Título del Trabajo

Tú Nombre

March 7, 2024

1 Introducción

A lo largo de este cuaderno podrá encontrar:

- En primer lugar, una función que le permitirá introducir la configuración inicial y final que desee. Más allá del propio juego podrá poner como final e inicial cualquiera sin necesidad de estar en formato de torre, aunque recuerde que en ningún caso un disco mayor estará sobre uno de menor tamaño.
- Después, encontrará desglosado en apartados los requerimientos de la propia práctica, así como los algoritmos por separado en su respectivo orden.

Clase:

```
# Librer as
import time
from collections import deque
import copy
from queue import PriorityQueue

# Clase para el problema
class Hanoi:
    def __init__(self, e_i, e_f, esp):
        self.e_ini = e_i
        self.e_fin = e_f
        self.espacio_estados = esp

    def __es_final(self, estado):
        return self.e_fin == estado

    def __expandir_amplitud(self, estado):
        sucesores = deque()
        for origen in range(3):
            for destino in range(3):
                if origen != destino and estado[origen] and (
                        not estado[destino] or estado[destino][-1]
    > estado[origen][-1]):
                    nuevo = copy.deepcopy(estado)
                    nuevo[destino].append(nuevo[origen].pop())
                    sucesores.append(nuevo)

        return sucesores

    def __expandir_dfs(self, estado):
        vecinos = []
        for origen in range(3):
            for destino in range(3):
                if origen != destino and estado[origen] and (
```

```
                    not estado[destino] or estado[destino][-1]
> estado[origen][-1]):
                nuevo = copy.deepcopy(estado)
                nuevo[destino].append(nuevo[origen].pop())
                vecinos.append(nuevo)
        return vecinos

    def __expandir_heuristica(self, estado):
        sucesores = []
        for i in range(3):
            for j in range(3):
                if i != j and (estado[i] and (not estado[j] or
estado[i][-1] < estado[j][-1])):
                nuevo_estado = [list(torre) for torre in estado
]
                if nuevo_estado[i]:  # Verificar si la lista no
 est  vac a antes de llamar a pop()
                    nuevo_estado[j].append(nuevo_estado[i].pop
())
                    sucesores.append((nuevo_estado, self.
__calcular_distancia_hamming(nuevo_estado)))
        return sucesores

    def __calcular_distancia_hamming(self, estado):
        distancia_hamming = 0
        for torre_actual, torre_objetivo in zip(estado, self.e_fin)
:
            for disco_actual, disco_objetivo in zip(torre_actual,
torre_objetivo):
                if disco_actual != disco_objetivo:
                    distancia_hamming += 1
        return distancia_hamming

    def bfs(self):
        t_ini = time.time()
        visitados = []
        q = deque([self.e_ini])

        while q and len(visitados) <= self.espacio_estados:
            actual = q.popleft()
            if actual not in visitados:
                visitados.append(actual)
                if self.__es_final(actual):
                    t_fin = time.time()
                    return visitados, t_fin-t_ini
                q.extend(vecino for vecino in self.
__expandir_amplitud(actual) if (vecino not in visitados))

        return [], 0

    def dfs(self):
        t_ini = time.time()
        visitados = []

        stack = [self.e_ini]

        while stack and len(visitados) <= self.espacio_estados:
```

```
            actual = stack.pop()
            if actual not in visitados:
                visitados.append(actual)
                if actual == self.e_fin:
                    t_fin = time.time()
                    return visitados, t_fin - t_ini
                vecinos = self.__expandir_dfs(actual)
                stack.extend(vecino for vecino in reversed(vecinos)
     if vecino not in visitados)

        return [], 0

    def heuristica_estatica(self):
        t_ini = time.time()
        frontera = PriorityQueue()
        frontera.put((0, self.e_ini))
        padres = {tuple(map(tuple, self.e_ini)): None}
        costo_camino = {tuple(map(tuple, self.e_ini)): 0}
        cont = 1

        while not frontera.empty():
            _, estado_actual = frontera.get()

            if self.__es_final(estado_actual):
                camino = []
                while estado_actual:
                    camino.append(estado_actual)
                    estado_actual = padres[tuple(map(tuple,
    estado_actual))]
                t_fin = time.time()
                return camino[::-1], t_fin-t_ini

            for sucesor, distancia in self.__expandir_heuristica(
    estado_actual):
                nuevo_costo = costo_camino[tuple(map(tuple,
    estado_actual))] + 1
                if tuple(map(tuple, sucesor)) not in costo_camino
    or nuevo_costo < costo_camino[tuple(map(tuple, sucesor))]:
                    costo_camino[tuple(map(tuple, sucesor))] =
    nuevo_costo
                    prioridad = nuevo_costo + distancia
                    frontera.put((prioridad, sucesor))
                    padres[tuple(map(tuple, sucesor))] =
    estado_actual
        return [], 0


def pedir_valores():
    ini = [[], [], []]
    fin = [[], [], []]
    num_discos = int(input("Introduzca el n mero de discos: "))
    if num_discos > 0:
        print("\nAhora, establezca el ESTADO INICIAL s_0:")

        for disco in reversed(range(1, num_discos+1)):
            torre = int(input(f"Introduce la torre (0, 1 o 2) en la
     que quieres apilar el disco {disco}: "))
```

```
131
132             while torre != 0 and torre != 1 and torre != 2:
133                 torre = int(input(f"Por favor, introduce 0, 1 o 2
    para introducir el disco {disco} en dicha torre: "))
134
135             ini[torre].append(disco)
136
137         print("\nAhora, establezca el ESTADO META s_f:")
138
139         for disco in reversed(range(1, num_discos+1)):
140             torre = int(input(f"Introduce la torre (0, 1 o 2) en la
     que quieres apilar el disco {disco}: "))
141
142             while torre != 0 and torre != 1 and torre != 2:
143                 torre = int(input(f"Por favor, introduce 0, 1 o 2
    para introducir el disco {disco} en dicha torre: "))
144
145             fin[torre].append(disco)
146
147         esp = int(input("\nAhora, introduce el tama o m ximo que
    podr  tener el espacio de estados: "))
148
149     else:
150         print("El n mero de discos debe ser mayor que 0.")
151         return None, None, 0
152
153     return ini, fin, esp
```

Playground:

```
1  # C digo principal
2  e_ini, e_fin, esp = pedir_valores()
3  hanoi = Hanoi(e_i=e_ini, e_f=e_fin, esp=esp)
4  if e_ini and e_fin:
5      sol_bfs, time_bfs = hanoi.bfs()
6      sol_dfs, time_dfs = hanoi.dfs()
7      sol_star, time_star = hanoi.heuristica_estatica()
8
9      print('\n')
10     print("Algoritmo de B squeda en Amplitud:")
11     if sol_bfs:
12         for idx_bfs, paso_bfs in enumerate(sol_bfs):
13             print(f"Paso {idx_bfs+1}: {paso_bfs}")
14         print(f"TIEMPO DE EJECUCI N: {time_bfs}")
15     else:
16         print("No se ha encontrado una soluci n con el tama o
    m ximo asignado al conjunto de estados")
17
18     print('\n')
19     print("Algoritmo de B squeda en Profundidad:")
20     if sol_dfs:
21         for idx, paso_dfs in enumerate(sol_dfs):
22             print(f"Paso {idx+1}: {paso_dfs}")
23         print(f"TIEMPO DE EJECUCI N: {time_dfs}")
24     else:
```

```
25          print("No se ha encontrado una soluci n con el tama o
    m ximo asignado al conjunto de estados")
26
27      print('\n')
28      print("Algoritmo de B squeda A*:")
29      if sol_star and (len(sol_star) <= esp):
30          for idx_star, paso_star in enumerate(sol_star):
31              print(f"Paso {idx_star+1}: {paso_star}")
32          print(f"TIEMPO DE EJECUCI N: {time_star}")
33      else:
34          print("No se ha encontrado una soluci n con el tama o
    m ximo asignado al conjunto de estados")
```

2 Desarrollo

En esta sección puedes desarrollar los puntos principales de tu trabajo.

a) Descripción del problema como una búsqueda en un espacio de estados

- **Estado inicial s_0:** El estado inicial es elegido por el usuario, aunque típicamente es aquel en el que todos los discos (D_1, D_2 ,..., D_n) se encuentran apilados de mayor a menor tamaño en la aguja *A*.
- **Estado meta s_f:** El estado meta es aquel en el que todos los discos están apilados en el orden de tamaños (de abajo a arriba) en la aguja *B* o la aguja *C*: n, $n-1$, ..., 1.
- **Operadores:** Mover un disco de una aguja a otra.
- **Restricciones:**
  - Un disco nunca puede reposar encima de otro de menor tamaño (es decir, los discos siempre estarán colocados de la forma $(D_1, D_2, ..., D_n)$).
  - Cuando se proceda a mover un disco de una aguja a otra, solo se podrán mover aquellos que se encuentren en la cima de la aguja (es decir, cada aguja corresponde a una estructura LIFO).

b) Descripción de los operadores que se pueden aplicar para la función expandir(nodos)

Descripción:

1. Seleccionar una torre de origen (de las tres torres disponibles: A, B, C).
2. Seleccionar una torre de destino (distinta de la torre de origen y que cumpla con las reglas del juego, es decir, el disco a mover debe ser más pequeño que el disco en la cima de la torre de destino).

3. Verificar que la torre de origen no esté vacía y que la torre de destino acepte el disco a mover según las reglas del juego.
4. Extraer el disco superior de la torre de origen.
5. Colocar el disco extraído en la cima de la torre de destino.
6. Generar una nueva configuración del juego después de aplicar este movimiento.
7. Agregar la nueva configuración a la lista de nodos expandidos.

c) Descripción del conjunto de los estados posibles

El conjunto de estado se define por la disposición de los discos en las tres agujas (A, B y C). Se puede describir de la siguiente manera:

- **Estado inicial:** Todos los discos están apilados en orden decreciente de tamaño en la aguja A, mientras que las agujas B y C están vacías.
- **Estados intermedios:** Durante el proceso de mover los discos de una aguja a otra, se generan múltiples estados intermedios donde los discos se encuentran distribuidos entre las tres agujas, manteniendo siempre la condición de que ningún disco más grande esté sobre uno más pequeño.
- **Estado final:** El estado final se alcanza cuando todos los discos han sido transferidos a la aguja B o C, manteniendo el orden decreciente de tamaño en cada una de ellas.
- **Estados inválidos:** Cualquier estado donde un disco más grande esté sobre uno más pequeño sería un estado inválido.

d) Algoritmo de Búsqueda en amplitud

```python
def expandir_bfs(estado):
    vecinos = deque()
    for origen in range(3):
        for destino in range(3):
            if origen != destino and estado[origen] and (
                    not estado[destino] or estado[destino
    ][-1] > estado[origen][-1]):
                nuevo = copy.deepcopy(estado)
                nuevo[destino].append(nuevo[origen].pop())
                vecinos.append(nuevo)
    return vecinos

def hanoi_bfs(e_ini, e_fin):
    t_ini = time.time()
    visitados = []
    q = deque([e_ini])

    while q:
        actual = q.popleft()
        if actual not in visitados:
            visitados.append(actual)
            if actual == e_fin:
                t_fin = time.time()
                return visitados, t_fin-t_ini
            q.extend(vecino for vecino in expandir_bfs(actual
    ) if (vecino not in visitados))

    return [], 0

configs_ini = []
configs_fin = []

# Configuraciones inicial y objetivo
configuracion_inicial = [[3, 2, 1], [], []]
configuracion_objetivo = [[], [], [3, 2, 1]]

path, t = hanoi_bfs(e_ini=configuracion_inicial, e_fin=
    configuracion_objetivo)

if path:
    for idx, item in enumerate(path):
        print(f"Paso {idx+1}: {item}")
    print(f"\nTIEMPO: {t}")
```

e) Algoritmo de Búsqueda en Profundidad

```python
def expandir_dfs(estado):
    vecinos = []
    for origen in range(3):
        for destino in range(3):
            if origen != destino and estado[origen] and (
                    not estado[destino] or estado[destino
    ][-1] > estado[origen][-1]):
                nuevo = copy.deepcopy(estado)
                nuevo[destino].append(nuevo[origen].pop())
                vecinos.append(nuevo)
    return vecinos

def hanoi_dfs(e_ini, e_fin):
    t_ini = time.time()
    visitados = []

    stack = [e_ini]

    while stack:
        actual = stack.pop()
        if actual not in visitados:
            visitados.append(actual)
            if actual == e_fin:
                t_fin = time.time()
                return visitados, t_fin - t_ini
            vecinos = expandir_dfs(actual)
            stack.extend(vecino for vecino in reversed(
    vecinos) if vecino not in visitados)

    return [], 0

configs_ini = []
configs_fin = []

# Configuraciones inicial y objetivo
configuracion_inicial = [[3, 2, 1], [], []]
configuracion_objetivo = [[], [], [3, 2, 1]]

path, t = hanoi_dfs(e_ini=configuracion_inicial, e_fin=
    configuracion_objetivo)

if path:
    for idx, item in enumerate(path):
        print(f"Paso {idx+1}: {item}")
    print(f"\nTIEMPO: {t}")
```

f) Proponer y describir una heurística

Distancia de Hamming: contar los discos que se encuentran en diferentes posiciones entre la configuración inicial y la final e incrementar en uno el coste por cada disco que se encuentre en una posición diferente entre ambas configu-

raciones.

Descripción:

1. Calcular la distancia de Hamming entre la configuración actual y la configuración final del problema.

2. Cuantos más discos estén en diferentes posiciones, mayor será la distancia de Hamming y mayor será la estimación de la cantidad de movimientos necesarios.

3. La heurística elegirá la configuración de expansión que minimice la distancia de Hamming, es decir, que más se acerque a la configuración objetivo.

g) Programa en Python de la búsqueda de la solución usando el algoritmo de A* con la heurística propuesta

```
1  def calcular_distancia_hamming(configuracion_actual,
       configuracion_objetivo):
2      distancia_hamming = 0
3      for torre_actual, torre_objetivo in zip(configuracion_actual,
       configuracion_objetivo):
4          for disco_actual, disco_objetivo in zip(torre_actual,
       torre_objetivo):
5              if disco_actual != disco_objetivo:
6                  distancia_hamming += 1
7      return distancia_hamming
8
9  def esFinal(configuracion, configuracion_objetivo):
10     return configuracion == configuracion_objetivo
11
12 def expandir(configuracion, configuracion_objetivo):
13     sucesores = []
14     for i in range(3):
15         for j in range(3):
16             if i != j and (configuracion[i] and (not configuracion[
       j] or configuracion[i][-1] < configuracion[j][-1])):
17                 nuevo_estado = [list(torre) for torre in
       configuracion]
18                 if nuevo_estado[i]:
19                     nuevo_estado[j].append(nuevo_estado[i].pop())
20                     sucesores.append((nuevo_estado,
       calcular_distancia_hamming(nuevo_estado, configuracion_objetivo
       )))
21     return sucesores
22
23 def heuristicaEstatica(configuracion_inicial,
       configuracion_objetivo):
24     t_ini = time.time()
25     frontera = PriorityQueue()
26     frontera.put((0, configuracion_inicial))
27     padres = {tuple(map(tuple, configuracion_inicial)): None}
28     costo_camino = {tuple(map(tuple, configuracion_inicial)): 0}
29     contador = 0
30
```

```
    while not frontera.empty():
        _, estado_actual = frontera.get()

        if esFinal(estado_actual, configuracion_objetivo):
            camino = []
            while estado_actual:
                camino.append(estado_actual)
                estado_actual = padres[tuple(map(tuple, estado_actual))]
                t_fin = time.time()
            return camino[::-1], t_fin-t_ini

        for sucesor, distancia in expandir(estado_actual, configuracion_objetivo):
            nuevo_costo = costo_camino[tuple(map(tuple, estado_actual))] + 1
            if tuple(map(tuple, sucesor)) not in costo_camino or nuevo_costo < costo_camino[tuple(map(tuple, sucesor))]:
                costo_camino[tuple(map(tuple, sucesor))] = nuevo_costo
                prioridad = nuevo_costo + distancia
                frontera.put((prioridad, sucesor))
                padres[tuple(map(tuple, sucesor))] = estado_actual
```

3 Conclusiones

Puedes escribir tus conclusiones aquí.

4 Referencias

Aquí van las referencias bibliográficas utilizadas en tu trabajo.

5 Imágenes

A continuación se muestra un ejemplo de cómo insertar una imagen en tu documento.

Figure 1: Descripción de la imagen.