INSTITUTO POLITÉCNICO NACIONAL

UNIDAD PROFESIONAL INTERDISCIPLINARIA DE INGENIERIA CAMPUS ZACATECAS

Análisis y Diseño de Algoritmos

REPORTE

Algoritmo de Backtracking para el Problema de las N Reinas

DOCENTE

M. en C. Erika Sánchez-Femat

POR EL ALUMNO

Cristian García Nieves

GRUPO

3CM1

FECHA

Miércoles 17 de Enero del 2024

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

Implementación Básica

Desarrollar una implementación inicial del algoritmo de backtracking para resolver el problema de las N Reinas en Python.

reinas.py > ...

```python
#ORIGINAL

def is_valid(row,cols,reinas):
    for r in range(row):
        if cols ==  reinas[r]:
            return False
        elif abs(cols-reinas[r]) == abs(row - r):
            return False

    return True

def place_reinas(row,reinas,n):
    if row == n:
        print(reinas)
        return 1
    else:
        total_sols = 0
        for cols in range(n):
            if is_valid(row,cols,reinas):
                reinas[row] = cols
                total_sols += place_reinas(row+1,reinas,n)
        return total_sols

def n_reinas(n):
    reinas = [""]*n
    row = 0
    return place_reinas(row,reinas,n)


```

Figura 1: N-Reinas básico

Este algoritmo utiliza un enfoque de backtracking para probar todas las posibles configuraciones de reinas en un tablero de ajedrez de NxN, evitando ubicaciones donde las reinas se atacarían mutuamente

Estrategia 1

Aplicar una técnica de poda que reduzca la cantidad de nodos explorados durante la búsqueda, mejorando así el tiempo de ejecución

```python
#ESTRATEGIA 1: Técnica de poda

def is_valid(row,cols,reinas):
    for r in range(row):
        if cols ==  reinas[r] or abs(cols-reinas[r]) == abs(row - r):
            return False

    return True

def place_reinas(row,reinas,n):
    if row == n:
        print(reinas)
        return 1
    else:
        total_sols = 0
        for cols in range(n):
            if is_valid(row,cols,reinas):
                reinas[row] = cols
                total_sols += place_reinas(row+1,reinas,n)
        return total_sols

def n_reinas_poda(n):
    reinas = [""]*n
    row = 0
    return place_reinas(row,reinas,n)
```

Figura 2: Técnica de poda

Ésta técnica de poda se utiliza para evitar explorar ciertas ramas del árbol de decisiones cuando se determina que no conducirán a una solución válida.

Estrategia 2

Aplicar heurísticas inteligentes que guíen la colocación inicial de las reinas para acelerar la convergencia a soluciones válidas

```python
#ESTRATEGIA 2: Heuristicas

def is_valid(row, col, reinas):
    for r in range(row):
        if col ==  reinas[r] or abs(col-reinas[r]) == abs(row - r): # en esta condición implementamos ambos
            return False
    return True

def get_remaining_values(row, reinas, n):
    valid_cols = [col for col in range(n) if is_valid(row, col, reinas)]
    return sorted(valid_cols, key=lambda x: sum(1 for r in range(row + 1, n) if is_valid(r, x, reinas)))

def place_reinas(row, reinas, n):
    if row == n:
        print(reinas)
        return 1
    else:
        total_sols = 0
        remaining_values = get_remaining_values(row, reinas, n)
        for col in remaining_values:
            reinas[row] = col
            total_sols += place_reinas(row + 1, reinas, n)
        return total_sols

def n_reinas_heuristica(n):
    reinas = [0]*n
    row = 0
    return place_reinas(row,reinas,n)
```

Figura 3: Heuristicas

Ésta estrategia implementa heurísticas para mejorar la eficiencia en la búsqueda de soluciones al problema de las N reinas donde se introduce una nueva función que devuelve las columnas válidas restantes para colocar una reina en una fila específica.

Análisis de complejidad

Estrategia 1: Técnica de Poda

- **Complejidad de Tiempo:** $O(n!)$ en el peor caso (sin poda sería $O(n!)$) debido al enfoque de "backtracking". La poda reduce significativamente la cantidad de configuraciones exploradas en comparación con el enfoque de fuerza bruta.
- **Complejidad de Espacio:** $O(n)$, utilizando un arreglo para el estado del tablero.

Estrategia 2: Heurísticas

- **Complejidad de Tiempo:** Depende de la efectividad de las heurísticas aplicadas. En el peor caso, sigue siendo exponencial, pero las heurísticas buscan mejorar la eficiencia práctica.
- **Complejidad de Espacio:** $O(n)$, utilizando un arreglo para el estado del tablero.

Estrategia Básica (Sin Poda ni Heurísticas)

- **Complejidad de Tiempo:** $O(n!)$ en el peor caso, ya que explora todas las configuraciones posibles sin optimizaciones.
- **Complejidad de Espacio:** $O(n)$, utilizando un arreglo para el estado del tablero.

Comparación:

- La Estrategia 1 es más eficiente que la básica debido a la poda.
- La Estrategia 2 busca mejoras prácticas mediante heurísticas.
- La versión básica es la menos eficiente, explorando todas las configuraciones sin optimizaciones.

Documentación Técnica

Para la implementación básica para resolver el problema de las N-Reinas se realizó de la siguiente manera:

Es necesario mencionar que este algoritmo no trabaja con un tablero de nxn, si no que trabaja principalmente con un arreglo donde cada posición de l arreglo representa cada columna del tablero, y el número que esté en cada posición del arreglo representa en que posición de la columna se encuentra la reina.

```
def n_reinas(n):
    reinas = [""]*n
    row = 0
    return place_reinas(row,reinas,n)

print(n_reinas(4))
```

Figura 4: Principal función para el algoritmo

En esta párte de la definición de la funcion n_reinas donde como argumento se pasa un entero que servirá para crear el arreglo del tamaño del tablero y se crea una variable row = 0 que servirá para ir recorriendo el arreglo y finalmente mandamos estas 3 variables a la funcion Place Reinas.

```
def place_reinas(row,reinas,n):
    if row == n:
        print(reinas)
        return 1
    else:
        total_sols = 0
        for cols in range(n):
            # print(reinas)
             if is_valid(row,cols,reinas):
                 reinas[row] = cols
                 total_sols += place_reinas(row+1,reinas,n)
        return total_sols
```

Figura 5: Place reinas

En esta función se realiza principalmente la lógica donde se crea un ciclo for que sirve para iterar hasta el tamaño de N para así ir probando y llenando el tablero, sin embargo no solo es llenar el tablero y ya, si no que es necesario aplicar una función para saber si la posición en la que se quiere colocar una reina es válida o no.

```python
def is_valid(row,cols,reinas):
    for r in range(row):
        if cols ==  reinas[r]:
            return False
        elif abs(cols-reinas[r]) == abs(row - r):
            return False
    return True
```

Figura 6: funcion para validar posición

Ésta función se encarga de determinar si la posición en la que se quiere asignar la reina es válida. Coinsiste en ver si la columna en la que se quiere colocar ya hay otra reina colocada y la otra en que si se quiere poner en diagonal de otra reina, con cualquiera de estos 2 casos que se presenten la solución ya no será válida.

```python
def place_reinas(row,reinas,n):
    if row == n:
        print(reinas)
        return 1
    else:
        total_sols = 0
        for cols in range(n):
            # print(reinas)
            if is_valid(row,cols,reinas):
                reinas[row] = cols
                total_sols += place_reinas(row+1,reinas,n)
        return total_sols
```

Figura 7: Función para asignar reinas.

De esta manera se repetirá recursivamente la función de place-Reians hasta encontar una solución e imprimir el resultado para continuar recorriendo las distintas posibilidades y así ir descubriendo nuevas soluciones y si no es solucion se regresa al paso anterior y prueba con otra posición.

Ejemplo para demostrar el funcionamiento

Para ir viendo el funcionamiento del algoritmo lo demostraremos con un arreglo tamaño 4.

```
print(n_reinas(4))
```

Figura 8: Llamada de función n-reinas

Al mandar a llamar la funcion y establecer el tamaño del arreglo en 4 vemos que se crea el siguiente arreglo:

```
['', '', '', '']
```

Figura 9: Arreglo vacio

Después se coloca la primera reina en la primera posición del tablero, y al estar vació el tablero esa posición si será válida

```
['', '', '', '']
Posición: 0  fué válida
[0, '', '', '']
Posición: 0  no fué válida
[0, '', '', '']
Posición: 1  no fué válida
[0, '', '', '']
Posición: 2  fué válida
[0, 2, '', '']
```

Figura 10: Iteraciones del algoritmo 1.1

Después de colocar la primera reina, el algoritmo se pasa a la siguiente columna e intenta asignar otra reina en la primera fila de la 2da columna y pues esto no es válido, luego intenta con la 2da fila y tampoco fué válido hasta la 3era fila ya es válido, en el caso del arreglo al empezar desde el 0 es válido hasta la fila 2.

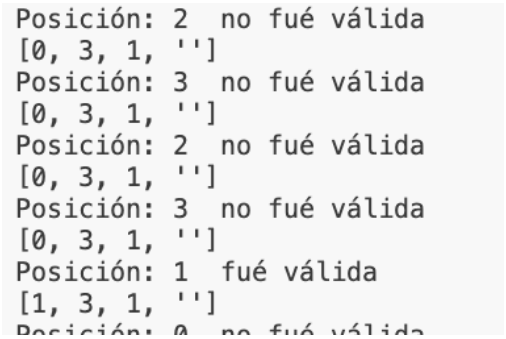

Figura 11: Iteraciones del algoritmo 1.2

Aqui observamos que después de intentar llenar el tablero con la primera reina en la fila y en la columna 0 no fue posible resolverlo, por lo que el se regresa el algoritmo y ahora intenta con la primera reina en la fila 1 en la columna 0.

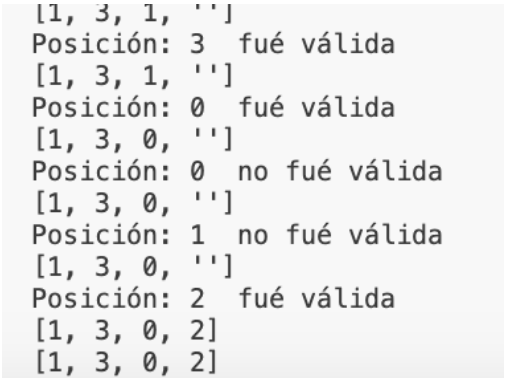

Figura 12: Iteraciones del algoritmo con 1era solución.

Despues de varios intentos el algoritmo encuentra la primera solución para después continuar y seguir buscando otras soluciones.

3. Comparación de rendimientos

Para implementar mediciones de tiempo de ejecución para la versión básica y las versiones optimizadas del algoritmo. Se crearon gráficas que visualicen el tiempo de ejecución en función del tamaño del problema (N) para cada estrategia de optimización. Analizar las gráficas para identificar tendencias y mejoras de rendimiento.

Comenzando con mandar a llamar a cada una de las funciones y teniendo unos medidores de tiempo para después obtener el tiempo que tardó cada algoritmo y después graficarlos

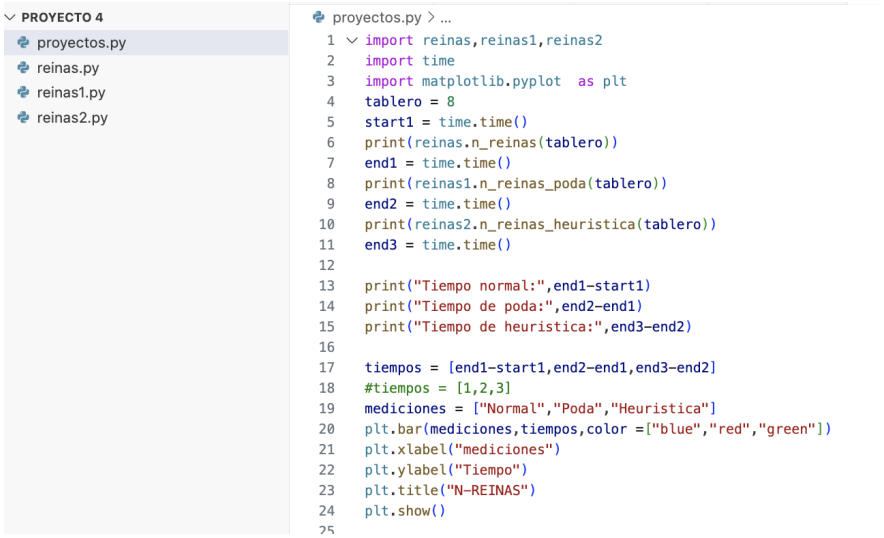

Figura 13: Implementación de todos los códigos.

Después de ejecutar el código vemos la siguiente gráfica.

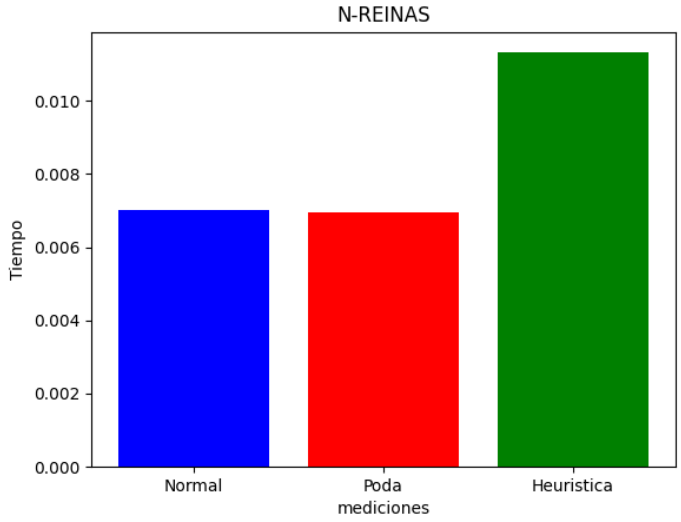

Figura 14: Implementación de todos los códigos con n = 8

Con esto observamos que el que mejor tiene rendimiento es el de la técnica de poda

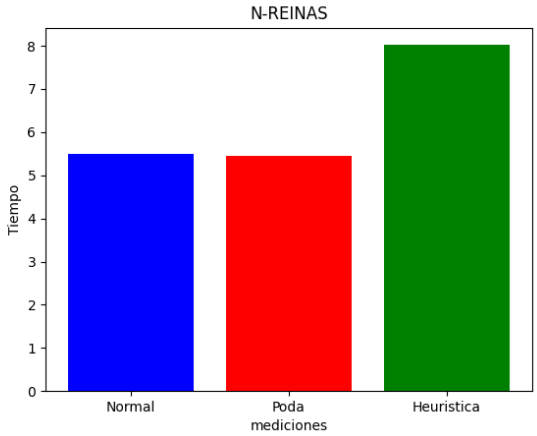

Figura 15: Implementación de todos los códigos con n = 12

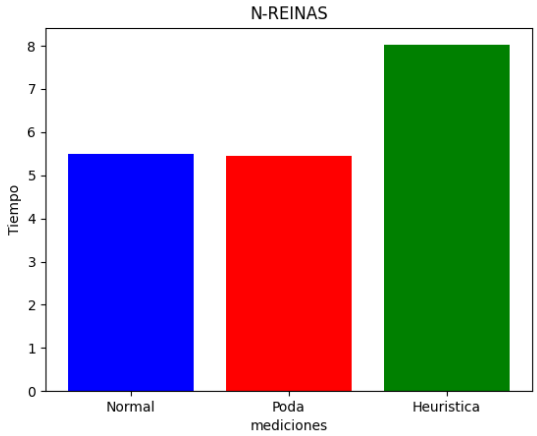

Figura 16: Implementación de todos los códigos con n = 4

4. Resultados esperados

En estas técnicas implementadas esperaba un mejor rendimiento con la técnica de heurística implementada para guiar la colocación de cada reina, sin embargo, es posible que tenga un peor rendimiento debido a la lógica que tiene que desarrollar para saber las posiciónes disponibles válidas. Sin embargo el ligero cambio para la estrategia 1 de la ténica de poda muestra un pequeño mejor desempeño a la hora de ejecutar el algorimto.