Algoritmos de optimización - Trabajo Práctico

Nombre y Apellidos: Francisco Joaquin Murcia Gomez

Url: https://github.com/fmurciag/03MIAR-Algoritmos-de-Optimizacion/blob/main/trabajo_final/Trabajo_Francisco_Joaquin_Murcia_Gomez.ipynb

Google Colab: https://colab.research.google.com/drive/1rPH289DNQ2eSy72MukgXAT71wo4gi4Ra?usp=sharing

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

Modelo

¿Como represento el espacio de soluciones?

La estructura de datos que ha elegido para representar la solución es una lista de listas, donde cada lista interna representa las tomas asignadas a un día específico de grabación, un ejemplo de solución seria:

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

donde cada sublista representa un día de grabación, del primer día al quinto día, en cada sublista loa números representan las tomas que se van a rodar ese día, en ejemplo dado, en el día 2 se van a rodar las tomas 5, 19, 6, 9, 21 y 2.

A la hora de imprimir el resultado he decidido que me imprima el orden de las tomas y hacer una tabla donde se represente los días, tomas a rodar, el número de actores convocados y el costo del día:

```
Calendario de sesiones:
Orden de las tomas: [18, 24, 7, 2, 9, 28, 26, 21, 16, 8, 13, 11, 25,
```

```
19, 30, 17, 22, 23, 29, 1, 20, 12, 14, 10, 15, 4, 3, 5, 27, 6]
Horario:
Día                  Tomas  Numero de actores  Coste de la sesion (€)
    1    18, 24, 7, 2, 9, 28                  6
180
    2  26, 21, 16, 8, 13, 11                  9
270
    3 25, 19, 30, 17, 22, 23                  5
150
    4  29, 1, 20, 12, 14, 10                  7
210
    5     15, 4, 3, 5, 27, 6                  6
180

Coste Total:: 990€
```

¿Cual es la función objetivo?

La función objetivo sería minimizar el numero ve veces que tengo que convocar a los actores (A_i) para que el coste de la producción sea el mínimo.

$$\text{Minimizar } Z = \sum_{i=1}^{D} A_i$$

Aquí, Z es el objetivo que buscamos minimizar, que representa el total de actores-día necesarios para completar la grabación, sumando los actores necesarios en cada uno de los D días de grabación. La meta es organizar el calendario de grabación de manera que este total (Z) sea lo más bajo posible, lo que indirectamente minimizará el costo total bajo la premisa de que el costo por actor por día es constante y no se considera en la optimización directamente.

¿Como implemento las restricciones?

Tendremos que aplicar dos principales restricciones:

- Restricción de tomas por día: Se asegura de que cada lista interna no contenga más de 6 tomas, lo que refleja la limitación de no poder grabar más de 6 tomas por día.
- Asignación de todas las tomas: Se debe asegurar que cada toma esté asignada a algún día.

Análisis

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

Sin restricciones, el número total de posibilidades es $\left(\binom{n}{0}+\binom{n}{1}+\ldots+\binom{n}{n-1}+\binom{n}{n}\right)n!=2^n n!$ tomas resulta en aproximadamente $2.8 \cdot 10^{41}$.

Con restricciones, considerando solo hasta 6 tomas por día, el número total de posibilidades se calcula como $\left(\binom{n}{0} + \binom{n}{1} + ... + \binom{n}{6} \right) n! $, resulta en aproximadamente $2.04 \cdot 10^{38}$.

Diseño

```
import numpy as np
import itertools
import pandas as pd
import random
TABLA_ESCENAS = np.array([
    [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, 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, 1, 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],
    [1, 1, 0, 0, 0, 0, 1, 0, 0, 0],
    [0, 0, 0, 1, 0, 0, 0, 0, 0, 1],
    [1, 0, 1, 0, 0, 0, 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, 1, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 1, 0, 1, 0, 0],
    [1, 1, 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, 1],
    [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]
])
N_ESCENAS, N_ACTORES = TABLA_ESCENAS.shape
N_DIAS = (N_ESCENAS + 5) // 6
MAX_TOMAS_POR_DIA = 6
COSTO_POR_ACTOR_POR_DIA = 1
N_INTENTOS_MAX = 10
```

```
C:\Users\Usuario\AppData\Local\Temp\ipykernel_15316\2764476722.py:3:
DeprecationWarning:
Pyarrow will become a required dependency of pandas in the next major
release of pandas (pandas 3.0),
(to allow more performant data types, such as the Arrow string type,
and better interoperability with other libraries)
but was not found to be installed on your system.
If this would cause problems for you,
please provide us feedback at
https://github.com/pandas-dev/pandas/issues/54466

  import pandas as pd
```

```python
def print_schedule(session_order=list(range(N_ESCENAS))):
    schedule = [session_order[n : n + MAX_TOMAS_POR_DIA] for n in
range(0, len(session_order), MAX_TOMAS_POR_DIA)]
    schedule_details = []

    print("\n\nCalendario de sesiones:")
    print(f"Orden de sesiones: {[x + 1 for x in session_order]}")

    total_cost = 0
    for day_num, day in enumerate(schedule, start=1):
        shots = [x + 1 for x in day]
        sum_shots = np.sum(TABLA_ESCENAS[day, :], axis=0)
        num_actors = np.count_nonzero(sum_shots)
        cost_per_day = num_actors * COSTO_POR_ACTOR_POR_DIA
        total_cost += cost_per_day

        schedule_details.append(
            {
                "Día": day_num,
                "Tomas": ", ".join(map(str, shots)),
                "Numero de actores": num_actors,
                "Coste de la sesion (€)": cost_per_day,
            }
        )

    df_schedule = pd.DataFrame(schedule_details)

    print("Horario:")
    print(df_schedule.to_string(index=False))

    print(f"\nCoste Total:: {total_cost}€")
def evaluar_solucion(solucion, data):
    coste = 0
    for sesion in solucion:
        data_sesion = data[sesion, :]
        coste += np.any(data_sesion != 0, axis=0).sum()
```

```
    return coste
```

Fuerza bruta

La fuerza bruta es conceptualmente simple y sencillo de implementar. El enfoque de fuerza bruta, aunque garantiza encontrar la solución óptima al evaluar todas las combinaciones posibles de tomas y actores por día, resulta extremadamente ineficiente para este problema grandes debido al vasto número de combinaciones a considerar, lo que lleva a tiempos de ejecución muy largos. A pesar de que las restricciones del problema ayudan a reducir este número, la escala del problema hace que el método siga siendo poco práctico.

```
BEST_COST = float("inf")
BEST_SCHEDULE = np.array([])


def get_shot_combinations(remaining_shots, n):
    """Genera todas las combinaciones posibles de tomas a partir de
las tomas restantes."""
    return np.array(list(itertools.combinations(remaining_shots, n)),
dtype=int)


def calculate_actors_in_shots(shots):
    """Calcula el número de actores que participan en un conjunto de
tomas."""
    return len(np.unique(TABLA_ESCENAS[shots].nonzero()[1]))


def save_posible_best_schedule(selected_shots, total_actors):
    global BEST_COST
    global BEST_SCHEDULE

    if total_actors < BEST_COST:
        BEST_COST = total_actors
        BEST_SCHEDULE = selected_shots.copy()
        #print_schedule(BEST_SCHEDULE.flatten())
        print(f"Coste: {BEST_COST*COSTO_POR_ACTOR_POR_DIA}")


def generate_schedule_brute_algorithm(remaining_shots, selected_shots,
total_actors):
    """Ejecuta el algoritmo por fuerza bruta"""

    if len(remaining_shots) == 0:
        save_posible_best_schedule(selected_shots, total_actors)
        return

    shot_combinations = get_shot_combinations(remaining_shots,
```

```
MAX_TOMAS_POR_DIA)

    for shots in shot_combinations:
        actors = calculate_actors_in_shots(shots)
        new_selected_shots = np.vstack((selected_shots, shots))
        new_total_actors = total_actors + actors
        new_remaining_shots = np.array([shot for shot in
remaining_shots if shot not in shots])

        generate_schedule_brute_algorithm(new_remaining_shots,
new_selected_shots, new_total_actors)

initial_schedule = np.arange(N_ESCENAS)
np.random.shuffle(initial_schedule)
try:
    generate_schedule_brute_algorithm(initial_schedule, np.empty((0,
MAX_TOMAS_POR_DIA), dtype=int), 0)
except KeyboardInterrupt:
    print_schedule(BEST_SCHEDULE.flatten())
    #print(evaluar_solucion(BEST_SCHEDULE,TABLA_ESCENAS))
```

```
Coste: 36
Coste: 35
Coste: 34
Coste: 33
Coste: 32


Calendario de sesiones:
Orden de sesiones: [15, 28, 3, 11, 13, 22, 20, 9, 29, 18, 24, 12, 10,
8, 26, 27, 23, 14, 2, 6, 1, 19, 17, 7, 4, 25, 30, 5, 16, 21]
Horario:
 Día                 Tomas  Numero de actores  Coste de la sesion (€)
   1 15, 28, 3, 11, 13, 22                  7                       7
   2 20, 9, 29, 18, 24, 12                  6                       6
   3 10, 8, 26, 27, 23, 14                  7                       7
   4    2, 6, 1, 19, 17, 7                  5                       5
   5  4, 25, 30, 5, 16, 21                  7                       7

Coste Total:: 32€
```

Algoritmo voraz

Utilizar una técnica voraz es interesante porque permite construir una solución óptima a partir de soluciones óptimas parciales, como, por ejemplo, organizar las tomas por día. Si se encuentra una forma óptima de organizar las tomas para un día dado, esta solución no se ve tan afectada por cómo se organicen las tomas en los otros días. En otras palabras, la solución para un día específico (una sesión de doblaje) se elige sin necesidad de reconsiderar las decisiones tomadas para las sesiones anteriores. Esto sugiere que el problema se descompone en subproblemas más pequeños (cada día de doblaje), para los cuales se buscan soluciones óptimas locales

(minimizar el costo por día), asumiendo que estas contribuyen al óptimo global (minimizar el costo total)

La función voraz elige la mejor toma para añadir a la sesión actual, buscando minimizar el total de actores involucrados. Evalúa el impacto de añadir cada toma disponible a la sesión en términos del número de actores adicionales que implicaría. La toma seleccionada es aquella que, una vez añadida, resulta en el menor incremento posible en el número de actores diferentes requeridos para el día.

Por lo tanto, en el algoritmo propuesto la función `greedy_selection` elige la "mejor" toma para añadir a la sesión actual basándote en el costo actual de la solución. Este costo se intenta minimizar seleccionando la toma que, al añadirla a la sesión, resulta en el menor aumento posible del costo total. La función `greedy_search` añade esa toma seleccionada por la función `greedy_selection` a la lista de escenas de ese día y elimina ese día del total de escenas disponibles, cuando día de grabación esta completa rellena el siguiente.

El algoritmo pose una complejidad de $O\left(n^2 \cdot p \cdot d\right)$ siendo:

- n: nº total de tomas
- p: nº de tomas por sesión, en nuestro caso es 6
- d: nº de días de grabación, en nuestro caso hemos considerado el mínimo posible que serían $\frac{n}{p}$

```
import numpy as np
from copy import deepcopy

MAX_TOMAS_POR_DIA = 6


def evaluate_solution(solution, data):
    """evalua la solucion"""
    cost = 0
    for session in solution:
        cost += np.any(data[session, :] != 0, axis=0).sum()

    return cost

def sort_scenes(scenes, data):
    """Ordena las escenas"""
    sum_rows = np.sum(data, axis=1)
    sorted_indices = np.argsort(sum_rows)[::-1]
    return scenes[sorted_indices]

def greedy_selection(solution, session, scenes, data):
    """Selecciona las mejores tomas para esa sesion"""
    best_scene = scenes[0]

    temp_solution = deepcopy(solution)
    temp_session = deepcopy(session)
```

```
    temp_solution.append(temp_session + [best_scene])
    lowest_cost = evaluate_solution(temp_solution, data)
    for i in range(1, len(scenes)):
        temp_solution = deepcopy(solution)
        temp_session = deepcopy(session)
        temp_solution.append(temp_session + [scenes[i]])
        cost = evaluate_solution(temp_solution, data)

        if cost < lowest_cost:
            best_scene = scenes[i]
            lowest_cost = cost

    return best_scene

def greedy_search(data):
    """Construye el horario minimizando el coste por dia"""
    solution = []
    scenes = np.arange(data.shape[0])
    scenes = sort_scenes(scenes, data)

    session_count = 0
    while scenes.shape[0] > 0:
        session = []
        for _ in range(MAX_TOMAS_POR_DIA):
            best_scene = greedy_selection(solution, session, scenes,
data)
            session += [best_scene]
            idx = np.argwhere(scenes == best_scene)
            scenes = np.delete(scenes, idx)
        solution.append(session)
        session_count += 1

    return solution


sol = greedy_search(TABLA_ESCENAS)
print("Costo: ", evaluate_solution(sol, TABLA_ESCENAS))
print_schedule(np.array(sol).flatten())
```

```
Costo:  28


Calendario de sesiones:
Orden de sesiones: [16, 27, 13, 28, 30, 25, 17, 19, 23, 14, 18, 24,
21, 5, 8, 9, 12, 22, 3, 15, 4, 11, 1, 6, 2, 20, 26, 7, 10, 29]
Horario:
 Día                  Tomas  Numero de actores  Coste de la sesion (€)
   1 16, 27, 13, 28, 30, 25                  5                       5
   2 17, 19, 23, 14, 18, 24                  3                       3
   3    21, 5, 8, 9, 12, 22                  6                       6
```

```
   4     3, 15, 4, 11, 1, 6                    7                          7
   5   2, 20, 26, 7, 10, 29                    7                          7

Coste Total:: 28€
```

Algoritmo genético

Los algoritmos genéticos se están volviendo muy populares, ofrecen implementaciones sencillas y obtiene buenos resultados con un coste bajo. En nuestro caso para resolver este problema con un algoritmo genético creamos una población inicial de p individuos completamente aleatorios, y generamos g generaciones.

La función de cruce `crossover` crea un punto de cruce aleatorio y se divide al padre y madre en dos, crea un primer individuo uniendo un primer trozo del "padre" y el segundo trozo mitad de la "madre" y un mellizo con el primer trozo de la "madre" y el segundo trozo del "padre", finalmente se escoge con hijo ganador al que mejor resultado tenga. Ejemplo:

```
Padre = [1, 2, 3, 4, 5, 6]
Madre = ['a', 'b', 'c', 'd', 'e', 'f']
Hijo 1 = [1, 2, 3, 'd', 'e', 'f'] -> coste = 8
Hijo 2 = ['a', 'b', 'c', 4, 5, 6] -> coste = 3
Hijo 2 nace
```

Como esta función de cruce puede hacer que se dupliquen elementos, aprovechamos y mutamos a los individuos con la función de mutación `mutate` que coge los elementos repetidos y los sustituye por elementos aleatorios que no están en esa lista.

```
Individuo original: [1, 2, 3, 2, 1]
Individuo después de la mutación: [1, 2, 3, 4, 5]
```

El algoritmo pose una complejidad de $O\left(n^2 \cdot p \cdot g\right)$ siendo:

- n: nº total de tomas
- p: tamaño de la población
- g: nº de generaciones

```python
def generate_population(size):
    """Genera una población de secuencias de escenas aleatorias."""
    return [np.random.permutation(N_ESCENAS).tolist() for _ in
range(size)]


def calculate_cost(schedule):
    """Calcula el costo total basado en el número de actores por
día."""
    day_sums = np.sum(TABLA_ESCENAS[schedule, :], axis=0)
    actors = np.count_nonzero(day_sums)
```

```
    return actors * COSTO_POR_ACTOR_POR_DIA


def fitness(individual):
    """Calcula la aptitud de un individuo basada en el costo total."""
    horario = [individual[n : n + MAX_TOMAS_POR_DIA] for n in range(0,
N_ESCENAS, MAX_TOMAS_POR_DIA)]
    costo_total = sum(calculate_cost(dia) for dia in horario)
    # print(f"Costo: {costo_total} ")
    return costo_total


def crossover(ind1, ind2):
    """Realiza el cruzamiento entre dos individuos y devuelve el mas
prometedor."""
    size = len(ind1)
    crossover_point = random.randint(1, size - 1)
    new_ind1 = ind1[:crossover_point] + ind2[crossover_point:]
    new_ind2 = ind2[:crossover_point] + ind1[crossover_point:]
    new_ind1 = mutate(new_ind1)
    new_ind2 = mutate(new_ind2)
    if fitness(new_ind1) < fitness(new_ind2):
        return new_ind1
    else:
        return new_ind2


def mutate(ind):
    """Aplica una mutación intercambiando los elementos repetidos por
el cruce por valores aleatorios."""
    for i in range(len(ind)):
        if ind.count(ind[i]) > 1:
            ind[i] = random.choice([valor for valor in
range(N_ESCENAS) if valor not in ind])
    return ind


def selection(population, fitnesses, num_parents, num_direct_copies):
    """Selecciona los mejores individuos para la siguiente
generación."""
    sorted_idx = np.argsort(fitnesses)[:num_parents]
    parents = [population[int(i)] for i in sorted_idx]
    offsprings = parents[:num_direct_copies]
    for i in range(num_direct_copies, min(num_parents - 1,
len(parents) - 1)):
        offsprings.append(crossover(parents[i], parents[i + 1]))
    return offsprings


def generate_schedule_genetic_algorithm(population_size=500,
```

```
generations=500):
    """Ejecuta el algoritmo genético."""
    population = generate_population(population_size)
    for _ in range(generations):
        fitnesses = [fitness(ind) for ind in population]
        population = selection(population, fitnesses, population_size
// 2, population_size // 4)
    best_session = min(population, key=fitness)
    return best_session


best_session =
generate_schedule_genetic_algorithm(population_size=900,
generations=400)
print_schedule(best_session)



Calendario de sesiones:
Orden de sesiones: [16, 5, 8, 11, 25, 12, 19, 13, 2, 27, 20, 28, 7, 6,
30, 1, 22, 9, 29, 24, 14, 18, 17, 23, 21, 10, 15, 4, 26, 3]
Horario:
 Día                  Tomas  Numero de actores  Coste de la sesion (€)
   1   16, 5, 8, 11, 25, 12                  8                       8
   2  19, 13, 2, 27, 20, 28                  4                       4
   3     7, 6, 30, 1, 22, 9                  5                       5
   4 29, 24, 14, 18, 17, 23                  4                       4
   5   21, 10, 15, 4, 26, 3                  8                       8

Coste Total:: 29€
```

Conclusión

En este trabajo, se exploraron tres algoritmos para optimizar el problema de coordinar las tomas de las sesiones de doblaje, cada uno con características únicas:

1. **Algoritmo de Fuerza Bruta:** Logró identificar la solución más óptima de los 3, demostrando su capacidad para encontrar el mejor horario posible (27) ya que prueba todas las combinaciones posibles. Sin embargo, su elevado coste temporal (aproximadamente $O(n!)$), tomando hasta 5 minutos para ofrecer esa solución, limita severamente su aplicabilidad práctica.

2. **Algoritmo Voraz:** Se destacó por su capacidad para ofrecer resultados próximos al óptimo de manera casi instantánea. A pesar de que su implementación resultó ser la más compleja de los 3, este algoritmo demostró un balance óptimo entre calidad de resultados y eficiencia temporal ya que pesca a tener un coste cuadrático ($O(n^2 \cdot p \cdot d)$) obtiene un resultado de 28, posicionándose como la mejor solución

3. **Algoritmo Genético:** Este algoritmo no alcanzó resultados satisfactorios, al principio la función cruce me unía la primera mitad del padre y la segunda de la madre, lo que otorgaba valores en torno a 35, no fue hasta implementar la estrategia de cruzamiento selectivo que empleo actualmente que los resultaron mejoraron, sin embargo no logra obtener resultados de la estrategia voraz, el mejor resultado ha sido 29 con una población de 900 y 400 generaciones, al poseer una complejidad de $O(n^2 \cdot p \cdot g)$ hace que el algoritmo sea más lento que el voraz debido a los escalares.

En conclusión, entre estos tres algoritmos, considero que el algoritmo voraz proporciona los mejores resultados, teniendo en cuenta tanto la calidad de las soluciones como el tiempo de computación. Sin embargo, es importante destacar que el enfoque voraz actual se está quedando atrapado en un mínimo local. Una propuesta de mejora sería utilizar una búsqueda tabú a partir del resultado del algoritmo voraz para escapar de ese mínimo y alcanzar los valores dados por el algoritmo de fuerza bruta. Por último, al implementar el algoritmo, se observó que las dos funciones más importantes son el mutado y el cruce, ya que al aplicar una nueva estrategia de cruce se mejoraron significativamente los resultados, lo que indica que probablemente haya margen para mejorar el resultado mediante un cambio de estrategias de estas funciones.

Caso real

Según https://www.20minutos.es/cinemania/noticias/cuanto-cobra-actor-doblaje-cifra-futuro-cine-5136629/, el salario de un actor de doblaje de películas en España es de entre 50 y 120 euros por jornada, elegimos la mitad que serían 85€ dando el siguiente horario:

```
Costo:  28


Calendario de sesiones:
Orden de sesiones: [16, 27, 13, 28, 30, 25, 17, 19, 23, 14, 18, 24,
21, 5, 8, 9, 12, 22, 3, 15, 4, 11, 1, 6, 2, 20, 26, 7, 10, 29]
Horario:
 Día                  Tomas  Numero de actores  Coste de la sesion (€)
   1 16, 27, 13, 28, 30, 25                  5                     425
   2 17, 19, 23, 14, 18, 24                  3                     255
   3    21, 5, 8, 9, 12, 22                  6                     510
   4     3, 15, 4, 11, 1, 6                  7                     595
   5   2, 20, 26, 7, 10, 29                  7                     595

Coste Total:: 2380€
```