



Instituto Politecnico Nacional
Unidad Profesional Interdedisciplinaria de
Ingenieria Campus Zacatecas

Analisis y diseño de algoritmos

Unidad III Practica 04

Optimización del Algoritmo de Backtracking
para el Problema de las N Reinas

Kevin Martin Ramirez Reyna
Fernando Ruiz Correa
3M2

Docente:
M. en C. Erika Sanchez-Fermat

1. Introducción

Este proyecto se centra en la implementación y optimización del algoritmo de backtracking en Python para resolver el problema de las N Reinas. Se explorarán dos estrategias específicas de optimización, y se realizará un análisis de rendimiento comparativo, incluyendo gráficas que ilustren el tiempo de ejecución de las diferentes versiones del código.

2. Desarrollo

2.1. Implementación Básica Backtracking

- Backtracking

Es una técnica algorítmica para encontrar soluciones a problemas que tienen una solución completa, en los que el orden de los elementos no importa, y en los que existen una serie de variables, a cada una de las cuales, debemos asignarle un valor teniendo en cuenta unas restricciones dadas. O lo que es lo mismo, es una estrategia algorítmica que busca todas las posibles soluciones dado un conjunto de variables inicial para encontrar el resultado definido por el problema. La técnica de Backtracking se apoya en el uso de la recursividad para la búsqueda exhaustiva de todas las combinaciones posibles.

- N-Reinas

El problema de las n-reinas consiste en colocar n reinas en un tablero de ajedrez de $n \times n$ de tal manera que no sea posible que dos reinas se capturen entre si, es decir, que no estén en la misma fila, ni en la misma columna ni en la misma diagonal. Se dice que hay una colisión si hay dos reinas que se pueden capturar entre si. Se trata pues de encontrar una configuración – elegir las n celdas donde colocar a las reinas- que minimice el número total de colisiones.

	Columna1	Columna2	Columna3	Columna4
Reina1			R1	
Reina2				R2
Reina3	R3			
Reina4		R4		

2.1.1. Implementación del Algoritmo

El algoritmo de N-Reinas, se implementara en el lenguaje de programación Python, con la funcion n queens. En las siguientes figuras se muestra el algoritmo implementado en python.



```
Queens.py 1 X
Queens.py > poda
1
2 import time
3 import matplotlib.pyplot as plt
4
5 def solución_valida(fila, col, queens):
6     for f in range(fila):
7         if col == queens[f]:
8             return False
9         elif abs(col - queens[f]) == abs(fila - f):
10            return False
11    return True
12
13 def tiempo():
14     inicio = time.time()
15     return time.time() - inicio
16
17 def n_queens(n):
18     queens = [' ' * n
19     resultado = colocar_queen(0, queens, n)
20     print(f"El número total de soluciones para el problema de las N reinas mediante backtracking con {n} reinas es: {resultado}")
21
22 def imprimir_tablero(tablero):
23     n = len(tablero)
24     for fila in range(n):
25         for col in range(n):
26             if tablero[fila] == col:
27                 print("Q", end=" ")
28             else:
29                 print(".", end=" ")
30         print()
31
32 def colocar_queen(fila, queens, n):
33     if fila == n:
34         imprimir_tablero(queens)
35         return 1
36     else:
37         solucion_total = 0
38         for col in range(n):
39             if solución_valida(fila, col, queens):
40                 queens[fila] = col
41                 solucion_total += colocar_queen(fila + 1, queens, n)
42         return solucion_total
43
44 def poda(fila, queens, n):
45     if fila == n:
46         imprimir_tablero(queens)
47         return 1
48     else:
49         sol = 0
50         for col in range(n):
51             if solución_valida(fila, col, queens):
52                 queens[fila] = col
53                 if fila == 0 or (fila > 0 and solución_valida(fila, col, queens)):
54                     sol += poda(fila + 1, queens, n)
55                 queens[fila] = -1
56         return sol
57
58 def n_queens_poda(n):
59     queens = ['-1'] * n
60     resultado = poda(0, queens, n)
61     print(f"El número total de soluciones para el problema de las N reinas mediante poda con {n} reinas es: {resultado}")
62
63 def heuristica(fila, queens, n):
64     for col in range(n):
65         if solución_valida(fila, col, queens):
66             return col
67     return None
68
69 def colocar_queen_heuristica(fila, queens, n):
70     if fila == n:
71         imprimir_tablero(queens)
72         return 1
73     else:
74         solucion_total = 0
75         posicion_valida = heuristica(fila, queens, n)
```

```
Queens.py 1 X
Queens.py > ...
75         if posicion_valida is not None:
76             queens[fila] = posicion_valida
77             solucion_total += colocar_queen_heuristica(fila + 1, queens, n)
78             queens[fila] = ' '
79         return solucion_total
80
81 def n_queens_heuristica(n):
82     queens = [' '] * n
83     resultado = colocar_queen_heuristica(0, queens, n)
84     print(f"El número total de soluciones para el problema de las N reinas mediante heurísticas inteligentes con {n} reinas es: {resultado}")
85
86 def medir_tiempo(tam, estrategia):
87     tiempos = []
88     for n in tam:
89         inicio = time.time()
90         if estrategia == "backtracking":
91             n_queens(n)
92         elif estrategia == "poda":
93             n_queens_poda(n)
94         elif estrategia == "heuristica":
95             n_queens_heuristica(n)
96         else:
97             raise ValueError("Estrategia no válida")
98         tiempo_ejecucion = time.time() - inicio
99         tiempos.append(tiempo_ejecucion)
100     return tiempos
101
102 def crear_grafica(tamano_problema, tiempos, estrategia):
103     plt.plot(tamano_problema, tiempos, label=estrategia)
104     plt.xlabel("Tamaño del problema (N)")
105     plt.ylabel("Tiempo de ejecución (segundos)")
106     plt.title("Tiempo de ejecución en función del tamaño del problema")
107     plt.legend()
108     plt.show()
109
110 # Prueba con N desde 1 hasta 10
111 tamano_problema = list(range(1, 10))
112 tamano_problema = list(range(1, 10))
113
114 # Medir tiempo de ejecución para la versión básica
115 tiempos_backtracking = medir_tiempo(tamano_problema, "backtracking")
116 crear_grafica(tamano_problema, tiempos_backtracking, "Backtracking")
117 # Medir tiempo de ejecución para la versión con poda por conflicto
118 tiempos_poda = medir_tiempo(tamano_problema, "poda")
119 crear_grafica(tamano_problema, tiempos_poda, "Poda")
120
121 # Medir tiempo de ejecución para la versión con heurística
122 tiempos_heuristica = medir_tiempo(tamano_problema, "heuristica")
123 crear_grafica(tamano_problema, tiempos_heuristica, "Heurística")
```

Resultado Backtracking

The image displays a screenshot of a Visual Studio Code editor window. The background features a large, faint illustration of a character with long black hair and a red headband. In the bottom right corner, there is a small, colorful anime-style character with pink hair and a surprised expression.

The editor's interface includes a top bar with the file name "Queens.py" and icons for Explorer, Search, and Run and Debug. Below this is a sidebar with tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL, and PORTS. The TERMINAL tab is active, showing the execution of the Python script.

The main workspace contains the following Python code:

```
def poda(fila, queens, n):  
    if fila == n:  
        imprimir_tablero(queens)  
        return 1  
    else:  
        sol = 0  
        for col in range(n):  
            if solución_valida(fila, col, queens):  
                queens[fila] = col  
                if fila == 0 or (fila > 0 and solución_valida(fila, col, queens)):  
                    sol += poda(fila + 1, queens, n)  
                queens[fila] = -1  
        return sol  
  
def n_queens_poda(n):  
    queens = [-1] * n  
    resultado = poda(0, queens, n)  
    print(f"El número total de soluciones para el problema de las N reinas mediante poda con {n} reinas es: {resultado}")
```


The terminal output at the bottom shows the board configurations for each solution found, represented by dots and 'Q' characters, followed by the final message: "El número total de soluciones para el problema de las N reinas mediante backtracking con 9 reinas es: 352".

En estas imagenes, se puede apreciar como es que se implemento los tres metodos de N Reinas las cuales nos sirven para poner en practica el modo Backtracking . Ademas de tener como ultima el resultado de Backtracking

2.1.2. Analisis Complejidad

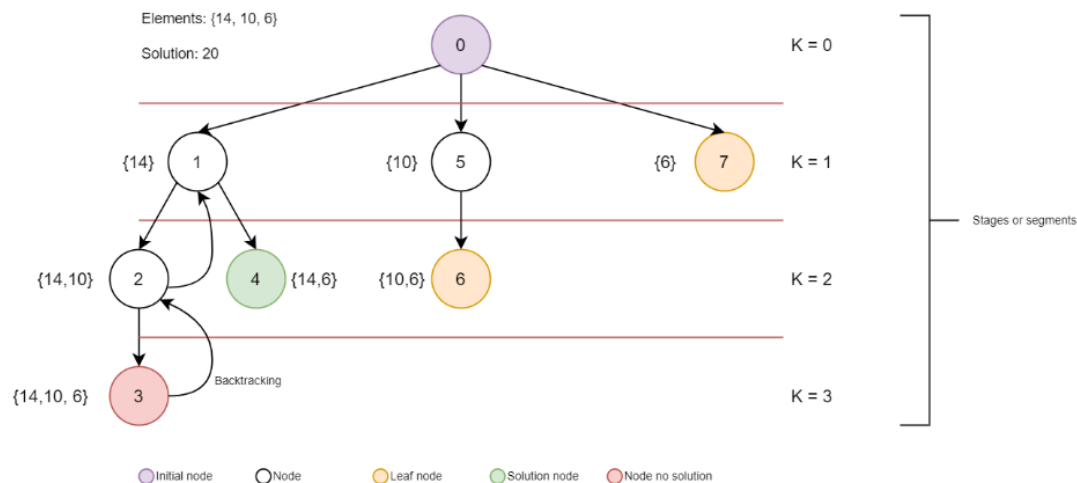
Para realizar una búsqueda exhaustiva en el espacio de soluciones del problema, los algoritmos de backtracking son bastante ineficientes. En general, se tienen tiempos con órdenes de complejidad factoriales o exponenciales $O(\text{Exp } n)$. Por esto, los algoritmos de backtracking se utilizan en problemas para los que no existen un algoritmo eficiente que los resuelva. Aunque se utilizan para resolver problemas para los que no existe un algoritmo eficiente.

2.1.3. Limitaciones

Algunas de las limitaciones que tiene este algoritmo es que es bastante ineficiente, es por ello que se utiliza generalmente cuando se necesita encontrar todas las soluciones posibles o cuando se debe encontrar una solución óptima sujeta a ciertas restricciones, además de que en

el peor de los casos, el algoritmo puede tomar mucho tiempo, ya que su complejidad temporal es exponencial, y puede requerir una cantidad significativa de memoria para mantener las estructuras de datos durante la recursión.

2.1.4. Ejemplos



En esta imagen se puede ver como es que funciona el backtracking, el cual es un metodo de vuelta atras, el cual es un algoritmo que se encarga basicamente de volver atras y seguir buscando mas opciones, en el caso de que el actual no haya sido el correcto.

Tomemos como ejemplo este caso, en el cual tenemos un conjunto de numero enteros(14,10 y 6) y se necesita encontrar algun subconjunto de esta que sumadas den como resultado 20.

De un inicio tenemos el nodo inicial que es el 0, en el cual tenemos a ninguno de estos numeros. Posteriormente tenemos los nodos 1, 5, 7 los cuales son cada uno de los valores $\text{Nodo1} = 14$, $\text{Nodo5} = 10$, $\text{Nodo7} = 6$, estos se ubican en el primer subconjunto. Posteriormente nos centraremos en el Nodo1 , donde primero tendremos la suma del Nodo1 y nodo5 , cuya suma es 24, es decir se pasa, por lo cual se va a la ultima suma que es la de Nodo7 , este subconjunto da la suma de los 3 es del subconjunto 3, para darnos cuenta que tampoco sirvio, y como son todos los elementos no ahi una siguiente opcion, lo que hacemos es poner en practica el **Backtracking**, con el cual volveremos atras para ver el Nodo4 , el cual es la suma del Nodo1 y Nodo7 , los cuales si nos dan como resultado 20, y de esta manera hemos ocupado el backtracking, para este problema.

2.2. Estrategia 1: Método de poda

En un problema dado da la mejor solución implementando un árbol de expansión donde sus nodos son los candidatos a solución. Se encarga de detectar en que ramificación la soluciones dadas ya no son optimas, para podar esa rama del árbol y no continuar gastando recursos y procesos. El algoritmo determina que rama va ramificar para llegar a la solución.

2.2.1. Implementación del Algoritmo



```
43 def poda(fila, queens, n):
44     if fila == n:
45         imprimir_tablero(queens)
46         return 1
47     else:
48         sol = 0
49         for col in range(n):
50             if solución_valida(fila, col, queens):
51                 queens[fila] = col
52                 if fila == 0 or (fila > 0 and solución_valida(fila, col, queens)):
53                     sol += poda(fila + 1, queens, n)
54                 queens[fila] = -1
55         return sol
56
57 def n_queens_poda(n):
58     queens = [-1] * n
59     resultado = poda(0, queens, n)
60     print(f"El número total de soluciones para el problema de las N reinas mediante poda con {n} reinas es: {resultado}")
61
```

En el contexto del problema de las N Reinas se refiere a la estrategia de eliminar ciertas ramas de un árbol de búsqueda para así evitar explorar combinaciones que se sabe que no conducirán a una solución válida, es decir que daría una solución errónea el camino. Esto con la intención de mejorar el algoritmo y hacerlo mejor eficiente, reduciendo el número total de nodos explorados.

En el problema de las N reinas, el método de poda se utiliza para evitar explorar configuraciones que, previamente, sabemos que no conducirán a una solución. Algunas de los tipos de poda pueden ser.

- Poda por Fila

Si mediante el proceso de poner las reinas en el tablero en una fila específica, se puede determinar que no hay ninguna posición segura para la reina en esa fila, se evita esa fila, debido a que no es necesaria seguir explorando ya que sabemos que no hay solución esa fila.

- Poda por Columna

Si se encuentra una posición segura para colocar una reina en una columna específica de una fila, se puede podar la búsqueda en otras columnas de esa misma fila, ya que no se pueden colocar dos reinas en la misma columna.

- Poda por Diagonal

Al verificar la seguridad de una posición para una reina, también se puede verificar si la diagonal que cruza esa posición ya contiene una reina. Si es así, no es necesario explorar más en esa dirección.

2.2.2. Analisis Complejidad

La idea principal de este enfoque es reducir el espacio de búsqueda mediante la poda de una fracción de los elementos de entrada y recurrir a los elementos de entrada válidos restantes. Después de algunas iteraciones, el tamaño de los datos de entrada será tan pequeño que podrá resolverse mediante el método de fuerza bruta en un tiempo constante c' .

Análisis de Complejidad de Tiempo para tales algoritmos:

Sea el tiempo requerido en cada iteración $O(n^k)$

donde:

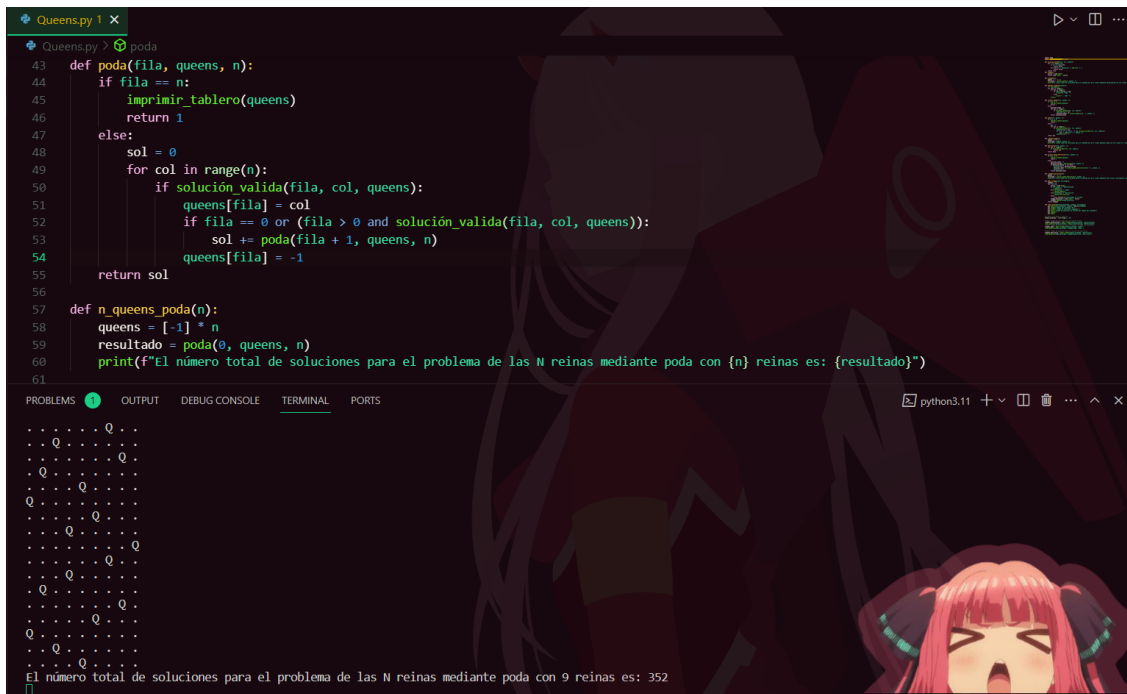
n = tamaño de los datos de entrada

k es una constante.

2.2.3. Limitaciones

El método de poda, implica descartar ciertas ramas de búsqueda para poder mejorar la eficiencia, tiene limitaciones a considerar. Entre ellas está la complejidad de implementación, la dependencia del conocimiento del problema, la posible pérdida de optimalidad, el impacto en la modularidad del código, el costo adicional de evaluación y la falta de garantía de eficiencia universal son aspectos importantes a tener en cuenta al aplicar técnicas de poda. Sin embargo puede acelerar la resolución de problemas, su éxito depende de una comprensión profunda del problema y la cuidadosa consideración de sus implicaciones en términos de complejidad y mantenimiento del código.

2.2.4. Ejemplos

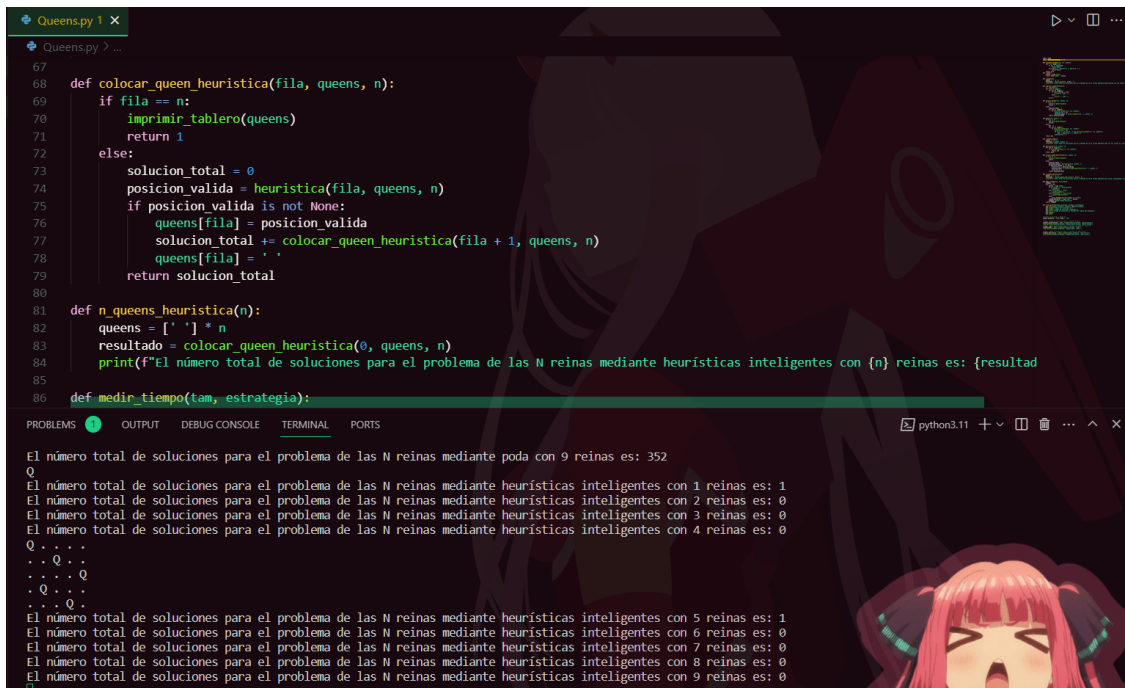


Un ejemplo de aplicación de los algoritmos de ramificación y poda lo encontramos en el problema de las N-reinas, el cuál consiste en colocar 8 reinas en un tablero de ajedrez cuyo tamaño es de 8 por 8 cuadros, las reinas deben de estar distribuidas dentro del tablero de modo que no se encuentren dos o más reinas en la misma línea horizontal, vertical o diagonal. Se han encontrado 92 soluciones posibles a este problema.

2.3. Estrategia 2: Heurística Inteligente

Los algoritmos heurísticos son métodos de solución de problemas en los cuales utilizan reglas o bien principios generales para poder encontrar soluciones aproximadas o subóptimas. A diferencia de los algoritmos exactos, los cuales buscan una solución óptima de una manera exhaustiva, y que ofrecen soluciones rápidas y cercanas a la mejor solución posible. Los algoritmos de heurísticos se basan en el razonamiento aproximado, en la experiencia y el juicio para guiar el proceso de búsqueda y llegar a una solución satisfactoria en un tiempo razonable.

2.3.1. Implementación del Algoritmo



```
Queens.py 1 x
Queens.py 7 ...
67
68 def colocar_queen_heuristica(fila, queens, n):
69     if fila == n:
70         imprimir_tablero(queens)
71         return 1
72     else:
73         solucion_total = 0
74         posicion_valida = heuristica(fila, queens, n)
75         if posicion_valida is not None:
76             queens[fila] = posicion_valida
77             solucion_total += colocar_queen_heuristica(fila + 1, queens, n)
78             queens[fila] = ' '
79         return solucion_total
80
81 def n_queens_heuristica(n):
82     queens = [' ' * n
83     resultado = colocar_queen_heuristica(0, queens, n)
84     print(f"El número total de soluciones para el problema de las N reinas mediante heurísticas inteligentes con {n} reinas es: {resultado}
85
86 def medir_tiempo(tam, estrategia):
87
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS python3.11 + v [ ] ... ^ x
El número total de soluciones para el problema de las N reinas mediante poda con 9 reinas es: 352
Q
El número total de soluciones para el problema de las N reinas mediante heurísticas inteligentes con 1 reinas es: 1
El número total de soluciones para el problema de las N reinas mediante heurísticas inteligentes con 2 reinas es: 0
El número total de soluciones para el problema de las N reinas mediante heurísticas inteligentes con 3 reinas es: 0
El número total de soluciones para el problema de las N reinas mediante heurísticas inteligentes con 4 reinas es: 0
Q . . .
. . Q .
. . . Q
. Q . .
. . . Q
El número total de soluciones para el problema de las N reinas mediante heurísticas inteligentes con 5 reinas es: 1
El número total de soluciones para el problema de las N reinas mediante heurísticas inteligentes con 6 reinas es: 0
El número total de soluciones para el problema de las N reinas mediante heurísticas inteligentes con 7 reinas es: 0
El número total de soluciones para el problema de las N reinas mediante heurísticas inteligentes con 8 reinas es: 0
El número total de soluciones para el problema de las N reinas mediante heurísticas inteligentes con 9 reinas es: 0
```

La heurística inteligente es un enfoque que utiliza reglas o estrategias que pueden no garantizar la solución óptima, pero que son efectivas en la mayoría de los casos y permiten encontrar soluciones rápidas. En el contexto del problema de las N reinas, la heurística inteligente podría incluir reglas específicas que guíen la búsqueda hacia soluciones prometedoras.

- **Heurística del primer ajuste**

Coloca la primera reina en la primera casilla disponible de la primera fila, la segunda reina en la primera casilla disponible de la segunda fila, y así sucesivamente. Esta estrategia no garantiza la solución óptima, pero puede ser rápida.

- **Heurística del mejor ajuste**

Coloca cada reina en la posición que cause menos conflictos con las reinas ya colocadas. Esta heurística evalúa diferentes posiciones y selecciona la que minimiza las amenazas.

- **Heurística del mínimo conflicto**

Mueve una reina a la posición que minimiza el número de reinas con las que entra en conflicto. Este enfoque puede ser útil cuando se trata de mejorar una solución existente.

- **Heurística del retorno atrás**

Explora todas las posibles combinaciones de colocación de reinas, retrocediendo cuando se detecta un conflicto. Utiliza una estrategia de retroceso para explorar diferentes soluciones.

- **Heurística genética**

Aplica conceptos inspirados en la evolución biológica, como la selección, cruzamiento y mutación, para encontrar soluciones. Genera una población inicial de disposiciones de reinas, evalúa su aptitud y evoluciona gradualmente hacia soluciones mejores.

Para este proyecto se utilizo principalmente la Heuristica del primer ajuste, la cual es usada frecuentemente en el caso de las N Reinas. Debido a que esta puede ser rapida y eficiente, pero a su vez la desventaja es que no garantiza una solucion optima, con los cuales puede nos puede llevar a casos inciales que requieren retocede

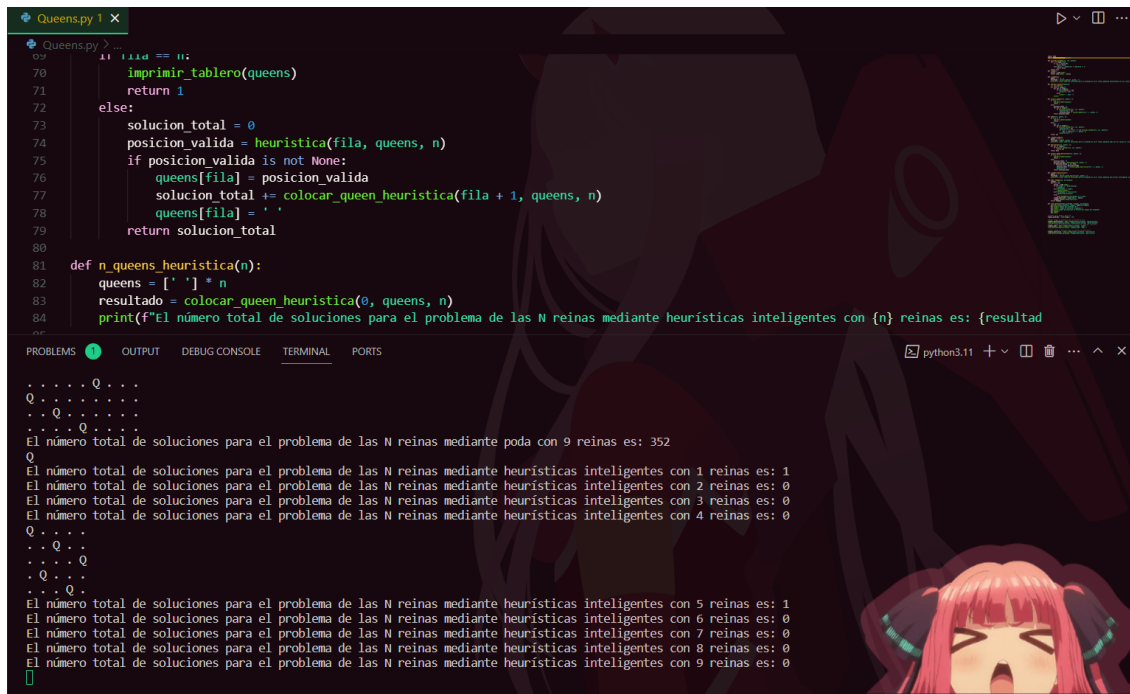
2.3.2. Analisis Complejidad

La complejidad computacional de las heurísticas inteligentes varía dependiendo de la implementación específica y del problema al que se apliquen. En general, las heurísticas inteligentes se utilizan para abordar problemas NP-completos, como el problema de las N reinas. NP-completo significa que no se conoce un algoritmo eficiente (polinómico) para resolver el problema en el peor de los casos.

2.3.3. Limitaciones

Las heurísticas inteligentes, aunque eficientes en muchos casos prácticos, no garantizan soluciones óptimas, son sensibles al problema y a la configuración inicial, y pueden quedar atrapadas en óptimos locales.

2.3.4. Ejemplos



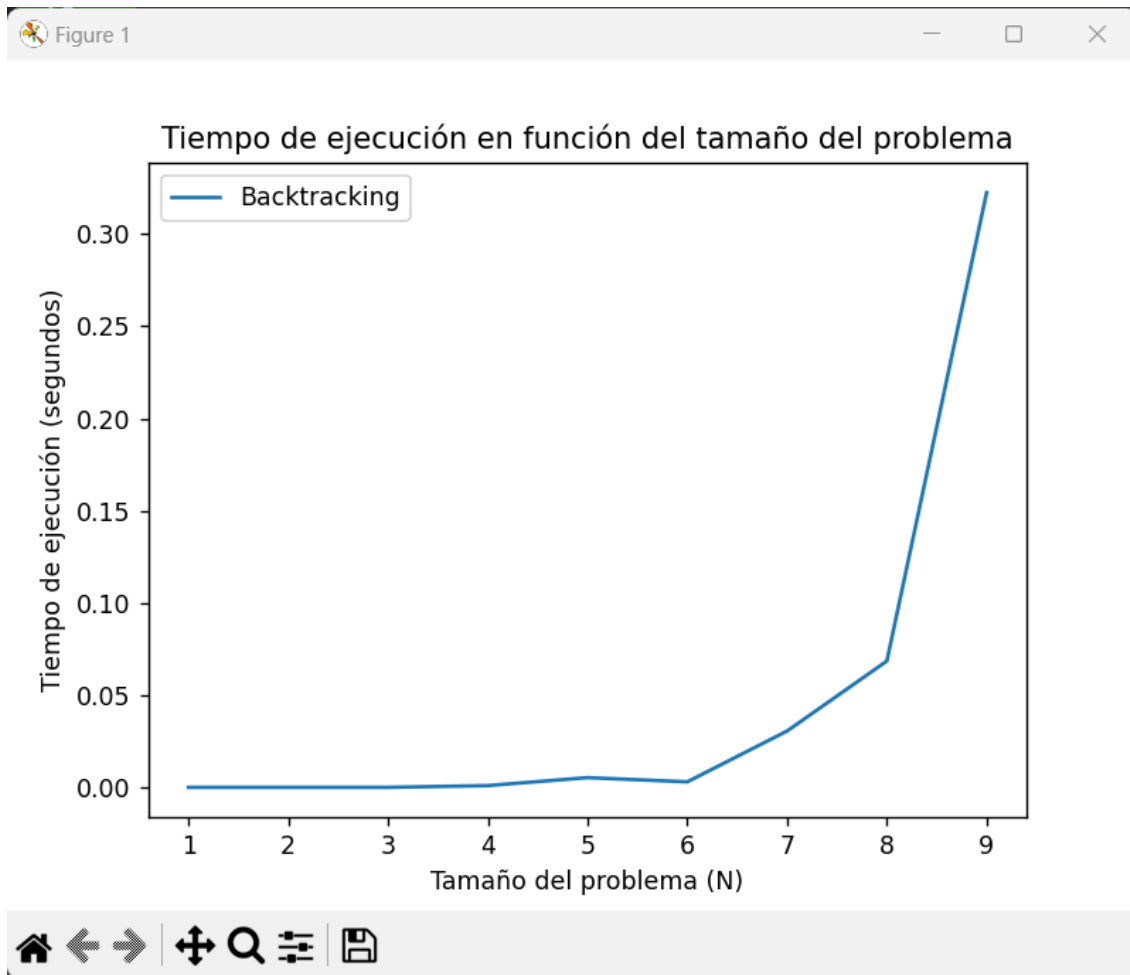
```
Queens.py x
70 if fila == n:
71     imprimir_tablero(queens)
72     return 1
73 else:
74     solucion_total = 0
75     posicion_valida = heuristica(fila, queens, n)
76     if posicion_valida is not None:
77         queens[fila] = posicion_valida
78         solucion_total += colocar_queen_heuristica(fila + 1, queens, n)
79         queens[fila] = ' '
80     return solucion_total
81
82 def n_queens_heuristica(n):
83     queens = [' '] * n
84     resultado = colocar_queen_heuristica(0, queens, n)
85     print(f"El número total de soluciones para el problema de las N reinas mediante heurísticas inteligentes con {n} reinas es: {resultado}")
86
87 PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS python3.11
88
89 . . . . . Q . . .
90 Q . . . . .
91 . . Q . . . .
92 . . . . Q . .
93 El número total de soluciones para el problema de las N reinas mediante poda con 9 reinas es: 352
94 Q . . . .
95 El número total de soluciones para el problema de las N reinas mediante heurísticas inteligentes con 1 reinas es: 1
96 El número total de soluciones para el problema de las N reinas mediante heurísticas inteligentes con 2 reinas es: 0
97 El número total de soluciones para el problema de las N reinas mediante heurísticas inteligentes con 3 reinas es: 0
98 El número total de soluciones para el problema de las N reinas mediante heurísticas inteligentes con 4 reinas es: 0
99 Q . . . .
100 . . Q . .
101 . . . . Q
102 . . Q . .
103 . . . . Q
104 El número total de soluciones para el problema de las N reinas mediante heurísticas inteligentes con 5 reinas es: 1
105 El número total de soluciones para el problema de las N reinas mediante heurísticas inteligentes con 6 reinas es: 0
106 El número total de soluciones para el problema de las N reinas mediante heurísticas inteligentes con 7 reinas es: 0
107 El número total de soluciones para el problema de las N reinas mediante heurísticas inteligentes con 8 reinas es: 0
108 El número total de soluciones para el problema de las N reinas mediante heurísticas inteligentes con 9 reinas es: 0
109
```

En el problema de las N reinas con heurística inteligente de backtracking, se explora recursivamente las posibles colocaciones de reinas, retrocediendo cuando se detectan conflictos, y aplicando estrategias heurísticas para guiar la búsqueda hacia soluciones prometedoras, reduciendo así la cantidad de ramas exploradas y mejorando la eficiencia del algoritmo.

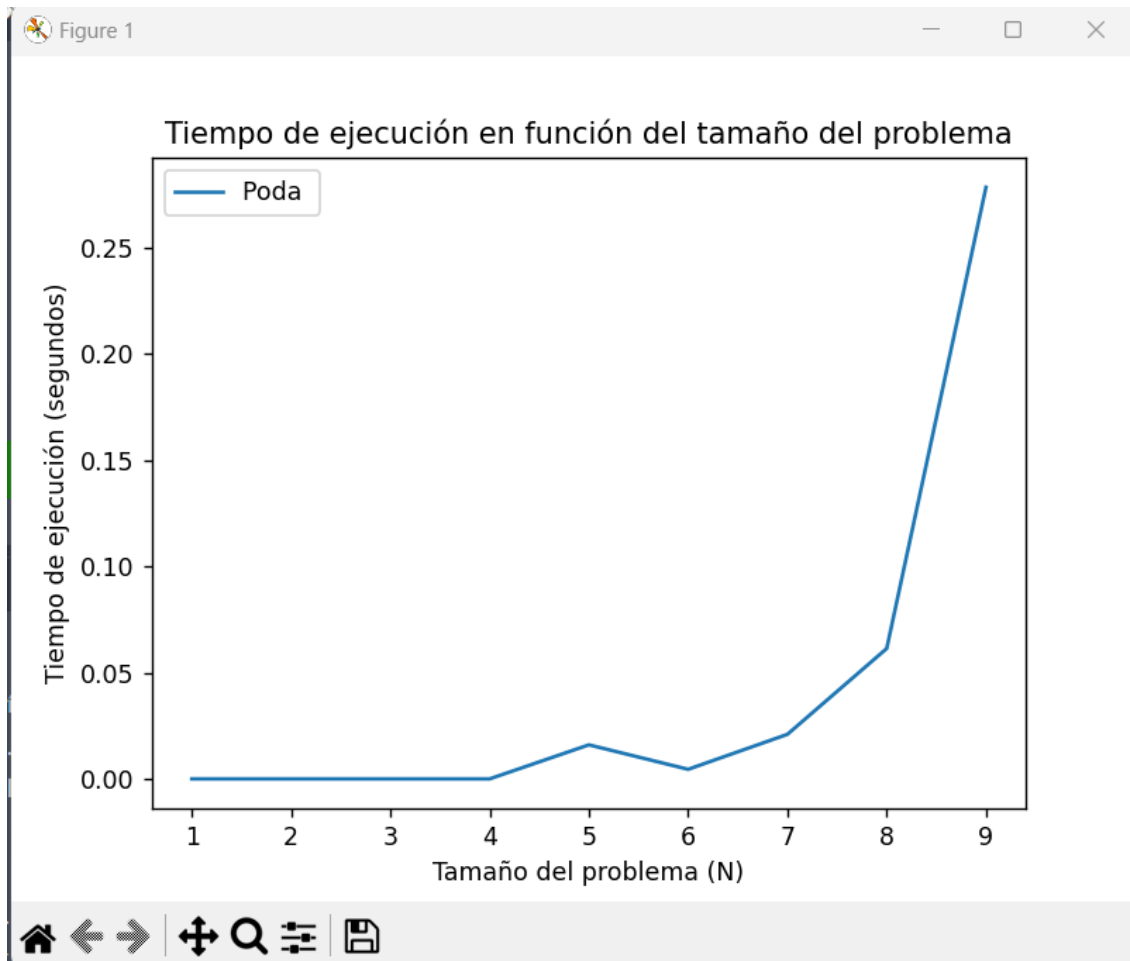
2.4. Comparación de rendimiento con gráficas

A continuación, se puede apreciar las gráficas de los resultados, con las cuales podemos ver cuánto fue la demora, según siendo el tamaño de estas y con ello, dándonos cuenta que el método de poda, sirvió para hacer más eficiente y rápido el método de backtracking, además de que el más rápido fue el método de heurística inteligente.

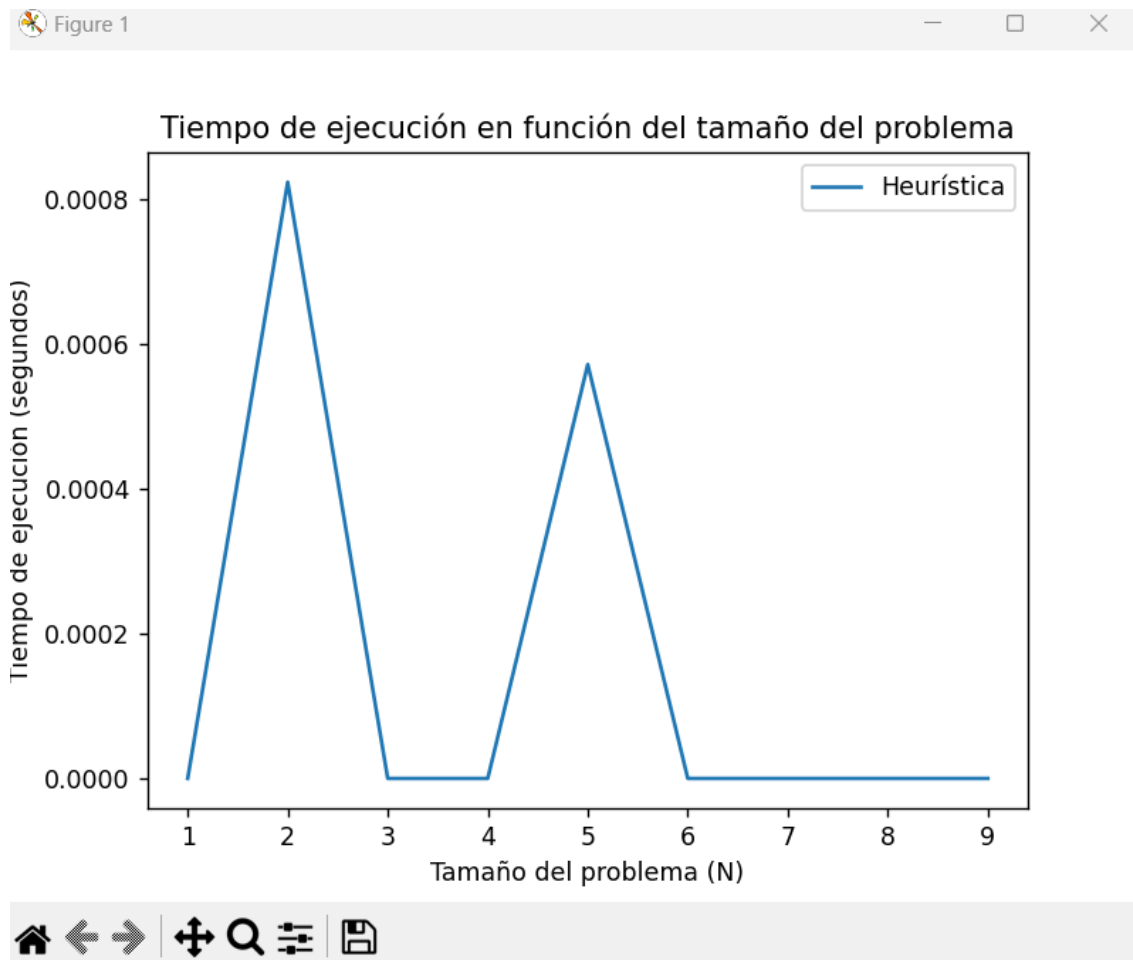
2.4.1. Gráfica del algoritmo de Backtracking



2.4.2. Gráfica del Método de poda



2.4.3. Gráfica del algoritmo de Heurística Inteligente



3. Conclusiones

- Conclusión: Kevin Martin Ramirez Reyna

Por ultimo podemos llegar a la conclusion de que el backtracking es una técnica de búsqueda exhaustiva la cual sirve para explora recursivamente todas las posibles soluciones de un problema determinaro, de tal manera que va retrocediendo cuando se hayan conflictos. El metodo de poda es una estrategia que sirva para optimizar el backtracking evitando la exploración de ramas innecesarias y con ello reduciendo así el tiempo de ejecución de este. El algoritmo de heurísticas en un enfoque basado en reglas

o estrategias que busca soluciones rápidas, sin embargo esto no garantizan que sea la óptima. Para el problema de las N reinas, estas técnicas se utilizan y se pueden combinar para encontrar soluciones de una manera mas eficiente, con ello evitando recorrer todas las combinaciones posibles que sabremos el resultado y mejorando el rendimiento del algoritmo.

- Conclusión: Fernando Ruiz Correa

En resumen, el backtracking es una técnica que busca exhaustivamente soluciones recursivamente, mientras que la poda optimiza este proceso eliminando ramas innecesarias. Las heurísticas, por otro lado, son estrategias prácticas que ofrecen soluciones rápidas aunque no siempre son las mejores alternativas, en ocasiones tienen limitaciones o solamente ocurren en casos muy específicos. Sin embargo se puede decir que estos enfoques permiten reducir la complejidad espacial y temporal de un algoritmo, no solamente del algoritmos de "n-reinas".

4. Referencias

- Corvo, H. S. (2020, 14 marzo). Programación dinámica: características, ejemplo, ventajas, desventajas. Lifeder.
<https://www.lifeder.com/programacion-dinamica/>
- Del Estado De Hidalgo, U. A. (s. f.). Boletín Científico:: UAEH.
<https://www.uaeh.edu.mx/scige/boletin/tlahuelilpan/n6/e2.html>
- 2.4 Backtracking - programación, refactoriza tu mente. (s. f.)
<https://docs.jjpeleato.com/algoritmia/backtracking>
- programacionyalgoritmia.com. (2023, 14 noviembre). Backtracking explicado: Claves para entender y aplicar esta técnica de programación – programacionyalgoritmia.
<https://programacionyalgoritmia.com/fundamentos-de-programacion/backtracking/>
- Greyrat, R. (2022, 5 julio). Podar y buscar — Una descripción general del análisis de complejidad – Barcelona Geeks.
<https://barcelonageeks.com/podar-y-buscar-una-descripcion-general-del-analisis-de-complejidad/>
- BlogAdmin, BlogAdmin. (2023, 7 julio). Algoritmos heurísticos: optimización inteligente. Informatica y Tecnologia Digital.
<https://informatecdigital.com/algoritmos/algoritmos-heuristicos-optimizacion-inteligente/googlevignette>