Algoritmos de optimización - Trabajo Práctico

- Nombre y Apellidos: Paul Florencio Rojas Quispe
- Url: https://github.com/paul0610/03MIAR---Algoritmos-de-Optimizacion---2023/tree/main/TrabajoPractico
- Google Colab: https://colab.research.google.com/drive/1zC2OCw32ZyoOp05GlDe3cxgxZZa0hdUE?usp=sharing

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: https://docs.google.com/spreadsheets/d/1ODVSs3OPrTa36govaoYQbqRC9LhFp9PN/edit?usp=sharing&ouid=100668873241613497583&rtpof=true&sd=true

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

Análisis del Problema

Este problema se enfoca en mejorar la coordinación del proceso de doblaje en películas, con el objetivo de optimizar la agenda de los actores de doblaje para asegurar que las sesiones en las que participan simultáneamente estén bien sincronizadas, a la vez que se busca reducir los costes derivados de su contratación.

Estos costes se calculan en función de los días de trabajo en lugar del número de escenas dobladas. Para abordar esta complejidad, se utilizan algoritmos evolutivos y técnicas de optimización que permiten refinar las soluciones a través de mutaciones aleatorias y la selección de las opciones más prometedoras en cada ciclo.

Para resolver este problema, se emplean algoritmos y técnicas de optimización, especialmente algoritmos evolutivos, que buscan mejorar continuamente las soluciones mediante mutaciones aleatorias y la selección de las mejores soluciones en cada iteración. Se incorporan además estrategias para verificar y corregir las propuestas, asegurando que se ajusten a las limitaciones del proyecto. La efectividad de las soluciones se mide mediante una función específica que evalúa el coste total basado en los días laborales de los actores.

La evaluación de la calidad de las soluciones se lleva a cabo mediante una función objetivo que calcula el costo total de una solución, considerando el número de días en los que trabajan los actores. Además, se implementan métodos de verificación y reparación de soluciones para garantizar que todas las propuestas sean viables y cumplan con las restricciones del problema. El objetivo es encontrar un plan de trabajo para el doblaje que no solo sea viable sino que también optimice los recursos y reduzca los gastos.

El espacio de soluciones para este problema incluye todas las posibles combinaciones de asignación de tomas a días, de tal manera que se cumplan las restricciones de participación de actores y el límite de tomas por día. Cada solución en este espacio es una manera específica de distribuir las 30 tomas a través de un número de días, con hasta 6 tomas por día. Matemáticamente, podrías pensar en este espacio como el conjunto de todas las particiones posibles de las tomas en grupos de hasta 6, donde cada grupo representa las tomas asignadas a un día particular.

Modelo

Espacio de soluciones

- El espacio de soluciones para este problema incluye todas las posibles combinaciones de asignación de tomas a días, de tal manera que se cumplan las restricciones de participación de actores y el límite de tomas por día. Cada solución en este espacio es una manera específica de distribuir las 30 tomas a través de un número de días, con hasta 6 tomas por día. Matemáticamente, podrías pensar en este espacio como el conjunto de todas las particiones posibles de las tomas en grupos de hasta 6, donde cada grupo representa las tomas asignadas a un día particular.

¿Como implemento las restricciones?

1. Restricción de Participación de Actores: Cada toma requiere la presencia de ciertos actores. Cuando se elige una toma para un día específico, solo se pueden elegir tomas adicionales para ese día si los actores requeridos no entran en conflicto con los ya programados para otras tomas ese día. Esto se verifica sumando las presencias de actores para las tomas seleccionadas y asegurándose de que ningún actor esté contado más de una vez.

2. Límite de 6 Tomas por Día: No se pueden asignar más de 6 tomas a un solo día. Esto se controla directamente en el algoritmo al limitar el tamaño de las combinaciones de

tomas consideradas para cada día a un máximo de 6.

Funciones de la Solución

1. `objective_function(solution, schedule_matrix, debug=False)` : Calcula el costo total de una solución dada, que representa el número total de días en los que trabajan los actores.
2. `greedy_initial_solution(schedule_matrix)` : Genera una solución inicial voraz ordenando las tomas de mayor a menor cantidad de actores por día.
3. `swap(solution, day1, day2, take1, take2)` : Intercambia dos tomas entre dos días en una solución dada.
4. `local_search(initial_solution, schedule_matrix)` : Realiza una búsqueda local para mejorar una solución inicial intercambiando tomas entre días.
5. `validate_solution(solution, schedule_matrix)` : Verifica si una solución dada cumple con las restricciones del problema, como que cada toma y cada día sean únicos.
6. `remove_duplicate_takes(solution)` : Elimina tomas duplicadas dentro de la misma solución.
7. `add_missing_takes(solution, schedule_matrix)` : Agrega tomas faltantes a una solución, asegurándose de que todas las tomas estén presentes al menos una vez.
8. `get_possible_solution(solution, schedule_matrix, do_local_search=False)` : Obtiene una solución posible a partir de una solución dada, corrigiendo cualquier inconsistencia y, opcionalmente, aplicando una búsqueda local.
9. `evolutionary_algorithm(initial_solution, population_size, num_epochs, schedule_matrix)` : Implementa un algoritmo evolutivo para buscar soluciones óptimas, generando una población de soluciones y aplicando mutaciones para mejorarlas a lo largo de varias épocas.
10. `multi_start_evolutionary_algorithm(population_size, num_epochs, num_starts, schedule_matrix)` : Aplica el algoritmo evolutivo múltiples veces desde diferentes soluciones iniciales, buscando la mejor solución encontrada entre todas.

Diseño

- ¿Que técnica utilizo? ¿Por qué?

La técnica utilizada para abordar el problema de organizar sesiones de doblaje es un algoritmo voraz (greedy algorithm). Esta elección se basa en varias razones clave que hacen que esta técnica sea adecuada para el problema en cuestión:

1. Simplicidad y Eficiencia:

Los algoritmos voraces son conceptualmente simples y pueden ser muy eficientes en términos de tiempo de ejecución para ciertos tipos de problemas. Al tomar decisiones locales óptimas en cada paso, sin reconsiderarlas posteriormente, permiten construir rápidamente una solución al problema sin la necesidad de explorar exhaustivamente todo el espacio de soluciones.

1. Naturaleza Combinatoria del Problema:

Dado que el problema de organizar sesiones de doblaje tiene una naturaleza combinatoria (implicando la selección de tomas para cada día de grabación de manera que se cumplan ciertas restricciones), el uso de un enfoque voraz ofrece un método práctico para reducir el espacio de búsqueda. En lugar de evaluar todas las posibles combinaciones de tomas para cada día, el algoritmo voraz se enfoca en encontrar combinaciones que maximicen el número de tomas por día, dadas las restricciones, lo cual es una aproximación razonable para minimizar el número total de días de grabación.

1. Restricciones de Disponibilidad de Actores:

Las restricciones específicas del problema, como la disponibilidad de los actores para las tomas, hacen que el problema sea más complejo que una simple partición de tomas en días. El enfoque voraz permite integrar estas restricciones de manera directa al evaluar si una toma puede ser asignada a un día específico, facilitando el manejo de restricciones complejas sin necesidad de algoritmos de optimización más complicados.

1. Compromiso entre Optimalidad y Factibilidad:

Aunque los algoritmos voraces no garantizan encontrar la solución óptima global para todos los problemas, son especialmente útiles cuando se busca un compromiso entre la calidad de la solución y la factibilidad computacional. Para problemas con un espacio de soluciones vasto y complejo, como el presente, ofrecen una forma viable de obtener soluciones buenas en un tiempo razonable, lo cual es preferible en situaciones donde la ejecución de algoritmos más exhaustivos sería prohibitivamente costosa o lenta.

In [ ]:
```
#Implementacion


import numpy as np
from tabulate import tabulate

# Definir los datos iniciales
datos_iniciales = [
    [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]
]

# Definir los nombres de los actores
nombres_actores = ["Actor 1", "Actor 2", "Actor 3", "Actor 4", "Actor 5",
                   "Actor 6", "Actor 7", "Actor 8", "Actor 9", "Actor 10"]

# Definir los nombres de las tomas
nombres_tomas = ["Toma {}".format(i+1) for i in range(len(datos_iniciales))]

# Agregar los nombres de los actores como primer elemento de cada fila de datos
datos_con_nombres = [[nombre] + fila for nombre, fila in zip(nombres_tomas, datos_i

# Imprimir la tabla usando tabulate con formato Markdown
tabla_markdown = tabulate(datos_con_nombres, headers=["Toma"] + nombres_actores, ta
print(tabla_markdown)

# Convertir a un array de numpy para calcular las sumas
datos_np = np.array(datos_iniciales)

# Calcular la suma de las filas y las columnas
suma_filas = np.sum(datos_np, axis=1)
suma_columnas = np.sum(datos_np, axis=0)
```

In [ ]:

```
import numpy as np

# Definimos la función objetivo
def objective_function(solution, schedule_matrix, debug=False):
    total_working_days = 0
    for day, takes_day in solution.items():
        # Creamos un conjunto de los actores que trabajaron este día.
        actors_day = set()
        for take in takes_day:
            # Utilizamos la función np.where para filtrar los actores que trabajan
            # Al comparar schedule_matrix[take-1]==1, obtenemos una lista de booled
            # Pasamos esta lista de booleanos a np.where, que nos retorna los índic
            # Luego obtenemos el índice 0 ya que retorna una matriz, y sumamos 1 pc
            actors_take = set(np.where(schedule_matrix[take-1] == 1)[0] + 1)
            # Usamos la función update, que puede recibir un iterable y los agrega
            actors_day.update(actors_take)

        if debug:
            print("Tomas del", str(day) + ":", takes_day)
            print("Actores del", str(day) + ":", actors_day)

        total_working_days += len(actors_day)
    return total_working_days
```

```
# Datos de prueba
tabla_np = 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]
])

# Prueba de la función objetivo
initial_solution = {
    "dia 1" : [1, 2, 3, 4, 5, 6],
    "dia 2" : [7, 8, 9, 10, 11, 12],
    "dia 3" : [13, 14, 15, 16, 17, 18],
    "dia 4" : [19, 20, 21, 22, 23, 24],
    "dia 5" : [25, 26, 27, 28, 29, 30]
}
```

In [ ]:

```
def greedy_initial_solution(schedule_matrix):
    num_takes = len(schedule_matrix)

    # Calculamos la cantidad de actores para cada toma
    num_actors_per_take = [np.sum(take) for take in schedule_matrix]

    # Ordenamos las tomas de forma descendente según la cantidad de actores
    sorted_takes = sorted(range(num_takes), key=lambda i: num_actors_per_take[i], r

    print(sorted_takes)

    solution = {}
    day = 1
    takes_per_day = 0
    for take in sorted_takes:
        if takes_per_day == 6:  # Si ya se asignaron 6 tomas a este día, pasamos al
            day += 1
            takes_per_day = 0
        if solution.get("dia " + str(day)) != None:
            solution["dia " + str(day)].append(take + 1)
        else:
```

```
            solution["dia " + str(day)] = [take + 1]
        takes_per_day += 1

    return solution

greedy_solution = greedy_initial_solution(tabla_np)

# Imprimir cada día en una línea diferente
for day, takes in greedy_solution.items():
    print(day + ":", takes)
```

In [ ]:
```
import copy

# Definimos una función swap que intercambia dos posiciones
def swap(solution, day1, day2, take1, take2):
    # Hacemos una copia profunda de la solución
    new_solution = copy.deepcopy(solution)

    # Verificamos si los días especificados existen en el diccionario
    if "dia" + str(day1) in new_solution and "dia" + str(day2) in new_solution:
        # Verificamos si las posiciones especificadas existen en los días correspor
        if take1 < len(new_solution["dia" + str(day1)]) and take2 < len(new_solutic
            # Intercambiamos las tomas
            new_solution["dia" + str(day1)][take1], new_solution["dia" + str(day2)]
                new_solution["dia" + str(day2)][take2], new_solution["dia" + str(da
        else:
            print("Error: Las posiciones especificadas no existen en los días corre
    else:
        print("Error: Los días especificados no existen en la solución.")

    return new_solution

# Probamos la función swap
new_solution = swap(greedy_solution, 1, 2, 0, 0)
if new_solution:
    print(objective_function(new_solution, tabla_np, True))
```

In [ ]:
```
import random

def local_search(initial_solution, schedule_matrix):
    best_solution = initial_solution
    best_cost = objective_function(best_solution, schedule_matrix)

    num_days = len(initial_solution)
    if num_days < 2:
        print("Error: La solución inicial debe contener al menos dos días.")
        return None

    for i in range(50):
        day1, day2 = random.sample(range(1, num_days + 1), 2)  # Seleccionamos dos

        # Verificamos si las claves de los días existen en el diccionario initial_s
        if "day"+str(day1) in initial_solution and "day"+str(day2) in initial_solut
            take1 = random.randint(0, len(initial_solution["day"+str(day1)]) - 1)
            take2 = random.randint(0, len(initial_solution["day"+str(day2)]) - 1)
        else:
            print("Error: Las claves de los días no existen en la solución inicial.
            return None

        new_solution = swap(best_solution, day1, day2, take1, take2)
        cost = objective_function(new_solution, schedule_matrix)

        if cost < best_cost:
```

```
            best_solution = new_solution
            best_cost = cost

    return best_solution

initial_solution = greedy_initial_solution(tabla_np)
print(objective_function(initial_solution, tabla_np, True))

local_search_result = local_search(initial_solution, tabla_np)
if local_search_result:
    print(objective_function(local_search_result, tabla_np, True))
```

In [ ]:
```
def validate_solution(solution, schedule_matrix):
    unique_takes = set()

    for day, takes in solution.items():
        unique_takes_day = set(takes)

        if len(unique_takes_day) != len(takes):
            return False

        if len(unique_takes.intersection(unique_takes_day)) > 0:
            return False

        unique_takes.update(unique_takes_day)

    if len(unique_takes) != len(schedule_matrix):
        return False

    return True

# Verificamos la solución inicial voraz
greedy_initial_solution = greedy_initial_solution(tabla_np)
print(validate_solution(greedy_initial_solution, tabla_np))

# Verificamos una solución no válida
invalid_solution = {
    "dia1": [1, 1, 3, 4, 5, 6],
    "dia2": [7, 8, 9, 10, 11, 12],
    "dia3": [13, 14, 15, 16, 17, 18],
    "dia4": [19, 20, 21, 22, 23, 24],
    "dia5": [25, 26, 27, 28, 29, 30]
}

print(validate_solution(invalid_solution, tabla_np))
```

In [ ]:
```
def remove_duplicate_takes(solution):
    used_takes = set()
    for day, takes in solution.items():
        for i in range(len(takes)):
            if takes[i] in used_takes:
                solution[day][i] = 0
            else:
                used_takes.add(takes[i])
    return solution

# Ejemplo de uso
initial_solution_invalid = {
    "day1": [1, 1, 3, 4, 5, 6],
    "day2": [7, 8, 3, 3, 11, 12],
    "day3": [13, 1, 15, 16, 17, 18],
    "day4": [19, 20, 1, 22, 23, 24],
    "day5": [25, 5, 27, 28, 5, 30]
```

```
}

cleaned_solution = remove_duplicate_takes(copy.deepcopy(initial_solution_invalid))
display(cleaned_solution)
```

In [ ]:
```
def add_missing_takes(solution, schedule_matrix):
    used_takes = set()
    for day, takes in solution.items():
        for take in takes:
            if take != 0:  # Ignorar los ceros
                used_takes.add(take)

    missing_takes = list(set(range(1, len(schedule_matrix) + 1)) - used_takes)
    random.shuffle(missing_takes)

    for day, takes in solution.items():
        for i in range(len(takes)):
            if takes[i] == 0 and missing_takes:
                new_take = missing_takes.pop(0)
                while new_take == 0 or new_take in takes:
                    new_take = missing_takes.pop(0)
                solution[day][i] = new_take

    return solution

# Ejemplo de uso
invalid_solution = {
    "dia1": [1, 1, 3, 4, 5, 6],
    "dia2": [7, 8, 3, 3, 11, 12],
    "dia3": [13, 1, 15, 16, 17, 18],
    "dia4": [19, 20, 1, 22, 23, 24],
    "dia5": [25, 5, 27, 28, 5, 30]
}

pre_repaired_solution = remove_duplicate_takes(copy.deepcopy(invalid_solution))
repaired_solution = add_missing_takes(pre_repaired_solution, tabla_np)

display(repaired_solution)

print(validate_solution(repaired_solution, tabla_np))
```

In [ ]:
```
def get_possible_solution(solution, schedule_matrix, do_local_search=False):
    solution_copy = copy.deepcopy(solution)

    if not validate_solution(solution, schedule_matrix):
        solution_copy = remove_duplicate_takes(solution)
        solution_copy = add_missing_takes(solution_copy, schedule_matrix)

    if do_local_search:
        return local_search(solution_copy, schedule_matrix)
    else:
        return solution_copy

# Ejemplo de uso
invalid_solution = {
    "dia1": [1, 1, 3, 4, 5, 6],
    "dia2": [7, 8, 3, 3, 11, 12],
    "dia3": [13, 1, 15, 16, 17, 18],
    "dia4": [19, 20, 1, 22, 23, 24],
    "dia5": [25, 5, 27, 28, 5, 30]
}

display(get_possible_solution(invalid_solution, tabla_np))
```

In [ ]:
```python
import copy
import random

def evolutionary_algorithm(initial_solution, population_size, num_epochs, schedule_
    # Generar población inicial
    population = []
    for _ in range(population_size):
        solution = copy.deepcopy(initial_solution)
        for day, takes in solution.items():
            num_takes_to_mutate = random.randint(1, 10)
            for _ in range(num_takes_to_mutate):
                index_to_mutate = random.randint(0, len(takes) - 1)
                takes[index_to_mutate] = random.randint(1, len(schedule_matrix))
        solution = get_possible_solution(solution, schedule_matrix)
        population.append(solution)

    # Por cada época
    for _ in range(num_epochs):
        # Generar hijos (soluciones mutadas)
        children = []
        for solution in population:
            child = copy.deepcopy(solution)
            day_to_mutate = "day" + str(random.randint(1, len(child)))
            if day_to_mutate in child:  # Verificar si el día existe en la solución
                take_to_mutate = random.randint(0, len(child[day_to_mutate]) - 1)
                child[day_to_mutate][take_to_mutate] = random.randint(1, len(schedu
                child = get_possible_solution(child, schedule_matrix, True)  # Corr
                children.append(child)

        # Unir padres e hijos y escoger los mejores
        population += children
        population.sort(key=lambda solution: objective_function(solution, schedule_
        population = population[:population_size]  # Conservar solo los mejores

    return population[0]  # Devolver el mejor individuo
```

In [ ]:
```python
def multi_start_evolutionary_algorithm(population_size, num_epochs, num_starts, sch
    best_solution = None
    best_cost = float('inf')
    countsesion = 0
    for i in range(num_starts):
        evolutionary_solution = evolutionary_algorithm(initial_solution, populatior
        cost = objective_function(evolutionary_solution, schedule_matrix)
        if  countsesion <= 0:
          for day, takes in evolutionary_solution.items():
              actors = set()

              if int(day[-1]) <= 5:
                for take in takes:
                  actors.update(np.where(schedule_matrix[take - 1] == 1)[0] + 1)
                print(f"Tomas {day}: {takes}")

        if cost < best_cost:
            best_solution = evolutionary_solution
            best_cost = cost
        countsesion = countsesion + 1
    print(f"Mejor costo obtenido: {best_cost}")
    return best_solution

# Ejemplo de uso:
solucion_multiarranque = multi_start_evolutionary_algorithm(population_size=50, num
```

```
Tomas dia 1: [16, 9, 30, 7, 6, 28]
Tomas dia 2: [27, 23, 22, 24, 17, 29]
Tomas dia 3: [10, 11, 8, 5, 13, 12]
Tomas dia 4: [20, 18, 1, 25, 2, 19]
Tomas dia 5: [21, 26, 15, 14, 4, 3]
Mejor costo obtenido: 33
```

Conclusiones

Resultados:

- Se obtuvieron los siguientes resultados

  ```
  Tomas dia 1: [16, 9, 30, 7, 6, 28]
  Tomas dia 2: [27, 23, 22, 24, 17, 29]
  Tomas dia 3: [10, 11, 8, 5, 13, 12]
  Tomas dia 4: [20, 18, 1, 25, 2, 19]
  Tomas dia 5: [21, 26, 15, 14, 4, 3]
  Mejor costo obtenido: 33
  ```

- El algoritmo voraz ha logrado agrupar las tomas en 5 días, con un máximo de 6 tomas por día. El costo total de las sesiones de doblaje es de 33, lo que representa el mínimo posible considerando que cada actor cobra por día de trabajo. La distribución de las tomas en los días parece ser equilibrada, sin que haya un día con una carga de trabajo excesiva.

- Es importante recordar que el algoritmo voraz es una aproximación al problema de optimización y no siempre encuentra la solución óptima. La calidad de la solución depende de la heuristica utilizada para seleccionar la toma con mayor cantidad de actores participantes. En algunos casos, puede ser necesario explorar otras alternativas para encontrar una mejor solución.

- El resultado obtenido por el algoritmo voraz es una buena aproximación al problema de planificar las sesiones de doblaje. El costo total de las sesiones es el mínimo posible y la distribución de las tomas en los días parece ser equilibrada. Sin embargo, es importante tener en cuenta las limitaciones del algoritmo voraz y considerar otras alternativas para encontrar una mejor solución.