Trabalho Computacional I - Bin Packing Problem

Luiz Filipe Moraes, 112229
Lucas Eduardo Nogueira Gonçalves, 122055

17 de dezembro de 2020

1 Introdução

Este trabalho tem como objetivo apresentar uma comparação entre os resultados apresentados por dois diferentes métodos de solução do problema do empacotamento (Bin Packing Problem), um algoritmo baseado em uma heurística e um modelo de otimização.

Neste problema, itens de diferentes tamanhos devem ser guardados em recipientes com mesma capacidade; o objetivo é minimizar a quantidade de recipientes utilizados para guardar estes itens. De maneira mais formal, seja $I = \{I_1, \dots, I_n\}$ um conjunto de itens cujos volumes são dados por $v_1, \dots, v_n \in \mathbb{R}_{\geq 0}$, onde o v_i é o volume do item I_i para cada $i \in \{1, \dots, n\}$. Seja, também, $B = \{B_1, \dots, B_k, \dots\}$ um conjunto de recipientes iguais com volume V, queremos depositar todos os itens de I na menor quantidade possível de recipientes de B de maneira que, se $S_j \subseteq I$ é o conjunto de itens no recipiente B_j, então

$$\sum_{I_i \in S_j} v_i \leq V.$$

Obs.: Esta notação para o problema será utilizada durante todo o texto.

2 Métodos

2.1 Heurística

A heurística desenvolvida para resolver o problema consiste em guardar os itens sequencialmente, do item com maior volume ao menor, de modo que cada item seja guardado no recipiente que, ao comportar o item, tenha a menor quantidade de espaço disponível. Com isso, garantimos que cada um dos itens de I está em um recipiente e que nenhum dos recipientes está

sobrecarregado. Além disso, como o conjunto I é ordenado inicialmente, guardamos primeiro os itens grandes deixando espaços pequenos nos recipientes já utilizados para os menores itens, que serão guardados no final.

O algoritmo foi desenvolvido em python com o auxílio das bibliotecas numpy e math. No algoritmo as entradas foram criadas como na construção do problema, os inteiros V e n indicam, respectivamente , o volume dos recipientes e a quantidade de itens e o vetor v contém os pesos de cada item. Como variáveis auxiliares temos B representando a quantidade de recipientes utilizados e S um vetor cujo i-ésimo elemento indica o volume disponível no i-ésimo bin, portanto no início do programa B é nulo e cada entrada de S é igual a V. Esta parte inicial foi programada da seguinte maneira:

```
dados = np.loadtxt("Hard28_BPP13.txt")
V = int(dados[1])
B = 0
v = teste[2:dados.size].astype(int)
n = v.size
v = -np.sort(-v, kind='heapsort')
S = (np.zeros(n) + V).astype(int)
```

No trecho de código, "Hard28_BPP13.txt" é um dos exemplos utilizados no trabalho (ver 2.3). A ordenação dos itens de forma decrescente com relação ao volume foi feita utilizando-se a função sort da biblioteca numpy. O segundo passo do alogritmo consiste em colocar o primeiro item de v no primeiro recipiente disponível, este passo foi feito como a seguir:

```
S[0] -= v[0]
B += 1
```

o que indica que o volume do primeiro item de v foi subtraído do volume disponível do primeiro recipiente e a quantidade de recipientes utilizados B aumentou em 1. A parte final do algoritmo é colocar cada um dos $n-1$ itens restantes nos próximos recipientes, este passo foi programado como mostra o trecho a seguir:

```
for i in v[1:n]:
    R = np.array([math.inf, -1])
    for j in range(B):
        if R[0] > (S[j] - i) and (S[j] - i) >= 0:
            R[0] = S[j] - i
```

```
            R[1] = j
    if R[0] == math.inf:
        S[B] = S[B] - i
        B = B + 1
    else:
        S[int(R[1])] = S[int(R[1])] - i
```

Neste trecho, o loop for principal percorre o vetor v ignorando o primeiro item. Neste loop, para o i-ésimo item de v, é definido um vetor R que vai indicar na segunda coordenada qual vai ser a escolha ideal para guardar o item i e a primeira coordenada é inicialmente um valor alto (inf da biblioteca math). A seguir é iniciado um novo loop for que percorre todo o vetor S de recipientes. Para cada recipiente j é calculado qual será o volume disponível se o item i for guardado no recipiente j. Se este valor for menor que a primeira coordenada de R, então o recipiente j comportará o item i de maneira melhor que o recipiente indicado por R até o momento. Assim, caso a condição seja satisfeita, a primeira coordenada de R deverá ser substituída por $S[j] - v[i]$ e a segunda por j. Ao final do segundo loop for, se a primeira coordenada de R for inf, então nenhum recipiente já utilizado comporta o i-ésimo item, assim, este deverá ser colocado em um novo recipiente e B é acrescido em 1, caso contrário, o item é colocado no recipiente indicado por R. O valor final de B é a solução do método.

2.2 Modelo

Para construir o modelo, note que o objetivo do problema é minimizar a quantidade de recipientes utilizados. Assim, seja $Y = (y_1, \dots, y_k, \dots)$ um vetor onde para cada $k \in \mathbb{N}$, $y_k = 1$ se algum item de I está em B_k e 0 caso contrário, então nosso objetivo é minimizar a soma das coordenadas de y. Mas, como no pior caso cada item estará em um recipiente diferente, para facilitar a implementação do modelo, podemos considerar apenas os n primeiros recipientes de B. Assim, consideramos o escrevemos a função objetivo como

$$\text{Minimizar} \quad \sum_{k=0}^{n} y_k.$$

As restrições do modelo seguem do fato de que cada item deve estar em algum recipiente (e não pode estar em dois recipientes ao mesmo tempo) e do fato de que nenhum recipiente pode ficar sobrecarregado, ou seja, que a soma dos volumes dos itens guardados em cada recipiente não pode ultrapassar V.

Assim, considere a matriz $X \in \{0,1\}^{n\times n}$ onde para cada $i, j \in \{1,\dots,n\}$, o elemento $x_{i,j}$ de X é dado por

$$x_{i,j} = \begin{cases} 1 & \text{se o elemento } I_i \text{ está em } B_j \\ 0 & \text{caso contrário} \end{cases}$$

Com isso, podemos escrever a condição de que cada item deve estar em algum dos recipientes da forma

$$\sum_{j=1}^{n} x_{i,j} = 1, \quad \forall i \in \{1,\dots,n\},$$

pois esta soma indica em quantos recipientes está o i-ésimo item. A segunda restrição, de que nenhum recipiente pode ficar sobrecarregado, pode ser escrita da forma

$$\sum_{i=1}^{n} v_i x_{i,j} \leq V y_i, \quad \forall j \in \{1,\dots,n\}$$

Além dessas restrições, também vamos considerar que cada elemento de x e de y devem esta em $\{0,1\}$. Assim, podemos escrever o modelo como

$$\begin{aligned}
\text{Minimizar} \quad & \sum_{k=0}^{n} y_k & & \\
\text{s.a.} \quad & \sum_{j=1}^{n} x_{i,j} = 1, & & \forall i \in \{1,\dots,n\} \\
& \sum_{i=1}^{n} v_i x_{i,j} \leq V y_i, & & \forall j \in \{1,\dots,n\} \\
& \sum_{k=0}^{n} y_k \geq 1 & & \\
& y_k \in \{0,1\}, & & \forall k \in \{1,\dots,n\} \\
& x_{i,j} \in \{0,1\}, & & \forall i,j \in \{1,\dots,n\}
\end{aligned}$$

Para a implementação do modelo, foi utilizada a biblioteca PuLP, do python, para resolver sistemas lineares. As variáveis do problema foram definidas em vetores de variáveis binárias da seguinte maneira:

```
x = [LpVariable("xvar{0}_{1}".format(i, j), cat = "Binary")
↪  for i in range(n) for j in range(n)]
y = [LpVariable("yvar{0}".format(i), cat = "Binary") for i in
↪  range(n)]
```

O modelo e as restrições foram definidos como a seguir

```
# Modelo
modelo = LpProblem("BPP", LpMinimize)

# Função Objetivo
modelo += sum(y)

# Restrições
for k in range(n):
    modelo += sum(x[j] for j in range(n*k, (n*k)+n)) == 1
    modelo += sum(x[j]*v[l] for j,l in zip(range(k, n**2, n),
    ↪  range(n))) <= V*y[k]

modelo += sum(y) >=1
```

Onde n é quantidade de itens, v é um vetor de tamanho n contendo os volumes de cada item e V é o volume de cada recipiente (assim como no modelo criado). No loop for, a primeira linha indica que a soma das entradas de cada coluna da matriz x deve ser sempre igual a 1 e a segunda linha representa a restrição de que nenhum recipiente deve estar sobrecarregado. O modelo foi resolvido com as ferramentas da própria biblioteca PuLP e foi utilizado um tempo limite de 3600 segundos de execução.

2.3 Testes

Como determinado inicialmente no trabalho, contamos com um total de oito problemas que consistem em arquivos do tipo *.txt* contendo um conjunto de dois dados, o primeiro se refere ao volume de cada um dos recipientes e segundo está atrelados a quantidade de itens e o volumes de cada um que deve ser alocado. Todos os testes foram retirados da página do *Operations Research Group Bologna* e podem ser obtidos em http://or.dei.unibo.it/library/bpplib.

Os parâmetros de cada um dos problemas são :

	Problema									
	13	14	40	47	60	119	144	175	178	181
Volume	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000
Itens	180	160	160	180	160	200	200	200	200	180

3 Resultados e Conclusão

	Problema									
	13	14	40	47	60	119	144	175	178	181
Heurística	68	62	60	72	64	77	74	84	81	73
Modelo	73	66	61	78	68	81	77	90	85	76
Solução	67	62	59	71	63	77	73	84	80	72

Como pode ser visto na tabela anterior, entre os dois métodos apresentados, o que mais se aproxima, ou se iguala, das soluções ótimas (obtidas no site do *Operations Research Group Bologna*) em todos os problemas propostos é a heurística. Mais ainda, temos que o tempo de execução da heurística é bem inferior ao tempo do modelo resolvido pelo PuLP. Na maioria dos casos a heurística, não chegou a exceder o tempo de 2 segundos, ao contrário do que acontecia no modelo resolvido pelo PuLP, onde para obter uma solução na casa das centenas era necessário pelo menos dois minutos de execução.

Podemos ressaltar também que os problemas em que o modelo resolvido pelo PuLP se saiu melhor, teve um resultado mais próximo do ótimo, são os casos em que a quantidade de itens é menor. Isso se dá pelo fato de que este é um problema da classe NP.

De modo geral, podemos concluir que a heurística obteve resultados mais satisfatórios, onde a diferença entre a solução ótima e a solução obtida não é maior que um recipiente. Além disso, temos que esta é computacionalmente menos custosa, acelerando assim seu tempo de execução.