29 de Diciembre 2022

EVIDENCIA 2. EVIDENCIA ANÁLISIS Y DISEÑO DE ALGORITMOS

TC1031.604
3er Semestre
Tecnológica de Monterrey

Rodrigo Núñez Magallanes - A01028310
Iker Garcia German - A01782767
Rafael Blanga Hanon - A01781442

Introducción

Como evidencia final de la materia de Programación de Estructuras de Datos y Algoritmos Fundamentales, el equipo diseñó e implementó dos algoritmos en el lenguaje de programación Python, que dan solución al famoso “problema de las 8 reinas”. Este problema consiste en colocar 8 reinas en un tablero de ajedrez sin que se amenazan mutuamente, y para resolverlo programáticamente, el equipo diseñó e implementó dos algoritmos con distintas técnicas, una de ellas “fuerza bruta”, y la otra “programación dinámica”, específicamente la técnica de “arriba hacia abajo”. La primera consiste en encontrar una solución al problema sin considerar los recursos utilizados, y la segunda consiste en dividir el problema en partes más pequeñas y reducir operaciones innecesarias. Los objetivos a alcanzar fueron, además de elaborar los programas exitosamente, poder analizar los algoritmos elaborados y describir sus complejidades en términos de la notación Big-O.

Metodología

Los pasos que siguió el equipo para la resolución del problema fueron los siguientes:

1. Discusión y modelación.
2. Programación dinámica.
3. Segunda discusión y nuevas ideas.
4. Programación fuerza bruta.
5. Documentación.

Al inicio, el equipo se reunió a discutir la manera en la que, de forma lógica, el problema de las 8 reinas puede ser resuelto. Tras establecer una línea de pensamiento sobre cómo abordar el problema, se comenzó a programar la solución dinámica, en la que usando un algoritmo recursivo que funciona colocando 8 reinas en un tablero de ajedrez. Utiliza un enfoque de “backtracking” para buscar posiciones válidas para cada reina, y si se encuentra una posición válida, la almacena en el tablero. Si no se encuentra una posición válida, retrocede e intenta una posición diferente. Posteriormente y una vez lograda la solución dinámica, se pasó a la discusión del algoritmo de fuerza bruta, en donde la nueva línea de pensamiento consistió en que se iban a tratar todas las combinaciones posibles de las reinas en el tablero hasta encontrar una solución, y al encontrarla, dejar de buscar. Para esto, se realizó la programación con 8 ciclos for anidados, en donde se colocaba en un inicio a todas las reinas en la columna 0, y se iba adelantando la última reina a la siguiente columna en cada iteración. Al finalizar su ciclo, la penúltima reina avanza un espacio y la última vuelve a realizar el ciclo completo. Y así sucesivamente, hasta llegar a la solución.

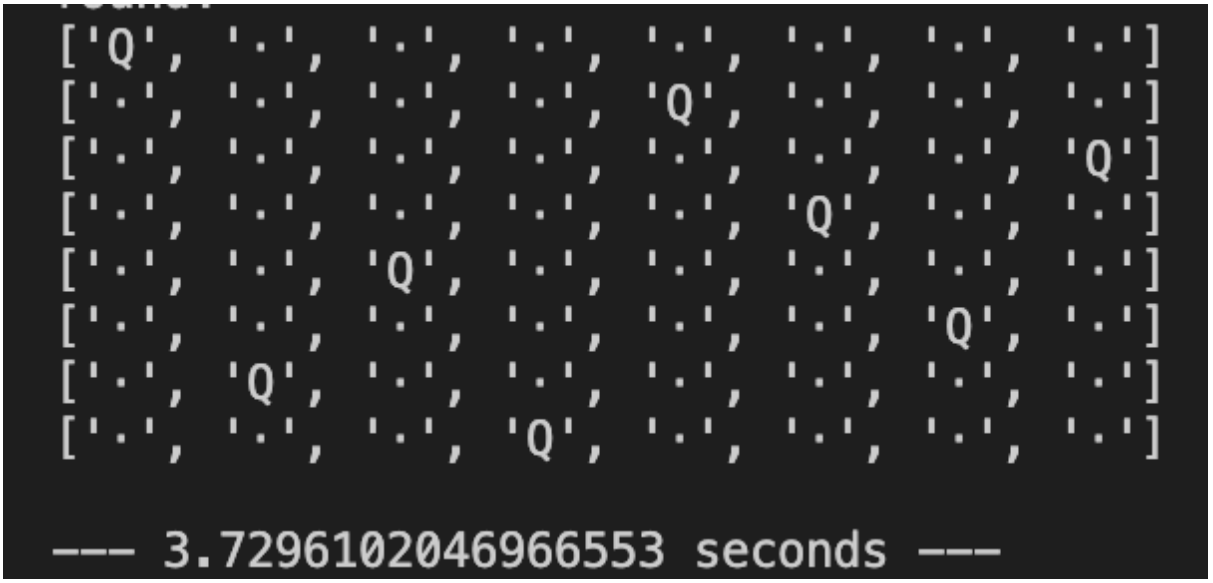
Naturalmente, en cada iteración general se debía evaluar si ya se encontraba el programa en una solución, para entonces salirse, pero esto agregó muchas evaluaciones al programa, cosa que agrega operaciones y por tanto complejidad al algoritmo, pero como se estaba en fuerza bruta, no importa. Finalmente, al llegar a la solución se pasó a comentar el código para dar contexto y explicación a cualquiera que lo lea, hacer unas modificaciones estéticas al mismo y a realizar el presente documento, junto con el análisis.

Implementación

Al correr ambos programas, inmediatamente fuimos capaces de identificar la diferencia entre ambos programas, En este caso, el algoritmo dinámico fue el mejor. Esto es debido a que fue capaz de

encontrar la solución más óptima con una complejidad temporal y espacial más baja que el algoritmo de fuerza bruta. Esto significa que el algoritmo dinámico fue mucho más eficiente, ya que se necesita menos tiempo y menos memoria para encontrar la solución. Además, el algoritmo dinámico se puede aplicar a problemas más complejos, mientras que el algoritmo de fuerza bruta es más adecuado para problemas más simples.

Para probar esta hipótesis agregamos una función de tiempo, la cual nos permite evaluar la velocidad y eficacia a la cual corrió cada uno de los programas.



Algoritmo de fuerza bruta.

Algoritmo dinámico.

Notaciones asintóticas:

También fuimos capaces de identificar la eficiencia de cada algoritmo gracias a su Big O notation:

Algoritmo Fuerza bruta: El código tiene una complejidad algorítmica de O(N^16), porque hay 8 bucles anidados que se ejecutan un número de veces igual al número de reina. Esto significa que el

tiempo de ejecución del algoritmo aumenta de manera exponencial con el aumento del número de reinas. además posee una última función la cuál se encarga evaluar de nuevo cada posición y verificar que no exista conflicto entre reinas.

Algoritmo dinámico: La notación Big O de este código es O (N ^ 4), ya que el algoritmo contiene 4 For anidados, cada uno con una complejidad de O (N).

Por ende, tenemos la evidencia necesaria para entender la diferencia entre ambos programas y sus respectivas eficacias.

Visualización

Para su mejor visualización, logramos hacer un código que implementa la librería de "chess" que utiliza la anotación de FEN y lo procesa como imágenes. Para esto tuvimos que implementar los códigos ya hechos en un google colabo y modificarlos para obtener dicha anotación de FEN. Podemos ver cómo funciona:

```
import chess
import chess.svg
import io
import time
```

Estas fueron las librerías que usamos para llegar a nuestro resultado

```
N = 8
# NxN matrix with all elements set to 0
board = [["em"]*N for _ in range(N)]
```

Con esto hacemos nuestro tablero de 8x8.

```
def attack(i, j):
    #checking vertically and horizontally
    for k in range(N):
        if board[i][k]=="wq" or board[k][j] == "wq":
            return True
    #checking diagonally
    for k in range(N):
        for l in range(N):
            if (k+l==i+j) or (k-l == i-j):
                if board[k][l] == "wq":
                    return True
    return False
```

Esta es nuestra función para saber si están bajo ataque alguna de las reinas. checando primero filas y columnas y luego diagonales.

```
def N_queens(n):
    if n == 0:
        return True
    for i in range(N):
        for j in range(N):
            if (not(attack(i,j))) and (board[i][j] != "wq"):
                board[i][j] = "wq"
                if N_queens(n-1) == True:
                    return True
                board[i][j] = "em"
    return False
```

Y esta función coloca nuestra reina en el tablero y está definido por “wq” que significa White Queen. Con todo esto corriendo el programa no da esto.

```
['wq', 'em', 'em', 'em', 'em', 'em', 'em', 'em']
['em', 'em', 'em', 'em', 'wq', 'em', 'em', 'em']
['em', 'em', 'em', 'em', 'em', 'em', 'em', 'wq']
['em', 'em', 'em', 'em', 'em', 'wq', 'em', 'em']
['em', 'em', 'wq', 'em', 'em', 'em', 'em', 'em']
['em', 'em', 'em', 'em', 'em', 'em', 'wq', 'em']
['em', 'wq', 'em', 'em', 'em', 'em', 'em', 'em']
['em', 'em', 'em', 'wq', 'em', 'em', 'em', 'em']
--- 2.296616554260254 seconds ---
```

Asi ya se podría entender que “em” significa “empty” y podríamos entender el resultado
Pero nosotros quisimos extender este resultado logrando convertir esta matriz en un string en formato FEN para que la librería de chess lo pueda leer.
Entonces primero implementamos una función que lee esta matriz y la convierte a un dicho string.

```
def board_to_fen(board):
    # Use StringIO to build string more efficiently than concatenating
    with io.StringIO() as s:
        for row in board:
            empty = 0
            for cell in row:
                c = cell[0]
                if c in ('w', 'b'):
                    if empty > 0:
                        s.write(str(empty))
                        empty = 0
                    s.write(cell[1].upper() if c == 'w' else
cell[1].lower())
                else:
```

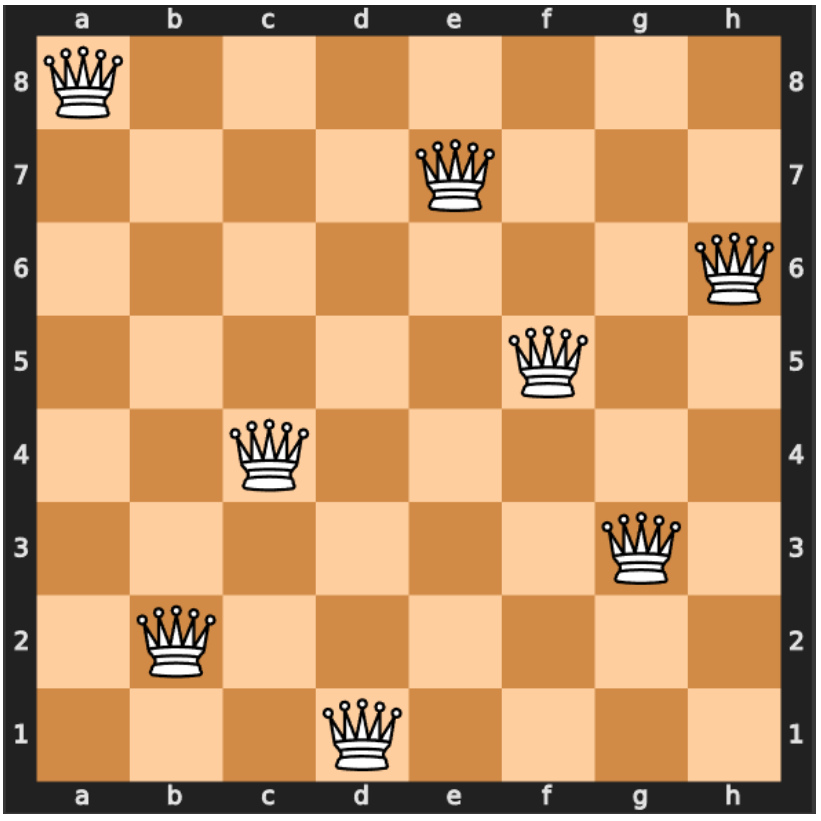
```
                    empty += 1
                if empty > 0:
                    s.write(str(empty))
                s.write('/')
            # Move one position back to overwrite last '/'
            s.seek(s.tell() - 1)
            # If you do not have the additional information choose what to
put
            s.write(' ')
            return s.getvalue()
```

Con esta función y con la librería de string IO terminamos con este string.

```
Q7/4Q3/7Q/5Q2/2Q5/6Q1/1Q6/3Q4
```

Y luego terminamos con la librería de Chess para que lea dicho string y nos quedamos con esto:

```
imgBoard = chess.Board(board_to_fen(board))
imgBoard
```



Resultado

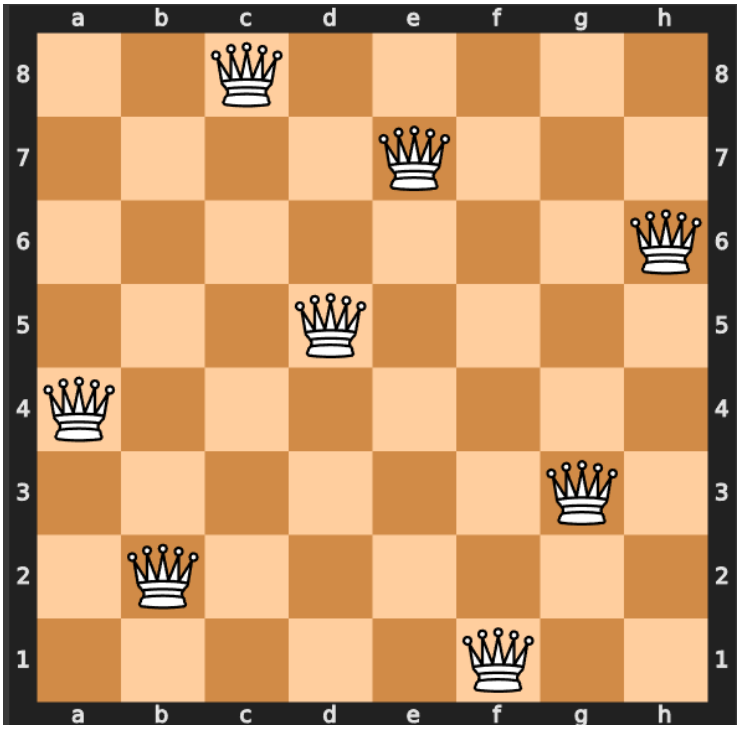
También podemos observar esto para el código de Fuerza Bruta. Lo modificamos tantito para poder darle un valor al inicio de la función para poder visualizar una solución en específico. El código es el siguiente:

```
def main2():
  board2 = [
        ["em","em","em","em","em","em","em","em"],
        ["em","em","em","em","em","em","em","em"],
        ["em","em","em","em","em","em","em","em"],
```

```
        ["em","em","em","em","em","em","em","em"],
        ["em","em","em","em","em","em","em","em"],
        ["em","em","em","em","em","em","em","em"],
        ["em","em","em","em","em","em","em","em"],
        ["em","em","em","em","em","em","em","em"],
        ]
inicio = int(input("Indique el numero de solución que desea visualizar: "))
final = 0
while True:
      for col0 in range(8):
              insert(0, col0)
              for col1 in range(8):
                      insert(1, col1)
                      for col2 in range(8):
                              insert(2, col2)
                              for col3 in range(8):
                                      insert(3, col3)
                                      for col4 in range(8):
                                              insert(4, col4)
                                              for col5 in range(8):
                                                      insert(5, col5)
                                                      for col6 in range(8):
                                                              insert(6, col6)
                                                              for col7 in range(8):
                                                                      insert(7, col7)
                                                                      if not conflict(0, col0) and not conflict(1, col1) and not
conflict(2, col2) and not conflict(3, col3) and not conflict(4, col4) and not conflict(5, col5) and not conflict(6, col6) and not
conflict(7, col7):
                                                                              print("found!")
                                                                              final += 1
                                                                              print("Solución encontrada #: "+str(final))
                                                                              if final == inicio:
                                                                                return False
```

Y el resultado es el siguiente si le damos a buscar la solución #5:

```
Indique el numero de solución que desea visualizar: 5
found!
Solución encontrada #: 1
found!
Solución encontrada #: 2
found!
Solución encontrada #: 3
found!
Solución encontrada #: 4
found!
Solución encontrada #: 5
```



Solución #5

Conclusiones

En cuanto al algoritmo de fuerza bruta, se puede decir que su gran desventaja, naturalmente, es que como desconsidera los recursos del sistema, y su complejidad en notación Big-O no es para nada deseable, se le exige trabajo que podríamos decir que es innecesario a la máquina. Por ejemplo, en la primera iteración, todas las reinas se encuentran en la misma columna, y obviamente en ese caso están amenazadas todas, y suceden varias iteraciones hasta que alguna de las reinas escapa de la amenaza. Su ventaja más notable es que garantiza que se va a encontrar una solución y, además, si se realiza un pequeño ajuste al código es posible encontrar todas las soluciones posibles, y esta garantía existe porque como el algoritmo va a probar todas las combinaciones posibles en el tablero, forzosamente encontrará las soluciones, solo que eso significaba otra desventaja como arma de doble filo, ya que el código entonces se ejecutaría para todas las combinaciones posibles de las reinas, cuyo número rebasa los 4 mil millones.

Por otro lado, la ventaja del dinámico es que ahorra recursos ya que evita operaciones "innecesarias" porque no va probando absolutamente todas las combinaciones posibles, si no que coloca a las reinas siempre y cuando sea posible hacerlo, es decir, la casilla en cuestión no esté amenazada ni el resto del tablero termine de tener posibilidades de encontrar solución para el número de reinas restantes. Además, naturalmente, este algoritmo toma mucho menos tiempo que el anterior porque precisamente evita las operaciones innecesarias. Ambos llegan a la misma solución porque es la primera posible solución al problema si vamos iterando todas las combinaciones posibles utilizando la última fila como medida más pequeña, pero el dinámico tarda mucho menos porque no hace operaciones que no necesita. Su desventaja probablemente podría ser que para un programador externo igual y es más difícil comprender el código ya que el de fuerza bruta es mucho más explícito, por ser menos "abstracto".

En general, se concluyó que mientras sea posible utilizar menos recursos y hacer algoritmos más "inteligentes", será más beneficioso en términos de costo (recursos) y tiempo. Principalmente, se considera que lo que hay que tener en cuenta es que un enfoque de fuerza bruta, aunque garantice una solución, puede ser que realice operaciones innecesarias, y si el número de estas operaciones es muy elevado, esto representa costo y tiempo, mientras que con un enfoque dinámico se puede ahorrar todo esto y lo más importante de todo, descomponer el problema en cuestión en partes más pequeñas.

Bibliografía

https://python-chess.readthedocs.io/en/latest/index.html