Aplicação de heurísticas em simulações numéricas para o problema da mochila com variáveis inteiras

Autor: *Gabriel Pietsiaki Izidoro*
Colaboradores: *Prof. Dr. Robinson Samuel Vieira Hoto, Larissa Maria dos Santos Fonseca*
Contato: *gabriel.pietsiaki@uel.br*
Resumo: *Problemas de otimização é uma classe de problemas de grande relevância para a matemática, computação e industria em geral, suas aplicações podem definir diferenças entre custos de milhares de dólares comparando soluções boas com ótimas. O presente artigo levanta em questão quatro tipos de heurísticas aplicadas matematicamente e computacionalmente ao problema da mochila realizando simulações com diversas classes de mochilas e tipos de itens.*

1 Introdução

Problemas baseados em encontrar máximos e mínimos de modelos surgem constantemente nas industrias ao encontrar múltiplos meios de resolução de um mesmo problema, dados recursos limitados de material na produção de itens, e restrições de manipulação em fábrica. Ainda, ao modelar tal problema de forma que seja expresso de uma função linear de variáveis finitas e restrições simples sob seus recursos, Cormen et al. [1] caracteriza este como um problema de programação linear.

Diversos métodos e algoritmos foram desenvolvidos para encontrar soluções ótimas e soluções razoáveis em função de tempo de processamento da tarefa disponível e magnitude das variáveis. Nesta conjuntura, as heurísticas míopes com ordenação de tamanho, prioridade e prioridade por tamanho surgem como uma maneira direta de busca por soluções aceitáveis desta classe de problemas. Juntamente, o Branch and Bound acompanha as heurísticas em busca de soluções ótimas realizando buscas em regiões onde há sentido haver boas soluções.

As seguintes seções deste artigo apresentam aplicações de heurísticas executadas sobre o problema da mochila com variáveis inteiras em 600 exemplares distintos de tamanhos variados gerados aleatoriamente. Análises numéricas são compiladas, a seguir, de forma a observar a eficiência de cada uma das quatro heurísticas apresentadas sob condições distintas.

2 Modelagem Matemática

O modelo matemático utilizado para o problema a seguir e a construção das dadas heurísticas possuem grande contribuição do Prof. Dr. Robinson Samuel Vieira Hoto (Universidade Estadual de Londrina). Comecemos analisando o problema em questão: queremos maximizar a função f que representa o somatório das prioridades acumuladas do dado problema, isto é, a forma geral do problema da mochila.

2.1 Problema

Considerando as seguintes informações, podemos descrever tal problema da forma onde, seja c a capacidade máxima do problema; $n \in \mathbb{N} : n > 0$ a quantidade de variáveis; l_i o tamanho (valor) da variável de índice i; x_i a quantidade de itens alocados da variável de índice i; p_i a prioridade da variável de índice i; f a prioridade acumulada; e , por fim, $i = 1, 2, 3, \cdots, n$.

Desta forma temos que $f(x_1, x_2, x_3, \cdots, x_n) = p_1x_1 + p_2x_2 + p_3x_3 + \cdots + p_nx_n$, isto é,

$$f(x_1, x_2, x_3, \cdots, x_n) = \sum_{i=1}^{n} p_i x_i \tag{1}$$

Também, generalizando o processo de inserção de itens na mochila, podemos calcular x_i sempre inserindo o máximo de itens da variável de índice i no espaço que há restante na mochila, note que para a primeira variável ($i = 1$) não é necessário calcular o espaço já ocupado por outros itens na mochila, então não há sentido em realizar o somatório, considere como 0 neste caso. Desta forma,

$$x_i = \left\lfloor \frac{c - \sum_{j=1}^{i-1} l_j x_j}{l_i} \right\rfloor \tag{2}$$

Ainda, como restrição, a capacidade máxima do problema descreve o tamanho da mochila, portanto a soma dos tamanhos de todos os itens contidos na mochila deve ser menor ou igual a capacidade da mochila para todos estarem suficientemente comportados dentro da mochila do problema. Por fim, com todas as informações definidas, podemos determinar o modelo matemático seguindo a forma-padrão e a função objetivo de Cormen et al. [1]:

$$\begin{aligned} \max f &= \sum_{i=1}^{n} p_i x_i \\ \text{s.a.} : &\sum_{i=1}^{n} l_i x_i \leq c \\ & x_i \geq 0 \text{ e inteiro}, i = 1, 2, 3, \cdots, n \end{aligned}$$

2.2 Heurísticas

Em problemas particularmente complexos ou de grande porte, uma solução que não é necessariamente a melhor possível pode ser aceita na tentativa de minimizar o esforço gasto. O objetivo é descobrir a solução com um esforço computacional razoável. Tal método de resolução é então chamado de heurística [2]. Os seguintes tópicos apresentam as quatro heurísticas alvo deste artigo.

2.2.1 Heurística Míope com Ordenação de Tamanho (MOT)

A heurística míope com ordenação de tamanho consiste em selecionar a próxima peça a ser adicionada à solução parcial com base em seu tamanho e em sua contribuição para a solução. Em outras palavras, a heurística míope com ordenação de tamanho escolhe a peça mais promissora em termos do seu valor, sem levar em conta as peças restantes ou o tamanho total da mochila durante a escolha. Essa heurística pode ser aplicada a uma variedade de problemas de otimização combinatória, incluindo o problema da mochila e problemas de programação inteira.

Matematicamente, a MOT requer primeiramente que a condição $l_1 \geq l_2 \geq l_3 \geq \cdots \geq l_n$ seja válida, portanto as variáveis devem ser ordenadas de maneira que seus tamanhos (valores) estejam em ordem decrescente. Posteriormente, o processo da dada heurística consiste em calcular diretamente $x_1, x_2, x_3, \cdots, x_n$ conforme o processo geral de inserção de itens na mochila. Os valores $(x_1, x_2, x_3, \cdots, x_n)$ calculados representam a solução do problema da mochila fornecido pela heurística MOT.

A MOT fornece soluções aproximadas de boa qualidade rapidamente, contudo ao não considerar todas as outras combinações possíveis de itens da mochila, esta permite desconsiderar melhores soluções.

2.2.2 Heurística Míope com Ordenação de Prioridade (MOP)

A heurística míope com ordenação por prioridade é uma estratégia de otimização que busca maximizar iterativamente sempre os elementos de maior prioridade para o problema, sem considerar outros fatores. Ela consiste em uma abordagem simplificada de resolução de problemas complexos, na qual se escolhe sempre a alternativa mais atrativa em termos de um critério específico, ignorando possíveis efeitos negativos a longo prazo. Essa abordagem é bastante utilizada em problemas de tomada de decisão com múltiplas opções e critérios conflitantes.

Matematicamente, a MOP requer primeiramente que a condição $p_1 \geq p_2 \geq p_3 \geq \cdots \geq p_n$ seja válida, portanto as variáveis devem ser ordenadas de maneira que suas prioridades estejam em ordem decrescente. Posteriormente, o processo da dada heurística consiste em calcular diretamente $x_1, x_2, x_3, \cdots, x_n$ conforme o processo geral de inserção de itens na mochila. Os valores $(x_1, x_2, x_3, \cdots, x_n)$ calculados representam a solução do problema da mochila fornecido pela heurística MOP, o que é semelhante à MOT.

A MOP é bastante útil em situações em que a complexidade do problema inviabiliza uma solução ótima em tempo hábil, ou em que os dados disponíveis são insuficientes para uma análise completa.

2.2.3 Heurística Míope com Ordenação de Prioridade/Tamanho (MOPT)

Matematicamente, a MOPT requer primeiramente calcular um fator λ de prioridade/tamanho, isto é,

$$\lambda_i = \frac{p_i}{l_i}, i = 1, 2, 3, \cdots, n \tag{3}$$

para cada variável e, então, produzir uma ordenação decrescente tal que a condição $\lambda_1 \geq \lambda_2 \geq \lambda_3 \geq \cdots \geq \lambda_n$ seja válida. Posteriormente, o processo da dada heurística consiste em calcular diretamente $x_1, x_2, x_3, \cdots, x_n$ conforme o processo geral de inserção de itens na mochila. Os valores $(x_1, x_2, x_3, \cdots, x_n)$ calculados representam a solução do problema da mochila fornecido pela heurística MOPT, o que é semelhante à MOT e à MOP.

Comparada às heurísticas míopes com ordenação de tamanho e de prioridade, a MOPT relaciona custo e benefício em um fator que pode melhor aproximar a solução dada à melhor solução possível ao problema da mochila, sendo então uma forma inteligente e eficiente de gerar uma solução aproximada por critérios relevantes. Entretanto a característica míope da MOPT não nos permite garantir que a dada heurística gerará a melhor solução ao problema da mochila para todos os casos.

2.2.4 Branch and Bound (BB)

Segundo Kianfar [3], o Branch and Bound é definido como um algoritmo com propósito geral de enumeração implícita em regiões viáveis que evita enumerar cada ponto da solução explicitamente. Inicialmente, foi proposto por Land e Doig [4] para resolver problemas de programação inteira.

O algoritmo utiliza o conceito de subproblemas do tipo de estratégia dividir e conquistar, desta forma decompondo o problema em subproblemas sobre a estrutura de uma árvore [3]. Uma vez que o Branch and Bound enumera implicitamente as soluções, ao fim do processo completo do algoritmo, o método terá percorrido toda a árvore de enumerações e encontrado a solução ótima para o problema de otimização, o que o torna um algoritmo para solução exata de problemas de otimização da sua classe. Entretanto, pode-se restringir os limites de enumeração a regiões da árvore mais relevantes e descartar outra possibilidades para otimizar o tempo de execução, como é discutido na seção 4, assim tornando-o uma heurística que produz uma solução ótima.

O tipo de enumeração implícita característica do algoritmo é dado pelo processo em percorrer a árvore de enumerações de soluções do problema limitando os ramos para percorrer com base em uma função de limitação que fornece um resultado ao próprio algoritmo, se este deve ou não computar os ramos posteriores baseado na estimativa da função.

2.2.4.1 Limitante Digamos que $(\hat{x}_1, \hat{x}_2, \hat{x}_3, \cdots, \hat{x}_k, \cdots, \hat{x}_n)$ consista na melhor solução conhecida até o momento, isto significa que:

$$\hat{f} = \sum_{i=1}^{n} p_i \hat{x}_i > f, \text{ para qualquer outra solução anterior} \tag{4}$$

Na folha onde está $\hat{x}_n$ não vale a pena explorar o ramo em que:

$$\tilde{x}_i = \hat{x}_i \text{ para } i = 1,2,3,\cdots,n-1 \text{ e } \tilde{x}_n = x_n - 1 \tag{5}$$

Pois,

$$\tilde{f} = \sum_{i=1}^{n} p_i\tilde{x}_i = \sum_{i=1}^{n-1} p_i\tilde{x}_i + p_n\tilde{x}_n = \sum_{i=1}^{n-1} p_i\hat{x}_i + p_n(\hat{x}_n - 1) = \sum_{i=1}^{n} p_i\hat{x}_i - p_n < \hat{f} \tag{6}$$

Assim, ao retornar no ramo $\hat{x}$ (melhor solução até o momento), seja k o primeiro índice, tal que $\hat{x}_k > 0$. A partir do vértice de $\hat{x}_k$ construiremos um novo ramo $\bar{x}$, tal que:

$$\bar{x_1} = \hat{x}_1, \bar{x}_2 = \hat{x}_2, \cdots, \bar{x}_{k-1} = \hat{x}_{k-1}, \bar{x}_k = \hat{x}_k, \bar{x}_{k+1} = ?, \cdots, \bar{x}_n = ? \tag{7}$$

Logo,

$$\bar{f} = \sum_{i=1}^{n} p_i\bar{x}_i = \sum_{i=1}^{k-1} p_i\hat{x}_i + p_k(\hat{x}_k - 1) + \sum_{\mathbf{i=k+1}}^{\mathbf{n}} \mathbf{p_i\bar{x}_i}, \text{ onde } \sum_{i=k+1}^{n} p_i\bar{x}_i \text{ é desconhecido.} \tag{8}$$

Para obtermos um limitante superior de $\bar{f}$, nós teremos que limitar superiormente $\sum_{i=k+1}^{n} p_i\bar{x}_i$. Sabemos que $\bar{x}$ necessita satisfazer $\sum_{i=1}^{n} l_i\bar{x}_i \leq c$, que é equivalente a

$$\sum_{i=1}^{k} l_i\bar{x}_i + \sum_{i=k+1}^{n} l_i\bar{x}_i \leq c \Leftrightarrow \sum_{i=1}^{k-1} l_i\hat{x}_i + l_k(\hat{x}_k - 1) + \sum_{i=k+1}^{n} l_i\bar{x}_i \leq c \tag{9}$$

$$\sum_{i=k+1}^{n} l_i\bar{x}_i \leq c - \left(\sum_{i=1}^{k-1} l_i\hat{x}_i + l_k(\hat{x}_k - 1)\right) \tag{10}$$

Agora, temos um limitante superior para $\sum_{i=k+1}^{n} l_i\bar{x}_i$, mas precisamos de um limitante superior para $\sum_{i=k+1}^{n} p_i\bar{x}_i$, para este, então seja

$$\bar{c} = c - \left(\sum_{i=1}^{k-1} l_i\hat{x}_i + l_k(\hat{x}_k - 1)\right) \tag{11}$$

Assim desejamos limitar superiormente,

$$\sum_{i=k+1}^{n} p_i\bar{x}_i = p_{k+1}\bar{x}_{k+1} + p_{k+2}\bar{x}_{k+2} + p_{k+3}\bar{x}_{k+3} + \cdots + p_{n-1}\bar{x}_{n-1} + p_n\bar{x}_n \tag{12}$$

Sabendo que,

$$\sum_{i=k+1}^{n} l_i\bar{x}_i = l_{k+1}\bar{x}_{k+1} + l_{k+2}\bar{x}_{k+2} + l_{k+3}\bar{x}_{k+3} + \cdots + l_{n-1}\bar{x}_{n-1} + l_n\bar{x}_n \leq \bar{c} \tag{13}$$

Partindo de $\sum_{i=k+1}^{n} p_i\bar{x}_i$, temos que:

$$\begin{aligned} p_{k+1}\bar{x}_{k+1} + \cdots + p_n\bar{x}_n &= p_{k+1}\bar{x}_{k+1} + \cdots + p_n\bar{x}_n + (l_{k+1}\bar{x}_{k+1} + \cdots + l_n\bar{x}_n) - (l_{k+1}\bar{x}_{k+1} + \cdots + l_n\bar{x}_n) \\ &= (p_{k+1} + l_{k+1} - l_{k+1})\bar{x}_{k+1} + \cdots + (p_n + l_n - l_n)\bar{x}_n \end{aligned} \tag{14}$$

Então,

$$p_{k+1}\bar{x}_{k+1} + p_{k+2}\bar{x}_{k+2} + \cdots + p_{n-1}\bar{x}_{n-1} + p_n\bar{x}_n = p_{k+1}\frac{1}{l_{k+1}}l_{k+1}\bar{x}_{k+1} + p_{k+2}\frac{1}{l_{k+2}}l_{k+2}\bar{x}_{k+2} + \cdots + p_{n-1}\frac{1}{l_{n-1}}l_{n-1}\bar{x}_{n-1} + p_n\frac{1}{l_n}l_n\bar{x}_n \tag{15}$$

Vamos assumir que:

$$\frac{p_{k+1}}{l_{k+1}} \geq \frac{p_{k+2}}{l_{k+2}} \geq \cdots \geq \frac{p_{n-1}}{l_{n-1}} \geq \frac{p_n}{l_n} \text{ para } k = 1,2,3,\cdots,n-1,n \tag{16}$$

Portanto assumiremos que:

$$\frac{p_1}{l_1} \geq \frac{p_2}{l_2} \geq \frac{p_3}{l_3} \geq \cdots \geq \frac{p_k}{l_k} \geq \cdots \geq \frac{p_{n-1}}{l_{n-1}} \geq \frac{p_n}{l_n} \tag{17}$$

Com isto,

$$\begin{aligned}
&\frac{p_{k+1}}{l_{k+1}}l_{k+1}\bar{x}_{k+1} + \frac{p_{k+2}}{l_{k+2}}l_{k+2}\bar{x}_{k+2} + \cdots + \frac{p_{n-1}}{l_{n-1}}l_{n-1}\bar{x}_{n-1} + \frac{p_n}{l_n}l_n\bar{x}_n \\
&\leq \frac{p_{k+1}}{l_{k+1}}l_{k+1}\bar{x}_{k+1} + \frac{p_{k+1}}{l_{k+1}}l_{k+2}\bar{x}_{k+2} + \cdots + \frac{p_{k+1}}{l_{k+1}}l_{n-1}\bar{x}_{n-1} + \frac{p_{k+1}}{l_{k+1}}l_n\bar{x}_n \\
&= \frac{p_{k+1}}{l_{k+1}}\left(l_{k+1}\bar{x}_{k+1} + l_{k+2}\bar{x}_{k+2} + \cdots + l_{n-1}\bar{x}_{n-1} + l_n\bar{x}_n\right) \\
&= \frac{p_{k+1}}{l_{k+1}}\sum_{i=k+1}^{n} l_i\bar{x}_i \\
&\leq \frac{p_{k+1}}{l_{k+1}}\bar{c}
\end{aligned} \tag{18}$$

Portanto, temos que o limitante é dado por

$$\bar{f} = \sum_{i=1}^{n} p_i\bar{x}_i \leq \sum_{i=1}^{k-1} p_i\hat{x}_i + p_k(\hat{x}_k - 1) + \frac{p_{k+1}}{l_{k+1}}\left[c - \sum_{i=1}^{k-1} l_i\hat{x}_i + l_k(\hat{x}_k - 1)\right] = \bar{F} \tag{19}$$

Mas também, podemos ajustar o limitante na forma de B de maneira que seja um inteiro, permitindo comparar adequadamente com $\bar{f}$ e respeitando a restrição $x_i \geq 0$ e inteiro, $i = 1,2,3,\cdots,n$

$$B = \lfloor \bar{F} + 1 \rfloor = \left\lfloor \sum_{i=1}^{k-1} p_i\hat{x}_i + p_k(\hat{x}_k - 1) + \frac{p_{k+1}}{l_{k+1}}\left[c - \sum_{i=1}^{k-1} l_i\hat{x}_i + l_k(\hat{x}_k - 1)\right] + 1 \right\rfloor \tag{20}$$

3 Modelagem Computacional

3.1 Algoritmos

Para os algoritmos abaixo foram utilizados os conceitos previamente apresentados no artigo e também os dois algoritmos apresentados por Kianfar [3] para o Branch and Bound em fluxogramas em problemas de otimização, que serviram de inspiração junto com a produção do professor Hoto (2023, comunicação pessoal) para construção do algoritmo 4.

3.1.1 MOT

Uma estrutura geral do processo de solução do problema da mochila pela heurística MOT pode ser construída como no algoritmo 1. Note que a função *Ordenar()* é uma função arbitrária que ordena as variáveis conforme os critérios estabelecidos para a MOT em 2.2.1.

Algoritmo 1: Heurística Míope com Ordenação de Tamanho (MOT)

Entrada: $n > 0$: número de variáveis (inteiro)
$c > 0$: capacidade da mochila (inteiro)
L: vetor de n inteiros positivos de tamanho dos itens
Saída: Vetor solução do problema da mochila pela MOT
Dados: i: varíavel inteira contadora
X: vetor de n inteiros positivos de quantidade de itens

$L \leftarrow \underline{\text{Ordenar}}\ (L)$
para $i \leftarrow 1$ **até** n **faça**
  $X[i] \leftarrow \left\lfloor \frac{c - \sum_{j=1}^{i-1} L[j]X[j]}{L[i]} \right\rfloor$
fim
retornar X

3.1.2 MOP

Uma estrutura geral do processo de solução do problema da mochila pela heurística MOP pode ser construída como no algoritmo 2. Note que a função *Ordenar()* é uma função arbitrária que ordena as variáveis conforme os critérios estabelecidos para a MOP em 2.2.2.

Algoritmo 2: Heurística Míope com Ordenação de Prioridade (MOP)

Entrada: $n > 0$: número de variáveis (inteiro)
$c > 0$: capacidade da mochila (inteiro)
L: vetor de n inteiros positivos de tamanho dos itens
P: vetor de n inteiros positivos de prioridade dos itens
Saída: Vetor solução do problema da mochila pela MOP
Dados: i: varíavel inteira contadora
X: vetor de n inteiros positivos de quantidade de itens

$L \leftarrow \underline{\text{Ordenar}}\ (L, P)$
para $i \leftarrow 1$ **até** n **faça**
  $X[i] \leftarrow \left\lfloor \frac{c - \sum_{j=1}^{i-1} L[j]X[j]}{L[i]} \right\rfloor$
fim
retornar X

3.1.3 MOPT

Uma estrutura geral do processo de solução do problema da mochila pela heurística MOPT pode ser construída como no algoritmo 3. Note que a função *Ordenar()* é uma função arbitrária que ordena as variáveis conforme os critérios estabelecidos para a MOPT em 2.2.3.

Algoritmo 3: Heurística Míope com Ordenação de Prioridade/Tamanho (MOPT)

Entrada: $n > 0$: número de variáveis (inteiro)
$c > 0$: capacidade da mochila (inteiro)
L: vetor de n inteiros positivos de tamanho dos itens
P: vetor de n inteiros positivos de prioridade dos itens
Saída: Vetor solução do problema da mochila pela MOPT
Dados: i: varíavel inteira contadora
X: vetor de n inteiros positivos de quantidade de itens
Λ: vetor de n racionais que contém o fator λ de cada variável

1 **para** $i \leftarrow 1$ **até** n **faça**
2 $\quad \Lambda[i] \leftarrow \frac{P[i]}{L[i]}$
3 **fim**
4 $L \leftarrow \underline{\text{Ordenar}}\ (L, \Lambda)$
5 **para** $i \leftarrow 1$ **até** n **faça**
6 $\quad X[i] \leftarrow \left\lfloor \frac{c - \sum_{j=1}^{i-1} L[j]X[j]}{L[i]} \right\rfloor$
7 **fim**
8 **retornar** X

3.1.4 BB

Uma estrutura geral do processo de solução do problema da mochila pelo Branch and Bound pode ser construída como no algoritmo 4. Note que a função *Ordenar()* é uma função arbitrária que ordena as variáveis conforme os critérios estabelecidos para BB em 2.2.4, enquanto *CalcularLimitante()* é a função que aplica a equação (20). Ainda, observe que a escolha por ajustar o algoritmo de maneira a percorrer os ramos mesmo onde $x_1 = 0$ (a variável com maior fator λ) implica no aumento muito expressivo do tempo de execução e custo computacional, contudo nos garante que o processo terá realizado a enumeração **implícita** de toda a árvore de soluções. Tal escolha foi realizada para proveito de melhores análises na subseção 4.2.

Algoritmo 4: Branch and Bound (BB)

Entrada: $n > 0$: número de variáveis (inteiro)
$c > 0$: capacidade da mochila (inteiro)
L: vetor de n inteiros positivos de tamanho dos itens
P: vetor de n inteiros positivos de prioridade dos itens
Saída: Vetor solução do problema da mochila pela BB
Dados: i: varíavel inteira contadora
X: vetor de n inteiros positivos de quantidade de itens
Λ: vetor de n racionais que contém o fator λ de cada variável
B: vetor de n elementos do tipo *Item*

Seja *Var* uma estrutura de dados que representa uma variável do problema, *Var* contém: l seu tamanho, p sua prioridade e λ seu fator. Pode-se operar um elemento do tipo *Var* da forma: $Var(l) \leftarrow 1$: atribui tamanho 1 à variável; $Var(p) \leftarrow 1$: atribui prioridade 1 à variável.

Seja *Item* uma estrutura de dados que representa uma item da mochila, *Item* contém: Um elemento *var* do tipo *Var* e *qty* a quantidade de itens alocados da variável na mochila. Pode-se operar um elemento do tipo *Item* da forma: $Item(qty) \leftarrow 1$: atribui 1 à quantidade de itens; $Item(var) \leftarrow var_2$: atribui os dados de uma variável ao item; $Item(var)(\lambda) \leftarrow 1$: atribui 1 ao fator da variável.

Função `MaximizarItem` (B, c, k) **:**
  Entrada: B: Um vetor de n elementos do tipo *Item*; c: capacidade da mochila; k: nível final
  Resultado: quantidade x de itens da variável alocados devidamente da mochila
  $x \leftarrow \left\lfloor \frac{c - \sum_{i=1}^{k} B[i](var)(l) \cdot B[i](qty)}{B[k](var)(l)} \right\rfloor$
  retornar x
fim
Função `MaximizarRamo` (B, c, n, k) **:**
  Entrada: B: Um vetor de n elementos do tipo *Item*; c: capacidade da mochila; n: total de variáveis; k: nível inicial
  Resultado: Novo B com elementos atualizados e S_p soma das prioridades acumuladas
  para $i \leftarrow k$ **até** n **faça**
    $B[i](qty) \leftarrow$ `MaximizarItem` (B, c, i)
  fim
  $S_p = \sum_{i=1}^{n} [B[i](qty) \cdot B[i](var)(p)]$
  retornar (B, S_p)
fim
Função `DescerNivel` (B, n) **:**
  $k \leftarrow n$
  enquanto $k > 1$ *e* $B[k](qty) \leq 0$ **faça**
    $k \leftarrow k - 1$
  fim
  $B[k](qty) \leftarrow B[k](qty) - 1$
  retornar (B, k)
fim

para $i \leftarrow 1$ **até** n **faça**
  $\Lambda[i] \leftarrow \frac{P[i]}{L[i]}$
fim
$(L, P) \leftarrow \underline{\text{Ordenar}}\ (L, \Lambda)$
para $i \leftarrow 1$ **até** n **faça**
  $B[i](var)(l) \leftarrow L[i]$
  $B[i](var)(p) \leftarrow P[i]$
  $B[i](var)(\lambda) \leftarrow \Lambda[i]$
  $B[i](qty) \leftarrow 0$
fim
$(B, k) \leftarrow$ `MaximizarRamo` $(B, c, n, 1)$
$S \leftarrow B$
$\bar{f} \leftarrow \sum_{i=1}^{n} B[i](qty) \cdot B[i](var)(p)$
$(B, k) \leftarrow$ `DescerNivel` (B, n)
enquanto $B[1](qty) \geq 0$ **faça**
  se $B[k](qty) \geq 0$ *e* $k < n$ **então**
    $L \leftarrow \underline{CalcularLimitante}(B, c, k)$
    se $L > \bar{f}$ **então**
      $(B, f) \leftarrow$ `MaximizarRamo` $(B, c, n, k + 1)$
      se $f > \bar{f}$ **então**
        $\bar{f} \leftarrow f$
        $S \leftarrow B$
      fim
    fim
  fim
  $(B, k) \leftarrow$ `DescerNivel` (B, n)
fim
retornar S

3.2 Implementação em linguagem C

Uma possível implementação em linguagem C pode ser obtida trabalhando sob os algoritmos providos na subseção 3.1 pela prática de programação em baixo nível que C pode promover, utilizando estruturas abstratas de dados para organizar os conjuntos de informações e mesmo simular a ramificação da árvore do Branch and Bound. Neste artigo consideramos o seguinte módulo de funções `heuristics.h` na figura 1,

```
#ifndef HEURISTICS_H
#define HEURISTICS_H

typedef void * Problem;
typedef void * Solution;

Problem ReadProblem (char * filename);
void ExportSolution (char * filename, Problem data, Solution solution);
void RemoveSolution (Solution solution);

Solution MOT  (Problem data);
Solution MOP  (Problem data);
Solution MOPT (Problem data);
Solution BranchAndBound (Problem data, char * output_log_dir, char * log_files_name);

#endif
```

Figura 1: Módulo de funções que provê uma abstração com as heurísticas

A partir deste módulo são definidos os tipos abstratos de dados `Problem` e `Solution` que estão implementados nas figuras 2(a) e (b). Por sua vez, as funções que representam as heurísticas calculam suas devidas soluções em cima destes ponteiros to tipo `void`, lidos em `ReadProblem` conforme as convenções de geração dos problemas estabelecidas (seção 4) e retornam a solução também abstrata. Como forma de saída destes valores, as funções `ExportSolution` e `RemoveSolution` controlam a formatação escolhida para exportação e desalocação de memória, respectivamente.

```
typedef struct StBackpack {
    int variables;
    int capacity;
    Var ** items;
} Backpack;
```

(a)

```
typedef struct StSolData {
    int * solution;
    int occupied_space;
    int accumulated_priority;
    double execution_time;
} SolData;
```

(b)

```
typedef struct StVar {
    int qty;
    int value;
    int priority;
    double factor;
} Var;
```

(c)

Figura 2: Implementação dos tipos abstratos de dados

3.2.1 Função MOT

Conforme os passos apresentados no algoritmo 1, uma implementação da função MOT pode ser obtida convertendo os conceitos vetoriais do algoritmo organizados dentro da estrutura em figura 2(c). Desta forma, primeiro deve-se utilizar algum algoritmo de ordenação que possa comparar cada item, como o `QuickSort`, considerando a comparação por valor de cada item (l_i). Então calcular diretamente o resultado de solução para cada item n vezes percorrendo e atribuindo ao vetor de solução cada resultado. Para otimizar cálculos, podem ser criadas variáveis acumuladoras `occupied` e `acc_priority` que armazenam o espaço atual ocupado na mochila e a prioridade acumulada atual da mochila em cada iteração, respectivamente. Assim, a figura 3 implementa.

```
1  Solution MOT (Problem data)
2  {
3      struct timespec start, end;
4      double elapsed;
5      clock_gettime(CLOCK_MONOTONIC, &start);
6
7      Backpack * pack           = (Backpack *) data;
8      SolData  * sol            = malloc(sizeof(SolData));
9      sol->solution             = malloc(sizeof(int) * (pack->variables));
10
11     QuickSort((Array) pack->items, CompareByValue, 0, pack->variables - 1);
12
13     int capacity              = pack->capacity;
14     int occupied              = 0;
15     int acc_priority          = 0;
16     sol->solution[0]          = floor(capacity / pack->items[0]->value);
17     pack->items[0]->qty       = sol->solution[0];
18     occupied                 += sol->solution[0] * pack->items[0]->value;
19     acc_priority             += sol->solution[0] * pack->items[0]->priority;
20     for (int i = 1; i < pack->variables; i++)
21     {
22         sol->solution[i]      = floor((capacity  - occupied)/ pack->items[i]->value);
23         occupied             += sol->solution[i] * pack->items[i]->value;
24         acc_priority         += sol->solution[i] * pack->items[i]->priority;
25         pack->items[i]->qty   = sol->solution[i];
26     }
27     sol->occupied_space       = occupied;
28     sol->accumulated_priority = acc_priority;
29     clock_gettime(CLOCK_MONOTONIC, &end);
30     elapsed = (end.tv_sec - start.tv_sec);
31     elapsed += (end.tv_nsec - start.tv_nsec) / 1000000000.0;
32     sol->execution_time       = elapsed;
33
34     return (Solution) sol;
35 }
```

Figura 3: Função MOT implementada em linguagem C

3.2.2 Função MOP

Conforme os passos apresentados no algoritmo 2, uma implementação da função MOP pode ser obtida convertendo os conceitos vetoriais do algoritmo organizados dentro da estrutura em figura 2(c). Desta forma, primeiro deve-se utilizar algum algoritmo de ordenação que possa comparar cada item, como o `QuickSort`, considerando a comparação por prioridade de cada item (p_i). Então calcular diretamente o resultado de solução para cada item n vezes percorrendo e atribuindo ao vetor de solução cada resultado. Para otimizar cálculos, podem ser criadas variáveis acumuladoras `occupied` e `acc_priority` que armazenam o espaço atual ocupado na mochila e a prioridade acumulada atual da mochila em cada iteração, respectivamente. Em termos de implementação, a função MOP difere da MOT apenas pela função de comparação que é passada ao algoritmo de ordenação para utilização. Para esta, agora é necessário uma função de comparação de prioridade dos itens, como implementado por `CompareByPriority`. Então, a linha 11 do código na figura 3 toma a forma abaixo.

```
QuickSort((Array) pack->items, CompareByPriority, 0, pack->variables - 1);
```

3.2.3 Função MOPT

Conforme os passos apresentados no algoritmo 3, uma implementação da função MOPT pode ser obtida convertendo os conceitos vetoriais do algoritmo organizados dentro da estrutura em figura 2(c). Desta forma, primeiro deve-se utilizar algum algoritmo de ordenação que possa comparar cada item, como o `QuickSort`, considerando a comparação por fator de cada item ($\lambda_i = \frac{p_i}{l_i}$), portanto, antes, um laço de repetição de n iterações calcula os fatores antes da ordenação. Em seguida, pode-se calcular diretamente o resultado de solução para cada item n vezes percorrendo e atribuindo ao vetor de solução cada resultado. Também, para otimizar cálculos, podem ser criadas variáveis acumuladoras `occupied` e `acc_priority` que armazenam o espaço atual ocupado na mochila e a prioridade acumulada atual da mochila em cada iteração, respectivamente. Em termos de implementação, a função MOPT difere da MOT e MOP pelo cálculo dos fatores λ de cada variável e posteriormente a utilização destes na passagem da função de comparação ao algoritmo de ordenação para utilização. Tem-se então que a linha que linha 11 do código presente na figura 3 é expandida com um laço de repetição para os cálculos e então é aplicada a ordenação como apresentado abaixo.

```
for (int i = 0; i < pack->variables; i++)
    pack->items[i]->factor = (double) pack->items[i]->priority / pack->items[i]->value;

QuickSort((Array) pack->items, CompareByFactor, 0, pack->variables - 1);
```

3.2.4 Função BranchAndBound

Conforme os passos apresentados no algoritmo 4, uma implementação da função BranchAndBound pode ser obtida primeiramente implementando a estrutura apresentada na figura 2(c), mas também junto a outras estruturas que possam permitir as operações de ramificação da árvore, como apresentadas na figura 4.

```
1 struct Variable {
2     int weight;
3     int value;
4     double factor;
5 };
6 typedef struct Item {
7     int quantity;
8     struct Variable variable;
9 } item;
```

(a)

```
1 typedef void * Branch;
2 typedef item * BranchImpl;
```

(b)

Figura 4: Estruturas e tipos de dados implementados que definem um item da mochila e um ramo

Para esta implementação, e também seguindo a organização proposta pelo algoritmo, outra funções auxiliares são definidas na figura 5: (a) `MaximizeItem` calcula a quantidade máxima que o item no nível k pode ocupar com o espaço que há livre na mochila, conforme descrita na equação (2); (b) `MaximizeBranch` invoca `MaximizeItem` para todos os itens a partir do nível $InitialK$ e considerando todos os outros itens anteriores fixos, isto é, calcular o restante do ramo que ainda não foi processado, ainda retornando a prioridade acumulada atual ao fim do processo; (c) `findNonZeroItem` realiza a busca pelo primeiro item de quantidade não-nula mais distante da raiz da árvore no ramo atual/ativo, retorna o índice k do nível do item; (d) `DownLevel` invoca `findNonZeroItem` e diminui uma unidade da quantidade do item retornado pela busca, retorna o índice k do nível do item.

```
int MaximizeItem(Branch branch, double c, int k)
{
    BranchImpl b = (BranchImpl) branch;
    int qtd;
    int occupied_space = 0;

    for (int i = 0; i <= k - 1; i++)
        occupied_space +=  b[i].variable.value * b[i].quantity;

    qtd = floor((c - occupied_space ) /
                (b[k].variable.value));
    return qtd;
}
```

(a)

```
int MaximizeBranch(Branch branch, double c, int VARS_AMOUNT, int InitialK)
{
    BranchImpl b = (BranchImpl) branch;
    int accumulated_priorities = 0;
    int i;
    for (i = 0; i < InitialK; i++)
        accumulated_priorities += b[i].quantity * b[i].variable.weight;
    for (i = InitialK; i < VARS_AMOUNT; i++)
    {
        b[i].quantity = MaximizeItem(b, c, i);
        accumulated_priorities += b[i].quantity * b[i].variable.weight;
    }
    return accumulated_priorities;
}
```

(b)

```
int findNonZeroItem(Branch branch, int VARS_AMOUNT)
{
    BranchImpl b = (BranchImpl) branch;
    int k;
    for (k = VARS_AMOUNT - 1; k > 0 && b[k].quantity <= 0; k--){};
    return k;
}
```

(c)

```
int DownLevel(Branch branch, int VARS_AMOUNT)
{
    BranchImpl b = (BranchImpl) branch;
    int k = findNonZeroItem(b, VARS_AMOUNT);
    b[k].quantity--;
    return k;
}
```

(d)

Figura 5: Funções operacionais auxiliares implementadas do Branch and Bound

Ainda, como característica do método, tem-se a função do limitante superior deduzida em 2.2.4.1, `CalculateBound`. A função calcula o limitante superior da prioridade acumulada na mochila a partir do índice k, assumindo que este item e todos os anteriores já foram alocados na mochila previamente, portanto estão fixos. Assim é implementada na figura 6.

Tendo todas as funções auxiliares implementadas, a função `BranchAndBound` segue a mesma estrutura de operações funcionais apresentada em seu algoritmo e pode ser validada no repositório [5]. Outras funções menos relevantes, `allocateBranch`, `copyBranch` e `Accumulated Priorities` são utilizadas à parte para contornar limitações da linguagem de programação ao, respectivamente: alocar a memória do ramo `current` que estará simulando o ramo ativo, e o ramo de solução `SolBranch`; a segunda para realizar a cópia de um ramo inteiramente a outro; a terceira para calcular a prioridade total acumulada no ramo (atual da mochila).

```c
int CalculateBound(Branch branch, double c, int k)
{
    BranchImpl b = (BranchImpl) branch;
    int accumulated_weights = 0;
    int occupied_space      = 0;

    int p_k = b[k].variable.weight;
    int l_k = b[k].variable.value;
    int hatx_k = b[k].quantity + 1;
    double nextFactor = b[k+1].variable.factor;

    for (int i = 0; i <= k-1; i++)
    {
        accumulated_weights += b[i].variable.weight * b[i].quantity;
        occupied_space      += b[i].variable.value  * b[i].quantity;
    }

    double barF = accumulated_weights + p_k * (hatx_k - 1) + nextFactor * (c - occupied_space - l_k * (
        hatx_k - 1));
    int L = floor(barF + 1);

    return L;
}
```

Figura 6: Função para cálculo do limitante superior implementada em linguagem C

4 Simulações

De modo a analisar a eficiência de cada uma das quatro heurísticas do artigo em aplicações práticas de possíveis problemas reais, foram geradas seis classes diferentes de problemas, cada uma com cem exemplares distintos gerados randomicamente. São as classes:

- Classe P1: Mochilas de 100 itens com capacidade de valor 500, os itens devem ter tamanho entre 10 e 200 e as prioridades valores entre 10 e 100. Deverão ser gerados 100 exemplos.
- Classe P2: Mochilas de 100 itens com capacidade de valor 500, os itens devem ter tamanho entre 200 e 500 e as prioridades valores entre 10 e 100. Deverão ser gerados 100 exemplos.
- Classe M1: Mochilas de 1.000 itens com capacidade de valor 500, os itens devem ter tamanho entre 10 e 200 e as prioridades valores entre 10 e 100. Deverão ser gerados 100 exemplos.
- Classe M2: Mochilas de 1.000 itens com capacidade de valor 500, os itens devem ter tamanho entre 200 e 500 e as prioridades valores entre 10 e 100. Deverão ser gerados 100 exemplos.
- Classe G1: Mochilas de 10.000 itens com capacidade de valor 500, os itens devem ter tamanho entre 10 e 200 e as prioridades valores entre 10 e 100. Deverão ser gerados 100 exemplos.
- Classe G2: Mochilas de 10.000 itens com capacidade de valor 500, os itens devem ter tamanho entre 200 e 500 e as prioridades valores entre 10 e 100. Deverão ser gerados 100 exemplos.

Os 600 experimentos numéricos totais produzidos foram gerados a partir do seguinte código também em linguagem C na figura 7.

```c
typedef struct Class {
    int itens;
    int capacity;
    int values_range[2];
    int weights_range[2];
    char * name;
} class;

int randomIntRange (int min, int max){return min + random() % (max - min + 1);}

void GenerateIntProblem(char * filename, int VARS_TOTAL, int capacity, int * VARS_WEIGHTS_RANGE, int * VARS_VALUES_RANGE)
{
    FILE * file = fopen(filename, "w");

    fprintf(file, "%d %d %d %d %d %d\n", VARS_TOTAL, capacity, VARS_WEIGHTS_RANGE[0], VARS_WEIGHTS_RANGE[1], VARS_VALUES_RANGE[0], VARS_VALUES_RANGE[1]);
    for (int i = 0; i < VARS_TOTAL; i++)
        fprintf(file, "%d %d\n", randomIntRange(VARS_WEIGHTS_RANGE[0], VARS_WEIGHTS_RANGE[1]), randomIntRange(VARS_VALUES_RANGE[0], VARS_VALUES_RANGE[1]));
    fclose(file);
}
```

(a)

```c
class P1 = {100,    500, {10,   200}, {10, 100}, "P1"};
class P2 = {100,    500, {200,  500}, {10, 100}, "P2"};
class M1 = {1000,   500, {10,   200}, {10, 100}, "M1"};
class M2 = {1000,   500, {200,  500}, {10, 100}, "M2"};
class G1 = {10000,  500, {10,   200}, {10, 100}, "G1"};
class G2 = {10000,  500, {200,  500}, {10, 100}, "G2"};

class * classes[] = {&P1, &P2, &M1, &M2, &G1, &G2};
```

(b)

Figura 7: Código em linguagem C gerador de problemas das seis classes

Na figura 7(a) e (b) é apresentada a estrutura parcial do código gerador de problemas. Primeiramente, uma estrutura de dados `Class` é definida para organizar as informações das classes de problemas, em seguida, a randomização de dados gerados é feita utilizando as funções `srandom()`, `random()` e `time()` da biblioteca `time.h`, nativa da linguagem C. A cada execução do código, uma nova semente de tempo é fornecida por `time()` e, a partir desta, os números gerados pela função `random()` são aleatorizados.

Tendo os 600 problemas organizados devidamente por sua classe em um diretório apenas de problemas. Por convenção do artigo, define-se um nome de problema da forma: ”problem-<classe>-<número>.txt”, onde <classe> deve assumir um dos tipos {P1, P2, M1, M2, G1, G2} e <número> é a numeração do problema dentro da classe, sendo $001, 002, 003, \cdots, 098, 099, 100$. A execução do código-fonte computa todos os problemas organizados em um diretório de problemas com as quatro heurísticas e, então, exporta os dados processados em um diretório de saída com formatação convencionada para compilação de dados.

4.1 Equipamento

As simulações foram realizadas em uma máquina equipada com um processador `Intel(R) Core(TM) i5-10600K CPU @ 4.10GHz 4.10 GHz`, `16GB` de memória RAM `Corsair Vengeance RT (2x8GB), 3600MHz, DDR4, CL18` e uma placa de vídeo `Gigabyte GeForce® GTX 1060 WINDFORCE OC 6G`. O sistema operacional utilizado foi `Windows 10 Pro (versão 21H2)` de `64 bits`. Placa-mãe `Gigabyte Z490M Gaming X`. Virtualmente foi utilizado o Windows Subsystem for Linux (WSL2) para compilação e execução do código-fonte em ambiente nativo Ubuntu.

4.2 Resultados

Todos os resultados obtidos pelos códigos em 3.2 foram integralmente exportados para um repositório em nuvem do autor [5] e podem ser validados abertamente junto ao códigos-fonte implementado. Em suma, foram obtidos os seguintes resultados por classe ao avaliar o tempo médio de execução dos programas por classe em cada uma das heurísticas avaliadas, na tabela 1. Note que, enquanto MOT, MOP, MOPT mantêm consistência no crescimento do tempo médio entre as classes, BB obteve crescimento muito maior, tendo em G2 tempo médio de aproximadamente 1 hora, 18 minutos e 8 segundos.

Classe	MOT (s)	MOP (s)	MOPT (s)	BB (s)
P1	11,70E-06	12,76E-06	13,37E-06	91,64E-04
P2	11,79E-06	12,50E-06	13,14E-06	89,14E-04
M1	1,27E-04	1,32E-04	1,57E-04	4,40E-02
M2	1,30E-04	1,27E-04	1,49E-04	68,41E-02
G1	13,80E-04	13,96E-04	17,67E-04	12,86E+00
G2	17,56E-04	14,15E-04	17,99E-04	46,88E+02

Tabela 1: Tempo médio de execução por problema

Analisando a tabela 2, pode-se observar o desempenho das heurísticas de maneira absoluta em termos dos problemas produzidos, comparando o que seria a eficiência destas numericamente aplicadas em dois fatores: a maximização da prioridade acumulada maximizada e o espaço restante na mochila que será desperdiçado.

Classe	MOT		MOP		MOPT		BB	
	$\bar{S}_p$	$\bar{R}$	$\bar{S}_p$	$\bar{R}$	$\bar{S}_p$	$\bar{R}$	$\bar{S}_p$	$\bar{R}$
P1	153,77	1,32	960,58	4,77	3126,86	5,38	3134,02	2,73
P2	58,02	2,39	127,19	79,66	186,27	66,94	191,48	36,78
M1	164,34	0,00	843,44	3,48	4422,45	1,91	4428,23	0,29
M2	52,92	0,04	137,27	68,59	196,78	89,22	199,80	63,23
G1	161,66	0,00	760,90	2,99	4934,00	0,00	4934,00	0,00
G2	55,49	0,00	138,99	68,94	199,44	98,00	200,00	94,96

Tabela 2: Médias aritméticas das prioridades acumuladas ($\bar{S}_p$) e dos espaços restantes nas mochilas ($\bar{R}$)

A tabela 3 apresenta uma avaliação das diferenças de prioridades acumuladas média das soluções das heurísticas comparadas com a solução fornecida pelo Branch and Bound, também os gráficos na figura 8(a), (b) e (c) mostram graficamente os dados. Para a análise de desempenho, tem-se que ΔS_p é diferença das médias aritméticas das prioridades acumuladas ($\bar{S}_p$) das soluções pelo BB em relação à MOT, MOP, ou MOPT; enquanto que $\Delta S_{p\%}$ é dado pela média aritmética dos fatores $1 - \gamma_p$ tal que γ_p é a razão da prioridade acumulada na solução fornecida por MOT, MOP ou MOPT pela solução fornecida por BB.

Classe	MOT		MOP		MOPT	
	ΔS_p	$\Delta S_{p\%}$	ΔS_p	$\Delta S_{p\%}$	ΔS_p	$\Delta S_{p\%}$
P1	2980,25	94,71%	2173,44	68,36%	7,16	0,26%
P2	133,46	69,68%	64,29	33,89%	5,21	2,76%
M1	4263,89	96,26%	3584,79	80,73%	5,78	0,14%
M2	146,88	73,50%	62,53	31,31%	3,02	1,51%
G1	4772,34	96,72%	4173,1	84,57%	0	0,00%
G2	144,51	72,26%	61,01	30,51%	0,56	0,28%

Tabela 3: Diferença média de prioridade acumulada por problema

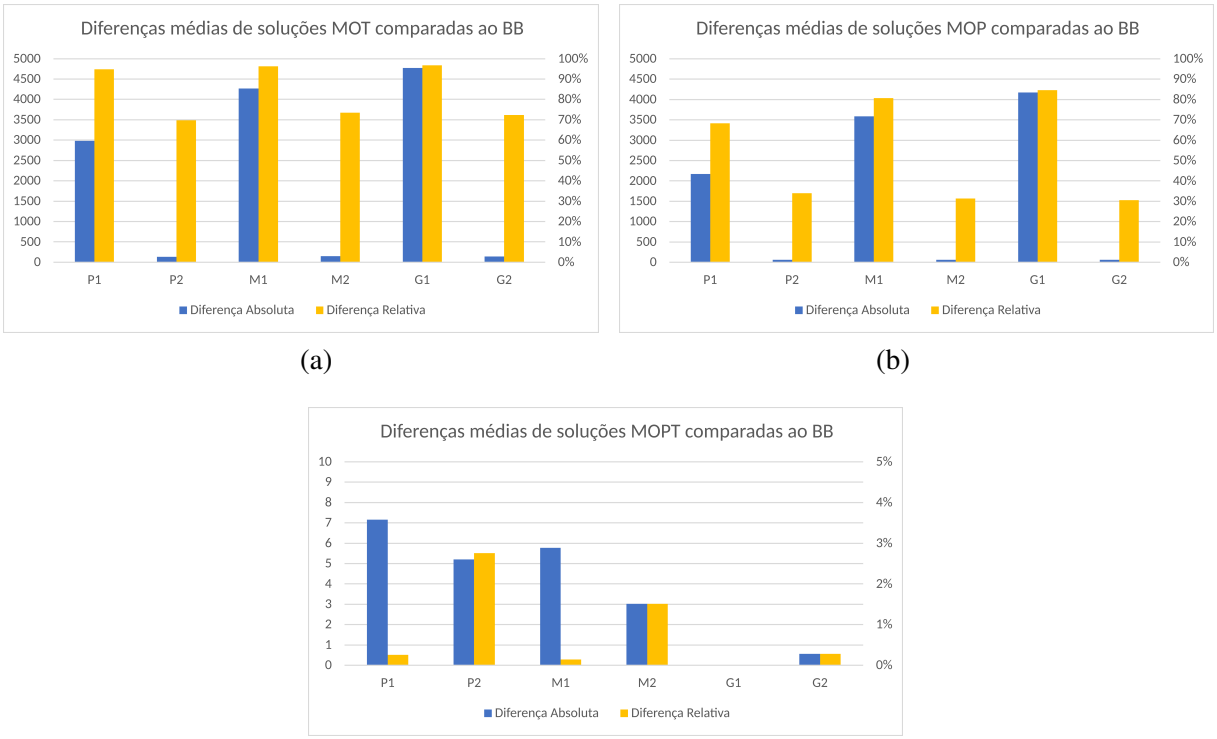

Figura 8: Gráficos de diferenças médias de prioridades acumuladas por classes de problemas e heurísticas

5 Conclusões

Este artigo se propôs a estudar quatro tipos de heurísticas comumente utilizadas para tentar de problemas de otimização, como no caso do problema da mochila, o qual é um desafio desafio diário para a indústria em geral, mas especificamente para às grandes transportadoras e logísticas terrestres, marítimas e aeroespaciais. Nas simulações desenvolvidas viu-se que o método Branch and Bound de fato foi eficaz e trouxe soluções ótimas com tempo de execução aceitável ao comparar com as demais heurísticas, especialmente nas classes de problemas com menor quantidade de variáveis e com itens de tamanhos menores, isto é, as classes pequenas do tipo 1. Ficam destacados os problemas da classe G1, os quais todos os exemplares puderam obter suas soluções ótimas diretamente pela MOPT.

Ainda, também, a classe de problemas G2 toma relevância por evidenciar as discrepâncias de tempo de execução e custo computacional do Branch and Bound em problemas com grande quantidade de variáveis e com tamanhos próximos da capacidade da mochila; o padrão formado pelo resultados de prioridade acumulada esterem todos na vizinhança de $\bar{S}_p = 200$; e também a diferença mínima entre a solução exata produzida pelo Branch and Bound com a solução heurística dada pela MOPT, que também se encontra na vizinhança de $\Delta S_p = 0,56$. Mostrando que para esta classe G2, a MOPT é notoriamente indicada.

Sem a implementação do método Branch and Bound seria inviável a produção deste artigo e, por isto, as contribuições do Prof. Dr. Robinson Hoto em sala de aula e em comunicação pessoal foram imprescindíveis para a construção e aplicação da função do limitante superior. A princípio, esta foi inicialmente complexa e, intuitivamente, parecia tomar um custo computacional muito maior, contudo conforme n o número de variáveis cresce, a árvore de enumerações parece tomar forma de $n!$ número de ramos, o que torna virtualmente inviável seu procedimento explícito. Desta forma, sendo um método interessante e de fácil implementação para encontrar soluções.

O artigo ainda levanta discussões a respeito do custo-benefício oferecido pelo Branch and Bound em determinados tipos de problemas e como contorná-los. Uma vez que em exemplares são destacados casos particulares de ótima eficiência das heurísticas e casos de péssima eficiência. Ao analisa-los, podemos pensar em um método que combine e aproveite os resultados das heurísticas MOT, MOP e MOPT para gerar uma condição inicial de ramo melhor otimizada e, então, aplicar o Branch and Bound; outra, também, seria aproveitar os recursos de processamento paralelo e desenvolver um método de múltiplas ramificações que otimizasse seus procedimentos.

Como contribuições, espera-se que com as implementações dadas e simulações numéricas feitas, novos trabalhos a respeito de problemas de otimização sejam promovidos, permitindo tornar mais eficientes os algoritmos propostos a partir de análises dos dados simulados por esta produção.

6 Bibliografia

1 CORMEN, T. H. et al. *Algoritmos*. 3. ed. Rio de Janeiro: Elsevier Editora Ltda, 2012. ISBN 978-85-352-3699-6.

2 TAILLARD, E. D. *Design of Heuristic Algorithms for Hard Optimization: With Python Codes for the Travelling Salesman Problem*. Cham, Switzerland: Springer Cham, 2022. (Graduate Texts in Operations Research). ISSN 2662-6012. ISBN 978-3-031-13713-6.

3 KIANFAR, K. Branch-and-bound algorithms. In: ____. *Wiley Encyclopedia of Operations Research and Management Science*. Hoboken, NJ, USA: John Wiley & Sons, Ltd, 2011. ISBN 9780470400531. Disponível em: ⟨https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470400531.eorms0116⟩.

4 LAND, A. H.; DOIG, A. G. An automatic method of solving discrete programming problems. *Econometrica*, [Wiley, Econometric Society], v. 28, n. 3, p. 497–520, 1960. ISSN 00129682, 14680262. Disponível em: ⟨http://www.jstor.org/stable/1910129⟩.

5 PIETSIAKI, G. *ARTICLE Application of heuristics in numerical simulations for the integer knapsack problem*. [S.l.]: GitHub, 2023. ⟨https://github.com/gabrielpie6/ARTICLE-Application-of-heuristics-in-numerical-simulations-for-the-integer-knapsack-problem⟩.