Universidad Autónoma Metropolitana

Análisis y Diseño de Algoritmos

Proyecto final: Problema de la mochila por fuerza bruta y algoritmo genético

Toral Maldonado Rosa Guadalupe
2153045948

Martes 23 de julio de 2019

Introducción

Relajación problema de la mochila, problema de la mochila continuo

Existen algunos problemas cuyas variables son binarias; es decir, 0 o 1. Al hablar de la relajación en estos, suponemos que estas variables pueden tomar, en realidad, valores entre el 0 y el 1; así, en el problema de la mochila la relajación se hace mediante el vector binario:

$$maximizar \sum_{j=1}^{n} p_j x_j$$

$$s.a. \sum_{j=1}^{n} w_j x_j \leq C$$

$$0 \leq x \leq 1, j = 1, 2, 3, \dots, n$$

Donde p_j es el beneficio del objeto j y w_j es el volumen del objeto j. Y x_j es la porción del objeto que se va a tomar.

La primera sumatoria indica el beneficio que se obtiene de todos los objetos escogidos, y la segunda, es el volumen total que utilizan de la mochila. Prácticamente indica que se pueden incluir desde fracciones de objetos a la mochila hasta objetos enteros.

Algoritmo genético y fuerza bruta

El objetivo de un algoritmo genético es llegar a una posible solución para el problema. No da la solución exacta, si no aproximaciones. Este algoritmo se utiliza para problemas de maximización, normalmente, aunque también puede usarse para minimizar un problema.

Lo que hace es imitar la cadena natural de la vida: un individuo nace, se reproduce y algunas veces ocurre una mutación en los genes de las nuevas generaciones. Lo que se busca con el algoritmo genético, es que cada generación provea mejores individuos.

Cada individuo es un cromosoma y es simulado por una solución a nuestro problema y cada cromosoma tiene genes que son los parámetros de cada solución.

En el algoritmo de la mochila, lo que esperamos es que el algoritmo genético encuentre mejores alternativas para llenar la mochila con los objetos que más le beneficien sin que se pase del volumen de esta.

En el algoritmo de fuerza bruta se espera lo mismo; sin embargo, la diferencia radica en que en este algoritmo, las soluciones serán más aleatorias y se irá guardando la mejor de todas conforme vayan apareciendo.

En esta práctica se compararán los resultados, no en velocidad, si no en cuál de los dos algoritmos devuelve una solución más prometedora: que el beneficio sea máximo sin que se pase del volumen.

Desarrollo

Los algoritmos se hicieron en lenguaje Python y para las gráficas se utilizó Latex. Y para obtener los datos, se hicieron un total de 50 pruebas sobre cada algoritmo brincando de 100 en 100 iteraciones/generaciones.

Algoritmo de fuerza bruta

Lo que se hizo fue generar un arreglo con números aleatorios entre 0 y 0.5 de manera aleatoria; esto representa a cuánto de cada objeto nos vamos a llevar en la mochila. De esa primera solución generamos otras 10 soluciones modificando una posición en el arreglo, escogido de manera aleatoria, entre el valor que tenía originalmente y 1, también de manera aleatoria.

Se evaluó cuánto beneficio y cuánto volumen necesitó cada solución y de ellas se escogió la mejor para volver a obtener 10 soluciones y repetir los mismos pasos.

Método crea_soluciones

Este método se encarga de generar las 10 soluciones aleatorias de la mejor obtenida.

```
def crea_soluciones(A):
    soluciones = []
    for x in range(len(A)):
        j = randint(0, n)
        if A[j] == 1.0:
            x-=1
        else:
            #Cambia el valor en la posición seleccionada y guarda la solución.
            aux=A[j]
            A[j] = float("{0:.1f}".format(uniform(A[j], 1)))
            soluciones.append(A[:])
            #Regresa a su valor original.
            A[j]=aux

    return soluciones
```

Imagen 1. Código del método crea_soluciones

Método encuentra_mejor_solucion

Este método obtiene los volúmenes y los beneficios de cada solución y escoge a la mejor de ellas.

```python
def encuentra_mejor_solucion(soluciones, mejor_s, mejor_b, mejor_v):
    for sol in soluciones:
        #pone el volumen y el beneficio de cada arreglo a cero antes de calcularlo
        volumen_indiv = 0
        beneficio_indiv = 0

        for k in range(n): #calcula beneficios y volúmenes
            volumen_indiv+=V[k]*sol[k]
            beneficio_indiv+=B[k]*sol[k]

        #Encuentra al mejor de todos
        if ((volumen_indiv <= C) and (beneficio_indiv > mejor_b)):
            mejor_v = volumen_indiv
            mejor_b = beneficio_indiv
            mejor_s = sol[:]

    return mejor_s, mejor_b, mejor_v
```

Imagen 2. Código del método crea_soluciones

Método problema_mochila_fb

En este método se guarda la mejor solución para crearle las 10 soluciones. También hace las pruebas y muestra a los mejores cromosomas que ha obtenido.

```python
problema_mochila_fb(A):
mejor_beneficio = 0
mejor_volumen = 0
soluciones = []
AUX = A[:]

for prueba in range(1, 51):
    A = AUX[:]
    print("=======================Prueba {}: =======================".format(prueba))
    for i in range(100*prueba): #Crea diferentes soluciones para A
        soluciones = crea_soluciones(A)
        A, mejor_beneficio, mejor_volumen = encuentra_mejor_solucion(soluciones, A, mejor_beneficio, mejor_volumen)

        print("\nMejor solución ({}): ".format(i))
        print(" ", A)

        print("         Mejor volumen: ", mejor_volumen)
        print("         Mejor beneficio: ", mejor_beneficio)

        # if i == prueba*100-1:
        #   print("({}, {})".format(prueba*10, mejor_beneficio))
        input()

    #Elimina el contenido de las soluciones para la siguiente prueba
    del soluciones[:]

return A
```

Imagen 3. Código del método crea_soluciones

Algoritmo genético

Para poder crear el algoritmo para el problema de la mochila, se modificaron algunos métodos y variables del utilizado para el problema del viajero.

Aquí lo que se hizo fue generar un conjunto de 10 soluciones iniciales (población), se cruzaron algunos cromosomas tomando en cuenta una probabilidad de cruzamiento, que prácticamente indica si un par de cromosomas pueden o no cruzarse para generar hijos.

Un cromosoma en este problema de la mochila sería un arreglo, una solución a dicho problema; mientras que un gen sería la porción que se toma de cada objeto para llevarlo en la mochila.

También se tomaron varias de esas soluciones para poder crear mutaciones y cada mutación fue considerada un nuevo hijo.

Método de cruzamiento

Se hizo un cruzamiento básico para este problema: se toman dos soluciones (padres) y un punto de corte. El punto de corte es una posición tomada al azar de ambos padres que crea un corte en ambos arreglos haciendo que cada padre se parta en dos mitades: la primera empieza en la primera posición del arreglo y termina en la posición donde se hizo el corte; y la segunda, empieza en el punto de corte y termina en la última posición del arreglo.

Para generar los hijos, se toma la primera mitad del padre 1 y se le concatena la segunda mitad del padre 2; y para generar al segundo hijo se toma la primera parte del padre 2 y sele concatena la segunda parte del padre 1.

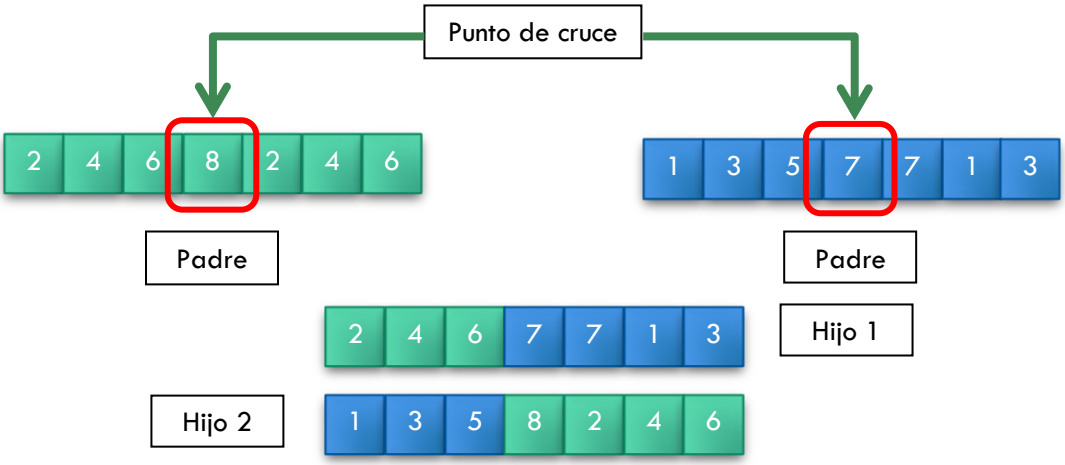

Imagen 3. Muestra de cómo se hace el cruzamiento

Método de mutación

Se toman dos genes al azar de una sola solución y se intercambia por un valor mayor entre el número actual que tiene y 1.

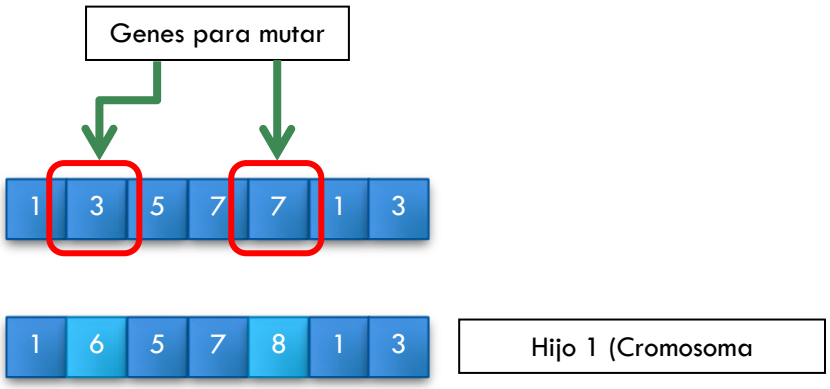

Imagen 4. Muestra de cómo se hace la mutación

En ambos métodos se obliga a que todos los hijos generados tengan un volumen por debajo de C. Así, si se crea alguno que no esté por debajo de éste límite, se vuelve a crear otro y el primero se desecha.

```python
def cruzamiento(padre1, padre2):
    n = len(padre1.solucion)
    hijo1 = Cromosoma()
    hijo2 = Cromosoma()
    while(True):
        punto_cruce = random.randint(1, n-1) #Para que no escoja el cero
        if(hijo1.volumen_indv>C):
            hijo1.solucion = padre1.solucion[:punto_cruce] + padre2.solucion[punto_cruce:]
            hijo1.evalua_fitnes(B, V, C)
        if hijo1.volumen_indv < C:
            break

    while(True):
        if(hijo2.volumen_indv>C):
            hijo2.solucion = padre2.solucion[:punto_cruce] + padre1.solucion[punto_cruce:]
            hijo2.evalua_fitnes(B, V, C)
        if hijo2.volumen_indv < C:
            break

    return [hijo1, hijo2]
```

```python
def mutacion(cromosoma):
    n = len(cromosoma.solucion)
    AUX = deepcopy(cromosoma)
    #Si el cromosoma obtenido de la mutación se pasa del volumen de la mochila, se desecha
    #y se crea otro
    while(True):
        a = 0
        b = 0
        cromosoma = deepcopy(AUX)
        while(a == b):
            a = random.randint(0, n-1)
            b =  random.randint(0, n-1)
        #Si alguno de los genes tiene un 1, se deja tal cual
        if cromosoma.solucion[a] != 1:
            gen = random.uniform(cromosoma.solucion[a], 1)
            cromosoma.solucion[a] = float("{0:.1f}".format(gen))
        if cromosoma.solucion[b] != 1:
            gen = random.uniform(cromosoma.solucion[b], 1)
            cromosoma.solucion[b] = float("{0:.1f}".format(gen))
        cromosoma.evalua_fitnes(B, V, C)
        if cromosoma.volumen_indv < C:
            break
    hijo = Cromosoma()
    hijo.solucion = cromosoma.solucion[:]
    hijo.evalua_fitnes(B, V, C)
    return hijo
```

Imagen 2. Métodos de cruzamiento (arriba) y de mutación (abajo)

Cromosoma

Cada cromosoma fue visto como un objeto con atributos una solución, un volumen y un beneficio, y con métodos *evalua_fitnes*, que evalúa el volumen y el beneficio de cada solución; y *crea_solucion_aleatoria* que crea un arreglo con números aleatorios entre 0 y 1. Nuevamente se consideró que si la solución creada no cumple con el volumen, entonces se desecha y se crea otra.

```python
class Cromosoma(object):
    def __init__(self):
        self.solucion = []
        self.volumen_indv = 0
        self.beneficio_indv = 0

    def genera_solucion_aleatoria(self, B, V, C): #Beneficios y volumenes tienen el mismo tamaño
        n = len(B)
        numero_genes = 0
        while numero_genes < n:
            #Inserta un número random entre 0 y 1 en el arreglo cromosoma
            self.solucion.append(float("{0:.1f}".format(random.uniform(0, 1))))
            if numero_genes == (n-1): #Ya lleno el arreglo
                self.evalua_fitnes(B, V, C) #Evalua el volumen total del
                if self.volumen_indv > C: #Si el volumen se paso, crea otra solucion
                    numero_genes = -1 #Por el incremento de "numero_genes", se pone a -1
                                      #para que pueda regresar a 0
                    self.volumen_indv = 0
                    del self.solucion[:]
            numero_genes = numero_genes + 1
        return self.solucion

    def evalua_fitnes(self, B, V, C):
        #Se ponen a 0, debido a que este algoritmo se usa para seleccionar las soluciones aleatorias
        self.volumen_indv=0
        self.beneficio_indv=0
        n = len(self.solucion)
        i=0
        while i < n:
            self.volumen_indv+=self.solucion[i]*V[i]
            self.beneficio_indv+=self.solucion[i]*B[i]
            i+=1
```

Imagen 5. Clase Cromosoma

Otras modificaciones

- **quicksort_GA:** se cambió la forma de ordenamiento de forma creciente a descendiente.
- **selecciona_progenitores:** se agregó una validación para que en "preseleccionados" guardara solamente aquellas soluciones cuyo volumen no excede a C.

Parámetros del algoritmo AG

Los parámetros que se usaron fueron los siguientes:

- numero_cromosomas = 10
- porcentaje_seleccionados = 90
- probabilidad_mutación = 0.3
- probabilidad_cruzamiento = 0.5

Resultados

Para obtener las gráficas, se tabuló el número de generaciones/iteraciones creadas contra el beneficio obtenido

Iteración/generación	Fuerza bruta	Algoritmo genético
100	103071.00000000001	139196.2
200	103132.20000000001	139197.1
300	109927.8	139149.80000000002
400	113827.8	139142.50000000003
500	127959.0	139218.2
600	132669.4	139218.30000000002
700	135697.4	139218.1
800	138725.40000000002	139253.40000000002
900	139431.40000000002	139207.80000000002
1000	139529.80000000002	139228.90000000002
1100	139529.80000000002	139258.2
1200	139529.80000000002	139260.3
1300	139529.80000000002	139261.1
1400	139529.80000000002	139264.90000000002
1500	139638.40000000002	139268.50000000003
1600	139638.40000000002	139257.6
1700	139638.40000000002	139256.00000000003
1800	139638.40000000002	139256.90000000002
1900	139638.40000000002	139251.50000000003
2000	139638.40000000002	139255.80000000002
2100	139638.40000000002	139258.2
2200	139638.40000000002	139243.2
2300	139638.40000000002	139284.80000000002
2400	139638.40000000002	139289.1
2500	139638.40000000002	139289.4
2600	139638.40000000002	139289.0
2700	139638.40000000002	139276.30000000002
2800	139638.40000000002	139259.7
2900	139638.40000000002	139263.50000000003
3000	139638.40000000002	139286.60000000003
3100	139638.40000000002	139286.90000000002
3200	139638.40000000002	139286.7
3300	139638.40000000002	139283.60000000003
3400	139638.40000000002	139292.90000000002
3500	139638.40000000002	139259.7
3600	139638.40000000002	139254.2
3700	139638.40000000002	139254.00000000003
3800	139638.40000000002	139250.9
3900	139638.40000000002	139266.6
4000	139638.40000000002	139263.2
4100	139638.40000000002	139278.40000000002
4200	139638.40000000002	139228.60000000003
4300	139638.40000000002	139315.2
4400	139638.40000000002	139315.2
4500	139638.40000000002	139310.2
4600	139638.40000000002	139305.00000000003

4700	139638.40000000002	139320.90000000002
4800	139638.40000000002	139327.1
4900	103071.00000000001	139330.40000000002

Tabla 1. Resultados de ambos algoritmos. Las iteraciones son para el algoritmo de fuerza bruta y su equivalente en el de genéticos es el número de generaciones.

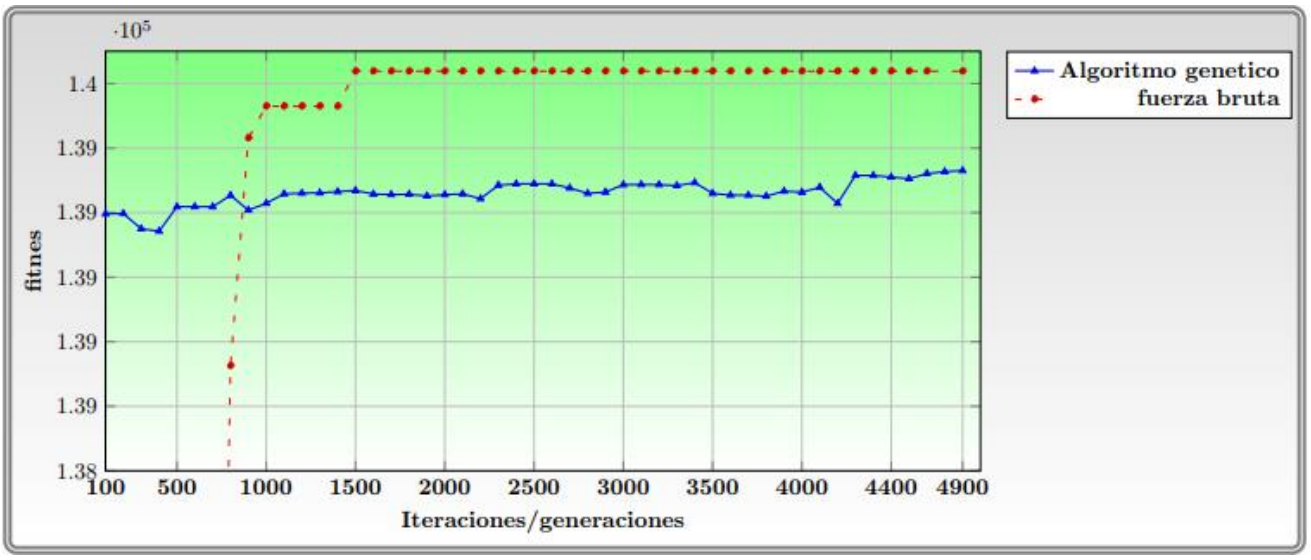

Imagen 6. Gráfica de resultados totales de ambos algoritmos.

```
Ingresa el valor de C:
341045
Volumenes:
[4912, 99732, 56554, 1818, 108372, 6750, 1484, 3072, 13532, 12050, 18440, 10972, 1940, 122094, 5558, 10630,
    2112, 6942, 39888, 71276, 8466, 5662, 231302, 4690, 18324, 3384, 7278, 5566, 706, 10992, 27552, 7548, 934,
     32038, 1062, 184848, 2604, 37644, 1832, 10306, 1126, 34886, 3526, 1196, 1338, 992, 1390, 56804, 56804,
    634]
Beneficios:
[1906, 41516, 23527, 559, 45136, 2625, 492, 1086, 5516, 4875, 7570, 4436, 620, 50897, 2129, 4265, 706, 2721,
    16494, 29688, 3383, 2181, 96601, 1795, 7512, 1242, 2889, 2133, 103, 4446, 11326, 3024, 217, 13269, 281,
    77174, 952, 15572, 566, 4103, 313, 14393, 1313, 348, 419, 246, 445, 23552, 23552, 67]



Generacion  99
     [0.4, 0.3, 0.3, 0.7, 0.2, 1.0, 0.1, 0.1, 0.8, 0.1, 0.2, 0.5, 0.5, 0.0, 0.2, 0.1, 0.9, 0.2, 0.9, 0.2, 0.2,
     0.1, 0.0, 0.3, 0.5, 0.5, 0.7, 0.5, 0.0, 0.2, 0.3, 0.2, 0.9, 0.5, 1.0, 0.3, 0.5, 0.6, 0.4, 0.8, 0.9, 0.1,
    0.7, 0.6, 1.0, 0.2, 0.0, 0.4, 0.2, 0.5]
        Volumen:  341033.20000000007
        Beneficio:  139096.59999999998

     [0.4, 0.3, 0.3, 0.7, 0.2, 1.0, 0.1, 0.1, 0.8, 0.1, 0.2, 0.5, 0.5, 0.0, 0.2, 0.1, 0.9, 0.2, 0.9, 0.2, 0.2,
     0.1, 0.0, 0.3, 0.5, 0.5, 0.7, 0.5, 0.0, 0.2, 0.3, 0.2, 0.9, 0.5, 1.0, 0.3, 0.5, 0.6, 0.4, 0.8, 0.9, 0.1,
    0.7, 0.6, 1.0, 0.2, 0.0, 0.4, 0.2, 0.5]
        Volumen:  341033.20000000007
        Beneficio:  139096.59999999998
```

```
Ingresa el valor de C:
341045
Volumenes:
[4912, 99732, 56554, 1818, 108372, 6750, 1484, 3072, 13532, 12050, 18440, 10972, 1940, 122094, 5558, 10630,
    2112, 6942, 39888, 71276, 8466, 5662, 231302, 4690, 18324, 3384, 7278, 5566, 706, 10992, 27552, 7548, 934,
     32038, 1062, 184848, 2604, 37644, 1832, 10306, 1126, 34886, 3526, 1196, 1338, 992, 1390, 56804, 56804,
    634]
Beneficios:
[1906, 41516, 23527, 559, 45136, 2625, 492, 1086, 5516, 4875, 7570, 4436, 620, 50897, 2129, 4265, 706, 2721,
    16494, 29688, 3383, 2181, 96601, 1795, 7512, 1242, 2889, 2133, 103, 4446, 11326, 3024, 217, 13269, 281,
    77174, 952, 15572, 566, 4103, 313, 14393, 1313, 348, 419, 246, 445, 23552, 23552, 67]



Arreglo inicial:
[0.3, 0.1, 0.2, 0.2, 0.2, 0.1, 0.2, 0.2, 0.1, 0.1, 0.2, 0.2, 0.3, 0.3, 0.2, 0.1, 0.0, 0.2, 0.0, 0.2, 0.0, 0.2,
     0.0, 0.1, 0.1, 0.1, 0.0, 0.1, 0.1, 0.0, 0.3, 0.1, 0.0, 0.3, 0.2, 0.0, 0.3, 0.2, 0.3, 0.1, 0.1, 0.1, 0.0,
    0.2, 0.3, 0.3, 0.1, 0.3, 0.2, 0.3]
======================Prueba 1: ======================

Mejor solución (0):
     [0.3, 0.1, 0.2, 0.2, 0.2, 0.1, 0.2, 0.2, 0.1, 0.1, 0.2, 0.2, 0.3, 0.3, 0.2, 0.1, 0.0, 0.2, 0.0, 1.0, 0.0,
     0.2, 0.0, 0.1, 0.1, 0.1, 0.0, 0.1, 0.1, 0.0, 0.3, 0.1, 0.0, 0.3, 0.2, 0.0, 0.3, 0.2, 0.3, 0.1, 0.1, 0.1,
    0.0, 0.2, 0.3, 0.3, 0.1, 0.3, 0.2, 0.3]
            Mejor volumen:  233085.80000000002
            Mejor beneficio:  96117.90000000001
```

Imagen 7. Impresiones que lanzan ambos algoritmos: genético(abajo) y de fuerza bruta (arriba)

Conclusiones

En ambas gráficas puede observarse que van mejorando con la cantidad de iteraciones que ejecuta cada algoritmo. Sin embargo, hay una vital diferencia entre ambos, pues el algoritmo genético va creando picos —sube en ciertos puntos y vuelve a bajar, incluso en valores más pequeños que en soluciones anteriores—. Y también desde un principio tiene una tendencia entre 139000 y 139400.

Esto puede ser debido a la misma naturaleza forzada del algoritmo, ya que desde un principio se ha impuesto que escoja únicamente las soluciones cuyo volumen es menor a C y al mismo tiempo tienen el mejor beneficio de entre todas las soluciones.

Esto puede verse mejor en el algoritmo de *mutacion* que cambia dos genes con valores mayores a los que tenía y que genera otra solución si la primera se pasó del volumen.

Y esto precisamente también puede afectar al algoritmo para que cree esos picos, pues algún cromosoma puede no ser uno de los mejores y por tener el volumen por debajo de C es considerado dentro de las soluciones. Y ese mismo cromosoma puede quedar como último de entre los mejores y afectar a la siguiente generación.

En cambio, en el algoritmo por fuerza bruta puede verse que va mejorando conforme se hacen más grandes las iteraciones; llegando a un punto en el que prácticamente se vuelve constante. También tiene variaciones entre 130000 y 139600.

Al dársele una sola solución y buscar la mejor de entre las 10 que se le generan la mejor de todas, hace directamente el trabajo lanzando muy buenos resultados desde un principio.

También puede verse que los valores del algoritmo de fuerza bruta llegan a ser mejores que los del genético.

Puede darse debido a que el de fuerza bruta de inmediato busca la mejor solución, mientras que el de genéticos da aproximaciones; y mientras más ejecuciones se den, mejores son los resultados.

Puede notarse entonces que el algoritmo genético, aunque sí obtiene mejores resultados, tiende demasiado al caos; es decir, que los valores no tienen un ritmo o una tendencia limpia como la del algoritmo por fuerza bruta.