

INTELIGENCIA ARTIFICIAL - PIA

DOCENTE:

Juan Pablo Rosas Baldazo

ALUMNOS:

Fátima Daniela Lozoya Leal
Javier Alberto Gaona Rodríguez
Carlos Sánchez Velázquez

DESCRIPCIÓN GENERAL DE PROBLEMA

TORRES DE HANÓI

Las Torres de Hanói es un rompecabezas o juego matemático que consiste en un número de discos perforados de radio creciente que se apilan insertándose en uno de los tres postes fijados a un tablero.

-PARAMETROS:

-Discos o aros:

- Cantidad (n): se puede realizar el problema con una cantidad distinta de discos.
- Tamaño: los discos tienen que ser de distintos tamaños, siempre serán ordenados de manera descendente.
- Función: Los discos se moverán a través de las barras hasta lograr el objetivo de colocarlos todos de nuevo en orden descendente en la barra destino.

-Barras o postes:

- Cantidad: Por definición solo hay 3 barras.
- Tamaño: Indefinido.
- Función: La primera barra tendrá todos los discos ordenados de manera descendente posicionando así el disco más grande en la base. El objetivo es transportar los discos de la barra inicial a la barra destino en el mismo orden pero cumpliendo 2 reglas sencillas.

-REGLAS:

- El agente se podrá mover hacia la derecha: Los discos pueden moverse hacia el poste de la derecha ya sea desde el primero al segundo, del primero al ultimo o del segundo al último. (A a B, A a C, B a C).
- El agente se podrá mover hacia la izquierda: Los discos pueden moverse hacia el poste de la izquierda ya sea desde el segundo al primero, del ultimo al primero o del del ultimo al segundo. (B a A, C a A, C a B).

-RESTRICCIONES:

1. No se puede mover más de un disco a la vez. Por definición en el problema de las torres de Hanoi no está permitido mover más de un disco a la vez, pues va en contra de las reglas del problema. Este fue creado para que su solución se de con solo mover este único disco.
2. No se puede colocar un disco grande sobre otro de menor tamaño. Por definición el problema de las torres de Hanoi no está permitido que el usuario pueda colocar un disco más grande sobre otro más pequeño.

Estás reglas son creadas por definición, haciéndolo así un poco más complicado de resolver, pues sabemos que para resolver este problema es en la menor cantidad de movimientos posibles, pero solo de esta manera podemos hacerlo de una manera más eficaz, pues así obtendremos el resultado de una manera más precisa y rápida.

Los algoritmos empleados para la búsqueda de una solución fueron DFS, BFS, Best FS.

PRE-REQUISITOS:

Para la ejecución de los scripts se necesitan varias librerías como
numpy pandas pygal lxml.

INSTALACIÓN:

Use la siguiente línea de comando para instalar las dependencias

```
pip install numpy pandas pygal lxml
```


PARA COMENZAR:







El seguimiento de los scripts es el siguiente, donde cada uno tiene una función específica y algunos en particular tienen un orden de ejecución.

- **torre-de-hanoi.py** - Este script lo que hace es hacer 20 iteraciones de los 3 algoritmos, donde se genera un estado inicial totalmente aleatorio y diferente de los demás.

	Estado Inicial	Tiempo DFS	Movimientos DFS	Tiempo BFS	Movimientos BFS	Tiempo BFS_e	Movimientos BFS_e
0	[[], [1, 2], [3, 4]]	0.099090	67	0.077027	15	0.004004	15
1	[[4], [2, 1], [3]]	0.163149	37	0.079072	15	0.002002	15
2	[[3], [1, 2], [4]]	0.005005	5	0.006005	3	0.002002	3
3	[[2], [3, 4], [1]]	0.186169	53	0.019017	6	0.002002	6
4	[[3, 2, 4], [1], []]	0.171156	68	0.031029	8	0.001001	8
5	[[4, 1], [3], [2]]	0.158144	49	0.042038	10	0.002002	10
6	[[4], [2], [1, 3]]	0.129117	76	0.065059	12	0.003003	12
7	[[2], [1], [3, 4]]	0.100091	66	0.086078	17	0.001001	17
8	[[], [4, 1, 2], [3]]	0.073066	58	0.063057	11	0.002002	11
9	[[4, 3], [1], [2]]	0.036033	38	0.058053	13	0.002002	13
10	[[2, 4, 1, 3], [], []]	0.019018	24	0.033030	8	0.001001	8
11	[[1], [4, 3], [2]]	0.120110	74	0.058053	13	0.002002	13
12	[[1, 2], [], [4, 3]]	0.005004	9	0.004004	2	0.002002	2
13	[[2], [4, 1], [3]]	0.071065	57	0.074067	15	0.001001	15
14	[[2], [], [4, 1, 3]]	0.251228	18	0.025023	6	0.001001	6
15	[[4, 2, 3, 1], [], []]	0.169154	34	0.065059	12	0.002002	12
16	[[2, 1], [4, 3], []]	0.128158	76	0.076027	15	0.001001	15
17	[[], [2], [4, 3, 1]]	0.005005	6	0.007006	3	0.001001	3
18	[[4, 2], [1, 3], []]	0.186169	72	0.025023	6	0.002042	6
19	[[], [1, 4, 3, 2], []]	0.213153	40	0.069062	11	0.001001	11

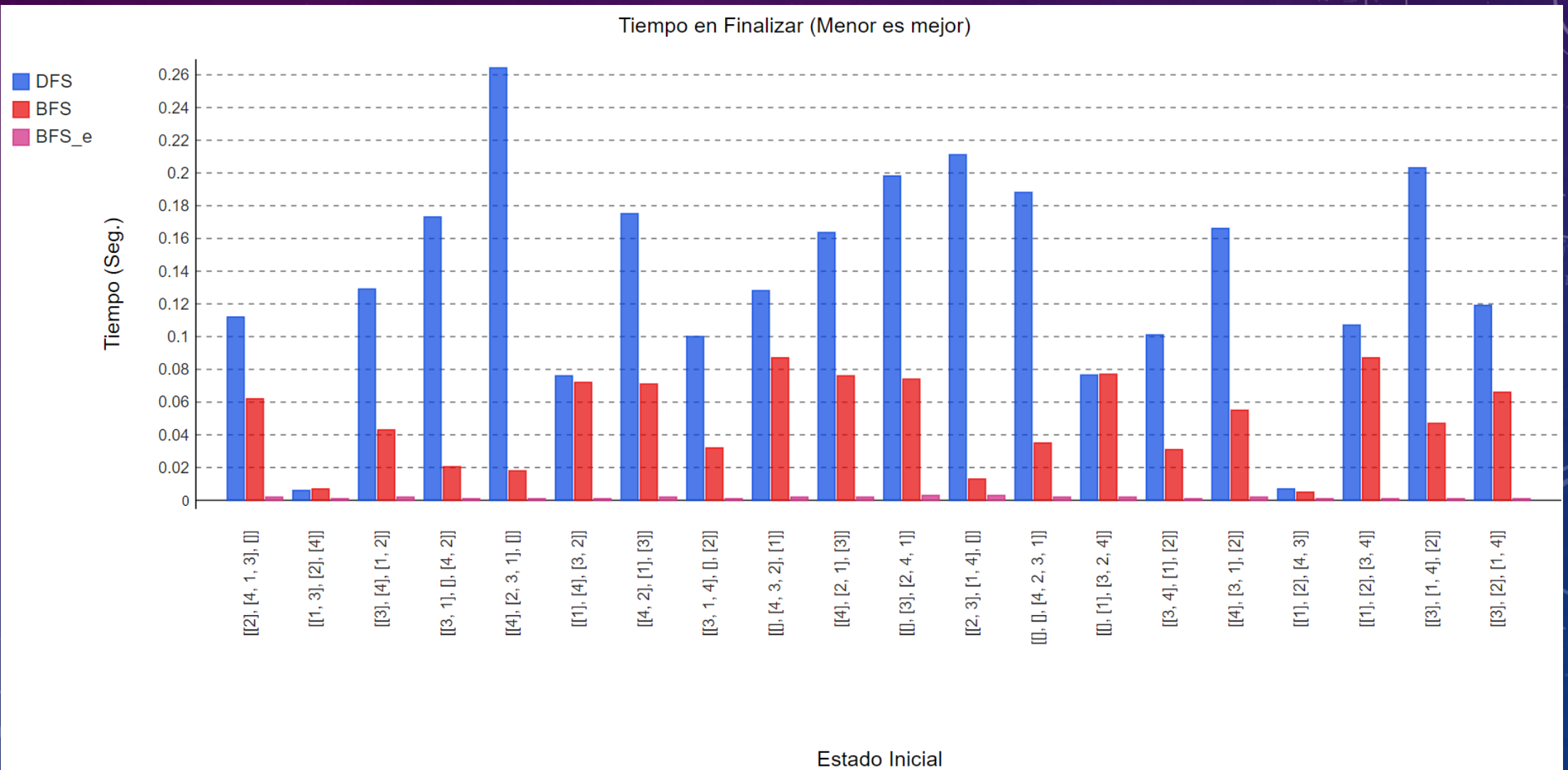
Presiona cualquier tecla para continuar...

Después de haber generado los recorridos, el script generara un archivo con extensión .csv, donde este contendrá los datos obtenidos de los recorridos de los algoritmos.

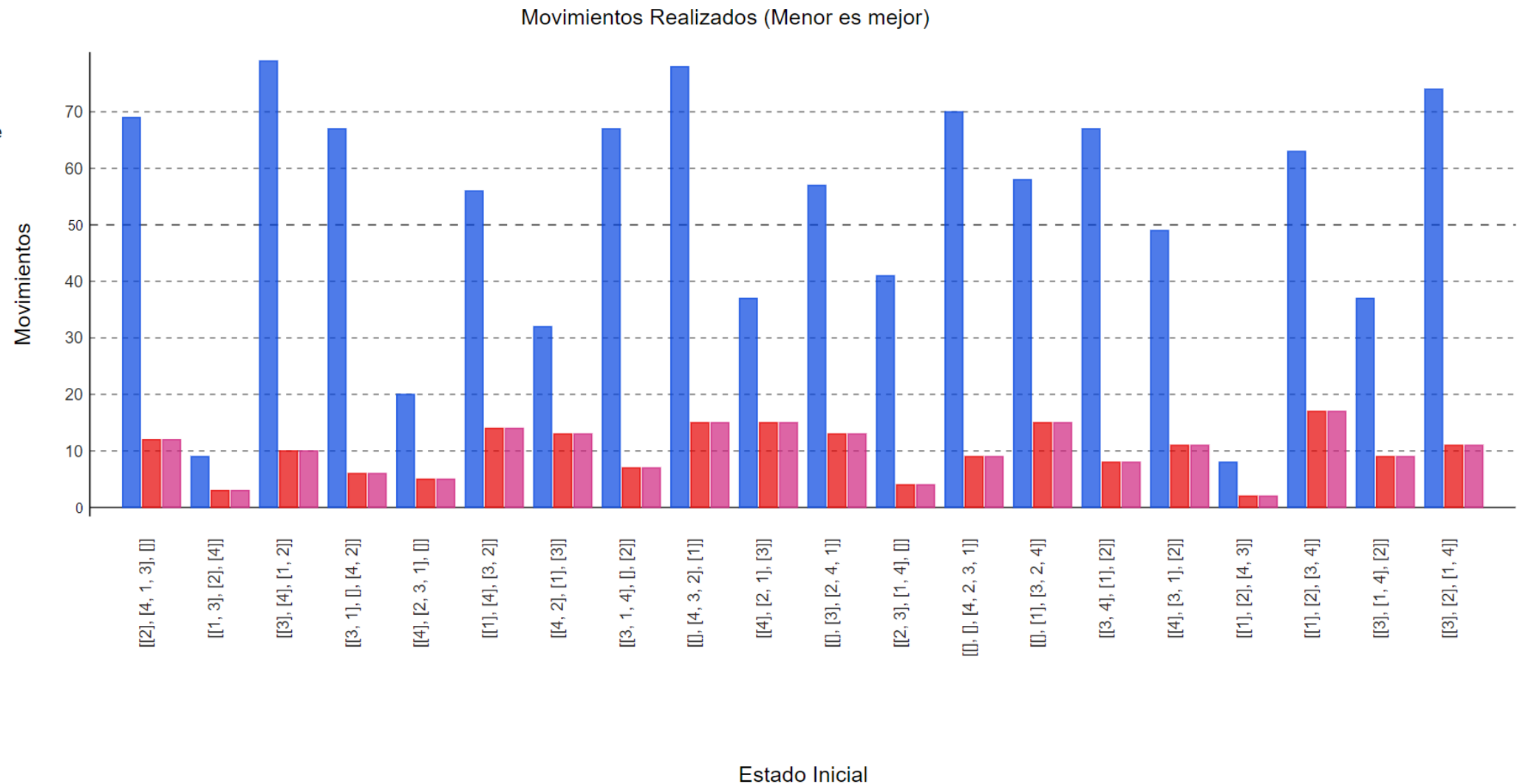
 .gitattributes	29/11/2020 23:41	Documento de te...	1 KB
 plot.csv	02/12/2020 22:11	Microsoft Excel C...	2 KB
 plot.py	01/12/2020 14:24	Python File	2 KB
 README.md	01/12/2020 13:41	Archivo MD	1 KB
 torre-de-hanoi.py	01/12/2020 14:25	Python File	10 KB
 torre-de-hanoi-graphic.py	01/12/2020 14:21	Python File	9 KB

plot.py - Después de haber generado el archivo .csv ahora podemos ejecutar de manera correcta **plot.py**, lo que hará este script es que con los datos recabados generara unas graficas.

Por ejemplo, una de las graficas es que compare el tiempo de los tres algoritmos, desde el estado inicial al estado meta.



Otra grafica compara los movimientos realizados de los tres algoritmos, desde el estado inicial al estado meta.



torre-de-hanoi-graphics.py - Este script puede ser de utilidad ya que nos ofrece de manera visible el como se encontró la solución de cada estado inicial en cualquiera de los 3 algoritmos, solo hay que elegir el algoritmo.

```
1. Depth First Search
2. Breadth First Search
3. Best First Search
4. Salir

Selecciona un algoritmo--> 1_
```

Después se introduce el estado inicial, del que quieres ver el procedimiento. OJO: en caso de que en alguna columna, poste,etc; este no tenga algún disco solo se da enter y no se teclea nada.

De tal modo que para el primer estado, de lo que genero **torre-de-hanoi.py** en la imagen de arriba que así.

```
1. Depth First Search
2. Breadth First Search
3. Best First Search
4. Salir

Selecciona un algoritmo--> 1

Ingresa el esatdo inicial:
Discos en la columna 1 :

Discos en la columna 2 :
1 2
Discos en la columna 3 :
3 4
```

Una vez ingresado el estado inicial, empezara el algoritmo a encontrar una solución, y esta se vera de esta manera. Donde en la parte de arriba busca la solución del estado inicial, una vez que lo encuentra, empieza a dibujar los estados que se dieron con el fin de llegar al estado meta.

```
Nodo 63 [[], [2], [4, 3, 1]]
Nodo 64 [[2], [], [4, 3, 1]]
Nodo 65 [[2, 1], [], [4, 3]]
Nodo 66 [[2], [1], [4, 3]]
Nodo 67 [[], [1], [4, 3, 2]]
Nodo 68 [[1], [], [4, 3, 2]]
Nodo 69 [[], [], [4, 3, 2, 1]]
```

Estado Meta Alcanzado

```
Nodo Numero: 69    Estado:  [[], [], [4, 3, 2, 1]]
```

```
Nodo Numero: 68    Estado:  [[1], [], [4, 3, 2]]
```

The diagram illustrates two scenarios of a vertical line with circles. In the left scenario, a vertical line has three circles at the bottom. In the right scenario, a vertical line has three circles at the bottom, and a stack of three circles is positioned to its right.

PSEUDOCODIGO

Inicio

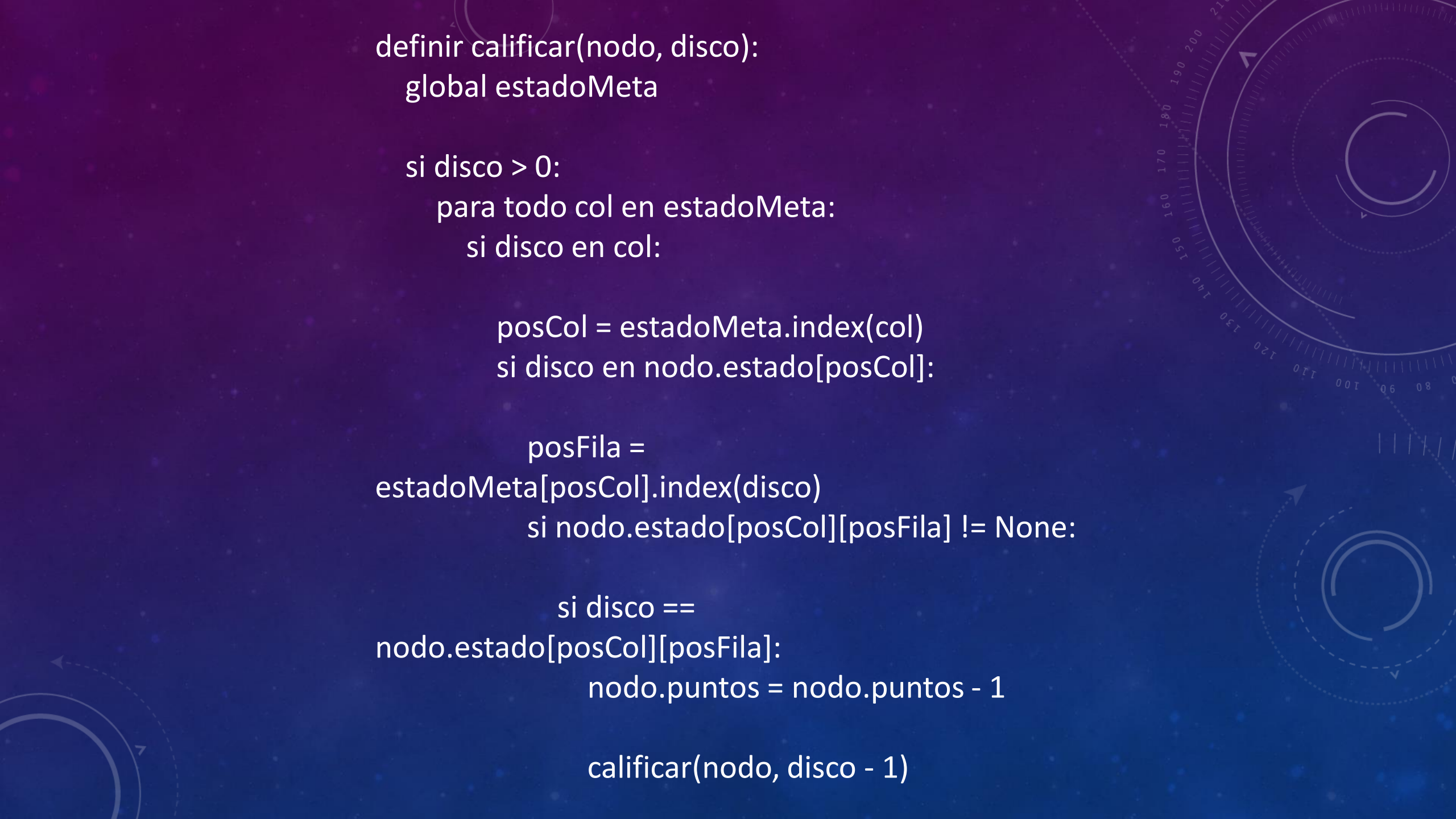
Clase Nodo:

```
definir __init__(self):  
    self.estado = [[], [], []]  
    self.numeroNodo = 0  
    self.padre = None  
    self.hijos = []  
    self.puntos = 0
```

```
definir evaluar(nodo):  
    global numDiscos, columnas
```

```
    disco = numDiscos  
    nodo.puntos = numDiscos * columnas
```

```
    calificar(nodo, disco)
```



```
def calificar(nodo, disco):  
    global estadoMeta
```

```
    si disco > 0:  
        para todo col en estadoMeta:  
            si disco en col:
```

```
                posCol = estadoMeta.index(col)  
                si disco en nodo.estado[posCol]:
```

```
                    posFila =  
estadoMeta[posCol].index(disco)  
                    si nodo.estado[posCol][posFila] != None:
```

```
                        si disco ==  
nodo.estado[posCol][posFila]:  
                            nodo.puntos = nodo.puntos - 1
```

```
                            calificar(nodo, disco - 1)
```



```
def definir movimientos(nodo):  
    global columnas, numeroNodo, estados  
    pila = []  
  
    para toda x en range(0, columnas):  
        para toda y en range(0, columnas):  
  
            pila = movimiento(nodo.estado[x],  
nodo.estado[y])  
  
            si pila es diferente None:  
                nodoSig = Nodo()  
                nodoSig = deepcopy(nodo)  
                nodoSig.estado[x] = deepcopy(pila[0])  
                nodoSig.estado[y] = deepcopy(pila[1])  
  
                si nodoSig.estado no esta en estados:  
                    numeroNodo = nodoSig.numeroNodo  
                    estados.append(nodoSig.estado)  
                    retornar nodoSig  
  
    retornar None
```




```
def definir movimiento(pila1, pila2):
```

```
    p1 = pila1[:]
```

```
    p2 = pila2[:]
```

```
    si len(p1) > 0:
```

```
        disco = p1[len(p1) - 1]
```

```
        indexP2 = len(p2) - 1
```

```
        si len(p2) == 0 or p2[indexP2] > disco:
```

```
            p2.append(disco)
```

```
            p1.pop()
```

```
            retornar p1, p2
```

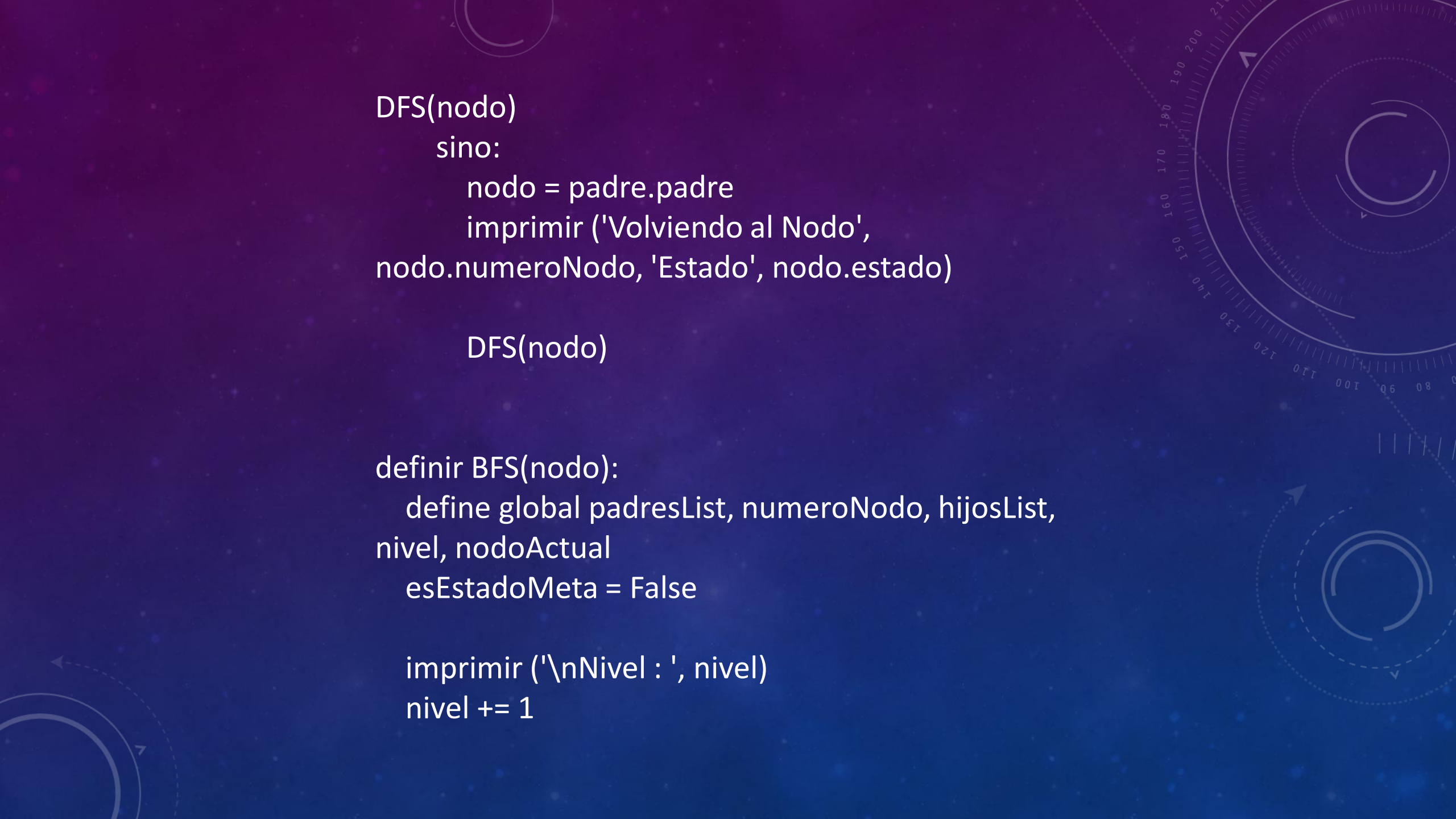
```
        sino:
```

```
            retornar None
```

```
    sino:
```

```
        retornar None
```

```
def DFS(nodo):  
    define global numeroNode, nodoActual  
    esEstadoMeta = False  
  
    si esEstadoMeta == False:  
        padre = deepcopy(nodo)  
        nodo = movimientos(nodo)  
  
        si nodo != None:  
            numeroNode += 1  
            nodo.numeroNode = numeroNode  
            nodo.padre = padre  
            imprimir ('Nodo ', nodo.numeroNode,  
nodo.estado, '\n')  
  
            si nodo.estado == estadoMeta:  
                imprimir('\n\nEstado Meta Alcanzado')  
                nodoActual = nodo  
                esEstadoMeta = True  
  
    retornar True
```



```
DFS(nodo)
    sino:
        nodo = padre.padre
        imprimir ('Volviendo al Nodo',
nodo.numeroNodo, 'Estado', nodo.estado)
```

```
DFS(nodo)
```

```
definir BFS(nodo):
    define global padresList, numeroNodo, hijosList,
nivel, nodoActual
    esEstadoMeta = False

    imprimir ('\nNivel : ', nivel)
    nivel += 1
```

```
para todo nodo en padresList:
    si esEstadoMeta == False:
        imprimir ('Nodo Padre:', nodo.numeroNodo, ' Estado :',
nodo.estado)
        tieneHijos = False
        padre = deepcopy(nodo)

    i = 1
    mientras tieneHijos == False:
        hacer
            i += 1
            hijo = movimientos(nodo)

            si hijo != None:
                numeroNodo += 1
                hijo.numeroNodo = numeroNodo
                hijo.padre = nodo
                padre.hijos.append(hijo)
                hijosList.append(hijo)
                imprimir (' L--Nodo Hijo:', hijo.numeroNodo,
'Estado:', hijo.estado)
```



```
si hijo.estado == estadoMeta:
    imprimir ('\n\nEstado Meta
Alcanzado')
    nodoActual = hijo
    esEstadoMeta = True

    retornar True
else:
    tieneHijos = True

padresList = deepcopy(hijosList)
hijosList = []

si esEstadoMeta == False:
    BFS(padresList)
```



```
definir BFS_enhanced(nodo):  
    define global padresList, numeroNodo,  
    hijosList, numDiscos, nodoActual
```

```
    esEstadoMeta = False  
    puntosMin = numDiscos * columnas
```

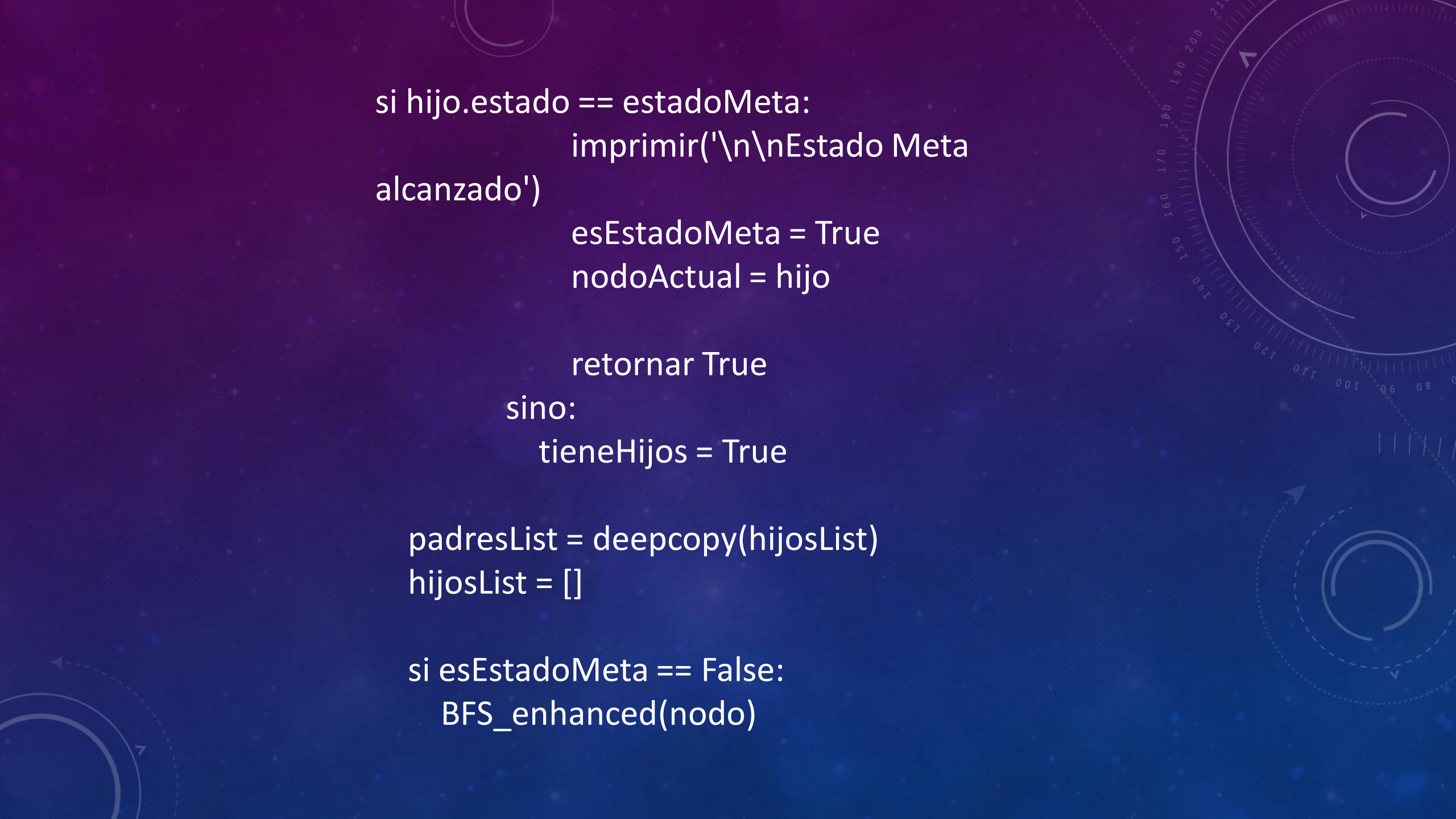
```
    para todo nodo en padresList:  
        evaluar(nodo)
```

```
        si nodo.puntos < puntosMin:  
            puntosMin = nodo.puntos
```

```
para todo nodo en padresList:
    si esEstadoMeta == False and nodo.puntos == puntosMin:
        imprimir('\nNodo Padre:', nodo.numeroNodo, ' Estado
:', nodo.estado, 'Costo = ', nodo.puntos)
        tieneHijos = False
        padre = deepcopy(nodo)

    i = 1
    mientras tieneHijos == False:
        hacer
            i += 1
            hijo = movimientos(nodo)

            si hijo != None:
                numeroNodo += 1
                hijo.numeroNodo = numeroNodo
                hijo.padre = nodo
                hijosList.append(hijo)
                print(' L--Nodo Hijo:', hijo.numeroNodo, 'Estado:',
hijo.estado)
```



```
si hijo.estado == estadoMeta:
    imprimir('\n\nEstado Meta
alcanzado')
    esEstadoMeta = True
    nodoActual = hijo

    retornar True
sino:
    tieneHijos = True

padresList = deepcopy(hijosList)
hijosList = []

si esEstadoMeta == False:
    BFS_enhanced(nodo)
```



```
def definir obtenerIntentos(nodo):  
    estados = deepcopy(nodo)
```

```
    i = 0
```

```
    mientras True:
```

```
        si nodo.padre != None:
```

```
            i += 1
```

```
            nodo = nodo.padre
```

```
        sino:
```

```
            romper
```

```
    retornar i
```



```
def dibujarSolucion(nodo):
    define global numDiscos, columnas

    mientras True:
        hacer
            estado = nodo.estado
            imprimir('\nNodo Numero: ', nodo.numeroNodo, ' Estado: ', estado)
            imprimir("\n ", imprimirDisco(None), end="")
            imprimir(" ", imprimirDisco(None), end="")
            imprimir(" ", imprimirDisco(None))

        para toda fila en range(numDiscos).__reversed__():
            para toda col en range(columnas):
                intentar:
                    imprimir(" ", imprimirDisco(estado[col][fila]),
end="")
                excepcion:
                    imprimir(" ", imprimirDisco(None), end="")
            imprimir("")
```




```
imprimir("#" * int(((numDiscos * 2) + 2) * 4))
```

```
    si nodo.padre != None:  
        nodo = nodo.padre  
    sino:  
        romper
```

```
def definir imprimirDisco(numDisco):  
    espacio = (numDiscos * 2) + 1  
    disco = ""
```

```
    si numDisco == 0 o numDisco == None:  
        para todo i en range(espacio):  
            si i == int(espacio / 2):  
                disco += "|"  
            sino:  
                disco += " "
```

```
    retornar disco
```

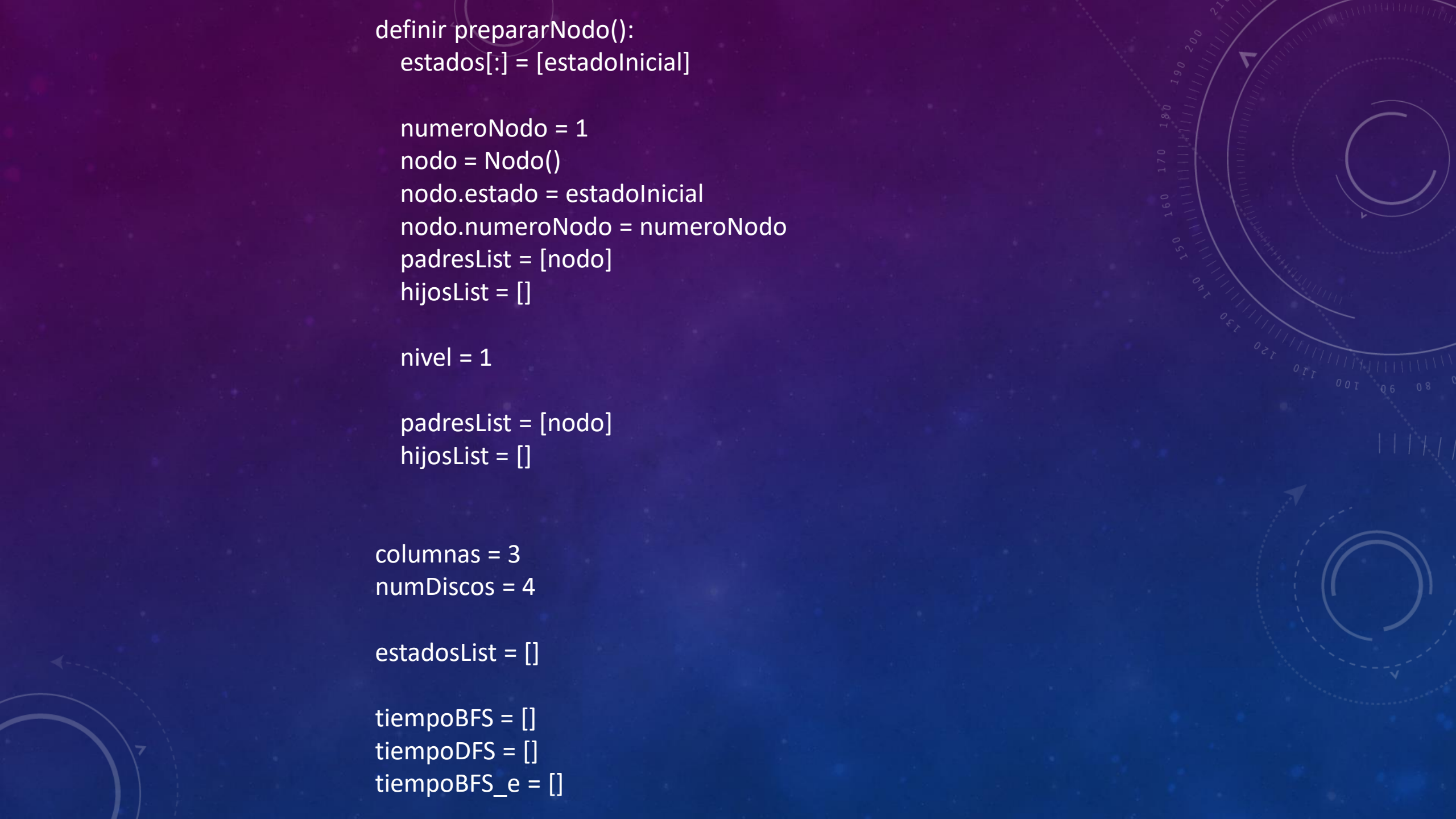
```
espacios = espacio - ((numDisco * 2) + 1)
para toda i en range(int(espacios / 2)):
    disco += " "

para toda i en range((numDisco * 2) + 1):
    disco += "o"

para toda i en range(int(espacios / 2)):
    disco += " "

retornar disco
```

```
def definir estadoAleatorio(numDiscos):  
    define global columnas  
    discos = []  
  
    para toda i en range(numDiscos):  
        discos.append(i + 1)  
  
    random.shuffle(discos)  
  
    estadoInicial = [[], [], []]  
  
    mientras len(discos) > 0:  
        hacer  
            columnaAleatoria = int(random.randint(columnas))  
            indexDiscoAleatorio = random.randint(len(discos))  
  
            estadoInicial[columnaAleatoria].append(discos.pop(indexDiscoAleatorio))  
  
        random.shuffle(estadoInicial)  
  
    retornar estadoInicial
```



```
defnir prepararNodo():  
    estados[:] = [estadoInicial]
```

```
    numeroNodo = 1  
    nodo = Nodo()  
    nodo.estado = estadoInicial  
    nodo.numeroNodo = numeroNodo  
    padresList = [nodo]  
    hijosList = []
```

```
    nivel = 1
```

```
    padresList = [nodo]  
    hijosList = []
```

```
columnas = 3  
numDiscos = 4
```

```
estadosList = []
```

```
tiempoBFS = []  
tiempoDFS = []  
tiempoBFS_e = []
```



```
movimientosBFS = []  
movimientosDFS = []  
movimientosBFS_e = []
```

```
estadoInicial = [[2], [3], [1, 4]]  
estadoMeta = [[], [], [4, 3, 2, 1]]
```

```
para toda i en range(20):
```

```
    mientras True:
```

```
        hacer
```

```
            estadoInicial = estadoAleatorio(numDiscos)
```

```
            si estadoInicial no esta en estadosList:
```

```
                estadosList.append(estadoInicial)
```

```
                romper
```

```
            imprimir("\n" * 3, "-" * 10, "Numero de iteración: ", i + 1, " ",  
                    "-" * 10)
```

```
            imprimir('Estado Inicial: ', estadoInicial)
```




```
estados = [estadoInicial]
```

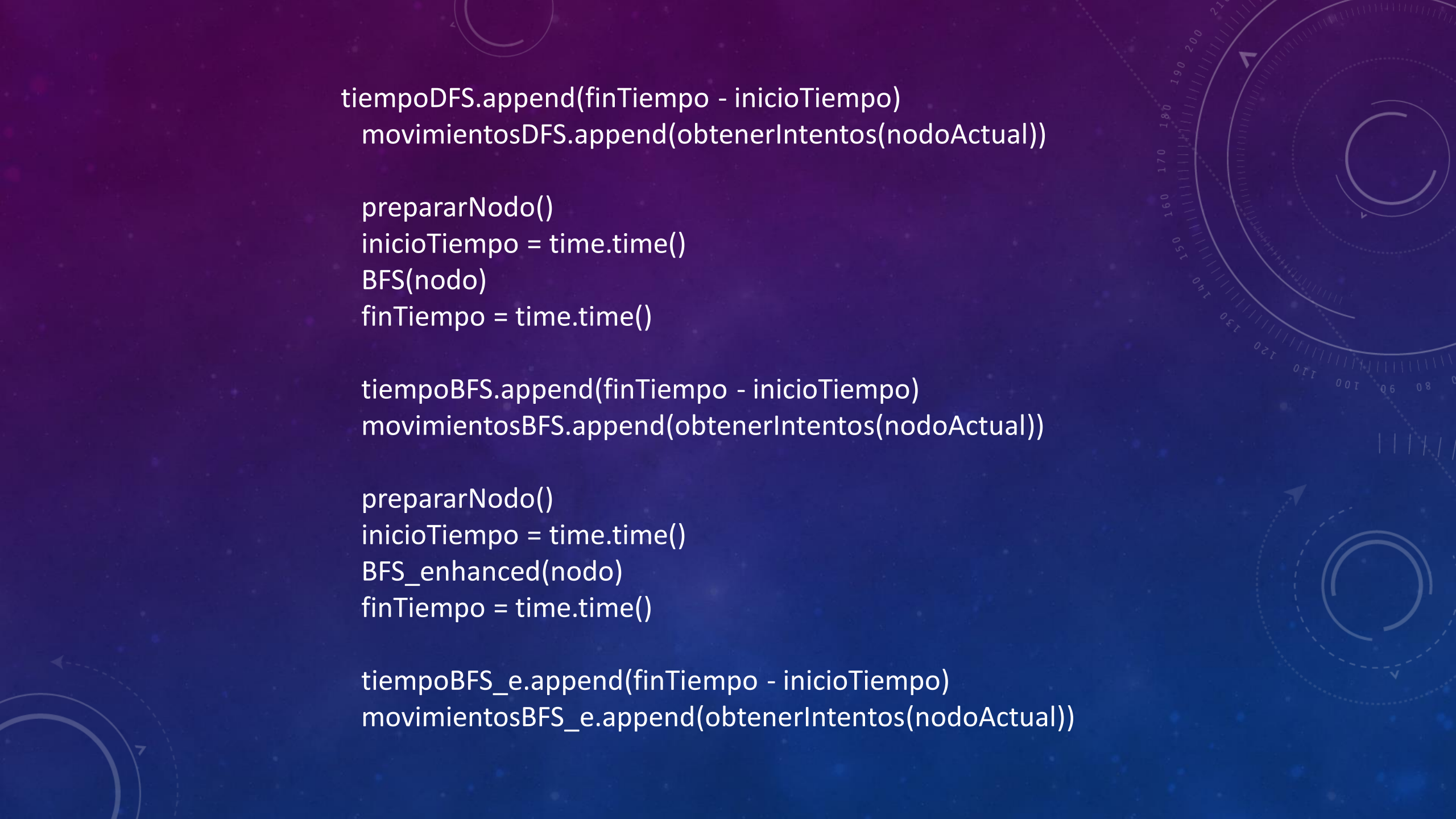
```
numeroNodo = 1  
nodo = Nodo()  
nodo.estado = estadoInicial  
nodo.numeroNodo = numeroNodo  
padresList = [nodo]  
hijosList = []
```

```
nivel = 1
```

```
padresList = [nodo]  
hijosList = []
```

```
nodoActual = Nodo()
```

```
prepararNodo()  
inicioTiempo = time.time()  
DFS(nodo)  
finTiempo = time.time()
```



```
tiempoDFS.append(finTiempo - inicioTiempo)
movimientosDFS.append(obtenerIntentos(nodoActual))
```

```
prepararNodo()
inicioTiempo = time.time()
BFS(nodo)
finTiempo = time.time()
```

```
tiempoBFS.append(finTiempo - inicioTiempo)
movimientosBFS.append(obtenerIntentos(nodoActual))
```

```
prepararNodo()
inicioTiempo = time.time()
BFS_enhanced(nodo)
finTiempo = time.time()
```

```
tiempoBFS_e.append(finTiempo - inicioTiempo)
movimientosBFS_e.append(obtenerIntentos(nodoActual))
```

```
csv = {'Estado Inicial': estadosList,  
      'Tiempo DFS': tiempoDFS,  
      'Movimientos DFS': movimientosDFS,  
      'Tiempo BFS': tiempoBFS,  
      'Movimientos BFS': movimientosBFS,  
      'Tiempo BFS_e': tiempoBFS_e,  
      'Movimientos BFS_e': movimientosBFS_e}
```

```
df = pandas.DataFrame(csv, columns=['Estado Inicial',  
                                   'Tiempo DFS',  
                                   'Movimientos DFS',  
                                   'Tiempo BFS',  
                                   'Movimientos BFS',  
                                   'Tiempo BFS_e',  
                                   'Movimientos BFS_e'])
```

```
imprimir("\n"*10)
```

```
imprimir(df)
```

```
df.to_csv(r'plot.csv', index = False, header=True)
```

```
mostrar("\nPresiona cualquier tecla para continuar...")
```

**¡GRACIAS POR
SU ATENCION!**