

Partidos de la liga

Francesco Tesolato

Url: https://github.com/Francesco-hub/Algoritmos_Trabajo

11/07/2025

1. Definición 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
- En primer lugar se clasifican los equipos en tres categorías según el numero de seguidores (que tiene relación directa con la audiencia). Hay 3 equipos en la categoría A, 11 equipos de categoría B y 6 equipos de categoría C.
- Se conoce estadísticamente la audiencia que genera cada partido según los equipos que se enfrentan y en horario de sábado a las 20h (el mejor en todos los casos) Si el horario del partido no se realiza a las 20 horas del sábado se sabe que se reduce.
- Debemos asignar obligatoriamente siempre un partido el viernes y un partido el lunes
- Es posible la coincidencia de horarios pero en este caso la audiencia de cada partido se verá afectada y se estima que se reduce en porcentaje.
- Los datos extraídos de esta definición se representan en las siguientes tablas:

TABLA 1

Equipo	Categoría
Barcelona	A
R.Madrid	A
Real Sociedad	A
Alaves	B
Athletic	B
Atletico Madrid	B
Betis	B
Celta	B
Espanyol	B
Getafe	B
Levante	B
Sevilla	B
Valencia	B
Villarreal	B
Eibar	C
Granada	C
Leganes	C
Mallorca	C
Osasuna	C
Valladolid	C

TABLA 2

DIA	HORA
Viernes	20
Sábado	12, 16, 18, 20
Domingo	12, 16, 18, 20
Lunes	20

TABLA 3

Categoría	A	B	C
A	2M	1.3M	1M
B	1.3M	0.9M	0.75M
C	1M	0.75M	0.47M

TABLA 4

	Viernes	Sábado	Domingo	Lunes
12h	-	0.55	0.45	-
16h	-	0.7	0.75	-
18h	-	0.8	0.85	-
20h	0.4	1	1	0.4

TABLA 5

Coincidencias	-%
0	0%
1	25%
2	45%
3	60%
4	70%
5	75%
6	78%
7	80%
8	80%

1. **TABLA 1:** Equipos y su categoría
2. **TABLA 2:** Horarios disponible
3. **TABLA 3:** Audiencia base por enfrentamientos en categorías
4. **TABLA 4:** Multiplicador de audiencia por franja horaria
5. **TABLA 5:** Penalización por coincidencia

2. Preguntas a resolver

1. ¿Cuantas posibilidades hay sin tener en cuenta las restricciones?
2. ¿Cuantas posibilidades hay teniendo en cuenta todas las restricciones.
3. ¿Cual es la estructura de datos que mejor se adapta al problema? Argumenta la respuesta (Es posible que hayas elegido una al principio y veas la necesidad de cambiar, argumenta)

4. ¿Cuál es la función objetivo?
 5. ¿Es un problema de maximización o minimización?
 6. Diseña un algoritmo para resolver el problema por fuerza bruta
 7. Calcula la complejidad del algoritmo por fuerza bruta
 8. Diseña un algoritmo que mejore la complejidad del algoritmo por fuerza bruta.
Argumenta porque crees que mejora el algoritmo por fuerza bruta
 9. Calcula la complejidad del algoritmo
 10. Según el problema (y tenga sentido), diseña un juego de datos de entrada aleatorio.
 11. Aplica el algoritmo al juego de datos aleatorio generado.
 12. Enumera las referencias que has utilizado(si ha sido necesario) para llevar a cabo el trabajo
 13. Describe brevemente en unas líneas 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
- A continuación, cada pregunta será contestada en orden, en el siguiente formato:
Pregunta -> Argumentación teórica -> Código en python (si procede)

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

Respuesta

Para calcular el número de posibilidades, es importante lo primero tener en cuenta qué exactamente compone un resultado. Según el ejemplo proporcionado, tenemos la siguiente jornada:

Partido	Categorías	Horario	Base(Mill.)	Ponderación	Base*Ponderación	Corrección Coincidencia
Celta - Real Madrid	B-A	V20	1,3	0,4	0,52	0,52
Valencia - R. Sociedad	B-A	S12	1,3	0,55	0,72	0,72
Mallorca - Eibar	C-C	S16	0,47	0,7	0,33	0,33
Athletic - Barcelona	B-A	S18	1,3	0,8	1,04	1,04
Leganés - Osasuna	C-C	S20	0,47	1	0,47	0,47
Villarreal - Granada	B-C	D16	0,75	0,75	0,56	0,42
Alavés - Levante	B-B	D16	0,9	0,75	0,68	0,51
Espanyol - Sevilla	B-B	D18	0,9	0,85	0,77	0,77
Betis - Valladolid	B-C	D20	0,75	1	0,75	0,75
Atlético - Getafe	B-B	L20	0,9	0,4	0,36	0,36
						Total: 5,88
						$=0,56 \times 0,75$
						$=0,68 \times 0,75$

De la siguiente tabla podemos extraer la siguientes conclusiones:

- El orden de los equipos en un partido no influye en la audiencia. ej Audiencia(Mallorca - Eibar) = Audiencia(Eibar - Mallorca).
- Cada equipo se enfrenta exactamente a 1 rival.

Dicho esto, podemos empezar a expresar matemáticamente nuestro problema:

- 20 Equipos distintos. Cada equipo se puede enfrentar a cualquiera de los demás: $20!$
- Dentro de cada partido el orden no importa, así que podemos eliminar algunos de los resultados: 2^{10} (2 representa el número de formas de ordenar los equipos en cada partido y 10 representa el número de partidos).
- Dentro de una jornada, también nos da igual el orden de los partidos, ya que eso viene definido posteriormente por la franja horaria, es decir, no vamos a contar como resultados distintos decir, por ejemplo:
 - **[Mallorca - Eibar | Valencia - Getafe] Y [Valencia - Getafe | Mallorca - Eibar]**
 Para ello, eliminaremos los resultados que pertenecerían a las permutas de estos partidos: $10!$
- Expresado en fórmula sería:

Número de partidos posibles

Permutas en el orden de partidos × Permutas en el orden de los equipos en el partido

- Y con valores numéricos:

$$\frac{20!}{10! \times 2^{10}}$$

- Sin embargo, el problema no acaba ahí, ya que éste sería sólo el número de combinaciones posibles sin tener en cuenta las franjas horarias.
- Al existir 10 franjas horarias distintas, si no tenemos en cuenta ninguna restricción, cada uno de los 10 partidos puede jugarse en cada una de las 10 franjas horarias: 10^{10} .
- Si añadimos eso a nuestra fórmula y calculamos el resultado, respondemos a la pregunta:

$$\frac{20!}{10! \times 2^{10}} \times 10^{10} = 654729075 \times 10^{10} \approx 6.547 \times 10^{18}$$

Existen 6.547×10^{18} combinaciones posibles sin restricciones

2. ¿Cuantas posibilidades hay teniendo en cuenta todas las restricciones?

Respuesta

Ahora que hemos calculado las posibilidades totales sin restricciones, cabe destacar que el problema sí plantea restricciones. La única restricción que hay se nos presenta de manera explícita:

- Debe de haber como mínimo un partido el **Viernes** y otro el **Lunes**.

Ahora, veremos cómo esta restricción afecta a nuestro cálculo inicial y calcularemos el nuevo total de posibilidades usando el principio de exclusión.

Lo primero es entender que en nuestra fórmula, tenemos las distintas combinaciones de partidos multiplicada por las distintas combinaciones de horarios. Dado que la restricción se da solo en relación a los horarios, solo restaremos eso al 10^{10} que hemos calculado anteriormente.

De nuestra fórmula inicial, eliminamos los siguientes casos:

- Casos en los que no hay ningún partido el Viernes.
- Casos en los que no hay ningún partido el Lunes.
- Casos en los que no hay ningún partido ni Viernes ni Lunes.

Para ello, eliminamos:

- Las que tengan 9 franjas sin el Viernes: 9^{10} .
- Las que tengan 9 franjas sin el Lunes: 9^{10} .
- Las que tengan 8 franjas, sin Viernes ni Lunes: 8^{10} .

En nuestra fórmula, queda así:

$$\left(\frac{20!}{10! \times 2^{10}} \right) \times \left(10^{10} - (2 \times 9^{10} - 8^{10}) \right)$$

que, calculado, da un nuevo total de:

$$\left(\frac{20!}{10! \times 2^{10}} \right) \times \left(10^{10} - (2 \times 9^{10} - 8^{10}) \right) \approx 2.694 \times 10^{18}$$

Existen 2.694×10^{18} combinaciones posibles con restricciones

Como ligera observación, con los datos que tenemos, solo tenemos una penalización por coincidencia hasta 8 partidos, lo cual se podría interpretar como que ese es el máximo de coincidencias posibles y, por tanto, habría que eliminar todos los casos con 9 y 10 coincidencias, como una restricción adicional.

Sin embargo, eliminar estas combinaciones significa eliminar un total de 910 combinaciones, lo cual supone un número ínfimo teniendo en cuenta la magnitud de las distintas combinaciones que hemos mencionado.

Esto, sumado a que no es una restricción ofrecida por el ejercicio sino una interpretación personal, hace que la vaya a ignorar en este caso para el cálculo. Aún así, cuando exponga el código de Python, he decidido incluirla, ya que, desde mi punto de vista, a pesar de no influir notablemente en la complejidad, es interesante y añade una capa más de consideración al código.

3. (*) ¿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

Inicialmente, también para ayudarme con la claridad a la hora de plantear el problema, he decidido crear una clase para cada uno de los distintos objetos:

```
class Equipo:
    def __init__(self, nombre, categoria):
        self.nombre = nombre
        self.categoría = categoria
    def __repr__(self):
        return f"{self.nombre}({self.categoría})"
```

Y, de igual forma, con Horario, Coincidencia, Partido y Jornada. Ésta es la forma más sencilla que se me ha ocurrido de representar el equivalente de las tablas con datos presentadas al principio del notebook.

Más específicamente, en la clase Partido, he decidido que la asignación del enfrentamiento lo haría mediante una tupla combinada con el sorted para guardar el enfrentamiento. Esto evita que (EquipoA vs EquipoB) != (EquipoB vs EquipoA), ya que siempre recibirán el mismo orden al ser guardados.

La alternativa que tenía era utilizar una lista pero consideré que el uso de **tuplas** se adaptaba mejor al problema, ya que:

- Aportan inmutabilidad.
- Permiten comparaciones directas y uso como claves en diccionarios.
- Son más ligeras y rápidas que las listas (al no tener métodos para cambiar su contenido).

De esta forma, eliminamos el problema del orden de los equipos en un partido.

En mi caso, no guardo los resultados de las jornadas generadas a no ser que su resultado de audiencia sea mejor que el acumulado hasta el momento, y evito repeticiones gracias a utilizar **itertools.combinations**, lo que me permite tener todas las combinaciones posibles de partidos sin repetición.

Pero, si decidiera guardarlas, también utilizaría una **tupla** para tener una lista ordenada de los partidos y evitar repeticiones de partidos en las que solo varía el orden.

4. (*)¿Cual es la función objetivo?

Respuesta

Para cada una de las jornadas calculadas, la función objetivo es la suma de la audiencia de todos los partidos. Dentro de cada partido, la audiencia se calcula con la siguiente fórmula:

$$\text{Audiencia Partido} = \text{Audiencia Base Enfrentamiento} \times \text{Ponderación Horario} \times (1 - \text{Penalización coincidencia})$$

La expresión formal para el cálculo de la audiencia total de una jornada, por lo tanto, es la siguiente:

$$\text{Audiencia total} = \sum_{i=1}^{10} \text{Audiencia Base Enfrentamiento}_i \times \text{Ponderación Horario}_i \times (1 - \text{Penalización coincidencia}_i)$$

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

Respuesta

Se trata de un problema de **maximización**, ya que estamos intentando, dentro de nuestras combinaciones de jornadas, encontrar el valor máximo para la audiencia total.

6. Diseña un algoritmo para resolver el problema por fuerza bruta

Respuesta

In [5]: *#Primero creamos los objetos necesarios para representar nuestros datos:*

```
import itertools
import time
from collections import defaultdict

class Equipo:
    def __init__(self, nombre, categoria):
        self.nombre = nombre
        self.categoria = categoria
    def __repr__(self):
        return f"{self.nombre}({self.categoria})"

class Horario:
    def __init__(self, franja, audiencia):
        self.franja = franja
```

```

        self.audiencia = audiencia
    def __repr__(self):
        return f"{self.franja}"


class Coincidencia:
    valores = {
        0: 0,
        1: 0.25,
        2: 0.45,
        3: 0.6,
        4: 0.7,
        5: 0.75,
        6: 0.78,
        7: 0.8,
        8: 0.8,
    }
    def __init__(self, coincidencias):
        self.valor = self.valores.get(coincidencias, 0.8)
    def __repr__(self):
        return f"Coincidencia({self.valor})"


class Partido:
    audiencia_base_matriz = {
        ('A', 'A'): 2,
        ('A', 'B'): 1.3,
        ('A', 'C'): 1,
        ('B', 'B'): 0.9,
        ('B', 'C'): 0.75,
        ('C', 'C'): 0.47,
    }
    def __init__(self, equipo1, equipo2, horario, coincidencia):
        self.equipo1 = equipo1
        self.equipo2 = equipo2
        self.horario = horario
        self.coincidencia = coincidencia
        self.equipos = f"{equipo1.nombre} vs {equipo2.nombre}"
        self.total = self.calcular_total()

    def calcular_total(self):
        cat1 = self.equipo1.categoría
        cat2 = self.equipo2.categoría
        par = tuple(sorted((cat1, cat2)))
        audiencia_base = self.audiencia_base_matriz.get(par, 0)
        return audiencia_base * self.horario.audiencia * (1 - self.coincidencia)

    def __repr__(self):
        return f"Partido({self.equipos}, {self.horario.franja}, total={self.tota

class Jornada:
    def __init__(self, partidos=None):
        self.partidos = partidos or []
        self.total = sum(p.total for p in self.partidos)
    def __repr__(self):
        partidos_str = "\n".join(f" - {p}" for p in self.partidos)
        return f"Jornada:\n Audiencia Total: {self.total:.2f}\n Partidos:\n{part

```

In [6]: #Ahora generamos el set de datos con todos los equipos y franjas horarias

```

equipos = [
    Equipo('Barcelona', 'A'),

```

```

Equipo('Real Madrid', 'A'),
Equipo('Real Sociedad', 'A'),
Equipo('Alaves', 'B'),
Equipo('Athletic', 'B'),
Equipo('Atletico Madrid', 'B'),
Equipo('Betis', 'B'),
Equipo('Celta', 'B'),
Equipo('Espanyol', 'B'),
Equipo('Getafe', 'B'),
Equipo('Levante', 'B'),
Equipo('Sevilla', 'B'),
Equipo('Valencia', 'B'),
Equipo('Villarreal', 'B'),
Equipo('Eibar', 'C'),
Equipo('Granada', 'C'),
Equipo('Leganes', 'C'),
Equipo('Mallorca', 'C'),
Equipo('Osasuna', 'C'),
Equipo('Valladolid', 'C'),
]

franjas = [
    Horario('V20', 0.4),
    Horario('S12', 0.55),
    Horario('S16', 0.7),
    Horario('S18', 0.8),
    Horario('S20', 1),
    Horario('D12', 0.45),
    Horario('D16', 0.75),
    Horario('D18', 0.85),
    Horario('D20', 1),
    Horario('L20', 0.4)
]

```

```

In [7]: def generar_mejor_jornada_fuerza_bruta(equipos, franjas):
    inicio = time.time()
    partidos_posibles = list(itertools.combinations(equipos, 2))
    jornadas_validas = []

    for partidos_en_jornada in itertools.combinations(partidos_posibles, len(equipos)):
        equipos_usados = set()
        for e1, e2 in partidos_en_jornada:
            if e1 in equipos_usados or e2 in equipos_usados:
                break
            equipos_usados.update([e1, e2])
        else:
            jornadas_validas.append(partidos_en_jornada)

    mejor_jornada = None
    mejor_total = 0

    for jornada_base in jornadas_validas:
        franjas_asignadas = itertools.product(franjas, repeat=len(jornada_base))
        jornada = []
        franja_counts = defaultdict(int)

        for (equipo1, equipo2), horario in zip(jornada_base, franjas_asignadas):
            franja_counts[horario.franja] += 1
        for (equipo1, equipo2), horario in zip(jornada_base, franjas_asignadas):
            coincidencia = Coincidencia(franja_counts[horario.franja] - 1)

```

```

        partido = Partido(equipo1, equipo2, horario, coincidencia)
        jornada.append(partido)

        total_audiencia = sum(p.total for p in jornada)
        hay_viernes = any(p.horario.franja == 'V20' for p in jornada)
        hay_lunes = any(p.horario.franja == 'L20' for p in jornada)

        if hay_viernes and hay_lunes and total_audiencia > mejor_total:
            mejor_total = total_audiencia
            mejor_jornada = Jornada(jornada)

        tiempo_total = time.time() - inicio

        print("\nMejor jornada encontrada:")
        print(mejor_jornada)
        print("\nTiempo total:")
        print(tiempo_total)

#La ejecución queda comentada porque tardaría mucho, sin embargo, más adelante,
#de datos reducido

#generar_mejor_jornada_fuerza_bruta(equipos, franjas)

```

7. Calcula la complejidad del algoritmo por fuerza bruta

Respuesta

Ya que la fuerza bruta va a probar todas las combinaciones posibles, anteriormente habíamos calculado que esto venía definido por la siguiente fórmula:

$$O\left(\frac{n!}{\left(\frac{n}{2}\right)! \times 2^{\frac{n}{2}}}\right) \times \left(h^{\frac{n}{2}} - \left(2 \times (h-1)^{\frac{n}{2}} - (h-2)^{\frac{n}{2}}\right)\right)$$

donde n = número de partidos y h = número de franjas horarias.

Como parte del análisis de la complejidad, lo primero que haríamos sería eliminar constantes, pero en este caso no tenemos. Después, descartamos los factores que crecen más lentos.

En este caso, significaría eliminar los dos exponentiales que tenemos, ya que crecen más lento que el factorial.

Eliminamos:

$$2^{\frac{n}{2}}, h^{\frac{n}{2}}, y - \left(2 \times (h-1)^{\frac{n}{2}} - (h-2)^{\frac{n}{2}} \right)$$

Ahora ya solo nos quedan dos factoriales, pero es obvio que uno crece más lento que el otro, ya que $n! > (n/2)!$.

Por lo tanto, la complejidad crece factorialmente.

$O(n!)$

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

Respuesta

```
In [8]: def generar_mejor_jornada_ramificacion_poda(equipos, franjas):
    import sys
    inicio = time.time()

    partidos_posibles = list(itertools.combinations(equipos, 2))
    jornadas_validas = []

    for partidos_en_jornada in itertools.combinations(partidos_posibles, len(equipo
        equipos_usados = set()
        for e1, e2 in partidos_en_jornada:
            if e1 in equipos_usados or e2 in equipos_usados:
                break
            equipos_usados.update([e1, e2])
        else:
            jornadas_validas.append(partidos_en_jornada)

    mejor_total = -sys.maxsize
    mejor_jornada = None

    audiencia_base_maxima = max(Partido.audiencia_base_matriz.values())
    audiencia_maxima = max(f.audiencia for f in franjas)

    def calcular_total_jornada_con_coincidencias(partidos):
        franja_map = defaultdict(list)
        for p in partidos:
            franja_map[p.horario.franja].append(p)

        total = 0
        for lista in franja_map.values():
            coincidencia = Coincidencia(len(lista) - 1)
            for p in lista:
                cat1, cat2 = p.equipo1.categoría, p.equipo2.categoría
                par = tuple(sorted((cat1, cat2)))
                audiencia_base = Partido.audiencia_base_matriz.get(par, 0)
                penalizada = audiencia_base * p.horario.audiencia * (1 - coincidencia)
```

```

        total += penalizada
    return total

def backtrack(jornada_base, idx, partidos_actuales, franja_counts):
    nonlocal mejor_total, mejor_jornada

    if idx == len(jornada_base):
        hay_viernes = any(p.horario.franja == 'V20' for p in partidos_actuales)
        hay_lunes = any(p.horario.franja == 'L20' for p in partidos_actuales)

        if hay_viernes and hay_lunes:
            total_con_coincidencias = calcular_total_jornada_con_coincidencias(partidos_actuales)
            if total_con_coincidencias > mejor_total:
                mejor_total = total_con_coincidencias
                mejor_jornada = Jornada(partidos_actuales[:])
    return

equipo1, equipo2 = jornada_base[idx]
for franja in franjas:
    partido = Partido(equipo1, equipo2, franja, Coincidencia(0))

    partidos_actuales.append(partido)
    franja_counts[franja.franja] += 1

    max_restante = (len(jornada_base) - idx - 1) * (audiencia_base_maxima * 2)
    total_parcial = calcular_total_jornada_con_coincidencias(partidos_actuales)
    if total_parcial + max_restante > mejor_total:
        backtrack(jornada_base, idx + 1, partidos_actuales, franja_counts)

    partidos_actuales.pop()
    franja_counts[franja.franja] -= 1

for jornada_base in jornadas_validas:
    backtrack(jornada_base, 0, [], defaultdict(int))

tiempo_total = time.time() - inicio

print("\nMejor jornada encontrada:")
print(mejor_jornada)
print("\nTiempo total:")
print(tiempo_total)

# Ejecutar con los datos dados

#generar_mejor_jornada_ramificacion_poda(equipos, franjas)

```

¿Por qué mejora a la fuerza bruta?

Porque, mediante el uso de ramificación y poda, en vez de iterar por todas las combinaciones posibles, si consideramos que, con la selección actual y la mejor audiencia teórica de los partidos y horarios restantes por asignar, no vamos a conseguir mejorar el mejor resultado encontrado hasta el momento, podamos esa rama y dejamos de explorarla.

De esta forma, evitamos explorar ramas que sabemos que no van a llevar a un resultado óptimo y, por lo tanto, mejoramos el tiempo de ejecución reduciendo iteraciones inútiles.

9.(*)Calcula la complejidad del algoritmo

Respuesta

En lugar de evaluar todas las posibles combinaciones de franjas (como hace fuerza bruta), se recorre un árbol de decisiones usando backtracking. Es posible que, en el peor de los casos, la poda no llegue a ocurrir, por lo que la complejidad teórica del algoritmo sigue siendo factorial, al igual que la fuerza bruta.

Sin embargo, la **complejidad práctica** es mucho más eficiente, especialmente cuando las combinaciones malas se podan pronto.

Si la diferencia entre las audiencias dependiendo de los partidos fuera drásticamente más alta, sería mucho más fácil para la poda identificar ramas que no producen un buen resultado, ya que una mala combinación de partidos ya condenaría toda la rama y sería podada pronto.

Por desgracia, en este caso, debido a que en muchos casos la audiencia total depende altamente de la asignación de audiencia, tomar el caso ideal de máxima audiencia en el horario a la hora de calcular la validez de una rama, hace que la poda ocurra muy tarde.

10. Segúin el problema (y tenga sentido), diseña un juego de datos de entrada aleatorios

Respuesta

```
In [9]: equipos_reducido = [
    Equipo('Barcelona', 'A'),
    Equipo('Real Madrid', 'A'),
    Equipo('Atletico Madrid', 'B'),
    Equipo('Sevilla', 'B'),
    Equipo('Celta', 'B'),
    Equipo('Getafe', 'B'),
    Equipo('Levante', 'B'),
    Equipo('Villarreal', 'B'),
    Equipo('Eibar', 'C'),
    Equipo('Valladolid', 'C'),
    Equipo('Ocasuna', 'C'),
    Equipo('Leganes', 'C')
]

franjas_reducido = [
    Horario('V20', 0.4),
    Horario('S12', 0.55),
    Horario('S16', 0.7),
    Horario('S20', 1),
    Horario('D12', 0.45),
    Horario('L20', 0.4)
]
```

11. Aplica el algoritmo al juego de datos generado

Respuesta

```
In [10]: #Dejo comentados los métodos para evitar tiempos de ejecución Largos, pero adjunto anterior

#Fuerza bruta:
print("\nFuerza Bruta")
#generar_mejor_jornada_fuerza_bruta(equipos_reducido, franjas_reducido)
print("\n-----")
#Ramificación y poda
print("\nRamificación y poda")
#generar_mejor_jornada_ramificacion_poda(equipos_reducido, franjas_reducido)
```

Fuerza Bruta

Ramificación y poda

Resultados:

Fuerza Bruta

Mejor jornada encontrada:

Jornada:

Audiencia Total: 3.98

Partidos:

- Partido(Barcelona vs Real Madrid, S20, total=2.00)
- Partido(Atletico Madrid vs Sevilla, S16, total=0.63)
- Partido(Celta vs Eibar, S12, total=0.41)
- Partido(Getafe vs Valladolid, V20, total=0.30)
- Partido(Levante vs Osasuna, D12, total=0.34)
- Partido(Villarreal vs Leganes, L20, total=0.30)

Tiempo total:

4614.917927742004

Ramificación y poda

Mejor jornada encontrada:

Jornada:

Audiencia Total: 3.98

Partidos:

- Partido(Barcelona vs Real Madrid, S20, total=2.00)
- Partido(Atletico Madrid vs Sevilla, S16, total=0.63)
- Partido(Celta vs Eibar, S12, total=0.41)
- Partido(Getafe vs Valladolid, V20, total=0.30)
- Partido(Levante vs Osasuna, D12, total=0.34)
- Partido(Villarreal vs Leganes, L20, total=0.30)

Tiempo total:

2834.926131248474

[EXTRA] utilización de métodos heurísticos

Se podría plantear la idea de explorar la utilización de métodos heurísticos para conseguir una muy buena solución, aunque no fuera la óptima.

Un ejemplo de esto sería utilizar un algoritmo **voraz** que tomara la lista de equipos y franjas por audiencia y asignara los emparejamientos siempre escogiendo la posibilidad que más audiencia proporciona en ese momento.

A continuación, la implementación y ejecución de un algoritmo voraz que hace esto mismo:

```
In [11]: from itertools import combinations
def generar_jornada_voraz(equipos, franjas):
    equipos_disponibles = set(equipos)
    franja_partidos = defaultdict(list)

    total_partidos = len(equipos) // 2
    while len(equipos_disponibles) >= 2:
        mejor_total_global = -1
        mejor_comb = None

        for equipo1, equipo2 in combinations(equipos_disponibles, 2):
            for franja in franjas:
                # Simular agregar este partido a esta franja
                simulacion_franjas = defaultdict(list)
                for k, v in franja_partidos.items():
                    simulacion_franjas[k] = v.copy()
                simulacion_franjas[franja].append((equipo1, equipo2))

                partidos_asignados = total_partidos - ((len(equipos_disponibles) - 1) * 2)
                hay_viernes = 'V20' in simulacion_franjas
                hay_lunes = 'L20' in simulacion_franjas
                if partidos_asignados == total_partidos and (not hay_viernes or
                    continue

                if mejor_total_global <= sum(v[1] for v in simulacion_franjas.values()):
                    mejor_total_global = sum(v[1] for v in simulacion_franjas.values())
                    mejor_comb = (equipo1, equipo2, franja)

    return mejor_comb
```

```
total_simulado = 0
for franja_key, partidos in simulacion_franjas.items():
    penalizacion = Coincidencia(len(partidos) - 1).valor
    franja_obj = next(f for f in franjas if f.franja == franja_key)
    for eq1, eq2 in partidos:
        cat1, cat2 = eq1.categoría, eq2.categoría
        par = tuple(sorted((cat1, cat2)))
        audiencia_base = Partido.audiencia_base_matriz.get(par,
            audiencia_real = audiencia_base * franja_obj.audiencia *
            total_simulado += audiencia_real

        if total_simulado > mejor_total_global:
            mejor_total_global = total_simulado
            mejor_comb = (equipo1, equipo2, franja)

    if not mejor_comb:
        break

equipo1, equipo2, franja = mejor_comb
franja_key = franja.franja

franja_partidos[franja_key].append((equipo1, equipo2))
equipos_disponibles.remove(equipo1)
equipos_disponibles.remove(equipo2)

partidos_resultado = []
for franja_key, emparejamientos in franja_partidos.items():
    franja = next(f for f in franjas if f.franja == franja_key)
    coincidencia = Coincidencia(len(emparejamientos) - 1)
    for equipo1, equipo2 in emparejamientos:
        partido = Partido(equipo1, equipo2, franja, coincidencia)
        partidos_resultado.append(partido)

jornada = Jornada(partidos_resultado)
return jornada

inicio = time.time()
jor = generar_jornada_voraz(equipos_reducido, franjas_reducido)
tiempo_total = time.time() - inicio

print("\nVoraz:")
print("\nMejor Jornada Encontrada:")
print(jor)
print("\nTiempo total:")
print(f"{{tiempo_total:.4f}}")
```

Voraz:

Mejor Jornada Encontrada:

Jornada:

Audiencia Total: 3.91

Partidos:

- Partido(Real Madrid vs Barcelona, S20, total=2.00)
- Partido(Celta vs Sevilla, S16, total=0.63)
- Partido(Levante vs Villarreal, S12, total=0.50)
- Partido(Atletico Madrid vs Getafe, D12, total=0.41)
- Partido(Valladolid vs Eibar, V20, total=0.19)
- Partido(Osasuna vs Leganes, L20, total=0.19)

Tiempo total:

0.0030

En este caso, podemos ver cómo la audiencia encontrada es muy parecida a la óptima generada por el set de datos aleatorios, siendo la óptima **3.98** y esta **3.91**.

Sin embargo, la diferencia del tiempo de ejecución es abismalmente alta ya que el algoritmo greedy construirá una única jornada en base a siempre elegir la máxima audiencia sin considerar las consecuencias futuras.

Hay que tener en cuenta que, con la variación de la coincidencia, ya que tanto el partido por asignar como los anteriores se verían afectados al ser asignados a la misma franja, nuestro algoritmo tendrá que tener eso en cuenta antes de considerar si está seleccionando la opción más óptima (intentando no afectar al la audiencia global por intentar obtener un máximo local). Por ello, la implementación de código es más compleja, pero el resultado final es excelente.

Gracias a que nuestro algoritmo voraz es tan rápido, podemos utilizarlo en el set de datos completo inicial, para ver cuál sería el resultado de una jornada completa casi óptima:

```
In [12]: inicio = time.time()
jor = generar_jornada_voraz(equipos, franjas)
tiempo_total = time.time() - inicio

print("\nVoraz:")
print("\nMejor Jornada Encontrada:")
print(jor)
print("\nTiempo total:")
print(f"{tiempo_total:.4f}")
```

Voraz:

Mejor Jornada Encontrada:

Jornada:

Audiencia Total: 7.17

Partidos:

- Partido(Barcelona vs Real Sociedad, S20, total=2.00)
- Partido(Villarreal vs Real Madrid, D20, total=1.30)
- Partido(Athletic vs Alaves, D18, total=0.77)
- Partido(Sevilla vs Valencia, S18, total=0.72)
- Partido(Getafe vs Espanyol, D16, total=0.68)
- Partido(Levante vs Atletico Madrid, S16, total=0.63)
- Partido(Betis vs Celta, S12, total=0.50)
- Partido(Eibar vs Valladolid, D12, total=0.21)
- Partido(Osasuna vs Mallorca, V20, total=0.19)
- Partido(Leganés vs Granada, L20, total=0.19)

Tiempo total:

0.0304

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

Respuesta

- **GeeksforGeeks.** (2023). Backtracking - Introduction.
<https://www.geeksforgeeks.org/backtracking-introduction/> (Accedido el 17 de junio de 2025).
- **GeeksforGeeks.** (2023). Branch and Bound Algorithm.
<https://www.geeksforgeeks.org/branch-and-bound-algorithm/> (Accedido el 16 de junio de 2025).
- **Brilliant.org.** (n.d.). Branch and Bound. <https://brilliant.org/wiki/branch-and-bound/> (Accedido el 20 de junio de 2025).
- **Towards Data Science.** (2020). The Power of Branch and Bound in Optimization.
<https://towardsdatascience.com/the-power-of-branch-and-bound-in-optimization-5c58fce7474d> (Accedido el 15 de junio de 2025).
- **Khan Academy.** (n.d.). Introduction to Combinatorics.
<https://www.khanacademy.org/computing/computer-science/cryptography/combinatorics/a/combinatorics-and-basic-counting> (Accedido el 15 de junio de 2025).

- **W3Schools.** (2023). Python Tutorial. <https://www.w3schools.com/python/> (Accedido el 10 de junio de 2025).
- **GeeksforGeeks.** (2023). Greedy Algorithm - Introduction. <https://www.geeksforgeeks.org/greedy-algorithms/> (Accedido el 23 de junio de 2025).

Manual de la Asignatura

- **Fuerza bruta:** Capítulo 2: Búsqueda exhaustiva Sección: "2.1 Fuerza bruta" Página 24
- **Ramificación y poda:** Capítulo 4: Ramificación y poda Páginas 60 a 66
- **Backtracking:** Capítulo 4: Ramificación y poda Página 60
- **Algoritmos voraces** (greedy): Capítulo 3: Algoritmos voraces Páginas 43 a 59
- **Heurísticas:** Capítulo 3 y 4 Página 57 y 61

13. Describe brevemente las líneas 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

La forma en la que yo variaría el problema es de dos formas:

- La primera: calcular un calendario completo de liga, con sus 38 jornadas, en las que cada equipo tiene que enfrentarse dos veces (ida y vuelta) a cada uno de los demás equipos, de una forma en la que las jornadas intenten maximizar la audiencia dentro de las limitaciones posibles.
- La segunda: incluir una condición que se da en la vida real en la que cada equipo alterne a jugar, de local y de visitante. y esto vaya alternando entre jornadas.

En cierto sentido, estas condiciones facilitan una parte del problema. A medida que vamos calculando jornadas, las combinaciones de los equipos con el resto se van reduciendo, ya que no pueden enfrentarse a los que ya se han enfrentado en la primera vuelta. Sin embargo, la variante de local-visitante, sí que complica las cosas.

En otros países, como Italia, existen equipos que comparten estadio. Esto supondría una limitación extra a la hora de calcular las jornadas ya que no podrían jugar ambos como

locales en la misma franja horaria. Esta variación también podría ser interesante complicar el enunciado problema.

Con respecto al estudio del problema, se podría hacer una combinación entre los dos métodos propuestos para mejorar nuestro algoritmo de ramificación y poda y obtener el resultado óptimo de la siguiente forma:

- Utilizar el algoritmo voraz para rápidamente obtener una jornada cercana a la óptima.
- Ejecutar el algoritmo de ramificación y poda con los datos pero utilizando como mejor jornada aquella que ha sido calculada por el algoritmo voraz.

De esta forma, la poda rápidamente podaría antes aquellas ramas que no mejoran una solución que ya de inicio es muy buena y, por tanto, tendría un tiempo de ejecución total menor.