TrabajoFinal_GeneracionTribunales

February 28, 2024

1 Algoritmos de optimización - Trabajo Práctico

Nombre y Apellidos: Pedro Javier Sánchez San José Url: https://github.com/Psancs05/03MIAR—Algoritmos-de-Optimizacion—2023 Google Colab: https://colab.research.google.com/drive/1qqv4h-GytI2MZ0Uhm9acVBUdZt_6OOcQ Problema: 3. Configuración de Tribunales

Descripción del problema: Se precisa configurar tribunales de evaluación para un grupo de 15 alumnos que desean presentar su Trabajo Fin de Máster (TFM). Cada tribunal está compuesto por tres profesores, cada uno desempeñando uno de los siguientes roles: Presidente, Secretario o Vocal. Se dispone de las disponibilidades de los profesores y los roles que pueden ejercer.

#Modelo - ¿Como represento el espacio de soluciones? - ¿Cual es la función objetivo? - ¿Como implemento las restricciones?

Para la representación del problema se ha comenzado creando un array con las disponibilidades de los profesores y con los roles que pueden ejercer.

[ ]:
```
import numpy as np
import random
```

[ ]:
```
#Respuesta
disponibilidad_excel = [
    [0,1,1,1,0,1,1,1,0,1,1,1,1,1,1,1,0,0,1,0,1,0,1,1,1,1,1,1,1,1,1,1,1,0,0],
    [1,1,1,1,0,0,0,0,1,1,1,1,0,0,1,0,0,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1],
    [0,0,1,1,0,1,1,1,1,1,1,0,0,1,1,1,1,1,1,1,1,1,1,0,1,1,1,0,1,0,1,1,0,1,0,1],
    [1,0,1,0,1,1,0,1,0,0,1,1,1,1,0,0,1,1,1,1,1,1,1,0,1,1,0,1,1,1,1,1,1,1,1,0],
    [1,1,0,1,0,1,1,1,1,1,1,0,1,1,1,1,0,1,1,0,1,1,0,1,1,1,1,0,1,1,1,0,1,1,1,1,0],
    [1,1,1,1,1,1,0,0,1,1,1,1,1,1,1,1,1,1,0,0,1,1,1,1,1,1,1,0,0,1,1,1,1,1,0,1,0,1],
    [0,1,1,1,1,1,1,1,1,1,0,1,1,0,1,0,0,1,1,1,0,0,1,1,1,0,0,0,0,1,1,1,1,1,1,1,0,1],
    [1,1,1,1,1,1,0,0,1,1,0,1,1,1,1,0,1,1,1,1,0,0,1,1,0,1,1,1,1,1,1,1,1,0,1,1,1,0,1,0],
    [1,0,1,1,0,1,0,0,1,1,1,1,1,1,1,1,1,1,0,0,0,1,1,1,1,1,1,1,1,1,1,0,1,0,1,1,1,1,1],
    [1,1,0,1,1,0,1,1,0,0,0,0,0,1,1,1,0,0,1,1,1,1,1,0,0,1,1,1,1,1,1,1,1,0,0,0,1],
]
disponibilidad = np.array(disponibilidad_excel)
print(disponibilidad.shape)

# Presidente | Secretario | Vocal
roles_excel = [
```

```
    [1, 1, 1],
    [1, 1, 1],
    [1, 0, 1],
    [0, 1, 1],
    [1, 1, 1],
    [1, 1, 1],
    [0, 1, 1],
    [0, 1, 1],
    [1, 1, 1],
    [1, 1, 1],
]
roles = np.array(roles_excel)
print(roles.shape)
```

```
(10, 35)
(10, 3)
```

[ ]:
```
numero_tribunales = 15
```

En este caso `disponibilidades` es un matriz de 10x35 en la que cada fila es la disponibilidad de un profesor y cada columna un slot de tiempo. Los valores a 1 representan horas en las que el profesor está disponible y los 0 en las que no. La matriz `roles` es una matriz 10x3 en la que cada fila representa a un profesor y cada columna si dispone del rol específico o no. En este caso la primera columna sería el rol de Presidente, la seguna Secretario, y la tercera Vocal.

El objetivo del problema es minimizar una función de coste que representa la puntuación que se le da a una asiganción de tribunales. Los tribunales que no cumplan con las restricciones, van a recibir una puntuación negativa. Los tribunales que sí cumplan con las restricciones, van a recibir puntuaciones más altas y se va a premiar aquellas configuraciones en las que la distribución de horarios entre los profesores sea uniforme, es decir, que todos los profesores tengan más o menos el mismo número de tribunales.

#Diseño - ¿Que técnica utilizo? ¿Por qué?

Para la resolución de este problema se ha utilizado un algoritmo genético. La idea detrás de esta aplicación es generar un conjunto de soluciones iniciales, e ir iterando sobre ellas con los procedimientos típicos de estos algoritmos (selección, cruce y mutación) hasta conseguir que las soluciones mejoren. Los algoritmos genéticos son métodos heurísticos, por lo que no podemos asegurar que la solución que nos proporcionen sea la óptima. También es importante destacar que estos algoritmo son no deterministas y proporcionarán diferentes soluciones para los mismos datos de entrada debido a que incorporan mecanismos aleatorios. He decidio utilizar este tipo de algoritmos para llevar a la práctica toda la teoría recibida en clase sobre estos algoritmos y porque este problema puede resolverse mediante este tipo de algoritmos.

Para la implementación de este algoritmo en concreto se han seguido los siguientes pasos:

- Se genera una población aleatoria. Cada individuo de la población es una matriz de 10x35 generada con valores seleccionados aleatoria, que tiene valores 0, 1, 2 y 3. Los 0s indican que el profesor no forma parte de ese tribunal; los 1s que forma parte como Presidente; los 2s que forma parte como Secretario; y los 3s que forma parte como Vocal.

- Se define una función de fitness que comprueba primero las restricciones de disponibilidad, roles y número de tribunales. Si alguna no se cumple, se añade una penalización para ese individuo. Si cumple las restricciones, se devuelve la varianza negativa sobre el número de tribunales que ejerce cada profesor. Esto significa que los valores más cercanos a 0 representan las mejores soluciones al problema. Cuanto más negativo sea el fitness de un individuo, peor será esa solución.
- Se define el mecanismo de seleccion. En este caso, he elegido el mecanismo de ruleta.
- Se define un mecanismo de cruce. Se genera aleatoriamente un punto de corte en una columna (una hora de tribunal) y se recombinan los padres teniendo en cuenta las restricciones para generar 2 hijos nuevos.
- Se define un mecanismo de mutación. En este caso se define un ratio de mutación y se comprueba para cada gen de la población. Si un gen muta, su valor cambia de forma aleatoria entre todos los posibles roles disponibles.
- También se ha incorporado un mecanismo de elitismo, que permite preservar los mejores individuos de cada población a la siguiente iteración. De esta forma nos aseguramos que los mejores individuos sobreviven al paso de las iteraciones.
- Por último, se itera con todo el algoritmo durante un número determinado de generaciones. Al final del proceso, se muestra el mejor individuo generado y su valor de fitness.

El primer paso es la representación del problema. En este caso, he considerado que cada indiviudo de la población sea una matriz de 10x35, muy similar a la matriz de disponibilidad, en la que cada fila representa un profesor y cada columna una hora. En este caso los valores posibles serán: 0 si el profesor no participa en el tribunal, 1 si participa como Presidente, 2 si participa como Secretario y 3 si participa como Vocal. Inicialmente, se va a generar una población inicial de forma aleatoria respetando las restricciones que impone el problema.

[ ]:
```
def generar_poblacion(tamanio, disponibilidad, roles, num_tribunales):
  poblacion = []

  for i in range(tamanio):
    num_profesores, num_horas = disponibilidad.shape
    individuo = np.zeros((num_profesores, num_horas), dtype=int)  # Creamos el␣
↪individuo con todo a 0

    tribunales_formados = 0

    # Iteramos formando los tribunales
    while tribunales_formados < num_tribunales:
      horas_disponibles = [hora for hora in range(num_horas) if np.
↪sum(individuo[:, hora] > 0) < 3]
      if not horas_disponibles:
        break  # No hay más horas disponibles para formar tribunales
      hora = np.random.choice(horas_disponibles)

      # Intentamos formar un tribunal con esa hora
      roles_asignados = 0
      for rol in range(1, 4):  # 1 presidente, 2 secretario, 3 vocal
```

```
        profesores_disponibles = np.where((disponibilidad[:, hora] == 1) &␣
↪(roles[:, rol-1] == 1))[0]
        profesores_disponibles = [prof for prof in profesores_disponibles if␣
↪individuo[prof, hora] == 0]  # Quitamos los profesores que ya tienen rol a esa␣
↪hora

        if profesores_disponibles:
          # Asignamos un profesor aleatorio de los disponibles
          profesor_elegido = np.random.choice(profesores_disponibles)
          individuo[profesor_elegido, hora] = rol
          roles_asignados += 1

      # Verificamos si se ha formado un tribunal completo a esa hora
      if roles_asignados == 3:
        tribunales_formados += 1

    poblacion.append(individuo)

  return poblacion
```

Ejemplo de población

[ ]:
```
poblacion_inicial = generar_poblacion(10, disponibilidad, roles,␣
↪numero_tribunales)
```

[ ]:
```
print(poblacion_inicial[0])
print(poblacion_inicial[0].shape)
print(len(poblacion_inicial))
```

```
[[0 0 1 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 2 0 0 0 2 1 0 0 0]
 [0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 1 0 0 0 0 0 0 1 0 3 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 1 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 3 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 3 0 0 0 0 0 2 0 0 2 1 0 0]
 [2 1 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 3 0 2]
 [0 3 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 3 0 0 3]
 [0 0 3 0 0 0 0 0 0 0 0 0 3 0 0 3 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0]
 [1 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 2 0 1]
 [0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
(10, 35)
10
```

A continuación vamos a definir la función de fitness. En este caso, la función va a devolver la varianza negativa de la distribución de tribunales entre los profesores. De esta forma, cuanto más repartidos estén los profesores, más alta será la puntuación de fitness de un individuo. Además, se va a implementar también un sistema de penalizaciones: si un individuo incumple alguna de las restricciones del problema, se añadirá una penalización al fitness de ese individuo concreto.

[ ]:
```python
def fitness(individuo, disponibilidad, roles, numero_tribunales):
    penalizacion = 100
    penalizacion_total = 0

    # Verificar restricciones y contar restricciones
    for profesor in range(individuo.shape[0]):
        for hora in range(individuo.shape[1]):
            if individuo[profesor, hora] > 0:
                # Penalizar si el profesor no está disponible en esa hora
                if disponibilidad[profesor, hora] == 0:
                    penalizacion_total += penalizacion
                # Penalizar si el profesor no puede desempeñar el rol asignado
                rol = individuo[profesor, hora]
                if roles[profesor, rol - 1] == 0:
                    penalizacion_total += penalizacion

    # Verificar si hay más de 3 profesores o más de un cargo en un tribunal
    for hora in range(individuo.shape[1]):
        if np.sum(individuo[:, hora] > 0) > 3:
            penalizacion_total += penalizacion
        for rol in range(1, 4):
            if np.sum(individuo[:, hora] == rol) > 1:
                penalizacion_total += penalizacion

    # Verificar que se formen exactamente numero_tribunales tribunales
    tribunales_formados = 0
    for hora in range(individuo.shape[1]):
        presidente = np.sum(individuo[:, hora] == 1)
        secretario = np.sum(individuo[:, hora] == 2)
        vocal = np.sum(individuo[:, hora] == 3)
        if presidente == 1 and secretario == 1 and vocal == 1:
            tribunales_formados += 1
    if tribunales_formados != numero_tribunales:
        penalizacion_total += penalizacion

    # Calcular la varianza en el número de tribunales que hace cada profesor
    tribunales_por_profesor = np.sum(individuo > 0, axis=1)
    varianza = np.var(tribunales_por_profesor)

    # La puntuación será la inversa de la varianza, ajustada por penalizaciones
    score = -varianza - penalizacion_total

    return score
```

Ejemplos de fitness para la población inicial

[ ]:
```python
[fitness(poblacion_inicial[i], disponibilidad, roles, numero_tribunales) for i␣
↪in range(len(poblacion_inicial))]
```

[ ]:
```
[-1.45, -3.45, -4.45, -3.25, -3.85, -5.45, -1.85, -4.45, -6.65, -2.65]
```

El siguiente paso es el método de selección. Para este problema he elegido el método de la ruleta, en el que a cada individuo se le asigna una probabilidad de ser elegido en función de su valor de fitness. A mejor puntuación, más oportunidades de ser elegido.

[ ]:
```
def seleccion_ruleta(poblacion, puntuacion_fitness):
    # Encuentra el valor más negativo y resta ese valor de todas las puntuaciones
    # para hacerlas positivas. Suma 1 para evitar puntuaciones de fitness de␣
↪cero.
    min_fitness = min(puntuacion_fitness)
    puntuaciones_ajustadas = [1 + (score - min_fitness) for score in␣
↪puntuacion_fitness]

    # Calcular la suma total de las puntuaciones de fitness ajustadas
    fitness_total = sum(puntuaciones_ajustadas)

    # Normalizar las puntuaciones de fitness ajustadas para que sumen 1
    puntuaciones_normalizadas = [score / fitness_total for score in␣
↪puntuaciones_ajustadas]

    # Calcular la suma acumulada de puntuaciones
    puntuaciones_acumuladas = np.cumsum(puntuaciones_normalizadas)

    # Seleccionar individuos
    individuos_seleccionados = []
    for _ in range(len(poblacion)):
        r = np.random.rand()
        # Utilizar búsqueda binaria para encontrar el índice del individuo a␣
↪seleccionar
        index = np.searchsorted(puntuaciones_acumuladas, r)
        individuos_seleccionados.append(poblacion[index])

    return individuos_seleccionados
```

El siguiente paso es la función de cruce. En este paso, dos individuos (a los que llamaremos padres), combinarán su genotipo para dar dos individuos nuevos (que llamaremos hijos). El tipo de cruce que se ha implementado es una recombinación de bloques de tiempo, es decir, recombinamos las columnas de los padres (que representan un tribunal completo) teniendo en cuenta las restricciones de disponibilidad y roles.

[ ]:
```
def cruce(padre_1, padre_2, disponibilidad, roles):
    num_profesores, num_horas = disponibilidad.shape
    hijo_1 = np.copy(padre_1)
    hijo_2 = np.copy(padre_2)

    # Identificar bloques de tiempo donde ambos padres tienen asignaciones␣
↪válidas y completas
```

```
    bloques_tiempo_validos = []
    for hora in range(num_horas):
        if (np.sum(padre_1[:, hora] > 0) == 3 and np.sum(padre_2[:, hora] > 0)
↪== 3 ):
            bloques_tiempo_validos.append(hora)

    # Realizar el cruce solo en bloques de tiempo válidos
    for hora in bloques_tiempo_validos:
        hijo_1[:, hora], hijo_2[:, hora] = hijo_2[:, hora], hijo_1[:, hora]

    return hijo_1, hijo_2
```

La última función auxiliar que necesitamos es la mutación. En este caso, cuando un individuo muta, altera uno de sus genes de forma aleatoria para ejercer otro rol de tribunal (manteniendo las restricciones de roles y disponibilidad)

[ ]:
```
def mutacion(individuo, disponibilidad, roles):
    num_profesores, num_horas = disponibilidad.shape

    for hora in range(num_horas):
        # Seleccionar un rol para mutar
        rol_a_mutar = np.random.choice([1, 2, 3])
        # Encontrar todos los profesores que actualmente tienen ese rol en esta
↪hora
        profesores_con_rol = np.where(individuo[:, hora] == rol_a_mutar)[0]

        # Si hay algún profesor con ese rol, intentar reasignarlo
        if profesores_con_rol.size > 0:
            # Seleccionar un profesor al azar que tenga el rol
            profesor_con_rol = np.random.choice(profesores_con_rol)
            # Encontrar profesores que pueden tomar el rol y no tienen un rol en
↪esta hora
            profesores_disponibles = np.where((disponibilidad[:, hora] == 1) &
↪(roles[:, rol_a_mutar - 1] == 1) & (individuo[:, hora] == 0))[0]

            if profesores_disponibles.size > 0:
                # Seleccionar un nuevo profesor para el rol
                nuevo_profesor = np.random.choice(profesores_disponibles)
                # Reasignar el rol
                individuo[profesor_con_rol, hora] = 0
                individuo[nuevo_profesor, hora] = rol_a_mutar

    return individuo
```

El último paso es poner todo en conjunción y ejecutar el algoritmo. Durante la ejecución del algoritmo, se va a crear una población inicial y se va a iterar sobre ella para ir mejorándola con el paso del tiempo. Al final, tendremos la población con los mejores individuos Un detalle importante es que vamos a implementar también elitismo, que consiste en que los mejores individuos de una

población van a pasar a formar parte de la siguiente iteración directamente, sin realizar selección, cruce y mutación. De esta forma, nos aseguramos de que los mejores individuos sobrevivan a la iteración.

[ ]:
```
# Parámetros del algoritmo genético
tamanio_poblacion = 50
numero_generaciones = 50
ratio_mutacion = 0.01
ratio_cruce = 0.6
num_elites = 4
num_tribunales = 15

# Inicializar la población
poblacion = generar_poblacion(tamanio_poblacion, disponibilidad, roles,␣
 ↪num_tribunales)

# Evaluar la población inicial
fitness_scores = [fitness(individuo, disponibilidad, roles, num_tribunales) for␣
 ↪individuo in poblacion]

# Bucle principal del algoritmo genético
for generacion in range(numero_generaciones):
    # Seleccionar individuos para la reproducción
    selected_individuals = seleccion_ruleta(poblacion, fitness_scores)

    # Elitismo, los mejores individuos los guardamos para la siguiente iteración
    indices_elites = np.argsort(fitness_scores)[-num_elites:]
    elites = [poblacion[i] for i in indices_elites]
    next_generation = elites.copy()  # Comenzar con los individuos élite

    # Iterar creando el resto de la población
    while len(next_generation) < tamanio_poblacion:
        # Seleccionar dos padres de manera aleatoria sin reemplazo
        parents = random.sample(selected_individuals, 2)
        parent1, parent2 = parents[0], parents[1]

        # Aplicar cruce con una cierta probabilidad
        if np.random.rand() < ratio_cruce:  # Solo cruzamos para cierta␣
 ↪probabilidad
          child_1, child_2 = cruce(parent1, parent2, disponibilidad, roles)
        else:
          child_1, child_2 = parent1.copy(), parent2.copy()

        # Aplicar mutación con una cierta probabilidad
        if np.random.rand() < ratio_mutacion:  # Solo mutamos para cierta␣
 ↪probabilidad
          child_1 = mutacion(child_1, disponibilidad, roles)
```

```
            child_2 = mutacion(child_2, disponibilidad, roles)

        # Aniadir los hijos a la próxima generación
        next_generation.append(child_1)
        if len(next_generation) < tamanio_poblacion:
            next_generation.append(child_2)

    # Reemplazar la población actual con la nueva generación
    poblacion = next_generation

    # Evaluar la nueva población
    fitness_scores = [fitness(individuo, disponibilidad, roles, num_tribunales)␣
↪for individuo in poblacion]

    # Opcional: Imprimir información sobre la generación actual
    print(f"Generación {generacion}: Mejor puntuación de fitness =␣
↪{max(fitness_scores)}")

# Encontrar el mejor individuo de la última generación
mejor_puntuacion = max(fitness_scores)
mejor_individuo = poblacion[fitness_scores.index(mejor_puntuacion)]

# Imprimir el mejor individuo y su puntuación de fitness
print("Mejor individuo encontrado:")
print(mejor_individuo)
print(f"Puntuación de fitness: {mejor_puntuacion}")
```

```
Generación 0: Mejor puntuación de fitness = -1.05
Generación 1: Mejor puntuación de fitness = -1.05
Generación 2: Mejor puntuación de fitness = -0.85
Generación 3: Mejor puntuación de fitness = -0.85
Generación 4: Mejor puntuación de fitness = -0.85
Generación 5: Mejor puntuación de fitness = -0.85
Generación 6: Mejor puntuación de fitness = -0.85
Generación 7: Mejor puntuación de fitness = -0.85
Generación 8: Mejor puntuación de fitness = -0.85
Generación 9: Mejor puntuación de fitness = -0.85
Generación 10: Mejor puntuación de fitness = -0.85
Generación 11: Mejor puntuación de fitness = -0.85
Generación 12: Mejor puntuación de fitness = -0.85
Generación 13: Mejor puntuación de fitness = -0.85
Generación 14: Mejor puntuación de fitness = -0.85
Generación 15: Mejor puntuación de fitness = -0.85
Generación 16: Mejor puntuación de fitness = -0.85
Generación 17: Mejor puntuación de fitness = -0.85
Generación 18: Mejor puntuación de fitness = -0.85
Generación 19: Mejor puntuación de fitness = -0.85
```

```
Generación 20: Mejor puntuación de fitness = -0.85
Generación 21: Mejor puntuación de fitness = -0.85
Generación 22: Mejor puntuación de fitness = -0.85
Generación 23: Mejor puntuación de fitness = -0.85
Generación 24: Mejor puntuación de fitness = -0.85
Generación 25: Mejor puntuación de fitness = -0.85
Generación 26: Mejor puntuación de fitness = -0.85
Generación 27: Mejor puntuación de fitness = -0.85
Generación 28: Mejor puntuación de fitness = -0.85
Generación 29: Mejor puntuación de fitness = -0.85
Generación 30: Mejor puntuación de fitness = -0.85
Generación 31: Mejor puntuación de fitness = -0.85
Generación 32: Mejor puntuación de fitness = -0.85
Generación 33: Mejor puntuación de fitness = -0.85
Generación 34: Mejor puntuación de fitness = -0.85
Generación 35: Mejor puntuación de fitness = -0.85
Generación 36: Mejor puntuación de fitness = -0.85
Generación 37: Mejor puntuación de fitness = -0.85
Generación 38: Mejor puntuación de fitness = -0.85
Generación 39: Mejor puntuación de fitness = -0.85
Generación 40: Mejor puntuación de fitness = -0.85
Generación 41: Mejor puntuación de fitness = -0.85
Generación 42: Mejor puntuación de fitness = -0.85
Generación 43: Mejor puntuación de fitness = -0.85
Generación 44: Mejor puntuación de fitness = -0.85
Generación 45: Mejor puntuación de fitness = -0.85
Generación 46: Mejor puntuación de fitness = -0.85
Generación 47: Mejor puntuación de fitness = -0.85
Generación 48: Mejor puntuación de fitness = -0.85
Generación 49: Mejor puntuación de fitness = -0.85
Mejor individuo encontrado:
[[0 0 3 0 0 0 2 2 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0]
 [0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 3 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 2 0 0 0 0 0 0 3 0 0 0]
 [0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 1 0 1 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 3 0 0 0 2 0 0]
 [0 0 2 0 0 0 0 0 0 0 0 0 0 2 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 3 0 0 0 0 2 0 0]
 [0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 3 0 2 0 0 0 0 0 0 0 0 0 0 1 3 0]
 [0 0 0 0 3 0 3 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0]]
Puntuación de fitness: -0.85
```

El algrotimo itera y mejora con las fases! Además, hemos obtenido el mejor individuo de todos los generados. Podemos crear una función auxiliar para comprobar si la solución es buena.

[ ]:
```python
def es_solucion_valida(individuo, disponibilidad, roles, numero_tribunales):
    num_profesores, num_horas = individuo.shape
```

```
    # Comprobar la disponibilidad y los roles permitidos para cada profesor
    for profesor in range(num_profesores):
        for hora in range(num_horas):
            rol = individuo[profesor, hora]
            if rol > 0:
                # Comprobar si el profesor está disponible en esa hora
                if disponibilidad[profesor, hora] == 0:
                    return False
                # Comprobar si el profesor puede desempeñar el rol asignado
                if roles[profesor, rol - 1] == 0:
                    return False

    # Comprobar que se formen exactamente num_tfms tribunales
    tribunales_formados = 0
    for hora in range(num_horas):
        # Comprobar que haya exactamente un presidente, un secretario y un vocal␣
↪en esta hora
        presidente = np.sum(individuo[:, hora] == 1)
        secretario = np.sum(individuo[:, hora] == 2)
        vocal = np.sum(individuo[:, hora] == 3)
        if presidente == 1 and secretario == 1 and vocal == 1:
            tribunales_formados += 1

    if tribunales_formados != numero_tribunales:
        return False

    # Si todas las comprobaciones son correctas, la solución es válida
    return True
```

[ ]:
```
es_solucion_valida(mejor_individuo, disponibilidad, roles, num_tribunales)
```

[ ]:
```
True
```

La solución es válida! Y además, cumple con los requisitos:

- Hay 15 tribunales formados.
- Cada tribunal tiene 3 profesores, con rol único.
- Los roles de los profesores son válidos.
- Las disponibilidades de los profesores son válidas.
- La distribución de profesores está equilibrada (todos los profesores tienen entre 4 y 6 tribunales).

Por tanto, la solución alcanzada es válida.

#Análisis - ¿Que complejidad tiene el problema?. Orden de complejidad y Contabilizar el espacio de soluciones

Para calcular la complejidad computación del algoritmo entero vamos a ir parte por parte:

- `generar_poblacion()`: La complejidad es O(tamaño * num_tribunales * num_profesores * num_horas) ya que se itera sobre todas estas variables. El resto de operaciones son elementales. Puesto que los tribunales, los profesores y las horas son fijas para este problema, la complejidad la podemos medir en cuanto al tamaño de la población que queremos.
- `fitness()`: La complejidad de esta función es O(num_profesores * num_horas) que son las variables que se iteran. El resto de operaciones son asignaciones y comaparaciones que consideramos elementales (esta complejidad es para evaluar un individuo).
- `seleccion_ruleta()`: Esta función tiene complejidad O(tamaño * log(tamaño)), ya que en el último bucle for se realizan operaciones de búsqueda binaria.
- `cruce()`: La complejidad de esta función es O(num_horas + bloques_tiempo), ya que en cada bulce se realizan únicamente operaciones elementales.
- `mutacion()`: La complejidad es O(num_horas * num_profesores), el resto de operaciones son elementales.
- `algoritmo genetico`: para dar la complejidad general del algoritmo vamos a tener en cuenta ls datos que sabemos que son fijos para este problema (15 tribunales, 10 profesores, 3 roles y 35 horas).

La complejidad total sería: O(num_generacions * tamaño_poblacion * (num_horas * num_tribunales * num_profesores + log(tamaño_poblacion))) Esto sería aproximadamente: O(num_generacions * tamaño_poblacion * (35 * 15 * 10 + log(tamaño_poblacion))) Por lo que el algoritmo dependería del tamaño de la población y del número de generaciones que se quiera iterar.