1) Diseñar una solución para el problema de colaración de grafos utilizando *backtracking*. Se supondrá que el grafo consta de `n` nodos y que utilizamos una representación con matriz de adyacencia `A[1..n][1..n]` de booleanos. En lugar de colores se asignarán etiquetas numéricas a los nodos $1, 2, 3 \ldots$ Una solución será representada como una tupla $S = (s_1, s_2, \ldots, sn)$ donde $s_i$, que puede tomar valores $1, 2, 3 \ldots$ representa el color asignado al nodo `i`.

1) Descripción del algoritmo

   El algoritmo explora todas las posibles asignaciones de colores a los nodos del grafo siguiendo estas reglas:

   - Se asigna un color a cada nodo en orden secuencial.
   - Antes de asignar un color a un nodo, se verifica si este es **compatible** (es decir, no se asigna un color que ya esté presente en los nodos adyacentes).
   - Si se alcanza una asignación completa y válida para todos los nodos, se reporta como solución.

   El **backtracking** asegura que si se asigna un color no válido en algún caso, se retrocede y se intenta otro color.

2) Representación de los datos

   1) La matriz de adyacencia `A[1..n][1..n]` representará el grafo.
      - `A[i][j]=True` si hay una arista entre `i` y `j`, y `False` en caso contrario.
   2) Un arreglo $S = [s_1, s_2, \ldots, s_n]$ almacenará el color (número) asignado a cada nodo.

3) Funciones clave del algoritmo

   1) `es_valido`**:** Verifica si asignar un color `c` al nodo `v` es válido (no está en conflicto con los nodos adyacentes).
   2) `backtracking`**:** Explora las asignaciones de colores a los nodos de manera recursiva.
   3) `solucionar_coloreo`**:** Inicia el proceso y muestra el resultado final.

4) Implementación en Python

```python
def es_valido(nodo, color, solucion, A, n):
    """
    Verifica si el color asignado al nodo es válido.
    - nodo: El nodo actual
    - color: Color que se intenta asignar al nodo.
    - solucion: Lista de colores asignados hasta el momento.
    - A: Matriz de adyacencia.
    - n: Número nodos en el grafo.
    """
    for vecino in range(n):
        if A[nodo][vecino] and solucion[vecino] == color:  # Verifica adyacencia y
            conflicto de colores
             return False
    return True

def backtracking(nodo, A, n, max_color, solucion):
    """
    Aplica backtracking para asignar colores a los nodos.
    - nodo: Nodo actual a colorear
    - A: Matriz de adyacencia.
    - n: Número de nodos.
    - max_color: Número máximo de colores permitidos.
    - solucion: Lista con los colores asignados.
    """
    if nodo == n: # Caso base: Todos los nodos han sido coloreados
        return True
```

```
    for color in range(1, max_color + 1): # Intenta con todos los colores posibles
        if es_valido(nodo, color, solucion, A, n):
            solucion[nodo] = color # Asigna el color

            if backtracking(nodo + 1, A, n, max_color, solucion): # Llamada recursiva
                return True

            solucion[nodo] = 0 # Backtracking: Desasignar el color

    return False

def solucionar_coloreo(A, n, max_color):
    """
    Resuelve el problema de coloración de grafos.
    - A: Matriz de adyacencia.
    - n: Número de nodos.
    - max_color: Número máximo de colores permitidos.
    """
    solucion = [0] * n # Inicializar solución vacía
    if backtracking(0, A, n, max_color, solucion):
        print("Solución encontrada:")
        print(solucion)

# Ejemplo de uso:
if __name__ == '__main__':
    n = 4 # Número de nodos
    A = [
        [False, True, True, True],
        [True, False, True, False],
        [True, True, False, True],
        [True, False, True, False]
    ]
    max_color = 3 # Número máximo de colores permitidos

    solucionar_coloreo(A, n, max_color)
```

5) Explicación del código

   1) `es_valido`:

      - Comprueba si asignar un color al nodo actual no entra en conflicto con los colores de los nodos adyacentes.

   2) `backtracking`:

      - Recursivamente asigna colores a los nodos.
      - Si encuentra una asignación válida para todos los nodos, termina.
      - De lo contrario, realiza un **backtrack** (deshace una asignación y prueba otra opción).

   3) `solucionar_coloreo`:

      - Es la función principal que inicializa el proceso y muestra el resultado.

6) Ejemplo de salida

   Para la matriz de adyacencia del ejemplo y con un máximo de **3 colores**, la salida es:

```
## Solución encontrada:
## [1,2,3,2]
```

7) Complejidad Temporal La complejidad del algoritmo es $O(m^n)$, donde $n$ es el número de nodos y $m$ el número máximo de colores. Aunque esta complejidad es exponencial en el peor caso, el backtracking con poda reduce el tiempo considerablemente en la práctica.

**2)** Diseñar una solución para el problema del ciclo hamiltoniano utilizando *backtracking*.

1) Problema del Ciclo Hamiltoniano

Un **ciclo hamiltoniano** es un recorrido que pasa **exactamente una vez** por todos los vértices de un grafo y regresa al vértice inicial. El objetivo del algoritmo es determinar si existe tal ciclo en un grafo dado y mostrar la secuencia de vértices que forma el ciclo.

2) Implementación en Python

```python
def es_valido(v, grafo, camino, pos):
    """
    Verifica si el vértice v puede añadirse al camino actual

    :param v: Vértice actual.
    :param grafo: Matriz de adyacencia que representa el grafo.
    :param camino: Lista con el camino actual.
    :param pos: Posición actual en el camino.
    """
    # Verifica si existe una arista desde el último vértice al vértice actual
    if not grafo[camino[pos - 1]][v]:
        return False

    # Verifica si el vértice ya ha sido visitado
    if v in camino:
        return False

    return True

def backtracking_hamiltoniano(grafo, camino, pos, n):
    """
    Aplica backtracking para encontrar un ciclo hamiltoniano

    :param grafo: Matriz de adyacencia.
    :param camino: Lista con el camino actual.
    :param pos: Posición actual en el camino.
    :param n: Número de vértices en el grafo.
    """
    # Caso base: si todos los vértices están en el camino
    if pos == n:
        # Verifica si existe un arista entre el último y el primer vértice
        if grafo[camino[pos - 1]][camino[0]]:
            return True
        else:
            return False

    # Prueba a añadir vértices al camino
    for v in range(1, n):
        if es_valido(v, grafo, camino, pos):
```

```
            camino[pos] = v # Añade el vértice al camino

            if backtracking_hamiltoniano(grafo, camino, pos + 1, n):
                return True

            # Backtrack: elimina el vértice del camino
            camino[pos] = -1

    return False

def encontrar_ciclo_hamiltoniano(grafo, n):
    """
    Resuelve el problema del ciclo hamiltoniano.

    :param grafo: Matriz de adyacencia.
    :param n: Número de vértices en el grafo.
    """
    camino = [-1] * n # Inicializa el camino
    camino[0] = 0 # Comienza en el vértice 0

    if backtracking_hamiltoniano(grafo, camino, 1, n):
        print("Ciclo Hamiltoniano encontrado:")
        print(camino + [camino[0]]) # Muestra el ciclo completo
    else:
        print("Ciclo Hamiltoniano no encontrado")

# Ejemplo de uso:
if __name__ == "__main__":
    n = 5 # Número de vértices
    grafo = [
        [0,1,0,1,0],
        [1,0,1,1,1],
        [0,1,0,0,1],
        [1,1,0,0,1],
        [0,1,1,1,0]
    ]

    encontrar_ciclo_hamiltoniano(grafo, n)
```

3) Representación del grafo

El grafo se representa mediante una **matriz de adyacencia**, donde:

- `grafo[i][j]=1` significa que hay una arista entre los vértices `i` y `j`.
- `grafo[i][j]=0` indica que no hay aristas entre ellos.

4) Explicación de las funciones

1) `es_valido`:

```
def es_valido(v, grafo, camino, pos):
    """
    Verifica si el vértice v puede añadirse al camino actual

    :param v: Vértice actual.
    :param grafo: Matriz de adyacencia que representa el grafo.
```

```
    :param camino: Lista con el camino actual.
    :param pos: Posición actual en el camino.
    """
    # Verifica si existe una arista desde el último vértice al vértice actual
    if not grafo[camino[pos - 1]][v]:
        return False

    # Verifica si el vértice ya ha sido visitado
    if v in camino:
        return False

    return True
```

- **Propósito:** Verifica si es válido añadir un vértice `v` al camino actual.
- **Chequeos:**
  1) Comprueba si existe una arista entre el último vértice del camino `v`.
  2) Verifica que `v` no haya sido visitado antes.

2) `backtracking_hamiltoniano`:

```
def backtracking_hamiltoniano(grafo, camino, pos, n):
    """
    Aplica backtracking para encontrar un ciclo hamiltoniano

    :param grafo: Matriz de adyacencia.
    :param camino: Lista con el camino actual.
    :param pos: Posición actual en el camino.
    :param n: Número de vértices en el grafo.
    """
    # Caso base: si todos los vértices están en el camino
    if pos == n:
        # Verifica si existe un arista entre el último y el primer vértice
        if grafo[camino[pos - 1]][camino[0]]:
            return True
        else:
            return False

    # Prueba a añadir vértices al camino
    for v in range(1, n):
        if es_valido(v, grafo, camino, pos):
            camino[pos] = v # Añade el vértice al camino

            if backtracking_hamiltoniano(grafo, camino, pos + 1, n):
                return True

            # Backtrack: elimina el vértice del camino
            camino[pos] = -1

    return False
```

- **Propósito:** Aplica **backtracking** para construir el camino hamiltoniano paso a paso.
- **Lógica:**

1) Si se han añadido todos los vértices (`pos==n`), verifica si existe una arista que conecte el último vértice con el inicial.

2) Prueba todos los vértices desde `1` hasta `n-1`:

- Si el vértice es válido (`es_valido`), lo añade al camino y llama recursivamente.
- Si no encuentra una solución, elimina el vértice (**backtrack**).

3) `encontrar_ciclo_hamiltoniado`:

```
def encontrar_ciclo_hamiltoniano(grafo, n):
    """
    Resuelve el problema del ciclo hamiltoniano.

    :param grafo: Matriz de adyacencia.
    :param n: Número de vértices en el grafo.
    """
    camino = [-1] * n # Inicializa el camino
    camino[0] = 0 # Comienza en el vértice 0

    if backtracking_hamiltoniano(grafo, camino, 1, n):
        print("Ciclo Hamiltoniano encontrado:")
        print(camino + [camino[0]]) # Muestra el ciclo completo
    else:
        print("Ciclo Hamiltoniano no encontrado")
```

- **Propósito:** Es la función principal que incializa el proceso.
- **Lógica:**

1) El camino empieza desde el vértice 0.

2) LLama a `backtracking_hamiltoniano` para encontrar el ciclo.

3) Si se encuentra una solución, muestra el camino junto con el vértice inicial al cerrar el ciclo.

5) Ejemplo de uso

Para el siguiente grafo de 5 vértices:

```
if __name__ == "__main__":
    n = 5 # Número de vértices
    grafo = [
        [0,1,0,1,0],
        [1,0,1,1,1],
        [0,1,0,0,1],
        [1,1,0,0,1],
        [0,1,1,1,0]
    ]
```

- El grafo tiene las siguientes aristas:
  - $0 \longleftrightarrow 1, 0 \longleftrightarrow 3$
  - $0 \longleftrightarrow 2, 1 \longleftrightarrow 3, 1 \longleftrightarrow 4$
  - $2 \longleftrightarrow 4$
  - $3 \longleftrightarrow 4$

Salida del programa

```
## Ciclo Hamiltoniano encontrado:
## [0, 1, 2, 4, 3, 0]
```

El ciclo recorre los vértice en el orden: $0 \to 1 \to 2 \to 4 \to 3 \to 0$, cumpliendo las condiciones del ciclo hamiltoniano.

6) Complejidad del algoritmo

- **Tiempo:** $O(n!)$, ya que prueban todas las permutaciones posibles de los $n$ vértices.

**3)** En un matriz cuadrada `M`, de tamaño $n \times n$ representamos un laberinto. Partimos de la posición $(1, 1)$ y el objetivo es moverse en la posición $(n, n)$. Podemos pasar por la casilla $(i, j)$ si y sólo si `M[i,j]=A` (abierta). Si `M[i,j]=C` (cerrada) entonces no podemos pasar por esa casilla. Desde cada casilla existen 4 posibles movimientos: arriba, abajo, izquierda y derecha.

| | | | |
|---|---|---|---|
| A | C | A | A |
| A | A | A | C |
| A | C | A | A |
| C | A | A | A |

Describe la forma de resolver el problema utilizando *backtracking*. Pista: ten en cuenta que en cada movimiento sólo necesitamos conocer la posición actual.

1) Descripción del problema

Dado un **laberinto** representado por una matriz cuadrada `M` de tamaño $n \times n$:

- Cada celda puede estar **abierta** (`A`) o **cerrada** (`C`).
- Se parte de la celda $(0, 0)$ (esquina superior izquierda).
- El objetivo es llega a la celda `(n-1,n-1)` (esquina inferior derecha) moviéndose en las **4 direcciones posibles**:
  - **Abajo:** `(x+1,y )`
  - **Derecha:** `(x,y+1)`
  - **Arriba:** `(x-1,y)`
  - **Izquierda:** `(x,y-1)`

2) Implementación en Python

```python
def es_valido(x, y, M, n, visitado):
    """
    Verifica si la posición (x, y) es válida para moverse.
    - x, y: Coordenadas actuales.
    - M: Matriz del laberinto.
    - n: Tamaño de la matriz.
    - visitado: Matriz que guarda las posiciones ya visitadas.
    """
    return 0 <= x < n and 0 <= y < n and M[x][y] == 'A' and not visitado[x][y]

def backtracking_laberinto(x, y, M, n, visitado, camino):
    """
    Aplica backtracking para encontrar un camino en el laberinto.
    - x, y: Posición actual.
    - M: Matriz del laberinto.
    - n: Tamaño de la matriz.
    - visitado: Matriz que guarda las posiciones ya visitadas.
```

```
    - camino: Lista que almacena el camino recorrido.
    """
    # Caso base: si llegamos a la posición final (n-1, n-1)
    if x == n - 1 and y == n - 1:
        camino.append((x, y))
        return True

    # Añadir la posición actual al camino y marcar como visitada
    camino.append((x, y))
    visitado[x][y] = True

    # Definir los 4 movimientos posibles: abajo, derecha, arriba, izquierda
    movimientos = [(1, 0), (0, 1), (-1, 0), (0, -1)]

    # Probar todos los movimientos posibles
    for dx, dy in movimientos:
        nuevo_x, nuevo_y = x + dx, y + dy
        if es_valido(nuevo_x, nuevo_y, M, n, visitado):
            if backtracking_laberinto(nuevo_x, nuevo_y, M, n, visitado, camino):
                return True

    # Backtrack: desmarcar la posición y eliminarla del camino
    visitado[x][y] = False
    camino.pop()
    return False

def resolver_laberinto(M, n):
    """
    Resuelve el problema del laberinto usando backtracking.
    - M: Matriz del laberinto.
    - n: Tamaño de la matriz.
    """
    visitado = [[False for _ in range(n)] for _ in range(n)]  # Matriz de posiciones
        visitadas
    camino = []  # Lista para almacenar el camino

    if backtracking_laberinto(0, 0, M, n, visitado, camino):
        print("Camino encontrado:")
        for paso in camino:
            print(paso)
    else:
        print("No se encontró un camino válido.")

# Ejemplo de uso
if __name__ == "__main__":
    M = [
        ['A', 'C', 'A', 'A'],
        ['A', 'A', 'A', 'C'],
        ['A', 'C', 'A', 'A'],
        ['C', 'A', 'A', 'A']
    ]
    n = len(M)
    resolver_laberinto(M, n)
```

3) Estrategia del algoritmo

   1) Empezamos desde la posición inicial $(0, 0)$ y seguimos probando movimientos válidos.

   2) En cada celda, realizamos los siguientes pasos:

      - Marcamos la celda como **visitada**.
      - Intentamos movernos a una de las **4 direcciones** posibles.
      - Si llegamos a la celda destino `(n-1, n-1)`, hemos encontrado un camino válido.

   3) Si un movimiento no lleva a un solución (camino sin salida), retrocedemos (**backtrack**) y probamos con otra dirección.

   4) El proceso se repite hasta que se encuentra un camino válido o se verifica que no hay solución.

4) Explicación de las funciones

   1) `es_valido`**:**

```python
def es_valido(x, y, M, n, visitado):
    """
    Verifica si la posición (x, y) es válida para moverse.
    - x, y: Coordenadas actuales.
    - M: Matriz del laberinto.
    - n: Tamaño de la matriz.
    - visitado: Matriz que guarda las posiciones ya visitadas.
    """
    return 0 <= x < n and 0 <= y < n and M[x][y] == 'A' and not visitado[x][y]
```

      - **Propósito:** Verificar si una celda `(x, y)` es válida para moverse.
      - **Condiciones:**

        1) La celda está dentro de los límites de la matriz.
        2) La celda está **abierta** `(M[x][y] == "A")`.
        3) La celda **no ha sido visitada** aún.

   2) `backtracking_laberinto`**:**

```python
def backtracking_laberinto(x, y, M, n, visitado, camino):
    """
    Aplica backtracking para encontrar un camino en el laberinto.
    - x, y: Posición actual.
    - M: Matriz del laberinto.
    - n: Tamaño de la matriz.
    - visitado: Matriz que guarda las posiciones ya visitadas.
    - camino: Lista que almacena el camino recorrido.
    """
    # Caso base: si llegamos a la posición final (n-1, n-1)
    if x == n - 1 and y == n - 1:
        camino.append((x, y))
        return True

    # Añadir la posición actual al camino y marcar como visitada
    camino.append((x, y))
    visitado[x][y] = True

    # Definir los 4 movimientos posibles: abajo, derecha, arriba, izquierda
```

```
    movimientos = [(1, 0), (0, 1), (-1, 0), (0, -1)]

    # Probar todos los movimientos posibles
    for dx, dy in movimientos:
        nuevo_x, nuevo_y = x + dx, y + dy
        if es_valido(nuevo_x, nuevo_y, M, n, visitado):
            if backtracking_laberinto(nuevo_x, nuevo_y, M, n, visitado, camino):
                return True

    # Backtrack: desmarcar la posición y eliminarla del camino
    visitado[x][y] = False
    camino.pop()
    return False
```

- **Lógica:**

  1) Si estamos en la posición final `(n-1, n-1)`, añadimos la celda al camino y retornamos `True`.

  2) Marcamos la celda actual como visitada y la añadimos al camino.

  3) Probamos los **4 movimientos posibles**:

     - Si el movimiento es válido, hacemos una llamada recursiva.
     - Si encontramos un camino válido, retornamos `True`.

  4) Si ningún movimiento es válido, **retrocedemos**:

     - Desmarcamos la celda como visitada.
     - Eliminamos la celda del camino.

3) `resolver_laberinto`**:**

```
def resolver_laberinto(M, n):
    """
    Resuelve el problema del laberinto usando backtracking.
    - M: Matriz del laberinto.
    - n: Tamaño de la matriz.
    """
    visitado = [[False for _ in range(n)] for _ in range(n)]  # Matriz de
        posiciones visitadas
    camino = []  # Lista para almacenar el camino

    if backtracking_laberinto(0, 0, M, n, visitado, camino):
        print("Camino encontrado:")
        for paso in camino:
            print(paso)
    else:
        print("No se encontró un camino válido.")
```

- **Propósito:** Inicia el proceso resolviendo el laberinto desde $(0, 0)$.
- **Resultado:**
  - Si se encuentra un camino válido, se imprime el camino.
  - Si no hay solución, se muestra un mensaje de error.

5) Ejemplo de uso:

**Entrada:**

```
M = [
    ['A', 'C', 'A', 'A'],
    ['A', 'A', 'A', 'C'],
    ['A', 'C', 'A', 'A'],
    ['C', 'A', 'A', 'A']
]
```

**Salida:**

```
## Camino encontrado:
## (0, 0)
## (1, 0)
## (1, 1)
## (1, 2)
## (2, 2)
## (3, 2)
## (3, 3)
```

6) **Tiempo:** $O(4^{n^2})$ en el peor caso, porque en cada celda probamos 4 direcciones posibles.

**4)** En una liga de fútbol participan `n` equipos (suponemos que `n` es par). En cada jornada se juegan $\frac{n}{2}$ partidos, que enfrentan a dos equipos, dirigidos por un árbitro. Existen `m` árbitros disponibles, siendo $m > \frac{n}{2}$. Cada equipo `i` valora cada árbitro `j` con una puntuación `P[i,j]` entre 0 y 10, indicando su preferencia por ese árbitro. un valor alto indica que le gusta el árbitro y un valor bajo que no le gusta. Si el árbitro y el equipo son de la misma región entonces `P[i,j]`$=-\infty$.
El objetivo es (para cada jornada contreta) asignar un árbitro distinto a cada partido, de manera que se maximice la puntuación total de los árbitros asignados, teniendo en cuenta las preferencias de todos los equipos.
Dar una solución óptima para el problema usando *backtracking*. Exponer cómo es la representación de la solución, cuál es el esquema que habría que utilizar, cuál la condición de fin y cómo son las soluciones del esquema (`Generar, MasHermanos, Criterio, Solucion...`).

1) Representación del problema

   - `n`: Número de equipo (par).
   - `m`: Número de árbitros disponibles $\left(\texttt{m} > \frac{\texttt{n}}{2}\right)$.
   - `P[i][j]`: Matriz $n \times m$ que representa la puntuación de preferencia del equipo `i` por el árbitro `j`:
     - `P[i][j]` alto: el equipo `i` prefiere al árbitro `j`.
     - `P[i][j]`$=-\infty$: el árbitro `j` no puede arbitrar al equipo `j` (misma región).

2) Representación de la solución

   - La solución se representa como una **tupla**:

   $$S = (a_1, a_2, \ldots, a_k)$$

   donde $a_k$ es el árbitro asignado al partido $k$, y $k = \frac{n}{2}$ es el número total de partidos por jornada.

3) Esquema de Backtracking

   - `Generar`:
     - En la posición $k$-ésima (el partido $k$), se intenta asignar un de los `m` **árbitros** disponibles que no haya sido asignado aún.
   - `MasHermanos`:

- Un árbitro `j` tiene **más hermanos** si existe otro árbitro candidato válido que no haya sido asignado.

- `Criterio:`
  - Se verifica que:
    1) El árbitro `j` no haya sido asignado a ningún partido anterior.
    2) El árbitro `j` no tiene preferencia $-\infty$ con los equipos que juegan en el partido `k`.
- `Solucion:`
  - La asignación se completa cuando se asignan árbitros a los $\mathbf{\frac{n}{2}}$ **partidos**.
- Función objetivo:
  - El objetivo es **maximizar la suma de puntuaciones** de los árbitros asignados según las preferencias de los equipos.

4) Condición de fin
   - La condición de fin se alcanza cuando:
     - Todos los partidos $k = 1, \ldots, \frac{n}{2}$ tienen asignado un árbitro válido.

5) Implementación del algoritmo

```python
import math

def es_valido(asignados, arbitro, partido, partidos, P):
    """
    Verifica si un árbitro puede ser asignado al partido actual.
    - asignados: Lista de árbitros asignados hasta ahora.
    - arbitro: Árbitro actual a evaluar.
    - partido: Índice del partido actual.
    - partidos: Lista de partidos con equipos enfrentados.
    - P: Matriz de puntuaciones de preferencias.
    """
    equipo1, equipo2 = partidos[partido]  # Equipos en el partido actual
    # Verificar si el árbitro ya fue asignado
    if arbitro in asignados:
        return False
    # Verificar que la preferencia no sea -infinito para ninguno de los equipos
    if P[equipo1][arbitro] == -math.inf or P[equipo2][arbitro] == -math.inf:
        return False
    return True

def backtracking_arbitros(partido, asignados, puntuacion_actual, max_puntuacion,
    partidos, P, solucion_actual,
                          mejor_solucion):
    """
    Aplica backtracking para asignar árbitros a partidos.
    - partido: Índice del partido actual.
    - asignados: Lista de árbitros ya asignados.
    - puntuacion_actual: Puntuación total de la asignación actual.
    - max_puntuacion: Valor máximo de la puntuación encontrada hasta ahora.
    - partidos: Lista de partidos con equipos enfrentados.
    - P: Matriz de puntuaciones de preferencias.
    - solucion_actual: Lista con la asignación actual.
    - mejor_solucion: Lista con la mejor asignación encontrada.
```

```
    """
    # Caso base: Se asignaron árbitros a todos los partidos
    if partido == len(partidos):
        if puntuacion_actual > max_puntuacion[0]:
            max_puntuacion[0] = puntuacion_actual
            mejor_solucion[:] = solucion_actual[:]
        return

    # Probar asignar cada árbitro disponible
    for arbitro in range(len(P[0])):
        if es_valido(asignados, arbitro, partido, partidos, P):
            # Actualizar asignación y puntuación
            equipo1, equipo2 = partidos[partido]
            nueva_puntuacion = puntuacion_actual + P[equipo1][arbitro] + P[equipo2][
                arbitro]
            asignados.append(arbitro)
            solucion_actual.append(arbitro)

            # Llamada recursiva al siguiente partido
            backtracking_arbitros(partido + 1, asignados, nueva_puntuacion,
                max_puntuacion, partidos, P,
                                  solucion_actual, mejor_solucion)

            # Backtrack: deshacer cambios
            asignados.pop()
            solucion_actual.pop()

def asignar_arbitros(n, m, P, partidos):
    """
    Encuentra la asignación óptima de árbitros para los partidos usando backtracking.
    - n: Número de equipos.
    - m: Número de árbitros.
    - P: Matriz de puntuaciones de preferencias.
    - partidos: Lista de partidos con equipos enfrentados.
    """
    max_puntuacion = [-math.inf]  # Puntuación máxima encontrada
    mejor_solucion = []  # Mejor asignación encontrada
    asignados = []  # Árbitros ya asignados
    solucion_actual = []  # Asignación actual

    backtracking_arbitros(0, asignados, 0, max_puntuacion, partidos, P, solucion_actual
        , mejor_solucion)

    print("Mejor puntuación total:", max_puntuacion[0])
    print("Asignación de árbitros a partidos:", mejor_solucion)

# Ejemplo de uso
if __name__ == "__main__":
    n = 4  # Número de equipos
    m = 3  # Número de árbitros
    P = [  # Matriz de preferencias
        [8, -math.inf, 7],
        [6, 9, -math.inf],
        [-math.inf, 7, 8],
```

```
        [7, 8, -math.inf]
    ]
    partidos = [(0, 1), (2, 3)]  # Lista de partidos (equipos enfrentados)
    asignar_arbitros(n, m, P, partidos)
```

6) Explicación del algoritmo

    1) Entrada:

        - `P:` Matriz de puntuaciones con preferencias de cada equipo para cada árbitro.
        - `partidos:` Lista con los equipos enfrenteados en cada partido.

    2) Backtracking:

        - El algoritmo intenta asignar árbitros a cada partido de manera secuencial.
        - Verifica si un árbitro es válido con la función `es_valido`:
            - No debe estar asignado aún.
            - La puntuación no debe ser $-\infty$ para ninguno de los equipos en el partido.
        - Se actualiza con la puntuación total conforme se asignan árbitros.

    3) Condición de Fin:

        - Si todos los partidos tienen asignado un árbitro, se compara la puntuación total obtenida con la máxima registrada.

    4) Solución Óptima:

        - Se guarda la asignación que maximiza la puntuación total.

7) Ejemplo de salida

    Para la matriz de preferencias y los partidos dados:

```
## Mejor puntuación total: 29
## Asignación de árbitros a partidos: [0, 1]
```