

▼ Proyecto Algoritmos de Optimización

Nombre: Alejandro Vergara Richart

Github:

Problema: Organizar sesiones de doblaje.

Descripción: 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.

- Número de actores: 10
- Número de tomas: 30
- Datos Actores/Tomas: <https://bit.ly/36D8luK>
 - 1 indica que el actor participa en la toma
 - 0 en caso contrario

+ Código

+ Texto

(*)¿Cuántas posibilidades hay sin tener en cuenta las restricciones?

¿Cuántas posibilidades hay teniendo en cuenta todas las restricciones.

▼ Respuesta

Sin considerar las restricciones, hay un total de $30!$ combinaciones posibles.

Sin embargo, si tenemos en cuenta las restricciones (limitando a 6 tomas por día como máximo), el número de combinaciones se puede determinar utilizando el número combinatorio $C(30, 6)$.

```
import math
combinaciones = math.factorial(30)
combinaciones_restricciones = math.comb(30, 6)

print(f'El número de combinaciones sin restricciones es {combinaciones}')
print(f'el número de combinaciones con restricciones es {combinaciones_restricciones}')
```

El número de combinaciones sin restricciones es 265252859812191058636308480000

Modelo para el espacio de soluciones

(*) ¿Cual es la estructura de datos que mejor se adapta al problema? Argumentalo. (Es posible que hayas elegido una al principio y veas la necesidad de cambiar, argumentalo)



Respuesta

Se ha elegido utilizar un DataFrame como estructura de datos, ya que facilita el almacenamiento de información en forma de tabla. En esta tabla, cada fila representará una escena que debe ser grabada, y cada columna corresponderá a un actor. En cada celda (i, j) de la tabla, se registrará un valor de 1 o 0 para indicar si el actor j debe o no asistir a la grabación de la escena i. Cada vez que un actor participe en un día de grabación, se incurrirá en un costo de 1 unidad.

```
from google.colab import drive
import pandas as pd
drive.mount('/content/drive')

df = pd.read_excel('/content/drive/MyDrive/Master_VIU/Algoritmos_
df = df.drop([30]) # Se elimina la ultima fila de totales
df = df.drop(['Total', 'Toma'], axis=1) # Se eliminan las columnas
df
```

Drive already mounted at /content/drive; to attempt to forcibly remount, call

	1	2	3	4	5	6	7	8	9	10	
0	1	1	1	1	1	0	0	0	0	0	
1	0	0	1	1	1	0	0	0	0	0	
2	0	1	0	0	1	0	1	0	0	0	
3	1	1	0	0	0	0	1	1	0	0	
4	0	1	0	1	0	0	0	1	0	0	
5	1	1	0	1	1	0	0	0	0	0	
6	1	1	0	1	1	0	0	0	0	0	
7	1	1	0	0	0	1	0	0	0	0	
8	1	1	0	1	0	0	0	0	0	0	
9	1	1	0	0	0	1	0	0	1	0	
10	1	1	1	0	1	0	0	1	0	0	
11	1	1	1	1	0	1	0	0	0	0	
12	1	0	0	1	1	0	0	0	0	0	

Según el modelo para el espacio de soluciones

(*)¿Cual es la función objetivo?

(*)¿Es un problema de maximización o minimización?

16 1 0 1 0 0 0 0 0 0 0

Respuesta

16 1 0 1 0 0 0 0 0 0 0

La función objetivo es la que se presenta a continuación. Esta función objetivo está diseñada para calcular el costo total de la asignación de actores a días de grabación. La función recorre cada día de grabación y calcula el costo asociado a ese día en función de la asignación de actores. El costo se calcula de la siguiente manera:

1. Se determina qué tomas están programadas para ese día según la lista asignacion.
2. Luego, se recorren las tomas de ese día y se identifican los actores que participan en esas tomas.
3. Se asegura de que cada actor se cuente solo una vez, utilizando un conjunto (set) para evitar duplicados.
4. Finalmente, suma el número de actores presentes en ese día al costo total.

El objetivo es minimizar dicha función. Al minimizar el costo total, se está tratando de encontrar una asignación de actores a tomas de manera eficiente para reducir la cantidad total de actores necesarios en la grabación.

```
def costo_total(asignacion, participacion_actores):
    dias = max(asignacion) + 1 # Determina el número de días de c
    costo = 0
    for dia in range(dias):
        # Encuentra las tomas asignadas a ese día
        tomas_dia = [i for i, d in enumerate(asignacion) if d ==

        # Encuentra los actores que participan en cada toma
        actores_presentes = set()
        for toma_idx in tomas_dia:
            actores_presentes.update(participacion_actores.iloc[t

        # Suma la cantidad de actores presentes en este día
        costo += len(actores_presentes)
    return costo
```

Diseña un algoritmo para resolver el problema por fuerza bruta

Respuesta

Nota: Se ha diseñado el siguiente algoritmo, en el que se recorren todas las combinaciones de posibles asignaciones. Sin embargo, no se va a ejecutar pues es muy costoso computacionalmente.

```
import itertools

# Número de actores y tomas
num_actores = len(df.columns)
num_tomas = len(df)

# Se generan todas las permutaciones de las tomas
permutaciones = list(itertools.permutations(range(num_tomas)))

mejor_asignacion = None
mejor_costo = float('inf')

# Probar todas las permutaciones para encontrar la mejor asignaci
for perm in permutaciones:
    costo_actual = costo_total(perm, df)
    if costo_actual < mejor_costo:
        mejor_costo = costo_actual
        mejor_asignacion = perm
```

```
print("Mejor asignación de tomas:", mejor_asignacion)
print("Costo mínimo:", mejor_costo)
```

Calcula la complejidad del algoritmo por fuerza bruta

Respuesta

La complejidad de generar todas las permutaciones de las tomas es de $n!$, mientras que la del cálculo del costo para cada asignación es de $d * k * n$, siendo d el número de días de grabación, k el número de actores y n el número de tomas. Por tanto, la complejidad del algoritmo es de $O(n! * m * k * n)$, o lo que es lo mismo, $O(n!)$. Esto hace que el algoritmo sea impracticable para un número alto de tomas.

(*)Diseña un algoritmo que mejore la complejidad del algoritmo por fuerza bruta. Argumenta porque crees que mejora el algoritmo por fuerza bruta

Respuesta

Se han diseñado dos algoritmos para mejorar la complejidad del algoritmo por fuerza bruta. A continuación se describen las ventajas que ofrecen estos:

1. Espacio de búsqueda grande: El problema implica encontrar una asignación óptima de actores a tomas, lo que genera un espacio de búsqueda grande, especialmente cuando hay un número significativo de tomas y actores. Ambos algoritmos pueden manejar espacios de búsqueda grandes y explorar diferentes soluciones de manera efectiva.
2. Heurísticas de mejora: Tanto la búsqueda tabú como el recocido simulado son algoritmos heurísticos que pueden encontrar soluciones cercanas a la óptima en problemas de optimización combinatoria. Pueden explorar soluciones de manera eficiente y utilizar estrategias para escapar de óptimos locales.
3. Manejo de restricciones: En el problema de asignación de actores, hay restricciones importantes, como la limitación de 6 tomas por día y la necesidad de que los actores coincidan en tomas específicas. Ambos algoritmos pueden incorporar restricciones en sus búsquedas y garantizar que se cumplan durante el proceso de optimización.
4. Exploración y explotación: Tanto la búsqueda tabú como el recocido simulado combinan la exploración y explotación en la búsqueda de soluciones óptimas. Pueden explorar el espacio de búsqueda de manera aleatoria o dirigida, lo que les permite escapar de soluciones subóptimas y encontrar soluciones de alta calidad.
5. Afinamiento de hiperparámetros: Ambos algoritmos tienen parámetros clave, como la longitud de la lista tabú en el caso de la búsqueda tabú o la temperatura inicial en el caso

del recocido simulado. Esto permite afinar el rendimiento de los algoritmos mediante experimentación y ajuste de parámetros.

6. Adaptación a condiciones cambiantes: Tanto la búsqueda tabú como el recocido simulado pueden adaptarse a cambios en las condiciones del problema, lo que los hace adecuados para situaciones donde las restricciones o los objetivos pueden cambiar con el tiempo.

▼ Búsqueda tabú

```
import random
import time

def evaluar_vecinos(vecindario, participacion_actores):
    mejor_vecino = None
    mejor_costo_vecino = float('inf')

    for vecino in vecindario:
        costo_vecino = costo_total(vecino, participacion_actores)
        if costo_vecino < mejor_costo_vecino:
            mejor_vecino = vecino
            mejor_costo_vecino = costo_vecino

    return mejor_vecino, mejor_costo_vecino

def costo_total(asignacion, participacion_actores):
    dias = max(asignacion) + 1
    costo = 0
    for dia in range(dias):
        tomas_dia = [i for i, d in enumerate(asignacion) if d == dia]
        actores_presentes = set()
        for toma_idx in tomas_dia:
            actores_presentes.update(participacion_actores.iloc[toma_idx])
        costo += len(actores_presentes)
    return costo

def construir_solucion(num_tomas, num_dias, num_max_tomas_por_dia):
    ''' Se construye una solución aleatoria '''
    asignacion = [-1] * num_tomas # Inicializar con valor no válido
    tomas_por_dia = [0] * num_dias
```

```
tomas_disponibles = list(range(num_tomas)) # Lista de tomas di

for dia in range(num_dias):

    random.shuffle(tomas_disponibles)
    tomas_asignadas = tomas_disponibles[:num_max_tomas_por_dia]
    for toma in tomas_asignadas:
        asignacion[toma] = dia
        tomas_por_dia[dia] += 1

    # Actualizar la lista de tomas disponibles
    tomas_disponibles = [t for t in tomas_disponibles if t not in asignacion]

    if not tomas_disponibles:
        break

return asignacion


def generar_vecindario(solucion_actual, num_dias, num_max_tomas_por_dia):
    vecindario = []

    # Vecindario 1: Intercambio de tomas
    for i in range(len(solucion_actual)):
        for j in range(i + 1, len(solucion_actual)):
            vecina = solucion_actual[:]
            vecina[i], vecina[j] = vecina[j], vecina[i]
            vecindario.append(vecina)

    # Vecindario 2: Inserción de un nodo en otra posición
    for i in range(len(solucion_actual)):
        for j in range(len(solucion_actual)):
            if i != j:
                vecina = solucion_actual[:]
                nodo = vecina.pop(i)
                vecina.insert(j, nodo)
                vecindario.append(vecina)

    # Vecindario 3: Cambio de día de una toma específica
    for toma in range(len(solucion_actual)):
        for nuevo_dia in range(num_dias):
            if solucion_actual[toma] != nuevo_dia:
                vecina = solucion_actual[:]
                vecina[toma] = nuevo_dia
                vecindario.append(vecina)
```

```

return vecindario

def busqueda_tabu(participacion_actores, num_dias, num_max_tomas_
    mejor_solucion = None
    mejor_costo = float('inf')
    memoria_tabu = set()

    solucion_actual = construir_solucion(len(participacion_actores)
    costo_actual = costo_total(solucion_actual, participacion_act

    for _ in range(max_iteraciones):
        vecindario = generar_vecindario(solucion_actual, num_dias

        mejor_vecino, costo_mejor_vecino = evaluar_vecinos(vecino

        if costo_mejor_vecino < costo_actual and tuple(mejor_veci
            solucion_actual = mejor_vecino
            costo_actual = costo_mejor_vecino

            if costo_actual < mejor_costo:
                mejor_solucion = solucion_actual[:]
                mejor_costo = costo_actual

            # Agregar el movimiento a la memoria tabú
            memoria_tabu.add(tuple(mejor_vecino))

            # Eliminar movimientos antiguos de la memoria tabú si
            if len(memoria_tabu) > max_memoria_tabu:
                memoria_tabu.remove(memoria_tabu.pop())

    print(f'Mejor solucion: {mejor_solucion} \n Mejor coste: {me
    return mejor_solucion, mejor_costo

# Definir parámetros
num_dias = 5
num_max_tomas_por_dia = 6
max_iteraciones = 100
max_memoria_tabu = 20 # Tamaño máximo de la memoria tabú

# Ejecutar la Búsqueda Tabú
t = time.time()
mejor_solucion, mejor_costo = busqueda_tabu(df, num_dias, num_max
t_elapsed = time.time() - t

```



```
print("Mejor solución encontrada:", mejor_solucion)
print("Mejor costo encontrado:", mejor_costo)
print("Tiempo empleado (s): ", round(t_elapsed, 2))
```

```
Mejor solucion: [2, 3, 1, 1, 0, 2, 0, 3, 3, 4, 0, 3, 0, 1, 1, 4, 4, 1, 0, 2, 1]
Mejor coste: 32
Mejor solución encontrada: [2, 3, 1, 1, 0, 2, 0, 3, 3, 4, 0, 3, 0, 1, 1, 4, 4, 4, 4, 4, 4]
Mejor costo encontrado: 32
Tiempo empleado (s): 944.02
```

▼ Recocido simulado

```
import time
import math

def genera_vecina(solucion, participacion_actores):
    mejor_solucion = []
    mejor_distancia = float('inf')
    for i in range(1, len(solucion)-1):
        for j in range(i+1, len(solucion)):
            vecina = solucion[:i] + [solucion[j]] + solucion[i+1:j] + [solucion[i]]

            distancia_vecina = costo_total(vecina, participacion_actores)

            if distancia_vecina <= mejor_distancia:
                mejor_distancia = distancia_vecina
                mejor_solucion = vecina

    return mejor_solucion

def construir_solucion(num_tomas, num_dias, num_max_tomas_por_dia):
    ''' Se construye una solución aleatoria '''
    asignacion = [-1] * num_tomas # Inicializar con valor no válido
    tomas_por_dia = [0] * num_dias

    tomas_disponibles = list(range(num_tomas)) # Lista de tomas disponibles

    for dia in range(num_dias):
        random.shuffle(tomas_disponibles)
        tomas_asignadas = tomas_disponibles[:num_max_tomas_por_dia]
        for toma in tomas_asignadas:
            asignacion[toma] = dia
            tomas_por_dia[dia] += 1
```

```

# Actualizar la lista de tomas disponibles
tomas_disponibles = [t for t in tomas_disponibles if t not in

if not tomas_disponibles:
    break

return asignacion

def recocido_simulado(participacion_actores, num_dias, num_max_tomas,
    solucion_actual = construir_solucion(len(participacion_actores), num_max_tomas)
    costo_actual = costo_total(solucion_actual, participacion_actores)

    mejor_solucion = solucion_actual[:]
    mejor_costo = costo_actual

    for iteracion in range(max_iteraciones):
        # Se actualiza la temperatura
        temperatura = temperatura_inicial / (1 + factor_enfriamiento * iteracion)

        # Se genera una solución vecina
        vecino = genera_vecina(solucion_actual, participacion_actores)
        costo_vecino = costo_total(vecino, participacion_actores)

        # Calcula la diferencia de costos entre la solución actual y la vecina
        delta_costo = costo_vecino - costo_actual

        # Si el vecino es mejor o se acepta con cierta probabilidad
        if delta_costo < 0 or random.random() < math.exp(-delta_costo / temperatura):
            solucion_actual = vecino
            costo_actual = costo_vecino

        # Actualiza la mejor solución si es necesario
        if costo_actual < mejor_costo:
            mejor_solucion = solucion_actual[:]
            mejor_costo = costo_actual

    return mejor_solucion, mejor_costo

t_inicial = 10 # Temperatura inicial
factor_enfriamiento = 0.95
num_dias = 5
num_max_tomas_por_dia = 6
max_iteraciones = 100

```

```
# Ejecución del algoritmo de recocido simulado
t = time.time()
mejor_solucion, mejor_costo = recocido_simulado(
    df, num_dias,
    num_max_tomas_por_dia,
    t_inicial,
    factor_enfriamiento,
    max_iteraciones
)
t_elapsed = time.time() - t

print("Mejor solución encontrada:", mejor_solucion)
print("Mejor costo encontrado:", mejor_costo)
print("Tiempo empleado (s): ", round(t_elapsed, 2))

    Mejor solución encontrada: [0, 3, 0, 0, 1, 2, 1, 4, 2, 2, 1, 4, 2, 4, 0, 0, 3,
    Mejor costo encontrado: 32
    Tiempo empleado (s):  296.2
```

(*)Calcula la complejidad del algoritmo

Respuesta

En el caso de la búsqueda tabú, la complejidad depende del número máximo de iteraciones (en este caso se han determinado 100) y la complejidad de calcular un vecindario (el más complejo es cercano a n^2). Por tanto la complejidad es $O(n^2)$.

Para el Recocido simulado, la complejidad depende del número de iteraciones (en este caso 100) y de la función generadora de una solución vecina, que tiene una complejidad de $O(n^2)$. Por tanto, la complejidad del algoritmo es de $O(n^2)$

Así pues, vemos como ambos algoritmos reducen la complejidad del algoritmo de fuerza bruta.

Según el problema (y tenga sentido), diseña un juego de datos de entrada aleatorios

Respuesta

```
import numpy as np

# Tomando un número de actores de 0 a 10 y un número de escenas c
num_actores = random.randint(1, 10)
num_escenas = random.randint(1, 30)
```

```
data = [[random.randint(0, 1) for _ in range(num_actores)] for _
df = pd.DataFrame(data)
df
```

	0	1	2	3	4	5	6
0	1	0	0	1	0	1	1
1	0	0	1	0	0	1	0
2	1	1	1	0	1	0	0
3	1	0	0	1	0	0	0
4	1	0	1	1	0	0	0
5	1	1	1	1	1	1	0
6	0	1	0	1	1	1	1
7	0	1	0	1	0	1	1
8	1	1	1	0	1	0	0
9	1	0	1	1	0	0	1
10	1	0	1	1	0	0	0
11	1	0	1	0	0	0	0
12	0	0	1	0	1	1	0
13	1	1	0	1	0	0	1
14	1	0	0	0	1	1	1
15	0	0	0	1	1	0	1
16	1	1	1	0	1	0	0

Aplica el algoritmo al juego de datos generado

Respuesta

```
# Búsqueda tabú
```

```
num_dias = 3
num_max_tomas_por_dia = 6
max_iteraciones = 100
max_memoria_tabu = 20 # Tamaño máximo de la memoria tabú
```

```
# Ejecutar la Búsqueda Tabú
```

```
t = time.time()
mejor_solucion, mejor_costo = busqueda_tabu(df, num_dias, num_max_tomas_por_dia)
```

```

t_elapsed = time.time() - t
print("Mejor solución encontrada:", mejor_solucion)
print("Mejor costo encontrado:", mejor_costo)
print("Tiempo empleado (s): ", round(t_elapsed, 2))

Mejor solucion: [0, 0, 1, 2, 2, 1, 1, 1, 2, 0, 0, 2, 1, 1, 0, 0, 2]
Mejor coste: 18
Mejor solución encontrada: [0, 0, 1, 2, 2, 1, 1, 1, 2, 0, 0, 2, 1, 1, 0, 0, 2]
Mejor costo encontrado: 18
Tiempo empleado (s): 158.69

```

Recocido simulado

```

t_inicial = 10 # Temperatura inicial
factor_enfriamiento = 0.95
num_dias = 3
num_max_tomas_por_dia = 6
max_iteraciones = 100

```

Ejecución del algoritmo de recocido simulado

```

t = time.time()
mejor_solucion, mejor_costo = recocido_simulado(
    df, num_dias,
    num_max_tomas_por_dia,
    t_inicial,
    factor_enfriamiento,
    max_iteraciones
)
t_elapsed = time.time() - t

```

```

print("Mejor solución encontrada:", mejor_solucion)
print("Mejor costo encontrado:", mejor_costo)
print("Tiempo empleado (s): ", round(t_elapsed, 2))

```

```

Mejor solución encontrada: [2, 0, 0, 1, 1, 0, 2, 2, 0, 1, 1, 1, 0, 1, 2, 2, 0]
Mejor costo encontrado: 17
Tiempo empleado (s): 48.71

```

Enumera las referencias que has utilizado(si ha sido necesario) para llevar a cabo el trabajo

Respuesta

Describe brevemente las lineas de como crees que es posible avanzar en el estudio del problema. Ten en cuenta incluso posibles variaciones del problema y/o variaciones al alza del tamaño

Respuesta

Se proponen 2 líneas a considerar para avanzar en el estudio del problema:

- Emplear otros algoritmos de optimización avanzada, como la programación lineal entera (PLE), permitiendo encontrar soluciones óptimas o cercanas a la óptima. Asimismo, permite modelar las restricciones de forma precisa, lo que puede ser interesante para variaciones del problema. No obstante, PLE puede ser útil para tamaños de problema pequeño a mediano, donde el espacio de búsqueda no es excesivamente grande y puede proporcionar soluciones óptimas en tiempos razonables.
- Considerar el uso de multiprocessing (procesamiento paralelo) puede ser una estrategia efectiva para mejorar el rendimiento y la escalabilidad al resolver el problema de asignación de actores, especialmente si se cuenta con una computadora con múltiples núcleos de CPU o acceso a un clúster de computadoras. Esto permite aprovechar los recursos hardware, dividir las tareas, paralelizar la evaluación de soluciones y tener mayor escalabilidad.