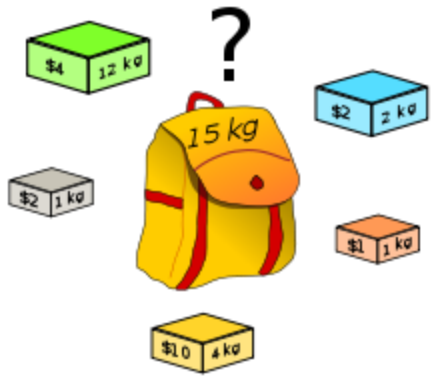
Algoritmo de la colonia de hormigas

Aplicado al problema de la mochila



Autores: Juan Eugenio Cortés Díaz
David Rubio Lucas

Resumen

En este proyecto se comenzará por una pequeña introducción de los tres ejes en los que gira el mismo, el algoritmo de las colonias de hormigas, el problema de la mochila y el lenguaje Python.

Tras ello el proyecto se divide en tres partes, la primera son seis puntos que recorren desde un marco genérico para el algoritmo de las hormigas, pasando por diferentes versiones hasta alcanzar la mejor que resuelva el problema de la mochila con los elementos exigidos, acabando con pruebas entre la mejor versión con la anterior y pruebas con un algoritmo de fuerza bruta, comparando el rendimiento y otros elementos.

La segunda parte trata lo solicitado de un cambio sustancial en cada iteración y unas pruebas comparándolo con lo anterior.

Finalizando con la tercera parte donde únicamente se habla de la bibliografía usada para terminar este proyecto de la forma más definitiva.

ÍNDICE

1. Introducción

En este trabajo trataremos de resolver el problema de la mochila mediante el algoritmo de las hormigas.

En nuestro mundo natural, las hormigas (inicialmente) vagan de manera aleatoria, al azar, y una vez encontrada comida regresan a su colonia dejando un rastro de feromonas. Si otras hormigas encuentran dicho rastro, es probable que estas no sigan caminando aleatoriamente, puede que estas sigan el rastro de feromonas, regresando y reforzándolo si estas encuentran comida finalmente.

Sin embargo, al paso del tiempo el rastro de feromonas comienza a evaporarse, reduciéndose así su fuerza de atracción. Cuanto más tiempo le tome a una hormiga viajar por el camino y regresar de vuelta otra vez, más tiempo tienen las feromonas para evaporarse. Un camino corto, en comparación, es marchado más frecuentemente, y por lo tanto la densidad de feromonas se hace más grande en caminos cortos que en los largos. **La evaporación de feromonas también tiene la ventaja de evitar convergencias a óptimos locales.** Si no hubiese evaporación en absoluto, los caminos elegidos por la primera hormiga tenderían a ser excesivamente atractivos para las siguientes hormigas. En este caso, el espacio de búsqueda de soluciones sería limitado.

Por tanto, cuando una hormiga encuentra un buen camino entre la colonia y la fuente de comida, hay más posibilidades de que otras hormigas sigan este camino y con una retroalimentación positiva se conduce finalmente a todas las hormigas a un solo camino. La idea del algoritmo colonia de hormigas es imitar este comportamiento con "hormigas simulada" caminando a través de un grafo que representa el problema en cuestión.

Esto es lo que vamos aplicar para resolver el antes citado problema de la mochila, el cual es un problema de optimización combinatoria, es decir, que busca la mejor solución entre un conjunto (finito) de posibles soluciones a un problema. Modela una situación análoga al llenar una mochila, incapaz de soportar más de un peso determinado, con todo o parte de un conjunto de objetos, cada uno con un peso y valor específicos. Los objetos colocados en la mochila deben maximizar el valor total sin exceder el peso máximo.

Ambas cuestiones deben ser tratadas y valoradas en este proyecto teniendo en cuenta lo citado y utilizando el lenguaje solicitado python.

Python es un lenguaje de programación interpretado cuya filosofía hace hincapié en una sintaxis que favorezca un código legible.

Se trata de un lenguaje de programación multiparadigma, ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional. Es un lenguaje interpretado, usa tipado dinámico y es multiplataforma.

2. Primera parte

2.1. Marco genérico del problema en Python.

01-Algoritmo Generico.py

En este algoritmo se prepara la base para realizar los diferentes algoritmos, en el mismo fichero se define la hormiga para realizar un problema de distancias de ciudades, y se deja implementado para su muestra.

Captura 1:

```
# Elige un paso para una hormiga, teniendo en cuenta los valores
# las feromonas y descartando los nodos ya visitados.
def eligeNodo(valors, ferom, visitados):
    #Se calcula la tabla de pesos de cada ciudad
    listaValores  = []
    disponibles = []
    actual      = visitados[-1]
```

Al principio definimos el método elige nodo, que elegirá los “nodos” que pueden ser objetos, ciudades como en el ejemplo que se usa en el fichero, o cualquier elemento.

Como bien se explica descarta los que ya ha visitado y tiene en cuenta las feromonas y el valor del objeto.

Captura 2:

```
    # Influencia de cada valor (alfa: feromonas; beta: valor)
    alfa = 1.0
    beta = 0.5

    # El parámetro beta (peso de los valores) es 0.5 y alfa=1.0
    for i in range(len(valors)):
        if i not in visitados:
            fer  = math.pow((1.0 + ferom[actual][i]), alfa)
            peso = math.pow(1.0/valors[actual][i], beta) * fer
            disponibles.append(i)
            listaValores.append(peso)

    # Se elige aleatoriamente una de los nodos disponibles,
    # teniendo en cuenta su peso relativo.
    valor     = random.random() * sum(listaValores)
    acumulado = 0.0
    i         = -1
    while valor > acumulado:
        i         += 1
        acumulado += listaValores[i]

    return disponibles[i]
```

Aquí calculamos la influencia de las feromonas en base a los parámetros alfa y beta, usando la función math.pow que eleva la suma de la feromona a la cantidad actual, elevándola a la misma. Con el peso se hace igual pero se va dividiendo el valor y todo se multiplica por la feromona.

Como se puede comprobar luego se añade a “disponibles” el nodo i y a la lista de valores el peso, ¿porque se hace esto?, muy sencillo, se comprueba que el nodo no ha sido visitado aun, es decir es un objeto que aún no ha recorrido, se calcula la feromona y con ella el valor (peso) por lo que al final se incluye en posibles nodos a elegir por el método con su correspondiente valor.

Captura 3:

```
    # Se elige aleatoriamente una de los nodos disponibles,
    # teniendo en cuenta su peso relativo.
    valor     = random.random() * sum(listaValores)
    acumulado = 0.0
    i         = -1
    while valor > acumulado:
        i          += 1
        acumulado += listaValores[i]

    return disponibles[i]
```

Como bien se explica ahora se escoge un nodo de la lista de disponibles teniendo en cuenta su valor (peso) y como el peso se ha calculado entorno a las feromonas al final nos ira eligiendo los nodos mejores y tendremos una mejor solución ya que este método lo llama el método elegir camino, que es el método que nos da la solución final:

Captura 4:

```
# Genera una " hormiga " , que eligirá un camino (nodos que visita) teniendo en cuenta
# los valores y los rastros de feromonas. Devuelve una tupla
# con el camino (nodos visitados) y su longCamino (Suma de valores).
def eligeCamino(matriz, feromonas):
    # El nodo inicial siempre es el 0
    camino     = [0]
    longCamino = 0

    # Elegir cada paso según los valores y las feromonas
    while len(camino) < len(matriz):
        nodo      = eligeNodo(matriz, feromonas, camino)
        longCamino += matriz[camino[-1]][nodo]
        camino.append(nodo)

    # Para terminar hay que volver al nodo de origen (0)
    longCamino += matriz [camino[-1]][0]
    camino.append(0)

    return (camino, longCamino)
```

Como podemos comprobar el “camino” es un array o tupla donde vienen en orden los objetos, ciudades, o como les llamamos de forma genérica nodos.

Y el longcamino que es una variable de valor numérica donde vendrá la suma total de sus valores. Esta versión genérica se basa en que los objetos solo tienen un variable asociada, en el caso de la mochila que tienen valor y peso se calcularían más cosas que veremos más adelante.

Como podemos comprobar en este método se le mete la matriz de objetos y la de feromonas que en como veremos en el método de definir hormiga primero se crea vacía y se le va llenando con la cantidad de feromona de cada nodo.

Ahora la condición while simplemente se basa en que si aún quedan objetos o no, si ya ha recorrido todos ya ha acabado.

El nodo se selecciona llamando al método anteriormente explicado. Y como podemos comprobar el longcamino se calcula con el valor del nodo elegido sumado a lo anterior calculado.

En esta versión que usamos genérica vuelve al nodo de origen, cuando hagamos lo de la mochila esa parte no se usa.

Captura 5:

```
# Resuelve el problema del viajante de comercio mediante el
# algoritmo de la colonia de hormigas. Recibe una matriz de
# distancias y devuelve una tupla con el mejor camino que ha
# obtenido (lista de índices) y su longitud
def hormigas(matriz, iteraciones, distMedia):
    # Primero se crea una matriz de feromonas vacía
    n = len(matriz)
    feromonas = [[0 for i in range(n)] for j in range(n)]

    # El mejor camino y su longitud (inicialmente "infinita")
    mejorCamino     = []
    longMejorCamino = sys.maxsize

    # En cada iteración se genera una hormiga, que elige una camino,
    # y si es mejor que el mejor que teníamos, deja su rastro de
    # feromonas (mayor cuanto más corto sea el camino)
    for iter in range(iteraciones):
        (camino,longCamino) = eligeCamino(matriz, feromonas)

        if longCamino <= longMejorCamino:
            mejorCamino     = camino
            longMejorCamino = longCamino

        rastroFeromonas(feromonas, camino, distMedia/longCamino)
        # En cualquier caso, las feromonas se van evaporando
        evaporaFeromonas(feromonas)

    # Se devuelve el mejor camino que se haya encontrado
    return (mejorCamino, longMejorCamino)
```

Como se puede comprobar en los comentarios, este método ya está preparado para el problema del viajante y ya está enfocado a él, cuando hagamos el de la mochila cambiara sustancialmente. Los comentarios dejan bastante claro el asunto.

Captura 6:

```
#Genera una matriz de valores de n x n, asignando valores aleatorios y poniendo un valor tope
#Los valores de un nodo A a un nodo B son los mismos que de un nodo B a A, siendo A y B dos nodos cualesquiera
#Le asignamos valores aleatorios
def matrizDistancias(n):
    matriz = [[0 for i in range(n)] for j in range (n)]

    for i in range(n):
        for j in range(i):
            matriz[i][j] = i+1
            matriz[j][i] = matriz[i][j]

    return matriz


# Ejemplo de uso
# Generación de una matriz de prueba
numCiudades      = 4
distanciaMaxima  = 100
ciudades         = matrizDistancias(numCiudades)

# Obtención del mejor camino, las iteraciones son la cantidad de hormigas que empleamos para resolverlo
iteraciones = 1000
distMedia   = numCiudades*distanciaMaxima/2
(camino, longCamino) = hormigas(ciudades, iteraciones, distMedia)
print("Camino: ", camino)
print("Longitud del camino: ", longCamino)
```

Como se puede ver, primero generamos una matriz de ciudades con valores genéricos, que serán las distancias entre las ciudades, decidimos el número de ciudades que influirán en el generar matriz, la distancia máxima se usara para calcular la distancia media que será usada por las hormigas este dato luego se modificara para el de la mochila.

Iteraciones es igual al número de hormigas en métodos posteriores se cambia el nombre de la variable.

Captura 7:

```
>>>
 RESTART: C:/Users/Juanoide_El_puto_amo/Desktop/Tutoria de mañana/algoritmo gene
rico.py
Camino:  [0, 2, 3, 1, 0]
Longitud del camino:  13
```

Como podemos comprobar nos da una solución.

2.2. Versiones previas

El primer problema que se planteaba era que nuestro programa genérico se inspiraba en el problema del viajante y estaba diseñado para una sola variable. Sin contar que el desarrollo del mismo se había implementado para el mismo problema del que se inspiraba.

Aquí surgían dos situaciones realizar otro programa genérico completamente distinto o usar el que ya se tenía adaptándolo al de la mochila.

En el caso de que no se pudiera adaptar significaría que el programa genérico no era genérico.

El problema inicial es que usábamos una matriz de coordenadas i e j y para el índice [i][j] es el mismo valor que para [j][i] ya que la distancia entre por ejemplo cadiz-sevilla, es la misma que de Sevilla-cadiz.

Un ejemplo de matriz para el algoritmo generico:

	0	**1**	**2**	**3**
0	0	2	3	4
1	2	0	2	6
2	3	2	0	8
3	4	6	8	0

Como podemos comprobar cuando i=j el resultado es cero. Primero lo aplicamos a que solo tuviera en cuenta el peso limitándolo por si superase el mismo.

Y decidimos que tendría que ser una matriz que el “valor” no podía ser el mismo de 0 a 1 que de 1 a 0 además de quitar la opción de que volví al nodo inicial cosa que en el algoritmo de la mochila no tenía sentido.

Por lo que si teníamos estos objetos con sus pesos y valores:

Objeto 1= 5
Objeto 2 = 10
Objeto 3 = 1
Objeto 4 = 10

Y adaptado a una tabla que pudiera resolver nuestro algoritmo genérico:

	0	**1**	**2**	**3**	**4**
0	0	5	10	1	10
1	0	0	10	1	10
2	0	5	0	1	10
3	0	5	10	0	10
4	0	5	10	1	0

Como funciona esta tabla, bien primero se tiene en cuenta que lo que antes era distancia ahora es el peso o valor que tiene el objeto al que se va, se incluye un objeto 0 desde el que partir siempre y cuando el objeto vaya hacia a si mismo sería cero.

Si vemos la tabla desde las columnas, vemos que es siempre el valor de ese objeto excepto cuando viene de sí mismo.

Para esta teoría se realizó una versión que leyera de fichero una tabla como la planteada arriba.

El fichero usado:

01-exemplo3.txt

Que en el interior del mismo:

Captura 8:

```
Identificador Peso Valor
0 5 10 1 10
0 0 10 1 10
0 5 0 1 10
0 5 10 0 10
0 5 10 1 0
```

Se introduce una línea con texto, para ir planteando ya el obviarla en el fichero final.

El código usado:

02-algoritmo que lee una matriz preparada.py

La nomenclatura usada es año mas mes más día y la versión usada, pero podemos obviar eso y centrarnos siempre en que al principio del nombre habrá un número.

Cambios realizados:

Captura 9:

```
# Mostramos por pantalla lo que leemos desde el fichero
print('>>> Cargamos la matriz de los txts')
f = open('01-exemplo3.txt')
# Primera lectura para obviar la cabecera
print(f.readline())
# Cargamos los datos
data = f.read().strip()
# Se cierra el fichero
f.close()
#Separamos los datos en un arrya
M = [[int(num) for num in line.strip().split()] for line in data.split('\n')]
print (M)
```

Como podemos ver el método más interesante es el de la penúltima línea que separa por espacio y por línea generando así una matriz. Como la que generábamos antes, pero ahora leyéndola desde fichero.

El gran problema que se planteaba ahora es, que nuestro algoritmo genérico no diferenciaba de visitados y de solución, dándonos una serie de errores. Nosotros los diferenciamos incluyendo ambos arrays en los métodos originales:

¿Cómo funciona la diferencia?, se ve mejor en el elige camino:

Captura 10:

```
# Genera una " hormiga " , que eligirá un camino (nodos que visita) teniendo en cuenta
# los valores y los rastros de feromonas. Devuelve una tupla
# con el camino (nodos solucion) y su longCamino (Suma de valores).
def eligeCamino(matriz, feromonas):
    # El nodo inicial siempre es el 0
    camino     = [0]
    visitados  = [0]
    longCamino = 0

    # Elegir cada paso según los valores y las feromonas
    while len(visitados) < len(matriz):
        nodo      = eligeNodo(matriz, feromonas, visitados, camino)

      #  print("camino: ", longCamino)

        camino.append(nodo)
        longCamino += matriz[visitados[-1]][nodo]
        if (longCamino > 20):
            camino.remove(nodo)
            longCamino -= matriz[visitados[-1]][nodo]
        visitados.append(nodo)

    return (camino, longCamino)
```

Como podemos comprobar ahora “solución” se llama camino y los nodos elegidos se guardan en ambos sitios pero si la elección de ese nodo ha superado el peso total que hemos impuesto, en este caso 20, es eliminado de “camino” pero se elimine o no, siempre se incluye en visitados.

Consiguiendo así una solución, haciendo simplemente pequeños cambios en elige nodo y elige camino.

La solución que nos muestra es esta:

Captura 11:

```
 RESTART: C:\Users\Juanoide_El_puto_amo\Dropbox\IA\Trabajo de ia\02-ACO20160604v4.py
>>> Cargamos la matriz de los txts
Identificador Peso Valor

[[0, 5, 10, 1, 10], [0, 0, 10, 1, 10], [0, 5, 0, 1, 10], [0, 5, 10, 0, 10], [0, 5, 10, 1, 0]]
Camino:  [0, 1, 3, 4]
Longitud del camino:  16
```

En la imagen se ve que primero imprimimos el fichero luego el camino tomado, o la solución para luego exponer la longitud del camino o lo que es lo mismo el peso total guardado en la mochila, como bien sabemos hemos puesto el límite de 20 y cumple con lo establecido.

Como es obvio ahora hay que conseguir realizarlo teniendo en cuenta los valores y leyendo el fichero como se exige.

Así que modificamos una sería de cosas para esta versión intermedia.

El fichero a usar sería este:

02-exemplo7.txt

Y el código este:

03-algoritmo que coge una solucion pero no la mejor.py

Para ello modificamos algo el leer fichero ya que hay que leer este txt:

Identificador	Peso	Valor
1	5	10
2	10	1
3	100	50
4	500	100
5	11	99
6	53	33
7	12	100
8	66	500
9	13	5
10	7	77
11	5	10
12	10	1
13	100	50
14	500	100
15	11	99

16	53	33
17	12	100
18	66	500
19	7	77
20	13	5

Y adaptarlo al formato que expusimos antes, para ello hemos elaborado este método:

Captura 12:

```
def matrizPesos(n):

#Primero cogemos los pesos de la matriz creada al leer el txt, añadiendo un cero al inicio

    matrizdePesos=[0]
    for i in range(len(n)):
        matrizdePesos.append(n[i][1])

#Ahora nos creamos una matriz con n+1 filas y columnas


    matriz = [[0 for i in range(len(n)+1)] for j in range (len(n)+1)]

#Aqui metemos en cada fila los valores de los pesos.
    for i in range(len(n)+1):
        for j in range(len(matriz)):
            matriz[i][j]=matrizdePesos[j]

#Aqui en cada valor donde j=i, ejemplo 3-3, ponemos un cero.
    for i in range(len(matriz)):
        for j in range(len(matriz)):
            if(j==i):

                matriz[i][j] = 0

    return matriz
```

En los comentarios queda bien claro cómo funciona, hacemos lo mismo para el de valores.

Más tarde editamos el elige camino, el elige nodo se queda igual

Captura 13:

```
def eligeCamino(matriz, matrizPesos, feromonas, pesoMochila):
    # El nodo inicial siempre es el 0
    camino     = [0]
    visitados  = [0]
    longCamino = 0

    # Elegir cada paso según los valores y las feromonas
    while len(visitados) < len(matriz):
        nodo      = eligeNodo(matriz, feromonas, visitados, camino)

      #  print("camino: ", longCamino)

        camino.append(nodo)
        longCamino += matriz[visitados[-1]][nodo]

        # for para calcular el peso de la matriz
        peso = 0
        for x in range(len(camino)):
                peso = peso + matrizPesos[0][camino[x]]
        if (peso > pesoMochila):
            camino.remove(nodo)
            longCamino -= matriz[visitados[-1]][nodo]

        visitados.append(nodo)


    return (camino, longCamino)
```

Como podemos comprobar se incluyen dos datos nuevos, el peso de la mochila y la matriz de pesos. Y se usan para resolver que datos quedan y que nodo ha de quitarse porque ha limitado con el peso.

Ahora modificaremos el método para definir hormigas:

Captura 14:

```
def hormigas(matriz, matrizPesos, iteraciones, pesoMochila):
    # Primero se crea una matriz de feromonas vacía
    n = len(matriz)
    feromonas = [[0 for i in range(n)] for j in range(n)]

    # El mejor camino y su longitud (inicialmente "infinita")
    mejorCamino     = []
    longMejorCamino = sys.maxsize

    # En cada iteración se genera una hormiga, que elige una camino,
    # y si es mejor que el mejor que teníamos, deja su rastro de
    # feromonas (mayor cuanto más corto sea el camino)
    hormiga = 1
    for iter in range(iteraciones):
      #  print ("Hormiga: ", hormiga)
        hormiga = hormiga + 1
        (camino,longCamino) = eligeCamino(matriz, matrizPesos, feromonas, pesoMochila)

        if longCamino <= longMejorCamino:
            mejorCamino     = camino
            longMejorCamino = longCamino

        rastroFeromonas(feromonas, camino, pesoMochila/longCamino)
        # En cualquier caso, las feromonas se van evaporando
        evaporaFeromonas(feromonas)

    # Se devuelve el mejor camino que se haya encontrado
    peso = 0
    for x in range(len(mejorCamino)):
            peso = peso + matrizPesos[0][mejorCamino[x]]
    return (mejorCamino, longMejorCamino, peso)
```

Como podemos comprobar, lo interesante es que el mejor camino se basa en los pesos, este método intermedio tiene el gran fallo que ignora los valores…

Pero nos da una solución válida, lee de fichero y nos calcula el valor de dicha solución.

Además hemos incluido una variable hormiga, para si quisiéramos imprimir por cual hormiga va el algoritmo, al final hemos incluido algunos “prints” para darle más oficialidad.

La solución que da es esta:

Captura 15:

```
 RESTART: C:\Users\Juanoide_El_puto_amo\Dropbox\IA\Trabajo de ia\03-ACO20160606v10 para valores.py
>>> Cargamos las matriz del txt
Llamamos a las hormigas para que resuelvan el problema
Numero de hormigas usadas:  50
Tiempo de ejecución: 0.0781500340 seconds.
Nuestra mochila puede cargar con  1000 kilos
Longitud del camino basado en los valores:  1350
SOLUCIÓN AL POBLEMA:  [12, 8, 9, 17, 2, 20, 1, 6, 10, 15, 4, 11, 5, 13, 3, 16, 19, 7]
PESO DE LA MOCHILA SOLUCIÓN:  988
VALOR DE LA MOCHILA SOLUCIÓN: 1350
```

2.3. Problema de la mochila con 10 objetos

Para este apartado usaremos dos cosas:

03-exemplo7b.txt

Este fichero que como podemos comprobar es el anterior con los 10 primeros objetos y el código siguiente:

04-algoritmo solo de valores.py

Si abrimos el código nos damos cuenta que lo primero que se modifica respecto al 03, es el elige camino que ahora tiene en cuenta el valor actual como bien refleja esta parte:

Captura 16:

```
# Elegir cada paso según los valores y las feromonas
while len(visitados) < len(matriz):
    nodo      = eligeNodo(matriz, feromonas, visitados, camino)

    camino.append(nodo)

    valorActual = 0
    for x in range(len(camino)):

            if x != 0:
                valorActual = valorActual + matrizSinTransformar[camino[x]-1][2]

    pesoActual = 0
    for x in range(len(camino)):
            pesoActual = pesoActual + matrizPesos[0][camino[x]]


    if (pesoActual > pesoMochila):
        camino.remove(nodo)

    else:
        longCamino = pesoActual / valorActual

    visitados.append(nodo)


return (camino, longCamino)
```

Como podemos observar ahora calcula el valor actual y lo tiene en cuenta a la hora de calcular el longcamino, y como en el anterior en elige nodo y elige camino entra la matriz de valores.

Teniendo ya una versión definitiva con una solución óptima:

Captura 17:

```
Hormigas:  9800
Hormigas:  9900
Hormigas:  10000
Numero de hormigas usadas:  10000
Tiempo de ejecución: 6.2597143650 seconds.
Nuestra mochila puede cargar con  14 kilos
Longitud del camino basado en la suma de pesos entre valores:  0.111111111111111
1
SOLUCIÓN AL POBLEMA:  [5]
PESO DE LA MOCHILA SOLUCIÓN:  11
VALOR DE LA MOCHILA SOLUCIÓN: 99
```

Si hubiéramos usado el fichero anterior con 20 objetos la solución sería esta:

Captura 18:

```
Hormigas:  9700
Hormigas:  9800
Hormigas:  9900
Hormigas:  10000
Numero de hormigas usadas:  10000
Tiempo de ejecución: 16.4511983395 seconds.
Nuestra mochila puede cargar con  14 kilos
Longitud del camino basado en la suma de pesos entre valores:  0.090909090909090
91
SOLUCIÓN AL POBLEMA:  [19, 10]
PESO DE LA MOCHILA SOLUCIÓN:  14
VALOR DE LA MOCHILA SOLUCIÓN: 154
```

Tengamos en cuenta que para ver por dónde va el algoritmo hemos dejado implementado el print hormigas pero aplicando que solo imprima de 100 en 100, con un método sencillo basado en el módulo de 100.

2.4. Problema mejorado

¿Qué hemos planteado para mejorar lo presentado?, pues hemos ido más allá y en vez de usar una matriz de los valores usamos una del peso partido por el valor, ¿porque no al revés?, porque nuestro problema genérico minimizaba y al ser fracciones pequeñas la más pequeña es la que tiene más valor y menor peso.

Para comprobar dicha mejora usaremos este fichero:

05-algoritmo mejorado fraccion entre valores y pesos.py

Como comentábamos este pequeño cambio sutil, provoca que tengamos una mejora sustanciosa y demostremos que es mucho más rápida que la anterior en encontrar una solución, necesita ya muchas menos hormigas.

Captura 17:

```
matrizdeValoresPesos=[0]
for i in range(len(n)):
    matrizdeValoresPesos.append((n[i][1])/(n[i][2]))
```

Simplemente con este pequeño cambio ya conseguimos una mejora muy grande, debido a lo citado anteriormente del número de hormigas que se reduce. En el punto 2.5 se pueden ver diferentes pruebas de ambas y comparándolas.

2.5. Pruebas del problema solución y el mejorado.

Aquí vamos a poner pruebas de los algoritmos anteriores y demostrar que efectivamente el algoritmo cambia:

Empezamos sacando datos del primer algoritmo sobre una prueba con 20 objetos para tener más o menos los resultados controlados, el fichero que vamos a usar para pruebas es el siguiente:

examplo7.txt

Todas las pruebas sobre el conjunto de 20 objetos intentarán optimizar un peso de 14 kilos.

Comenzamos con la primera versión del algoritmo, esta versión del algoritmo se basa en que los nodos representan el valor del objeto, sin tener en cuenta el peso, aunque a la hora de aplicar la dosis (para que esta sea correcta dependiendo de la calidad de la solución) si lo tenemos en cuenta.

Fichero algoritmo:

04-algoritmo solo de valores.py

Los primeros resultados que nos devuelve el algoritmo con pocas hormigas son los siguientes, los datos que nos muestran por pantalla son pintados de color azul:

Primera ejecución:

200 hormigas

```
>>> Cargamos las matriz del txt
Llamamos a las hormigas para que resuelvan el problema
Hormigas:  100
Hormigas:  200
Numero de hormigas usadas:  200
Tiempo de ejecución: 0.2500121593 seconds.
Nuestra mochila puede cargar con  14 kilos
SOLUCIÓN AL POBLEMA:  [5]
PESO DE LA MOCHILA SOLUCIÓN:  11
VALOR DE LA MOCHILA SOLUCIÓN: 99
```

Segunda Ejecución:

300 hormigas

```
>>> Cargamos las matriz del txt
Llamamos a las hormigas para que resuelvan el problema
Hormigas:  100
Hormigas:  200
Hormigas:  300
Numero de hormigas usadas:  300
Tiempo de ejecución: 0.2343902588 seconds.
Nuestra mochila puede cargar con  14 kilos
SOLUCIÓN AL POBLEMA:  [15]
PESO DE LA MOCHILA SOLUCIÓN:  11
VALOR DE LA MOCHILA SOLUCIÓN: 99
```

Tercera ejecución:

Con **1000 hormigas** sin dar la mejor solución:

```
>>> Cargamos las matriz del txt
Llamamos a las hormigas para que resuelvan el problema
Hormigas:  100
Hormigas:  200
Hormigas:  300
Hormigas:  400
Hormigas:  500
Hormigas:  600
Hormigas:  700
Hormigas:  800
Hormigas:  900
Hormigas:  1000
Numero de hormigas usadas:  1000
Tiempo de ejecución: 1.1719350815 seconds.
Nuestra mochila puede cargar con  14 kilos
SOLUCIÓN AL POBLEMA:  [1, 10]
PESO DE LA MOCHILA SOLUCIÓN:  12
VALOR DE LA MOCHILA SOLUCIÓN: 87
```

Para comprobar que efectivamente el algoritmo funciona pongamos 5000 hormigas: Incluso hemos tenido que repetir la prueba varias veces ya que debido a la aleatoridad de las hormigas con **5000 hormigas** el 50% de las veces no devuelve el mejor resultado que es la **mochila completa con los 14 kilos y el valor de la mochila 154.**

Cuarta ejecución:

```
Hormigas:  3300
Hormigas:  3400
Hormigas:  3500
Hormigas:  3600
Hormigas:  3700
Hormigas:  3800
Hormigas:  3900
Hormigas:  4000
Hormigas:  4100
```

Hormigas: 4200
Hormigas: 4300
Hormigas: 4400
Hormigas: 4500
Hormigas: 4600
Hormigas: 4700
Hormigas: 4800
Hormigas: 4900
Hormigas: 5000
Numero de hormigas usadas: 5000
Tiempo de ejecución: 5.3705339432 seconds.
Nuestra mochila puede cargar con 14 kilos
SOLUCIÓN AL POBLEMA: [10, 19]
PESO DE LA MOCHILA SOLUCIÓN: 14
VALOR DE LA MOCHILA SOLUCIÓN: 154

Procedamos ahora a probar con las pruebas del segundo algoritmo, recordemos que este algoritmo guarda en la matriz el peso divido entre el valor, para facilitar la elección de la hormiga.
Adjuntamos el fichero del algoritmo:

Fichero del algoritmo:

05-ACO20160608v13.py

Con **200 hormigas** nos devuelve lo siguiente:

Llamamos a las hormigas para que resuelvan el problema
Hormigas: 100
Hormigas: 200
Numero de hormigas usadas: 200
Tiempo de ejecución: 0.1580128670 seconds.
Nuestra mochila puede cargar con 14 kilos
Longitud del camino basado en la suma de pesos entre valores: 0.12
SOLUCIÓN AL POBLEMA: [17]
PESO DE LA MOCHILA SOLUCIÓN: 12
VALOR DE LA MOCHILA SOLUCIÓN: 100

Comprobamos que con unos **1000 hormigas** ya nos devuelve la solución optima:
Llamamos a las hormigas para que resuelvan el problema repetimos el proceso varias veces para comprobar la mejora y efectivamente da un buen resultado:

Hormigas: 100
Hormigas: 200
Hormigas: 300
Hormigas: 400
Hormigas: 500
Hormigas: 600
Hormigas: 700
Hormigas: 800
Hormigas: 900

Hormigas: 1000
Numero de hormigas usadas: 1000
Tiempo de ejecución: 0.5937829018 seconds.
Nuestra mochila puede cargar con 14 kilos
Longitud del camino basado en la suma de pesos entre valores: 0.09090909090909091
SOLUCIÓN AL POBLEMA: [19, 10]
PESO DE LA MOCHILA SOLUCIÓN: 14
VALOR DE LA MOCHILA SOLUCIÓN: 154

Incluso hemos comprobado que con 500 hormigas nos devuelve muchas veces la solución óptima, esto era impensable antes.

En el apartado 3.1 de esta memoria comprobaremos los resultados con el algoritmo que ajusta la dosis en base a resultados anteriores, como se indica en el apartado 4 del enunciado propuesto en la asignatura:

2.6. Pruebas con el algoritmo de fuerza bruta.

Aquí vamos a comparar con el algoritmo de fuerza bruta:

06-KNA20160607v4.py

Este algoritmo aplica con fuerza bruta el problema de la mochila, este es el algoritmo que hemos usado para poder comparar los resultados, a continuación haremos pruebas comparando todos los algoritmos.

Aportamos los valores soluciones para las pruebas que hemos realizado en esta memoria sobre 10 objetos, 20 objetos y 100 objetos.

10 objetos y un peso de 14 kilos la solución más óptima nos devuelve un valor de 154.
20 objetos y un peso de 14 kilos la solución más óptima nos devuelve un valor de 154.
100 objetos y un peso de 100 kilos la solución que nos devuelve este algoritmo es 1024.

3. Segunda Parte

3.1. Problema de la mochila mejorada por iteraciones

Ahora comprobaremos los resultados con el algoritmo que ajusta la dosis en base a

resultados anteriores. Este algoritmo se basa en alterar la dosis, evaluando la respuesta de hormigas anteriores, dando mejores dosis a los resultados que aportaran una mejor solución.

La versión del algoritmo sería la siguiente y sería la versión final del algoritmo (la que nos aporta mejores soluciones):

Fichero del algoritmo:

07-ACO20160615v15.py

3.2. Análisis comparativo con la mejora de iteraciones

Vamos a compararlo con los dos anteriores el normal y el mejorado, para ver la eficacia del cambio realizado.

Recordamos los resultados anteriores, para el primer algoritmo nos daba la solución con 5000 hormigas y el segundo algoritmo llegaba a la solución con 1000 hormigas a continuación vemos que resultado nos arroja este último algoritmo con **50 hormigas:**

Primera ejecución:

```
>>> Cargamos las matriz del txt
Llamamos a las hormigas para que resuelvan el problema
Numero de hormigas usadas:  50
Tiempo de ejecución: 0.0200054646 seconds.
Nuestra mochila puede cargar con  14 kilos
SOLUCIÓN AL POBLEMA:  [10, 19]
PESO DE LA MOCHILA SOLUCIÓN:  14
VALOR DE LA MOCHILA SOLUCIÓN: 154
>>>
```

Como podemos ver mejora bastante la solución con respecto a los algoritmos anteriores, podemos decir que esta versión del algoritmo sería la más óptima para este conjunto de pruebas.

A continuación repitamos todas las pruebas sobre 100 objetos y un peso de 100 kilos para observar los resultados, ya comprobamos con el algoritmo de fuerza bruta que la solución óptima era un valor de 1012.

- Comencemos con el algoritmo basado en valores:

  Resultado que nos ofrece para unas 10000 hormigas:

  ```
  Hormigas:  9400
  Hormigas:  9500
  Hormigas:  9600
  Hormigas:  9700
  Hormigas:  9800
  ```

Hormigas: 9900
Hormigas: 10000
Numero de hormigas usadas: 10000
Tiempo de ejecución: 172.4794726372 seconds.
Nuestra mochila puede cargar con 100 kilos
Longitud del camino basado en la suma de pesos entre valores:
0.14410480349344978
SOLUCIÓN AL POBLEMA: [52, 28, 25, 51, 80]
PESO DE LA MOCHILA SOLUCIÓN: 99
VALOR DE LA MOCHILA SOLUCIÓN: 687

La solución se mejora para 100.000 hormigas, pero sigue sin ser la mejor:

Hormigas: 99400
Hormigas: 99500
Hormigas: 99600
Hormigas: 99700
Hormigas: 99800
Hormigas: 99900
Hormigas: 100000
Numero de hormigas usadas: 100000
Tiempo de ejecución: 1645.9555885792 seconds.
Nuestra mochila puede cargar con 100 kilos
Longitud del camino basado en la suma de pesos entre valores:
0.12483912483912483
SOLUCIÓN AL POBLEMA: [57, 80, 8, 7]
PESO DE LA MOCHILA SOLUCIÓN: 97
VALOR DE LA MOCHILA SOLUCIÓN: 777

La solución está lejos de los 1012 que debería ser la más óptima.

- Probamos ahora con el algoritmo mejorado, el que basa la decisión de los nodos en dividir el peso entre el valor:

  Numero de hormigas usadas: 5000
  Tiempo de ejecución: 78.1101379395 seconds.
  Nuestra mochila puede cargar con 100 kilos
  Longitud del camino basado en la suma de pesos entre valores:
  0.09788092835519677
  SOLUCIÓN AL POBLEMA: [60, 30, 87, 15, 80, 40, 10, 20, 70, 90, 35, 100]
  PESO DE LA MOCHILA SOLUCIÓN: 97
  VALOR DE LA MOCHILA SOLUCIÓN: 991

  Está solución se acerca un poco más a la óptima.

  Repitamos la prueba con 15000 hormigas:
  Hormigas: 14600
  Hormigas: 14700
  Hormigas: 14800
  Hormigas: 14900
  Hormigas: 15000
  Numero de hormigas usadas: 15000
  Tiempo de ejecución: 242.1690187454 seconds.
  Nuestra mochila puede cargar con 100 kilos

Longitud del camino basado en la suma de pesos entre valores: 0.09881422924901186
SOLUCIÓN AL POBLEMA: [100, 90, 30, 15, 70, 40, 45, 75, 50, 10, 80, 5]
PESO DE LA MOCHILA SOLUCIÓN: 100
VALOR DE LA MOCHILA SOLUCIÓN: 1012

Esta vez nos ha devuelto la solución óptima, con lo cual demostramos que este algoritmo claramente, encuentra la solución antes si repetimos las pruebas salen resultados parejos.

- Probemos a continuación con el último algoritmo (ajuste la dosis en función de resultados anteriores):

Hormigas: 3600
Hormigas: 3700
Hormigas: 3800
Hormigas: 3900
Hormigas: 4000
Numero de hormigas usadas: 4000
Tiempo de ejecución: 61.8975832462 seconds.
Nuestra mochila puede cargar con 100 kilos
SOLUCIÓN AL POBLEMA: [45, 35, 50, 70, 60, 30, 10, 90, 95, 80, 40, 75]
PESO DE LA MOCHILA SOLUCIÓN: 100
VALOR DE LA MOCHILA SOLUCIÓN: 1012

Con 4000 hormigas nos ha dado la mejor solución.

Repetimos la prueba un par de veces más y comprobamos que efectivamente nos devuelve la mejor solución:

Hormigas: 3600
Hormigas: 3700
Hormigas: 3800
Hormigas: 3900
Hormigas: 4000
Numero de hormigas usadas: 4000
Tiempo de ejecución: 66.5798122883 seconds.
Nuestra mochila puede cargar con 100 kilos
SOLUCIÓN AL POBLEMA: [100, 20, 25, 70, 45, 5, 50, 90, 60, 95, 40, 30]
PESO DE LA MOCHILA SOLUCIÓN: 100
VALOR DE LA MOCHILA SOLUCIÓN: 1012

4. Tercera parte

4.1. Posibles mejoras.

4.2. Conclusiones

Las conclusiones que sacamos de este proyecto, es que cualquier algoritmo está en cuestión, el que no se pueda mejorar.

Trabajando en esta idea nos hemos percatado que con pequeños cambios en el código, que requerían mucho esfuerzo mental, conseguíamos mejoras sustanciales y el rendimiento y la velocidad aumentaban exponencialmente.

Te hace pensar que la naturaleza tiene algoritmos implícitos en ella usables en diferentes problemas de nuestra sociedad y que adaptándolos y mejorándolos conseguimos grandes cosas.

Todo es cuestión de percatarse o vislumbrar una mejora, plantearla, estudiarla, probar su implementación a pequeña escala para luego implementarla, para por ultimo valorar dicha mejora con la versión anterior.

Por lo que podemos deducir que nos ha demostrado que no solo el algoritmo de la hormiga era mejorable, sino que cualquier algoritmo estudiándolo y dedicándole un tiempo determinado seguramente se pueda mejorar sustancialmente.

4.3. Bibliografía

Hemos usado material o nos hemos inspirado en las siguientes fuentes para realizar este trabajo:

Artículos de la Wikipedia:
https://es.wikipedia.org/wiki/Algoritmo_de_la_colonia_de_hormigas

https://es.wikipedia.org/wiki/Problema_de_la_mochila

https://es.wikipedia.org/wiki/Python

Artículo aportado por la asignatura:
http://www.iiisci.org/journal/CV$/risci/pdfs/C821AF.pdf

Códigos en Github:
https://github.com/trevlovett/Python-Ant-Colony-TSP-Solver

https://github.com/pjmattingly/ant-colony-optimization

https://github.com/kzyma/MMKP_Heuristics

https://github.com/balarsen/Ants

Pseudocódigo:
http://liris.cnrs.fr/csolnon/publications/bioma04.pdf

Libro, Inteligencia Artificial Avanzada - Raul Benítez, Gerard Escudero:
https://drive.google.com/file/d/0B9TsLZzbZBEYbEFsbG1Sa25QYlU/view

Enlace a implementaciones del algoritmo en Python
https://pypi.python.org/pypi/ACO-Pants/0.5.2