Algoritmos de optimización - Seminario

Nombre y Apellidos: Javier Rodríguez Juárez
Url: Link a Github
Problema:

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

(*) La respuesta es obligatoria

Importamos todas las librerías necesarias:

```
import csv
import copy
import random
import numpy as np
import itertools
import math
```

Soluciones posibles

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

Respuesta

PERMUTACIONES de n elementos: posibles ordenaciones de un conjunto de n elementos distintos, siendo n la cantidad de tomas distintas a doblar por los actores.

$P_n = n!$

$P_{30} = 30! = 265252859812191058636308480000000 = 2.65252859812191058636308480000000 \times 10^{32} \approx 265 \text{ quintillones}$

```
P_30 = math.factorial(30)
print(P_30)
```

```
265252859812191058636308480000000
```

¿Cuantas posibilidades hay teniendo en cuenta todas las restricciones.

Respuesta

VARIACIONES de n elementos tomados de r en r : posibles muestras ordenadas de r elementos distintos que se pueden extraer de un conjunto de n elementos $(r \leq n)$, siendo n la cantidad de tomas distintas a doblar por los actores y r la restricción del número máximo de tomas posibles a realizar al día.

Así, como $1 \leq r \leq 6$: el número de días distintos a realizar tomas será $m = \lceil \frac{n}{r} \rceil$ (redondeo al entero siguiente)

$V_n^{m_1,m_2,...m_k} = V_n^{m_1} + V_n^{m_2} + \ldots + V_n^{m_k} = n \times (n-1) \times \ldots \times (n-m_1+1) + n \times (n-1) \times \ldots \times (n-m_2+1) + \ldots + n \times (n-1) \times \ldots \times (n-m_k+1)$

$V_n^{m_1|m_2,m_3,m_4,m_5,m_6} = V_n^{m_1} + V_n^{m_2} + V_n^{m_3} + V_n^{m_4} + V_n^{m_5} + V_n^{m_6} =$

$= V_n^{\lceil \frac{30}{1} \rceil} + V_n^{\lceil \frac{30}{2} \rceil} + V_n^{\lceil \frac{30}{3} \rceil} + V_n^{\lceil \frac{30}{4} \rceil} + V_n^{\lceil \frac{30}{5} \rceil} + V_n^{\lceil \frac{30}{6} \rceil} = V_{30}^{30} + V_{30}^{15} + V_{30}^{m_{10}} + V_{30}^{8} + V_{30}^{6} + V_{30}^{5} =$

$= 1 + 30 \times 29 \times \ldots \times 17 \times 16 + 30 \times 29 \times \ldots \times 12 \times 11 + 30 \times 29 \times \ldots \times 10 \times 9 + 30 \times 29 \times \ldots \times 8 \times 7 + 30 \times 29 \times \ldots \times 7 \times 6$

O también:

$= \frac{30!}{30!} + \frac{30!}{15!} + \frac{30!}{10!} + \frac{30!}{8!} + \frac{30!}{6!} + \frac{30!}{5!}$

$V_{30}^{30} + V_{30}^{15} + V_{30}^{m_{10}} + V_{30}^{8} + V_{30}^{6} + V_{30}^{5} = 1 + 20284320493172736000 + 730965773291972714496000000 + 65786919596277544304640000000 + 3684067497391542481059840000000 + 221044049843492548863590400000000 =$

$= 25854990369138798933755289600 01 = 2.58549903691387989337552896001 \times 10^{30} \approx 2 \text{ quintillones}$

```
V_30_1al6 = 1
for i in range(2, 6 + 1):
    V_30_1al6 += math.factorial(30) / math.factorial(math.ceil(30/i))

print(V_30_1al6)
```

```
2.58549903691388e+30
```

Modelo para el espacio de soluciones

(*) ¿Cuál es la estructura de datos que mejor se adapta al problema? Arguméntalo.

(Es posible que hayas elegido una al principio y veas la necesidad de cambiar, arguméntalo)

Respuesta

El espacio de soluciones se representa por un vector con longitud igual al número de tomas a doblar. en este problema, el vector tendrá 30 posiciones de tal manera que la primera posición representa a la primera toma (índice 0), la segunda posición representa la segunda toma (índice 1), etc.

El contenido de este vector es el día que se realiará el doblaje de esa toma.

Por ejemplo, para la planificación:

Escena:	1	2	3	4	5	6	7	...	28	29	30
Día:	4	1	2	5	3	3	5	...	4	2	1

Correspondería el vector:

$[4, 1, 2, 5, 3, 3, 5, 3, 1, 5, 3, 2, 5, 2, 1, 2, 3, 4, 1, 5, 5, 4, 1, 2, 3, 4, 4, 4, 2, 1]$

- La primera escena se doblará el 4º día
- La segunda escena se doblará el 1º día
- La tercera escena se doblará el 2º día
- ...
- La última escena se doblará el 1º día

Además, existirá otra estrutura de datos dónde se almacenan los actores necesarios en cada toma de doblaje, representado por una matriz con tantas filas como tomas y tantas columnas como actores.

Así, la primera fila representa la primera toma, la segunda fila la siguiente toma, etc.

Dentro de cada fila (toma) la primera posición representa si es necesaria la participación del primer actor (1) o no (0), la segunda posición representa la necesidad del segundo actor, etc.

Según el modelo para el espacio de soluciones

(*) ¿Cual es la función objetivo?

Respuesta

La función objetivo es que el gasto por los servicios de los actores de doblaje sea el menor posible.

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

Respuesta

Minimización

Fuerza bruta

Diseña un algoritmo para resolver el problema por fuerza bruta

Respuesta

```
def combinaciones_tomas(escenas, min_dia, max_dia):
    num_escenas = len(escenas)
    sol_base = []

    # Generamos las posibles soluciones base
    for i in range(min_dia, max_dia + 1):
        sol = []
        dia = 1
        for j in range(math.ceil(num_escenas / i)):
            if num_escenas - len(sol) > j:
                sol += [dia] * i
                dia += 1
            elif num_escenas - len(sol) > 0:
                sol += [dia] * (num_escenas - len(sol))
        sol_base.append(sol)

    soluciones = sol_base

    # Permutamos las soluciones base
    for i in sol_base:
        permutaciones = itertools.permutations(i, num_escenas)
        for perm in list(permutaciones):
            if list(perm) not in soluciones:
                soluciones.append(list(perm))

    return soluciones
```

Esta función genera todas las posibles combinaciones de soluciones, es decir, genera $n!$ soluciones, siendo n el número de escenas.

Teniendo en cuenta que el la complejidad computacional de $itertools.permutations(i, num\ escenas)$ es de $\mathcal{O}(n^3)$, obtener el conjunto de soluciones, con esta función, tiene un coste de $\mathcal{O}(n^2 + n^3 \times n^2) = \mathcal{O}(n^5)$

No confundir con que esta función devuelve $n!$ elementos

Creamos una función que, dada una lista, nos devuelva las posiciones iguales a ese elemento.

Así, para una solución dada y un día determinado, nos devuelve un listado con las tomas que se doblarán ese día.

```
def encuentra_indices(solucion, dia):
    indices = []
    for idx, valor in enumerate(solucion):
        if valor == dia:
            indices.append(idx)
    return indices
```

El coste computacional de esta función es $\mathcal{O}(n)$

Una vez obtenidas las todas las posibles soluciones, junto con la matriz de datos (con la relación de actores en cada escena) es necesario definir una función que evalúe cada solución.

Para una solución y un día dados, calculamos cuantos actores participarán ese día haciendo un **OR** lógico entre las líneas de la matriz de datos que contiene con 0 ó 1 la ausencia o participación de los actores en la toma.

Así, suponiendo que en un día realizamos las tomas 1, 4 y 16:

Actores:	1	2	3	4	5	6	7	8	9	10
Toma 1:	1	1	1	1	1	0	0	0	0	0
Toma 4:	1	1	0	0	0	0	1	1	0	0
Toma 16:	0	0	0	1	0	0	0	0	0	1
OR	1	1	1	1	1	0	1	1	0	1

Sumando la última fila **OR** obtenemos 8, que es el total de actores que intervienen ese día.

```
def coste_solucion(datos, solucion):
    # Lista de días únicos de doblaje
    dias_doblaje = list(set(solucion))
```

```
    coste = 0
    for dia in dias_doblaje:
        # Lista de días únicos de doblaje
        indices = encuentra_indices(solucion, dia)
        a = datos[indices[0]]
        for i in range(1, len(indices)):
            b = datos[indices[i]]
            a = [x or y for x, y in zip(a,b)]
        coste+= sum(a)

    return coste
```

El coste computacional de esta función es $\mathcal{O}(n^3)$

Finalmente, recorriendo todas las soluciones, calculando el coste de cada una y obteniendo el mínimo de ellos:

```
def fuerza_bruta(datos, min_al_dia, max_al_dia):
    soluciones_tomas = combinaciones_tomas(datos, min_al_dia, max_al_dia)

    mejor_sol = []
    menor_coste = float('inf')

    for solucion in soluciones_tomas:
        coste_sol = coste_solucion(datos, solucion)
        if coste_sol < menor_coste:
            menor_coste = coste_sol
            mejor_sol = solucion

    return menor_coste, mejor_sol
```

El coste computacional de esta función es $\mathcal{O}(n)$

Calcula la complejidad del algoritmo por fuerza bruta

Respuesta

Tomando como referencia esta última función, cuyo coste computacional es $\mathcal{O}(n)$ y que la catidad de TOTAL soluciones que tiene que computar en el bucle es $n!$ (todas las posibles permutaciones), el COSTE COMPUTACIONAL POR FUERZA BRUTA es $\mathcal{O}(n!)$

Mejora del algoritmo de fuerza bruta

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

Respuesta

Cargamos los datos del archivo

```
datos = []

with open('Datos problema doblaje.csv', newline='') as csvfile:
    csv_reader = csv.reader(csvfile, delimiter=',')
    for row in csv_reader:
        datos.append(row)

display(datos[:10], '...', datos[-5:])
```

```
[['', 'Actor', '', '', '', '', '', '', '', '', '', '', ''],
```

El coste computacional de esta función es $\mathcal{O}(n)$

```
 ['2', '0', '0', '1', '1', '1', '0', '0', '0', '0', '0', '', '3'],
```

Eliminamos los datos innecesarios para la estructura de datos elegida

```
 ['5', '0', '1', '0', '1', '0', '0', '0', '1', '0', '0', '', '3'],
```

```
datos = [list(map(int, row[1:-2])) for row in datos[2:-2]]

display(datos[:5], '...', datos[-5:])
```

```
[[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, 0, 1, 1, 0, 0],
 [0, 1, 0, 1, 0, 0, 0, 1, 0, 0]]
'...'
[[1, 0, 1, 0, 1, 0, 0, 0, 1, 0],
 [0, 0, 0, 1, 1, 0, 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]]
```

El coste computacional de esta función es $\mathcal{O}(n)$

Definimos las constantes del problema y un vector solución inicializado a cero (ninguna toma tiene asignado día de realización del doblaje)

```
MIN_AL_DIA = 1  # Al menos 1 toma al día
MAX_AL_DIA = 6  # Como máximo 6 tomas al día
tomas = [0] * len(datos)
num_actores = len(datos[0])

print(tomas)

print(f"Numero de tomas: {len(tomas)}")
print(f"Numero de actores: {num_actores}")
```

```
[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, 0, 0, 0, 0]
Numero de tomas: 30
Numero de actores: 10
```

Creamos una función que dados un conjunto de tomas y un número de tomas por día (constante), de manera voraz, vaya asignando a cada toma un día de doblaje de tal manera que asigne consecutivamente todas las tomas que corresponden a un actor, y una vez completadas todas las tomas, pasa al siguiente actor. El actor por el que empieza también es un parámetro de la función y si no se indica, se empezará por el primero.

```
def escenas_x_dia_greedy(tomas, tamanyo, actor=1):
    actor -= 1
    num_actores = len(tomas[0])
    long = len(tomas)
    solucion = [0] * long

    toma_dia = 0
    dia = 0
    for act in range(num_actores):
        a = (act + actor) % num_actores
        for i, toma in enumerate(tomas):
            if toma[a] == 1 and solucion[i] == 0:
                toma_dia += 1
                solucion[i] = dia + 1
                if toma_dia + 1 > tamanyo:
                    dia += 1
                    toma_dia = 0

    return solucion
```

El coste computacional de esta función es $\mathcal{O}(n^2)$

Para cada solución, buscaremos las soluciones vecinas (Búsqueda local) para ver si mejoran las solucion de partida:

```
def genera_vecina(datos, solucion):
    mejor_sol = []
    menor_coste = float('inf')

    it_sin_mejora = 0
```

```
    max_it_sin_mejora = math.ceil(0.2 * len(solucion))    # Si no mejora en el 20% de la long. de soluciones, se detiene
    for i in range(1, len(solucion) - 1):
        for j in range(i + 1, len(solucion)):
            # Intercambiamos si los valores son distintos
            if solucion[i] != solucion[j]:
                vecina = solucion[:i] + [solucion[j]] + solucion[i + 1:j] + [solucion[i]] + solucion[j + 1:]
                coste_vecina = coste_solucion(datos, vecina)

                if coste_vecina <= menor_coste:
                    menor_coste = coste_vecina
                    mejor_sol = vecina
                    it_sin_mejora = 0
                else:
                    it_sin_mejora += 1
            if it_sin_mejora >= max_it_sin_mejora:
                break
        if it_sin_mejora >= max_it_sin_mejora:
                break

    return menor_coste, mejor_sol
```

El coste computacional de esta función es $\mathcal{O}(n^2)$

Generamos soluciones desde 1 toma por día hasta 6 tomas por día y repartiendo las escenas, iniciando por cada uno de los distintos actores

```
def algoritmo_mejorado(datos, min_al_dia, max_al_dia):
    poblacion = []
    num_actores = len(datos[0])

    for r in range(min_al_dia, max_al_dia + 1):
        for a in range(1, num_actores + 1):
            sol = escenas_x_dia_greedy(datos, r, a)
            if sol not in poblacion:
                poblacion.append(sol)

    print(f"Total soluciones generadas: {len(poblacion)}")

    mejor_sol = []
    menor_coste = float('inf')

    for solucion in poblacion:
        coste_sol = coste_solucion(datos, solucion)
        if coste_sol < menor_coste:
            menor_coste = coste_sol
            mejor_sol = solucion

        mejor_coste_vecina, mejor_sol_vecina = genera_vecina(datos, solucion)
        if mejor_coste_vecina < menor_coste:
            menor_coste = mejor_coste_vecina
            mejor_sol = mejor_sol_vecina

    return menor_coste, mejor_sol
```

El coste computacional de esta función es $\mathcal{O}(n^2 \times n^2) = \mathcal{O}(n^4)$

```
min_coste, best_sol = algoritmo_mejorado(datos, MIN_AL_DIA, MAX_AL_DIA)
print("Menor coste :",min_coste,"\nMejor solución: " , best_sol)
print("Jornadas de doblaje: ", set(best_sol))

for i, sol in enumerate(best_sol):
    print(f"- La toma {i+1:2d} se doblará el día {sol}")
```

```
 Total soluciones generadas: 60
 Menor coste : 31
 Mejor solución:  [1, 1, 1, 4, 4, 1, 1, 2, 4, 3, 1, 3, 2, 3, 4, 4, 5, 3, 5, 2, 3, 5, 5, 3, 4, 2, 2, 5, 2, 5]
 Jornadas de doblaje:  {1, 2, 3, 4, 5}
 - La toma  1 se doblará el día 1
 - La toma  2 se doblará el día 1
 - La toma  3 se doblará el día 1
 - La toma  4 se doblará el día 4
 - La toma  5 se doblará el día 4
 - La toma  6 se doblará el día 1
 - La toma  7 se doblará el día 1
 - La toma  8 se doblará el día 2
 - La toma  9 se doblará el día 4
 - La toma 10 se doblará el día 3
 - La toma 11 se doblará el día 1
 - La toma 12 se doblará el día 3
 - La toma 13 se doblará el día 2
```

```
 - La toma 14 se doblará el día 3
 - La toma 15 se doblará el día 4
 - La toma 16 se doblará el día 4
 - La toma 17 se doblará el día 5
 - La toma 18 se doblará el día 3
 - La toma 19 se doblará el día 5
 - La toma 20 se doblará el día 2
 - La toma 21 se doblará el día 3
 - La toma 22 se doblará el día 5
 - La toma 23 se doblará el día 5
 - La toma 24 se doblará el día 3
 - La toma 25 se doblará el día 4
 - La toma 26 se doblará el día 2
 - La toma 27 se doblará el día 2
 - La toma 28 se doblará el día 5
 - La toma 29 se doblará el día 2
 - La toma 30 se doblará el día 5
```

(*) Calcula la complejidad del algoritmo

Respuesta

Teniendo en cuenta las llamadas a funciones anidadas, y que la generación de soluciones inicial arroja un total de soluciones igual a $n \times r$, es decir, el producto del numéro de escenas por los distintos valores que pueden tomar las restricciones (número máximo de tomas al día), el coste computacional del algoritmo es $\mathcal{O}(n^4)$

El algoritmo mejora a la fuerza bruta en cuanto reduce el espacio de búsqueda de soluciones frente a la búsqueda de la mejor solución entre todas las soluciones posibles, aunque es necesario generar más soluciones para explorar más posibilidades.

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

Respuesta

Creamos una función que reparta las distintas tomas de manera aleatoria por días, entre un intervalo mínimo y máximo de tomas por cada día

```
def escenas_x_dia_random(tomas, min_tam, max_tam):
    long = len(tomas)
    lista = [x for x in range(long)]
    grupos = []
    solucion = [0] * long

    while long > max_tam:
        tam = np.random.randint(min_tam, max_tam + 1)
        grupos.append(tam)
        long -= tam
    grupos.append(long)

    for dia in range(len(grupos)):
        for _ in range(grupos[dia]):
            seleccion = random.choice(lista)
            solucion[seleccion] = dia + 1
            lista.remove(seleccion)

    return solucion
```

El coste computacional de esta función es $\mathcal{O}(n^2)$

Aplica el algoritmo al juego de datos generado

Respuesta

Al conjunto de soluciones anterior le añadimos las soluciones generadas aleatoriamente, para cada combinacion entre parámetros mínimo y máximo, crearemos 20 veces más soluciones como en el conjunto anterior.

```
def algoritmo_mejorado_aleatorio(datos, min_al_dia, max_al_dia):
    poblacion = []
    num_actores = len(datos[0])

    for r in range(min_al_dia, max_al_dia + 1):
        for a in range(1, num_actores + 1):
            sol = escenas_x_dia_greedy(datos, r, a)
            if sol not in poblacion:
```

```
                poblacion.append(sol)

    num_sol_random = len(poblacion) * 20

    for i in range(min_al_dia, max_al_dia + 1):
        for j in range(i, max_al_dia + 1):
            for _ in range(num_sol_random):
                sol = escenas_x_dia_random(datos, i, j)
                if sol not in poblacion:
                    poblacion.append(sol)

    print(f"Total soluciones generadas: {len(poblacion)}")

    mejor_sol = []
    menor_coste = float('inf')

    for solucion in poblacion:
        coste_sol = coste_solucion(datos, solucion)
        if coste_sol < menor_coste:
            menor_coste = coste_sol
            mejor_sol = solucion

        mejor_coste_vecina, mejor_sol_vecina = genera_vecina(datos, solucion)
        if mejor_coste_vecina < menor_coste:
            menor_coste = mejor_coste_vecina
            mejor_sol = mejor_sol_vecina

    return menor_coste, mejor_sol
```

```
min_coste_random, best_sol_random = algoritmo_mejorado_aleatorio(datos, MIN_AL_DIA, MAX_AL_DIA)
print("Menor coste :",min_coste_random,"\nMejor solución: " , best_sol_random)
print("Jornadas de doblaje: ", set(best_sol_random))

for i, sol in enumerate(best_sol_random):
    print(f"- La toma {i+1:2d} se doblará el día {sol}")
```

```
 Total soluciones generadas: 25260
 Menor coste : 31
 Mejor solución:  [1, 1, 1, 4, 4, 1, 1, 2, 4, 3, 1, 3, 2, 3, 4, 4, 5, 3, 5, 2, 3, 5, 5, 3, 4, 2, 2, 5, 2, 5]
 Jornadas de doblaje:  {1, 2, 3, 4, 5}
 - La toma  1 se doblará el día 1
 - La toma  2 se doblará el día 1
 - La toma  3 se doblará el día 1
 - La toma  4 se doblará el día 4
 - La toma  5 se doblará el día 4
 - La toma  6 se doblará el día 1
 - La toma  7 se doblará el día 1
 - La toma  8 se doblará el día 2
 - La toma  9 se doblará el día 4
 - La toma 10 se doblará el día 3
 - La toma 11 se doblará el día 1
 - La toma 12 se doblará el día 3
 - La toma 13 se doblará el día 2
 - La toma 14 se doblará el día 3
 - La toma 15 se doblará el día 4
 - La toma 16 se doblará el día 4
 - La toma 17 se doblará el día 5
 - La toma 18 se doblará el día 3
 - La toma 19 se doblará el día 5
 - La toma 20 se doblará el día 2
 - La toma 21 se doblará el día 3
 - La toma 22 se doblará el día 5
 - La toma 23 se doblará el día 5
 - La toma 24 se doblará el día 3
 - La toma 25 se doblará el día 4
 - La toma 26 se doblará el día 2
 - La toma 27 se doblará el día 2
 - La toma 28 se doblará el día 5
 - La toma 29 se doblará el día 2
 - La toma 30 se doblará el día 5
```

Pese a las soluciones exploradas, no se mejora la obtenida inicialmente por el algorítmo voraz.

El menor coste sería 31, en 5 sesiones de doblaje

Referencias

Enumera las referencias que has utilizado (si ha sido necesario) para llevar a cabo el trabajo

Respuesta

- Documentación la librería *itertools* de Python
- 'Apuntes de Algorítmica' - A. Marzal, M.J. Castro y P. Aibar - Universidad Politécnica de Valencia

Trabajos futuros

Describe brevemente las lineas 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

Respuesta

El algorítmo voraz que genera las primeras soluciones podría modificarse de tal modo que, en vez de ir asignando sucesivamente a los actores, lo haga ordanadamente: de aquellos con más tomas a doblar primero a aquellos con menos al final. Una técnica similar es utilizada en problemas de "Selección de actividades" y resuelta con algorítmos voraces obteniendo la solución óptima.

Para mejorar la solución al problema se podrían aplicar técnicas de algoritmos genéticos, realizando mutaciones de soluciones y cruces entre ellas, siendo necesario comprobar la factibilidad de las soluciones generadas por estos métodos ya que podría no cumplir con las restricciones del problema.

Invirtiendo los datos de entrada

Si cambiamos los datos de entrada, invirtiendo los actores que participan en la toma por ausencia, y viceversa

Cargamos de nuevo los datos

```
datos = []

with open('Datos problema doblaje.csv', newline='') as csvfile:
    csv_reader = csv.reader(csvfile, delimiter=',')
    for row in csv_reader:
        datos.append(row)

display(datos[:10], '...', datos[-5:])
```

```
     [['', 'Actor', '', '', '', '', '', '', '', '', '', '', ''],
      ['Toma', '1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '', 'Total'],
      ['1', '1', '1', '1', '1', '1', '0', '0', '0', '0', '0', '', '5'],
      ['2', '0', '0', '1', '1', '1', '0', '0', '0', '0', '0', '', '3'],
      ['3', '0', '1', '0', '0', '1', '0', '1', '0', '0', '0', '', '3'],
      ['4', '1', '1', '0', '0', '0', '0', '1', '1', '0', '0', '', '4'],
      ['5', '0', '1', '0', '1', '0', '0', '0', '1', '0', '0', '', '3'],
      ['6', '1', '1', '0', '1', '1', '0', '0', '0', '0', '0', '', '4'],
      ['7', '1', '1', '0', '1', '1', '0', '0', '0', '0', '0', '', '4'],
      ['8', '1', '1', '0', '0', '0', '1', '0', '0', '0', '0', '', '3']]
     '...'
     [['28', '1', '0', '0', '1', '0', '0', '0', '0', '0', '0', '', '2'],
      ['29', '1', '0', '0', '0', '1', '1', '0', '0', '0', '0', '', '3'],
      ['30', '1', '0', '0', '1', '0', '0', '0', '0', '0', '0', '', '2'],
      ['', '', '', '', '', '', '', '', '', '', '', '', ''],
      ['TOTAL', '22', '14', '13', '15', '11', '8', '3', '4', '2', '2', '', '']]
```

```
datos = [list(map(int, row[1:-2])) for row in datos[2:-2]]

display(datos[:5], '...', datos[-5:])
```

```
     [[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, 0, 1, 1, 0, 0],
      [0, 1, 0, 1, 0, 0, 0, 1, 0, 0]]
     '...'
     [[1, 0, 1, 0, 1, 0, 0, 0, 1, 0],
      [0, 0, 0, 1, 1, 0, 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]]
```

```
datos_inv = copy.deepcopy(datos)
for toma in range(len(datos)):
    for actor in range(len(datos[toma])):
        datos_inv[toma][actor] = 0 if datos[toma][actor] == 1 else 1

display(datos_inv[:10], '...', datos_inv[-5:])
```

```
[[0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
 [1, 1, 0, 0, 0, 1, 1, 1, 1, 1],
 [1, 0, 1, 1, 0, 1, 0, 1, 1, 1],
 [0, 0, 1, 1, 1, 1, 0, 0, 1, 1],
 [1, 0, 1, 0, 1, 1, 1, 0, 1, 1],
 [0, 0, 1, 0, 0, 1, 1, 1, 1, 1],
 [0, 0, 1, 0, 0, 1, 1, 1, 1, 1],
 [0, 0, 1, 1, 1, 0, 1, 1, 1, 1],
 [0, 0, 1, 0, 1, 1, 1, 1, 1, 1],
 [0, 0, 1, 1, 1, 0, 1, 1, 0, 1]]
'...'
[[0, 1, 0, 1, 0, 1, 1, 1, 0, 1],
 [1, 1, 1, 0, 0, 1, 1, 1, 1, 1],
 [0, 1, 1, 0, 1, 1, 1, 1, 1, 1],
 [0, 1, 1, 1, 0, 0, 1, 1, 1, 1],
```

Calculamos la mejor solución con el algoritmo mejorado

```
min_coste_inv, best_sol_inv = algoritmo_mejorado(datos_inv, MIN_AL_DIA, MAX_AL_DIA)
print("Menor coste :",min_coste_inv,"\nMejor solución: " , best_sol_inv)
print("Jornadas de doblaje: ", set(best_sol_inv))

for i, sol in enumerate(best_sol_inv):
    print(f"- La toma {i+1:2d} se doblará el día {sol}")
```

```
Total soluciones generadas: 60
Menor coste : 45
Mejor solución:  [5, 1, 1, 4, 1, 4, 4, 4, 4, 5, 5, 2, 2, 5, 1, 2, 1, 2, 3, 1, 5, 3, 2, 5, 3, 2, 3, 3, 3]
Jornadas de doblaje:  {1, 2, 3, 4, 5}
- La toma  1 se doblará el día 5
- La toma  2 se doblará el día 1
- La toma  3 se doblará el día 1
- La toma  4 se doblará el día 4
- La toma  5 se doblará el día 1
- La toma  6 se doblará el día 4
- La toma  7 se doblará el día 4
- La toma  8 se doblará el día 4
- La toma  9 se doblará el día 4
- La toma 10 se doblará el día 4
- La toma 11 se doblará el día 5
- La toma 12 se doblará el día 5
- La toma 13 se doblará el día 2
- La toma 14 se doblará el día 2
- La toma 15 se doblará el día 5
- La toma 16 se doblará el día 1
- La toma 17 se doblará el día 2
- La toma 18 se doblará el día 1
- La toma 19 se doblará el día 2
- La toma 20 se doblará el día 3
- La toma 21 se doblará el día 1
- La toma 22 se doblará el día 5
- La toma 23 se doblará el día 3
- La toma 24 se doblará el día 2
- La toma 25 se doblará el día 5
- La toma 26 se doblará el día 3
- La toma 27 se doblará el día 2
- La toma 28 se doblará el día 3
- La toma 29 se doblará el día 3
- La toma 30 se doblará el día 3
```

Calculamos la mejor solución con el algoritmo mejorado, incluyendo aleatoriedad

```
min_coste_random_inv, best_sol_random_inv = algoritmo_mejorado_aleatorio(datos_inv, MIN_AL_DIA, MAX_AL_DIA)
print("Menor coste :",min_coste_random_inv,"\nMejor solución: " , best_sol_random_inv)
print("Jornadas de doblaje: ", set(best_sol_random_inv))

for i, sol in enumerate(best_sol_random_inv):
    print(f"- La toma {i+1:2d} se doblará el día {sol}")
```

```
Total soluciones generadas: 25260
Menor coste : 45
Mejor solución:  [5, 1, 1, 4, 1, 4, 4, 4, 4, 5, 5, 2, 2, 5, 1, 2, 1, 2, 3, 1, 5, 3, 2, 5, 3, 2, 3, 3, 3]
Jornadas de doblaje:  {1, 2, 3, 4, 5}
- La toma  1 se doblará el día 5
- La toma  2 se doblará el día 1
- La toma  3 se doblará el día 1
- La toma  4 se doblará el día 4
- La toma  5 se doblará el día 1
- La toma  6 se doblará el día 4
- La toma  7 se doblará el día 4
- La toma  8 se doblará el día 4
- La toma  9 se doblará el día 4
- La toma 10 se doblará el día 4
- La toma 11 se doblará el día 5
- La toma 12 se doblará el día 5
- La toma 13 se doblará el día 2
```

```
- La toma 14 se doblará el día 2
- La toma 15 se doblará el día 5
- La toma 16 se doblará el día 1
- La toma 17 se doblará el día 2
- La toma 18 se doblará el día 1
- La toma 19 se doblará el día 2
- La toma 20 se doblará el día 3
- La toma 21 se doblará el día 1
- La toma 22 se doblará el día 5
- La toma 23 se doblará el día 3
- La toma 24 se doblará el día 2
- La toma 25 se doblará el día 5
- La toma 26 se doblará el día 3
- La toma 27 se doblará el día 2
- La toma 28 se doblará el día 3
- La toma 29 se doblará el día 3
- La toma 30 se doblará el día 3
```

Mezclando aleatoriamente las intervenciones de los actores en cada toma

Si cambiamos los datos de entrada, mezclando aleatoriamente los actores que participan en cada toma.

Cargamos de nuevo los datos

```
datos = []

with open('Datos problema doblaje.csv', newline='') as csvfile:
    csv_reader = csv.reader(csvfile, delimiter=',')
    for row in csv_reader:
        datos.append(row)

display(datos[:10], '...', datos[-5:])
```

```
[['', 'Actor', '', '', '', '', '', '', '', '', '', '', ''],
 ['Toma', '1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '', 'Total'],
 ['1', '1', '1', '1', '1', '1', '0', '0', '0', '0', '0', '', '5'],
 ['2', '0', '0', '1', '1', '1', '0', '0', '0', '0', '0', '', '3'],
 ['3', '0', '1', '0', '0', '1', '0', '1', '0', '0', '0', '', '3'],
 ['4', '1', '1', '0', '0', '0', '0', '1', '1', '0', '0', '', '4'],
 ['5', '0', '1', '0', '1', '0', '0', '0', '1', '0', '0', '', '3'],
 ['6', '1', '1', '0', '1', '1', '0', '0', '0', '0', '0', '', '4'],
 ['7', '1', '1', '0', '1', '1', '0', '0', '0', '0', '0', '', '4'],
 ['8', '1', '1', '0', '0', '0', '1', '0', '0', '0', '0', '', '3']]
'...'
[['28', '1', '0', '0', '1', '0', '0', '0', '0', '0', '0', '', '2'],
 ['29', '1', '0', '0', '0', '1', '1', '0', '0', '0', '0', '', '3'],
 ['30', '1', '0', '0', '1', '0', '0', '0', '0', '0', '0', '', '2'],
 ['', '', '', '', '', '', '', '', '', '', '', '', ''],
 ['TOTAL', '22', '14', '13', '15', '11', '8', '3', '4', '2', '2', '', '']]
```

```
datos = [list(map(int, row[1:-2])) for row in datos[2:-2]]

display(datos[:5], '...', datos[-5:])
```

```
[[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, 0, 1, 1, 0, 0],
 [0, 1, 0, 1, 0, 0, 0, 1, 0, 0]]
'...'
[[1, 0, 1, 0, 1, 0, 0, 0, 1, 0],
 [0, 0, 0, 1, 1, 0, 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]]
```

```
datos_shuffle = copy.deepcopy(datos)
for toma in range(len(datos_shuffle)):
    random.shuffle(datos_shuffle[toma])

display(datos_shuffle[:5], '...', datos_shuffle[-5:])
```

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

Calculamos la mejor solución con el algoritmo mejorado

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

```
min_coste_shuffle, best_sol_shuffle = algoritmo_mejorado(datos_shuffle, MIN_AL_DIA, MAX_AL_DIA)
print("Menor coste :",min_coste_shuffle,"\nMejor solución: " , best_sol_shuffle)
print("Jornadas de doblaje: ", set(best_sol_shuffle))

for i, sol in enumerate(best_sol_shuffle):
    print(f"- La toma {i+1:2d} se doblará el día {sol}")
```

```
Total soluciones generadas: 60
Menor coste : 37
Mejor solución:  [2, 2, 3, 1, 1, 2, 4, 1, 1, 2, 5, 1, 3, 2, 4, 4, 5, 1, 5, 3, 4, 3, 4, 3, 3, 5, 4, 5, 2, 5]
Jornadas de doblaje:  {1, 2, 3, 4, 5}
- La toma  1 se doblará el día 2
- La toma  2 se doblará el día 2
- La toma  3 se doblará el día 3
- La toma  4 se doblará el día 1
- La toma  5 se doblará el día 1
- La toma  6 se doblará el día 2
- La toma  7 se doblará el día 4
- La toma  8 se doblará el día 1
- La toma  9 se doblará el día 1
- La toma 10 se doblará el día 2
- La toma 11 se doblará el día 5
- La toma 12 se doblará el día 1
- La toma 13 se doblará el día 3
- La toma 14 se doblará el día 2
- La toma 15 se doblará el día 4
- La toma 16 se doblará el día 4
- La toma 17 se doblará el día 5
- La toma 18 se doblará el día 1
- La toma 19 se doblará el día 5
- La toma 20 se doblará el día 3
- La toma 21 se doblará el día 4
- La toma 22 se doblará el día 3
- La toma 23 se doblará el día 4
- La toma 24 se doblará el día 3
- La toma 25 se doblará el día 3
- La toma 26 se doblará el día 5
- La toma 27 se doblará el día 4
- La toma 28 se doblará el día 5
- La toma 29 se doblará el día 2
- La toma 30 se doblará el día 5
```

Calculamos la mejor solución con el algoritmo mejorado, incluyendo aleatoriedad

```
min_coste_random_shuffle, best_sol_random_shuffle = algoritmo_mejorado_aleatorio(datos_shuffle, MIN_AL_DIA, MAX_AL_DIA)
print("Menor coste :",min_coste_random_shuffle,"\nMejor solución: " , best_sol_random_shuffle)
print("Jornadas de doblaje: ", set(best_sol_random_shuffle))

for i, sol in enumerate(best_sol_random_shuffle):
    print(f"- La toma {i+1:2d} se doblará el día {sol}")
```

```
Total soluciones generadas: 25260
Menor coste : 37
Mejor solución:  [2, 2, 3, 1, 1, 2, 4, 1, 1, 2, 5, 1, 3, 2, 4, 4, 5, 1, 5, 3, 4, 3, 4, 3, 3, 5, 4, 5, 2, 5]
Jornadas de doblaje:  {1, 2, 3, 4, 5}
- La toma  1 se doblará el día 2
- La toma  2 se doblará el día 2
- La toma  3 se doblará el día 3
- La toma  4 se doblará el día 1
- La toma  5 se doblará el día 1
- La toma  6 se doblará el día 2
- La toma  7 se doblará el día 4
- La toma  8 se doblará el día 1
- La toma  9 se doblará el día 1
- La toma 10 se doblará el día 2
- La toma 11 se doblará el día 5
- La toma 12 se doblará el día 1
- La toma 13 se doblará el día 3
- La toma 14 se doblará el día 2
- La toma 15 se doblará el día 4
- La toma 16 se doblará el día 4
- La toma 17 se doblará el día 5
- La toma 18 se doblará el día 1
- La toma 19 se doblará el día 5
- La toma 20 se doblará el día 3
- La toma 21 se doblará el día 4
- La toma 22 se doblará el día 3
- La toma 23 se doblará el día 4
- La toma 24 se doblará el día 3
- La toma 25 se doblará el día 3
- La toma 26 se doblará el día 5
- La toma 27 se doblará el día 4
```

- La toma 28 se doblará el día 5
- La toma 29 se doblará el día 2
- La toma 30 se doblará el día 5