Algoritmos de optimización - Trabajo Práctico

Nombre y Apellidos: Juan Pablo Rodriguez Rojas
Url: https://github.com/.../03MAIR---Algoritmos-de-Optimizacion---/tree/master/TrabajoPractico
Google Colab: https://colab.research.google.com/drive/xxxxxxxxxxxxxxx
Problema:

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

Descripción del problema:

Problema 1. Organizar sesiones de doblaje(I)

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

Modelo

¿Cómo represento el espacio de soluciones?

Se representa mediante una población de individuos, donde cada individuo es una posible solución al problema. En este contexto, cada solución (individuo) es un arreglo de enteros donde cada elemento representa el día asignado para realizar una toma específica. Por ejemplo, si tenemos 30 tomas y un máximo de 5 días para filmarlas, una solución podría ser un arreglo de 30 elementos con valores en el rango de 1 a 5, indicando el día en que se realizará cada toma.

¿Cuál es la función objetivo?

La función objetivo o funcion de aptitud evalúa qué tan buena es una solución (individuo) dentro del espacio de

soluciones. El objetivo es minimizar el costo total, que se calcula en base a:

- El número de días utilizados: buscamos usar eficientemente todos los días disponibles sin sobrepasarlos.
- El número de actores requeridos cada día: se penaliza el exceso en el número de tomas por día para no sobrepasar el límite de 6 tomas, intentando minimizar el número total de actores requeridos a lo largo de todos los días.
- La función de aptitud devuelve un valor negativo, donde un valor más negativo indica una solución de menor costo (mejor solución).

¿Cómo implemento las restricciones?

Las restricciones del problema se implementan de la siguiente manera:

Restricción de no más de 6 tomas por día: Se penaliza fuertemente en la función de aptitud cualquier solución que asigne más de 6 tomas a un solo día, reduciendo significativamente su valor de aptitud.

Restricción de usar todos los días disponibles de manera efectiva: Se penaliza las soluciones que no utilizan todos los días disponibles, incentivando así el uso eficiente de cada día dentro del límite máximo de días permitidos.

Estas penalizaciones ayudan a guiar el proceso de selección natural del algoritmo genético hacia soluciones que cumplan con las restricciones del problema. La implementación de restricciones mediante penalizaciones en la función de aptitud es una estrategia común en los algoritmos genéticos, permitiendo explorar el espacio de soluciones mientras se favorecen aquellas que cumplen con los criterios deseados.

```
#Algoritmo
!pip install numpy
import numpy as np
import random

# Matriz de disponibilidad de actores para las escenas
data = np.array([
    [1,0,0,1,0,1,1,1,1,1,1,1,1,1,1,0,1,0,1,1,0,1,1,0,1,1,0,1,1,1], # Actor 1
    [1,0,1,1,1,1,1,1,1,1,1,1,0,0,1,0,0,0,0,0,0,1,0,0,1,0,0,0,0,0], # Actor 2
    [1,1,0,0,0,0,0,0,0,1,1,1,0,1,0,0,1,1,1,1,0,1,1,1,0,1,0,0,0,0], # Actor 3
    [1,1,0,0,1,1,1,0,1,0,0,1,1,0,0,1,0,0,0,1,0,1,0,0,1,0,1,1,1,0], # Actor 4
    [1,1,1,0,0,1,1,0,0,0,1,0,1,0,0,0,0,0,0,1,0,0,0,0,0,1,1,0,1,0], # Actor 5
    [0,0,0,0,0,0,0,1,0,1,0,1,0,1,0,0,0,1,0,0,1,0,0,1,0,0,0,0,1,0], # Actor 6
    [0,0,1,1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0], # Actor 7
    [0,0,0,1,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0], # Actor 8
    [0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0], # Actor 9
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0]  # Actor 10
])

num_actors, num_scenes = data.shape
population_size = 50
num_generations = 100
max_days = 5  # Ajustado a 5 días como máximo
max_scenes_per_day = 6

def initialize_population():
    population = []
    for _ in range(population_size):
        individual = np.random.choice(range(1, max_days + 1), size=num_scenes, replace=True
        population.append(np.array(individual))
    return population

def calculate_fitness(individual):
    total_cost = 0
    days_workload = {day: [] for day in range(1, max_days + 1)}

    # Llenar days_workload con las escenas asignadas a cada día
    for scene, day in enumerate(individual):
        days_workload[day].append(scene)

    # Aplicar penalizaciones
    for day, scenes in days_workload.items():
        # Penalización por exceder el límite de tomas por día
        if len(scenes) > max_scenes_per_day:
            total_cost -= (len(scenes) - max_scenes_per_day) * 1000  # Aumentar la penaliza

        # Costo por cada actor requerido ese día
        actors_today = set()
        for scene in scenes:
            actors_today.update([actor for actor in range(num_actors) if data[actor, scene]
        total_cost -= len(actors_today)
```

```
        # Penalización por no utilizar todos los días de manera efectiva
        used_days = len([day for day in days_workload if days_workload[day]])
        if used_days < max_days:
            total_cost -= (max_days - used_days) * 1000  # Aumentar la penalización para incent

        return total_cost

    def select_parents(population):
        tournament_size = 5
        tournament = random.sample(population, tournament_size)
        tournament.sort(key=calculate_fitness, reverse=True)
        return tournament[0], tournament[1]

    def crossover(parent1, parent2):
        crossover_point = random.randint(1, num_scenes - 1)
        child1 = np.concatenate([parent1[:crossover_point], parent2[crossover_point:]])
        child2 = np.concatenate([parent2[:crossover_point], parent1[crossover_point:]])
        return child1, child2

    def mutate(individual):
        mutation_chance = 0.1
        for i in range(num_scenes):
            if random.random() < mutation_chance:
                individual[i] = random.randint(1, max_days)

    def best_solution_from(population):
        best_individual = max(population, key=calculate_fitness)
        return best_individual

    def interpret_solution(solution):
        days_mapping = {day: [] for day in range(1, max_days + 1)}
        for scene, day in enumerate(solution, start=1):
            days_mapping[day].append(scene)

        for day, scenes in sorted(days_mapping.items()):
            if scenes:
                scenes_str = ", ".join(map(str, scenes))
                print(f"Día {day}: Tomas {scenes_str}.")

    def algorithm_genetico():
        population = initialize_population()
        for generation in range(num_generations):
            new_population = []
            for _ in range(len(population) // 2):
                parent1, parent2 = select_parents(population)
                child1, child2 = crossover(parent1, parent2)
                mutate(child1)
                mutate(child2)
                new_population.extend([child1, child2])
            population.extend(new_population)
```

```
        population = sorted(population, key=calculate_fitness, reverse=True)[:population_s
    return best_solution_from(population)
```

Análisis

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

La complejidad del problema pueden analizarse desde dos perspectivas principales: la complejidad computacional y el tamaño del espacio de soluciones.

La complejidad computacional de este problema, en términos del algoritmo genético utilizado para resolverlo, depende de varios factores, incluyendo el tamaño de la población, el número de generaciones, el número de tomas (escenas) y el número de actores. Sin embargo, calcular una complejidad exacta en términos de notación Big O para algoritmos genéticos es complicado debido a su naturaleza estocástica y a la variedad de operaciones involucradas (selección, cruce, mutación).

En general, si consideramos N como el número de tomas, M como el tamaño de la población y G como el número de generaciones, podemos decir que el algoritmo tiene una complejidad temporal aproximada de O(G*M*N) por cada ciclo principal del algoritmo, sin contar el tiempo adicional que puedan requerir operaciones específicas de cruce y mutación, ni la evaluación de la función de aptitud, que añade su propia complejidad dependiendo de cómo esté implementada.

El espacio de soluciones se refiere al conjunto total de posibles asignaciones de días a tomas. Si tenemos N tomas y cada toma puede ser asignada a cualquier día dentro de un rango de D días disponibles, entonces cada toma tiene D opciones para su asignación. Esto nos da un espacio de soluciones total de D^N.

Por ejemplo, si se restringe el problema a 5 días y hay 30 tomas, el espacio de soluciones es 5^30, lo que es un número extremadamente grande y muestra la naturaleza exponencial del problema O(D^N). Este enorme espacio de soluciones justifica el uso de algoritmos genéticos para buscar soluciones aproximadas, ya que una búsqueda exhaustiva sería computacionalmente inviable para valores de N y D relativamente pequeños.