Algoritmos de optimización - Trabajo Práctico

Nombre y Apellidos: **Guillermo Rios Gómez**
Url: https://github.com/GuiRiGo88/03MIAR---Algoritmos-de-Optimizacion---2023/blob/main/TrabajoPractico
Google Colab: https://colab.research.google.com/github/GuiRiGo88/03MIAR---Algoritmos-de-Optimizacion---2023/blob/main/TrabajoPractico?hl=es
Problema:

> 1. **Sesiones de doblaje**

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 contrario rario ntrario

In [1]:
```
# IMPORTAR LIBRERIA Y DECLARACIÓN DE VARIABLES
import random
import numpy as np
import pandas as pd

# Datos de entrada
num_actores = 10
num_tomas = 30
costo_por_dia = 1

# Parámetros del algoritmo genético
tam_poblacion = 100
num_generaciones = 1000
tasa_mutacion = 0.01
```

¿Como represento el espacio de soluciones?

El espacio de soluciones se representa como una matriz binaria de tamaño `num_tomas` x `num_actores`. Cada entrada en la matriz es una variable binaria que indica si un actor específico está trabajando en una toma específica (1 si el actor está trabajando, 0 en caso contrario). Por lo tanto, el espacio de soluciones es el conjunto de todas las posibles matrices binarias de este tamaño.

In [2]:
```python
# Función para generar una solución aleatoria
def generar_solucion():
    solucion = [[0 for _ in range(num_tomas)] for _ in range(num_actores)]
    for j in range(num_tomas):
        actores_en_toma = random.sample(range(num_actores), random.randint(1, 6))
        for i in actores_en_toma:
            solucion[i][j] = 1
    return solucion
```

¿Cual es la función objetivo?

La función objetivo es minimizar el costo total de las sesiones de doblaje. Este costo se calcula como la suma de los costos de cada actor por cada día que trabaja. Esto se implementa en la función `calcular_costo`, que toma una solución (es decir, una asignación de actores a tomas) y devuelve el costo total de esa solución.

In [3]:
```python
# Función para calcular el costo de una solución
def calcular_costo(solucion):
    costo = 0
    for i in range(num_actores):
        for j in range(num_tomas):
            if solucion[i][j] == 1:
                costo += costo_por_dia
    return costo
```

¿Como implemento las restricciones?

Las restricciones del problema se implementan en la forma en que generamos y modificamos las soluciones. Por ejemplo, la restricción de que no es posible grabar más de 6 tomas por día se implementa en la función `generar_solucion`, que genera una solución aleatoria asegurándose de que no se seleccionen más de 6 actores para cada toma. Del mismo modo, la restricción de que cada toma debe ser grabada exactamente una vez se implementa en la función `mutar`, que modifica una solución existente cambiando la asignación de un actor a una toma, pero solo si este cambio no viola la restricción.

In [4]:
```python
# Función para mutar una solución
def mutar(solucion):
```

```
    for i in range(num_actores):
        for j in range(num_tomas):
            if random.random() < tasa_mutacion:
                solucion[i][j] = 1 - solucion[i][j]
    return solucion
```

In [5]:

```
########CUERPO DE CODIGO########

# Función para cruzar dos soluciones
def cruzar(solucion1, solucion2):
    corte = random.randint(0, num_tomas)
    hijo1 = solucion1[:corte] + solucion2[corte:]
    hijo2 = solucion2[:corte] + solucion1[corte:]
    return hijo1, hijo2

# Inicializar la población
poblacion = [generar_solucion() for _ in range(tam_poblacion)]

# Bucle principal del algoritmo genético
for _ in range(num_generaciones):
    # Calcular el costo de cada solución en la población
    costos = [calcular_costo(solucion) for solucion in poblacion]
    # Seleccionar las mejores soluciones para la próxima generación
    poblacion = [solucion for _, solucion in sorted(zip(costos, poblacion))]
    # Cruzar y mutar las soluciones para generar la próxima generación
    while len(poblacion) < tam_poblacion:
        if random.random() < 0.5:
            poblacion.append(mutar(random.choice(poblacion)))
        else:
            poblacion.extend(cruzar(random.choice(poblacion), random.choice(poblaci

# Imprimir la mejor solución encontrada
mejor_solucion = min(poblacion, key=calcular_costo)
mejor_solucion_transpuesta = list(map(list, zip(*mejor_solucion)))
# Convertir la mejor solución a un DataFrame de pandas
df = pd.DataFrame(mejor_solucion_transpuesta)

# Calcular la suma de cada fila y cada columna
df['Suma_Filas'] = df.sum(axis=1)
df.loc['Suma_Columnas'] = df.sum()

# Imprimir el DataFrame resultante
print(df)
```

	0	1	2	3	4	5	6	7	8	9	Suma_Filas
0	0	1	0	0	0	0	0	1	1	0	3
1	0	0	0	0	0	1	0	1	0	0	2
2	0	1	0	0	0	0	1	0	0	0	2
3	0	0	0	1	0	0	0	0	1	1	3
4	0	0	0	0	1	0	0	0	1	0	2
5	0	0	0	1	0	0	0	0	0	0	1
6	0	0	1	0	0	1	1	0	0	1	4
7	0	0	1	0	0	0	0	0	0	0	1
8	1	0	1	0	0	0	0	1	1	1	5
9	0	1	0	0	0	1	0	0	0	0	2
10	1	1	0	0	1	0	0	0	1	0	4
11	0	0	0	1	0	1	1	0	0	0	3
12	1	1	0	1	1	0	1	0	1	0	6
13	0	1	1	1	0	1	0	1	0	0	5
14	0	0	0	0	0	1	0	0	0	0	1
15	0	1	0	1	0	0	0	0	0	1	3
16	0	0	1	0	0	0	0	0	0	0	1
17	1	0	0	0	1	1	0	0	1	1	5
18	1	0	0	1	0	0	1	0	1	1	5
19	0	0	0	0	1	0	0	0	0	0	1
20	0	1	1	1	0	1	0	0	0	0	4
21	0	0	0	0	0	0	1	0	1	0	2
22	1	0	0	1	0	0	0	0	0	0	2
23	1	0	0	0	0	0	0	0	0	0	1
24	0	0	0	1	0	0	0	0	0	1	2
25	0	0	0	0	0	0	1	0	0	0	1
26	0	1	1	0	0	0	0	1	0	0	3
27	0	0	0	0	0	0	1	0	0	0	1
28	0	0	0	0	0	0	0	1	1	0	2
29	0	0	0	0	0	0	1	1	0	0	2
Suma_Columnas	7	9	7	10	5	8	9	7	10	7	79

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

En este caso, si denotamos el número de actores por A, el número de tomas por T, el tamaño de la población por P y el número de generaciones por G, entonces la complejidad temporal del algoritmo es aproximadamente $O(A*T*P*G)$. Esto se debe a que para cada generación, el algoritmo debe calcular la función de fitness (que tiene una complejidad de $O(A*T)$) para cada solución en la población.

In [6]:
```
print("Costo de la mejor solución:", calcular_costo(mejor_solucion))
```

```
Costo de la mejor solución: 79
```

¿Que técnica utilizo? ¿Por qué?

Utilicé un algoritmo genético para resolver este problema. Los algoritmos genéticos son técnicas de optimización y búsqueda inspiradas en la teoría de la evolución natural. Son particularmente útiles para problemas de optimización combinatoria como este, donde el

espacio de soluciones es grande y las soluciones pueden ser fácilmente codificadas como una secuencia de decisiones binarias.

Elegí un algoritmo genético por varias razones:

1. **Capacidad para manejar grandes espacios de soluciones**: Los algoritmos genéticos son capaces de explorar eficientemente grandes espacios de soluciones. En este problema, tenemos varias instancias (tomas) y atributos (actores), lo que significa que el número total de posibles asignaciones de actores a tomas (es decir, el tamaño del espacio de soluciones) es grande.

2. **Flexibilidad**: Los algoritmos genéticos son altamente flexibles y se pueden adaptar fácilmente a diferentes tipos de problemas de optimización. En este caso, podemos representar una solución como una matriz binaria y definir una función de fitness (en este caso, la función `calcular_costo`) que calcula el costo total de una solución.

3. **Robustez**: A diferencia de otros métodos de optimización que pueden quedar atrapados en óptimos locales, los algoritmos genéticos tienen una buena capacidad para evitar óptimos locales debido a su enfoque estocástico y poblacional.

4. **Paralelización**: Los algoritmos genéticos son inherentemente