República Bolivariana de Venezuela
Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación
Técnicas Avanzadas de Programación
Semestre 1-2025

Optimizaciones de *Backtracking* para el Problema de las N-Reinas

Estudiante:
César Carios
CI 30.136.117

9 de mayo del 2025

Optimización 1: Una Reina por Fila

Tomando en cuenta que la complejidad temporal de la implementación del *backtracking* sin optimizar es $O(N^N)$[1], podemos pensar en otras formas de mejorar la eficiencia de este algoritmo para el problema de las N Reinas. La optimización por filas se basa en el principio de que, en un tablero de $N \times N$, cada solución válida debe tener exactamente una reina en cada fila. Por lo tanto, podemos estructurar la búsqueda para colocar automáticamente una reina por fila, eliminando la necesidad de considerar múltiples reinas en la misma fila.

La forma de implementar esta optimización es con una técnica de marcado: se coloca una reina en cada fila, luego se marcan los lugares a los que la reina puede acceder con sus movimiento (columna y diagonales). Para evitar el problema de que al desmarcar una casilla donde el movimiento de dos reinas se intercepten se termine desmarcando la reina que si está bien ubicada, usamos una solución simple: que cada reina y sus marcas tengan una identificación única.

Esta optimización reduce la complejidad temporal del backtracking clásico a $O(N!)$[2]:

- Cada reina tiene como máximo N posiciones posibles
- Pero cada reina subsiguiente tiene $N-1$, $N-2$, etc.
- El número total de permutaciones es $N \times (N-1) \times \cdots \times 1 = N!$
- La verificación de diagonales añade $O(N)$ por nodo, pero no cambia el orden factorial

En el caso de la complejidad espacial, tenemos que es $O(N^2)$:

- La matriz es $board[N][N]$: $O(N^2)$

Los resultados de las pruebas para las diferentes cantidades de reinas fueron los siguientes:

Cuadro 1: Tiempos de ejecución para diferentes tamaños de tablero N de la optimización 1.

N	Iteraciones	Tiempo (ms)	Tiempo (ml)	Tiempo (s)	Tiempo (min)
8	100,000	11.6928	0.0116928	0	0
16	10,000	5003.84	5	0	0
32	50	7.03×10^{12}	70327.7	703	12
64	1	∞	∞	∞	∞

Se observa que, dependiendo del tamaño de N, hay un aumento rápido del tiempo en el que se tarda el algoritmo en encontrar la primera solución. El caso especial acá es cuando $N = 64$, luego de que el código se ejecutara por horas no se obtuvo ningún resultado.

[1] Artificial Intelligence: A Modern Approach. (Russell And Norvig, 4th edition)
[2] `https://jeffe.cs.illinois.edu/teaching/algorithms/`

Optimización 2: Arreglo de Tamaño N

Esta implementación resuelve el problema de las N reinas utilizando backtracking con permutaciones y poda temprana, representando el tablero mediante un arreglo unidimensional de tamaño N para mayor eficiencia. Los índices del arreglo representan las filas y los valores del arreglo representan las columnas, de esta forma se asegura de que haya una reina por cada fila y columna. Por lo tanto, solo habría que encontrar las diferentes permutaciones del arreglo que cuenten como una solución, y esto se hace validando solamente las diagonales de las reinas con una fórmula sencilla: si $A_{fila} - B_{fila} = A_{columna} - B_{columna}$ entonces las dos reinas que se están comparando están en la misma diagonal, por lo que hay que hacer una permutación.

La complejidad temporal de este algoritmo es de orden factorial:

- El algoritmo genera permutaciones de columnas mediante backtracking. Para un tablero de $n \times n$:

$$\text{Permutaciones} = n \times (n-1) \times (n-2) \times \cdots \times 1 = n!$$

- En cada paso, verifica conflictos en diagonales para las reinas ya colocadas:

$$\text{Operaciones por nodo} = \sum_{k=1}^{n} k = \frac{n(n-1)}{2} \in O(n^2)$$

- Aunque la verificación es $O(n^2)$ por nodo, domina el término factorial:

$$T(n) = n! \times O(n^2) \in O(n!)$$

La complejidad espacial es lineal:

- **Arreglo de reinas**:

$$\text{queens}[n] \Rightarrow O(n)$$

Para esta optimización, los resultados de las pruebas fueron los siguientes:

Cuadro 2: Tiempos de ejecución para diferentes tamaños de tablero N de la optimización 2.

N	Iteraciones	Tiempo (ms)	Tiempo (ml)	Tiempo (s)	Tiempo (min)
8	100,000	472	0.00472009	4.72×10^{-01}	$7866.81666667\times10^{-05}$
16	10,000	1848.78	2	0.002	0.00003333333
32	50	25195300	25195.3	25.1953	0.4199216667
64	1	∞	∞	∞	∞

Claramente hay una mejora en los tiempos de ejecución respecto a la optimización anterior (a pesar de que tiene la misma complejidad temporal), sin embargo, ocurre lo mismo cuando se llega a $n = 64$: luego de que el código se ejecutara por horas, no arrojó ningún resultado.

Optimización 3: Versión Probabilística de la Primera Optimización

Esta optimización sigue prácticamente la misma lógica de la primera solución que se implementó, la diferencia es que el propósito es que el algoritmo se apoye en la aleatoriedad para disminuir la búsqueda de la soluciones. En este caso, solo se usa 30 % de las posiciones de cada fila, esto se logra haciendo uso del algoritmo *Mersenne Twister 19937*[3], la función *shuffle*[4] e *iota*[5].

La complejidad temporal es:

$$T(n) \in O(N \cdot N! \cdot P)$$

donde:

- $N =$ BOARD_SIZE (tamaño del tablero)
- $P =$ PROBABILITY (probabilidad de exploración)

- El algoritmo explora solo una fracción P de las columnas en cada nivel:

$$\text{Posiciones a verificar} = \lceil P \cdot N \rceil$$

- Para cada columna candidata, verifica conflictos con $O(n)$ reinas ya colocadas.

En el caso de la complejidad espacial:

$$S(n) \in O(n^2)$$

- Matriz del tablero:

$$\text{board}[n][n] \Rightarrow O(n^2)$$

[3] `https://cplusplus.com/reference/random/mt19937/`
[4] `https://cplusplus.com/reference/algorithm/shuffle/`
[5] `https://en.cppreference.com/w/cpp/algorithm/iota`

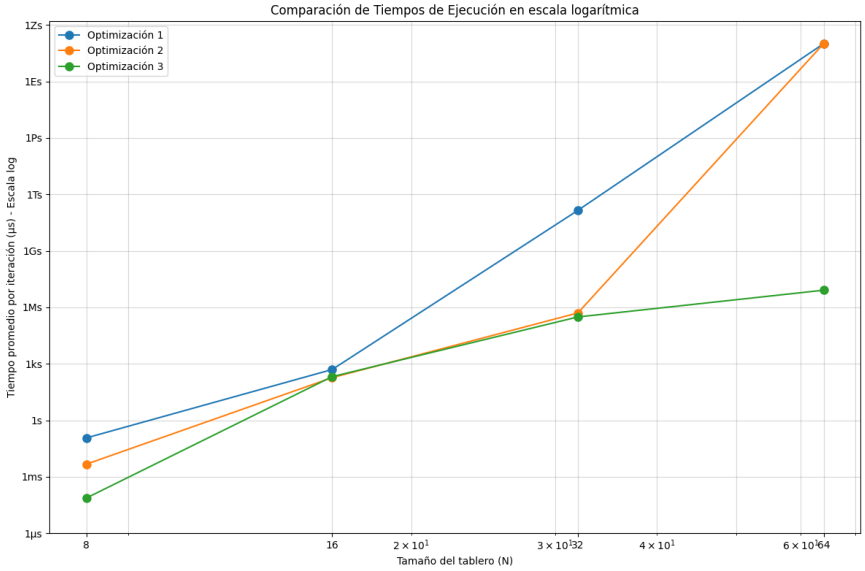
Veamos los resultados obtenidos a partir de las pruebas que se hicieron con este algoritmo:

Cuadro 3: Tiempos de ejecución para diferentes tamaños de tablero N de la optimización 3.

Iteraciones	N	Tiempo (ms)	Tiempo (ml)	Tiempo (s)	Tiempo (min)	Tasa de éxito
100,000	8	7,55	0,007 55	$7{,}55 \times 10^{-6}$	$1{,}26 \times 10^{-7}$	0.14 %
10,000	16	2092,44	2,092 44	$2{,}09 \times 10^{-3}$	$3{,}49 \times 10^{-5}$	0.30 %
50	32	15 467,10	15 467,100 00	$1{,}55 \times 10^{1}$	$2{,}58 \times 10^{-1}$	100 %
10	64	81 421 800,00	81 421,800 00	$8{,}14 \times 10^{1}$	1,36	100 %

Los resultados obtenidos nos muestran que es un algoritmo muy inestable: se puede conseguir solución de una forma muy rápida o a veces no se consigue ninguna solución. Podemos destacar que es la única optimización que si arrojó resultados para $N = 64$

Observemos el comportamiento de las tres optimizaciones en un gráfico:



Podemos concluir que las dos primeras implementaciones (sobre todo la segunda) sirven para tableros pequeños como $N \leq 32$, pero la optmización probalística funciona perfectamente para tableros más grandes, porque en los pequeños se vuelve ineficaz e ineficiente.