Algoritmos de optimización - Seminario

Nombre y Apellidos: Alberto Rodriguez Arizaga
Url: https://github.com/brtIn05/03MIAR---Algoritmos-de-Optimizacion/tree/main/Seminario
Url Collab: https://colab.research.google.com/drive/1xTsS-v6Pt410uZQ4Zp12ir0tCN3r0dhD?usp=sharing
Problema:

> 1. Organizar los horarios de partidos de La Liga

Descripción del problema:

Desde la La Liga de fútbol profesional se pretende organizar los horarios de los partidos de liga de cada jornada. Se conocen algunos datos que nos deben llevar a diseñar un algoritmo que realice la asignación de los partidos a los horarios de forma que maximice la audiencia.

Se dispone de 10 horarios diferentes y 20 equipos en 3 Categorias A,B y C (que tiene relación directa con la audiencia). Se proporciona la matriz de datos de audiencia en función de los enfrenmtamientos por categoria de equipos.

Si el horario no se realiza a las 20 horas se aplica una penalización.

Por último, se aplica otra penalización de audiencia en base al número de partidos programados en la misma franja horaria.

(*) La respuesta es obligatoria

1.- (*)¿Cuantas posibilidades hay sin tener en cuenta las restricciones?

En el problema se plantean la diustribución de 20 equipos (10 partidos) en 10 horarios diferentes. Por tanto, las posibilidades equivaldrían a todas las posibilidades de combinación de partidos en esos horarios. La repetición se admite, puesto que puede haber un partido en el horario 1 y otro en el 2 y la sequencia es relevante puesto que no es lo mismo ponerlo un partido en el primer horario o en el último. Por tanto las posibilidades sin restricciónes equivaldrían a las Variaciones con repetición de m elementos tomados en tuplas de n. Lo cual equivale a 10 elevado a 10 posibilidades.

$$VR_{10}^{10} = m^n = 10^{10}$$

¿Cuantas posibilidades hay teniendo en cuenta todas las restricciones?

En el caso de que tomemos en cuanta la restricción de que obligatoriamente un partido se juega el viernes y otro el sabado equivaldria a asignar directamente dos partidos a dos slots determinados. Esto nos reduciría en ambos casos el espacio de soluciones a tomar 1 partido menos dos veces. Esta reducción se declararía como:

$$RED = 2 * 9^{10}$$

Esta reducción, sin embargo, incorpora también los horarios de sabado y viernes Por tanto, si restamos estas combinaciones del problema global estaríamos eliminado demasiadas soluciones. Ya que en el subconjunto no fijado se incluyen tambien los horarios de viernes y lunes. Habría que aplicar algún tipo de incremento a este valor. Pero pordría darnos una idea del orden de magnitud.

Modelo para el espacio de soluciones
2.- (*) ¿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, arguentalo)

El modelo de datos parte de una lista que contiene los equipo ordenados por categoría. Cada índice de esta lista representa un equipo y su valor su correspondiente categoría (A,B,C se representa por 0,1,2).

In [ ]:
```
category = [0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2]
```

Esta estructura permite facilmente indexar su valor de audiencia base. Para ellos se genera una matriz simétrica con los datos de audiencia por categoría.

De este modo, por ejemplo, el enfrentamiento entre el equipo 13 y 14 (índices) corresponderá a un equipo de categoria 1 y 2 (valores, correspondiente a A y B) y se indexa en fila 1 columna 2 de la matriz de audiencia.

In [ ]:
```
audience = [
            [2, 1.3, 1],
            [1.3, 0.9, 0.75],
            [1, 0.75, 0.47],
]
```

Para modelar las penalizaciones en el caso de los coeficientes por horario se utiliza una lista donde cada índice corresponde al horario:

In [ ]:
```
penalty_slot = [0.4,0.55,0.7,0.8,1,0.45,0.75,0.85,1,0.4]
```

Para modelar las penalizaciones por coincidencia, se generó una lista y se invirtieron los porcentajes de modo que se apliquen como un coeficiente. El índice representa el número de partidos coincidentes y el valor en la lista el coeficiente aplicado.

In [ ]:
```
penalty_coincidence = [1, 1, 0.75,0.55,0.4,0.30,0.25,0.23,0.2,0.2]
```

Finalmente podemos generar una lista de tuplas que analice todos los posibles enfrentamientos de cada equipo sabiendo que:

- Un equipo no juega contra el mismo
- Por ejemplo, El equipo 0 contra el equipo 1 es el mismo enfrentamiento que el equipo 1 contra el 0

y añada como tercer elemento la audiencia base del partido:

In [ ]:
```
# match_audience = [(team1, team2, audience[category[team1]][category[team2]])  for
```

Durante el proceso de selección de algoritmo se valoro el uso de un modelo basado en mixed integer linear programing en ese caso se añadieron dos tipos de variables:

- Una variable de decisión binaria que decide si el partido se programa en un horario o no
- Una varible entera que cuenta el número de partidos en cada horario

El camino de usar un algoritmo basado en MILP no se utilizó finalmente debido a la complejidad de modelar la penalización por coincidencia. Añado un ejemplo de uso de la libreria pulp de python para modelar variables de decisión en un modelo MILP y el algoritmo diseñado durante el proceso en mi github https://github.com/brtln05/03MIAR---Algoritmos-de-Optimizacion/blob/main/Seminario/MILP_max_audiciencia_ARA.ipynb.

In [ ]:
```
# var_partido_horario = pulp.LpVariable.dicts("var_partido_horario", horario_partid
# var_numero_partidos_por_horario = pulp.LpVariable.dicts("var_numero_partidos_por_
```

3.- Según el modelo para el espacio de soluciones
(*)¿Cual es la función objetivo?

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

La función objetivo sumará todos los valores de audiencia calculados para cada partido en cada horario asignaco. Para cada partido, tomará el valor de audiencia base de la tupla que contiene el partido seleccionado y le aplicará los coeficientes de reducción por simultaneidad y por horario.

In [ ]:
```
# audience += match[-1]*penalty_slot[slot]*penalty_coincidence[len(matches)]
```

Es un problema de maximización que busca encopntrar la mejor combinación de partidos y horarios para obtener un mayor dato de audiencia acumulada.

4.- Diseña un algoritmo para resolver el problema por fuerza bruta

Respuesta

In [ ]:

5.- Calcula la complejidad del algoritmo por fuerza bruta

Un algoritmo por fuerza bruta debería analizar y recorrer todas las posibles combinaciones de partidos analizando en cada una de ellas el valor de la audiencia calculada. Esto se traduciría en una un bucle que recorrería del orden de n! combinaciones siendo n el numero de partidos posibles. Por tanto su complejidad sería $O(n!)$.

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

El algoritmo planteado para la resolución de este problema se basa en la utilización de una técnica voraz. En cada etapa intentaremos elegir el partido y horario que más audiencia proporcione teniendo en cuenta las condiciones y restricciones. Una de las claves para lograr esto se basa en que ordenamos previamente los partidos por su audiencia base por lo que en cada iterción estamos eligiendo el enfrentamiento base con mas audiencia. Sumando la audiencia obtenida en todas las etapas de asignación encontraremos el valor de la audiencia total. Este algoritmo no nos garantiza un máximo global, sin embargo nos puede proporcionar una buena aproximación al mismo.
Por el contrario, el algoritmo por fuerza bruta deberá analizar todas las posibles combinaciones de partidos en horarios y proporcionarnos la mejor combinación encontrada sin embargo las posibles combinaciones de este problema son muy elevadas.

La implementación del algoritmo voraz que resuelve el problema se presenta en la fúncion planteada a continuación y denominada max_audienc_per_slot. Los comentarios se han añadido en el código para facilitar su comprensión.

In [ ]:

```
def max_audienc_per_slot(teams, slots, category ,audience,penalty_slot,penalty_coin
    """
    Función que asigna slots a cada partido utilizando un algoritmo voraz
    """
    # Inicializamos la lista de partidos usados
    teams_already_used = []

    # Inicializamos el diccionario donde colocaremos cada uno de los partidos calcu
    # Las llaves serán los horarios y los valores los equipos
    slots_usage = {slot:[] for slot in range(slots)}

    # Creamos la lista de posibles combinaciones de partidos
    # Utilizamos una tupla que contendrá el equipo 1 el equipo 2 y su audiencia en
    # Por ejemplo los equipos 0, 1, 2 serán los 3 equipos de Categoria A (primeros
    # que se indexa luego para obtener la audiencia en la matriz de audiencias
    match_audience = [(team1, team2, audience[category[team1]][category[team2]])  f

    # Ordenamos los partidos por su audiencia, de modo que en cada iteración el par
    match_audience_sorted = sorted(match_audience,key=lambda x: x[2], reverse=True)

    # Recorremos la lista de partidos ordenada de mayor a menor audiencia
    for index, match in enumerate(match_audience_sorted):

        # Seleccionamos la parte de la tupla que contiene el partido
        match_teams = list(match[:2])
        # Inicializamos las variables de horario y mejor horario y valor de mejor h
        slot = 0
        best_slot_value = 0
        best_slot = 0

        # Verificamos si los equipos ya han sido utilizados para eviatar que un equ

        if match_teams[0] not in teams_already_used and match_teams[1] not in teams

            # Verficamos que no sean los últimos dos partidos, en cuyo caso
            # asumimos que se programarán viernes y sabado

            if len(teams_already_used) < teams-4:

                # Los incluimos en la lista de partidos usados
                teams_already_used.extend(match_teams)

                # Recorremos todos los slots
                # Este bucle es la clave del algoritmo y analizará cual es el mejor
                # de la penalización del mismo y de la coincidencia de horarios que
                # Almacenará dichos valores en las variables de mejor solución. Con
                # de que elegimos la mejor solución en cada etapa.
                while slot<slots-1:

                    slot_value = penalty_slot[slot]*penalty_coincidence[len(slots_u

                    if  slot_value > best_slot_value:

                        best_slot_value = slot_value
                        best_slot = slot

                    slot += 1

                # Añadimos el partido al mejor slot encontrado
```

```
                # Añadimos el partido al mejor slot encontrado

                slots_usage[best_slot].append(match)

                # Printeamos la solución para ese partido

                print(match[:2],f" in slot {best_slot}")

            else:
                # En caso de que sean los dos últimos partidos se envían directamen
                # al horario del lunes o el viernes
                if len(slots_usage[0]) == 0:
                    teams_already_used.extend(match_teams)
                    slots_usage[0].append(match)
                    #audience_value += penalty_slot[0] * match[-1]
                    print(match[:2],f" in slot {0}")

                else:
                    teams_already_used.extend(match_teams)
                    slots_usage[9].append(match)
                    #audience_value += penalty_slot[9] * match[-1]
                    print(match[:2],f" in slot {9}")
                    break

    # Creamos una variable informativa que cuente cuantos partidos hemos puesto en
    slots_usage_qty = {slot:len(slots_usage[slot]) for slot in range  (slots)}

    # Devolvemos la solúción encontrada
    print("\n #### Resultado de asignación de horarios: \n")
    print(slots_usage)
    print(slots_usage_qty)


    return slots_usage, slots_usage_qty
```

Añadimos una función que calcule la audiencia de una solución.

In [ ]:

```
def calculate_audience(slots_usage, penalty_slot, penalty_coincidence):
    """
    Función que calcula audiencia de una combinación dada de partidos y slots
    """
    audience = 0
    # Recorremos el diccionario de partidos asignados a horarios
    # Contamos su audiencia base, reducimos por tipo de horario y por coincidencia
    for slot, matches in slots_usage.items():
        for match in matches:
            audience += match[-1]*penalty_slot[slot]*penalty_coincidence[len(matche

    audience = round(audience,3)
    print("\n\n #### Audiencia obtenida: ", audience)

    return audience
```

Inicializamos el problema y lanzamos las funciones.

In [ ]:
```
teams = 20
slots = 10
Cat_A = 3
Cat_B = 11
Cat_C = 6

category = []

for i in range (teams):

    if i < Cat_A: # Categoría A
        category.append(0)
    elif i < Cat_B + Cat_A: # Categoría B
        category.append(1)
    else: # Categoría C
        category.append(2)

print(category)
```

```
[0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2]
```

In [ ]:
```
# Lanzamos el algoritmo con los datos del problema
slots_usage, slots_usage_qty = max_audienc_per_slot(teams, slots, category, audienc
audience_result = calculate_audience(slots_usage, penalty_slot, penalty_coincidence
```

```
(0, 1)  in slot 4
(2, 3)  in slot 8
(4, 5)  in slot 7
(6, 7)  in slot 3
(8, 9)  in slot 4
(10, 11)  in slot 6
(12, 13)  in slot 8
(14, 15)  in slot 2
(16, 17)  in slot 0
(18, 19)  in slot 9

 #### Resultado de asignación de horarios:

{0: [(16, 17, 0.47)], 1: [], 2: [(14, 15, 0.47)], 3: [(6, 7, 0.9)], 4: [(0, 1, 2),
(8, 9, 0.9)], 5: [], 6: [(10, 11, 0.9)], 7: [(4, 5, 0.9)], 8: [(2, 3, 1.3), (12, 1
3, 0.9)], 9: [(18, 19, 0.47)]}
{0: 1, 1: 0, 2: 1, 3: 1, 4: 2, 5: 0, 6: 1, 7: 1, 8: 2, 9: 1}


 #### Audiencia obtenida:  6.69
```

El algoritmo voraz es capaz de obtener un resultado de asignación de partidos a horarios obteniendo un resultado de audiencia de 6.69 millones de espectadores.

7.- (*)Calcula la complejidad del algoritmo

El algoritmo emplea dos bucles anidados. Un bucle for que recorre los posibles partidos y un bucle while que analiza el valor de todos los slots posibles para cada partido. En el caso mejor ambos bucles recorrerian exclusivamente el número de horarios disponible sin embargo, existe la posibilidad de que el primer bucle recorra alguna vez mas tratando de encontrar enfrentamientos entre equipos no utilizados. Adicionalmente, se emplea una función de ordención de la lista principal de partidos por tanto la complejidad se puede aproximar a:

$$O(n^2)$$

Cabe mencionar que en algunos escenarios donde el número de partidos fuese muy grande pero el número de horarios se mantuviese la ordenación de la lista podría tomar mas protagonismo y cambiar la complejidad.

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

Respuesta

In [ ]:

9.- Aplica el algoritmo al juego de datos generado

Respuesta

In [ ]:

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

Como se ha comentado previamente, antes de utilizar el algoritmo voraz se planteo el uso de MILP. Para ello se realizó una preimplementación en python en la que se consiguio resolver todas las restricciónes excepto la parte de tratamiento de la coincidencia de partidos. Imponiendo una restricción fija a un maximo de 2 partidos. Para ellos se analizaron diversas referencias sobre implementación de algoritmos de programación lineal y programación por restricciones:

Mi código en python:

https://github.com/brtIn05/03MIAR---Algoritmos-de-Optimizacion/blob/main/Seminario/MILP_max_audiciencia_ARA.ipynb

Referencias Pulp:

https://coin-or.github.io/pulp/

Tambien se analizó este problema en comparación son el problema clásico *sports league schedule* que se resuelve a traves de programación con restricciones utilizando algorítmos de busqueda y heurísticas para ello:

https://ibmdecisionoptimization.github.io/docplex-doc/mp/sports_scheduling.html

https://www.ibm.com/docs/en/icos/22.1.1?topic=optimizer-using-specialized-constraints-tuples-scheduling-teams

11.- 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

Como se ha mencionado este problema está estrechamente ligado al problema tipo de *sports league schedule*. En nuestro caso un algoritmo voraz esta ligado cuadraticamente a las posibles combinaciones de horarios por lo que en problemas grandes podríamos tener problemas. Adicionalmente el valor maximo que proporciona podría no ser adecuado dependiendo de las condiciones en las que se desarrolle el problema.
La utilización de algoritmos de constraint programming que se valgan de algoritmos de busqueda y técnicas de inferencia podría ayudar a reducir la copmplejidad del problema y encontrar soluciones más optimas. En una de las referencias que aporto se puede ver el desarrollo de un problema similar implementado a través de IBM CPLEX o CPO (Constraint programming optimizer).