Optimizar la distribución de cajas en containers (3D Bin Packing Problem)

Grupo 9: Jhon Vargas

I. Resumen:

Presentamos el problema de distribucion de cajas(items) en containers(bins) que es basado en el problema de “3D bin packing problem”. En el presente trabajo estudiamos 2 metaheuristicas PSO y GRASP estudiadas en clase y 1 heuristica relacionada al empaquetamiento de cajas dentro de un contenedor llamdo EMS(Empty Maximal Spaces). Tambien incluimos resultados y conclusiones del trabajo realizado.

II. Problem Descripcion:

El problema 3D de distribucion de cajas en contenedores (3D-BPP) es un problema de optimización de la investigación de operaciones conocido que modela muchas aplicaciones industriales de produccion y de transporte tales como la carga de contenedores, programación de programas de TV, carga y gestión de almacenes, etc.

En terminos generales el problema 3D-BPP considera empaquetar “m”cajas rectangulares de diferentes dimensiones en un contenedor rectangular. El presente trabajo se considera un problema de empaque de contenedores tridimensionales en el que las cajas de diferenetes volúmenes se empaquetan en un solo contenedor para maximizar el número de objetos empaquetados.

La **formulacion de matematica** del problema 3D-BPP es descrita en las siguientes lineas. La funcion objetivo es maximizar la cantidad de cajas(volumen) que se puede empaquetar en el contenedor(LxWxH):

$$Max \ \frac{\sum_i (l_i w_i h_i) x_i}{L*W*H} \ , i = 1, 2 \ldots k \ items \qquad (1)$$

Las principales restricciones que se tienen para este problema son las siguientes:

$$\textit{Capacity constraint}: \tag{2}$$
$$\sum_i c_i x_i \leqslant C_i$$

$$\textit{Non} - \textit{intersection between two items}: \tag{3}$$
$$a_{im} + a_{mi} + b_{im} + b_{mi} + c_{im} + c_{mi} \geq 1,\ i \neq m$$
$$x - axis:\ x_i + w_i(o_{i1} + o_{i4}) + l_i(o_{i2} + o_{i5}) + h_i(o_{i3} + o_{i6}) \leqslant x_m + M(1 - a_{im}),\ i \neq m$$
$$y - axis:\ y_i + w_i(o_{i1} + o_{i6}) + l_i(o_{i2} + o_{i3}) + h_i(o_{i4} + o_{i5}) \leqslant y_m + M(1 - b_{im}),\ i \neq m$$
$$z - axis:\ z_i + w_i(o_{i1} + o_{i2}) + l_i(o_{i3} + o_{i4}) + h_i(o_{i5} + o_{i6}) \leqslant z_m + M(1 - c_{im}),\ i \neq m$$

$$\textit{All items within the bin dimension}: \tag{4}$$
$$x - axis:\ (x_i + w_i(o_{i1} + o_{i4}) + l_i(o_{i2} + o_{i5}) + h_i(o_{i3} + o_{i6}))x_i \leqslant L_i$$
$$y - axis:\ (y_i + w_i(o_{i1} + o_{i6}) + l_i(o_{i2} + o_{i3}) + h_i(o_{i4} + o_{i5}))y_i \leqslant W_i$$
$$z - axis:\ (z_i + w_i(o_{i1} + o_{i2}) + l_i(o_{i3} + o_{i4}) + h_i(o_{i5} + o_{i6}))z_i \leqslant H_i$$

$$\textit{One orientation of each item}: \tag{5}$$
$$o_{i1} + o_{i2} + o_{i3} + o_{i4} + o_{i5} + o_{i6} = 1$$

Existen varias heuristicas que para resolver este problema de empaquetado de cajas en el conteniner tales como el Deepest Bottom Left with Fill (DBLF), Simple Deepest Bottom Left with Fill (S-DBLF), Improved Deepest Bottom Left with Fill (I-DBLF) y Empty Maximal Spaces(EMSs) que es la que usaremos para para nuestro trabajo.

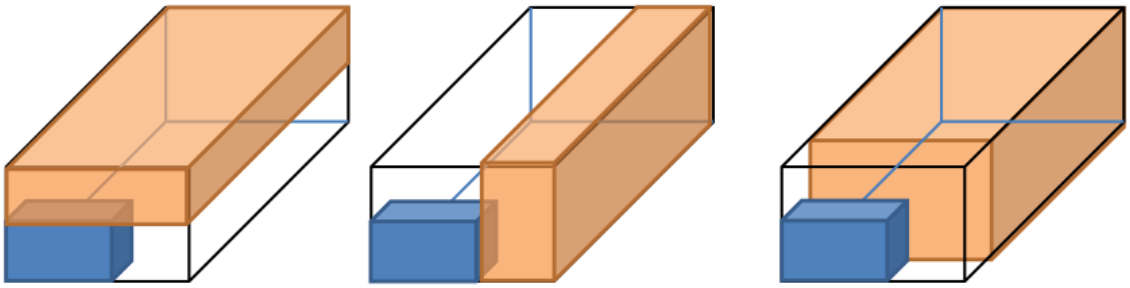

Fig1.

La **heuristica EMS** es esta asociado al espacio libre en el contenedor. Para definir el concepto usemos la Fig1. Como ejemplo. En esta figura podemos ver que la caja azul esta situada en el esquina inferior izquierda y genera 3 espacios EMS los cuales son representados por las cajas anaranjadas. Los maximos espcios son representados por las minimas y maximas coordenadas. Para definir la prioridad de espacios maximos EMS, se sigue con la regla de comparacion de los vertices de 2 espacios EMS.

La razón detrás de esta priorización es que queremos colocar las cajas primero en una esquina y sus lados adyacentes del contenedor, luego en su espacio interior. Ordenamos los espacios vacíos en cada contenedor de acuerdo con la prioridad definida anteriormente después de actualizar los EMS cada vez.

Para poder empaquetar las cajas en el contenedor tambien necesitamos conocer las diferentes orientaciones. En la Fig2., podemos ver las 6 diferentes orientaciones que podemos usar al momento de usar la heuristic EMS.

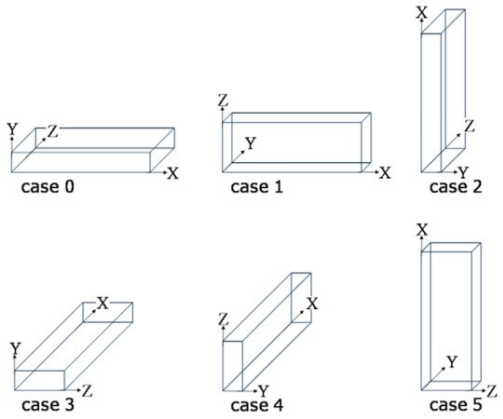

Fig2.

El pseudo-codigo de la heuristica EMS es la siguiente:

```
Input: Items and Bin
Output: A packing solution or null
While Container ≠ 0 do:
        i ← 1
        while Items ≠ 0 and boxplaced = false do:
                EMS list ← 0
                for i = 1 to kb do:
                        for all 6 orientations do
                                if Box can be placed in EMS with orientation then
                                        Add this placement combination to P
                        if P ≠ 0 then
                                Make the placement indicted by P
                                Update EMS list
                                boxplaced ← true
        if boxplaced = false then
                return null
return Packing solution
```

El **dataset de cajas** que usaremos para crear las cajas estan basados en segun las siguientes reglas:

Type	Length(L)	Width(W)	Height(H)
Class 1	[1 , L/2.0]	[2*W/3.0 , W]	[2*H/3.0 , H]
Class 2	[2*L/3.0 , L]	[1, W/2.0]	[2*H/3.0 , H]
Class 3	[2*L/3.0 , L]	[2*W/3.0 , W]	[1 , H/2.0]
Class 4	[L/2.0 , L]	[W/2.0 , W]	[H/2.0 , H]
Class 5	[1, L/2.0]	[1, H/2.0]	[1, D/2.0]

Tabla1.

Donde L, W y H son las dimensiones de largo, ancho y altura del contenedor respectivamente.

El enfoque que usaremos para usar las Meta-Heuristicas es de acuerdo al flujo siguiente:

1. **Dataset**: Creamos database de las cajas(items) de manera aleatorio segun la Tabla1.
2. **Init**: Añadimos las cajas al contenedor
3. **EMS**: Pasamos esta informacion a la heuristica EMS para validar que se puedan empaquetar en el contendor obteniendo las cajas factibles.
4. **Meta-Heuristica:** Luego corremos las Meta-Heuristicas(PSO o GRASP) y generamos nuevas soluciones en base a sus algoritmos.
5. **Normalizacion**: Luego normalizamos soluciones para que puedan ser tomadas nuevamente por los pasos 2 y 3
6. **Output**: Repetimos estos segun la cantidad de iteraciones indicadas o satisfacer las condiciones de parada.

Todo este flujo esta resumido en la siguiente figura Fig3.:

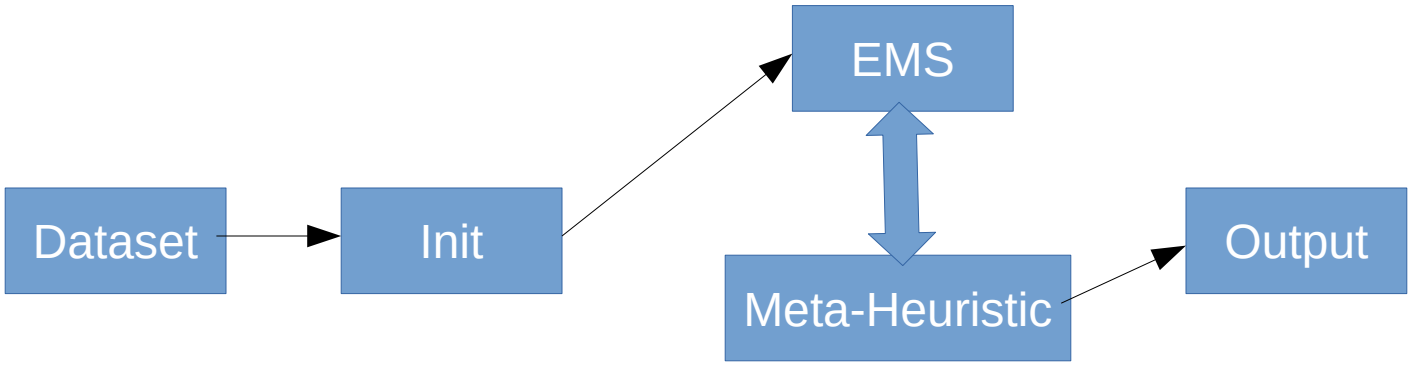

Fig3.

En los siguientes puntos pasaremos con la explicacion de las adaptaciones que fueron realizadas a las Meta-Heuristicas para poder aplicarlas al problem 3D-BPP.

III. Meta-Heurística PSO

La optimización por enjambre de partículas (PSO, Particle Swarm Optimization) hace referencia a una metaheurística que evoca el comportamiento de las partículas en la naturaleza. Este tecnica pertenece al grupo de los algoritmos basados en poblacion que esta inspirada en el comportamiento social.

PSO principalmente define 3 fuerzas o factores:

1. **Cognitiva**: Asociada a cada particula, conocimiento sobre el entorno.
2. **Grupal**: Asociada al Lider, conocimiento historico o de experiencias anteriores.
3. **Inercia**: Usada para continuar buscando, experiencia de los individuos situados a los alrededores.

Estas 3 fuerzas la podemos resumir segun la ecuacion (7). Y en la ecuacion (8) podemos ver como avanza la particula dentro de su ambiente.

$$\text{Posicion de particula}: \quad x_i(t) = [x_{i1},\ x_{i2},\ x_{i3},\ x_{i4},\ ....,\ x_{in}] \tag{6}$$

$$\text{Funcion de vuelo}: \quad v_{i+1}(t) = wv_i(t) + c_1 r_1 (P_{best} - x_i(t)) + c_2 r_2 (G_{global} - x_i(t))) \tag{7}$$

$$\text{Actualizacion de posicion de particula}: \quad x_i(t) = x_i(t) + v_{i+1}(t) \tag{8}$$

La palabra enjambre hace referencia a grupos de particulas que crean su propia prespectiva y en cada una de esta estan presente las 3 fuerzas. Durante la busqueda cada particula conoce su posicion y la del resto y no pierde de cerca la posicion del lider(la mejor solucion haste el momento) para no alejarse de la direccion de busqueda. Para cada busqueda se tiene una velocidad v(t), la cual representa la direccion de busqueda y esta contiene las 3 fuerzas indicada lineas arriba.

Para el caso del problem 3D-BPP, debemos hacer unos cambios en el algoritmo PSO. Este cambio esta asociado al formtato de la posicion de la particula que esta asociado a los items que son pesos fijos que tenemos que saber cual es la combinacion perfecta para maximizar la funcion objetivo(1). Para esto hemos considerado pasar el vector de pesos x(i) a un formato binario y para esto usaremos una funcion de activacion como Sigmoid, Softmax , ReLu.

La formulacion del algoritmo PSO modificado esta plasmado en el siguiente pseudocódigo(parte core):

```
Input: Items and Bin
Output: A best packing solution or null

while not(condición de parada) do:
    While Container ≠ 0 do:
        i ← 1
        while Items ≠ 0 and boxplaced = false do:
            EMS list ← 0
            for i = 1 to kb do:
                for all 6 orientations do
                    if Box can be placed in EMS with orientation then
                        Add this placement combination to P
                if P ≠ 0 then
                    Make the placement indicted by P
                    Update EMS list
                    boxplaced ← true
        if boxplaced = false then
            return null
    return Packing solution

    While Packing solution ≠ 0 and function =1 do:
        if (function == 1):  # Sigmoid
            return 1 / (1 + np.exp(-x))
        elif (function == 2):  # Softmax
            expo = np.exp(x)
```

```
                expo_sum = np.sum(np.exp(x))
                return expo / expo_sum
        elif (function == 3):  # ReLu
                return np.maximum(0, x)


    function PSOBinary(user_options, algorithm_options):
        x = randomPosition
        v = velocity(w, c1, c2)
        x= x+v
    return best packing solution list
return best packing solution
```

El flujo de trabajo esta plasmado en la Fig4.:

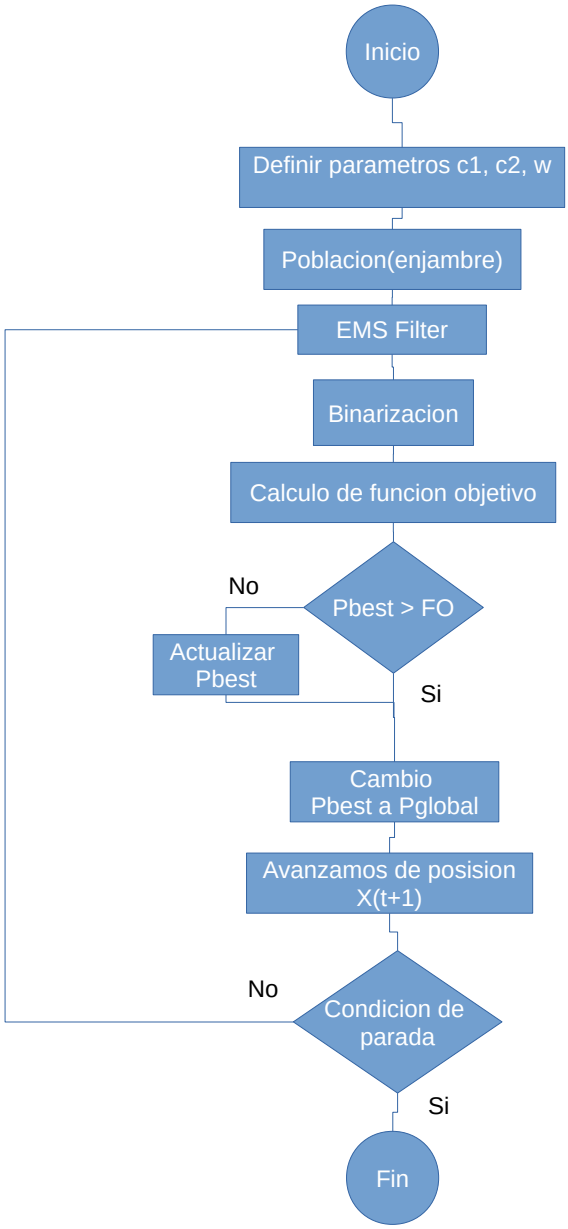

Fig4.

Los resultados generados para las siguientes escenarios la podemos ver en la Tabla2.

# Items	Bin Capacity	Time Avarage
10	408046.73	10.90ms
20	988268.33	13.44ms
25	998470.67	15.38ms
35	921697.06	18.50ms

Tabla2.

En la Fig5. Podemos ver el resultado en la Iteracion de 25 Items:

In [63]:
```
num_items = 25
array_items = []
array_item_dim = []

for i in np.arange(num_items):
    #posicion[i] = random.uniform(limites_inf[i],self.limites_sup[i])
    x = random.uniform(clase1_L[0],clase1_L[1])
    y = random.uniform(clase1_W[0],clase1_W[1])
    z = random.uniform(clase1_H[0],clase1_H[1])
    array_items.append([x,y,z])
    array_item_dim.append(x*y*z)

n_items = 25
dimensions = n_items
iteracions = 100
n_particles = 100

PSO()
```

```
1373847.5650669215 45

Iteraciones Solucion
1117282.460194356 46

Iteraciones Solucion
817421.989365662 47

Iteraciones Solucion
900472.4824176985 48

Iteraciones Solucion
1092802.8148520337 49
18

PSO-Binario - Time Execution: 15.38ms (using NumPy)

PSO-Binario - Solution: 998470.67 (using NumPy)

PSO-Binario - Posicion (using NumPy): [1 0 0 0 0 0 1 1 0 0 0 0 1 0 0 0 1 0 0 1 1 0 0 0 0]
```

Fig5.

IV. Meta-Heurística GRASP

La metaheurística GRASP (Greedy Randomized Adaptive Search Procedure) es un algoritmo que se aplica comúnmente a los problemas de optimización combinatoria. Como muchos métodos constructivos, la aplicación de GRASP consiste en crear una solución inicial y luego realizar una búsqueda local para mejorar la calidad de la solución. Su diferencial frente a otros métodos radica en la generación de esta solución inicial, basada en las tres primeras iniciales de sus siglas en inglés: codicioso (Greedy), random (Randomized) y adaptive (Adaptive).

Esta tecnica tiene 3 fases:

1. **Fase de procesamiento**: Identificacion de esquemas de alta calidad para encontrar una solucion inicial aceptable.
2. **Fase de Construccion**: Consiste en llevar la solucion hasta el limite de la factibilidad o infactibilidad.
3. **Fase de Busqueda**: Luego de encontrarnos en el limite de la factibilidad tenemos que hacer una busqueda(fija o variable) en la vecindad cercana hasta encontrar mejores puntos de mejor calidad.

Estos 3 fases se aplican en cada iteracion. Durante la fase de construccion, para recorer el camino podemos usar una heruristica basica y una **lista reducidad de candidatos**(LRC). Durante cada iteracion se guardan las mejores soluciones y estas se guardan en una **lista elite**.

Para el caso del problem 3D-BPP, debemos hacer unos cambios en el algoritmo GRASP. Este cambio esta asociado al formtato de la posicion de la particula que esta asociado a los items que son pesos fijos que tenemos que saber cual es la combinacion perfecta para maximizar la funcion objetivo(1). Para esto hemos considerado pasar el vector de pesos x(i) a un formato binario y para esto usaremos una funcion de activacion como Sigmoid, Softmax , ReLu.

La formulacion del algoritmo PSO modificado esta plasmado en el siguiente pseudocódigo(parte core):

```
Input: Items and Bin
Output: A best packing solution or null

while not(condición de parada) do:
        Items ← EMS Filter(Bin, Items)
        aceptable_solution = Greedy Search()
        while not(condición de parada) do:
                x=construccion_semi_greedy()
                x=búsqueda_local(x)
                If func_obj(x) > aceptable_solution
                        x* = x
                        aceptable_solution= func_obj(x)
        return best aceptable_solution list
return best aceptable_solution
```

El flujo de trabajo esta plasmado en la Fig6.:

Fig6.

En la Fig7. Podemos ver el resultado en la Iteracion de 10 Items:

```python
def grasp_runtime():
    iterator = 20
    while iterator <= 20:
        #file = open("entradas/input" + str(iterator) + ".in", "r")
        file = open("input" + str(iterator) + ".in", "r")

        iterator += 1
        weight = []
        value = []
        id = []
        i = 0
        n = int()
        capacity = int()
        for line in file:
            i += 1
            line = line.rstrip()
            numbers = re.findall("[0-9]+", line)
            if i == 1:
                n = int(numbers[0])
            elif 1 < i <= n + 1:
                id.append(int(numbers[0]) - 1)
                value.append(int(numbers[1]))
                weight.append(int(numbers[2]))
            else:
                capacity = int(numbers[0])

        max_iterations = 100
        s = "Instancia " + str(iterator - 1) + " : " + str(grasp(max_iterations, id, value, weight, capacity)) +
        print(s)
        #file = open("Output/grasp.out", "a+")
        #file = open("grasp.out", "a+")
        #file.write(s)
```

```python
grasp_runtime()
```

```
Instancia 20 : (654, 893921, 654, 0.007393360137939453)
```

Fig7.

Los resultados generados para las siguientes escenarios la podemos ver en la Tabla3.

# Items	Bin Capacity	Time Avarage
10	893921	7.39ms
20	905930	17.54ms
25	985268	33.91ms
35	971215	33.39ms

Tabla3.

V. Resultados y Conclusiones

1. Para poder elabrar una meta-heuristic modificada para el problema 3D-BPP se tiene que hacer un filtro antes de usar el algoritmo debido a que se necesita saber si los items escogidos son los se pueden empaquetar en el contenedor.
2. Para poder adaptar el problema de 3D-BPP a una meta-heuristic como PSO, se necesita hacer una adaptacion binaria a la posicion de las particulas para que puedan ser procesados por el algoritmo.
3. Todos los resultados son experimentales y aun falta mejorar la logica de adaptacion para tamaños mas grandes de items.

VII. Bibliografia

[1] Kang, K., Moon, I., Wang, H., 2012. A hybrid genetic algorithm with a new packing strategy for the three-dimensional bin packing problem. Applied Mathematics and Computation 219 (3), 1287–1299.
[2] G.R. Raidl, G. Kodydek, Genetic Algorithms for the Multiple Container Packing Problem, Parallel Problem Solving from Nature – PPSN V: Lecture Notes in Computer Science 1498 (1998) 875–884.
[3] Li, Xueping & Zhao, Zhaoxia & Zhang, Kaike. (2014). A genetic algorithm for the three-dimensional bin packing problem with heterogeneous bins. IIE Annual Conference and Expo 2014.
[4] Dube, Erick & Kanavathy, Leon & K@i, Leon & Za, Owave. (2006). Optimizing Three-Dimensional Bin Packing Through Simulation.