Calculo posibilidades de partidos

Creo recordar que dijiste que se debía de añadir en el cálculo de las posibilidades las combinatorias de los equipos por lo que lo dejaré como constante `c` que se multiplicará con las posibilidades que se calcularán posteriormente.

Hay **20 equipos** y cada jornada consta de **10 partidos**. Además teniendo en cuenta las jornadas reales se estipulan las siguinetes condiciones:

- Cada equipo juega **exactamente un partido** en la jornada por lo que **no se puede repetir ningún equipo**.
- **Importa quién es local y quién es visitante** en cada partido.

Para calcularlo lo dividiré en 3 partes:

Parte 1. Emparejar los equipos en partidos

Primero contaré cuántas formas distintas existen de emparejar los 20 equipos en 10 partidos **sin considerar todavía quien es el local y el visitante**.

Para un conjunto de $2n$ elementos ($n = 10$ el número de partidos), el número de emparejamientos perfectos (es decir, particiones en parejas) viene dado por la fórmula:

$$\frac{2n!}{2^n \cdot n!}$$

donde:

- $2n!$ cuenta todas las permutaciones posibles de los equipos.
- 2^n elimina el orden interno de cada pareja.
- $n!$ elimina el orden entre los partidos ya que en esta parte solo se calcula el número de combinaciones de los posibles partidos, el orden se hará en los apartados siguientes.

Parte 2. Asignar local y visitante

Para cada partido existen **2 posibilidades**:

- Equipo A local, equipo B visitante
- Equipo B local, equipo A visitante

Como hay n partidos, el número total de asignaciones es: 2^n

Parte 3. Número total de posibles

La fórmula general será el producto de ambas posibilidades:

$$\frac{2n!}{2^n \cdot n!} \times 2^n$$

Los factores 2^n se cancelan, quedando:

$$\frac{2n!}{n!}$$

Por lo que el número total de partidos distintos posibles, **teniendo en cuenta la localía** sabiendo que hay 10 partidos, es:

$$c = \frac{20!}{10!}$$

Postdata

Como he dicho consideré que sí importa quién juega de local ya que en las restricciones y en la realidad sí que importa pero si **no importara** quién es local y visitante, el número de jornadas sería:

$$c = \frac{20!}{2^{10} \cdot 10!}$$

¿Cuántas posibilidades hay sin tener en cuenta las restricciones?

Al tener **10 partidos** y **12 horarios (slots)**, y cada partido puede asignarse independientemente a cualquiera de los horarios al no tener restricciones, el número total de configuraciones posibles sería:

$$\underbrace{12 \times 12 \times \cdots \times 12}_{10 \text{ veces}} = 12^{10}$$

$$12^{10} = 61.917.364.224 \text{ combinaciones posibles.}$$

A esto habría que multiplicarlo con la constante `c` que son las posibilidades de partidos distintas.

¿Cuántas posibilidades hay teniendo en cuenta todas las restricciones?

Sin restricciones, cada uno de los $N = 10$ partidos puede asignarse a cualquiera de los $S = 12$ slots horarios, por lo que el número total de configuraciones posibles viene dado por la regla del producto: 12^{10} como se ha explicado previamente.

Ahora queremos saber cuántas cumplen las restricciones. La única restricción que limita el número de posibilidades es la restricción de jornada válida, la cual obliga a que haya al menos un partido en cada día (viernes, sábado, domingo y lunes). Esto en los slots

corresponden a: slot 1 (viernes), alguno de los slots 2–6 (sábado), alguno de los slots 7–11 (domingo) y el slot 12 (lunes).

Para determinar el número de soluciones que cumplen con la restricción, aplicamos el **Principio de Inclusión-Exclusión**. La fórmula para calcular las configuraciones válidas es:

$$Config = Total - \sum Vacías_1 + \sum Vacías_2 - \sum Vacías_3 + \sum Vacías_4$$

Donde:

- **Total:** El espacio muestral completo sin restricciones: 12^{10}.
- $\sum Vacías_1$**:** Combinaciones donde una zona específica (V, S, D o L) está vacía. (Cardinalidades simples)
- $\sum Vacías_2$**:** Configuraciones donde dos zonas están vacías simultáneamente. (Intersecciones dobles)
- $\sum Vacías_3$**:** Configuraciones donde tres zonas están vacías simultáneamente (Intersecciones de orden 3).
- $\sum Vacías_4$**:** Configuraciones donde las cuatro zonas están vacías. En este caso el valor es 0, ya que los 10 partidos deben asignarse obligatoriamente a algún slot.

Cardinalidades simples Cada partido puede seleccionarse independientemente entre las franjas restantes. El cardinal de cada conjunto de exclusión $|A_i|$ representa las combinaciones donde la zona i queda desierta:

- **Sin Viernes (Slot 1):** Quedan 11 slots disponibles $\implies |A_1| = 11^{10}$
- **Sin Sábado (Slots 2–6):** Quedan 7 slots disponibles $\implies |A_2| = 7^{10}$
- **Sin Domingo (Slots 7–11):** Quedan 7 slots disponibles $\implies |A_3| = 7^{10}$
- **Sin Lunes (Slot 12):** Quedan 11 slots disponibles $\implies |A_4| = 11^{10}$

Suma de la primera iteración sería: $\sum |A_i| = 2(11^{10}) + 2(7^{10})$

Intersecciones Dobles ($\sum |A_i \cap A_j|$):

- **Sin Viernes y Sábado ($A_1 \cap A_2$)** Quedan 6 (Slots 7–12) $\implies 6^{10}$
- **Sin Viernes y Domingo ($A_1 \cap A_3$)** Quedan 6 (Slots 2–6, 12) $\implies 6^{10}$
- **Sin Viernes y Lunes ($A_1 \cap A_4$)** Quedan 10 (Slots 2–11) $\implies 10^{10}$
- **Sin Sábado y Domingo ($A_2 \cap A_3$)** Quedan 2 (Slots 1, 12) $\implies 2^{10}$
- **Sin Sábado y Lunes ($A_2 \cap A_4$)** Quedan 6 (Slots 1, 7–11) $\implies 6^{10}$
- **Sin Domingo y Lunes ($A_3 \cap A_4$)** Quedan 6 (Slots 1–6) $\implies 6^{10}$

Suma de la segunda iteración sería: $\sum |A_i| = 4(6^{10}) + 10^{10} + 2^{10}$

Intersecciones Triples ($\sum |A_i \cap A_j \cap A_k|$):

- Sin Viernes, Sábado y Domingo: Queda 1 slot (12) $\implies 1^{10} = 1$
- Sin Viernes, Sábado y Lunes: Quedan 5 slots (7–11) $\implies 5^{10}$
- Sin Viernes, Domingo y Lunes: Quedan 5 slots (2–6) $\implies 5^{10}$
- Sin Sábado, Domingo y Lunes: Queda 1 slot (1) $\implies 1^{10} = 1$

Suma de la tercera iteración sería: $\sum |A_i| = 2(5^{10}) + 2(1^{10}) = 2(5^{10}) + 2$

Intersección Cuádruple:

- Sin ninguna de las franjas obligatorias: 0 slots restantes $\implies 0^{10} = 0$.

por lo tanto la ecuación inicial quedaría:
$12^{10} - (2(11^{10}) + 2(7^{10})) + (4(6^{10}) + 10^{10} + 2^{10}) - (2(5^{10}) + 2) = 19.699.899.000$ combinaciones posibles.

A esto habría que multiplicarlo con la constante `c` que son las posibilidades de partidos distintas.

¿Cuál es la estructura de datos que mejor se adapta al problema?

Sobre el genotipo (Codificación del Cromosoma) en el modelo para representar al cromosoma (Jornada), he elegido listas de enteros de longitud fija ($N = 10$ número de partidos que se juegan en cada jornada). Esto lo he hecho ya que el uso de enteros permite que el cruce (intercambio de segmentos) y la mutación (cambio de valor o intercambio de posiciones) sean operaciones con una complejidad mucho menor a si hubiera usado una lista binaria que exigiría mapear cada posible asignación a un bit o conjunto de bits lo que generaría cromosomas muy largos y dificultaría la interpretación, y una lista con valores floats no tendría sentido ya que he asignado a cada horario un número, se podria haber asociado a cada día un número y dividir el incremento entre los distintos horarios del día pero lo he visto bastante más complicado. Cada posición del índice representa un partido específico (gen), mientras que el valor almacenado representa el slot horario asignado (0-11). No lo he mencionado pero la población sería el conjunto de cromosomas, es decir, el conjunto de jornadas que se evaluará con la función fitness.

Para el fenotipo (Representación de la Jornada) he usado diccionario de Listas donde las claves son los horarios y el día en forma de tupla, y los valores son los partidos asignados en una lista. Esto lo he hecho ya que permite identificar de forma sencilla los partidos que se disputan junto con el horario a modo de calendario.

A fuerza bruta

La función fitness y todas las funciones auxiliares de la misma están en la parte del algoritmo genético, te coloco está parte antes ya que así aparece en la presentación y no se va a reproducir.

In [1]:
```python
import itertools

def resolver_fuerza_bruta_total(partidos, slots_disponibles):

    num_partidos = len(partidos)
    num_slots = len(slots_disponibles)
```

```
    mejor_audiencia = -1
    mejor_configuracion = None

    # itertools.product produce todas las combinaciones posibles de los elemento
    # mediante producto cartesiano. He visto que es eficiente en memoria porque
    espacio_busqueda = itertools.product(range(num_slots), repeat=num_partidos)

    for combinacion in espacio_busqueda:
        # Convertimos la tupla de la combinación en el formato 'individuo' (list
        individuo = list(combinacion)

        # Reutilizamos la función de fitness (que incluye validación y audiencia
        audiencia_actual = fitness_final(individuo, partidos, slots_disponibles)

        if audiencia_actual > mejor_audiencia:
            mejor_audiencia = audiencia_actual
            mejor_configuracion = individuo

    return mejor_configuracion, mejor_audiencia
```

Calcula la complejidad del algoritmo por fuerza bruta.

El algoritmo de fuerza bruta fue diseñado para **encontrar el máximo global** mediante la evaluación de cada punto del espacio muestral o sea de todas las combinaciones posibles.

1. Análisis del Espacio de Búsqueda

Siendo N el número de partidos y S el número de slots horarios:

- Cada partido puede asignarse a cualquiera de los S slots de forma independiente.
- La combinatoria resultante es un producto cartesiano: $S \times S \times \ldots \times S$ (N veces) lo que es lo mismo S^N.

2. Complejidad Temporal (Big O)

- **Generación de combinaciones:** $O(S^N)$.
- **Evaluación de Fitness:** Por cada combinación, se ejecutan las funciones auxiliares con coste $O(N)$, la cual incluye validación de restricciones.
- **Complejidad Total:** $O(N \cdot S^N)$.

Algoritmo genético

In [1]:
```
# Partidos de la jornada (anfitrion, visitante)
partidos = [
    ("Athletic Club", "Real Madrid CF"),
    ("Getafe CF", "Real Sociedad de Fútbol"),
    ("Girona FC", "Club Atlético Osasuna"),
    ("Villarreal CF", "Deportivo Alavés"),
```

```
    ("FC Barcelona", "Club Atlético de Madrid"),
    ("Real Oviedo", "Real Betis Balompié"),
    ("Sevilla FC", "RC Celta de Vigo"),
    ("Valencia CF", "Elche CF"),
    ("Levante UD", "RCD Espanyol de Barcelona"),
    ("Rayo Vallecano de Madrid", "RCD Mallorca")
]
```

In [2]:
```
# Diccionario directo: Equipo -> Categoría
equipo_categoria = {
    # --- Categoría A ---
    "Real Madrid CF": "A",
    "FC Barcelona": "A",
    "Club Atlético de Madrid": "A",

    # --- Categoría B ---
    "Athletic Club": "B",
    "Real Sociedad de Fútbol": "B",
    "Villarreal CF": "B",
    "Real Betis Balompié": "B",
    "Sevilla FC": "B",
    "Valencia CF": "B",
    "Girona FC": "B",
    "Getafe CF": "B",
    "RC Celta de Vigo": "B",
    "Rayo Vallecano de Madrid": "B",

    # --- Categoría C ---
    "RCD Mallorca": "C",
    "Club Atlético Osasuna": "C",
    "Deportivo Alavés": "C",
    "RCD Espanyol de Barcelona": "C",
    "Elche CF": "C",
    "Levante UD": "C",
    "Real Oviedo": "C"
}
```

In [3]:
```
# Multiplicadores de audiencia por slot
# Formato: (Día, Hora): Coeficiente
coeficientes_horarios = {
    ("Viernes", "20:00"): 0.5,
    ("Sábado", "12:00"): 0.55,
    ("Sábado", "16:00"): 0.7,
    ("Sábado", "18:00"): 0.9,
    ("Sábado", "20:00"): 0.9,
    ("Sábado", "22:00"): 1.0, # Referencia base
    ("Domingo", "12:00"): 0.6,
    ("Domingo", "16:00"): 0.9,
    ("Domingo", "18:00"): 0.7,
    ("Domingo", "20:00"): 0.8,
    ("Domingo", "22:00"): 0.7,
    ("Lunes", "20:00"): 0.5
}
# Convertimos las claves en una lista para tener un orden fijo un ejemplo sería
# esto lo hago para tenerlos por indice y asi aplicar las mutaciones variando so
slots_disponibles = list(coeficientes_horarios.keys())

# Penalización por coincidencia de horarios
# Índice = número de coincidencias (además del propio partido)
```

```
# Si hay 2 partidos a la vez, hay 1 coincidencia para cada uno.
penalizaciones_coincidencia = {
    0: 0.0,  # 0% reducción
    1: 0.25, # 25% reducción
    2: 0.40, # 40% reducción
    3: 0.50, # 50% reducción
    4: 0.60, # 60% reducción
    5: 0.65, # 65% reducción
    6: 0.70, # 70% reducción
    7: 0.75  # 75% reducción
}
```

In [4]:
```
# Matriz de Audiencias Base (en Millones) - Sábado 22h
# Filas: Local, Columnas: Visitante
audiencia_base = {
    "A": {"A": 5.0, "B": 4.3, "C": 3.5},
    "B": {"A": 4, "B": 3.5, "C": 2.0},
    "C": {"A": 2.8, "B": 1.7, "C": 1.0}
}
```

In [5]:
```
def construir_jornada(indices_slots, partidos, slots_disponibles):
    # Inicializamos el diccionario vacío con los slots
    asignacion_jornada = {}
    for slot in slots_disponibles:
        asignacion_jornada[slot] = []

    # Llenamos el diccionario según los índices que propuso el algoritmo
    for i, idx_slot in enumerate(indices_slots):
        slot = slots_disponibles[idx_slot]
        partido = partidos[i]
        asignacion_jornada[slot].append(partido)

    return asignacion_jornada
```

In [6]:
```
def es_jornada_valida(asignacion_jornada):
    # No se va a comprobar si hay partidos en otros dias de la semana ya que se
    # Definimos los días obligatorios, los especifico en vez de restar 4 por si
    dias_requeridos = {"Viernes", "Sábado", "Domingo", "Lunes"}

    # Extraemos los días que tienen al menos un partido asignado
    dias_con_partidos = set()
    for (dia, hora), partidos in asignacion_jornada.items():
        if len(partidos) > 0:
            dias_con_partidos.add(dia)

    # Calculamos cuántos días de los obligatorios NO tienen partidos
    dias_faltantes = dias_requeridos - dias_con_partidos

    # Devolvemos el número entero de días que faltan (0, 1, 2, 3 o 4)
    return len(dias_faltantes)
```

In [7]:
```
def calcular_audiencia_partido(partido, dia, hora, num_partidos_en_slot):
    # Obtenmos audiencia base
    cat_local = equipo_categoria[partido[0]]
    cat_vis = equipo_categoria[partido[1]]
    base = audiencia_base[cat_local][cat_vis]


    # Aplicamos multiplicador de horario
```

```
    # Usamos .get() por seguridad, con 1.0 como valor por defecto
    coef_horario = coeficientes_horarios.get((dia, hora), 1.0)
    audiencia_con_horario = base * coef_horario

    # Aplicamos penalización por coincidencia de partidos
    # coincidencias = total de partidos en ese slot menos el partido actual
    coincidencias = num_partidos_en_slot - 1
    reduccion = penalizaciones_coincidencia.get(coincidencias, 0.0)
    audiencia_final = audiencia_con_horario * (1 - reduccion)

    return audiencia_final
```

Definición de la Función Fitness

En esta parte del proyecto (una única jornada), el fitness representa la **audiencia total estimada** de una jornada de liga, ajustada mediante penalizaciones por incumplimiento de restricciones.

La función `fitness_final` la he dividido en tres etapas:

1. **Decodificación:** Transforma los índices de slots del cromosoma en una estructura de `jornada` (diccionario) mediante la función `construir_jornada`.
2. **Cálculo de Audiencia Bruta:** Suma la audiencia de cada partido con la función `calcular_audiencia_partido` considerando:
   - La **categoría** de los equipos enfrentados.
   - El **coeficiente horario** asignado al slot (día y hora).
   - La **penalización por coincidencia**, que reduce la audiencia si varios partidos compiten en el mismo horario.
3. **Validación y Penalización por Restricciones:** Se comprueba si la jornada cumple con la restricción de tener partidos repartidos en los 4 días requeridos (Viernes, Sábado, Domingo y Lunes).

Para asegurar que el algoritmo converja hacia soluciones válidas (con jornadas en cada día), se resta una constante ($5,000,000$ por cada día faltante) al total de la audiencia.

Esto lo he hecho para prácticamente garantizar que, en la **Selección por Torneo**, cualquier jornada válida sea siempre superior a una inválida, pero permitiendo que las jornadas "menos malas" (con menos días faltantes) sobrevivan y evolucionen frente a las peores para así gracias al cruce o a las mutaciones puedan dar hijos válidos con gran audiencia.

In [ ]:
```
def fitness_final(indices_slots, lista_partidos,slots_disponibles):
    # Convertimos los indices de los partidos junto con la lista de partidos a e
    jornada = construir_jornada(indices_slots, lista_partidos,slots_disponibles)

    # Calculamos la audiencia total
    total_audiencia = 0
    for slot, partidos in jornada.items():
        dia, hora = slot
        num_partidos = len(partidos)
        for p in partidos:
            total_audiencia += calcular_audiencia_partido(p, dia, hora, num_part
```

```
    # Comprobamos si es válida en función de los días y penalizamos si no lo es,
    num_faltantes = es_jornada_valida(jornada)
    if num_faltantes == 0:
        return total_audiencia
    else:
        # En función del número de días faltantes se le penaliza en la audiencia
        return total_audiencia - (num_faltantes * 5000000)
```

In [24]:
```
import random
# predetermino 10 y 12 ya que en principio siempre tendrán estos valores
def generar_individuo_aleatorio(num_partidos=10, num_slots=12):
    individuo = []
    for i in range(num_partidos):
        slot_aleatorio = random.randint(0, num_slots - 1)
        individuo.append(slot_aleatorio)
    return individuo
```

In [21]:
```
def seleccionar_ganador_torneo(poblacion, lista_partidos, slots_disponibles, tam
    # Elegimos unos cuantos individuos al azar de la población
    participantes_torneo = random.sample(poblacion, tamaño_torneo)

    # Buscamos quién es el mejor de esos participantes
    mejor_individuo = participantes_torneo[0]
    mejor_audiencia = fitness_final(mejor_individuo, lista_partidos, slots_dispo

    for individuo in participantes_torneo[1:]:
        # Calculamos su audiencia usando la función de fitness que ya tenemos
        audiencia = fitness_final(individuo, lista_partidos, slots_disponibles)

        if audiencia > mejor_audiencia:
            mejor_audiencia = audiencia
            mejor_individuo = individuo

    return mejor_individuo
```

In [11]:
```
def cruzar_padres(padre1, padre2):
    # Elegimos un punto de corte mitad
    punto_corte = len(padre1) // 2

    # El hijo toma la primera parte del padre1 y la segunda del padre2
    hijo = []

    # Primera mitad
    for i in range(0, punto_corte):
        hijo.append(padre1[i])

    # Segunda mitad
    for i in range(punto_corte, len(padre2)):
        hijo.append(padre2[i])

    return hijo
```

In [12]:
```
# Mutación 1: Cambiar un partido a un slot nuevo al azar
def mutacion_aleatoria(individuo, num_slots=12, probabilidad=0.1):
    for i in range(len(individuo)):
        if random.random() < probabilidad:
            individuo[i] = random.randint(0, num_slots - 1)
    return individuo
```

```python
# Mutación 2: Intercambiar los horarios de dos partidos entre sí
def mutacion_intercambio(individuo, probabilidad=0.05):
    if random.random() < probabilidad:
        # Elegimos dos posiciones (partidos) al azar entre 0 y 9
        idx1 = random.randint(0, len(individuo) - 1)
        idx2 = random.randint(0, len(individuo) - 1)

        # Intercambiamos sus valores
        individuo[idx1], individuo[idx2] = individuo[idx2], individuo[idx1]

    return individuo
```

In [25]:

```python
import copy

def ejecutar_algoritmo_genetico(partidos, slots_disponibles,
                                tam_poblacion=1000,
                                generaciones_max=200,
                                paciencia=20):

    # PASO 1: Inicialización de la población
    poblacion = []
    for _ in range(tam_poblacion):
        poblacion.append(generar_individuo_aleatorio(len(partidos), len(slots_di

    mejor_audiencia_historica = -1
    mejor_calendario_historico = None
    historial_audiencias = [] # Para el método de parada

    print(f"Iniciando optimización para {len(partidos)} partidos...")

    # PASO 2: Bucle de Generaciones
    for gen in range(generaciones_max):

        # Evaluamos y ordenamos la población por la función fitness
        # Usamos una lista de tuplas (individuo, audiencia)
        puntuados = []
        for ind in poblacion:
            audiencia = fitness_final(ind, partidos, slots_disponibles)
            puntuados.append((ind, audiencia))

        # Ordenamos de mayor a menor audiencia
        puntuados.sort(key=lambda x: x[1], reverse=True)

        mejor_actual = puntuados[0][0]
        audiencia_actual = puntuados[0][1]

        # Guardar el mejor absoluto
        if audiencia_actual > mejor_audiencia_historica:
            mejor_audiencia_historica = audiencia_actual
            mejor_calendario_historico = copy.deepcopy(mejor_actual) # He visto

        historial_audiencias.append(audiencia_actual)

        # PASO 3: Verificación de Parada Temprana
        if gen >= paciencia:
            audiencia_hace_n = historial_audiencias[gen - paciencia]
            # Si la mejora es menor al 0.1% respecto a hace 'paciencia' generaci
            if audiencia_actual <= audiencia_hace_n * 1.001:
                print(f"Iteración {gen}: Parada temprana por falta de mejora sig
                break
```

```
        if gen % 10 == 0:
            print(f"Generación {gen}: Mejor audiencia = {audiencia_actual:.2f}")

        # PASO 4: Reproducimos una Nueva Generación
        nueva_poblacion = []

        # Pasamos los 2 mejores directamente para asegurar que ambas paseen ya q
        nueva_poblacion.append(puntuados[0][0])
        nueva_poblacion.append(puntuados[1][0])

        while len(nueva_poblacion) < tam_poblacion:
            # Selección por Torneo
            p1 = seleccionar_ganador_torneo(poblacion, partidos, slots_disponibl
            p2 = seleccionar_ganador_torneo(poblacion, partidos, slots_disponibl

            # Cruce
            hijo = cruzar_padres(p1, p2)

            # Mutaciones
            hijo = mutacion_aleatoria(hijo, len(slots_disponibles))
            hijo = mutacion_intercambio(hijo)

            nueva_poblacion.append(hijo)

        poblacion = nueva_poblacion

    return mejor_calendario_historico, mejor_audiencia_historica
```

In [29]:

```
# Ejecutamos el algoritmo
mejor_ind, mejor_aud = ejecutar_algoritmo_genetico(partidos, slots_disponibles)

# Traducimos los índices (números) al diccionario de la jornada para facilitar l
jornada_final = construir_jornada(mejor_ind, partidos, slots_disponibles)

print("\n" + "="*50)
print("     CALENDARIO DE LA JORNADA 19")
print("="*50)

for (dia, hora) in slots_disponibles:
    lista_partidos = jornada_final.get((dia, hora), [])

    # Solo mostramos los slots que tienen partidos asignados
    if lista_partidos:
        print(f"\n {dia.upper()} - {hora}")
        for p in lista_partidos:
            local, visitante = p
            # Calculamos la audiencia individual de este partido para dar más in
            aud_p = calcular_audiencia_partido(p, dia, hora, len(lista_partidos)
            print(f" {local} vs {visitante} |  Audiencia: {aud_p:.2f} M")

print("\n" + "="*50)
print(f"AUDIENCIA TOTAL ESTIMADA: {mejor_aud:.2f} MILLONES")
print("="*50)
```

```
Iniciando optimización para 10 partidos...
Generación 0: Mejor audiencia = 19.96
Generación 10: Mejor audiencia = 21.99
Generación 20: Mejor audiencia = 22.05
Generación 30: Mejor audiencia = 22.05
Iteración 32: Parada temprana por falta de mejora significativa.

================================================
     CALENDARIO DE LA JORNADA 19
================================================

 VIERNES - 20:00
 Levante UD vs RCD Espanyol de Barcelona |  Audiencia: 0.50 M

 SÁBADO - 16:00
 Valencia CF vs Elche CF |  Audiencia: 1.40 M

 SÁBADO - 18:00
 Getafe CF vs Real Sociedad de Fútbol |  Audiencia: 3.15 M

 SÁBADO - 20:00
 Athletic Club vs Real Madrid CF |  Audiencia: 3.60 M

 SÁBADO - 22:00
 FC Barcelona vs Club Atlético de Madrid |  Audiencia: 5.00 M

 DOMINGO - 16:00
 Sevilla FC vs RC Celta de Vigo |  Audiencia: 3.15 M

 DOMINGO - 18:00
 Rayo Vallecano de Madrid vs RCD Mallorca |  Audiencia: 1.40 M

 DOMINGO - 20:00
 Girona FC vs Club Atlético Osasuna |  Audiencia: 1.60 M

 DOMINGO - 22:00
 Villarreal CF vs Deportivo Alavés |  Audiencia: 1.40 M

 LUNES - 20:00
 Real Oviedo vs Real Betis Balompié |  Audiencia: 0.85 M

================================================
AUDIENCIA TOTAL ESTIMADA: 22.05 MILLONES
================================================
```

Justificación del uso del algoritmo genético frente a la fuerza bruta

El problema de asignar partidos a franjas horarias con el objetivo de maximizar la audiencia total es un problema de **optimización combinatoria discreta**, con un espacio de búsqueda de tamaño exponencial S^N, donde N es el número de partidos y S el número de slots disponibles. El algoritmo de fuerza bruta, al evaluar todas las combinaciones posibles, garantiza la obtención del máximo global, pero con una complejidad de $O(N \cdot S^N)$, que lo hace computacionalmente inviable para los valores propuestos en el problema, impidiendo su aplicación práctica.

Por otro lado , el algoritmo genético no explora el espacio de búsqueda completa, sino que lo muestrea mediante mecanismos basados en la evolución genética, con padres e hijos y mutaciones. Su complejidad es mucho menor $O(G \cdot P \cdot N)$, donde N vuelve a ser el número de partidos, G es el número de generaciones y P el tamaño de la población, permite un control del coste computacional al poder manipular los dos últimos parámetros explicados. Además, los operadores de selección, cruce y mutación facilitan la exploración del espacio de soluciones y aumentan las posibilidades de escapar de máximos locales.

Por lo que, aunque el algoritmo genético no garantiza el máximo global, proporciona soluciones de alta calidad en tiempos razonables, constituyendo una mejora práctica y eficiente frente al enfoque de fuerza bruta para problemas de optimización combinatoria como el presente.

Análisis de Complejidad del algoritmo genético

Para validar la eficiencia, desglosamos el coste de las funciones auxiliares que se ejecutan dentro del bucle principal.

Variables

- G: Generaciones (max 200).
- P: Población (max 1000).
- N: Número de Partidos (10).

1. Coste de Funciones Auxiliares

- `construir_jornada` : Recorre la lista de N partidos una vez en un bucle for. $\rightarrow O(N)$
- `calcular_audiencia_partido` : Son operaciones matemáticas directas sin bucles por lo que será siempre constante independientemente de la variación en el número de partidos o de los slots. $\rightarrow O(1)$
- `fitness_final` :
  - Construye jornada: $O(N)$.
  - Recorre todos los partidos agrupados para sumar audiencia: $O(N)$.
  - **Total Fitness:** $O(N) + O(N) = O(2N)$. que en Big o es $O(N)$

2. Coste por Generación

En cada generación ocurren estos pasos:

1. **Evaluación de la Población:**
   - Se ejecuta `fitness_final` para cada individuo (P).
   - Coste: $P \times O(N) = O(P \cdot N)$.

2. **Selección y Reproducción:**

- Para crear P hijos nuevos:
- **Torneo:** Selecciona $k = 5$ individuos dos veces. Si calculamos fitness dentro: $2 \times 5 \times O(N) \approx O(N)$.
- **Cruce:** Recorre el array de N elementos. $O(N)$.
- **Mutación:** Recorre el array de N elementos. $O(N)$.
- Coste total reproducción: $P \times O(N) = O(P \cdot N)$.

3. Complejidad Total del Algoritmo

Sumando todo y multiplicando por el número de generaciones (G):

$$\text{Total} = G \times (\underbrace{O(P \cdot N)}_{\text{Evaluar}} + \underbrace{O(P \cdot N)}_{\text{Reproducir}})$$

$$\text{Complejidad Big O} = O(G \cdot P \cdot N)$$

El algoritmo tiene la complejidad mencionada, siendo lineal respecto al número de generaciones, tamaño de la población y número de partidos.

Bibliografía

https://pmc.ncbi.nlm.nih.gov/articles/PMC10280284/#:~:text=Genotype%3A%20The%20com

https://www.bigocheatsheet.com/

https://www.geeksforgeeks.org/python/python-itertools-product/

https://es.wikipedia.org/wiki/Principio_de_inclusi%C3%B3n-exclusi%C3%B3n