Algoritmos de Optimización - Seminario

- **Nombres y Appellidos:** Gerardo Mauricio Gutiérrez Quintana
- **URL Github:** https://github.com/MauGutierrez/03MAIR---Algoritmos-de-Optimizacion---2021/tree/main/Seminario
- **URL Colab:** https://colab.research.google.com/drive/1PCJGt-fesQ_SY7kcH9T0RVmNIVrfh6LE?usp=sharing

Problema

1. Sesiones de Doblaje

Se precisa coordinar el doblaje de una película. Los actores del doblaje deben coincidir en las tomas en las que sus personajes aparecen juntos en las diferentes tomas. Los actores de doblaje cobran todos la misma cantidad por cada día que deben desplazarse hasta el estudio de grabación independientemente del número de tomas que se graben. No es posible grabar más de 6 tomas por día. El objetivo es planificar las sesiones por día de manera que el gasto por los servicios de los actores de doblaje sea el menor posible. Los datos son:

Número de actores: 10\ **Número de tomas:** 30\ **Actores/Tomas:** https://bit.ly/36D8IuK\

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

¿Cuantas posibilidades hay sin tener en cuenta las restricciones?

Sin tomar en cuenta las restricciones de este problema (no es posible grabar más de 6 tomas por día), podemos decir que la solución al problema sería grabar las 30 tomas en un solo día. De esta manera ya que todos los actores cobran la misma cantidad por día, al grabar todas sus tomas en un solo día el costo total de las sesiones de doblaje sería el total de actores involucrados.

Podríamos decir también que ya que todas las tomas serán grabadas en un solo día, el orden de estas sería irrelevante, pues los actores cobran por día de trabajo y no por toma. Por lo tanto, ya que independientemente del orden de tomas los actores grabaran en un solo día y las tomas no son repetidas, disponemos de posibles combinaciones, recordando que costo siempre será el mismo.

¿Cuantas posibilidades hay teniendo en cuenta las restricciones?

Ahora, tomando en cuenta la restricción de que no es posible grabar más de 6 tomas por día el problema cambia, ya que tendremos que dividir las 30 tomas en 5 sesiones para poder respetar la restricción. Tenemos que considerar que las tomas no pueden repetirse en diferentes sesiones, es decir, aunque el orden en que son elegidas no importa, una vez que hayamos asignado una toma en específico a una sesión, esta no podrá aparecer en otras sesiones.

Tomando en cuenta lo anterior, podemos decir que cada sesión tendrá un número m de tomas disponibles que podrán ser acomodadas de $m!$ maneras, y dado que para cada sesión no podrán haber tomas repetidas y el orden de estas es irrelevante, podemos modelar la cantidad de tomas para cada sesión como:

$$C_{mk} = \binom{m}{k} = \frac{m!}{k! \cdot (m-k)!}$$

donde la cantidad disponible para cada sesión será una combinatoria sin repetición de las tomas dispoinlbles. Para cada sesión tendremos un total m de tomas disponibles, donde $m = n - k$, n siendo el total de tomas diponibles por cada iteración y k el máximo de tomas disponibles por sesión (6).

Una vez que ya hemos obtenido el modelado de las posibilidades para una sesión en específico, podemos obtener el total de combinaciones posibles haciendo el productorio de las diferentes sesiones:

$$C_{mk} \approx 1.370874e^{18}$$

Por ultimo, tenemos que tomar en cuenta que existen diferentes maneras de ordenar las diferentes sesiones obtenidas. Concretamente se trata de un caso de permutación sin repetición. Por lo tanto, para obtener el resultado final del total de probabilidades tomando en cuenta las $n!$ posibles permutaciones sin repetición de las sesiones, habrá que dividir el resultado anteriormente obtenido entre las $n!$ posibles permutaciones sin repetición:

Tomando en cuenta la restricción de no poder grabar más de 6 tomas por día y que son 30 tomas en total, diremos que:

$$n = \frac{tomas}{restriccion}$$

obteniendo el resultado de final de:

Modelo para el espacio de soluciones

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

Ya que es un problema iterativo donde tendremos que iterar a través de todo el espacio de soluciones posibles, pienso que la estructura de datos más conveniente para este tipo de problemas es un array dinámico en donde guardemos el número de la toma y los actores involucrados en la toma:

$$Tomas = \begin{pmatrix} [0,\ actores] \\ [1,\ actores] \\ [2,\ actores] \\ \vdots \\ [n,\ actores] \end{pmatrix}$$

Intuitivamente podríamos decir que un diccionario sería la mejor estructura de datos por su rápido acceso, sin embargo tenemos que tomar en cuenta que dado que cada sesión debe de tener tomas no repetidas para calcular todas las posibilidades, un array dinámico nos trae más beneficios para un problema de esta magnitud.

Decimos que será un array dinámico ya que una vez que hayams obtenido el máximo número de tomas permitido para cada sesión, podemos eliminar estas tomas del array original para ya no considerarlas. He propuesto guardar el indíce original y las tomas en conjunto por que al ser un array dinámico, queremos guardar el número de la toma original independiemente de las tomas eliminadas.

Por último, ya que el orden de las tomas no importa, un array que guarde esta información nos será de mucha ayuda, ya que no buscamos una toma en particular y no es necesario que las tomas estén ordenadas.

Según el modelo para el espacio de soluciones

¿Cual es la función objetivo?

Como mencionamos anteriormente, si tomamos en cuenta la restricción de que solamente se pueden realizar 6 tomas por día, y además considerando que todos los actores cobran por día y no por toma, podemos decir que nuestra función objetivo es aquella en la que se sumen el total de actores involucrados por cada día de trabajo.

Diremos que nuestra función objetivo será:

$$f(s) = \sum_{i=0}^{m\ /\ n} len(\vec{Actores})$$

Dicho de otra manera, será la sumatoria desde $i = 1$ hasta m / n donde m es el número total de tomas y n es el máximo número de tomas por día, de la longitud del vector $actores$. Este vector será un array donde guardaremos todos los actores involucrados en la sesión de trabajo.

¿Es un problema de maximización o minimización?

Podemos decir que es un problema de minimización, ya que el objetivo de nuestro problema es minimizar la cantidad de actores participantes por cada día de trabajo para de esta manera, minimizar el costo total.

Diseña un algoritmo para resolver el problema por fuerza bruta

El algoritmo por fuerza bruta propuesto estaría basado en las $n!$ permutaciones posibles que hay de las n tomas. Lo que se busca es obtener las $n!$ posibles permutaciones de las tomas, y para cada n numero de tomas, calcular el precio para cada sesión según las restricciones del problema.

Lo que haremos será obtener las permutaciones posibles de las n tomas, iterar desde 0 hasta $n!$, dividir las $n!$ permutaciones en conjuntos iguales de n tomas, calcular el costo parcial para cada sesión desde 0 hasta m donde m es el máximo de tomas permitido por sesión y obtener el costo total.

Diremos que se ha encontrado una solución optima donde se minimice el costo total si una solucion parcial tiene un costo menor al actual costo total.

Con el objetivo de probar nuestro algoritmo por fuerza bruta, reduciremos el problema original de 30 tomas a 7, ya que aunque es un problema de menor tamaño, podemos demostrar que se encuentra una solución donde se minimice el costo.

In [16]:
```
import math
import itertools

tomas = [
    [1,1,1,1,1,0,0,0,0,0],
    [0,0,1,1,1,0,0,0,0,0],
    [0,1,0,0,1,0,1,0,0,0],
    [1,1,0,0,0,0,1,1,0,0],
    [0,1,0,1,0,0,0,1,0,0],
    [1,1,0,1,1,0,0,0,0,0],
```

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

# Restriccion del maximo de tomas por dia
maximo_tomas = 6

# Numero de actores participantes en la toma
numero_actores = len(tomas[0])

# Numero optimo de dias dependiendo de la restriccion y del numero de tomas
optimo = math.ceil(len(tomas) / maximo_tomas)

# Creamos un nuevo array donde guardaremos el indice original de la toma y la toma
tomas_nuevas = []
for i in range(len(tomas)):
    tomas_nuevas.append([i, tomas[i]])

# Obtenemos todas las permutaciones posibles de la lista de tomas
permutations_tomas = list(itertools.permutations(tomas_nuevas))


sesiones = dict()
solucion = dict()
costo_solucion = 9999
peor_costo = 0

# Inicializamos el diccionario donde guardaremos la solución optima
for i in range(optimo):
    sesiones[i] = 9999
    solucion[i] = 0

# Iteramos desde 0 hasta las n! permutaciones posibles de la lista de tomas
for i in range(len(permutations_tomas)):
    dia = 0
    sesion = 0
    actores = set()
    costo_parcial = 0
    tomas_id = []
    # Iteramos como tantas tomas haya para obtener una solución parcial
    for j in range(len(tomas)):
```

```
            tomas_id.append(permutations_tomas[i][j][0])
            # Para cada toma contamos los actores participantes en la toma
            for k in range(len(permutations_tomas[i][j][1])):
                if permutations_tomas[i][j][1][k] == 1:
                    # Agregamos los actores a un set para grabar valores unicos
                    actores.add(k)

            dia += 1
            # Si alcanzamos el maximo de tomas permitidas por dia
            # Asignamos para el día correspondiente de la sesión el valor parcial
            if dia == maximo_tomas or j == len(tomas)-1:
                sesiones[sesion] = tomas_id.copy()
                costo_parcial += len(actores)
                actores.clear()
                tomas_id.clear()
                sesion += 1
                dia = 0

        # Si el valor parcial es mayor al peor costo, actualizamos el peor costo
        if costo_parcial > peor_costo:
          peor_costo = costo_parcial

        # Si el costo parcial es menor al costo solución, actualizamos el costo solución
        if costo_parcial < costo_solucion:
            for i in range(len(sesiones)):
                solucion[i] = sesiones[i]

            costo_solucion = costo_parcial

print(f'Peor costo total: {peor_costo}')
print(f'Mejor costo total: {costo_solucion}')
print(solucion)
```

```
Peor costo total: 12
Mejor costo total: 10
{0: [0, 1, 2, 3, 5, 6], 1: [4]}
```

Podemos decir que con el problema propuesto y reduciendo la cantidad de tomas de 30 a 7, en el peor de los casos el costo total sería de 12 y en el mejor de los casos, el costo total sería de 10.

Calcula la complejidad del algoritmo por fuerza bruta

Podemos decir que la complejidad del algoritmo por fuerza bruta propuesto tiene una compejidad de $O(n!)$ ya que iteramos como tantas permutaciones posibles haya de nuestro problema inicial.

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

Al tratarse de un problema de combinatorias, nuestro objetivo será el de reducir este factor para obtener de una manera aproximada la planificación optima de las tomas para cada día de trabajo.

Ya que se trata de un problema de minimización de costos, en la planificación final se buscará agrupar la mayor cantidad posbile de tomas dadas las restricciones donde en cada día la variación entre artistas participantes sea la menor posible.

Tomando en cuenta que unos artistas participan en más tomas que otros, podemos ordenar las tomas de manera que daremos prioridad a las tomas donde aparezcan la mayor cantidad de artistas. Una vez ordenadas las tomas, tomaremos la primera toma con más artistas y a partir de esta comenzaremos a encontrar las tomas con más coincidencias, reduciendo en cada iteración las tomas disponibles. De esta manera, lograremos reducir el número de combinatorias necesarias.

Al agrupar las tomas con mayor cantidad de coincidencia de artistas, en cada iteración reduciremos los artistas participantes, dejando al final las tomas donde aparecen los artistas con menor cantidad de participaciones. Al reducir la variabilidad de artistas por días de trabajo y de mayor a menor participación, nos aseguramos minimizar costos evitando que todos los artistas aparezcan al menos una vez en todos los días de trabajo.

Podría decirse que el algoritmo propuesto es una especie de algoritmo voraz donde dependiendo del número de artistas y variaciones calculadas de artistas por toma, iremos agrupando tomas y reduciendo posibilidades.

Algoritmo propuesto

In [3]:
```
import math
import operator
import numpy as np


tomas = [
```

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


def get_tomas_parecidas(tomas_decimal, toma_objetivo, tomas):
    """
    Obtener las tomas parecidas para cada toma objetivo

    tomas_decimal:  array que contiene las tomas ordenadas por numero de actores y valor decimal
 de la toma.
                    [numero de actores en la toma, valor de la toma en decimal, indice en el arra
```

```
y tomas original]

    toma_objetivo:  toma con el mayor numero de actores que usaremos para comparar con las demas
 tomas
                    [numero de actores en la toma, valor de la toma en decimal, indice en el arra
y tomas original]

    tomas:  matriz que contiene todas las tomas originales representadas en una matriz de 1's y
 0's

    """

    # Diccionario para guardar las tomas por parecido ordenadas de mayor a menor
    tomas_por_parecido = {}
    parecidos = []

    # iteramos a traves de la toma objetivo como tantos actores tenga
    # Con esto determinaremos los niveles de coincidencia por toma objetivo
    # Supongamos el siguiente ejemplo de toma objetivo: 11110000
    # diremos que la toma objetivo podra tener niveles de coincidencia de 0, 1, 2, 3
    for i in range(toma_objetivo[0]):
        tomas_por_parecido[i] = []
        if i not in parecidos:
            parecidos.append(i)

    # Iteramos como tantas tomas haya en el array de tomas representadas en valor decimal
    for i in range(len(tomas_decimal)):
        # Hacemos la operacion logica OR entre la toma objetivo y cada toma del array
        # para determinar los todos actores involucrados entre la toma objetivo y una toma
        temp = tomas[toma_objetivo[2]] | tomas[tomas_decimal[i][2]]
        # para determinar el nivel de coincidencia hacemos una resta de los actores involucrados
        # entre la toma objetivo y cada una de las otras tomas
        parecido = abs(len(tomas[toma_objetivo[2]][tomas[toma_objetivo[2]] == 1]) - len(temp[temp
== 1]))
        # guardamos la toma en el diccionario dependiendo de su nivel de coincidencia con la toma
objetivo
        tomas_por_parecido[parecido].append([tomas_decimal[i][2], i])

    return tomas_por_parecido, parecidos
```

```python
def get_tomas_planificadas(tomas):
    """
    Obtener la planificacion de las tomas para optimizar el costo total
    """

    tomas_decimal = []
    tomas = np.array(tomas)

    # Convertimos las tomas a su valor decimal, y la ordenamos de mayor a menor.
    # Regresamos un array con la siguiente estructura:
    # [cantidad de 1's en la toma, toma convertida de binario a decimal, indice original de la to
ma]
    for i in range(len(tomas)):
        my_string = ''.join(map(str, tomas[i]))
        tomas_decimal.append([len(tomas[i][tomas[i] == 1]), int(my_string, 2), i])

    tomas_decimal = sorted(tomas_decimal, key=operator.itemgetter(0, 1), reverse=True)


    solucion_final = {}
    maximo_tomas = 6
    index_delete = []
    optimo = math.ceil(len(tomas) / maximo_tomas)
    costo_total = 0
    dia = 0

    # Inicializar el diccionario que tendra la solucion final
    # El diccionario tendra como llave el optimo numero de dias para cubrir todas las tomas
    for i in range(optimo):
        solucion_final[i] = []

    # Iteramos mientras haya tomas en el array secundario tomas decimal
    while len(tomas_decimal) > 0:
        index_delete = []

        # La toma solucion sera la toma con el mayor numero de actores por cada iteracion
        toma_solucion = tomas_decimal[0]
        # Obtenemos las tomas que son parecidas para cada toma solucion en forma de un diccionari
o
        # Este diccionario vendra ordenado de mayor coincidencia a menor coincidencia
```

```
        solucion_parcial, llaves = get_tomas_parecidas(tomas_decimal, toma_solucion, tomas)
        index = 0
        actores = set()
        # Iteramos hasta el maxim de tomas permitido por dia
        for i in range(maximo_tomas):
            while (index in llaves) and (len(solucion_parcial[index]) == 0):
                index += 1

            # Sacamos cada toma de mayor a menor coincidencia
            # En caso de que ya no se puedan obtener más tomas y hayamos llegado al final del dic
cionario de coincidencias,
            # quiere decir que ya no hay más tomas y terminamos este ciclo
            try:
                j = solucion_parcial[index].pop(0)
            except:
                break

            # Guardamos los indices que seran eliminados del array que contiene las tomas convert
idas a valor decimal
            index_delete.append(j[1])
            # Guardamos cada toma en el diccionario que contiene la solucion final
            solucion_final[dia].append(j[0])

            # Iteramos a traves de cada toma para obtener los actores involucrados por cada toma
            for i in range(len(tomas[j[0]])):
                if tomas[j[0]][i] == 1:
                    actores.add(i+1)


        # Eliminamos las tomas que ya han sido utilizadas para la solucion parcial pues ya no ser
a consideradas
        tomas_decimal = np.delete(tomas_decimal, tuple(index_delete), axis=0)
        # Calculamos el costo total contando el numero de actores por cada sesion
        costo_total += len(actores)
        print(f'Tomas para la sesión del día {dia+1}: {solucion_final[dia]}')
        print(f'Costo del día {dia+1}: {len(actores)}\n')
        dia += 1

    print(f'El costo total sera: {costo_total}')
```

```
get_tomas_planificadas(tomas)
```

```
Tomas para la sesión del día 1: [0, 21, 5, 6, 19, 8]
Costo del día 1: 5

Tomas para la sesión del día 2: [11, 7, 13, 16, 18, 22]
Costo del día 2: 5

Tomas para la sesión del día 3: [10, 3, 25, 14, 12, 28]
Costo del día 3: 9

Tomas para la sesión del día 4: [24, 27, 29, 15, 4, 26]
Costo del día 4: 6

Tomas para la sesión del día 5: [9, 17, 23, 20, 2, 1]
Costo del día 5: 9

El costo total sera: 34
```

Calcula la complejidad del algoritmo

Función get_tomas_planificadas():

Primero recibimos el array de tomas e iteramos a traves de todo el array para tomar las tomas, convertirlas a un valor decimal y guardarlas en otro array. Esta es una operación que solamente la realizamos una vez y que tiene una complejidad de $O(n)$ donde n es el tamaño del array.

Una vez que hemos terminado de iterar a través de todo el array y obtenemos el array secundario, haremos un sort de este array para ordenar las tomas de mayor número de actores a menor número de actores participantes. La función sort de Python tiene una complejidad de $O(n \cdot Log \cdot n)$ tanto para el peor caso como para cualquier otro caso.

Después de ordenar las tomas e inicializar la estructura de datos que guardara la solución final, haremos un ciclo While donde iteraremos como tantas tomas queden disponibles en el array donde guardamos las tomas ordenadas. Ya que este es un array dinámico que irá reduciendo su tamaño conforme vamos llenando sesiones, podemos decir que este ciclo tendrá una complejidad de $O(m / n)$ donde m es el número de tomas totales y n es el máximo número de tomas disponibles por día. Sin embargo, ya que dentro del ciclo while iteramos desde 0 hasta n donde n es el máximo número de tomas disponibles por día, y despreciando ciclos internos de menor complejidad, podemos decir que esta función tendrá una complejidad de $O(m)$ lineal por lo siguiente:

- complejidad ciclo while:

$$O(m / n)$$

- complejidad del ciclo interno while:

$$O(n)$$

- Multiplicación de ambas complejidades:

$$O(m / n) \cdot O(n) = O(m)$$

donde m es el tamaño del array tomas, o el número de tomas total.

Función get_tomas_parecidas():

- En la función get_tomas_parecidas tenemos dos ciclos for, sin embargo, ambos tienen una complejidad constante de $O(n)$ menor a la complejidad de la función la función get_tomas_planificadas, por lo tanto, estas complejidades son despreciables.

Podemos concluir que nuestro algoritmo tendrá una complejidad de $O(n \cdot Log \cdot n)$ donde n será la cantidad de tomas total. Esto debido a que utilizamos un sort para ordenar el array de tomas inicial y a partir un array ordenado, resolvemos el problema.

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

Para poner a prueba nuestro algoritmo, haremos una matriz aleatoria que contenga valores entre 0 y 1 usando la librería de numpy.

In [4]:
```
import numpy as np


def generar_random(cota_1, cota_2= None):
  """
  Funcion para generar numeros aleatorios
  """
  numero_aleatorio = 0
  if cota_2:
    numero_aleatorio = np.random.randint(cota_1, cota_2)
  else:
```

```
    numero_aleatorio = np.random.randint(cota_1)

  return numero_aleatorio

def generar_tomas_aleatorias():
  """
  Funcion para generar una matriz de datos aleatorios
  """
  # Variable para obtener un numero random de actores en un rango entre 10 y 20
  actores_random = generar_random(10, 20)

  # Variable para obtener un numero random de tomas en un rango entre 6 y 30
  numero_tomas = generar_random(6, 30)

  # Array para guardar las tomas finales
  tomas_random = []

  # Iteramos como tantas tomas haya
  for i in range(numero_tomas):
    # Array para guardar los actores por cada toma
    actores_por_toma = []
    # Generamos aleatoriamente los actores para cada toma
    for i in range(actores_random):
      actores_por_toma.append(generar_random(2))

    # Para agregar aleatoridad, hacemos un shuffle al array de la toma
    np.random.shuffle(actores_por_toma)
    # Agregamos la toma al array tomas
    tomas_random.append(actores_por_toma)

  return tomas_random
```

Aplica el algoritmo al juego de datos generado

In [11]:
```
tomas_aleatorias = generar_tomas_aleatorias()
print(f'Actores participantes: {len(tomas_aleatorias[0])}')
print(f'Numero de tomas: {len(tomas_aleatorias)}\n')
```

```
# Aplicar el algoritmo al juego de datos aleatorio
get_tomas_planificadas(tomas_aleatorias)
```

```
Actores participantes: 13
Numero de tomas: 7

Tomas para la sesión del día 1: [6, 1, 2, 0, 4, 5]
Costo del día 1: 13

Tomas para la sesión del día 2: [3]
Costo del día 2: 7

El costo total sera: 20
```

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

- **Fractional Knapsack problem** (https://afteracademy.com/blog/fractional-knapsack-problem)
- **Fractional Knapsack Problem: Greedy algorithm with Example** (Fhttps://www.guru99.com/fractional-knapsack-problem-greedy.html)

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

Desde mi punto de vista, este tipo de problemas en donde es necesaria la optimización, podrían ser candidatos a ser resueltos usando Algoritmos Genéticos. La idea general de como podría ser abordado este problema, sería la de calcular la primera mejor solución que optimice el costo parcial. Una vez que ya hayamos obtenido la primera solución, podemos utilizarla para calcular las siguientes soluciones iterativamente eligiendo siempre la más optima a través de una función objetivo.

La primera mejor solución serían los padres y en este sentido, utilizariamos las mejores tomas para criar nuevas agrupaciones. El proceso terminaría cuando hayamos encontrado una solución en la que el costo sea el mínimo o en otro caso, hasta que hayamos encontrado un número fijo de generaciones.