INSTITUTO POLITÉCNICO NACIONAL

Unidad Profesional Interdisciplinaria de Ingeniería
Campus Zacatecas.

MATERIA:

- Analisis y diseño de algoritmos

MAESTRA:

- Erika Sánchez Femat

ALUMNAS:

- Naylin Yusseth Martínez Mireles
- America Lizbet Flores Alcala
- Dalia Naomi García Macías

PROYECTO FINAL

Índice

1. INTRODUCCIÓN

1.1. ¿DE QUÉ TRATA?

Como proyecto final, presentaremos la implementación del algoritmo de Backtracking para resolver el problema de las N reynas.
Antes de comenzar con la explicación del código, tenemos que conocer que es este algoritmo y como es que funciona.
Primero consideramos relevante entender el algoritmo de forma teórica antes de poner en práctica, porque a si sería más fácil el ejecutarlo en código, al igual que lograríamos entender mejor lo que vamos a realizar.

Backtracking o también conocida como "vuela atrás.[es] una estrategia para encontrar soluciones a problemas con restricciones. El término "Backtrack"fue utilizado por primera vez por el matemático estadounidense D.H. Lehmer en la década de 1950.
Esta es una técnica algorítmica para hacer una búsqueda exhaustiva y sistemática por todas las configuraciones posibles del espacio de búsqueda del problema. Se suele aplicar en la resolución de un gran número de problemas, muy especialmente en los de decisión de optimización.

La técnica backtracking está muy relacionada con la búsqueda binaria, básicamente, la idea es encontrar la mejor combinación posible en un momento determinado, por estas características, se dice que este algoritmo es una búsqueda a profundidad.
Ahora conociendo esto es que podemos determinar como aplicar este algoritmo para resolución del problema de las N reinas.

El problema de las N reinas consiste en encontrar una distribución de "nreinas en un tablero de ajedrez de n x n, de manera que no se ataquen, es decir, no pueden encontrarse dos reinas en una misma columna, fila o en diagonal.

2. DESARROLLO

2.1. ¿CÓMO FUNCIONA?

El algoritmo de Backtracking es una técnica de resolución de problemas que se basa en la exploración sistemática de todas las posibles soluciones a un problema hasta encontrar la solución deseada.

1. Definir el problema: El primer paso es definir claramente el problema que se va a resolver y la estructura de datos que se utilizará para representar el estado actual.

2. Representar el problema: Se debe representar el problema en términos de un árbol de decisiones, donde cada nodo representa un estado y cada rama representa una opción tomada. Cada nodo debe tener información sobre el estado actual y las decisiones tomadas hasta ese punto.

3. Establecer condiciones de terminación: Es importante definir condiciones de terminación que indiquen cuándo se ha alcanzado una solución o cuándo se debe retroceder. Esto podría ser la satisfacción de un conjunto de restricciones o la llegada a un estado final.

4. Explorar decisiones: El algoritmo comienza explorando decisiones en profundidad. Se elige una opción, y se avanza hacia adelante en el árbol de decisiones.

5. Verificar restricciones: Después de tomar una decisión, se deben verificar las restricciones para asegurarse de que el estado actual es válido. Si las restricciones no se cumplen, se retrocede y se prueba con otra opción.

6. Condición de terminación: Si se alcanzan las condiciones de terminación, se ha encontrado una solución. Si no, se continúa explorando opciones.

7. Retroceso (backtrack): Si se llega a un punto donde no hay más decisiones que tomar o se violan restricciones, el algoritmo retrocede al estado anterior y prueba una opción diferente.

8. Iteración: Se repiten los pasos 4 a 7 hasta encontrar una solución o explorar todas las posibilidades.

2.2. IMPLEMENTACIÓN BÁSICA:

Si bien, en el documento que teniamos como guia para desarrollar el proyecto final nos pedia que teniamos que hacer una verificacion en la correccion de la solucion mediante pruebas exhaustivas para corroborar el algoritmo.
Además de utilizar este algoritmo para la solución del problema de las n reynas.
Para resolver este problema lo podemos utilizar de esta manera:

1. **Definir el problema:**
   Colocar N reinas en un tablero NxN de manera que ninguna reina amenace a otra.

2. **Representar el problema:**
   Representa el estado actual con la posición de las reinas en el tablero.

3. **Condiciones de terminación:**
   Terminar cuando todas las reinas estén colocadas o cuando no sea posible colocar más reinas sin amenazarse mutuamente.

4. **Explorar decisiones:**
   Para cada fila del tablero, intenta colocar una reina en cada columna.

5. **Verificar restricciones:**
   Verifica si la posición actual de la reina es válida y no amenaza a ninguna otra reina en las filas anteriores.

6. **Condición de terminación:**
   Si todas las reinas están colocadas, se ha encontrado una solución. Si no, sigue explorando.

7. **Retroceso (backtrack):**
   Si la posición actual no es válida, retrocede y prueba la siguiente columna para la reina en la fila actual.

8. **Iteración:**
   Repite los pasos 4 a 7 hasta encontrar una solución o explorar todas las posibilidades.

2.3. Codigo(implementacion basica)

Código para la implementación basica de este algoritmo.

```
import time
import matplotlib.pyplot as plt

def no_valido(fila, col, reinas):
    for r in range(fila):
        if col == reinas[r] or abs(col - reinas[r]) == abs(fila - r):
            return False
    return True

def imprimir_tablero(reinas):
    for fila in range(len(reinas)):
        line = ""
        for col in range(len(reinas)):
            if col == reinas[fila]:
                line += "Q"
            else:
                line += ". "
        print(line)
    print("\n")

def lugar_reina(fila, reinas, n):
    if fila == n:
        imprimir_tablero(reinas)
        return 1
    else:
        sol_total = 0
        for col in range(n):
            if no_valido(fila, col, reinas):
                reinas[fila] = col
                sol_total += lugar_reina(fila+1, reinas, n)
        return sol_total

def n_reinas(n):
    reinas = [-1] * n
    fila = 0
    return lugar_reina(fila, reinas, n)

# Medir el tiempo de ejecucion
start_time = time.time()
soluciones = n_reinas(8)
end_time = time.time()

print("Numero total de soluciones:", soluciones)
print("Tiempo de ejecucion:", end_time - start_time, "segundos")

# Visualizacion grafica
plt.bar(["Soluciones", "Tiempo de ejecucion"], [soluciones, end_time -
    start_time])
plt.ylabel("Valor")
plt.title("Desempeno del algoritmo de N reinas")
plt.show()
```

2.4. ESTRATEGIA 1:

La poda alfa-beta es una técnica de optimización utilizada en algoritmos de búsqueda, y en este caso, se aplica al algoritmo de backtracking para resolver el problema de las N reinas. La poda alfa-beta permite reducir la cantidad de nodos explorados, eliminando la necesidad de explorar subárboles que no afectarán el resultado final. Esto se logra mediante la comparación de dos valores, alfa y beta, que representan los límites superior e inferior respectivamente.

Explicacion en el código:

. Inicialización de alpha y beta: En la llamada inicial, alpha se inicializa con el valor negativo de la máxima representación numérica (float('-inf')) y beta con la máxima representación numérica (float('inf')).

. Comparación con alpha y beta en el bucle for: Después de realizar la recursión en lugar-reina, se actualiza alpha tomando el máximo entre su valor actual y el negativo de sol-total. Si alpha es mayor o igual a beta, se rompe el bucle, ya que no es necesario seguir explorando esta rama, ya que no mejorará la solución actual.

2.5. Codigo(Estrategia 1:)

Código para la estrategia 1(poda)

```
import tkinter as tk
from tkinter import ttk
import time
import matplotlib.pyplot as plt

def no_valido(fila, col, reinas):
    for r in range(fila):
        if col == reinas[r] or abs(col - reinas[r]) == abs(fila - r):
            return False
    return True

def imprimir_tablero(reinas):
    for fila in range(len(reinas)):
        line = ""
        for col in range(len(reinas)):
            if col == reinas[fila]:
```

```
                line += "Q "
            else:
                line += ". "
        print(line)
    print("\n")

def lugar_reina(fila, reinas, n, alpha, beta):
    if fila == n:
        imprimir_tablero(reinas)
        return 1
    else:
        sol_total = 0
        for col in range(n):
            if no_valido(fila, col, reinas):
                reinas[fila] = col
                sol_total += lugar_reina(fila+1, reinas, n, -beta, -alpha)

                # Poda alfa-beta
                alpha = max(alpha, -sol_total)
                if alpha >= beta:
                    break
        return sol_total

class NReinasGUI:
    def __init__(self, root):
        self.root = root
        self.root.title("N-Reinas Solver")
        self.label = tk.Label(root, text="Cantidad de reinas:")
        self.label.pack()
        self.entry = tk.Entry(root)
        self.entry.pack()
        self.solve_button = tk.Button(root, text="Resolver",
            command=self.solve)
        self.solve_button.pack()
        self.canvas = tk.Canvas(root, width=400, height=400)
        self.canvas.pack()
        self.results_label = tk.Label(root, text="")
        self.results_label.pack()

    def solve(self):
        n = int(self.entry.get())
        reinas = [-1] * n
        fila = 0
        alpha, beta = float('-inf'), float('inf')
        start_time = time.time()
        soluciones = self.lugar_reina(fila, reinas, n, alpha, beta)
        end_time = time.time()

        print("Numero total de soluciones:", soluciones)
        print("Tiempo de ejecucion:", end_time - start_time, "segundos")
        self.results_label.config(text=f"Numero de soluciones:
            {soluciones}\nTiempo de ejecucion: {end_time - start_time:.4f}
            segundos")

        # Visualizacion grafica
        self.plot_results(soluciones, end_time - start_time)
```

```
70
71      def lugar_reina(self, fila, reinas, n, alpha, beta):
72          if fila == n:
73              self.imprimir_tablero(reinas)
74              return 1
75          else:
76              sol_total = 0
77              for col in range(n):
78                  if self.no_valido(fila, col, reinas):
79                      reinas[fila] = col
80                      sol_total += self.lugar_reina(fila+1, reinas, n, -beta,
                            -alpha)
81                      # Poda alfa-beta
82                      alpha = max(alpha, -sol_total)
83                      if alpha >= beta:
84                          break
85              return sol_total
86
87      def no_valido(self, fila, col, reinas):
88          for r in range(fila):
89              if col == reinas[r] or abs(col - reinas[r]) == abs(fila - r):
90                  return False
91          return True
92
93      def imprimir_tablero(self, reinas):
94          self.canvas.delete("all")
95          cell_size = 400 // len(reinas)
96          for fila in range(len(reinas)):
97              for col in range(len(reinas)):
98                  x1, y1 = col * cell_size, fila * cell_size
99                  x2, y2 = x1 + cell_size, y1 + cell_size
100                 if col == reinas[fila]:
101                     self.canvas.create_rectangle(x1, y1, x2, y2,
                            fill="lightgreen")
102                     self.canvas.create_text((x1 + x2) // 2, (y1 + y2) // 2,
                            text="Q", font=("Arial", 12, "bold"))
103                 else:
104                     self.canvas.create_rectangle(x1, y1, x2, y2,
                            fill="white", outline="black")
105
106     def plot_results(self, soluciones, tiempo_ejecucion):
107         plt.bar(["Soluciones", "Tiempo␣de␣ejecucion"], [soluciones,
                tiempo_ejecucion])
108         plt.ylabel("Valor")
109         plt.title("Desempeno␣del␣algoritmo␣de␣N␣reinas")
110         plt.show()
111
112 if __name__ == "__main__":
113     root = tk.Tk()
114     app = NReinasGUI(root)
115     root.mainloop()
```

2.6. ESTRATEGIA 2:

En este código se utiliza una heurística simple para inicializar la posición de las reinas en el tablero antes de aplicar el algoritmo de backtracking. La heurística consiste en colocar las reinas aleatoriamente en diferentes columnas.

Esta heurística aleatoria es un intento de proporcionar una buena configuración inicial para el algoritmo de backtracking, que puede mejorar la eficiencia y acelerar la convergencia hacia soluciones válidas.

2.7. Codigo(Estrategia 2:)

Código para la estrategia 2(heurística):

```
import tkinter as tk
from tkinter import messagebox, ttk
import time
import random
import matplotlib.pyplot as plt

def no_valido(fila, col, reinas):
    for r in range(fila):
        if col == reinas[r] or abs(col - reinas[r]) == abs(fila - r):
            return False
    return True

def imprimir_tablero_text(tablero_text, reinas):
    tablero_text.delete(1.0, tk.END)
    for fila in range(len(reinas)):
        line = ""
        for col in range(len(reinas)):
            if col == reinas[fila]:
                line += "Q␣"
            else:
                line += ".␣"
        line += "\n"
        tablero_text.insert(tk.END, line)

def lugar_reina(fila, reinas, n, tablero_text):
    if fila == n:
        imprimir_tablero_text(tablero_text, reinas)
        return 1
    else:
        sol_total = 0
        for col in range(n):
            if no_valido(fila, col, reinas):
                reinas[fila] = col
                sol_total += lugar_reina(fila+1, reinas, n, tablero_text)
        return sol_total
```

```
36
37 def inicializar_reinas(n):
38     return random.sample(range(n), n)
39
40 def n_reinas(n, tablero_text):
41     reinas = inicializar_reinas(n)
42     fila = 0
43     return lugar_reina(fila, reinas, n, tablero_text)
44
45 def resolver_y_mostrar_solucion(n, tablero_text, tiempo_label):
46     tablero_text.delete(1.0, tk.END)
47     try:
48         start_time = time.time()
49         soluciones = n_reinas(n, tablero_text)
50         end_time = time.time()
51
52         if soluciones == 0:
53             messagebox.showinfo("Solucion", "No hay soluciones para el
                   tablero de " + str(n) + " reinas.")
54         else:
55             messagebox.showinfo("Solucion", "Numero total de soluciones: "+
                   str(soluciones))
56
57         tiempo_ejecucion = end_time - start_time
58         tiempo_label.config(text=f"Tiempo de ejecucion {tiempo_ejecucion:.4f}
               segundos")
59
60         # Agregar visualizacion grafica del tiempo de ejecucion
61         plt.bar(["Soluciones", "Tiempo de
               ejecucion"],[soluciones,tiempo_ejecucion])
62         plt.ylabel("Valor")
63         plt.title("Desempeno del algoritmo de N reinas")
64         plt.show()
65
66     except Exception as e:
67         messagebox.showerror("Error", str(e))
68
69 def on_solve_button_click():
70     try:
71         n = int(entry.get())
72         if n <= 0:
73             raise ValueError("El numero de reinas debe ser un entero
                   positivo.")
74
75         # Crear una etiqueta para mostrar el tiempo de ejecucion
76         tiempo_label = tk.Label(root, text="")
77         tiempo_label.pack()
78
79         resolver_y_mostrar_solucion(n, tablero_text, tiempo_label)
80     except ValueError as e:
81         messagebox.showerror("Error", str(e))
82
83 # Crear la interfaz grafica
84 root = tk.Tk()
85 root.title("N-Reinas Solver")
86
```

```
label = tk.Label(root, text="Numero de reinas:")
label.pack()

entry = tk.Entry(root)
entry.pack()

solve_button = tk.Button(root,text="Resolver",command=on_solve_button_click)
solve_button.pack()

# Widget Text para mostrar el tablero
tablero_text = tk.Text(root, height=8, width=16)
tablero_text.pack()
root.mainloop()
```

3. ANÁLISIS DE COMPLEJIDAD

3.1. Simple

La complejidad del algoritmo de backtracking suele expresarse en términos de la notación O(1) para el peor caso. En el peor escenario, la complejidad puede ser exponencial, pero el uso efectivo de técnicas de poda puede mejorar significativamente el rendimiento.
Es importante tener en cuenta que la eficiencia de un algoritmo de backtracking también depende de la naturaleza específica del problema que estás resolviendo. En algunos casos, puede haber optimizaciones específicas que reduzcan la complejidad en situaciones prácticas.
Es importante tener en cuenta que esta es una simplificación y que la complejidad real puede variar dependiendo de varios factores, como la existencia de técnicas de poda (pruning) para evitar explorar ciertas ramas del árbol que no conducirán a soluciones válidas.
En resumen, la complejidad de tiempo para una implementación básica de backtracking se expresa en términos de la ramificación y la profundidad del árbol de recursión.

3.2. Poda

La complejidad del algoritmo de backtracking para el problema de "nreinas, con la estrategia de poda, puede expresarse en términos de ramificación y profundidad del árbol de recursión.
La ramificación se refiere al número de opciones disponibles para colocar una reina en una fila determinada y la profundidad del árbol es el número total de reinas que se deben colocar.
Con la estrategia de poda efectiva, la complejidad de tiempo para el problema de "n" reynas utilizando backtracking puede ser considerablemente menor que $o(n^2)$. Sin embargo, es difícil dar una fórmula exacta, ya que la eficacia de las podas puede variar según la implementación y las condiciones específicas del problema.

3.3. Heurística

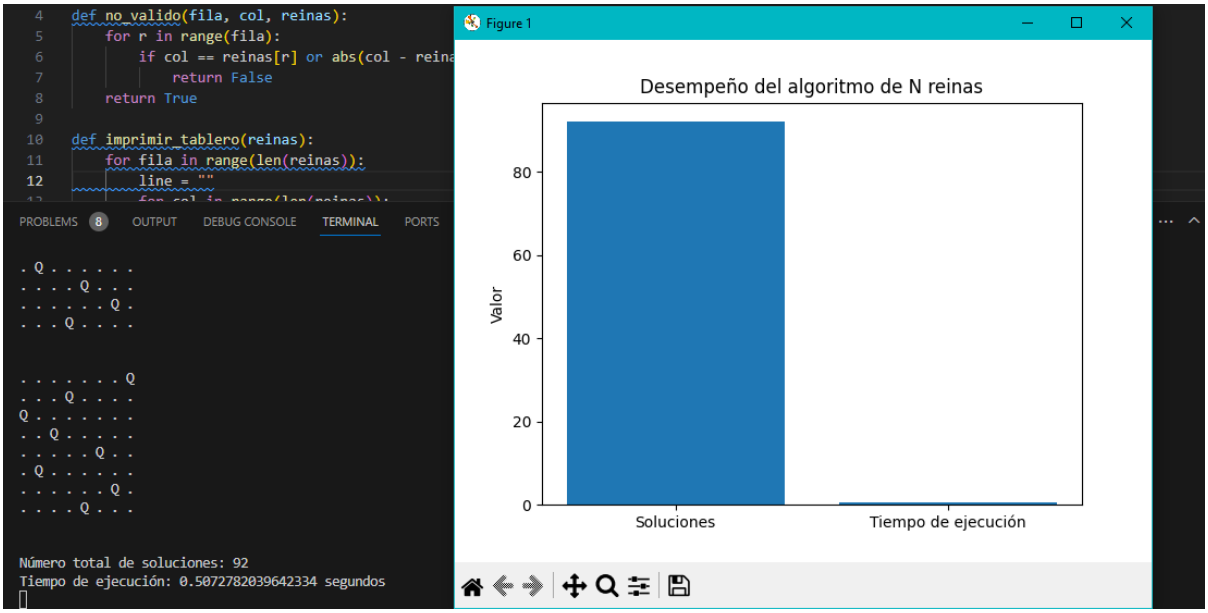
La complejidad de tiempo con heurísticas es difícil de expresar de manera general, ya que depende de la eficacia de las heurísticas específicas utilizadas y cómo afectan la búsqueda en el espacio de soluciones. En el mejor de los casos, las heurísticas bien diseñadas pueden reducir significativamente la complejidad en la práctica, pero en el peor caso, la complejidad seguirá siendo exponencial.
En este caso usamos una heuristica aleatoria, si consideramos el algoritmo de backtracking con una heurística aleatoria para el problema de las "nreinas, podríamos tener una complejidad de aproximadamente tendría una complejidad exponencial O(n!), ya que podría explorar todas las permutaciones posibles.

4. COMPARACIÓN DE RENDIMIENTO CON GRÁFICAS

En este apartado, presentamos algunos ejemplos de las tres implementaciones solicitadas, para asi poder analizar el rendimiento de cada una.

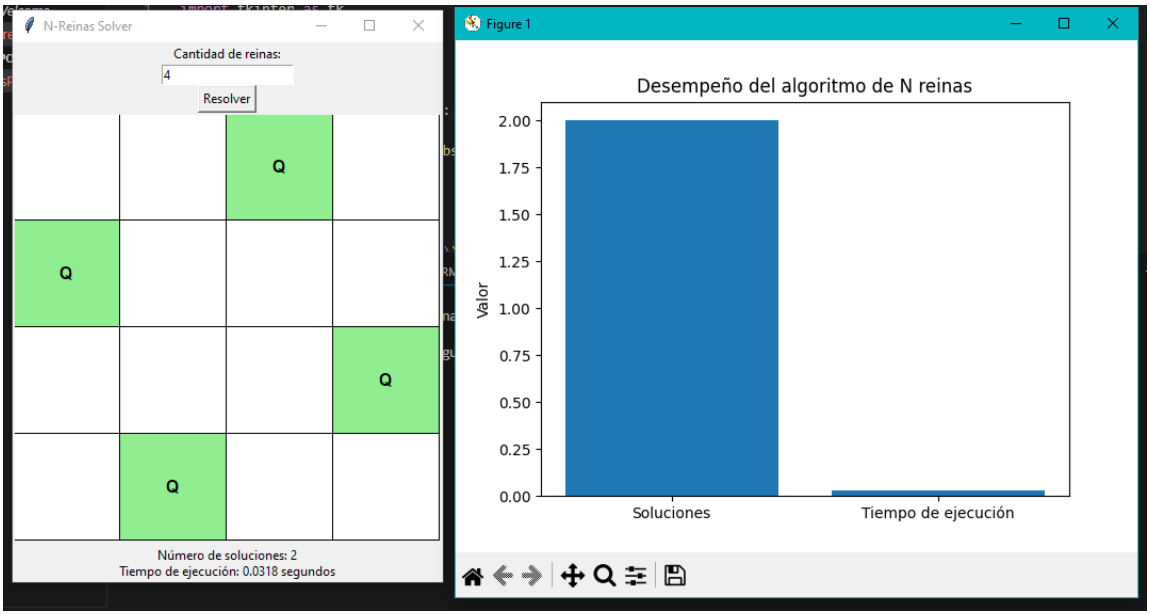
Ejemplo 1.

Este ejemplo representa, la primera implementación del algoritmo de backtracking para las "nreynas.



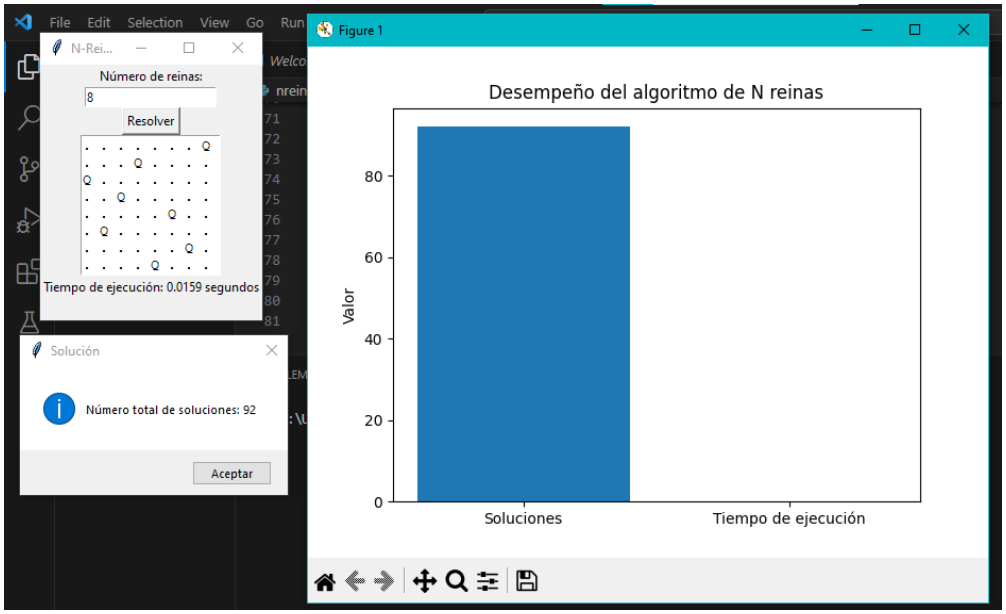
Ejemplo 2.

Este ejemplo representa, la segunda implementación del algoritmo de backtracking utilizando la estrategia de "poda"para el problema de las "nreynas.



Ejemplo 3.

Este ejemplo representa, la tercera implementación del algoritmo de backtracking utilizando "heuristicas", en este caso una heuristica aleatoria para el problema de las "nreynas.



5. CONCLUSIÓN

Para concluir este trabajo, solo queda agregar que el algoritmo de backtracking era un algoritmo que nunca habíamos conocido ni trabajado, o al menos eso pensábamos, al ponernos a investigar acerca de este mismo, nos dimos cuenta de que ya podríamos haber trabajado levemente con él.

Al resolver el problema de las "N" reynas, pudimos entender que es algo interesante y a la vez no tan complicado de entender. Fue muy gratificante ver como los tres códigos, buscaban las posibles soluciones tratando de que fueran en un tiempo considerable para que se ejecutara de manera correcta.
En resumen, aunque encontrar una solución exacta para el problema de las N reinas nos costo un poco, pero nos dimos cuenta que existen estrategias eficientes para abordar instancias prácticas del problema, aprovechando técnicas heurísticas y algoritmos especializados.

En resumen, el aprendizaje del problema de las N reinas y el algoritmo de backtracking va más allá de la resolución específica del problema, proporcionando una base sólida para comprender la complejidad computacional, desarrollar estrategias de resolución eficientes y aplicar esos conocimientos a problemas del mundo real.

6. REFERENCIAS

https://programacionyalgoritmia.com/fundamentos-de-programacion/backtracking/#:~:text=Pasos%20del%20algoritmo%20de%20Backtracking%201%20Elige%20una,pasos%20hasta%20explorar%20completamente%20el%20espacio%20de%20b%C3%BAsqueda.
https://docs.jjpeleato.com/algoritmia/backtracking
https://programacionyalgoritmia.com/fundamentos-de-programacion/backtracking/
https://www.academia.edu/28323442/Algoritmos_Gen%C3%A9ticos_Aplicaci%C3%B3n_al_Juego_de_las_N_Reinas
https://sisbib.unmsm.edu.pe/bibvirtualdata/monografias/basic/riojas_ca/cap4.pdf