Técnicas de Soft Computing para Aprendizaje y optimización. Redes Neuronales y Metaheurísticas, programación evolutiva y bioinspirada

Problema de la mochila (Knapsack Problem)

Alberto Armijo Ruiz
Antonio Manuel Milán Jiménez

Presentación del problema

Este problema NP-Completo se basa en la existencia de una mochila que tiene una capacidad de peso máximo y una serie de objetos con un peso y un beneficio asociados. Así, la idea es decidir qué objetos se meten en la mochila cumpliendo la restricción de peso máximo e intentando conseguir el máximo beneficio posible.

Representación del problema

Para computacionalmente representar este problema se trabaja con un vector binario cuya longitud será el número de objetos con el que se trabaje y para cada posición se indicará con un “1” si el objeto correspondiente a esa posición se incluye y un “0” de lo contrario.

Evaluación de soluciones

Una solución se considerará factible si la suma de los pesos de aquellos objetos que se han incluido no supera el peso máximo de la mochila.

Para la evaluación de las soluciones factibles, función “fitness”, simplemente consiste en la suma de los beneficios de todos aquellos objetos que se hayan incluido.

Algoritmo Greedy

La aproximación Greedy para este problema consiste en intentar ir incluyendo el objeto de mayor beneficio siempre que la capacidad de la mochila lo permita. A continuación se muestra el pseudocódigo del algoritmo:

```
solucion = [0 0 ….. 0 ]  #tam nº objetos
indicesObjetosOrdenados = ordenar(objetos) #Se ordenan los objetos de mayor a menor en función de sus beneficios
pesoActual = 0

for indice in indicesObjetosOrdenados:
    if (pesoActual + pesos[indice] <= pesoMaximo):
        solucion[indice] = 1
        pesoActual += pesos[indice]

    if(pesoActual == pesoMaximo):
        break

return solucion
```

Algoritmo Búsqueda Local

En el algoritmo de búsqueda local se parte de una solución inicial, la cuál puede ser la solución Greedy obtenida anteriormente, y sobre ella se van realizando diferentes cambios o mutaciones. Cada vez que para una mutación se consigue una mejor solución que la obtenida hasta ese momento, esta solución mutada pasa a ser la mejor solución hasta el momento y se continúa con el proceso.

Operación de mutación

Esta operación de mutación consiste en simplemente seleccionar dos posiciones aleatorias de la solución e intercambiarlas:

```
indice1 = random(0,len(solucion))
indice2 = random(0,len(solucion))

while(indice1 == indice2): #Se obliga a que sean diferentes ambos indices
    indice2 = random(0,len(solucion))

aux = solucion[indice1]
solucion[indice1] = solucion[indice2]
solucion[indice2] = aux

return solucion
```

Este es el pseudocódigo del algoritmo de búsqueda local construido:

```
mejorSolucion = solucionGreedy
mejorBeneficio = beneficioGreedy

from 0 to 1000: #Un numero de iteraciones suficientemente alto

    nuevaSolucion = mutar(mejorSolucion)
    while(calcularPeso(nuevaSolucion) > pesoMaximo): #Aseguramos que la solucion creada es factible
        nuevaSolucion = mutar(mejorSolucion)

    beneficio = calcularBeneficio(nuevaSolucion)

    if(beneficio > mejorBeneficio):  #Es mejor solucion
        mejorSolucion = nuevaSolucion
        mejorBeneficio = beneficio

return mejorSolucion
```

Algoritmo Genético

Este algoritmo bioinspirado se basa en trabajar sobre una “población” de soluciones sobre la que se van sucediendo diversas “generaciones” de nuevas soluciones. En esta evolución de la población suceden recombinaciones y mutaciones, inspirado en la evolución biológica, que persiguen una mejora de la calidad de la población de soluciones.

Además, este algoritmo genético será estacionario lo cual significa que en cada generación se reemplazarán a tantos individuos como hijos se hayan creado.

Esta es la estructura del algoritmo genético:

```
poblacion = generarPoblacion(pesos, beneficios, maximoPeso, tamPoblacion)

for 0 to nIteraciones:  #Número de generaciones

    padres = seleccionPadres(poblacion, nHijos*2)   #Selección de padres

    for i in [0,nHijos]:

        r = random(0,1)
        if(r < probCruce):  #Existe una probabilidad de cruce
            padre1 = [i*2]
            padre2 = [i*2 +1]
            hijo = cruce(padre1,padre2) #Se realiza el cruce entre padres
            hijos.append(hijo)

    for i in [0,len(hijos)]:

        hijo = hijos[i]
        for 0 to tamMutaciones: #Hay un numero definido de mutaciones sobre los hijos

            r = random(0,1)
            if(r < probMutacion): #Existe una probabilidad de mutacion

                mutacion = mutar(hijo) #Se realiza la mutacion del hijo
                while(calcularPeso(mutacion) > pesoMaximo): #Aseguramos que es una solucion factible
                    mutacion = mutar(hijo)
                hijo = mutacion

        hijosMutados.append(hijo)


    poblacion = ordenar(poblacion)  #Se ordenado la poblacion de mayor a menor en función del fitness de cada individuo

    n = len(poblacion)-1
    for k in [0,len(hijosMutados)]: #Se sustituyen los peores individuos de la poblacion por los hijos mutados
        poblacion[n-k] = hijosMutados[k]


poblacion = ordenar(poblacion)  #Se ordenado la poblacion de mayor a menor en función del fitness de cada individuo
return poblacion[0] #Se devuelve el mejor de la poblacion
```

Creación de la población inicial

El primer paso es crear una población inicial de soluciones, la cual estará formada por una serie de soluciones aleatorias junto con la solución Greedy del problema para asegurar que de inicio haya alguna solución con más calidad en la población.

Este es el pseudocódigo de la generación de soluciones aleatorias:

```
solucion = [0 0 ... 0] #tam nº objetos
indices = [0 1 ... len(objetos)-1]

while(indices not empty):

    indice = random.choice(indices) #Se escoge un indice aleatoriamente
    indices.remove(indice)  #Se elimina el indice
    solucion[indice] = 1    #Se añade a la solucion el objeto correspondiente al indice

    if(calcularPeso(solucion) > pesoMaximo):
        solucion[indice] = 0    #Si se sobrepasa el peso maximo se desecha el objeto añadido
    elif(calcularPeso(solucion) == pesoMaximo): #Ya no es posible añadir más objetos
        break

return solucion
```

Y aquí el pseudocódigo de la creación de la población inicial:

```
for 0 to tamPoblacion-1:
    poblacion.append(generarSolucionAleatoria())

poblacion.append(solucionGreedy)

return poblacion
```

Selección de padres

Para la selección de padres se ha optado por la selección por ruleta. Consiste en asignar una probabilidades de selección como padre a cada individuo en función de su “fitness”. De esta forma, se da prioridad de cruce a los mejores individuos (explotación) aunque también se da la posibilidad de cruce a peores individuos que en algún momento sus características podrían ser interesantes.

Este es el pseudocódigo de la selección de padres. Para los experimentos se han escogido 4 padres para generar así 2 hijos en cada generación.

```
poblacionAux = {[individuo1, individuo2, ...],[fitness1, fitness2, ...]}

for 0 to nParents:

    allFitness = poblacionAux.getFitness()
    probabilidades = allFitness/sum(allFitness) #Lista de probabilidades para cada objeto en función de su fitness

    ruleta = probabilidades[0]
    for i in [1,len(probabilidades)]:   #Se realiza una suma acumulativa de las probabilidades
        ruleta.append(ruleta[i-1] + probabilidades[i])

    r = random(0,1) #Numero aleatorio entre 0 y 1
    seleccionado = len(ruleta) - 1
    for i in [0,len(ruleta)]:
        if(r<ruleta[i]):    #Si el numero aleatorio es menor que la suma acumulada para un objeto, éste será el objeto seleccionado
            seleccionado = i
            break

    padres.append(poblacion[seleccionado])  #Se añade como padre el objeto seleccionado
    del poblacionAux[seleccionado]  #Se eliminado el objeto seleccionado para evitar que vuelva a salir como padre

return padres
```

Cruce de padres

Para el cruce de padres se ha optado por la recombinación aritmética. Así, si una característica está presente en ambos padres (ambos han seleccionado o no el mismo objeto), esa característica la heredará el hijo. Por el contrario, si un padre ha seleccionado un objeto y el otro no, se decide aleatoriamente si escogerlo o no para dar la posibilidad de que se hereden características únicas de cada padre.

```
hijo = [0 0 ... 0] #tam nº objetos
for i in [0, len(padre1)]:

    if(padre1[i] == padre2[i]): #Es la misma caracteristica
        hijo[i] = padre1[i]
    else:
        r = random(0,1)
        if(r < 0.5):
            hijo[i] = padre1[i]
        else:
            hijo[i] = padre2[i]

    if(calcularPeso(hijo) > maximoPeso): #Si el ultimo objeto añadido la hace una solucion no factible, se elimina dicho objeto
        hijo[i] = 0
    elif(calcularPeso(hijo) == maximoPeso):
        break

return hijo
```

Mutación de hijos

Para cada hijo creado, se aplica el operador de mutación visto en la búsqueda local con una cierta probabilidad y un número determinado de veces.

Reemplazo de individuos

Tratándose de un algoritmo genético estacionario, se opta por un reemplazo elitista en el que se reemplazarán los peores individuos de la población por los hijos que se hayan creado, intentando que en cada generación aumente la calidad media de la población.

Algoritmo Memético

En el algoritmo memético se realiza una hibridación entre el algoritmo genético construido y una optimización de los hijos creados en cada generación mediante la búsqueda local, aumentando por lo tanto la fase de explotación.

```
poblacion = generarPoblacion(pesos, beneficios, maximoPeso, tamPoblacion)

for 0 to nIteraciones:  #Número de generaciones

    padres = seleccionPadres(poblacion, nHijos*2)   #Selección de padres

    for i in [0,nHijos]:

        r = random(0,1)
        if(r < probCruce):  #Existe una probabilidad de cruce
            padre1 = [i*2]
            padre2 = [i*2 +1]
            hijo = cruce(padre1,padre2) #Se realiza el cruce entre padres
            hijos.append(hijo)

    for i in [0,len(hijos)]:

        hijo = hijos[i]
        for 0 to tamMutaciones: #Hay un numero definido de mutaciones sobre los hijos

            r = random(0,1)
            if(r < probMutacion): #Existe una probabilidad de mutacion

                mutacion = mutar(hijo) #Se realiza la mutacion del hijo
                while(calcularPeso(mutacion) > pesoMaximo): #Aseguramos que es una solucion factible
                    mutacion = mutar(hijo)
                hijo = mutacion

        hijosMutados.append(hijo)


    for i in [0,len(hijosMutados)]: #Optimización por búsqueda local de los hijos creados
        hijosMutados[i] = busquedaLocal(hijosMutados[i])


    poblacion = ordenar(poblacion)  #Se ordenado la poblacion de mayor a menor en función del fitness de cada individuo

    n = len(poblacion)-1
    for k in [0,len(hijosMutados)]: #Se sustituyen los peores individuos de la poblacion por los hijos mutados
        poblacion[n-k] = hijosMutados[k]


poblacion = ordenar(poblacion)  #Se ordenado la poblacion de mayor a menor en función del fitness de cada individuo
return poblacion[0] #Se devuelve el mejor de la poblacion
```

Algoritmo de colonia de hormigas

El algoritmo de ACO (Optimización de Colonia de Hormigas o Ant Colony Optimization en inglés) se trata de un algoritmo bioinspirado en el comportamiento de las hormigas para obtener caminos más cortos; este algoritmo suele usarse en problemas de grafos para calcular la ruta óptima, aunque también puede adaptarse a el problema que se estudia en esta práctica.

El algoritmo ACO se basa en el uso de una feromona que las hormigas van depositando por los caminos que van pasando, o en el caso de este problema los objetos que se van seleccionando. En cada momento, una hormiga elige un objeto con una cierta probabilidad y la añade a su solución; una vez todas las hormigas han elegido su solución, se actualiza la mejor solución si se ha obtenido una mejor a la mejor solución que haya hasta el momento y se actualiza el valor de la feromona depositada en cada objeto dependiendo de las soluciones. Este proceso se repite un número de iteraciones fijado al comienzo del algoritmo.

Para evitar que haya una convergencia muy rápida por el algoritmo y se quede en óptimos locales, se realiza un proceso de evaporación de la feromona; este proceso multiplica la feromona depositada en cada uno de los objetos por un factor de evaporación, el cual se encuentra entre [0,1). Para esta práctica se utilizará un valor igual a 0.95.

La estructura de este algoritmo es la siguiente.

```
mejorSolucion = generarSolucionInicial()

for 0 to nIteraciones:
    for 0 to nHormigas:
        s = generarSolucion()
        if (s > mejorSolucion):
            mejorSolucion = s
        soluciones.append(s)

    for sol in soluciones:
        actualizarFeromona(sol)

    evaporacionFeromona()

return mejorSolucion
```

La solución inicial que se utiliza en este algoritmo es la solución proporcionada por un algoritmo Greedy.

Generación de una solución

Para este algoritmo la generación de una solución se realiza mediante la selección de objetos de forma semi-aleatoria dependiendo de la probabilidad de cada objeto. Para calcular dicha probabilidad se utiliza el valor de la feromona de cada objeto multiplicada por su atractivo y dividido entre la sumatoria de este producto para todos los objetos; dicho atractivo se calcula como la división entre el valor del objeto y su peso al cuadrado. Si un objeto ya se encuentra en la solución de la hormiga, la probabilidad de elegir este objeto es 0. Este proceso se repite hasta que no quepan más objetos en la mochila.

Inicialmente, los valores de las feromonas por cada objeto es la misma para todos, en nuestro caso se ha utilizado 1; de forma que la primera vez que se generan soluciones las hormigas solamente se fijan en el atractivo de cada objeto; conforme se va iterando, el valor de las feromonas cambia (por su actualización y evaporación) y va tomando más importancia a la hora de seleccionar un objeto.

La estructura para la generación de una solución sería la siguiente.

```
def generarSolucion():
    while(capacidad < capacidad_mochila):
        objeto = seleccionarObjeto(feromonas,atractivo, solucion)
        solucion[objeto] = 1
        capacidad += pesos[objeto]
    return solucion

def seleccionarObjeto(feromonas, atractivo, solucion):
    probabilidades = feromonas
    probabilidades[solucion == 1] = 0

    suma_total = probabilidades.transpuesta * atractivo
    suma_probs = 0

    for i in [0,len(probabilides)]
        probabiliades[i] = (probabilidades[i]*atractivo[i])/suma_total
        suma_probs += probabilidades[i]


    probabilidades_norm = probabilidades / suma_probs

    return random([0,len(probabilidades)],probs=probabilidades_norm)
```

Actualización y evaporación de las feromonas

Para actualizar las feromonas se le añade a la feromona de cada objeto elegido en la solución una cantidad calculada anteriormente; esta cantidad se calcula de la siguiente forma.

$$\frac{1}{1+(\frac{valorsolucion + valormejorsolucion}{valormejorsolucion})}$$

La estructura de es algoritmo sería la siguiente.

```
def actualizarFeromona(feromonas,solucion,valor_mejor_solucion,valor_solucion):
    ratio_mejora = 1 / (1+((valor_solucion+valor_mejor_solucion)/valor_solucion))

    for i in [0,len(feromonas)]:
        if(solucion[i] == 1):
            feromonas[i] += ratio_mejora

    return feromonas
```

Para la evaporación de feromonas, simplemente hay que multiplicar cada uno de los valores de las feromonas por el factor de evaporación. La estructura es la siguiente.

```
def evaporacionFeromona(feromonas, factorEvaporacion):
    return feromonas*factorEvaporacion
```

Resultados obtenidos

Problema 1

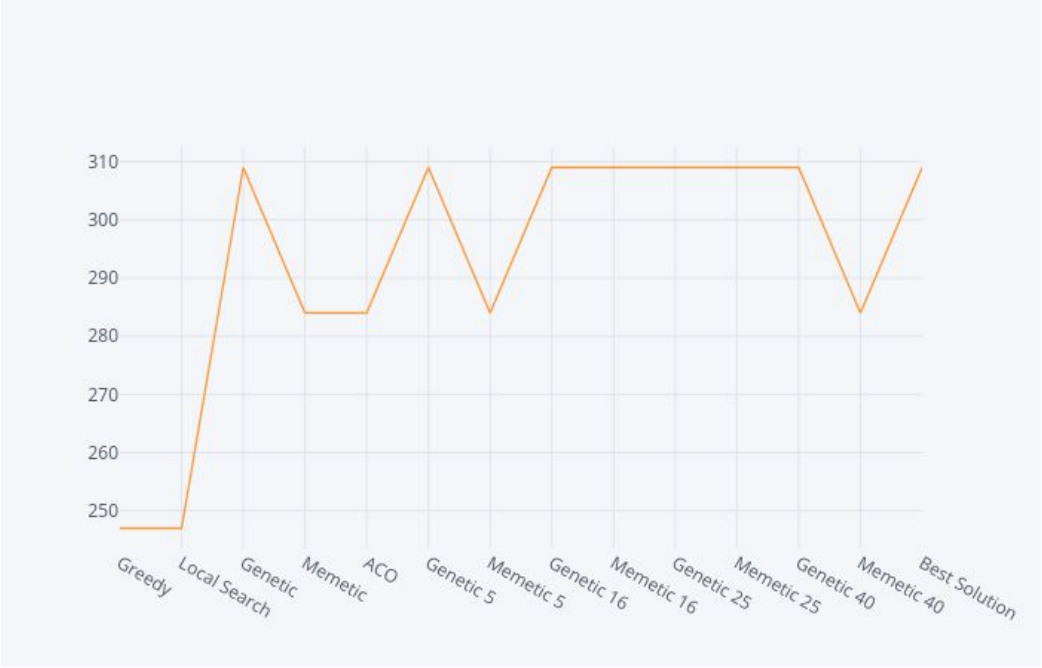
- Número de objetos: 10
- Capacidad de la mochila: 165
- Pesos de los objetos: [23, 31, 29, 44, 53, 38, 63, 85, 89, 82]
- Beneficios de los objetos: [92, 57, 49, 68, 60, 43, 67, 84, 87, 72]
- Solución óptima: [1, 1, 1, 1, 0, 1, 0, 0, 0, 0]
- Beneficio solución óptima: 309

Algoritmo (iters)	Tamaño población	Solución construida	Tiempo (s)	Beneficio Total
Greedy	0	[1. 0. 0. 1. 0. 0. 0. 0. 1. 0.]	0.00004	247
Búsqueda Local	0	[1. 0. 0. 1. 0. 0. 0. 0. 1. 0.]	0.01013	247
Genético (1000)	8	[1. 1. 1. 1. 0. 1. 0. 0. 0. 0.]	0.18161	309
Memético (100x50)	8	[1. 1. 0. 1. 0. 0. 1. 0. 0. 0.]	0.13207	284
ACO (200)	10	[1. 1. 0. 1. 0. 0. 1. 0. 0. 0.]	2.1429	284

Además, se ha trabajado sobre los algoritmos genéticos y meméticos con diferentes tamaños de poblaciones:

Algoritmo (iters)	Tamaño población	Solución construida	Tiempo (s)	Beneficio Total
Genético (1000)	5	[1. 1. 1. 1. 0. 1. 0. 0. 0. 0.]	0.179	309
Memético (100x50)	5	[1. 1. 0. 1. 0. 0. 1. 0. 0. 0.]	0.14	284
Genético (1000)	8	[1. 1. 1. 1. 0. 1. 0. 0. 0. 0.]	0.2	309

Memético (100x50)	8	[1. 1. 0. 1. 0. 0. 1. 0. 0. 0.]	0.149	284
Genético (1000)	16	[1. 1. 1. 1. 0. 1. 0. 0. 0. 0.]	0.149	309
Memético (100x50)	16	[1. 1. 1. 1. 0. 1. 0. 0. 0. 0.]	0.182	309
Genético (1000)	25	[1. 1. 1. 1. 0. 1. 0. 0. 0. 0.]	0.31	309
Memético (100x50)	25	[1. 1. 1. 1. 0. 1. 0. 0. 0. 0.]	0.155	284
Genético (1000)	40	[1. 1. 1. 1. 0. 1. 0. 0. 0. 0.]	0.408	309
Memético (100x50)	40	[1. 1. 0. 1. 0. 0. 1. 0. 0. 0.]	0.165	284.0



Para este problema observamos que el algoritmo genético consigue alcanzar la solución óptima para los diferentes tamaños de la población. El algoritmo memético consigue también alcanzar la solución óptima en algunas ocasiones, obteniendo otras veces un solución igualmente bastante buena al igual que el algoritmo ACO. Dentro de los esperado, son los algoritmos Greedy y Búsqueda Local los que calculan las peores soluciones, aunque es cierto que mucho más rápido que el resto de algoritmos.

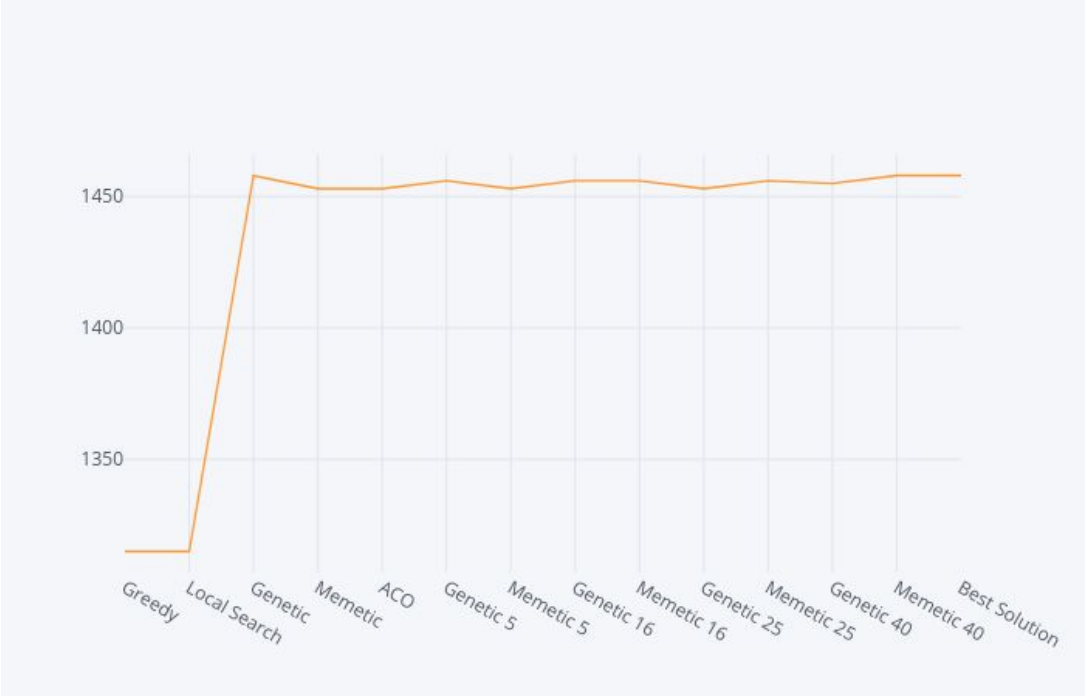
Problema 2

- Número de objetos: 15
- Capacidad de la mochila: 750
- Pesos de los objetos: [70, 73, 77, 80, 82, 87, 90, 94, 98, 106, 110, 113, 115, 118, 120]
- Beneficios de los objetos: [135, 139, 149, 150, 156, 163, 173, 184, 192, 201, 210, 214, 221, 229, 240]
- Solución óptima: [1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1]
- Beneficio solución óptima: 1458

Algoritmo (iters)	Tamaño población	Solución construida	Tiempo (s)	Beneficio Total
Greedy	0	[0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1]	0.00002	1315
Búsqueda Local	0	[0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1]	0.008	1315
Genético (1000)	8	[1. 0. 1. 0. 1. 0. 1. 1. 1. 0. 0. 0. 0. 1. 1.]	0.23	1458
Memético (100x50)	8	[0. 0. 0. 0. 0. 0. 1. 1. 1. 0. 0. 1. 1. 1. 1.]	0.115	1453
ACO (200)	10	[1. 1. 1. 0. 1. 0. 0. 1. 0. 0. 0. 0. 1. 1. 1.]	3.7221	1453

Algoritmo (iters)	Tamaño población	Solución construida	Tiempo (s)	Beneficio Total
Genético (1000)	5	[0. 1. 1. 1. 0. 0. 1. 1. 1. 0. 0. 0. 0. 1. 1.]	0.217	1456
Memético (100x50)	5	[0. 0. 0. 0. 0. 0. 1. 1. 1. 0. 0. 1. 1. 1. 1.]	0.099	1453
Genético (1000)	8	[1. 0. 1. 0. 1. 0. 1. 1. 1. 0. 0. 0. 0. 1. 1.]	0.23	1458
Memético (100x50)	8	[0. 0. 0. 0. 0. 0. 1. 1. 1. 0. 0. 1. 1. 1. 1.]	0.115	1453

Genético (1000)	16	[0. 1. 1. 1. 0. 0. 1. 1. 1. 0. 0. 0. 0. 1. 1.]	0.281	1456
Memético (100x50)	16	[0. 1. 1. 1. 0. 0. 1. 1. 1. 0. 0. 0. 0. 1. 1.]	0.111	1456
Genético (1000)	25	[1. 1. 1. 0. 1. 0. 0. 1. 0. 0. 0. 0. 1. 1. 1.]	0.354	1453
Memético (100x50)	25	[0. 1. 1. 1. 0. 0. 1. 1. 1. 0. 0. 0. 0. 1. 1.]	0.127	1456
Genético (1000)	40	[1. 1. 0. 0. 0. 1. 1. 1. 1. 0. 0. 0. 0. 1. 1.]	0.45	1455
Memético (100x50)	40	[1. 0. 1. 0. 1. 0. 1. 1. 1. 0. 0. 0. 0. 1. 1.]	0.135	1458



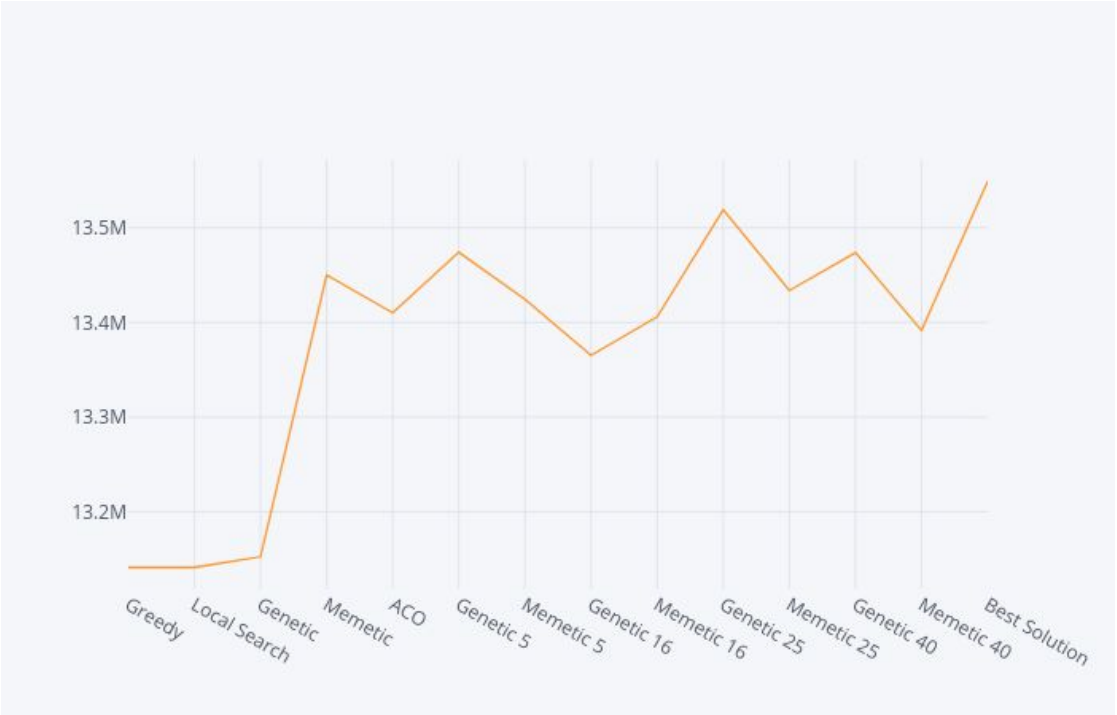
Para este problema, los algoritmos bioinspirados consiguen soluciones muy próximas a la solución óptima, incluso alcanzándola como es el caso del Genético con un población de 8 individuos o el Memético con una población de 40. Nuevamente, ‘Greedy’ y ‘Local Search’ obtienen las peores soluciones.

Problema 3

- Número de objetos: 24
- Capacidad de la mochila: 6404180
- Pesos de los objetos:[382745, 799601, 909247, 729069, 467902, 44328, 34610, 698150, 823460, 903959, 853665, 551830, 610856, 670702, 488960, 951111, 323046, 446298, 931161, 31385, 496951, 264724, 224916, 169684]
- Beneficios de los objetos: [825594, 1677009, 1676628, 1523970, 943972, 97426, 69666, 1296457, 1679693, 1902996, 1844992, 1049289, 1252836, 1319836, 953277, 2067538, 675367, 853655, 1826027, 65731, 901489, 577243, 466257, 369261]
- 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]
- Beneficio solución óptima: 13549094

Algoritmo (iters)	Tamaño población	Solución construida	Tiempo (s)	Beneficio Total
Greedy	0	[0. 1. 1. 0. 0. 0. 0. 0. 1. 1. 1. 0. 0. 0. 0. 1. 0. 0. 1. 0. 0. 0. 1. 0.]	0.00003	13141140
Búsqueda Local	0	[0. 1. 1. 0. 0. 0. 0. 0. 1. 1. 1. 0. 0. 0. 0. 1. 0. 0. 1. 0. 0. 0. 1. 0.]	0.009	13141140
Genético (1000)	8	[0. 1. 1. 0. 0. 1. 0. 0. 1. 1. 1. 0. 1. 0. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0.]	0.293	13152395
Memético (100x50)	8	[1. 1. 0. 1. 0. 0. 0. 0. 1. 1. 1. 0. 1. 0. 0. 1. 1. 0. 0. 0. 0. 0. 0. 0.]	0.123	13410139
ACO (200)	10	[1. 1. 0. 1. 1. 1. 1. 0. 0. 0. 1. 0. 1. 0. 1. 1. 1. 0. 0. 1. 0. 1. 1. 1.]	5.74995	13389984

Algoritmo (iters)	Tamaño población	Solución construida	Tiempo (s)	Beneficio Total
Genético (1000)	5	[1. 1. 0. 0. 0. 0. 1. 0. 1. 1. 1. 0. 1. 0. 0. 1. 1. 0. 0. 1. 0. 1. 1. 1.]	0.287	13474183
Memético (100x50)	5	[1. 1. 0. 1. 1. 0. 0. 0. 0. 1. 1. 0. 1. 1. 0. 1. 0. 0. 0. 1. 0. 0. 0. 0.]	0.102	13424474
Genético (1000)	8	[0. 1. 1. 0. 0. 1. 0. 0. 1. 1. 1. 0. 1. 0. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0.]	0.293	13152395
Memético (100x50)	8	[1. 1. 0. 1. 0. 0. 0. 0. 1. 1. 1. 0. 1. 0. 0. 1. 1. 0. 0. 0. 0. 0. 0. 0.]	0.123	13410139
Genético (1000)	16	[1. 0. 0. 0. 0. 1. 1. 0. 1. 1. 1. 1. 1. 0. 1. 1. 1. 0. 0. 0. 0. 1. 0. 1.]	0.35	13365178
Memético (100x50)	16	[1. 1. 0. 0. 1. 0. 1. 0. 1. 1. 1. 0. 1. 0. 0. 1. 1. 0. 0. 0. 0. 0. 1. 0.]	0.127	13405920
Genético (1000)	25	[1. 1. 0. 1. 0. 1. 1. 0. 1. 1. 1. 0. 1. 0. 0. 1. 0. 0. 0. 0. 0. 1. 0. 0.]	0.405	13518963
Memético (100x50)	25	[1. 1. 0. 0. 1. 1. 0. 0. 1. 1. 1. 0. 1. 0. 0. 1. 1. 0. 0. 0. 0. 0. 1. 0.]	0.134	13433680
Genético (1000)	40	[1. 1. 0. 1. 0. 1. 1. 0. 1. 1. 1. 0. 1. 0. 0. 1. 0. 0. 0. 1. 0. 0. 1. 0.]	0.503	13473708
Memético (100x50)	40	[1. 1. 0. 1. 0. 0. 1. 0. 1. 1. 1. 0. 0. 0. 0. 1. 0. 1. 0. 0. 0. 1. 0. 1.]	0.143	13391617



Para este último problema de mayor dificultad, si bien ningún algoritmo alcanza la mejor solución, sí que se consiguen soluciones bastantes buenas por parte de los Genéticos con poblaciones de 25 y 5 individuos. Además, observamos que el algoritmo ACO llega a obtener mejor resultados que algunos algoritmos genéticos y meméticos.

Variación multiobjetivo

Existe la posibilidad de realizar una aproximación de multiobjetivo para este problema de la mochila. Así, se podría establecer como objetivo, no solo maximizar el beneficio de los objetos sino también intentar minimizar el peso llevado en la mochila. La solución ideal sería obtener el mayor beneficio posible con el menor peso posible.

Para adaptar esta variación multiobjetivo bastaría con modificar la función fitness para que también se tuviese en cuenta este segundo aspecto. Por ejemplo, así se podría calcular este fitness:

$$fitness = (1 - \frac{pesoSolucion}{pesoMaximo}) * \alpha + \frac{beneficioSolucion}{beneficioMaximo} * \beta$$

Con los parámetros α y β se podría establecer la importancia de la minimización del peso y la maximización del beneficio respectivamente. Si se quiere que ambos aspectos tengan la misma relevancia, ambos parámetros se establecerían a 0.5. Por ejemplo, si se quisiese dar mayor importancia al peso de la solución, los parámetros podrían ser α=0.75 y β = 0.25. No sería necesario realizar más modificaciones sobre el problema original y los algoritmos construidos pues seguiría tratándose de maximizar el valor fitness de las soluciones.