LIII Simpósio Brasileiro de Pesquisa Operacional
João Pessoa - PB, 3 a 5 de novembro de 2021

Uma heurística ILS para o Problema da Mochila com Penalidades

Ana Flávia Ciríaco
Universidade Federal de Minas Gerais
Belo Horizonte - MG
anafciriaco@gmail.com

Thiago Ferreira de Noronha
Universidade Federal de Minas Gerais
Belo Horizonte - MG
tfn@dcc.ufmg.br

Eduardo Theodoro Bogue
Universidade Federal de Mato Grosso do Sul
Ponta Porã - MS
eduardo.theodoro@ufms.br

RESUMO

Neste trabalho, estudamos uma variante do Problema da Mochila, denominada de o Problema da Mochila com Penalidades. Nesta variante, acrescenta-se um conjunto de pares distintos de itens, chamados de pares de penalidades, de modo que um par de penalidade é composto por itens que, quando selecionados juntos para a solução, implicam no pagamento de uma penalidade ao custo da solução. Neste artigo, uma heurística baseada em Busca Local Iterada é proposta para o problema. Os resultados obtidos mostraram que a heurística desenvolvida obteve melhores soluções que o presente estado da arte da literatura para as instâncias conhecidas.

PALAVRAS CHAVE. Problema da mochila com penalidades, busca local iterada, metaheurística.

Área principal: Metaheurísticas.

ABSTRACT

This article tackles a variant of the Knapsack Problem, called the Knapsack Problem with Forfeits. In this variant, a set of distinct pairs of items is added, called forfeit pairs, so that a forfeit pair is composed of items that, when included together in the solution, imply the payment of a penalty at the cost of the solution. In this article, a heuristic based on Iterated Local Search is developed for the problem. The results obtained showed that the developed heuristic obtained better solutions than the best heuristic proposed in the literature for known instances.

KEYWORDS. Knapsack problem with forfeits, iterated local search, metaheuristics.

Main area: Metaheuristics.

1. Introdução

Problemas de otimização combinatória ocorrem em diversas situações práticas do dia a dia, como por exemplo em áreas de logística de transporte e distribuição, alocação de recursos, entre outras. Problemas de empacotamento e de roteamento de veículos são exemplos de problemas clássicos e muito bem estudados em otimização combinatória. Novas variantes destes problemas surgem à medida que aplicações exigem restrições adicionais ou relaxam algumas restrições do problema. Neste trabalho, estudamos uma variante do Problema da Mochila proposta em [Cerulli et al., 2020], denominada de o Problema da Mochila com Penalidades (em inglês, *Knapsack Problem with Forfeits* - KPF).

O Problema da Mochila com Penalidades é uma generalização do Problema da Mochila, que é definido sobre um conjunto de itens I, onde cada item $i \in I$ é associado a um peso $w_i \in \mathbb{N}$ e um benefício $p_i \in \mathbb{N}$. O Problema da Mochila consiste em encontrar um subconjunto $I' \subseteq I$, sujeito à soma dos pesos dos itens em I' não excederem uma capacidade $H \in \mathbb{N}$. O objetivo é encontrar um subconjunto I^*, de forma que a soma dos benefícios dos itens em I^* seja máxima.

No Problema da Mochila com Penalidades, acrescenta-se um conjunto F de pares distintos de itens, chamados de pares de penalidades, de modo que um par de penalidade $f \in F$ é composto por itens que, quando selecionados juntos para a solução, implicam no pagamento de uma penalidade $d_f \in \mathbb{N}$ ao custo da solução. A adição de pares de penalidade é motivada pela existência de problemas de otimização com escolhas mutualmente exclusivas. Nestes problemas, há uma coleção de pares de itens na qual no máximo um item de cada par pode fazer parte da solução [Yamada et al., 2002]. No KPF, essa restrição é atenuada: permite-se que itens antes mutualmente exclusivos sejam selecionados juntos mediante o pagamento de uma penalidade.

A estrutura desse problema pode ser utilizada como modelagem de diversas aplicações. Como exemplo, suponha que cada item seja uma máquina que precisa de um operador e um par indique máquinas que só podem ser operadas pela mesma pessoa. Nesse caso, na versão de escolhas mutualmente exclusivas, apenas uma das máquinas poderia ser utilizada. Ao se utilizar a versão atenuada, uma penalidade indicaria a contratação de um novo trabalhador para operar uma das máquinas do par e, assim, ambas as máquinas poderiam ser utilizadas.

O problema NP-difícil do Problema da Mochila [Karp, 1972] é um caso particular do Problema da Mochila com Penalidades quando $F = \emptyset$, logo o KPF também é NP-difícil. Como não existe uma técnica conhecida para projetar algoritmos exatos de tempo polinomial para problemas NP-difíceis, este trabalho se concentra em algoritmos heurísticos. Mais especificadamente, o trabalho propõe uma heurística de Busca Local Iterada (em inglês, *Iterated Local Search* - ILS) para o problema.

O restante deste artigo está organizado da seguinte forma. Trabalhos relacionados são discutidos na Seção 2. Na Seção 3 é descrita uma formulação de programação linear inteira apresentada em [Cerulli et al., 2020] para o problema, enquanto na Seção 4 a heurística ILS proposta é descrita. Por fim, os experimentos computacionais são relatados na Seção 5, enquanto observações e trabalhos futuros são apresentados na última seção.

2. Trabalhos Relacionados

Restrições de conflitos e de escolhas mutualmente exclusivas são adicionadas a diversos problemas clássicos de otimização combinatória, como por exemplo o Problema de Fluxo Máximo [Pferschy e Schauer, 2011; Şuvak et al., 2020] e o Problema do Empacotamento [Epstein e Levin, 2008; Sadykov e Vanderbeck, 2013]. Em especial, o Problema da Mochila com Grafos

de Conflito tem sido extensivamente estudado na literatura [Pferschy e Schauer, 2009; Hifi e Otmani, 2012; Bettinelli et al., 2017; Pferschy e Schauer, 2017; Gurski e Rehs, 2019; Coniglio et al., 2020]. No entanto, até o presente momento, pelo nosso conhecimento, apenas o trabalho de [Cerulli et al., 2020] trata especificamente o Problema da Mochila com Penalidades, no qual as restrições de conflito são suavizadas. Nesse trabalho, os autores propõem uma formulação de programação linear inteira para o problema, além de duas heurísticas, denominadas de *GreedyForfeits* e *CarouselForfeits*.

A heurística *GreedyForfeits* é um algoritmo guloso que, sequencialmente, e enquanto houver capacidade, insere itens na mochila. O critério guloso utilizado pelo algoritmo, denominado de critério da densidade, determina que o item escolhido em uma iteração é aquele de maior benefício incremental por unidade de peso. O benefício incremental de um item em um determinado instante é o valor que ele agregaria a solução se adicionado a mochila naquele momento, ou seja, o benefício incremental de um item $i \in I$ é igual a $p_i - (\sum_{f \in F} d_f, \forall f = \{i, j\} \in F \mid j \in S)$, onde S é o conjunto dos itens adicionados a solução até o momento.

Já a heurística *CarouselForfeits* utiliza-se do paradigma *Carousel Greedy* para melhorar a solução gulosa. O paradigma *Carousel Greedy*, originalmente proposto em [Cerrone et al., 2017], provê um *framework* generalizado para aprimorar algoritmos gulosos. O algoritmo parte da intuição de que as escolhas gulosas feitas nos primeiros passos podem levar a soluções com qualidade comprometida, dada a falta de conhecimento a princípio sobre a subsequente estrutura da solução. Assim sendo, *Carousel Greedy* propõe que escolhas iniciais sejam iterativamente reconsideradas e, eventualmente, substituídas por outras.

A heurística *CarouselForfeits* foi avaliada em um conjunto de instâncias, também apresentadas em [Cerulli et al., 2020], que são divididas entre pequenas e grandes instâncias. Para as instâncias pequenas, os autores mostram que a heurística apresentou um *gap* médio de 4.7% entre a solução obtida pela heurística *CarouselForfeits* e a melhor solução obtida pela resolução da formulação de programação linear inteira através do algoritmo de *branch-and-bound* do CPLEX, enquanto para as instâncias grandes o *gap* obtido foi de 4.5%.

3. Formulação de Programação Linear Inteira

Nesta seção, apresentamos uma formulação de programação linear inteira para o Problema da Mochila com Penalidades. A formulação apresentada foi descrita em [Cerulli et al., 2020] para o KPF. Sejam as variáveis binárias $x_i \in \{0, 1\}$, de modo que $x_i = 1$ se o item $i \in I$ for selecionado para a solução final e $x_i = 0$ caso contrário, e sejam as variáveis v_f, onde $v_f = 1$ caso os itens do par de penalidades $f \in F$ sejam selecionados para a solução final, e $v_f = 0$ caso contrário. Podemos formular o KPF como um problema de programação linear inteira definido por (1) - (5).

$$\text{Maximize} \quad \sum_{i \in I} p_i x_i - \sum_{f \in F} d_f v_f \tag{1}$$

$$\text{sujeito a} \quad \sum_{i \in I} w_i x_i \leq H \tag{2}$$

$$x_i + x_j - v_f \leq 1, \quad \forall f = \{i, j\} \in F \tag{3}$$

$$0 \leq v_f \leq 1 \quad \forall f \in F \tag{4}$$

$$x_i \in \{0, 1\} \quad \forall i \in I \tag{5}$$

A função objetivo (1) maximiza a soma do benefício dos itens selecionados, sujeito aos custos relacionados aos pares de penalidades. A restrição de capacidade é imposta pela desigualdade (2), enquanto as restrições em (3) garantem que caso ambos os itens de um par de penalidades $f \in F$ sejam selecionados, temos então que $v_f = 1$ e, portanto, a penalidade d_f é aplicada ao custo da solução. Por fim, as restrições (4) e (5) são restrições de integralidade. Essa formulação é resolvida pelo algoritmo de *branch-and-bound* do CPLEX.

4. Heurística *ILSForfeits*

Como a complexidade de pior caso dos algoritmos de *branch-and-bound* baseados na formulação PLI da Seção 3 cresce exponencialmente com o número de itens, nesta seção é proposta uma heurística denominada *ILSForfeits*, sendo esta baseada em uma Busca Local Iterada [Lourenço et al., 2003]. A heurística ILS é um método de trajetória desenvolvido em torno de uma busca local, que utiliza-se de mecanismos para escapar da estagnação em ótimos locais e, assim, explorar novas áreas da região factível. A heurística inicia sua busca a partir de uma solução inicial s, sobre a qual uma busca local é realizada. Em um algoritmo de busca local, é definido para cada solução uma vizinhança composta por um conjunto de soluções com características similares. Dada uma solução corrente, uma busca na vizinhança da solução é então realizada visando encontrar melhores soluções. Se tal solução vizinha é encontrada, torna-se a nova solução corrente e o procedimento é repetido novamente. Caso contrário, a solução corrente é um ótimo local em relação à vizinhança adotada.

Para que a heurística ILS não fique estagnada no ótimo local retornado pelo procedimento da busca local e consiga explorar novas vizinhanças, um procedimento chamado perturbação é aplicado a solução s retornada pela busca local com o intuito de modificá-la. Neste trabalho, o procedimento de perturbação realiza uma remoção aleatória de um porcentagem pré-definida de itens da solução. Assim, uma nova busca local pode ser aplicada a esta solução visando a obtenção de um novo ótimo local. A heurística ILS consiste então na execução de forma intercalada dos procedimentos de busca local e perturbação até que um critério de parada seja alcançado.

No procedimento de busca local descrito a seguir, a vizinhança de uma solução s é definida pela remoção de um item da solução seguida pela inserção de um conjunto de itens na solução, inclusive o conjunto vazio. O Algoritmo 1 apresenta o pseudocódigo do procedimento. A variável s' representa a solução corrente e a variável s'' representa a melhor solução encontrada, sendo ambas inicializadas com a solução recebida como parâmetro de entrada (linhas 1 e 2). O laço das linhas 4-20 é executado sempre que uma nova solução vizinha é encontrada. Nesse laço, inicialmente, os itens da solução são ordenados em ordem crescente de densidade (linha 6). A densidade de um item $i \in I$ corresponde ao seu benefício incremental por unidade de peso, ou seja, ela pode ser calculada por $\frac{p_i - (\sum_{f \in F} d_f, \forall f = \{i,j\} \in F | j \in S)}{w_i}$. Em seguida, o laço das linhas 9-20 é executado até que uma nova solução vizinha seja encontrada. Para encontrar uma solução vizinha, o item de menor densidade pertencente a solução s' e que ainda não foi avaliado é removido da solução (linhas 10-12). Depois, de maneira gulosa, os itens de maior densidade que não estão na solução corrente são adicionados a solução caso a capacidade da mochila não seja ultrapassada (linhas 13-15). Quando a solução obtida é diferente da solução corrente, tem-se uma nova solução vizinha (linhas 16-19) e o procedimento volta ao passo 5 para uma nova busca na vizinhança da solução vizinha encontrada. Caso contrário, o algoritmo retorna ao passo 10 e o próximo item de menor densidade é removido da solução corrente. A melhor solução vigente s'' é atualizada sempre que o valor de uma solução vizinha encontrada é maior que o valor da melhor solução vigente (linhas 18-19). O procedimento de busca local em s' é encerrando quando toda solução obtida através da remoção do item de menor densidade e posterior inclusão dos itens de maior densidade que não fazem parte da solução não

modifica a solução s', retornando então na linha 21 a última solução s' encontrada e a melhor solução s'' obtida durante o procedimento.

Algoritmo 1: Busca Local

Entrada: $Inst$: Instância do Problema da Mochila com Penalidades
s: Uma solução viável
Saída: (s',s''): Duas soluções viáveis, sendo s' a última solução visitada e s'' a melhor solução encontrada

```
1  s' ← s
2  s'' ← s
3  s'_mudou ← true
4  while s'_mudou = true do
5      s'_mudou ← false
6      lista_itens ← itens de s' em ordem crescente de densidade
7      num_itens_lista ← tamanho de lista_itens
8      i ← 1
9      while i ≤ num_itens_lista and s'_mudou = false do
10         pior_item ← i-ésimo item de lista_itens
11         s'_aux ← s'
12         s' ← s' − {pior_item}
13         while houver item em I\s' cujo benefício incremental seja positivo e o peso
             não exceda a capacidade da mochila da solução s' do
14             melhor_item ← item de maior densidade em I\s'
15             s' ← s' + {melhor_item}
16         if s' ≠ s'_aux then
17             s'_mudou ← true
18             if valor de s' > valor de s'' then
19                 s'' ← s'
20         i ← i + 1
21 return (s',s'')
```

Como dito, o ILS utiliza uma solução inicial s como ponto de partida. Na heurística ILS desenvolvida, s consiste da solução obtida pela heurística gulosa *GreedyForfeits*. O pseudocódigo da heurística *GreedyForfeits* é apresentado no Algoritmo 2. A heurística *GreedyForfeits* recebe como parâmetro de entrada uma instância do KPF e retorna uma solução viável para essa instância. O laço das linhas 3-20 é responsável por construir a solução. A cada iteração, um conjunto denominado X_{iter} é construído a partir dos itens da instância que ainda não foram selecionados e cujos pesos não excedam a capacidade restante da mochila (linhas 4-7). Os itens desse conjunto são então avaliados quanto aos seus respectivos benefícios incrementais (linhas 10-15), de modo que o item de maior densidade é então inserido na solução (linhas 16-20). A heurística termina sua execução quando $X_{iter} = \emptyset$, ou seja, todos os itens que não fazem parte da solução excedem a capacidade

remanescente da mochila (linhas 8-9).

Algoritmo 2: GreedyForfeits

Entrada: $Inst$: Instância do Problema da Mochila com Penalidades
Saída: Uma solução S viável

```
1  S ← ∅
2  b ← H
3  while I\S ≠ ∅ do
4      X_iter ← ∅
5      for i ∈ X do
6          if w_i ≤ b and i ∉ S then
7              X_iter ← X_iter ∪ {i}
8      if X_iter = ∅ then
9          return S
10     for i ∈ X_iter do
11         p'_i ← p_i
12         for F_k = {i, j} ∈ F do
13             if j ∈ S then
14                 p'_i ← p'_i − d_k
15         ratio_i ← p'_i / w_i
16     i* ← argmax[ratio]
17     if ratio_{i*} < 0 then
18         return S
19     S ← S ∪ {i*}
20     b ← b − w_{i*}
21 return S
```

Por fim, o pseudocódigo da heurística ILS proposta é apresentado no Algoritmo 3. Conforme mencionado, o algoritmo inicia a partir da solução obtida pela heurística gulosa *GreedyForfeits*. Atribuímos essa solução à s, que representa a solução corrente (linha 1). Em seguida, o procedimento de busca local é aplicado a s na linha 2. Como visto, o procedimento de busca local retorna a última solução alcançada e a solução de maior custo encontrada durante o procedimento, sendo estas armazenadas em s' e s'', respectivamente. Atribuímos s' à s (linha 3), que é variável de solução corrente, e s'' à s^* (linha 4), que é a variável de melhor solução. Em seguida, o laço das linhas 6-16 é executado enquanto o número de iterações sem melhoria não alcançar o limite pré-estabelecido. A cada iteração, um procedimento de perturbação é aplicado (linhas 8-11) e, como explicado anteriormente, remove aleatoriamente uma porcentagem α de itens da solução corrente. Após cada perturbação, o procedimento de busca local é aplicado a solução corrente (linha 12). A solução corrente sempre é a solução s' retornada pela última chamada do algoritmo de busca local (linha 13). Se a busca local retorna uma solução $s'' > s^*$, s^* é atualizado e o número de iterações sem melhoria é zerado (linhas 14-16). Ao final, após o número de iterações sem melhoria na solução

ser atingido, a melhor solução s^* é retornada (linha 17).

Algoritmo 3: ILSForfeits

Entrada: $Inst$: Instância do Problema da Mochila com Penalidades
Saída: Uma solução s^* viável

```
1  s ← greedyForfeits(Inst)
2  (s',s'') ← localSearch(Inst,s)
3  s ← s'
4  s* ← s''
5  i ← 0
6  while i < limite de iterações sem melhoria do
7      i++
8      size ← número de itens em s
9      for j ← 1 to α × size do
10         x ← item aleatório de s
11         s = s − {x}
12     (s',s'') ← localSearch(Inst,s)
13     s ← s'
14     if s'' > s* then
15         s* ← s''
16         i ← 0
17 return s*
```

Neste trabalho, o valor de α foi definido experimentalmente em 0.1, isto é, o procedimento de perturbação exclui 10% dos itens da solução, e a condição de parada foi estabelecida em 1000 iterações sem melhoria na solução.

5. Experimentos Computacionais

Os experimentos computacionais relatados nesta seção avaliam o desempenho do algoritmo de *branch-and-bound* do CPLEX com base na formulação (1) - (5), da heurística ILSForfeits, e da heurística *CarouselForfeits* proposta em [Cerulli et al., 2020] para o problema. Nos referimos a esses algoritmos como B&B, ILSForfeits e *CarouselForfeits*, respectivamente. Esses algoritmos foram implementados em C++ e compilados com o GNU *gcc* versão 6.3. A formulação de PLI foi resolvida pelo IBM CPLEX versão 12.10, com a configuração padrão, e utilizando um tempo limite de execução de 3 horas. Todas os experimentos foram realizadas em um único núcleo de uma máquina Intel Core i5 com 1.60 GHz de velocidade de clock e 16 GB de memória RAM.

Para a avaliação experimental, foi utilizado o conjunto de instâncias proposto em [Cerulli et al., 2020]. Ele é dividido em 4 conjuntos de instâncias, com o número de itens $n \in \{500, 700, 800, 1000\}$. Para cada instância, o número de pares de penalidades é igual a $n \times 6$, e a capacidade H da mochila é definida em $n \times 3$. Cada conjunto é composto de 8 instâncias, totalizando 32 instâncias. Comparamos os custos das soluções heurísticas com o custo da melhor solução obtida pelo CPLEX. Para isso, definimos o *gap* da solução como $1 - (Sol_{Heuristica}/Sol_{CPLEX})$, onde $Sol_{Heuristica}$ consiste no custo da solução da heurística utilizada e Sol_{CPLEX} no custo da melhor solução obtida pelo CPLEX.

A Tabela 1 contém os resultados das instâncias para as quais o CPLEX encontrou a solução ótima, sendo estas as instâncias pequenas, com 500 e 700 itens. Na Tabela 2 são apresentados os valores médios do *gap* em relação ao custo da solução ótima e o tempo médio de

execução das 8 instâncias de cada conjunto. Para as instâncias menores com 500 e 700 itens, podemos observar através das Tabelas 1 e 2 que o *gap* médio da heurística ILSForfeits foi de 1.86%, enquanto a heurística *CarouselForfeits* obteve um *gap* médio de 4.83%. Ademais, o maior *gap* obtido pela heurística ILSForfeits foi de 3.88%, valor inferior ao *gap* médio da heurística *CarouselForfeits*. Todavia, com relação ao tempo de execução, podemos observar que o tempo médio da heurística ILSForfeits proposta foi de 54.66 segundos, enquanto a heurística *CarouselForfeits* obteve um tempo médio de execução inferior a 1 segundo.

Instância		**B&B**	**CarouselForfeits**			**ILSForfeits**		
Nº de itens	Id	Solução	Solução	Gap (%)	Tempo (s)	Solução	Gap (%)	Tempo (s)
500	1	2626	2510	4.42	0.11	2524	3.88	25.10
	2	2660	2556	3.91	0.01	2601	2.22	21.01
	3	2516	2400	4.61	0.10	2464	2.07	28.57
	4	2556	2441	4.50	0.10	2500	2.19	25.68
	5	2625	2502	4.69	0.10	2604	0.80	44.73
	6	2615	2500	4.40	0.10	2564	1.95	39.59
	7	2627	2470	5.98	0.10	2619	0.30	23.17
	8	2556	2471	3.33	0.10	2530	1.02	25.69
700	1	3589	3448	3.93	0.19	3510	2.20	44.48
	2	3679	3449	6.25	0.19	3621	1.58	95.05
	3	3664	3512	4.15	0.19	3582	2.24	55.56
	4	3647	3457	5.21	0.19	3565	2.25	49.21
	5	3596	3447	4.14	0.19	3556	1.11	148.90
	6	3542	3319	6.30	0.19	3431	3.13	72.49
	7	3619	3389	6.36	0.01	3563	1.55	57.11
	8	3652	3462	5.20	0.19	3604	1.31	118.23

Tabela 1: Resultados para as instâncias com 500 e 700 itens.

	CarouselForfeits		**ILSForfeits**	
Nº de itens	Gap médio (%)	Tempo médio (s)	Gap médio (%)	Tempo médio (s)
500	4.48	0.09	1.80	29.19
700	5.19	0.17	1.92	80.13
média	4.83	0.13	1.86	54.66

Tabela 2: Média do *gap* e do tempo de execução para instâncias com 500 e 700 itens.

A Tabela 3 contém os resultados das instâncias para as quais o CPLEX não encontrou a solução ótima, sendo estas as instâncias grandes, com 800 e 1000 itens. Na Tabela 4 são apresentados os valores médios do *gap* em relação ao custo da melhor solução encontrada e o tempo médio de execução das 8 instâncias de cada conjunto. Para as instâncias grandes com 800 e 1000 itens, podemos observar através das Tabelas 3 e 4 que o *gap* médio da heurística ILSForfeits foi de 1.54%, enquanto a heurística *CarouselForfeits* obteve um *gap* médio de 4.82%. Para as instâncias grandes, o maior *gap* obtido pela heurística ILSForfeits foi de 2.88%, valor 40% inferior ao *gap* médio da heurística *CarouselForfeits*. Contudo, com relação ao tempo de execução, podemos observar novamente que o tempo médio da heurística ILSForfeits foi superior a heurística *CarouselForfeits*, com um tempo médio de 130.29 e 0.29 segundos, respectivamente.

Instância		**B&B**	**CarouselForfeits**			**ILSForfeits**		
Nº de itens	Id	Solução	Solução	Gap (%)	Tempo (s)	Solução	Gap (%)	Tempo (s)
800	1	4184	4024	3.82	0.26	4123	1.46	113.69
	2	4065	3827	5.85	0.25	3981	2.07	88.77
	3	4104	3886	5.31	0.27	3986	2.88	98.30
	4	4056	3850	5.08	0.01	3981	1.85	86.79
	5	4086	3894	4.70	0.02	4056	0.73	59.84
	6	4249	4084	3.88	0.27	4145	2.45	58.36
	7	4121	3895	5.48	0.02	4055	1.60	61.83
	8	4063	3859	5.02	0.25	4032	0.76	77.48
1000	1	4940	4655	5.77	0.38	4864	1.54	266.59
	2	4969	4756	4.29	0.39	4893	1.53	117.52
	3	5177	4897	5.41	0.44	5080	1.87	114.41
	4	5143	4916	4.41	0.47	5061	1.59	244.69
	5	5136	4935	3.91	0.49	5080	1.09	197.92
	6	5078	4858	4.33	0.06	5034	0.87	215.63
	7	5119	4876	4.75	0.56	5075	0.86	187.60
	8	5183	4916	5.15	0.51	5102	1.56	87.28

Tabela 3: Resultados para as instâncias com 800 e 1000 itens.

	CarouselForfeits		**ILSForfeits**	
Nº de itens	Gap médio (%)	Tempo médio (s)	Gap médio (%)	Tempo médio (s)
800	4.89	0.17	1.72	80.63
1000	4.75	0.41	1.36	179.95
média	4.82	0.29	1.54	130.29

Tabela 4: Média do *gap* e do tempo de execução para instâncias com 800 e 1000 itens.

Pode-se observar pelos resultados obtidos que a heurística ILSForfeits foi capaz de encontrar melhores soluções que a heurística *CarouselForfeits* em todas as instâncias avaliadas. Como o *gap* obtido entre os dois grupos (pequenas e grandes instâncias) são similares, há a hipótese de que o custo da solução obtida pelo CPLEX seja próximo ao custo da solução ótima para as instâncias grandes. Nesse sentido, existe um indicativo de que também seja possível obter as soluções ótimas das instâncias grandes com um incremento no tempo limite de execução do CPLEX para esse grupo de instâncias.

6. Conclusão

Neste trabalho, estudamos uma variante do Problema da Mochila, denominada de o Problema da Mochila com Penalidades. Nesta variante, um conjunto de pares de itens, denominado pares de penalidades, é adicionado ao problema, de modo que a inclusão de ambos os itens de um par na solução envolve o pagamento de uma penalidade ao custo da solução. Uma heurística ILS é proposta para o problema, tendo esta apresentado resultados superiores a melhor heurística da literatura para o problema, obtendo um *gap* de 1.54% com relação a melhor solução obtida pelo algoritmo de $B\&B$ do CPLEX para as instâncias grandes, e de 1.86% para as instâncias pequenas.

Trabalhos futuros podem investigar o comportamento das heurísticas propostas para instâncias mais difíceis, bem como propor novos métodos exatos não baseados em Programação Linear In-

teira para o problema. Alternativamente, outros métodos heurísticos como algoritmos genéticos e heurísticas baseadas em programação por restrições podem ser desenvolvidos.

Referências

Bettinelli, A., Cacchiani, V., e Malaguti, E. (2017). A branch-and-bound algorithm for the knapsack problem with conflict graph. *INFORMS J. on Computing*, 29(3):457–473. ISSN 1526-5528.

Cerrone, C., Cerulli, R., e Golden, B. (2017). Carousel greedy: A generalized greedy algorithm with applications in optimization. *Comput. Oper. Res.*, 85:97–112.

Cerulli, R., D'Ambrosio, C., Raiconi, A., e Vitale, G. (2020). The knapsack problem with forfeits. In Baïou, M., Gendron, B., Günlük, O., e Mahjoub, A. R., editors, *Combinatorial Optimization*, p. 263–272, Cham. Springer International Publishing.

Coniglio, S., Furini, F., e San Segundo, P. (2020). A new combinatorial branch-and-bound algorithm for the knapsack problem with conflicts. *European Journal of Operational Research*, 289.

Epstein, L. e Levin, A. (2008). On bin packing with conflicts. *SIAM Journal on Optimization*, 19: 1270–1298.

Gurski, F. e Rehs, C. (2019). The knapsack problem with conflict graphs and forcing graphs of bounded clique-width. In Fortz, B. e Labbé, M., editors, *Operations Research Proceedings 2018*, p. 259–265, Cham. Springer International Publishing. ISBN 978-3-030-18500-8.

Hifi, M. e Otmani, N. (2012). An algorithm for the disjunctively constrained knapsack problem. *International Journal of Operational Research*, 13:22–43.

Karp, R. M. (1972). Reducibility among combinatorial problems. In Miller, R. E. e Thatcher, J. W., editors, *Proceedings of a symposium on the Complexity of Computer Computations*, The IBM Research Symposia Series, p. 85–103.

Lourenço, H. R., Martin, O. C., e Stützle, T. (2003). Iterated local search. In Glover, F. e Kochenberger, G. A., editors, *Handbook of Metaheuristics*, p. 320–353. Springer US, Boston, MA. ISBN 978-0-306-48056-0. URL https://doi.org/10.1007/0-306-48056-5_11.

Pferschy, U. e Schauer, J. (2009). The knapsack problem with conflict graphs. *J. Graph Algorithms Appl.*, 13:233–249.

Pferschy, U. e Schauer, J. (2011). The maximum flow problem with conflict and forcing conditions. In Pahl, J., Reiners, T., e Voß, S., editors, *Network Optimization*, p. 289–294, Berlin, Heidelberg. Springer Berlin Heidelberg. ISBN 978-3-642-21527-8.

Pferschy, U. e Schauer, J. (2017). Approximation of knapsack problems with conflict and forcing graphs. *Journal of Combinatorial Optimization*, 33.

Sadykov, R. e Vanderbeck, F. (2013). Bin packing with conflicts: A generic branch-and-price algorithm. *INFORMS Journal on Computing*, 25(2):244–255.

Yamada, T., Kataoka, S., e Watanabe, K. (2002). Heuristic and exact algorithms for the disjunctively constrained knapsack problem.

Şuvak, Z., Altinel, K., e Aras, N. (2020). Exact solution algorithms for the maximum flow problem with additional conflict constraints. *European Journal of Operational Research*, 287.