O problema de empacotamento de conjuntos
Relatório

Bruno Ramos Lima Netto

Outubro 2020

Universidade Federal
do Rio de Janeiro

Abstract: Nesse trabalho, relato os estudos e procedimentos realizados para encontrar uma solução do problema de Empacotamento de conjunto ou *Set packing* através de métodos como a *Relaxação Lagrangiana*, através de um algoritmo baseado no método do subgradiente. Além disso, um método *Branch-and-bound* baseado no princípio de explorar a árvore de problemas, tomando aproximações sucessivas para as entradas não inteiras das soluções de problemas relaxados. E o método do *Branch-and-cut*, através da adição de planos de corte, induzindo uma faceta do fecho convexo no espaço das soluções viáveis, a partir de planos de cliques encontrados no grafo de conflito relacionado. Apresento também as ideias por trás da implementação desses métodos, bem como testes em instâncias de diversas magnitudes afim de explorar onde cada um é mais ou menos eficiente.

0 Introdução

0.1 O trabalho

A estrutura desse trabalho é a seguinte: nessa sessão, apresento as ideias e modelagens realizadas em relação ao problema do empacotamento de conjuntos ou *Set Packing*. Nas sessões 1,2,3 comento sobre os métodos e algoritmos implementados e testados para a solução de instâncias de tal problema. Na sessão 4 e 5 disserto sobre os experimentos com tais algoritmos e realizo a conclusão dos resultados, bem como mostro alguns métodos alternativos interessantes para a solução ou aproximação de tal problema. Todo código utilizado se encontra no notebook do jupyter. Para a facilidade do usuário, transcrevi as funções para um arquivo .jl também.

0.2 O problema

No ano de 1971, Stephen Cook demonstrou em [3] que o problema da satisfatibilidade , conhecido como **SAT** se tratava de um problema **NP-completo**, Richard Karp publicou em [7], no ano seguinte, uma sequência de 20 problemas que poderiam ser reduzidos no problema de **SAT**. Concluindo que, todos eles eram da mesma classe de complexidade. Abaixo segue uma figura que aparece em tal publicação, com a relação de cada problema e suas reduções até chegar no problema **SAT**:

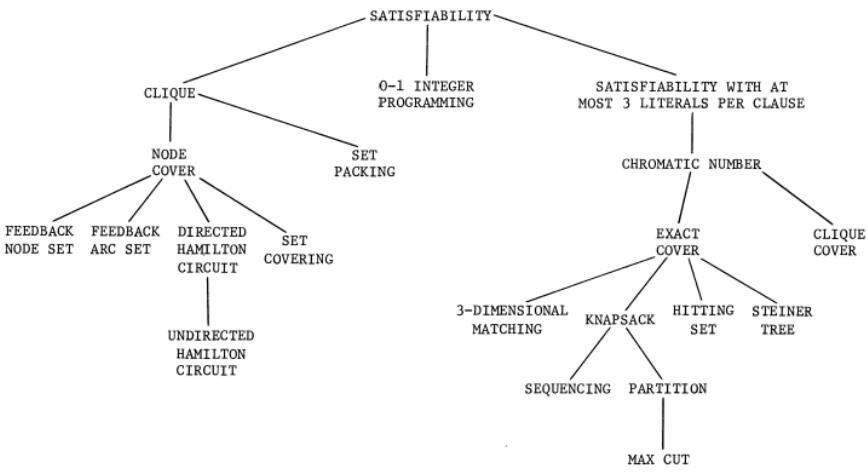


FIGURE 1 - Complete Problems

96

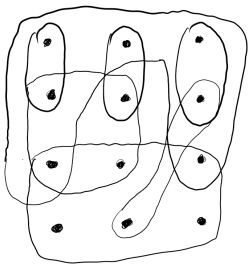
RICHARD M. KARP

Podemos observar que nessa lista aparece o nosso problema de interesse: o Set Packing. Na formulação de Karp, ele possui o seguinte enunciado:

Set Packing - Karp(1972)

Dada uma família de conjuntos $\{S_j\}$, e um inteiro k. Queremos saber se existem k conjuntos mutuamente disjuntos contidos em $\{S_j\}$.

Podemos ilustrá-lo com a seguinte imagem, que contém um conjunto universo de 12 pontos e 7 subconjuntos desses 12 pontos representando cada um dos S_j do nosso enunciado.

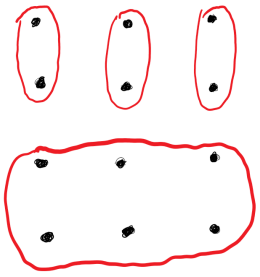


Assim, podemos nos perguntar para os inteiros $1 \leq k \leq 7$ se existem j conjuntos mutuamente disjuntos contidos em $\bigcup_j S_j$. Além disso, podemos apenas estar interessados no maior valor de k que tal propriedade seja satisfeita. Podendo então formular o problema como um problema de otimização com o seguinte enunciado:

Set Packing - Otimização

Qual o maior inteiro k tal que para uma dada uma família de conjuntos $\{S_j\}$, existam k conjuntos mutuamente disjuntos contidos em $\{S_j\}$.

Para tal instância pequena, podemos conferir que a solução é dada por $k = 4$ com a seguinte configuração:



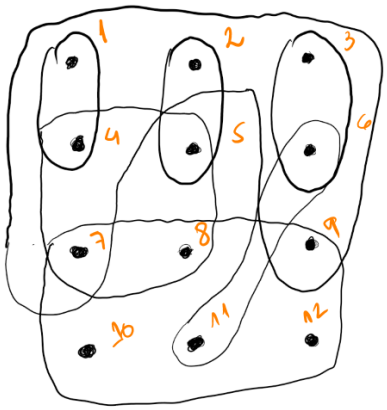
Para modelar o problema iremos considerar uma matriz $A_{I\times J}$, onde I é o número de pontos e J é o número de subconjuntos considerados pela instância. Bem como, iremos considerar uma variável binária $x \in \mathbb{B}^J$, onde cada entrada $x_i = 1$ nos indica que o subconjunto S_i pertence à solução e $x_i = 0$ caso contrário. Assim, podemos chegar na seguinte formulação que tem o formato de uma problema linear binário:

$$\begin{aligned} \max \quad & \sum_{j=1}^{J} x_j \\ \text{s.a.} \quad & \sum_{j=1}^{J} a_{ij}x_j \leq 1, \forall i \in [I] \\ & x \in \{0,1\}^J \\ & a_{i,j} \in \{0,1\}^{I\times J} \end{aligned} \tag{1}$$

Nesse trabalho, entretanto, estaremos considerando uma versão ponderada do problema. O que será conferido atribuindo um custo c_j para cada um dos subconjuntos. Assim, consideramos como função objetivo, maximizar o total dos custos e chegamos na seguinte formulação para o problema do *Set Packing*:

$$\begin{aligned} \max \quad & \sum_{j=1}^{J} c_j x_j \\ \text{s.a.} \quad & \sum_{j=1}^{J} a_{ij}x_j \leq 1, \forall i \in [I] \\ & x \in \{0,1\}^J, c \in \mathbb{R}_+^J \\ & a_{i,j} \in \{0,1\}^{I\times J} \end{aligned} \tag{2}$$

Assim, nossa instância exemplo pode ser modelada pela seguinte matriz:




```
12×7 Array{Int64,2}:
 1  0  0  0  1  0  0
 0  1  0  0  1  0  0
 0  0  1  0  1  0  0
 1  0  0  1  1  0  0
 0  1  0  1  0  0  0
 0  0  1  0  1  1  0
 0  0  0  1  1  0  1
 0  0  0  1  0  0  1
 0  0  0  0  1  0  1
 0  0  0  0  0  0  1
 0  0  0  0  0  1  1
 0  0  0  0  0  1  1
```

0.3 Os métodos: uma visão geral

Para realizar os estudos sobre esse problema, com essa formulação, utilizei de alguns métodos para aproximar e estimar a solução ótima. Tendo em vista que temos um problema **NP-difícil**, não existe um algoritmo polinomial para resolvê-lo. Entretanto para as instâncias que foram consideradas, podemos resolver com precisões arbitrárias com algoritmos de hoje em dia, devido ao seu tamanho relativamente pequeno. Nesse trabalho, realizei a implementação e teste dos seguintes métodos de solução do problema:

- Relaxação Lagrangiana, através do método do subgradiente;
- Branch and Bound;
- Branch and Cut.

Tratarei nas sessões seguintes com mais afinco sobre detalhes das suas implementações e escolhas de parâmetros.

1 Relaxação Lagrangiana

O primeiro método implementado foi um método do subgradiente que resolve sucessivas vezes o problema dualizado. Primeiramente, iremos considerar a seguinte formulação dualizada do problema:

$$\begin{aligned} \max \quad & \sum_{j=1}^{J} c_j x_j + \sum_{i=1}^{I} \mu_i (1 - \sum_{j=0}^{J} a_{ij} x_j) \\ \text{s.a.} \quad & \\ & x \in \{0,1\}^J, c \in \mathbb{R}_+^J, \mu \in \mathbb{R}_+^I \\ & a_{i,j} \in \{0,1\}^{I \times J} \end{aligned} \tag{3}$$

Assim estamos atribuindo para cada uma das I restrições um multiplicador de *Lagrange* correspondente. A ideia do algoritmo é iniciarmos com um vetor μ e iremos reduzindo ele na direção de um subgradiente do problema multiplicado por um passo que pode alterar ou não. E isso acontece até atingirmos uma precisão na diferença entre estimações por baixo e por cima da solução ou até atingirmos um numero máximo de iterações. O pseudocódigo tem a seguinte forma, recebendo como *input* a matriz A e um vetor de custos c:

Data: $A_{I \times J}, c \in \mathbb{R}_+^I$
iter = 0;
u = new_u;
while $iter \leq maxiter$ ***or*** $|L_b - U_b| \geq tol$ **do**
  $u \leftarrow u - step * g$;
  nU_b = **Solve** $\mathcal{L}(u)$;
  iter += 1 ;
  if $nU_b \leq U_b$ **then**
    $U_b \leftarrow nU_b$;
    $L_b \leftarrow updt(x)$;
  end
  $g \leftarrow new\_g$;
  $step \leftarrow new\_step$
end

Algorithm 1: Rotina do Subgradiente

Observe que nesse algoritmo teremos o vetor u, representando os multiplicadores de *Lagrange*, os números U_b e L_b que representam limitantes superiores e inferiores respectivamente, o problema dualizado representado por $\mathcal{L}(u)$, g que representa um subgradiente e *step* que é o tamanho do passo da incrementação dos multiplicadores. Agora irei explicar como que cada uma dessas variáveis se comporta.

Primeiro inicializamos um contador de iterações *iter* para termos como uma condição de parada o numero de iterações do algoritmo sob controle. Bem como inicializamos o vetor u. Fiz alguns testes com o vetor u apresentado pelo artigo [5], dado por:

$$u_j^0 = \frac{\sum_i a_{ij} \frac{c_i}{\sum_k a_{kj}}}{\sum_p a_{pj}},$$

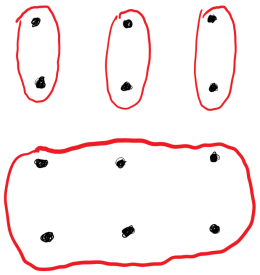
entretanto, na maioria dos testes o vetor inicial dado por $u = \bar{\mathbf{1}}$ foi o suficiente para convergência do algoritmo. Além disso, começamos com os valores $U_b = +\infty$ e $L_b = 0$, os quais iremos melhorando a cada iteração do algoritmo até eles se aproximarem de uma certa tolerância *tol* dada. Os valores de U_b serão obtidos resolvendo o problema dual a cada iteração, já os de L_b são obtidos através de uma heurística gulosa, que irá ordenar os subconjuntos em ordem crescente de pesos e tentando se aproximar do valor máximo não restrito a cada iteração.

O valor do passo de incrementação dos multiplicadores de *Lagrange* pode ser de várias maneiras. Para instâncias que convergiam rapidamente, tomar $step\frac{1}{iter}$ era o suficiente para garantir tal convergência. Entretanto, para instâncias mais complexas tanto em tamanho quanto em densidade, tomar um passo pequeno e constante, na ordem de $\frac{1}{I \times J}$ trouxe resultados muito satisfatórios.

No nosso exemplo pequeno, dado pela sessão anterior, considerando um vetor de custos $c = \bar{\mathbf{1}}$, temos a seguinte solução dada pelo método aqui apresentado:

```
Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    1.2000000e+01   0.000000e+00   0.000000e+00      0s

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       1    7.0000000e+00   0.000000e+00   0.000000e+00      0s

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       2    4.5000000e+00   0.000000e+00   0.000000e+00      0s

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       3    4.3333333e+00   0.000000e+00   0.000000e+00      0s

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       4    4.0333333e+00   0.000000e+00   0.000000e+00      0s

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       5    4.0000000e+00   0.000000e+00   0.000000e+00      0s

Out[160]: ([1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 1.0], 4.0)
```

Ou seja, em apenas 6 iterações (observe que começamos na iteração 0, como consta na primeira coluna), e em um tempo ínfimo, observado na última coluna, obtemos a solução esperada, constituida dos conjuntos 1,2,3 e 7. Representando os conjuntos em vermelho a seguir:



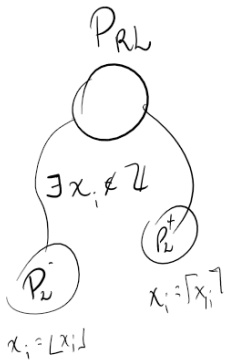
2 Branch and Bound

A ideia desse método está baseada na exploração de uma árvore de subproblemas relacionados, com um espaço de soluções viáveis cada vez menor, até obtermos uma solução que garantimos que é a melhor possível ou limitantes superiores e inferiores o tão perto quanto a gente queira.

Começamos definindo a raiz dessa árvore, que será o problema com suas restrições linearmente relaxadas. Assim, temos um simples problema linear que pode ser rapidamente resolvido, nos dando uma resposta que pode ou não ser inteira. Caso ela seja inteira, isso implica que temos uma solução que é o ótimo global para o problema inteiro original e retornamos essa solução. Caso contrário começamos o processo de *Branching* ou ramificação.

O processo consiste em percorrer a nossa solução do problema relaxado até encontrarmos a sua entrada x_i que está mais longe de um número inteiro que qualquer outra. Ou seja, queremos x_i mais próximo de 0.5 possível e iremos construir dois problemas relacionados ao primeiro. O primeiro problema, P^+ terá a restrição que $x_i = 1$, arredondando x_i pelo seu teto. O seguinte, P^- irá arredondar $x_i = 0$, seu piso.

Assim, nossa árvore terá a seguinte aparência, após essa ramificação:



Agora, iremos escolher o próximo nó da árvore para relaxarmos linearmente e iterar o processo. Observe que alguma hora pode acontecer de encontrarmos um nó com um problema que possua como conjunto viável o conjunto $\varnothing$. Quando isso acontecer, iremos obter um ramo da árvore que chegou ao seu fim e, portanto, não precisa mais ser explorado. Outra coisa que pode acontecer, é que eventualmente pode ser que obtenhamos uma solução inteira. Isso irá nos conferir um limitante superior para a resposta do problema original. Assim sendo, iremos chegando cada vez mais perto dos nossos critérios de parada do algoritmo. A ideia do método é percorrer essa árvore resolvendo os problemas

relaxados, que são fáceis de se resolver, de maneira que resolvemos o mínimo de nós possível para nos dar uma resposta desejada.

Um pseudocódigo para tal método tem a seguinte forma:

Data: $A_{I\times J}, c \in \mathbb{R}_+^I$
nodes = $[\mathcal{P}_{\mathcal{RL}}]$;
while *nodes* $\neq \varnothing$ *or* $\mid U_b - L_b \mid \geq tol$ **do**
  $p \leftarrow nodes[1]$;
  $p_{sol} =$ **Solve**(p);
  if $p_{sol} \notin \mathbb{Z}$ **then**
    $p_- \leftarrow \lfloor p_{sol} \rfloor$;
    $p_+ \leftarrow \lfloor p_{sol} + 1 \rfloor$;
    nodes $\leftarrow p_+, p_-$;
  end
  if *Infeasible* **then**
    nextnode;
  end
  Temos upperbounds!;
end

Algorithm 2: Rotina do Branch and Bound

Observe que como entrada do algoritmo temos as mesmas do método anterior. Começamos ele definindo o nosso primeiro nó, contendo o problema original com sua relaxação linear. Enquanto existir algum nó para explorar ou nossos limitantes superiores e inferiores ainda estão distantes, iremos escolher o próximo nó da árvore, resolvê-lo, calcular novos limitantes, ramificar se for necessário e repetir o processo.

Para instâncias cuja solução do problema relaxado linear é inteira, o código proposto funciona perfeitamente. Para instâncias mais complicadas, tive dificuldade em programar a DFS que é necessária para explorar tal árvore dos problemas. Para contornar tal problema, abri mão do meu código e utilizei o algoritmo do solver do **Gurobi** com alguns parâmetros modificados. Podemos observar na foto a seguir, um pedaço do código do meu algoritmo Branch and Bound, enquanto mostrava o percurso da árvore, em cada linha que termina com *Branching* mostrando que ele é capaz de descer na busca. Entretanto ele sofre problemas na hora de retornar para um nó antigo e explorar um outro nó.

```
Solved in 10 iterations and 0.00 seconds
Optimal objective  1.145000000e+02
----------> Any[1] <--------- BRANCHING
Solved in 11 iterations and 0.00 seconds
Optimal objective  9.633333333e+01
----------> Any[1, 4] <--------- BRANCHING
Solved in 7 iterations and 0.00 seconds
Optimal objective  9.200000000e+01
----------> Any[1, 4, 9] <--------- BRANCHING

--> Any[114.5, 96.33333333333334, 77.0, 92.0] <---
Gurobi Optimizer version 9.0.3 build v9.0.3rc0 (win64)
Optimize a model with 17 rows, 15 columns and 112 nonzeros
Model fingerprint: 0x1216ab33
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [3e+00, 8e+01]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+00]
Presolve removed 10 rows and 15 columns
Presolve time: 0.00s

Solved in 0 iterations and 0.00 seconds
Infeasible model
```

Por exemplo, o vetor $[1, 4, 9]$ significa que ele está considerando um problema com as coordenadas 1,4 e 9 já arredondadas. Observe que nesse nó, temos um problema inviável, portanto teremos que voltar para um nó menos fundo que esse ná árvore e cortar todo esse ramo na hora da exploração. Esse é o código resultante de uma chamada da função $BranchNBound$. Além dela, escrevi a função $Branch\_Bound$ que realiza o método através do solver do *Gurobi*

Podemos encontrar na literatura, mais especificamente em [6] que utilizar uma DFS para a escolha dos próximos nós, bem como escolher arredondar a coordenada menos inteira (mais próxima de 0.5) são responsáveis por acelerar a busca na árvore na maioria dos problemas inteiros. Além disso, podemos adicionar restrições extras, representando cortes no espaço de soluções viáveis, chamadas de desigualdades de cortes. Estas, por reduzirem o tamanho do nosso espaço são responsáveis por acelerar a busca também. E, são tão importantes que um algoritmo com essas restrições adicionais é chamado de Branch and Cut, o próximo método apresentado.

3 Branch and Cut

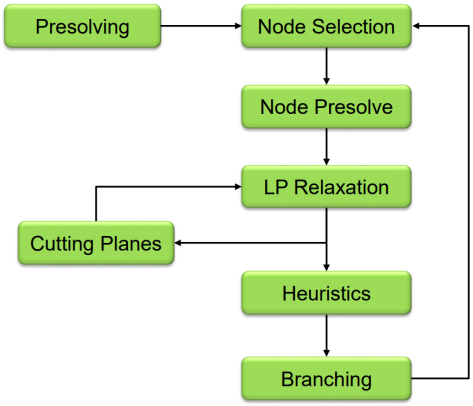
Analogamente ao método anterior, teremos um processo de *branching* da mesma forma, bem como uma exploração em profundidade da árvores de problemas arredondados associados. Entretanto, a cada iteração, podemos dinamicamente adicionar restrições de cortes. Uma dessas restrições, famosas por induzirem facetas do fecho convexo dos pontos viáveis, como demonstrado em [2], são as determinadas por cliques no grafo de conflito.

Observe que analisando a matriz A dada como *input* do nosso problema, podemos encontrar subconjuntos que não podem pertencer na solução simultamenamente. Assim, podemos montar um grafo que possui um vértice para cada coluna de A, ou seja, cada um dos subconjuntos, e existe uma aresta v_i, v_j caso v_i não possa pertencer a solução ao mesmo tempo que v_j. Observe que tal grafo pode ser construido em $O(mn^2)$ operações, onde n é o número de colunas

e m o numero de linhas, fazendo uma simples busca por A. Após tal grafo ser formado, podemos utilizar do algoritmo de *Bron–Kerbosch* para encontrar as cliques maximais de tal grafo com complexidade do pior caso de $O(3^{n/3})$.

Assim, seremos capazes de encontrar a maior clique em tal grafo que seja maximal. Com ela, podemos adicionar uma nova restrição para o problema, já dualizada, para restringir o espaço das soluções viáveis e acelerar a busca do Branch and Bound. Utilizei a implementação do **Gurobi** para realizar experimentos com tal método, visto que ele já possui uma implementação bastante otimizada do algoritmo para encontrar cliques maximais no grafo de conflito. O que é uma coisa necessária tendo em vista sua complexidade exponencial no pior caso, que é algo que temos que ter cuidado em instâncias mais complexas.

Entretanto, fiz algumas modificações nos parâmetros de entrada para impedir que o solver realizasse coisas demais. Tendo em vista que o gurobi realiza o seguinte algoritmo para resolver seus problemas inteiros:



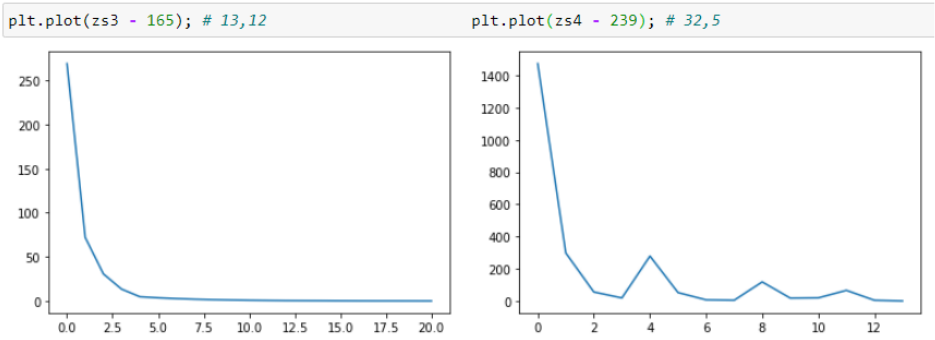
tive que passar parâmetros para impedir a etapa do *Presolving, Heuristics*, *Node Presolve* e, no caso do Branch and Bound, a etapa do *Cutting Planes*. Dessa forma pude capturar com mais precisão um método de Branch and Bound ou Branch and Cut mais puro.

4 Experimentos

Para realizar a maioria dos testes, criei as minhas próprias instâncias aleatoriamente. Para cada instância, construí a matriz A sorteando para cada entrada um lançamento de moeda para $a_{ij} = 1$ ou $a_{ij} = 0$. Dessa forma, assintoticamente, obtive instâncias com densidade cada vez mais próximas de $\frac{1}{2}$. O que representou um desafio para o meu código do método do subgradiente, visto que começou a demorar bem mais tempo e iterações que os métodos posteriormente mencionados.

4.1 Experimentos pequenos

Primeiramente, considerei alguns testes para o algoritmo do subgradiente com instâncias bem pequenas. Aqui apresento gráficos onde o eixo x representa o número de iterações e o eixo y representa o valor do upper bound do problema. O grafo é o erro absoluto do limitante superior com a solução ótima.



O gráfico da esquerda representa uma instância balanceada, onde $I, J = 12, 13$, enquanto que no da direita temos uma instância com $I, J = 5, 32$, ou seja temos muito mais subconjuntos que pontos. Observe que o perfil da primeira parece o de um método de gradiente descendente, visto que é uma curva suave que converge rapidamente para o seu valor ótimo. A segunda já nos representa um perfil mais característico do método do subgradiente, por possuir uma característica mais sinuosa. O que indica que o tamanho do passo esteja grande. Entretanto, bem satisfatório que o método implementado resolva tais instâncias pequenas em pouquíssimas iterações e, consequentemente, em pouquíssimo tempo. Outra observação é que as repostas foram encontradas dentro do número máximo de iterações passado e, depois foram verificadas com métodos de solução do gurobi e assim garantimos que estão corretas, assegurando a corretude do método.

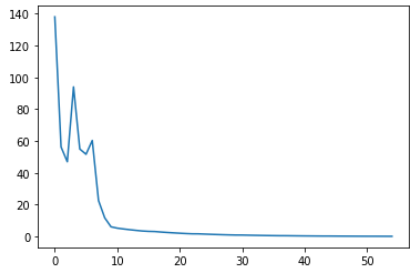
Nessas instâncias, o branch and bound e branch and cut foram ambos igualmente rápidos e precisos, não existindo nenhuma diferença entre os seus desempenhos. Ambos os métodos foram bem mais rápidos que o método do subgradiente. Mostrando o quão bem implementados e otimizados estão esses solvers.

4.2 Experimentos maiores

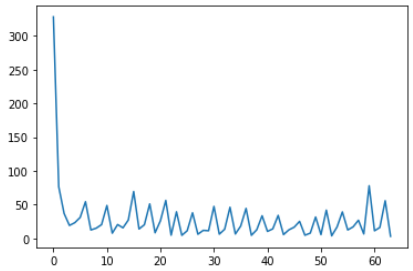
Dando continuidade aos experimentos, como não conseguimos extrair muita informação sobre os outros métodos, iremos aumentar a dimensão das nossas instâncias aleatórias. Primeiramente, temos esse conjunto médio de testes, onde em ambos os casos temos a característica oscilatória na convergência do método do subgradiente. Em ambas as instâncias, o método convergiu em menos de 0.5 segundos, realizando em torno de 60 iterações até convergir quando o limi-

tante superior e inferior se aproximaram. Como podemos conferir nos seguintes gráficos:

```
plt.plot(zm1 - 98); # 64,6
```

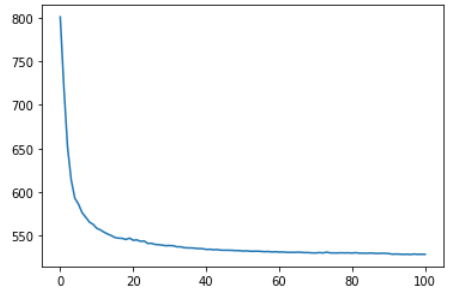



```
plt.plot(zm2 - 61); # 25,25
```

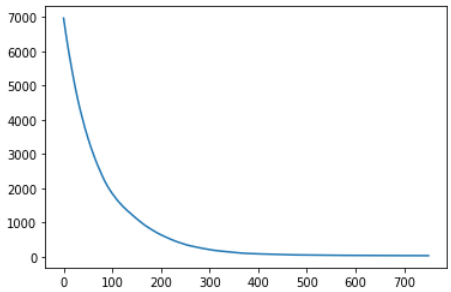


Novamente, os métodos que utilizam o solver do *Gurobi* resolveu com muita trivialidade mesmo retirando os processos de *presolving*. Mostrando que para conseguir compará-los iremos precisar de instâncias maiores ainda. Portanto, damos continuidade considerando uma instância 100×500 e outra 300×200. Nessas tive que utilizar o passo constante na ordem de 10^{-5} para obter uma convergência segura no subgradiente. Podemos observar na próxima figura que os métodos ja começaram a demorar mais de uma centena de iterações, ou mais ainda, na instância 300×200 chegando a ultrapassar 700 iterações. Dessa vez, o método demorou cerca de $1s$ e $4s$ respectivamente.

```
plt.plot(zgig); # 500x100
```




```
plt.plot(zgig2 - 421);  # 200x300
```



Já para os métodos de Branch and Bound e Branch and Cut, já começamos a perceber uma grande diferença entre suas execuções e desempenho. O método do Branch and Cut resolvia a instância em metade do tempo do Branch and Bound! Podemos perceber tal diferença de desempenho numa instância um pouco maior, como consta nas seguintes figuras, mostrando a chamada do branch and bound e branch and cut, respectivamente, para uma instância 350×500:

In [1241]:
```
rp, rp_x = setSPP(Gurobi.Optimizer, c, A)

# 500 x 350

println("Solving...");
set_optimizer_attribute(rp, "Presolve", 0)
set_optimizer_attribute(rp, "Cuts", 0)
set_optimizer_attribute(rp, "Heuristics", 0)
optimize!(rp)
```

```
Academic license - for non-commercial use only
Solving...
Academic license - for non-commercial use only
Gurobi Optimizer version 9.0.3 build v9.0.3rc0 (win64)
Optimize a model with 500 rows, 350 columns and 87330 nonzeros
Model fingerprint: 0xa0d1fe57
Variable types: 0 continuous, 350 integer (350 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [1e+00, 1e+02]
  Bounds range     [0e+00, 0e+00]
  RHS range        [1e+00, 1e+00]
Variable types: 0 continuous, 350 integer (350 binary)

Root relaxation: objective 1.562546e+02, 307 iterations, 0.03 seconds

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0  156.25463    0   79          -  156.25463      -     -    0s
     0     0  156.25463    0   79          -  156.25463      -     -    0s
     0     2  156.25463    0   79          -  156.25463      -     -    0s
*    2     2               1      99.0000000  155.00045  56.6%  80.0    1s

Explored 212 nodes (10426 simplex iterations) in 2.13 seconds
Thread count was 8 (of 8 available processors)

Solution count 1: 99

Optimal solution found (tolerance 1.00e-04)
Best objective 9.900000000000e+01, best bound 9.900000000000e+01, gap 0.0000%
```

Observe que o código do branch and bound resolveu o problema em 2.13 segundos, explorando 212 nós.

```
In [1242]: rp, rp_x = setSPP(Gurobi.Optimizer, c, A)

           # 500 x 350

           println("Solving...");
           set_optimizer_attribute(rp, "Presolve", 0)
           set_optimizer_attribute(rp, "Cuts", 0)
           set_optimizer_attribute(rp, "Heuristics", 0)
           set_optimizer_attribute(rp, "CliqueCuts", 1)
           optimize!(rp)
```

```
Academic license - for non-commercial use only
Solving...
Academic license - for non-commercial use only
Gurobi Optimizer version 9.0.3 build v9.0.3rc0 (win64)
Optimize a model with 500 rows, 350 columns and 87330 nonzeros
Model fingerprint: 0xa0d1fe57
Variable types: 0 continuous, 350 integer (350 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [1e+00, 1e+02]
  Bounds range     [0e+00, 0e+00]
  RHS range        [1e+00, 1e+00]
Variable types: 0 continuous, 350 integer (350 binary)

Root relaxation: objective 1.562546e+02, 307 iterations, 0.03 seconds

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0  156.25463    0   79          -  156.25463      -     -    0s
*    0     0               0      99.0000000   99.00000  0.00%     -    0s

Cutting planes:
  Clique: 1

Explored 1 nodes (443 simplex iterations) in 0.83 seconds
Thread count was 8 (of 8 available processors)

Solution count 1: 99

Optimal solution found (tolerance 1.00e-04)
Best objective 9.900000000000e+01, best bound 9.900000000000e+01, gap 0.0000%
```

Entretanto, o branch and cut levou apenas 0.83 segundos para resolver, explorando apenas 1 nó. Mostrando o poder do corte de clique para o problema do *Set packing*.

Para instâncias desse tamanho, o código da relaxação lagrangiana demorava muitas iterações para se aproximar do valor da relaxação linear, visto que o tamanho do passo necessário para garantir a convergência é muito pequeno. Portanto, podemos considerar que o tempo de convergência desse método é de $+\infty$. Por fim, considerei uma instância 1000×1000, com densidade próxima de 0.5, me aproveitando do poder do solver do Gurobi.

```
In [1276]: rp, rp_x = setSPP(Gurobi.Optimizer, c, A)

           # 1000 x 1000

           println("Solving...");
           set_optimizer_attribute(rp, "Presolve", 0)
           set_optimizer_attribute(rp, "Cuts", 0)
           set_optimizer_attribute(rp, "Heuristics", 1)
           optimize!(rp)
```

```
Academic license - for non-commercial use only
Solving...
Academic license - for non-commercial use only
Gurobi Optimizer version 9.0.3 build v9.0.3rc0 (win64)
Optimize a model with 1000 rows, 1000 columns and 499993 nonzeros
Model fingerprint: 0x3538eacf
Variable types: 0 continuous, 1000 integer (1000 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [1e+00, 1e+02]
  Bounds range     [0e+00, 0e+00]
  RHS range        [1e+00, 1e+00]
Found heuristic solution: objective 82.0000000
Variable types: 0 continuous, 1000 integer (1000 binary)

Root relaxation: objective 1.672537e+02, 1102 iterations, 0.39 seconds

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0  167.25367    0  170   82.00000  167.25367   104%     -    1s
H    0     0                       97.0000000  167.25367  72.4%     -    1s
H    0     0                       98.0000000  167.25367  70.7%     -    1s
H    0     0                       99.0000000  167.25367  68.9%     -    1s
     0     0  167.25367    0  170   99.00000  167.25367  68.9%     -    2s
     0     2  166.22779    0  170   99.00000  166.22779  67.9%     -    7s
     5     2  165.95875    3  162   99.00000  165.95875  67.6%   252   10s
    23     2  163.13437   12  154   99.00000  163.52554  65.2%   258   15s
    56     2  159.47683   29  168   99.00000  159.73932  61.4%   238   20s
    77     3  157.58449   39  172   99.00000  157.73620  59.3%   234   25s
   101     2     cutoff   51        99.00000  155.79578  57.4%   226   30s
   153     4  152.48937   78  145   99.00000  152.70973  54.3%   211   35s
   207     2     cutoff  104        99.00000  148.11420  49.6%   203   40s
   267     2  144.06595  135  130   99.00000  144.10628  45.6%   192   46s
   305     4     cutoff  153        99.00000  141.43055  42.9%   187   50s
   354     8  138.39151  178  116   99.00000  138.54903  39.9%   178   55s
   405     2     cutoff  203        99.00000  135.87803  37.3%   170   60s
   453     2     cutoff  227        99.00000  132.77404  34.1%   164   66s
   495     2     cutoff  248        99.00000  129.61580  30.9%   160   70s
   539     2     cutoff  270        99.00000  126.56387  27.8%   155   76s
   579     2     cutoff  290        99.00000  123.96743  25.2%   151   80s
   617     2     cutoff  309        99.00000  121.98558  23.2%   147   86s
   663     2     cutoff  332        99.00000  119.67646  20.9%   143   90s
   691     2     cutoff  346        99.00000  117.95466  19.1%   139   96s
   747     2     cutoff  374        99.00000  112.86426  14.0%   133  101s
   803     2     cutoff  402        99.00000  108.37775  9.47%   126  107s

Explored 857 nodes (103297 simplex iterations) in 107.89 seconds
Thread count was 8 (of 8 available processors)

Solution count 4: 99 98 97 82

Optimal solution found (tolerance 1.00e-04)
Best objective 9.900000000000e+01, best bound 9.900000000000e+01, gap 0.0000%
```

Observe que nesse código do branch and bound, permiti que o *Gurobi* realizasse seus procedimentos heurísticos. Ele foi capaz de resolver tal instância em 107.89 segundos, explorando um total de 857 nós. Portanto, pelos testes passados, esperamos que a mesma instância com o branch and cut possa ser resolvida em pelo menos 50 segundos.

```
In [1274]: rp, rp_x = setSPP(Gurobi.Optimizer, c, A)

           # 1000 x 1000

           println("Solving...");
           set_optimizer_attribute(rp, "Presolve", 0)
           set_optimizer_attribute(rp, "Cuts", 0)
           set_optimizer_attribute(rp, "Heuristics", 0)
           set_optimizer_attribute(rp, "CliqueCuts", 1)
           optimize!(rp)
```

```
Academic license - for non-commercial use only
Solving...
Academic license - for non-commercial use only
Gurobi Optimizer version 9.0.3 build v9.0.3rc0 (win64)
Optimize a model with 1000 rows, 1000 columns and 499993 nonzeros
Model fingerprint: 0x3538eacf
Variable types: 0 continuous, 1000 integer (1000 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [1e+00, 1e+02]
  Bounds range     [0e+00, 0e+00]
  RHS range        [1e+00, 1e+00]
Variable types: 0 continuous, 1000 integer (1000 binary)

Root relaxation: objective 1.672537e+02, 1102 iterations, 0.38 seconds

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0  167.25367    0  170          -  167.25367      -     -    1s
     0     0  167.25367    0  170          -  167.25367      -     -    2s
     0     2  167.25367    0  170          -  167.25367      -     -    3s
*    2     2               1      92.0000000  167.22688  81.8%   252    4s
     5     2  166.71281    3  170   92.00000  166.71281  81.2%   243    5s
*    6     2               3      97.0000000  166.70912  71.9%   245    5s
*   12     2               6      98.0000000  165.74325  69.1%   244    8s
    19     2  164.16287   10  162   98.00000  164.55984  67.9%   258   10s
*   20     2              10      99.0000000  164.15799  65.8%   259   10s
    81     3     cutoff   41        99.00000  158.21389  59.8%   242   15s
   191     5     cutoff   96        99.00000  150.51555  52.0%   214   20s
   328     5     cutoff  164        99.00000  140.64486  42.1%   190   25s
   472     2  131.72729  236  124   99.00000  131.72729  33.1%   175   30s
   625     1     cutoff  313        99.00000  120.94933  22.2%   157   35s

Explored 831 nodes (110759 simplex iterations) in 37.61 seconds
Thread count was 8 (of 8 available processors)

Solution count 4: 99 98 97 92

Optimal solution found (tolerance 1.00e-04)
Best objective 9.900000000000e+01, best bound 9.900000000000e+01, gap 0.0000%
```

Entretanto, mais uma vez, podemos perceber o poder do branch and cut, quando ele resolve a mesma instância 1000×1000 em aproximadamente 37 segundos, quase 1/3 do tempo. Analogamente à instância anterior, o método do subgradiente da relaxação lagrangiana não foi capaz de resolver essa instância devido a suas proporções.

Coloquei os resultados das instâncias aleatórias na seguinte tabela:

Instâncias	RL	Iters/Tempo	BB	Iters/Tempo	BC	Iters/Tempo
12 x 7	4	6 iterações	4	1 iteração	4	1 iteração
25 x 25	135	69 iterações	135	4 iterações	135	1 iteração
200 x 300	421	737 iterações	421	0.46s	421	0.17s
500 x 350	-	-	99	2.13s	99	0.83s
1000 x 250	-	-	98	1.97s	98	0.94s
1000 x 1000	-	-	99	107.89s	99	37.61s

Onde **RL**,**BB**,**BC** representam os resultados obtidos a partir dos métodos da relaxação lagrangiana, branch and bound e branch and cut respectivamente.

Similarmente, testei os métodos com umas instâncias encontradas na pasta *Data*, que foram retiradas do github: https://github.com/xgandibleux/metaSPPstu. Obtive os seguintes resultados:

Instâncias	RL	Iters/Tempo	BB	Iters/Tempo	BC	Iters/Tempo
500x100	530.1	104 iterações	372	0.11s	372	0.04s
1000x200	-	-	416	15.52s	416	9.8s
2500x500	-	-	-	-	-	-

Mesmo com a implementação do *Gurobi*, a ultima instância demorava mais do que o tempo máximo considerado (15 minutos). Portanto, trata-se de uma instância bem complexa. Entretanto, mesmo sem tal resultado, creio que esses testes foram bem conclusivos para uma comparação entre os métodos.

5 Conclusão

"I always thought something was fundamentally wrong with the universe" [1]

A partir dessas comparações diretas entre os métodos com instâncias de diferentes magnitudes, podemos tirar as seguintes conclusões. Primeiramente, podemos notar que o método do subgradiente é pouco robusto, em relação a sua convergência para casos não muito pequenos. Nas diferentes instâncias maiores, foram necessários ajustes diversos no tamanho do passo e tolerância pedida. Entretanto, por ter sido um método implementado por mim, de maneira bem direta, é fácil de ser manipulado e alterado para adicionar diferentes restrições, como as de cliques ou outras.

Além disso, podemos notar que o *Gurobi* é uma ferramenta poderosíssima. Devido a sua facilidade de resolver instâncias com tamanhos imensos e extrema facilidade. Contudo, saber chamar o gurobi com as especificações e parâmetros corretos é uma arte em si. Tendo em vista que seu desempenho com os cortes de cliques foi extremamente melhor que o sem os cortes de cliques, nada nos impede de supor que exista alguma heurística que possa acelerar ainda mais o seu processamento. O que podemos garantir é que não exista um corte melhor, tendo em vista que esse corte induz uma faceta do fecho convexo.

Outra coisa que vale a pena ser mencionada é o surgimento de outras heurísticas e aproximações na literatura. No artigo [4], os autores introduzem uma metaheurística gulosa e estocástica, chamada de $GRASP$ para resolver instâncias do *Set packing*. Esta apresenta resultados muito positivos nesse artigo, comparando com outros métodos da época. Mais uma heurística são os chamados algoritmos genéticos, inspirados na teoria evolucionista. Porém, esses são algoritmos mais pesados e difíceis de se programar. Tendo em vista que precisam criar uma população com um tamanho muito grande, dependendo da instância do problema considerado.

Por fim, concluo esse trabalho agradecendo ao professor Abilio Lucena, por ter lecionado a disciplina de Otimização Combinatória nesse periodo letivo especial de 2020.

Referências

[1] D. Adams. *The Hitchhiker's Guide to the Galaxy*. San Val, 1995.

[2] Lázaro Cánovas, Mercedes Landete, and Alfredo Marín. Facet obtaining procedures for set packing problems. *SIAM Journal on Discrete Mathematics*, 16(1):127–155, 2002.

[3] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

[4] Xavier Delorme, Xavier Gandibleux, and Joaquin Rodriguez. Grasp for set packing problems. *European Journal of Operational Research*, 153(3):564–580, 2004.

[5] Yunsong Guo, Andrew Lim, Brian Rodrigues, and Jiqing Tang. Using a lagrangian heuristic for a combinatorial auction problem. In *17th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'05)*, pages 5–pp. IEEE, 2005.

[6] Incorporate Gurobi Optimization. Gurobi optimizer reference manual. *URL http://www. gurobi. com*, 2018.

[7] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.