

Algoritmos de optimización - Seminario

Nombre y Apellidos: Roberto Merchán González

Url: https://github.com/robertomergon/03MIAR-Algoritmos-de-Optimizacion/blob/main/Final/RMG_Seminario_Algoritmos.ipynb

Problema:

1. Sesiones de doblaje

Descripción del problema: Se precisa coordinar el doblaje de una película. Los actores del doblaje deben coincidir en las tomas en las que sus personajes aparecen juntos en las diferentes tomas. Los actores de doblaje cobran todos la misma cantidad por cada día que deben desplazarse hasta el estudio de grabación independientemente del número de tomas que se graben. No es posible grabar más de 6 tomas por día. El objetivo es planificar las sesiones por día de manera que el gasto por los servicios de los actores de doblaje sea el menor posible. Los datos son: Número de actores: 10 Número de tomas : 30 Actores/Tomas : <https://bit.ly/36D8IuK>

- 1 indica que el actor participa en la toma
- 0 en caso contrario

(*) La respuesta es obligatoria

```
In [ ]: tomas = [
    [1, 1, 1, 1, 1, 0, 0, 0, 0, 0],
    [0, 0, 1, 1, 1, 0, 0, 0, 0, 0],
    [0, 1, 0, 0, 1, 0, 1, 0, 0, 0],
    [1, 1, 0, 0, 0, 1, 1, 0, 0, 0],
    [0, 1, 0, 1, 0, 0, 0, 1, 0, 0],
    [1, 1, 0, 1, 0, 0, 0, 0, 0, 0],
    [1, 1, 0, 1, 1, 0, 0, 0, 0, 0],
    [1, 1, 0, 1, 1, 0, 0, 0, 0, 0],
    [1, 1, 0, 0, 0, 1, 0, 0, 0, 0],
    [1, 1, 0, 1, 0, 0, 0, 0, 0, 0],
    [1, 1, 0, 0, 0, 1, 0, 0, 0, 0],
    [1, 1, 0, 1, 0, 0, 0, 0, 0, 0],
    [1, 1, 0, 0, 0, 1, 0, 0, 0, 0],
    [1, 1, 1, 0, 1, 0, 0, 1, 0, 0],
    [1, 1, 1, 1, 0, 1, 0, 0, 0, 0],
    [1, 0, 0, 1, 1, 0, 0, 0, 0, 0],
    [1, 0, 1, 0, 0, 1, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 1, 0, 0, 0, 0],
    [1, 0, 1, 0, 0, 0, 0, 0, 0, 0],
    [1, 0, 1, 1, 0, 0, 0, 0, 0, 0],
    [1, 0, 1, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 1, 0, 0, 0, 0],
    [1, 1, 0, 1, 0, 0, 0, 0, 0, 0],
    [1, 0, 1, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 1, 0, 0, 0, 0],
    [1, 1, 0, 1, 0, 0, 0, 0, 0, 0],
    [1, 0, 1, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 1, 0, 0, 0, 0],
    [1, 1, 0, 1, 0, 0, 0, 0, 0, 0],
    [1, 0, 1, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 1, 0, 0, 0, 0],
    [1, 1, 0, 1, 0, 0, 0, 0, 0, 0],
    [1, 0, 1, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 1, 0, 0, 0, 0],
    [1, 1, 0, 1, 0, 0, 0, 0, 0, 0],
    [1, 0, 1, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 1, 0, 0, 0, 0],
    [1, 1, 0, 1, 0, 0, 0, 0, 0, 0],
    [1, 0, 1, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 1, 0, 0, 0, 0],
    [1, 0, 0, 1, 0, 0, 0, 0, 0, 0]
]
```

```
[1, 0, 0, 0, 1, 1, 0, 0, 0, 0],
[1, 0, 0, 1, 0, 0, 0, 0, 0, 0],
]
```

(*) ¿Cuantas posibilidades hay sin tener en cuenta las restricciones?

Sin tener en cuenta las restricciones, se seleccionan 6 tomas de 30, por lo que el número de posibilidades es el número combinatorio de 30 en 6, que se calcula como:

$$C(30, 6) = \frac{30!}{6!(30-6)!} = \frac{30!}{6!24!} = 593775$$

¿Cuantas posibilidades hay teniendo en cuenta todas las restricciones.

Las restricciones principales son:

1. No podemos grabar más de 6 tomas por día.
2. Los actores deben coincidir en las tomas en las que sus personajes aparecen juntos.

Para resolver el problema, se puede plantear como un problema de asignación, en el que se asignan actores a tomas de manera que se minimice el coste total. Para ello, se puede utilizar el algoritmo de la fuerza bruta, que consiste en probar todas las posibles combinaciones de actores y tomas, y seleccionar la que tenga el menor coste. Sin embargo, debido a la complejidad del problema, se puede utilizar un algoritmo de optimización, como el algoritmo de la búsqueda tabú, que permite encontrar una solución aproximada en un tiempo razonable.

Modelo para el espacio de soluciones

(*) ¿Cuál es la estructura de datos que mejor se adapta al problema? Argumentalo.(Es posible que hayas elegido una al principio y veas la necesidad de cambiar, argüentalo)

Para representar el problema, se puede utilizar una matriz en la que cada fila representa una toma y cada columna representa un actor. Los valores de la matriz indican si un actor participa en una toma o no. Por ejemplo, si el valor de la matriz en la posición (i, j) es 1, significa que el actor j participa en la toma i . Si el valor es 0, significa que el actor no participa en la toma. De esta manera, se puede representar de forma sencilla la relación entre actores y tomas, y se pueden aplicar algoritmos de optimización para encontrar la solución óptima.

```
In [ ]: import numpy as np
tomas = np.array(tomas)
```

Según el modelo para el espacio de soluciones

(*)¿Cual es la función objetivo?

La función objetivo es minimizar el coste total de los servicios de los actores de doblaje. Para ello, se puede definir una función que calcule el coste total en función de la asignación de actores a tomas. La función objetivo se puede definir como:

$$\text{Coste} = N_{\text{dias}} \times N_{\text{actores}}$$

Donde N_{dias} es el número de días necesarios para grabar todas las tomas y N_{actores} es el número de actores que participan en las tomas. El objetivo es minimizar el coste total, es decir, minimizar el número de días necesarios para grabar todas las tomas y el número de actores que participan en las tomas.

(*)¿Es un problema de maximización o minimización?

Es un problema de minimización, ya que el objetivo es minimizar el coste total de los servicios de los actores de doblaje.

Diseña un algoritmo para resolver el problema por fuerza bruta

```
In [ ]: from itertools import combinations

# Número de tomas y actores
num_tomas = tomas.shape[0]
num_actores = tomas.shape[1]

# Función para verificar si un conjunto de tomas puede ser grabado en un día
def es_valido_dia(tomas, participacion):
    actores_en_dia = np.zeros(num_actores, dtype=int)
    for toma in tomas:
        actores_en_dia = np.logical_or(actores_en_dia, participacion[toma])
    # Verificamos que ningún actor se repita en las tomas
    return np.all(actores_en_dia <= 1)

# Función de fuerza bruta para encontrar la solución óptima
def fuerza_bruta(participacion):
    min_dias = float('inf')
    mejor_combinacion = None

    # Generar todas las combinaciones posibles de días
    for k in range(1, 7):
        for dias in combinations(range(num_tomas), k):
            if es_valido_dia(dias, participacion):
                tomas_restantes = set(range(num_tomas)) - set(dias)
                if len(tomas_restantes) == 0:
                    if k < min_dias:
                        min_dias = k
                        mejor_combinacion = [dias]
                else:
                    for k2 in range(1, 7):
                        for dias2 in combinations(tomas_restantes, k2):
                            if es_valido_dia(dias2, participacion):
                                tomas_restantes2 = tomas_restantes - set(dias2)
                                if len(tomas_restantes2) == 0:
```

```

        if 2 < min_dias:
            min_dias = 2
            mejor_combinacion = [dias, dias2]
        else:
            # Continuar este proceso anidado hasta cubrir todos los días
            pass # Simplificación para ilustrar el concepto

    return mejor_combinacion, min_dias

# Llamar a la función de fuerza bruta
mejor_combinacion, min_dias = fuerza_bruta(tomas)
print("Mejor combinación de días:", mejor_combinacion)
print("Número mínimo de días:", min_dias)

```

Mejor combinación de días: [(0, 1, 2, 3, 4, 5)]
Número mínimo de días: 1

Calcula la complejidad del algoritmo por fuerza bruta

El algoritmo tiene una complejidad de $O(n!)$, ya que se prueban todas las posibles combinaciones de actores y tomas para encontrar la solución óptima. Dado que el número de actores y tomas es de 10 y 30 respectivamente, el número de posibles combinaciones es de $10! \times 30!$, lo que hace que el algoritmo sea muy ineficiente para problemas de gran tamaño de tomas y actores.

(*)Diseña un algoritmo que mejore la complejidad del algoritmo por fuerza bruta. Argumenta porque crees que mejora el algoritmo por fuerza bruta

In []:

```

import random

num_tomas = tomas.shape[0]
num_actores = tomas.shape[1]

# Función para calcular la aptitud de una solución
def calcular_aptitud(solucion):
    dias = len(solucion)
    return dias

# Función para generar una solución aleatoria sin repetir tomas
def generar_solucion_aleatoria():
    tomas = list(range(num_tomas))
    random.shuffle(tomas)
    solucion = []
    dia_actual = []
    actores_dia = np.zeros(num_actores, dtype=int)
    for toma in tomas:
        if len(dia_actual) < 6 and np.all(np.logical_or(actores_dia, tomas[toma])):
            dia_actual.append(toma)
            actores_dia = np.logical_or(actores_dia, tomas[toma])
        else:
            solucion.append(dia_actual)
            dia_actual = [toma]
            actores_dia = tomas[toma]
    if dia_actual:
        solucion.append(dia_actual)
    dia_actual = []
    actores_dia = tomas[toma]
    return solucion

```

```

# Función de cruce para generar un nuevo hijo sin repetir tomas
def cruce(padre1, padre2):
    hijo = []
    tomas_usadas = set()
    for dia1, dia2 in zip(padre1, padre2):
        nuevo_dia = []
        for toma in dia1:
            if toma not in tomas_usadas:
                nuevo_dia.append(toma)
                tomas_usadas.add(toma)
        for toma in dia2:
            if toma not in tomas_usadas and len(nuevo_dia) < 6:
                nuevo_dia.append(toma)
                tomas_usadas.add(toma)
        hijo.append(nuevo_dia)
    # Añadir tomas faltantes al hijo
    for toma in range(num_tomas):
        if toma not in tomas_usadas:
            for dia in hijo:
                if len(dia) < 6:
                    dia.append(toma)
                    tomas_usadas.add(toma)
                    break
    return hijo

# Función de mutación para alterar una solución sin repetir tomas
def mutacion(solucion):
    if random.random() < 0.1: # Probabilidad de mutación
        dia1, dia2 = random.sample(range(len(solucion)), 2)
        if solucion[dia1] and solucion[dia2]:
            toma1 = random.choice(solucion[dia1])
            toma2 = random.choice(solucion[dia2])
            solucion[dia1].remove(toma1)
            solucion[dia2].remove(toma2)
            solucion[dia1].append(toma2)
            solucion[dia2].append(toma1)
    return solucion

# Algoritmo genético
def algoritmo_genetico(tamano_poblacion, generaciones):
    # Generar población inicial
    poblacion = [generar_solucion_aleatoria() for _ in range(tamano_poblacion)]
    for generacion in range(generaciones):
        # Evaluar aptitud
        aptitud = [calcular_aptitud(solucion) for solucion in poblacion]
        # Selección
        padres = random.choices(poblacion, weights=aptitud, k=2)
        # Cruce
        hijo = cruce(padres[0], padres[1])
        # Mutación
        hijo = mutacion(hijo)
        # Reemplazo
        peor_solucion = max(range(tamano_poblacion), key=lambda i: aptitud[i])
        poblacion[peor_solucion] = hijo
    mejor_solucion = min(poblacion, key=calcular_aptitud)
    return mejor_solucion

# Ejecutar el algoritmo genético
mejor_solucion = algoritmo_genetico(100, 1000)

```

```

print("Mejor solución encontrada (días de grabación):")
for i, dia in enumerate(mejor_solucion):
    print(f"Día {i+1}: Tomas { [d+1 for d in dia] }")
    participacion_dia = np.zeros(num_actores, dtype=int)
    for toma in dia:
        participacion_dia = np.logical_or(participacion_dia, tomas[toma])
    print("Participación de actores en el día:", np.where(participacion_dia)[0])
    print()

```

Mejor solución encontrada (días de grabación):

Día 1: Tomas [3, 24, 10, 12, 27, 30]

Participación de actores en el día: [1 2 3 4 5 6 7 9]

Día 2: Tomas [18, 11, 7, 19, 2, 28]

Participación de actores en el día: [1 2 3 4 5 6 8]

Día 3: Tomas [29, 4, 22, 21, 17, 14]

Participación de actores en el día: [1 2 3 4 5 6 7 8]

Día 4: Tomas [6, 26, 8, 15, 20, 1]

Participación de actores en el día: [1 2 3 4 5 6 7 9]

Día 5: Tomas [9, 23, 25, 16, 5, 13]

Participación de actores en el día: [1 2 3 4 5 8 10]

(*)Calcula la complejidad del algoritmo

El algoritmo genético es del orden $O(G \times P \times N)$, donde G es el número de generaciones, P es el tamaño de la población y N es el número de genes en cada individuo. En este caso, G es el número de generaciones, P es el tamaño de la población y N es el número de tomas. Por lo tanto, la complejidad del algoritmo genético es del orden $O(G \times P \times N)$. Lo cual es mucho más eficiente que el algoritmo por fuerza bruta.

Según el problema (y tenga sentido), diseña un juego de datos de entrada aleatorios

Generearemos un juego de datos de entrada aleatorios con 12 tomas y 10 actores con máximo de 5 actores por toma. Se ha elegido este juego de datos para que sea más fácil de visualizar y entender, además de ser más rápido de ejecutar en un tiempo razonable para el algoritmo de fuerza bruta.

```

In [ ]: def generar_datos_aleatorios(num_tomas, num_actores, max_actores_por_toma):
    # Inicializar la matriz de participación con ceros
    participacion = np.zeros((num_tomas, num_actores), dtype=int)

    for i in range(num_tomas):
        # Determinar cuántos actores participan en esta toma (al menos 1 y como
        num_actores_participantes = random.randint(1, max_actores_por_toma)

        # Seleccionar actores aleatoriamente para esta toma
        actores_participantes = random.sample(range(num_actores), num_actores_pa

        # Marcar la participación de los actores en la matriz
        for actor in actores_participantes:
            participacion[i, actor] = 1

```

```

    return participacion

# Parámetros
num_tomas = 12 # Número de tomas
num_actores = 10 # Número de actores
max_actores_por_toma = 5 # Máximo número de actores por toma (puede ser ajustada)

# Generar datos aleatorios
tomas = generar_datos_aleatorios(num_tomas, num_actores, max_actores_por_toma)

print("Participación de actores en las tomas:")
print(tomas)

```

Participación de actores en las tomas:

```

[[1 0 0 0 0 0 0 1 0]
 [1 1 0 1 0 0 0 1 1 0]
 [0 1 0 0 1 0 0 0 0 0]
 [0 1 0 0 1 0 1 0 1 0]
 [0 1 0 0 0 0 0 0 0 0]
 [1 0 1 0 0 0 0 0 1 1]
 [1 1 1 0 0 0 0 0 0 0]
 [0 1 0 0 0 0 0 0 0 0]
 [1 1 0 0 0 0 0 1 0 1]
 [0 0 0 0 1 0 0 0 0 0]
 [0 0 1 0 0 0 0 0 0 0]
 [1 0 0 0 1 0 1 1 0 1]]

```

Aplica el algoritmo al juego de datos generado

```

In [ ]: # Llamar a la función de fuerza bruta
mejor_combinacion, min_dias = fuerza_bruta(tomas)
print("Mejor combinación de días:", np.array(mejor_combinacion) + 1)
print("Número mínimo de días:", min_dias)
print()

# Ejecutar el algoritmo genético
mejor_solucion = algoritmo_genetico(100, 1000)
print("Mejor solución encontrada (días de grabación):")
for i, dia in enumerate(mejor_solucion):
    print(f"Día {i+1}: Tomas { [d+1 for d in dia] }")
    participacion_dia = np.zeros(num_actores, dtype=int)
    for toma in dia:
        participacion_dia = np.logical_or(participacion_dia, tomas[toma])
    print("Participación de actores en el día:", np.where(participacion_dia)[0])
    print()

```

Mejor combinación de días: [[1 2 3 4 5 6]

[7 8 9 10 11 12]]

Número mínimo de días: 2

Mejor solución encontrada (días de grabación):

Día 1: Tomas [6, 10, 8, 5, 1, 12]

Participación de actores en el día: [1 2 3 5 7 8 9 10]

Día 2: Tomas [2, 4, 3, 7, 11, 9]

Participación de actores en el día: [1 2 3 4 5 7 8 9 10]

Describe brevemente las líneas de como crees que es posible avanzar en el estudio del problema. Ten en cuenta incluso posibles variaciones del problema y/o variaciones al alza del tamaño

El problema de la planificación de sesiones de doblaje es un problema de optimización combinatoria que se puede abordar de diversas maneras. Una posible variación del problema es la inclusión de restricciones adicionales, como la disponibilidad de los actores en determinados días o la limitación del número de tomas que pueden grabar en un día. Para abordar estas restricciones, se pueden utilizar algoritmos de optimización más avanzados, como los algoritmos genéticos o los algoritmos de búsqueda tabú, que permiten encontrar soluciones aproximadas en un tiempo razonable. Otra posible variación del problema es la inclusión de costes variables para los actores, en función de su disponibilidad o de la dificultad de las tomas. En este caso, se pueden utilizar algoritmos de optimización multiobjetivo para encontrar soluciones que minimicen el coste total y maximicen la calidad de las tomas. En general, el problema de la planificación de sesiones de doblaje es un problema complejo que se puede abordar de diversas maneras, y que puede ser de interés en diversos ámbitos, como la producción cinematográfica, la televisión o la publicidad.