Grupo 21

PRÁCTICA 3

Ejercicio 1

Si encuentra una solución pero **no las mejores** por que no es exhaustivo. Si es eficiente puesto que evita la exploración de una parte del espacio de estados.

¿Ha encontrado el algoritmo el óptimo global?¿Ha encontrado la misma solución en distintas ejecuciones? No, no encuentra la misma solución para distintas ejecuciones por lo tanto no puede encontrar un óptimo global, se puede dar el caso de que lo encuentre una vez pero como cada vez da una solución distinta el algoritmo **no** es óptimo.

Este algoritmo es similar al algoritmo A* con la diferencia de que A* nunca se va a quedar bloqueado si no encuentra una solución mejor(óptimo local). También se parece al algoritmo de profundidad limitada.

No explora todo el espacio de estados, como máximo solo encuentra una solución en cambio los algoritmos de búsqueda exhaustiva si lo hace, por ejemplo *Búsqueda en anchura* tiene que recorrer todo. Al no ser exhaustivo reduce el número de nodos para analizar.

Ejercicio 2

Ejecutamos ambos algoritmos para las siguientes rejillas y obteniendo los siguientes resultados: 1.grid1 = [[0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 11.2, 0.70, 1.40], [2.20, 1.80, 0.70, 0.00, 0.00, 9.00, 0.00, 0.00, 0.00]] 11.2 2.2 2.grid2 = [[2.20, 1.80, 0.70, 0.00, 0.00, 9.00, 0.00, 0.00, 0.00], [2.20, 1.80, 4.70, 6.50, 4.30, 1.80, 0.70, 0.00, 0.00], [0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.70, 1.40]] 9.0 2.2 3.grid3 = [[2.20, 1.80, 4.70, 8.50, 4.30, 1.80, 0.70, 0.00, 0.00]] 8.5 2.2 4.grid4 = [0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.70, 1.40] 1.4 0.0
Como podemos observar el algoritmo de enfriamiento simulado encuentra el punto máximo con más éxito y frecuencia. Por otro lado, vemos cómo repetidas veces el algoritmo de máxima pendiente se queda en un óptimo local como es el 2.2.

Ejercicio 3

Función población inicial:

```
def poblacion_inicial(problema_genetico, size):
    individuos = [0, 1]
    poblacion = list()
    l = []
    for i in range(size):
        l = []
        for j in range(problema_genetico.longitud_individuos):
            l.append(random.choice(individuos))
        poblacion.append(l)

    return poblacion
```

```
In [331]: poblacion_inicial(cuadrados,10)

Out[331]: [[0, 0, 0, 1, 0, 1, 0, 0, 0, 0],
           [0, 1, 0, 0, 1, 1, 1, 1, 1, 0],
           [1, 1, 1, 1, 0, 1, 0, 1, 1, 0],
           [1, 1, 0, 0, 1, 0, 0, 0, 0, 1],
           [0, 0, 0, 1, 0, 1, 0, 0, 1, 1],
           [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
           [1, 0, 0, 1, 0, 1, 0, 0, 1, 0],
           [0, 1, 0, 0, 1, 1, 0, 0, 0, 0],
           [0, 0, 1, 1, 1, 1, 0, 0, 1, 1],
           [1, 1, 1, 0, 0, 0, 1, 1, 0, 1]]
```

Esta función nos devuelve una población inicial. Tenemos dos bucles for el primero se encarga de tener el número adecuado de individuos por población y el segundo se encarga de generar n genes(longitud de los individuos) de forma aleatoria para crear un cromosoma. Se usa la función random para elegir cada gen. Cómo se puede ver en la imagen de la derecha se ha generado una población de 10 individuos con diez genes cada uno y ha elegido para cada posición un de forma aleatoria entre el 1 y el 0.

Función de cruce:

```
def cruza_padres(problema_genetico,padres):
    l = []

    for i in range(len(padres)):
        hijos = []
        if i == len(padres)-1 :
            hijos = problema_genetico.cruza(padres[i], padres[0])

        else:
            hijos = problema_genetico.cruza(padres[i], padres[i+1])
        if problema_genetico.fitness(hijos[0]) >  problema_genetico.fitness(hijos[1]):
            l.append(hijos[0])
        else:
            l.append(hijos[1])

    return l
```

A la hora de cruzar , para que nos salga el mismo número de individuos en la siguiente generación, lo que hacemos es cruzar el primero con el segundo, el segundo con el tercero, el tercero con el cuarto y así sucesivamente hasta que

llegamos al último que lo cruzamos con el primero. De esta forma conseguimos que todos los individuos se crucen el mismo número de veces y que el número de individuos no varíe evitando tener que copiar individuos de la generación anterior.

```
In [333]: p1 = [[1, 1, 0, 1, 0, 1, 0, 0, 0, 1],
                [0, 1, 0, 1, 0, 0, 1, 0, 1, 1],
                [0, 0, 1, 0, 0, 0, 1, 1, 1, 0],
                [0, 0, 1, 1, 1, 1, 1, 1, 1, 0],
                [0, 1, 1, 0, 0, 0, 0, 0, 0, 0],
                [1, 0, 1, 1, 1, 0, 1, 1, 0, 1]]

          cruza_padres(cuadrados,p1)
          # Posible salida
          # [[1, 1, 0, 1, 0, 0, 1, 0, 1, 1],
          #  [0, 1, 0, 1, 0, 1, 0, 0, 0, 1],
          #  [0, 0, 1, 1, 1, 1, 1, 1, 1, 0],
          #  [0, 0, 1, 0, 0, 0, 1, 1, 1, 0],
          #  [0, 1, 1, 1, 1, 0, 1, 1, 0, 1],
          #  [1, 0, 1, 0, 0, 0, 0, 0, 0, 0]]

Out[333]: [[1, 1, 0, 1, 0, 0, 1, 0, 1, 1],
           [0, 0, 1, 0, 0, 0, 1, 0, 1, 1],
           [0, 0, 1, 0, 0, 1, 1, 1, 1, 0],
           [0, 1, 1, 0, 0, 1, 1, 1, 1, 0],
           [0, 1, 1, 0, 0, 0, 1, 1, 0, 1],
           [1, 1, 0, 1, 0, 0, 1, 1, 0, 1]]
```

Función mutación:

```
def muta_individuos(problema_genetico, poblacion, prob):
    l = list()
    for i in range (len(poblacion)):
        l.append(problema_genetico.muta(poblacion[i], prob))
    return l
    # hay que llamar a  problema_genetico.muta(x,prob) para todos los individuos de la poblacion.
```

Llama a la función muta de problema genético en cada individuo de la población con la misma probabilidad de que mute en todos los individuos.

Ejemplos de salidas:

```
In [335]: muta_individuos(cuadrados,p1,0.5)
          # Posible salida:
          #  [[1, 1, 0, 1, 0, 1, 0, 0, 0, 1],
          #   [0, 1, 0, 1, 0, 0, 1, 0, 0, 1],
          #   [0, 0, 1, 0, 0, 0, 1, 0, 1, 0],
          #   [0, 0, 1, 1, 1, 1, 1, 1, 1, 0],
          #   [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
          #   [1, 0, 1, 1, 1, 0, 1, 1, 0, 1]]

Out[335]: [[1, 1, 0, 1, 1, 1, 0, 0, 0, 1],
           [0, 1, 0, 1, 0, 0, 1, 0, 1, 1],
           [0, 0, 0, 0, 0, 0, 1, 1, 1, 0],
           [0, 0, 1, 1, 1, 1, 1, 0, 1, 0],
           [0, 1, 1, 0, 0, 0, 0, 0, 0, 0],
           [1, 0, 1, 1, 1, 0, 1, 1, 0, 1]]

In [336]: p1 = [[1, 1, 0, 1, 0, 1, 0, 0, 0, 1],
                [0, 1, 0, 1, 0, 0, 1, 0, 1, 1],
                [0, 0, 1, 0, 0, 0, 1, 1, 1, 0],
                [0, 0, 1, 1, 1, 1, 1, 1, 1, 0],
                [0, 1, 1, 0, 0, 0, 0, 0, 0, 0],
                [1, 0, 1, 1, 1, 0, 1, 1, 0, 1]]

In [337]: muta_individuos(cuadrados,p1,0.5)

Out[337]: [[1, 1, 0, 1, 0, 1, 0, 0, 0, 0],
           [0, 1, 0, 1, 0, 0, 1, 0, 1, 0],
           [0, 0, 1, 0, 0, 1, 1, 1, 1, 0],
           [0, 0, 1, 1, 1, 1, 1, 1, 1, 0],
           [0, 1, 1, 0, 0, 1, 0, 0, 0, 0],
           [1, 0, 1, 1, 1, 0, 1, 1, 0, 1]]
```

Ejercicio 5

Resultado obtenido tras la ejecución:

```
In [343]: algoritmo_genetico(cuadrados,3,min,20,10,0.7,0.1)
          # Salida esperada: (0, 0)
Out[343]: (128, 16384)

In [344]: algoritmo_genetico(cuadrados,3,max,20,10,0.7,0.1)
          # Salida esperada: (1023, 1046529)
Out[344]: (1015, 1030225)

In [345]: algoritmo_genetico(cuadrados,3,min,20,10,0.7,0.7)
Out[345]: (7, 49)
```

Valoración:

Primera ejecución: (16,256) Segunda ejecución: (895, 801025) Tercera ejecución:(8, 64). Al no tener una alta probabilidad de mutación en las dos primeras ejecuciones observamos que los individuos resultantes en cada generación son muy similares, en la mayoría de los casos iguales y es muy probable que un individuo con un buen fitness pase de generación en generación como es el caso en ambas ejecuciones: el 16 comienza en la generación 13 y el 895 en la generación 10. Añadido a la baja probabilidad de mutación favorece que un individuo con un buen fitness en un estado temprano/medio avance hasta la población final y sea elegido como ganador del torneo, como los individuos son iguales el cruce (si se produce) devuelve a los mismos y no hay variabilidad en la población. Aún así al depender de factores aleatorios podemos conseguir la solución esperada o incluso llegar a mejorarla.

```
In [54]: algoritmo_genetico(cuadrados,3,min,20,10,0.7,0.1)
         # Salida esperada: (0, 0)
Out[54]: (0, 0)

In [55]: algoritmo_genetico(cuadrados,3,max,20,10,0.7,0.1)
         # Salida esperada: (1023, 1046529)
Out[55]: (1023, 1046529)
```

Por otro lado, en la tercera ejecución hemos variado la probabilidad de mutación a 0.7 consiguiendo así una mayor diversidad de individuos en cada generación. Resulta llamativo cómo el ganador del torneo (8) aparece en la última generación. Esto es porque en cada generación pueden aparecer individuos mejores que los ya existentes con mayor frecuencia, sustituyendolos así e independientemente del

resultado del cruce.

```
In [58]: algoritmo_genetico(cuadrados,3,min,20,10,0.7,0.7)
Out[58]: (2, 4)
```

Demostramos que, a mayor probabilidad de mutación, mayor será la diversidad de la población durante las distintas generaciones dando así más viabilidad a resultados con mejor fitness.

Ejercicio 6

```
def decodifica(cromosoma, n, pesos, capacidad):
    peso_en_mochila = 0
    l = []
    for i in range(n):
        if cromosoma[i] == 1 and peso_en_mochila + pesos[i] <= capacidad:
            l.append(1)
            peso_en_mochila += pesos[i]
        elif cromosoma[i]== 0 or peso_en_mochila + pesos[i] > capacidad:
            l.append(0)
    return l
```

La función decodifica recorre posición a posición el cromosoma comprobando tanto si el gen está a uno y si no se supera la capacidad máxima con el propio peso del gen actual y lo que teníamos en la mochila. Si alguna de estas condiciones se incumple ese gen lo cambia a cero lo que quiere decir que ese objeto no se añadiría a la mochila. Y si no se incumple se añade el elemento a la mochila.

```
def fitness_mochila(cromosoma, n_objetos, pesos, capacidad, valores):
    l = list()
    l = decodifica(cromosoma, n_objetos, pesos, capacidad)
    valor = 0
    for i in range(len(l)):
        if l[i] == 1:
            valor = valor + valores[i]

    return valor
```

La función **fitness_mochila** lo primero que hace es decodificar el cromosoma, luego suma el valor[i] de la lista de valores pasado como parámetro y que le corresponde a cada posición de la lista si el valor de dicha posición cuando hemos decodificado el cromosoma se encuentra a 1, si está a 0 no lo suma.

```
In [350]: fitness_mochila([1,1,1,1], 4, [2,3,4,5], 4, [7,1,4,5])
Out[350]: 7
```

Problema mochila 1:

In [351]:
```
# Problema de la mochila 1:
# 10 objetos, peso máximo 165
pesos1 = [23,31,29,44,53,38,63,85,89,82]
valores1 = [92,57,49,68,60,43,67,84,87,72]

print(decodifica([1,1,1,1,1,1,1,1,1,1], 10, pesos1, 165))
print(fitness_mochila([1,1,1,1,1,1,1,1,1,1], 10, pesos1, 165, valores1))

# Solución óptima= [1,1,1,1,0,1,0,0,0,0], con valor 309
```

```
[1, 1, 1, 1, 0, 1, 0, 0, 0, 0]
309
```

Problema mochila 2:

In [352]:
```
# Problema de la mochila 2:
# 15 objetos, peso máximo 750

pesos2 = [70,73,77,80,82,87,90,94,98,106,110,113,115,118,120]
valores2 = [135,139,149,150,156,163,173,184,192,201,210,214,221,229,240]

print(decodifica([1,1,1,1,1,1,1,1,1,1,1,1,1,1,1], 15, pesos2, 750))
print(fitness_mochila([1,1,1,1,1,1,1,1,1,1,1,1,1,1,1], 15, pesos2, 750, valores2))

# Solución óptima= [1,0,1,0,1,0,1,1,1,0,0,0,0,1,1] con valor 1458
```

```
[1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0]
1249
```

Problema mochila 3:

In [353]:
```
# Problema de la mochila 3:
# 24 objetos, peso máximo 6404180
pesos3 = [382745,799601,909247,729069,467902, 44328,
       34610,698150,823460,903959,853665,551830,610856,
       670702,488960,951111,323046,446298,931161, 31385,496951,264724,224916,169684]
valores3 = [825594,1677009,1676628,1523970, 943972,  97426,
       69666,1296457,1679693,1902996,
       1844992,1049289,1252836,1319836, 953277,2067538, 675367,
       853655,1826027, 65731, 901489, 577243, 466257, 369261]

print(decodifica([1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1], 24, pesos3, 6404180))
print(fitness_mochila([1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1], 24, pesos3, 6404180, valores3))

# Solución óptima= [1,1,0,1,1,1,0,0,0,1,1,0,1,0,0,1,0,0,0,0,0,1,1,1] con valoración 13549094
```

```
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
12808431
```

Podemos observar con estas tres instancias del problema, no se obtienen los mismos resultados de fitness que en la **solución óptima** por que no consigue llegar todas las veces a la **solución óptima** en la función decodifica consiguiendo así un fitness más bajo que la **solución óptima**.

Ejercicio 7

Según aumentamos el número de generaciones, individuos de la población y genes que procesar la eficiencia se reduce notablemente, de unos segundos puede llegar a tardar hasta varios minutos. Como el problema se resuelve con selección por torneo que incluye un factor aleatorio el resultado que nos genera puede ser distinto cada vez que se ejecuta el algoritmo que tiene como consecuencia que a veces da como resultado una solución peor, igual o mejor de la propuesta. Aunque la probabilidad de mutación es muy baja también puede influir en que salgan diferentes individuos en cada ejecución.

```
# >>> algoritmo_genetico_t(m1g,3,max,100,50,0.8,0.05)
# ([1, 1, 1, 1, 0, 1, 0, 0, 0, 0], 309)

algoritmo_genetico(m1g,3,max,100,50,0.8,0.05)
```

```
([1, 1, 0, 1, 0, 0, 1, 0, 0, 0], 284)
```

En este caso nos da una solución peor de la que se nos propone. Ya que tiene un fitness más bajo.

```
# >>> algoritmo_genetico_t(m1g,3,max,100,50,0.8,0.05)
# ([1, 1, 1, 1, 0, 1, 0, 0, 0, 0], 309)

algoritmo_genetico(m1g,3,max,100,50,0.8,0.05)
```

```
([1, 1, 1, 1, 0, 1, 0, 0, 0, 0], 309)
```

Como se puede ver en los siguientes ejemplos nuestra solución ha mejorado.

```
# >>> algoritmo_genetico_t(m3g,3,max,2000,100,0.75,0.1)
# ([0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0], 13366296)

algoritmo_genetico(m3g,3,max,2000,100,0.75,0.1)
```

```
([0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1],
 13471217)
```

```
# >>> algoritmo_genetico_t(m3g,3,max,1000,100,0.75,0.1)
# ([1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0], 13412953)

algoritmo_genetico(m3g,3,max,1000,100,0.75,0.1)
```

```
([1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0],
 13518963)
```

2. Resolución problema de las tareas

Representación

En este problema hemos decidido representar los individuos en decimal indicando que tareas tiene asignadas, como en el siguiente ejemplo: [0, -1, 0, -1, 0, 4, 0, 0]

Cada índice i de la lista indica el número de la tarea y este número es, a la vez, su cualificación; y el valor:

- -1 significa que el individuo no tiene asignada la tarea i
- 0 significa que tiene asignada la tarea i pero esta no tiene dependencia con otra tarea
- cualquier otro número indica la tarea de la cual depende la tarea i

La longitud del cromosoma depende del número de tareas que hay.
A la clase Problema Genético le hemos añadido un parámetro, que es una tupla que representa la tupla de las tareas: (0,0,0,2,0,4,0,0) en nuestro ejemplo. En esta tupla se indican las dependencias que hay entre las tareas. Recordemos que aquellas a 0 no dependen de ninguna otra tarea y que **el índice de la tupla de las tareas se considera que comienza en 1**, por una razón que veremos posteriormente.

Métodos

La **función** de **cruzar** bla bla bla bla
//imagen por aquí
//descripción por allá

La **función** de **mutar** la hemos adaptado a nuestra representación: si tenemos que mutar entonces comprobamos que si en la posición elegida del cromosoma hay un -1 (que indica que no tiene esa tarea asignada) le asignamos una tarea y si tiene una tarea asignada ponemos esa posición a -1.

```
def fun_mutar(cromosoma, prob, tareas):
    """Elige un elemento al azar del cromosoma y lo modifica con una probabilidad igual
    l = len(cromosoma)
    p = random.randint(0,l-1)
    if prob > random.uniform(0,1):
        if(cromosoma[p] != -1):
            cromosoma[p] = -1
        else:
            cromosoma[p] = tareas[p]
    return cromosoma
```

La **función** de **fitness**: de cara a optimizar la solución del problema, buscamos la paralelización de la ejecución de tantas tareas como sea posible.Con este fin hay dos casos diferenciables dada la naturaleza de las tareas:

- Ejecución de tareas independientes
- Ejecución de tareas dependientes

Para traducir esta distribución de tareas en la valoración de un cromosoma puntuaremos positivamente, en primer lugar, la asignación de tareas a un cromosoma. Como buscamos la ejecución de tareas en paralelo (para minimizar el tiempo total) si algún cromosoma tiene una tarea dependiente de otra favorecemos que tenga aquellas de las que depende. De igual modo, aquellos con tareas independientes se verán penalizados si cuentan con otras tareas dependientes.

```
def fun_fitness(cromosoma, tareas): #redefinido
    """Función de valoración de un trabajador con sus tareas asigandas"""
    n = 0
    k = len(cromosoma) - 1
    dependencia = 0
    while k >= 0:
        if(cromosoma[k] != -1):                    #si tiene dependencias
            n += 2
            dependencia = tareas[k-1]
            if(dependencia > 0):
                if(cromosoma[dependencia] != -1):
                    n += 3
                else:
                    n -=1
            else:                                   #si no tiene dependencias
                if(k == 1):
                    n += 3
                else:
                    if(cromosoma[k-1] != -1):
                        n -= 1
                    else:
                        n += 5
        k -= 1
    return n
```

Población inicial: elegimos dos representaciones para generar la población inicial. La primera versión genera los individuos en base a un número aleatorio. Dependiendo del valor de este número va formando los individuos, ya sea poniéndoles un ‘-1’ o bien una tarea (indicando la dependencia si la hay).

```python
def poblacion_inicial22(problema_genetico, size): #size es el número de trabajadores en
    individuos = [0, 1]
    poblacion = list()
    l = []
    for i in range(size):
        l = []
        for j in range(problema_genetico.longitud_individuos):
            v = random.choice(individuos)
            if(v):
                l.append(problema_genetico.tareas[j])
            else:
                l.append(-1)
        poblacion.append(l)
    return poblacion
```

Hemos decidido mantener la selección por torneo donde tenemos n individuos y vamos a elegir a k participantes y la opción opt que indica si es la función max o min.

Algoritmo genético

Utilizamos las funciones provistas en el enunciado para realizar la selección por torneo, las nuevas generaciones y, por último, la población final. Para poder adaptarlo al problema generamos, como hemos explicado anteriormente, una población inicial de posibles asignaciones para un trabajador cualquiera. Efectuamos los cruces y las mutaciones en cada generación y elegimos los candidatos más válidos según el sistema de valoración (máx.). Para proporcionar una mayor variedad en la población final utilizamos una probabilidad de mutación ligeramente elevada. El algoritmo recibe, como parámetro adicional, una tupla con las cualificaciones de cada trabajador (el índice indica el trabajador y el valor su cualificación) para poder devolver, con la última población, el cromosoma que se le asigna a cada trabajador. Cada generación cuenta con una población con un número de individuos mayor o igual al de los trabajadores del problema.

```python
def algoritmo_genetico(problema_genetico,k,opt,ngen,size,prop_cruces,prob_mutar, cualificaciones):
    poblacion= poblacion_inicial22(problema_genetico,size)
    n_padres=round(size*prop_cruces)
    n_padres= int (n_padres if n_padres%2==0 else n_padres-1)
    n_directos = size-n_padres
    for _ in range(ngen):
        poblacion= nueva_generacion(problema_genetico,k,opt,poblacion,n_padres, n_directos,prob_mutar)

    mejor_cr= opt(poblacion, key=problema_genetico.fitness)
    mejor=problema_genetico.decodifica(mejor_cr)
    print("poblacion")
    for i in range(len(poblacion)):
        print(poblacion[i])
    return asignar_trabajadores(problema_genetico, poblacion, cualificaciones, opt)
```

Asignar trabajadores

Una vez contamos con la población final, el siguiente paso es asociar un cromosoma a cada trabajador. Dada la restricción de la cualificación, buscamos

emparejar primero los trabajadores de alta cualificación con aquellos cromosomas que tengan la tarea de dicha cualificación asignada. Para ello seleccionamos todos los cromosomas con esa tarea y de ahí nos quedamos con el que tenga mejor fitness. Esto se repite para cada trabajador hasta que todos tienen asociado un cromosoma. Si un trabajador no encuentra cromosomas con tarea para su cualificación asignada, se le otorgará un cromosoma sin ninguna tarea asignada.

```
def asignar_trabajadores(problema_genetico, poblacion, cualificaciones, opt):
    """Para cada trabajador y dada la poblacion final asignarle un cromosoma con posibles tareas a realizar"""
    i = len(cualificaciones) - 1
    trabajadores = list()
    while i >= 0:
        subl = list()
        for j in poblacion:
            if(j[cualificaciones[i]] != -1):
                subl.append(j)
        if(subl):
            mejor_cr= opt(subl, key=problema_genetico.fitness)
            trabajadores.insert(0, [cualificaciones[i]]+mejor_cr)
            poblacion.remove(mejor_cr)
        else:
            empt = [-1 for i in range(problema_genetico.longitud_individuos)]
            trabajadores.insert(0, [cualificaciones[i]] + empt)
        i -= 1
    print("trabajadores")
    for i in range(len(trabajadores)):
        print(trabajadores[i])
    return trabajadores
```

Asignar tareas

El último paso consiste en determinar qué trabajador realizará cada tarea. Para ello tenemos en cuenta que cada trabajador solo puede realizar aquellas tareas que se encuentren en el rango de su cualificación. La asignación de tareas se hace de la siguiente manera:

- Búsqueda de un trabajador con la tarea disponible y ninguna otra asignada en la solución (favorecemos la paralelización).
- Si no encontramos un trabajador con estas condiciones, le otorgaremos la tarea al trabajador con menos tareas asignadas hasta el momento y dicha tarea disponible.

```
#(0,0,0,2,0,4,0,0)
def asignar_tareas(poblacion, tareas):
    """Extraer la secuencia en la que se ejecutan las tareas teniendo en cuenta las restricciones de cualificacion"""
    asignadas = [-1 for _ in range(len(tareas))]                    #tareas ya asignadas
    for i in range(len(tareas)):
        #if(tareas[i] == 0):
        found = False
        j = 0
        while (j < len(poblacion)) and (not found):                 #buscamos asignar a un trabajador sin tareas
            if(i <= poblacion[j][0]):                               #si la tarea no excede la cualificacion
                if((poblacion[j][i] != -1) and (j not in asignadas)):
                    asignadas[i] = j
                    found = True
            j += 1
        if(not found):                                              #si no encontramos asignamos al que menos tenga
            asignadas[i] = get_min_task(asignadas,poblacion,i)
    return asignadas
```

La búsqueda de dicho trabajador se realiza llamando a la función **get_min_task** mostrada a continuación.

```python
def get_min_task(asignadas,poblacion,tarea):
    """El trabajador con menos tareas ya asignadas con la tarea que recibe dispobile"""
    my_list = deepcopy(asignadas)
    my_list.sort()
    min_elem = -1
    min_count = 0
    current_elem = -1
    current_count = 0
    for i in my_list:
        if(i != -1):
            if(poblacion[i][tarea] == -1):
                del my_list[i]
    for i in my_list:
        if (i != -1):
            if(current_elem != i):
                if(current_elem == -1):
                    min_elem = i
                    min_count = 1
                elif(min_elem == -1 or min_count > current_count):
                    min_elem = current_elem
                    min_count = current_count
                current_elem = i
                current_count = 1
            else:
                current_count += 1
                if(min_elem == current_elem):
                    min_count = current_count

    if(min_elem == -1 or min_count > current_count):
        min_elem = current_elem

    return min_elem
```

A INTERÉS DEL/LA CORRECTOR/A

En un alarde de inspiración mientras realizábamos esta memoria, hemos caído en una resolución mucho más simple y correcta en términos de materia, desarrollo y ejecución del problema, en el cual los cromosomas son listas representando las tareas y los valores son los trabajadores asignados a dicha lista. Las poblaciones generadas solo tendrían que centrarse en asignar una tarea a un trabajador válido Esta aproximación, con un fitness enfocado en valorar el tiempo total de ejecución de las tareas según dicha lista es mucho más simple e, intuitiva y probablemente, evalúa poblaciones más variadas llegando así a resultados más óptimos.

Lamentablemente no hemos contado con el tiempo para implementar esta solución tras ~~perderlo~~ invertirlo en la resolución actual.