Instituto Politécnico Nacional

Ingeniería en Sistemas Computacionales

Optimización del algoritmo de Backtracking para el problema de las N reinas

Análisis y diseño de algoritmos

Maestra:
Erika Sanchez-Femat
Autor:
Jose Martin Cortes Lozano
3CM1

8 de Enero del 2024

Introducción

En este trabajo se implementa el algoritmo que resuelve el problema de las N reinas, donde se programa mediante el Backtracking y se hacen dos implementaciones más donde se optimiza el algoritmo, además se evalúa su tiempo de ejecución para cada implementación, igualmente se evalúa la complejidad Big (O) del algoritmo.
En el juego de ajedrez la reina amenaza a aquellas piezas que se encuentren en su misma fila, columna o diagonal. El problema de las N reinas consiste en poner sobre un tablero de ajedrez N reinas sin que estas se amenacen entre ellas.

Desarrollo

Primero se empezó investigando conceptos importantes que van a usarse a la hora de hacer los códigos, uno de ellos es sobre el Backtracking.
En el Backtracking, se tiene que hacer un recorrido de profundidad a un árbol, y el objetivo del recorrido es encontrar soluciones para algún problema. A medida que progresa el recorrido, conseguirá soluciones parciales para el problema; estas soluciones parciales limitan donde se puede encontrar o no encontrar una solución completa. El recorrido tiene éxito si se llega a encontrar una solución completa. Si solo busca una solución, se detendrá el algoritmo, en caso de que quiera encontrar todas las posibles soluciones, seguirá buscando hasta que encuentre todas. El algoritmo no tendrá éxito si en alguna parte de las soluciones parciales esta incompleta. Para este caso, el recorrido vuelve atrás en el recorrido de profundidad y elimina las opciones no viables y si hay nodos sin explorar, seguirá buscando una solución.
Otro concepto importante es la heurística inteligente, que consiste en un método de resolución de problemas que utilizan reglas o principios generales para encontrar soluciones aproximadas o subóptimas. Estos algoritmos se basan en el razonamiento aproximado, la experiencia y el juicio para guiar el proceso de búsqueda y llegar a una solución satisfactoria en un tiempo razonable, en otras palabras, son algoritmos que toman decisiones y con esas decisiones obtienen experiencia para conseguir una solución aproximada y óptima.
De igual manera se importaron las librerías mostradas en la figura 2, que serán usadas a la hora de mostrar los resultados.

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import time
```

Figura 2: Librerías usadas

Backtracking

Para este código, primero se hizo una función que verifica si es valido poner una reina en x posición del tablero. Con esta función se verifica si hay alguna reina en la misma columna, diagonal principal o fila. Si no se cumple alguna estas condiciones, se devuelve un False que indica que esa posición no es valida, del otro lado, si devuelve un True, significa que la posición es segura. Esto se muestra en a figura 3.

```
def valido(fila,colum,tablero):
    for i in range(fila):
        if(tablero[i]==colum or (tablero[i]-i)==(colum-fila) or (tablero[i]+i)==(colum+fila)):
            return False
    return True
```

Figura 3: Función "valido"

En la figura 4,muestra la siguiente función, es la que calcula cada solución posible, de esta manera la función se llamó "solución". La primera parte de la función analiza si se hay una reina colocada en cada

fila, entonces la solución es valida y se agrega una copia del tablero actual en una lista de todas las soluciones y retorna.

```
def solucion(n,fila,tablero,soluciones):
    if (fila==n):
        soluciones.append(tablero[:])
        return
```

Figura 4: Primer iteración de la solución

La siguiente parte de la función hace una iteración a través de las columnas, de este modo , itera sobre cada columna en la fila actual, después en el if se llama la función "valido"para validar la colocación de una reina en esta posición, si es valido se realiza la asignación de la reina y se llama de manera recursiva a "solución", finalmente se realiza una vuelta atrás, deshaciendo la ultima decisión y probando con la siguiente columna, como se observa en la figura 5.

```
for columna in range(n):
    if valido(fila,columna,tablero):
        tablero[fila]=columna
        solucion(n,fila+1,tablero,soluciones)
        tablero[fila]=-1
```

Figura 5: Segunda iteración de la solución

La siguiente función crea el tablero de las n reinas inicializando las todas las posiciones posibles con -1 e invoca a "solución"para que empiece la búsqueda de soluciones y guardarlas en una lista, después retorna todas las soluciones que encontró, como se ve en la figura 6.

```
def crear_tablero(n):
    soluciones=[]
    tablero=[-1]*n
    solucion(n,0,tablero,soluciones)
    return soluciones
```

Figura 6: Tablero del Backtracking

La ultima función, en la figura 7, imprime el tablero con todas las soluciones posibles para que se muestren en pantalla, donde "R"son las reinas y "° "son los espacios vacíos.

```
def imprimir(tablero):
    x=len(tablero)
    for i in tablero:
        print(" ".join(" R " if j==i else " ° " for j in range(x)))
```

Figura 7: Función que imprime el tablero

Poda del Árbol

Para esta optimización se usó la técnica de poda llamada "Forward Checking", que es una técnica que utiliza restricciones y reduce el dominio de las variables, para así reducir el espacio de búsqueda al descartar ciertos valores para las variables que no son asignadas. En el caso de las n reinas, marca como

no disponibles las columnas que ya han sido utilizadas en las filas anteriores, esto para evitar explorar ramas del árbol que no llevan a ninguna solución valida.
En este código también se utiliza una función que valida la posición de la reina, por lo tanto, se usó la misma que la del Backtracking. Del mismo modo, para imprimir la solución se usó la función "imprimir" del Backtracking.
En la figura 8 se ve que para la creación del tablero se inicializó cada posición con -1 e invoca a la función "optim2" para que busque una solución y la retorne.

```
def tablero_optim2(n):
    tabla=[-1]*n
    return optim2(n,0,tabla)
```

Figura 8: Tablero de la primera optimización

La función "optim2 ", en la figura 9, es la que usa la técnica de poda, primero e igual que en el anterior, verifica con un if si hay alguna reina en cada fila y si se cumple la condición, retorna la solución de manera de arreglo llamada tablero.

```
def optim2(n,fila,tablero):
    if(fila==n):
        return [fila for fila in tablero]
```

Figura 9: verificación en la poda del árbol

Luego se inicializa la solución como None, en seguida se itera sobre cada columna en la fila actual, llama a la función "valido", y lo pone en un if, si es valido colocar la reina en esa posición, se coloca y se hace una llamada recursiva para la siguiente fila en una variable que es un resultado parcial, luego si ese resultado parcial no está vacío, se retorna ese resultado, si no, se deshace la ultima decisión y se pasa a la siguiente columna. Finalmente, se retorna la solución que puede ser vacía en caso de que no haya ninguna solución, como se observa en la figura 10.

```
solu=None
for columna in range(n):
    if valido(fila,columna,tablero):
        tablero[fila]=columna
        r_parcial=optim2(n,fila+1,tablero)
        if(r_parcial is not None):
            return r_parcial
        tablero[fila]=-1
return solu
```

Figura 10: Poda del árbol para encontrar la solución

Heurísticas

Para las heurísticas, se implementó la técnica de Hill Climbing, que es comúnmente utilizada en la inteligencia artificial, donde esta técnica busca una solución los suficientemente buena para el problema, aunque esta solución no puede ser el óptimo global. Este es un algoritmo de generación y prueba, ya que hace una retroalimentación del procedimiento de prueba. Luego utiliza esta retroalimentación para decidir el próximo movimiento. Para el caso de las n reinas, se busca mejorar iterativamente la solución inicial mediante la exploración de vecinos y escoger aquellos que reduzcan los conflictos.

En este código se importa la librería randomz se usa al momento de crear el tablero, donde cada reina se coloca de manera aleatoria de su respectiva fila, como se muestra en la figura 11.

```
def tableroptim3(n):
    return [random.randint(0,n-1) for i in range(n)]
```

Figura 11: Tablero de la heurística

Luego, en la figura 12, se hizo una función que calcula los conflictos, donde con los ciclos se verifica que no haya dos reinas en la misma columna o en la misma diagonal.

```
def conflicto(tab):
    m=len(tab)
    conflictos=0
    for i in range(m):
        for j in range(i+1,m):
            if tab[i]==tab[j] or abs(tab[i]-tab[j])==abs(i-j):
                conflictos += 1
    return conflictos
```

Figura 12: Función que calcula los conflictos

En seguida, se hizo el algoritmo de hill climbing, donde:
Se declara una variable llamada max y se iguala a 1000, se crea un tablero aleatorio inicial con las n reinas y cada reina se coloca en una columna aleatoria de su fila, después se calcula la cantidad de conflictos en el tablero creado anteriormente, figura 13.

```
def hill_climbing(n):
    max=1000
    m_tab=tableroptim3(n)
    m_con=conflicto(m_tab)
```

Figura 13: Declaración de variables

Luego se inicia una iteración desde i hasta max o hasta que encuentre una solución. En seguida, se crea un vecino con la función para crear tablero para generar un vecino aleatorio cambiando algunas de las reinas de posición en el tablero actual. Luego se calculan los conflictos en el tablero de vecinos, figura 14.

```
for i in range(max):
    vecino = tableroptim3(n)
    con_vecino = conflicto(vecino)
```

Figura 14: Iteración principal

Después se verifica si los conflictos del vecino son menores a los conflictos del tablero inicial, se actualiza el tablero inicial con vecino y se actualiza la cantidad de conflictos, luego si los conflictos actuales son igual a 0 entonces se sale de la iteración y finalmente retorna la solución que encontró, figura 15.

```
        if (con_vecino < m_con):
            m_tab=vecino
            m_con=con_vecino
        if (m_con==0):
            break
    return m_tab
```

Figura 15: Calculo de la solución dentro del ciclo

Por ultimo, se hizo una nueva función para imprimir y mostrar en pantalla la solución encontrada por el hill climbing, como se observa en la figura 16.

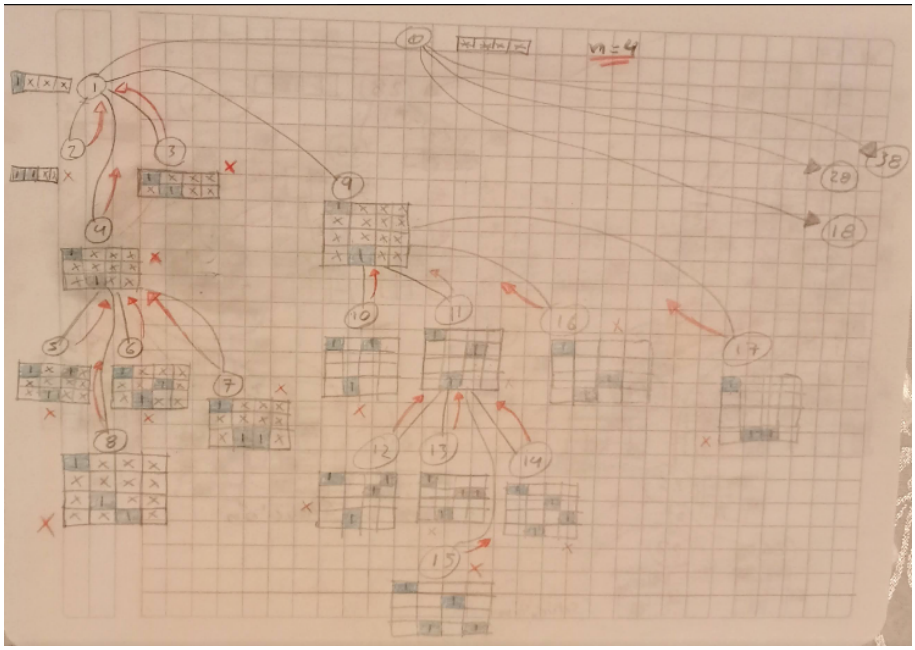
```
def imp_optim3(tabl):
    m=len(tabl)
    for i in range(n):
        print(" ".join(" R " if j == tabl[i] else " ° " for j in range(m)))
```

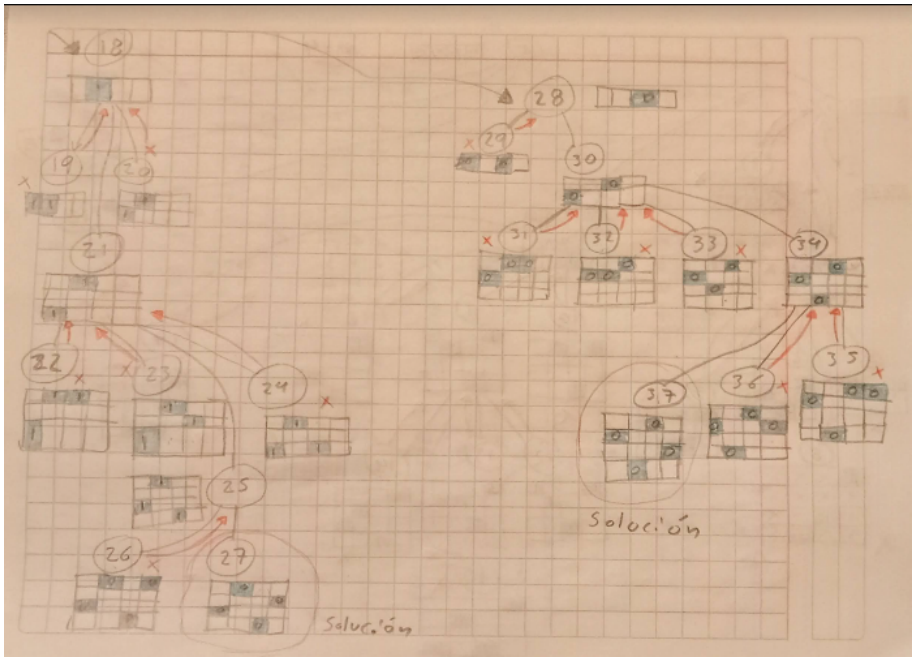
Figura 16: Imprime la solución encontrada

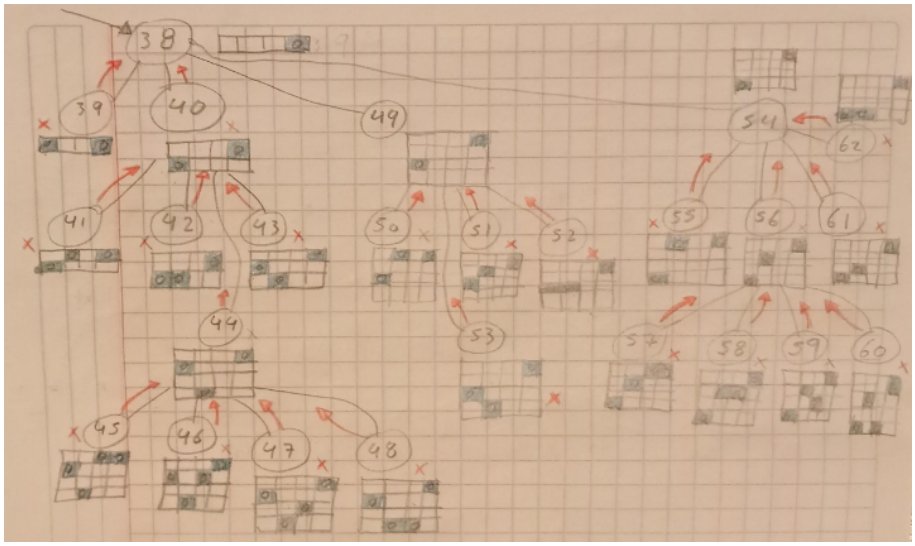
Resultados

Complejidad Big O

La complejidad Big O del algoritmo usando el Backtracking sin alguna optimización es del tipo $O(N^N)$, ya que explora todas las posibles combinaciones que hay en el tablero, esto hace que mientras n sea un numero pequeño, el tiempo sera aceptable, pero mientras la n sea cada vez mayor, esto provoca que la búsqueda de la solución se vuelva lenta, porque el tiempo de ejecución aumenta tanto como sea el tamaño de n.
Las siguientes figuras muestran el calculo de la solución de forma manual con Backtracking para el caso de 4 reinas, donde se haya la complejidad Big O. Para este calculo los primeros nodos se presentan en modo de arreglo, de este modo, la posición [0] es la primera en ser analizada.




18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
Solución
Solución


38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62

4 Reinas

Para los resultados, primero se hizo cuando n vale 4, que es el caso mas sencillo.
En el Backtracking, se muestran todas las posibles soluciones, por lo tanto, para el problema de 4 reinas solo hay dos soluciones, como se muestra en la figura 17.

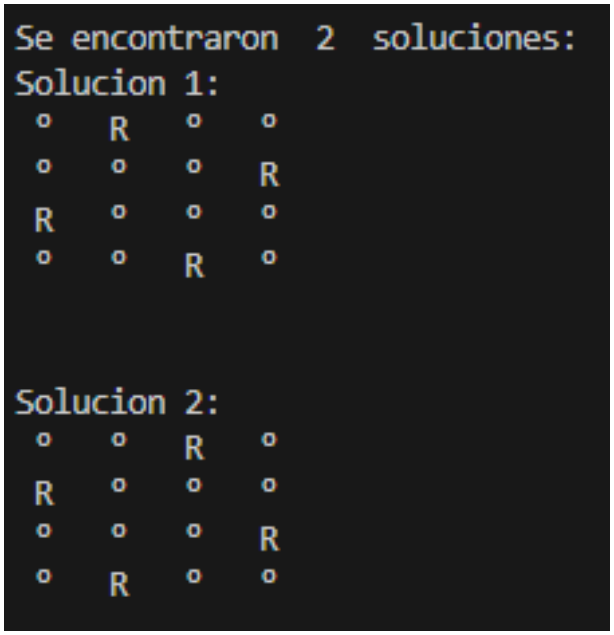


Figura 17: Solución Backtracking

En la poda del árbol, solo se muestra la solución que se encontró después de descartar las columnas que no llegan a una solución, por lo que en la figura 18 se muestra la solución a la que llegó.

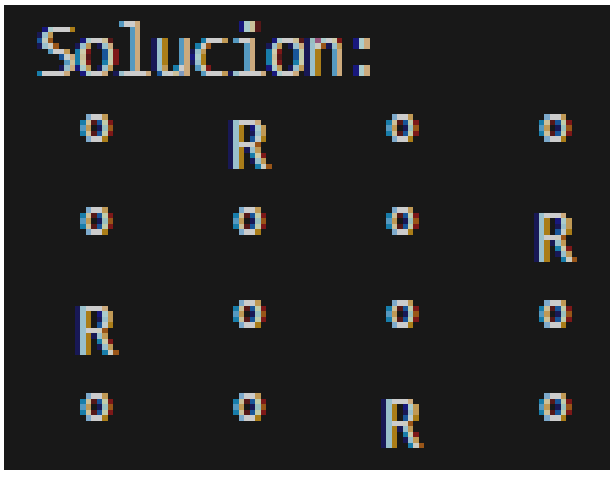


Figura 18: Solución con poda del árbol

Con las heurísticas, se llegó a la misma solución que con la de poda, pero la diferencia radica en el tiempo de ejecución, en la figura 19 se muestra la solución encontrada.

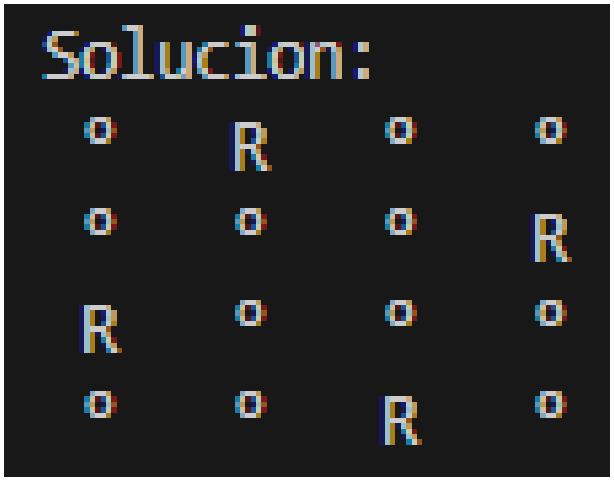


Figura 19: Solución con heurísticas

En el calculo de los tiempos, lo que se hizo fue medir el tiempo de inicio desde que se invoca el método que soluciona las n reinas y el tiempo de terminación que es hasta que termina de imprimir la solución(es) encontradas, enseguida de eso, se graficó los tiempos obtenidos en una gráfica de barras, como se muestra en la figura 20.

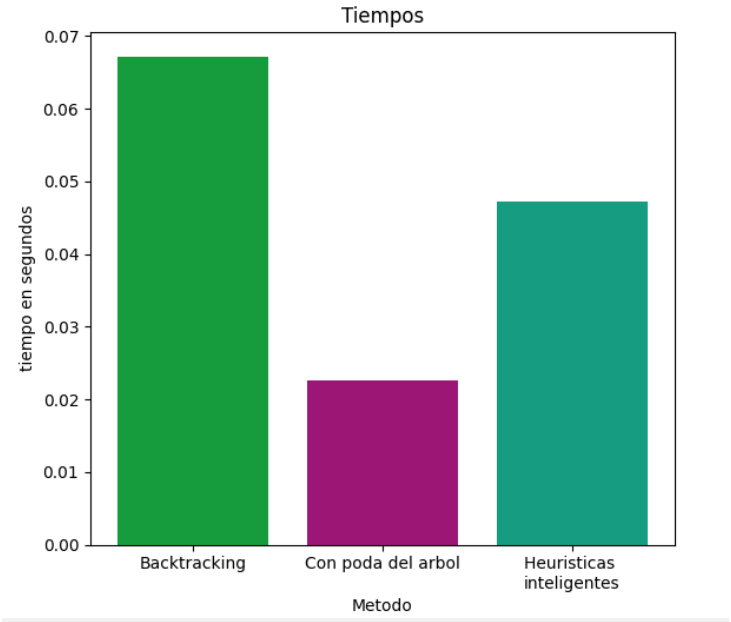


Figura 20: Gráfica para 4 reinas

En la figura anterior se observa que el tiempo de ejecución de la poda es el mejor de los 3, por lo que lo convierte en la opción mas viable, mientras que para el Backtracking, el tiempo de ejecución es mayor, lo que lo deja como la opción menos viable.

7 Reinas

Enseguida de eso, se probó para n igual a 7, donde los resultados conseguidos son los siguientes.
Para el Backtracking, en la figura 21 solo se muestra la solución 40, que fue la ultima solución obtenida, esto para ahorrar espacio.

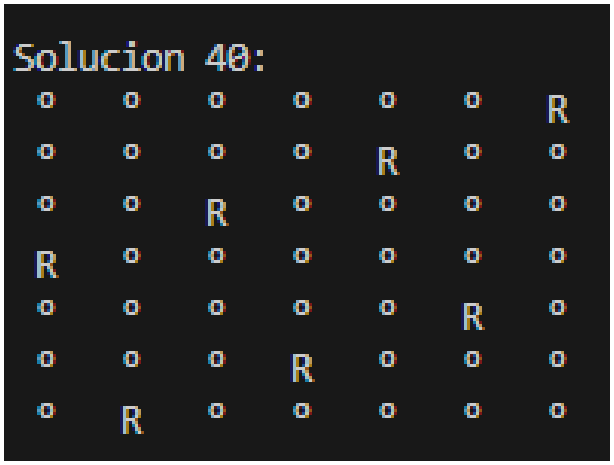


Figura 21: solución 40 del Backtracking

En la figura 22, se muestra la solución obtenido por el método de la poda.

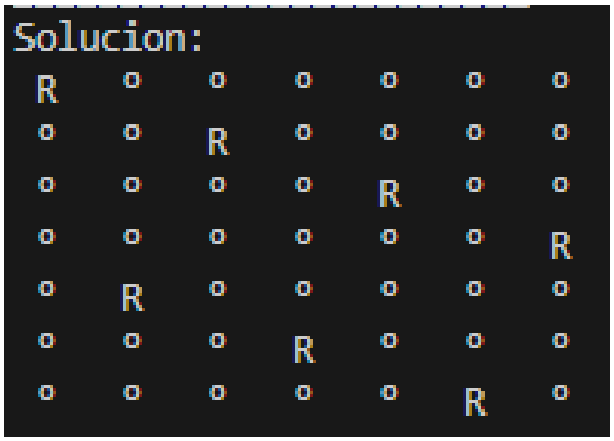


Figura 22: Solución 7 reinas con poda

En la heurística, también se encontró solo una solución pero esta vez es diferente a la solución de la poda, e igual que en el anterior, hay una diferencia notable en el tiempo de ejecución. En la figura 23 se muestra la solución obtenida.

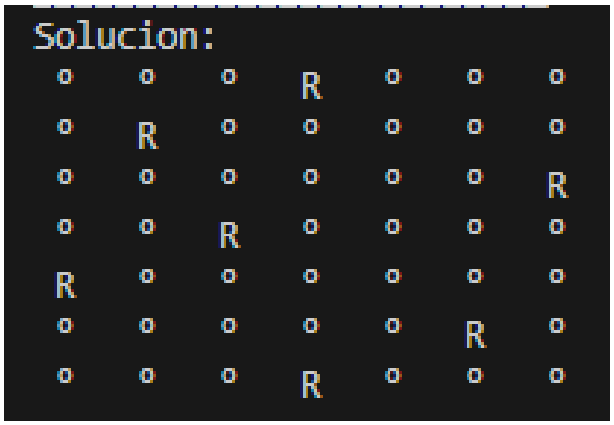


Figura 23: Solución 7 reinas con heurísticas

Los tiempos se calcularon de la misma manera explicada anteriormente, por tanto, la figura 24 muestra

la gráfica obtenida.

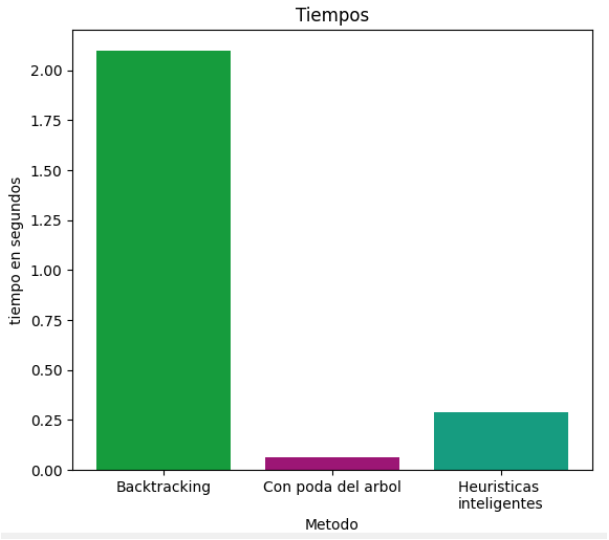


Figura 24: Gráfica de 7 reinas

En dicha gráfica se observa que la diferencia de tiempos entre la poda y la heurística empieza a hacerse mayor, por lo que la técnica de poda sigue siendo la mas viable. Para el caso de la peor opción, se observa que la diferencia de tiempos entre el Backtracking y la poda es demasiado, por lo tanto, el Backtracking sigue siendo la peor opción.

9 Reinas

La ultima prueba es cuando n es igual a 9, y los resultados obtenidos se muestran a continuación.
En el Backtracking, la figura 25 muestra la ultima solución obtenida, donde el algoritmo calculó 352 posibles soluciones.

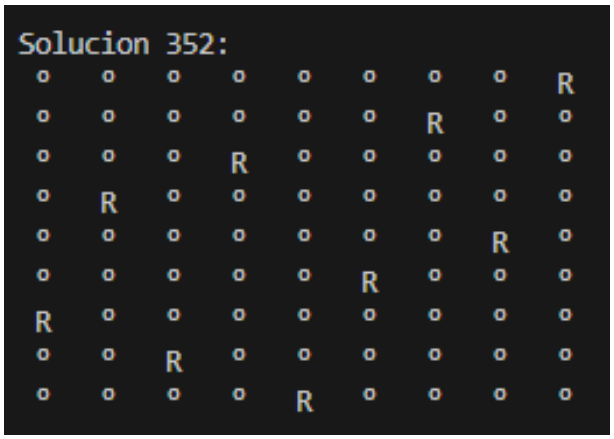


Figura 25: Ultima solución del Backtracking

La figura 26 muestra la solución encontrada por la poda del árbol, donde se encontró una única solución.

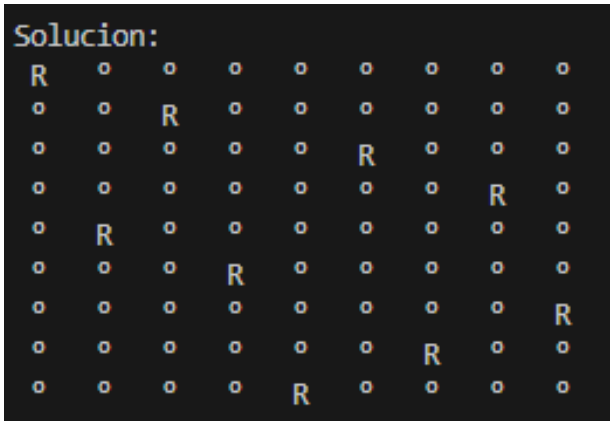


Figura 26: Ultima solución de la poda del árbol

Para la heurística, la figura 27 muestra la solución obtenida e igualmente es diferente a la obtenida por la poda.

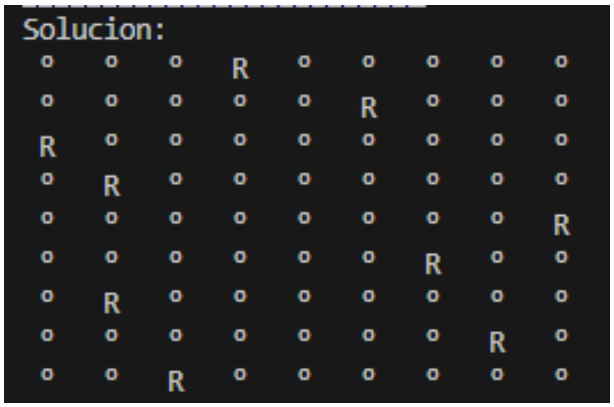


Figura 27: Ultima solución con heurísticas

Los tiempos de ejecución de la gráfica en la figura 28, muestran que la diferencia de tiempo entre el Backtracking y las otras dos opciones es abismal, por lo que deja al Backtracking como la peor opción de las 3.

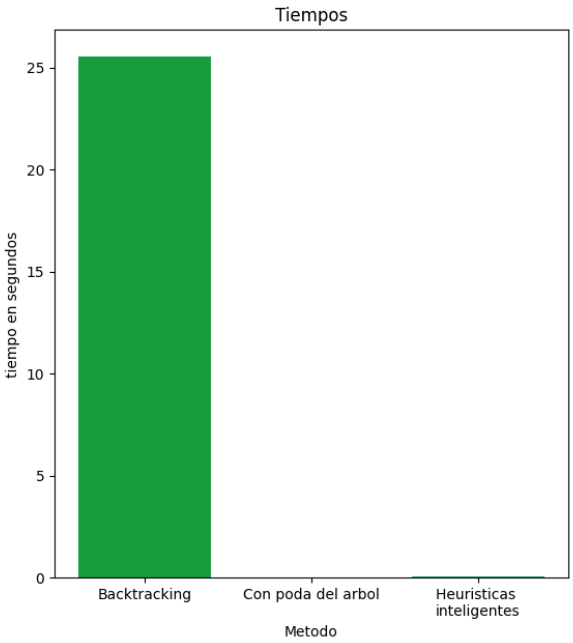


Figura 28: Gráfica con 9 reinas

Se hizo zoom en la parte de la poda y la heurística (figura 29) y se observa que el tiempo de la poda sigue siendo menor que la heurística, aunque no es un tiempo malo en comparación con el Backtracking, por lo que, la opción mas viable sigue siendo la técnica de poda.

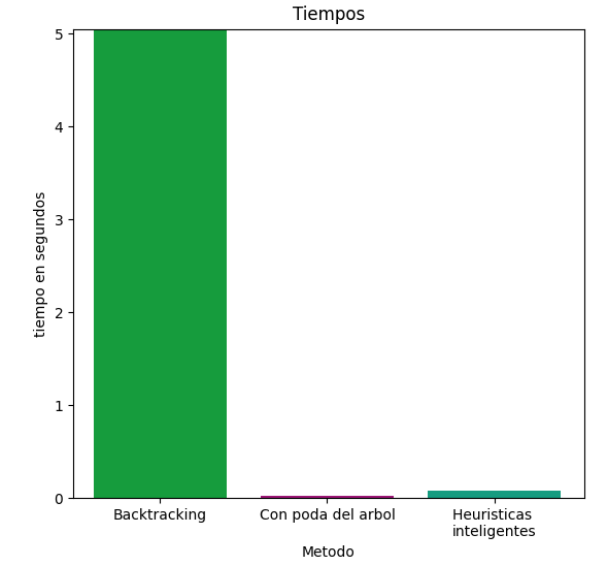


Figura 29: Zoom hecho en la gráfica

Conclusión

Como conclusión puedo decir que conforme las "n"vayan aumentando, el tiempo de ejecución tiende a subir por lo que lo mejor es hacer una optimización, en este caso, fueron con una poda del árbol y usando heurísticas.

Haciendo esto, se espera a que baje el tiempo en comparación con el código original, el que usa Backtracking, y al momento de medir el tiempo, se observa que la poda al no tomar en cuenta las columnas que ya revisó en filas anteriores, el tiempo se reduce considerablemente.
Para las heurísticas, también reduce el tiempo pero no tanto con la poda del árbol, pero esto no significa que la opción se mala, ya que hace un tiempo aceptable.
Puedo deducir que en el caso del Backtracking, al calcular todas las posibles soluciones, hace que el tiempo de ejecución sea mucho mayor al aumentar la cantidad de "n". Esto indica que si se cambia el código para solo calcular 1 solución se reduciría bastante el tiempo, pero seguiría siendo mayor que las otras dos opciones.
Otro punto a tomar en cuenta es, que, en el Backtracking se calculan todas las soluciones que puedan hallarse, esto hace que puedas escoger una solución que sea mas óptima globalmente. Pero para las otras dos técnicas, al calcular las soluciones no asegura obtener la solución óptima de manera global, sino, que encuentra la solución mas óptima de forma local.
Por ultimo, se puede concluir que la mejor técnica de optimización para el problema de las n reinas es usando la poda del árbol, mientras que la heurística si reduce el tiempo de ejecución, provoca que esta opción no es tan óptima.

Referencias

- EcuRed. (s. f.). Vuelta atrás (backtracking) - ECUReD. https://www.ecured.cu/Vuelta_atrás_(backtracking)
- Salcedo, L. (s. f.). El problema de las N-Reinas - Algoritmos Genéticos. Mi Diario Python. https://pythondiario.com/2018/05/el-problema-de-las-n-reinas-algoritmos.html
- BlogAdmin, & BlogAdmin. (2023, 7 julio). Algoritmos heurísticos: optimización inteligente. Informatica y Tecnologia Digital. https://informatecdigital.com/algoritmos/algoritmos-heuristicos-optimizacion-inteligente/
- Greyrat, R. (2022b, julio 5). Introducción a la escalada — Inteligencia Artificial – Barcelona Geeks. https://barcelonageeks.com/introduccion-a-la-escalada-inteligencia-artificial/