Algoritmos de optimización - Trabajo Práctico

Nombre y Apellidos: Carlos Somoza Martínez Url: https://github.com/csomoza96/VIU_Optimizacion Google Colab: https://colab.research.google.com/drive/1RniTFE7DM7Q_oTHh_8Lqqft0Pns-pBDP?usp=sharing
Problema:

1. Sesiones de doblaje
2. Organizar los horarios de partidos de La Liga
3. Configuración de Tribunales

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.

Modelo

¿Como represento el espacio de soluciones?

Tomas y días de grabación: Cada toma de la película debe asignarse a uno de los días de grabación. Por lo tanto, el espacio de soluciones consiste en todas las posibles asignaciones de tomas a días de grabación. Cada solución en el espacio representa una distribución de las tomas en los días de grabación.

Variables de decisión: Las variables de decisión son las que determinan la asignación de las tomas a los días de grabación. En un enfoque de búsqueda local, las variables de decisión podrían ser binarias, donde un valor de 1 indica que una toma está asignada a un día específico, y un valor de 0 indica que no lo está.

Vecindarios: El espacio de soluciones también incluye vecindarios, que son conjuntos de soluciones cercanas entre sí. En el caso de la Búsqueda Local, un vecindario podría definirse como la posibilidad de intercambiar una toma de un día a otro, manteniendo el límite de 6 tomas por día.

¿Cual es la función objetivo?

La función objetivo es minimizar el gasto total por los servicios de los actores de doblaje.

¿Como implemento las restricciones?

Los actores del doblaje deben coincidir en las tomas en las que sus personajes aparecen juntos en las diferentes tomas. No es posible grabar más de 6 tomas por día. Para la primera restricción, debemos asegurarnos de que los actores que participan en una toma juntos estén programados para trabajar en el mismo día. Para la segunda restricción, debemos garantizar que el número total de tomas por día no exceda 6. Se pasará por un constructor la restricción a tener en cuenta a la hora del cálculo de las opciones

Análisis

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

El enfoque de búsqueda local utilizado en el ejemplo proporcionado tiene una complejidad que depende principalmente del número de iteraciones y del tamaño del vecindario que se explora en cada iteración. Si tenemos n tomas y m actores,la solución inicial aleatoria tiene una complejidad de O(nlogn) debido al uso de la función de mezcla aleatoria. En cuanto a la búsqueda de vecinos para cada iteración tiene una complejidad de n^2^, ya que se deben considerar todas las combinaciones posibles de intercambio de tomas entre días. Si hay n tomas y m actores, y se graban en d días, el espacio de soluciones tiene una cantidad total de combinaciones que puede ser bastante grande, del orden de n^d^

```
from os import path
import pandas as pd
# from google.collab import drive
nombres_columnas = ['Toma', 'A1', 'A2', 'A3', 'A4', 'A5',
                    'A6', 'A7', 'A8', 'A9', 'A10', 'NaN',
                    'Total']
#Renombre archivo y lo descargué como un csv lo adjuntare junto al
trabajo en github
ruta=path.join('30tomas10actores.csv')
data=pd.read_csv(ruta,skiprows=[0],names=nombres_columnas)
dataDroped=data.drop(columns=['NaN'])

dataDropedFilter=dataDroped.set_index("Toma")
dataFilter=dataDropedFilter[1:].copy(deep=True)
dataFilter=dataFilter.dropna(how='any')
display(dataFilter)
actores = list(dataFilter.columns[:-1])
tomas = list(dataFilter.index)
```

```
       A1   A2   A3   A4   A5   A6   A7   A8   A9  A10 Total
Toma
```

1	1.0	1.0	1.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	5
2	0.0	0.0	1.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	3
3	0.0	1.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	3
4	1.0	1.0	0.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	4
5	0.0	1.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	3
6	1.0	1.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	4
7	1.0	1.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	4
8	1.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	3
9	1.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	3
10	1.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	4
11	1.0	1.0	1.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	5
12	1.0	1.0	1.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	5
13	1.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	3
14	1.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	3
15	1.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	3
16	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	1.0	2
17	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2
18	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	2
19	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2
20	1.0	0.0	1.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	4
21	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0	2
22	1.0	1.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	4
23	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2
24	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	2
25	1.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	1.0	4
26	1.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	4
27	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	2
28	1.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	2
29	1.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	3
30	1.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	2

Diseño

¿Que técnica utilizo? ¿Por qué? Utilizo la técnica de la búsqueda Local; comienzo con una solución inicial y busco iterativamente vecinos de esa solución, intentando mejorarla mediante pequeños cambios. Si se encuentra un vecino que mejora la solución actual, se acepta como nueva solución actual y se continúa la búsqueda. Este proceso se repite hasta que no se pueden encontrar vecinos que mejoren la solución actual o hasta que se alcanza un criterio de parada predefinido, como un número máximo de iteraciones. En este problema, el espacio de búsqueda no es muy grande, lo que hace que la búsqueda local sea una buena opción en términos de eficiencia, además de más sencillo de implementar

```
import random

class PlanificacionDoblaje:
    def __init__(self, tomas, actores, max_tomas_por_dia):
        self.tomas = tomas
```

```
        self.actores = actores
        # Restricciones
        self.max_tomas_por_dia = max_tomas_por_dia
        self.solucion_actual = None

   # Generar una solución inicial asignando aleatoriamente tomas a
días
    def generar_solucion_inicial(self):
        random.shuffle(self.tomas)
        self.solucion_actual = [self.tomas[i:i+self.max_tomas_por_dia]
for i in range(0, len(self.tomas), self.max_tomas_por_dia)]
   # Calcular el costo total de la solución
    # Esta función recorre cada día en la solución y calcula el número
de actores
    # que necesitan estar presentes en ese día.
    # El costo total se calcula como la suma del número de actores en
cada día.
    def calcular_costo(self, solucion):
        costo_total = 0
        for dia in solucion:
            actores_dia = set()
            for toma in dia:

actores_dia.update(dataFilter.loc[toma, :].dropna().index)
            costo_total += len(actores_dia)
        return costo_total

    def buscar_mejor_vecino(self, solucion_actual):
        mejor_vecino = None
        mejor_costo = float('inf')

        # Generar vecinos intercambiando tomas entre días
        for i in range(len(solucion_actual)):
            for j in range(len(solucion_actual)):
                if i != j:
                    vecino = solucion_actual.copy()
                    toma = vecino[i].pop()
                    vecino[j].append(toma)
                    costo_vecino = self.calcular_costo(vecino)
                    if costo_vecino < mejor_costo:
                        mejor_vecino = vecino
                        mejor_costo = costo_vecino

        return mejor_vecino, mejor_costo

    def buscar_solucion(self, max_iteraciones=1000):
        # Generar solución inicial
        self.generar_solucion_inicial()
        mejor_solucion = self.solucion_actual.copy()
        mejor_costo = self.calcular_costo(mejor_solucion)
```

```
        iteracion = 0
        while iteracion < max_iteraciones:
            vecino, costo_vecino =
self.buscar_mejor_vecino(self.solucion_actual)
            if costo_vecino < mejor_costo:
                mejor_solucion = vecino
                mejor_costo = costo_vecino
            else:
                break  # No se encontró una mejora significativa en
esta iteración
            self.solucion_actual = vecino
            iteracion += 1
        return mejor_solucion, mejor_costo

def obtenerActoresToma(num):
    a=dataFilter.iloc[num]==1.0
    x=a[a==True]
    columnas_true = x.index[x].tolist()
    return columnas_true
def obtenerActoresDia(lista_list,toret):
    for num in lista_list:
        toret.extend(obtenerActoresToma(num))
    print(toret)
lista_actores=[]
# No más de 6 tomas por día
max_tomas_por_dia = 6
# Instanciamos el problema
planificador = PlanificacionDoblaje(tomas, actores, max_tomas_por_dia)
# Buscaamos solución
mejor_solucion, mejor_costo = planificador.buscar_solucion()
# Muestra de resultado resultados
for i, dia in enumerate(mejor_solucion, 1):

    print(f"Día {i}: {dia} ")

print("Costo total de servicios de doblaje:", mejor_costo)
```

```
Día 1: ['7', '14', '17', '8', '11', '4']
Día 2: ['20', '22', '27', '26', '6', '12']
Día 3: ['10', '1', '2', '21', '19', '18']
Día 4: ['24', '5', '3', '30', '15', '13']
Día 5: ['29', '16', '23', '28', '25', '9']
Costo total de servicios de doblaje: 55
```