NJ: Addison-Wesley Professional, 2011.*

[4]. *M. Held and R. M. Karp, "A dynamic programming approach to sequencing problems," J. Soc. Ind. Appl. Math., vol. 10, no. 1, pp. 196–210, 1962.*

[5]. *S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," Science, vol. 220, no. 4598, pp. 671–680, May 1983.*

[6]. *B. Schneier, Applied Cryptography, 2nd ed. New York, NY: Wiley, 1996.*

[7]. *S. S. Skiena, The Algorithm Design Manual, 2nd ed. New York, NY: Springer, 2008.*

[8]. J. Kleinberg and E. Tardos, Algorithm Design. Boston, MA: Addison-Wesley, 2006.

[9]. *T. H. Cormen, Algorithms Unlocked. Cambridge, MA: MIT Press, 2013.*

[10]. *R. Bellman, "Dynamic programming treatment of the travelling salesman problem," J. ACM, vol. 9, no. 1, pp. 61–63, 1962.*

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

[12]. *D. E. Knuth, The Art of Computer Programming, Vol. 1: Fundamental Algorithms, 3rd ed. Reading, MA: Addison-Wesley, 1997.*

[13]. *D. E. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching, 2nd ed. Reading, MA: Addison-Wesley, 1998.*

[14]. C. H. Papadimitriou, Computational Complexity. Reading, MA: Addison-Wesley, 1994.

[15]. R. M. Karp, "Reducibility among combinatorial problems," in Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, Eds. New York, NY: Plenum Press, 1972, pp. 85–103.

[16]. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms. Reading, MA: Addison-Wesley, 1974.

[17]. *R. S. Boyer and J. S. Moore, "A fast string searching algorithm," Commun. ACM, vol. 21, no. 10, pp. 762–772, Oct. 1978.*

[18]. *D. E. Knuth, J. H. Morris, and V. R. Pratt, "Fast pattern matching in strings," SIAM J. Comput., vol. 6, no. 2, pp. 323–350, 1977.*

[19]. *E. W. Dijkstra, "A note on two problems in connexion with graphs," Numer. Math., vol. 1, pp. 269–271, 1959.*

[20]. *A. M. Turing, "On computable numbers, with an application to the Entscheidungsproblem," Proc. London Math. Soc., vol. 42, pp. 230–265, 1936.*