UM ALGORITMO HEURÍSTICO APLICADO AO PROBLEMA DE CORTE UNIDIMENSIONAL

Flávio Barreiro Landes, Gustavo Silva Semaan
Instituto do Noroeste Fluminense de Educação Superior, Universidade Federal Fluminense
Avenida João Jasbick, s/n – Santo Antônio de Pádua, Rio de Janeiro, 28470-000, Brasil
[flavio_landes, gustavosemaan]@id.uff.br

Puca Huachi Vaz Penna
Departamento de Ciência da Computação, Universidade Federal de Ouro Preto
Campus Morro do Cruzeiro, Ouro Preto, Minas Gerais, 35400-000, Brasil
puca@iceb.ufop.br

RESUMO

O presente estudo aborda o Problema de Corte Unidimensional (PCU) com o objetivo de minimizar a quantidade de objetos e *setups*. Para tanto, um algoritmo heurístico híbrido baseado nas meta-heurísticas GRASP e ILS é proposto. Adicionalmente, dois novos métodos são apresentados, em que um atua na construção de padrões de corte BP (do inglês *Build Patterns*) enquanto o outro é considerado para refinar soluções removendo o excesso de objetos e *setup*, o RE (do inglês *Remove Excess*). Experimentos computacionais foram realizados com 1800 problemas-teste da literatura e os resultados encontrados foram comparados com os resultados de outros métodos existentes. O método proposto alcançou bons resultados, melhorando os custos das soluções de algumas classes de instâncias, e apresenta-se como uma alternativa interessante para tratar o PCU.

PALAVRAS CHAVE. Problema do Corte Unidimensional, Otimização Combinatória, Meta-heurísticas.

Área Principal: OC – Otimização Combinatória; MH – Meta-heurísticas; IND – PO na Indústria

ABSTRACT

This study addresses the One-dimensional Cutting Stock Problem (1D-CSP) with the objective of minimizing the number of objects and setups. Thus, a hybrid heuristic algorithm, based on GRASP and ILS meta-heuristics, is proposed. In addition, two new methods are presented. One used for cutting patterns construction, named Build Patterns (BP). The second, named Remove Excess (RE), applied to refine a solution by removing objects and setup excess. Computational experiments were done in 1800 test-problems from the literature. The proposed method achieved good results, improving the solution costs of some classes of instances, being an interesting alternative to deal with the 1D-CSP, when compared with existing methods in the literature.

KEYWORDS. One-dimensional Cutting Problem, Combinatorial Optimization, Metaheuristics.

Main Area: OC – Combinatorial optimization; MH – Metaheuristics; IND – PO in Industry

1. Introdução

O Problema do Corte Unidimensional (PCU) consiste em selecionar padrões de corte de acordo com uma peça base, de modo que atendam à demanda de um conjunto de itens e, analogamente, minimize o custo ou maximize o benefício. Em problemas de minimização, podem ser encontrados: minimização da quantidade de objetos (peças bases utilizadas); da quantidade de *setup* (padrões de cortes diferentes que foram utilizados); da quantidade da sobra total (somatório do desperdício de cada peça base); e tanto da quantidade de objetos quanto da quantidade de *setup*. A peça base pode ser composta de diferentes materiais ou formatos como, por exemplo, tora de madeira, tubo de papel, barra de aço, placa de vidro entre outros. Assim, o PCU é encontrado nas mais diversas indústrias, o que demonstra a sua importância nos estudos de otimização computacional.

Na literatura é possível encontrar trabalhos que consideram apenas um critério na função objetivo como, por exemplo, minimizar a quantidade de objetos ou a sobra total. Entretanto, também têm sido foco de estudo uso de dois critérios, sejam eles: a minimização da quantidade de objetos e da quantidade de *setups* (Golfeto *et al.* , 2007; Brandão *et al.* , 2009; Kobersztajn *et al.* , 2013). Segundo Brandão *et al.* (2009), os custos das trocas de padrões são relevantes quando a demanda precisa ser atendida em curto prazo, pois exigem que sejam feitos ajustes nas máquinas, o que resulta em tempo e trabalho (geram custos).

No presente estudo é abordado o PCU que considera os dois critérios na função objetivo, que correspondem a minimizar: (i) a quantidade de objetos; (ii) a quantidade de *setups*. Para tanto, um algoritmo heurístico híbrido é proposto. Este algoritmo utiliza a meta-heurística *Greedy Randomized Adaptive Search Procedure* – GRASP (Feo and Resende , 1995) para gerar uma solução inicial de boa qualidade e a meta-heurística *Iterated Local Search* – ILS (Lourenço *et al.* , 2010) como busca local.

Como busca local da heurística ILS é utilizado uma variação do método da descida randômica (*Random Descent Method*), denominado Método Randômico Não Ascendente – MRNA (Souza , 2011). Dois novos procedimentos são propostos, um para a geração de padrões de corte utilizado para a criação da solução e outro como um procedimento de refinamento dentro do ILS para excluir excessos de padrões ou objetos.

O trabalho está organizado da seguinte maneira: A Sessão 2 apresenta o Problema do Corte Unidimensional. A Sessão 3 apresenta uma revisão da literatura. Já a Sessão 4 descreve a heurística proposta bem como seus componentes. Em seguida, na Sessão 5 são descritos os experimentos e resultados computacionais alcançados. Por fim, a Sessão 6 apresenta as conclusões e propostas para trabalhos futuros.

2. Descrição do Problema

O Problema do Corte Unidimensional consiste em cortar peças base de tamanho W em diferentes tamanhos. Com base em uma quantidade m de itens diferentes, em que cada item i possui tamanho $w_i < W,\ i = 1, \ldots, m$, deve-se atender as demandas $d_i,\ i = 1, \ldots, m$ minimizando os custos envolvidos.

O problema é divido em duas tarefas, em que a primeira consiste na criação de diferentes padrões de corte *setups* e a segunda trata da seleção das quantidades de cada padrão de corte (denominados objetos). Nesse sentido, o modelo matemático descrito em Golfeto *et al.* (2007) é apresentado pelas Equações (1) – (3).

$$min\ \ c_1 \sum_{j=1}^{n} x_j + c_2 \sum_{j=1}^{n} \delta(x_j) \tag{1}$$

Sujeito a

$$\sum_{j=1}^{n} a_{ij}x_j \geq d_i, \quad i = 1, \ldots, m \tag{2}$$

$$x_j \in \mathbb{N}, \quad j = 1, \ldots, n \tag{3}$$

onde, $\delta(x_j)$ indica se o objeto j é selecionado:

$$\delta(x_j) = \begin{cases} 0, & \text{se} \quad x_j = 0, \\ 1, & \text{se} \quad x_j > 0. \end{cases}$$

Dados de entrada:

- c_1 = custo de cada objeto;
- c_2 = custo de cada setup;
- a_{ij} = quantidade do item i contida no padrão j;
- d_i = demanda do item i;
- m = quantidade de itens diferentes; e
- n = quantidade de padrões diferentes.

Variável de decisão:

- x_j = quantidade do objeto j;

3. Revisão da Literatura

A literatura apresenta diversos trabalhos em relação ao PCU. Segundo Golfeto *et al.* (2007), o primeiro estudo realizado sobre o problema foi feito por Kantorovich (1960) na década de 1930 e publicado apenas 30 anos depois. Desde então, vários estudos foram realizados porém, destaca-se que foram consideradas diferentes funções de avaliação.

Em alguns estudos o objetivo é minimizar a quantidade de objetos cortados (Filho and Moretti , 2015; Neto *et al.* , 2013). Outros consideram a minimização envolvendo os custos: (i) da quantidade de objetos; (ii) da quantidade de *setups* (Golfeto *et al.* , 2007; Brandão *et al.* , 2009; Kobersztajn *et al.* , 2013). Destaca-se que , especificamente sobre essa variante, conforme Brandão *et al.* (2009), o primeiro autor a tratar essa variante do problema foi Haessler (1975).

Além das variantes relatadas, existem abordagens que que consideram também a minimização da quantidade de ciclos de serra (Salles Neto *et al.* , 2014).

Com base na proposta de função de avaliação adotada no presente trabalho, alguns métodos existentes na literatura com o mesmo objetivo são destacados.

- Haessler (1975) desenvolveu o método *SHP*, que utiliza um meio exaustivo em que, iterativamente, calcula alguns parâmetros de aspiração, e é feita uma busca por padrões de corte para atenderem os tais parâmetros, até que satisfaça a demanda.

- Foerster and Wascher (2000) apresentou um método denominado *Kombi234* que tem por objetivo combinar padrões visando a diminuição da quantidade de *setups* e que, ainda assim, se mantenha a quantidade original de objetos.

- Salles Neto and Moretti (2005) aplicaram o algoritmo *ANLCP300* ao problema. Nele é utilizado o método lagrangiano aumentado em um problema com uma função objetivo, proposta por Haessler (1975), suavizada. Além disso, adaptou o método de geração de colunas de Gilmore and Gomory (1961), e para obter uma solução viável, utilizou uma heurística para arredondamento.

- Golfeto *et al*. (2007) propuseram o método *Symbio*, em que os autores utilizam um processo simbiótico entre a população de soluções e a população de padrões para gerar padrões em conjunto com soluções. Três versões do *Symbio* foram apresentadas para tratar diferentes pesos para os custos.

- Brandão *et al*. (2009) apresentou o *SingleGA10*, o qual utiliza uma abordagem baseada no trabalho de Golfeto *et al*. (2007) através de um Algoritmo Genético que considera a mesma função de avaliação.

4. Heurísticas Desenvolvidas

Com base nas especificações do PCU, inicialmente, é necessário criar padrões de corte. Em seguida, deve-se obter as melhores combinações *quantidade por padrão de corte*, com o objetivo de atender a demanda. Nesse sentido, foi desenvolvido o procedimento `BP` (*Build Patterns*) para criar padrões de corte de maneira aleatória, de modo que seja possível atender à demanda. Assim, para cada item da demanda existe ao menos um padrão de corte que o atenda. Em seguida, com o objetivo de selecionar a melhor combinação da quantidade para cada padrão de corte, foi adotada uma abordagem híbrida que considera as meta-heurísticas GRASP e ILS, Além disso, como busca local da ILS foi utilizado o *Método Randômico Não Ascendente* (MRNA). O procedimento `RE` (*Remove Excess*) foi proposto com o objetivo de remover os excessos de objetos e *setups* de uma dada solução.

O pseudocódigo do algoritmo proposto, nomeado `FiveA`[1], é apresentado no Algoritmo 1.

Algoritmo 1: FiveA($maxIterGRASP$, α, $maxIterILS$, $maxIterMRNA$, $maxPInv$)

Dados: $qtdePadroesIni$, $itens$, $tamanhoBase$

1 buildPatterns($qtdePadroesIni$, $itens$, $tamanhoBase$, $maxPInv$)
2 $s \leftarrow$ GRASP($maxIterGRASP$, α, $maxIterILS$, $maxIterMRNA$)
3 **retorna** s

4.1. Procedimento Build Patterns

Com base no Algoritmo 2 (*Build Patterns*) após a inicialização das variáveis, cada padrão é construído selecionando-se um item aleatório válido (que seja possível colocá-lo no padrão). Nesse caso a variável $sobra$ deve ser positiva (linhas 7-15). Em seguida, após a criação de padrões, é realizado um teste para verificar se existe ao menos um item de cada tipo no conjunto de padrões construídos. Se a quantidade de conjunto com padrões inválidos for maior do que $maxPadroesInvalidos$, então a quantidade inicial de padrões é incrementada e o processo de criação de novos padrões é reiniciado (linhas 17-25).

4.2. Procedimento GRASP

A meta-heurística GRASP (Feo and Resende , 1995) funciona através de uma abordagem iterativa na qual gera uma solução utilizando um método guloso e aleatório. Em seguida ela tenta refinar essa solução utilizando uma buscas locais.

O procedimento GRASP foi implementado conforme descrito na literatura e usa uma busca local baseada na meta-heurística ILS. A fase de construção da solução inicial é feita como segue.

[1] `FiveA` significa *Five Algorithms* fazendo referência aos cinco algoritmos usados (BP, GRASP, ILS, MRNA e RE)

Algoritmo 2: Procedimento buildPatterns($qtdePadroesIni$, $itens$, $tamanhoBase$, $maxPInv$)

```
1  padroesInvalidos ← 0
2  padroesValidos ← false
3  qtdePadroes ← qtdePadroesIniciais
4  repita // cria padroes até eles serem validos
5      padroes ← ∅
6      para indicePadrao ← 1 até qtdePadroes faça
7          repita
8              selecione aleatoriamente um item
9              padroes ← padroes ∪ indicePadrao[item]
10             sobra ← calcularSobra(itens, padroes, tamanhoBase)
11             se (sobra < 0) então // desfaz, retirando o último inserido
12                 padroes ← padroes \ indicePadrao[item]    // remove o padrão
13                 sobra ← tamanhoMenorItem
14             fim
15         até (sobra < tamanhoMenorItem)
16     fim
17     padroesValidos ← padroesValidos(padroes, qtdePadroes, itens)
18     se (não encontrou padroesValidos) então
19         padroesInvalidos ← padroesInvalidos + 1
20         se (padroesInvalidos > maxPInv) então
21             qtdePadroes ← qtdePadroes + 1
22             padroes ← ∅
23             padroesInvalidos ← 0
24         fim
25     fim
26 até encontrar padroesValidos
27 retorna padroes
```

Dentro do processo de iteração, até que uma solução atenda a demanda, são valorados e ordenados os elementos candidatos (ainda conforme a a demanda). Tal demanda é atualizada durante a fase de construção, e o valor de cada elemento é calculado através do $\sum_{j=1}^{m} p_{ij}d_j, i = 1, \ldots, n$ em que $m =$ quantidade de itens diferentes, $n =$ quantidade de padrões diferentes, $p_{ij} =$ quantidade do item j no padrão i e $d_j =$ demanda atual do item j. Logo após, é criada uma Lista Restrita de Candidatos LRC, em que somente são inseridos nessa lista LRC os elementos candidatos que satisfaçam o critério $e \geq min + \alpha(max - min)$ em que e é o valor do elemento. Por fim, α é a variável que controla o fator guloso para gerar solução, e é escolhido de maneira aleatória nesse estudo. Assim, um elemento contido na LRC e adicionado na solução e o processo é repetido até que toda a demanda seja atendida.

4.2.1. Iterated Local Search

A meta-heurística *Iterated Local Search* (ILS), segundo Temponi and Santos (2007), é baseada na simples ideia de que um procedimento de busca local pode ser melhorado gerando-se novas soluções de partida, as quais são obtidas por meio de perturbações na solução ótima local.

É importante destacar que, a perturbação tem que ser forte o suficiente para que a busca local consiga encontrar soluções diferentes, porém fraca o suficiente para que evite um reinicio aleatório.

Assim como o procedimento GRASP, o `ILS` foi implementado como descrito na literatura (Lourenço *et al.* , 2010) e usa o procedimento `MRNA` como busca local.

4.2.2. MRNA

O Método Randômico Não Ascendente – MRNA é uma variação do método da descida randômica (*Random Descent Method* – RDM). Ele escolhe, aleatoriamente, um vizinho da solução e o aceita se o seu valor for melhor ou igual ao valor da solução atual.

A ideia desse método parte do princípio que é possível d̈esbravarl̈ocais planos e, deste modo, sair de um ótimo local. Seu pseudocódigo é descrito no Algoritmo 3. No algoritmo (linhas 4 – 23), uma solução vizinha , s', é selecionada em $\mathcal{N}(s)$. Caso s' atenda demanda a solução é refinada pelo procedimento `RE` (Seção 4.2.3), obtendo-se a solução s''. Esta solução passa ser a solução corrente, se seu valor for melhor ou igual ao valor da solução atual (linhas 9 – 21). O método finaliza após $maxIterMRNA$ iterações sem melhora.

O `MRNA` utiliza apenas uma vizinhança $\mathcal{N}$ e consiste na ideia de tentar diminuir a quantidade de *setups* mantendo a quantidade de objetos. Para tanto, dois padrões p_1 e p_2 ($p_1 \neq p_2$) são selecionados, aleatoriamente. Em seguida, se a quantidade de p_1 for positiva, então a quantidade p_2 é incrementada da quantidade p_1 e a quantidade de p_1 é zerada. Caso contrário, repete os passos anteriores. Este procedimento pode retornar soluções inviáveis, ou seja, uma solução que não atende a demanda. Porém, ela é descartada pela busca local `MRNA` (linha 7).

Algoritmo 3: Procedimento MRNA(s, $maxIterMRNA$)

```
1  iterMRNA ← 0
2  contadorSolucaoIgual ← 0
3  s* ← RE(s)
4  enquanto (iterMRNA < maxIterMRNA) faça
5      iterMRNA ← iterMRNA + 1
6      Selecione s' ∈ N(s)
7      se (s' atende a demanda) então
8          s'' ←RE(s')
9          se (f(s'') ≤ f(s*)) então
10             se (f(s'') = f(s*)) então
11                 contadorSolucaoIgual ← contadorSolucaoigual + 1
12                 se (contadorSolucaoigual > 100) então
13                     Interrompe o laço Enquanto
14                 fim
15             senão
16                 contadorSolucaoIgual ← 0
17             fim
18             s ← s'
19             s* ← s''
20             iterMRNA ← 0
21         fim
22     fim
23 fim
24 retorna s*
```

4.2.3. Remove Excess (RE)

Com o objetivo de remover o excesso de objetos e *setups* de uma solução é proposto o algoritmo RE (*Remove Excess*). Ele parte do princípio que uma solução pode conter excesso de objetos ou de objetos e *setups*, em especial, em solucões iniciais. Experimentos preliminares indicaram que o uso do RE efetivamente reduziu a quantidade de objetos ou a quantidade de objetos e *setups* de soluções.

Algoritmo 4: Procedimento RE(s)

```
Ordenar solução de acordo com a função objetivo ou algum outro critério
p ← 0
qtdePadroes ←Quantidade total de padrões diferentes
enquanto p < qtdePadroes faça
    se (quantidade(p) em s > 0) então
        se ((quantidade(p) − 1 atende a demanda) então
            Decrementa 1 na quantidade do padrão p
            Atualiza s com a quantidade do padrão p
        senão
            p ← p + 1
        fim
    senão
        p ← p + 1
    fim
fim
retorna s
```

5. Resultados Computacionais

Com o objetivo de avaliar e a eficácia e eficiência do algoritmo proposto foi considerado o conjunto de problemas-teste apresentado por Golfeto *et al*. (2007). O conjunto possui 1800 instâncias organizadas em 18 classes, conforme suas características.

O algoritmo foi desenvolvido em C++ (compilador g++ 5.3.0) em um computador Celeron quad core N2940, 2.25 GHz, 4GB de RAM e sistema operacional Windows 8.1 de 64 bits.

Ele utilizou o custo de objeto $c_1 = 1$ e os custos de *setup* $c_2 = \{1, 5, 10\}$. Portanto, de acordo com os custos de *setup* foram feitas três versões do FiveA, nomeados de FiveA01, FiveA05 e FiveA10 representando, respectivamente, os diferentes valores de c_2.

As três versões foram executadas com os mesmos parâmetros, obtidos em experimentos preliminares. Para o `BuildPatterns` a quantidade de padrões iniciais, $qtdePadroesIni = 8 \times itens$, onde $itens$ é a quantidade de itens diferentes e o limite máximo de conjuntos de padrões inválidos $maxPInv = 30000$. Para o `GRASP` foi adotado $maxIterGRASP = 5$ e o valor de α selecionado aleatoriamente, no intervalo $[0, 1]$, em cada iteração da fase de construção de uma solução. O `ILS` adotou $maxIterILS = 25$, sendo o número de iterações do `MRNA`, $maxIterMRNA = 25$.

Por fim, os resultados alcançados são apresentados nas Tabelas 1 – 5 e comparados com os resultados dos algoritmos SHP (Haessler , 1975), Kombi234 (Foerster and Wascher , 2000), ANLCP300 (Salles Neto and Moretti , 2005), Symbio01, Symbio05 e Symbio10 (Golfeto *et al.* , 2007) e SingleGA10 (Brandão *et al.* , 2009).

A Tabela 1 apresenta os valores das soluções obtidas pelas 3 versões do algoritmo FiveA. Onde a primeira coluna representa a classe de problemas e cada par de colunas do FiveA, **Set.** e **Obj.** são, respectivamente, a média dos números de *setup* e a média dos números de objetos, para cada versão.

Em seguida, nas Tabelas 2 – 4, são apresentados os resultados com o custo total de cada algoritmo desenvolvidos comparados aos da literatura. Sendo, o custo total calculado pela Equação (4).

$$CustoTotal = c_1 \times MdObj + c_2 \times MdSet \tag{4}$$

Onde

c_1 = custo do objeto; c_2 = custo do *setup*; $MdObj$ = Média da quantidade de objetos dos resultados de todas classes. $MdSet$ = Média da quantidade de *setups* dos resultados de todas

Tabela 1: Resultados dos FiveA01, FiveA05 e FiveA10

	FiveA01		FiveA05		FiveA10	
Classe	Set.	Obj.	Set.	Obj.	Set.	Obj.
1	3,05	12,94	2,01	15,07	1,92	15,54
2	7,10	118,02	4,67	122,04	3,94	125,58
3	5,98	25,75	4,18	29,39	3,89	30,76
4	15,88	233,72	11,27	240,72	10,14	243,05
5	12,65	51,21	9,15	58,29	8,60	61,37
6	34,16	467,15	25,34	476,54	24,12	481,10
7	6,12	50,99	5,27	52,71	5,18	53,43
8	7,98	505,24	7,42	506,36	7,15	507,64
9	10,65	96,29	9,32	99,30	9,18	100,25
10	19,64	954,34	17,12	966,30	16,55	963,52
11	24,93	186,32	22,02	191,79	21,54	194,25
12	51,39	1852,92	46,97	1868,17	44,63	1879,21
13	7,29	64,19	6,64	65,27	6,52	66,17
14	9,23	637,48	8,74	641,24	8,64	639,41
15	14,45	122,00	12,98	124,60	12,82	125,08
16	23,97	1210,00	20,87	1212,33	20,39	1218,46
17	29,13	233,10	26,40	238,46	26,19	239,52
18	74,21	2337,27	63,11	2341,94	65,99	2342,74

classes.

Já a variação percentual do custo total da solução dos algoritmos comparado ao melhor custo de solução encontrado na literatura (*Best Known Solution* – BKS), denominado *gap*, é calculado pela Equação (5).

$$gap = \frac{CustoTotal - MelhorCustoTotal}{MelhorCustoTotal} \times 100 \quad (5)$$

Em que: $CustoTotal =$ Custo total alcançado pelo método atual.
$MelhorCustoTotal =$ Melhor custo total encontrado na literatura.
Os valores positivos do *gap* indicam um resultado pior que o valor do BKS.

Tabela 2: Custo total de cada método levando em conta o custo do objeto = 1 e *setup* = 1

Classe	**BKS**	SHP		Kombi234		ALCP300		Symbio01		**FiveA01**	
		Sol.	gap	Sol.	gap	Sol.	gap	Sol.	gap	Sol.	gap
1	**14,89**	18,12	21,69	**14,89**	0,00	17,32	16,32	15,68	5,31	15,99	7,39
2	**118,06**	122,41	3,68	**118,06**	0,00	125,98	6,71	121,28	2,73	125,12	5,98
3	**28,02**	30,29	8,10	**28,02**	0,00	29,98	7,00	31,70	13,13	31,73	13,24
4	**230,19**	232,64	1,06	**230,19**	0,00	232,14	0,85	246,52	7,09	249,60	8,43
5	**52,68**	53,76	2,05	53,71	1,96	**52,68**	0,00	67,06	27,30	63,86	21,22
6	**443,64**	444,40	0,17	450,15	1,47	443,64	0,00	503,02	13,38	501,31	13,00
7	**57,08**	64,68	13,31	58,11	1,80	**57,08**	0,00	57,94	1,51	57,11	0,05
8	**503,94**	525,52	4,28	509,48	1,10	**503,94**	0,00	519,01	2,99	513,22	1,84
9	**108,70**	125,73	15,67	108,70	0,00	125,10	15,09	110,30	1,47	106,94	**-1,62**
10	**951,60**	1020,96	7,29	**951,60**	0,00	983,16	3,32	987,27	3,75	973,98	2,35
11	**205,71**	235,00	14,24	**205,71**	0,00	251,84	22,42	221,30	7,58	211,25	2,69
12	**1803,51**	1910,30	5,92	**1803,51**	0,00	1884,88	4,51	1965,09	8,96	1904,31	5,59
13	**72,24**	79,35	9,84	72,24	0,00	76,18	5,45	72,51	0,37	71,48	**-1,05**
14	**641,94**	653,40	1,79	642,44	0,08	**641,94**	0,00	655,39	2,10	646,71	0,74
15	**136,31**	154,06	13,02	**136,31**	0,00	137,30	0,73	141,05	3,48	136,45	0,10
16	**1208,62**	1273,18	5,34	1211,71	0,26	**1208,62**	0,00	1271,37	5,19	1233,97	2,10
17	**256,14**	290,40	13,38	**256,14**	0,00	320,18	25,00	271,71	6,08	262,23	2,38
18	**2380,68**	2419,77	1,64	**2380,68**	0,00	2457,48	3,23	2446,77	2,78	2411,48	1,29
Média gap			7,92		**0,37**		6,15		6,40		4,76
FiveA01 melhor (%)		72,22		16,67		50,00		77,78			

A Tabela 2 apresenta os resultados para o custo de *setup* igual a 1. nela, o algoritmo FiveA01

obteve um *gap* de apenas 4,76%, sendo, na média, melhor que todos os demais algoritmos, exceto em relação ao Kombi234. Individualmente, o FiveA01 foi melhor do que o Symbio01 em 77,78% das classes, para o SHP em 72,22%, e 50% para o ALCP300, mas em apenas 16,67% melhor do que o Kombi234. É importante ressaltar que em duas classes o FiveA01 conseguiu alcançar resultados melhores do que os BKSs. São elas, as classes 9 (-1,62%) e 13 (-1,05%).

Já a Tabela 3 mostra os resultados para o custo de *setup* igual a 5. Novamente, o FiveA05 obteve melhores soluções médias que 3 dos 4 algoritmos. Superando o resultado do SHP em 66,67% das classes, do Symbio05 em 61,11%, Kombi234 em 55,56% e ALCP300 em 33,33%. No entanto, o FiveA05 obteve em 5 classes resultados melhores que os BKS's, melhorando os resultados em até 3,44%.

Tabela 3: Custo total de cada método levando em conta o custo do objeto = 1 e setup = 5

Classe	**BKS**	SHP		Kombi234		ALCP300		Symbio01		**FiveA05**	
		Sol.	gap (%)	Sol.	gap (%)	Sol.	gap (%)	Sol.	gap (%)	Sol.	gap (%)
1	**24,29**	33,92	39,65	28,49	17,29	29,40	21,04	**24,29**	0,00	25,12	3,42
2	**142,66**	146,17	2,46	149,30	4,65	143,98	0,93	**142,66**	0,00	145,39	1,91
3	**49,34**	50,29	1,93	51,58	4,54	**49,34**	0,00	51,43	4,24	50,29	1,93
4	**261,26**	261,88	0,24	287,23	9,94	**261,26**	0,00	286,19	9,54	297,07	13,71
5	**80,76**	81,24	0,59	96,71	19,75	**80,76**	0,00	103,05	27,60	104,04	28,83
6	**487,32**	487,64	0,07	551,91	13,25	**487,32**	0,00	589,61	20,99	603,24	23,79
7	**79,24**	100,04	26,25	89,71	13,21	79,24	0,00	80,44	1,51	79,06	**-0,23**
8	**535,94**	564,56	5,34	549,32	2,50	**535,94**	0,00	553,72	3,32	543,46	1,40
9	**151,76**	194,49	28,16	168,82	11,24	167,42	10,32	151,76	0,00	145,90	**-3,86**
10	**1028,72**	1098,44	6,78	**1028,72**	0,00	1039,00	1,00	1056,96	2,75	1051,90	2,25
11	**312,63**	363,80	16,37	320,67	2,57	331,84	6,14	312,63	0,00	301,89	**-3,44**
12	**1952,75**	2059,30	5,46	**1952,75**	0,00	1979,60	1,37	2091,49	7,10	2103,02	7,70
13	**98,74**	116,87	18,36	108,12	9,50	98,74	0,00	100,01	1,29	98,47	**-0,27**
14	**673,62**	692,80	2,85	683,72	1,50	**673,62**	0,00	687,81	2,11	684,94	1,68
15	**176,98**	226,18	27,80	203,83	15,17	**176,98**	0,00	194,33	9,80	189,50	7,07
16	**1264,14**	1351,70	6,93	1291,35	2,15	**1264,14**	0,00	1336,64	5,74	1316,68	4,16
17	**377,77**	427,96	13,29	381,98	1,11	410,58	8,69	377,77	0,00	370,46	**-1,94**
18	**2533,80**	2572,69	1,53	**2533,80**	0,00	2567,24	1,32	2579,49	1,80	2657,49	4,88
Média gap			11,34		7,13		**2,82**		5,43		5,17
FiveA05 melhor (%)		66,67		55,56		33,33		61,11			

Por fim, a Tabela 4 compara os resultados levando em conta o custo do *setup* igual à 10. Pela tabela, é possível afirmar que o FiveA10 obteve resultados melhores do que a maioria dos outros métodos. Conseguindo, assim, ser melhor do que os métodos SHP em 72,22% das classes, Kombi234 em 72,22% das classes, ALCP300 em 38,89% das classes, Symbio10 em 55,56% das classes e, por fim, em 83,33% das classes o FiveA10 foi melhor do que o SingleGA10. Por fim, o FiveA10 em 5 classes foi melhor do que o melhor resultado encontrado (coluna BKS) e em 1 classe teve o gap com diferença < 1% para igualar com o melhor resultado encontrado (coluna BKS).

Os tempos computacionais dos algoritmos são apresentados na Tabela 5. Para facilitar a comparação, apenas foi adicionado o tempo de execução do FiveA05, pois a diferença de tempo entre o FiveA01, FiveA05 e FiveA10 é mínima. O tempo computacional gasto pelo SingleGA10 não consta na tabela, pois os autores não o disponibilizaram. Apesar do FiveA05 se mostrar bastante competitivo em relação ao desempenho computacional não é possível efetuar maiores comparações devido a grande diferença entre as máquinas utilizadas pelos autores dos algoritmos em questão.

6. Conclusões

Este trabalho abordou o Problema de Corte Unidimensional (PCU) levando em consideração a minimização da quantidade de objetos produzidos e da quantidade de *setups* necessários. Para isso, foi desenvolvido um algoritmo híbrido, nomeado FiveA, baseado na meta-heurística GRASP, que utiliza um ILS como busca local. Além disso, foram desenvolvidos dois novos procedimentos. Um para criar os de padrões de corte, usado como entrada para o GRASP, e outro para remover excesso de objetos e *setups* de uma solução, refinamento dentro do ILS.

Tabela 4: Custo total de cada método levando em conta o custo do objeto = 1 e setup = 10

Classe	**BKS**	SHP		Kombi234		ALCP300		Symbio10		SingleGA10		**FiveA10**	
		Sol.	gap	Sol.	gap	Sol.	gap	Sol.	gap	Sol.	gap	Sol.	gap
1	**33,34**	53,67	60,98	45,49	36,44	44,50	33,47	**33,34**	0,00	35,26	5,76	34,74	4,20
2	**164,60**	175,87	6,85	188,35	14,43	166,48	1,14	**164,60**	0,00	165,16	0,34	164,98	0,23
3	**72,93**	75,29	3,24	81,03	11,11	73,54	0,84	**72,93**	0,00	74,30	1,88	69,66	**-4,48**
4	**297,66**	298,43	0,26	358,53	20,45	**297,66**	0,00	333,32	11,98	340,73	14,47	344,45	15,72
5	**115,59**	**115,59**	0,00	150,46	30,17	115,86	0,23	143,72	24,34	149,04	28,94	147,37	27,49
6	**541,69**	**541,69**	0,00	679,11	25,37	541,92	0,04	659,65	21,78	747,18	37,93	722,30	33,34
7	**105,85**	144,24	36,27	129,21	22,07	106,94	1,03	105,85	0,00	107,73	1,78	105,23	**-0,59**
8	**575,94**	613,36	6,50	599,12	4,02	**575,94**	0,00	592,25	2,83	578,91	0,52	579,14	0,56
9	**198,53**	280,44	41,26	243,97	22,89	220,32	10,98	201,72	1,61	198,53	0,00	192,05	**-3,26**
10	**1108,80**	1195,29	7,80	1125,12	1,47	**1108,80**	0,00	1136,12	2,46	1165,38	5,10	1129,02	1,82
11	**417,04**	524,80	25,84	464,37	11,35	431,84	3,55	417,04	0,00	489,21	17,31	409,65	**-1,77**
12	**2098,00**	2245,55	7,03	2139,30	1,97	**2098,00**	0,00	2232,60	6,42	2808,36	33,86	2325,51	10,84
13	**126,94**	163,77	29,01	152,97	20,51	**126,94**	0,00	134,12	5,66	133,23	4,96	131,37	3,49
14	**713,22**	742,05	4,04	735,32	3,10	**713,22**	0,00	732,62	2,72	724,63	1,60	725,81	1,77
15	**226,58**	316,33	39,61	288,23	27,21	**226,58**	0,00	258,77	14,21	259,16	14,38	253,28	11,78
16	**1333,54**	1449,85	8,72	1390,90	4,30	**1333,54**	0,00	1415,30	6,13	1455,24	9,13	1422,36	6,66
17	**503,82**	599,91	19,07	539,28	7,04	523,58	3,92	503,82	0,00	528,60	4,92	501,42	**-0,48**
18	**2704,44**	2763,84	2,20	2725,20	0,77	**2704,44**	0,00	2744,21	1,47	3089,09	14,22	3002,64	11,03
Média gap			16,59		14,70		**3,07**		5,64		10,95		6,58
FiveA10 melhor (%)		72,22		72,22		38,89		55,56		83,33			

Tabela 5: Tempo de execução de cada método

Classe	SHP Tempo[a] (s)	Kombi234 Tempo[b] (s)	ALCP300 Tempo[c] (s)	Symbio05 Tempo[d] (s)	FiveA05 Tempo (s)
1	0,01	0,14	1,12	18,54	1,06
2	0,08	1,14	3,17	37,88	1,05
3	0,17	1,74	0,89	33,25	5,07
4	0,21	16,00	1,15	68,11	6,05
5	0,27	38,03	0,64	58,29	44,03
6	0,31	379,17	0,91	158,04	64,42
7	0,01	0,07	18,10	19,62	0,97
8	0,02	0,20	11,78	48,48	1,04
9	0,04	3,37	105,49	38,75	3,83
10	0,06	3,25	106,59	127,25	5,78
11	0,22	36,26	216,97	117,85	52,18
12	0,32	76,31	376,04	426,08	55,08
13	0,01	0,08	7,43	17,66	0,92
14	0,02	0,13	2,02	31,19	0,97
15	0,03	1,81	60,02	41,12	5,18
16	0,04	2,60	39,87	133,90	5,07
17	0,16	50,93	267,02	153,45	52,54
18	0,24	70,94	538,92	388,89	40,20
Média	0,12	37,90	97,67	106,58	19,19

[a]: AMD AthlonXP 1800 MHz com 512MB de RAM
[b]: IBM PC 486/66
[c]: AMD AthlonXP 1800 MHz com 512MB de RAM
[d]: AMD SEMPRON 2300+ 1533 MHz com 640MB de RAM

Para verificar a eficácia e a eficiência do FiveA, foram realizados testes em um conjunto de 1800 instâncias existentes na literatura, divididas em 18 classes. Devido aos diferentes custos de *setup* existentes nessas instâncias, 3 versões do FiveA foram desenvolvidas: FiveA01, FiveA05 e FiveA10. Todas as versões do algoritmo conseguiram obter novas melhores soluções. O FiveA01 melhorou os resultados da literatura para duas classes, e as versões FiveA05 e FiveA10 para cinco classes, cada. Sendo que o maior ganho, 4,48%, foi obtido pelo FiveA10 para a classe 3.

Apesar dos bons resultados obtidos pelo FiveA, superando diversas vezes os resultados dos algoritmos comparados, algumas classes obtiveram desvios bastante elevados. Pretende-se, portanto, analisar as características dessas classes com intuito de aperfeiçoar o FiveA.

Agradecimentos Os autores deste artigo explicitam agradecimentos a Rodrigo Rabello Golfeto, Antônio Carlos Moretti e Luiz Leduíno de Salles Neto pela, gentil, disponibilização de suas instâncias. Esta pesquisa foi apoiada pela Fundação de Apoio à Pesquisa do Estado de Minas Gerais – FAPEMIG.

Referências

Brandão, J. L., Coelho, A. M. and Vasconcellos, J. F. V. Aplicação de algoritmo genético para minimizar o número de objetos processados e setup num problema de corte unidimensional. SOBRAPO (Ed.), *XLI Simpósio Brasileiro de Pesquisa Operacional – SBPO*, p. 2539 – 2550, 2009.

Feo, T. and Resende, M. (1995), Greedy randomized adaptive search procedures. *Journal of Global Optimization*, v. 6, p. 109 – 133.

Filho, A. A. and Moretti, A. C. Um simulated annealing no problema do corte unidimensional inteiro. SOBRAPO (Ed.), *Proceeding Series of the Brazilian Society of Applied and Computational Mathematics*, volume 3, p. 10406–1 – 10406–8. Trabalho apresentado no XXXV CNMAC, Natal-RN, 2014, 2015.

Foerster, H. and Wascher, G. (2000), Pattern reduction in one-dimensional cutting-stock problems. *International Journal of Production Research*, v. 38, p. 1657 – 1676.

Gilmore, P. C. and Gomory, R. E. (1961), A linear programming approach to the cutting stock problem. *Operations Research*, v. 9, p. 849 – 859.

Golfeto, R. R., Moretti, A. C. and Salles Neto, L. L. Algoritmo genético simbiótico aplicado ao problema de corte unidimensional. SOBRAPO (Ed.), *XXXIX Simpósio Brasileiro de Pesquisa Operacional – SBPO*, p. 1415 – 1426, Fortaleza - CE, 2007.

Haessler, R. (1975), Controlling cutting pattern changes in one-dimensional trim problems. *Operations Research*, v. 23, p. 483 – 493.

Kantorovich, L. V. (1960), Mathematical methods of organizing and planning production. *Management Science*, v. 6, p. 366–422.

Kobersztajn, H. A., Poldi, K. C. and Araujo, S. A. Algoritmo evolutivo para o problema de corte de estoque unidimensional com redução do número de padrões de corte. *Congresso de Matemática Aplicada e Computacional*, p. 408 – 413, 2013.

Lourenço, H. R., Martin, O. C. and Stützle, T. Iterated local search: Framework and applications. Gendreau, M. and Potvin, J.-Y. (Eds.), *Handbook of Metaheuristics*, volume 146 of *International Series in Operations Research & Management Science*, p. 363–397. Springer US, 2010.

Neto, E. V., Velasco, A. S. and Junior, G. G. P. Algoritmos grasp híbridos para o problema de corte unidimensional. SOBRAPO (Ed.), *XLV Simpósio Brasileiro de Pesquisa Operacional – SBPO*, p. 1869 – 1880, Natal – RN, 2013.

Salles Neto, L. L., Araujo, S. and Golfeto, R. (2014), Um problema de corte de estoque multi-objetivo. *Revista Eletrônica Pesquisa Operacional para o Desenvolvimento*, v. 6, n. 2, p. 183 – 201.

Salles Neto, L. L. and Moretti, A. C. Modelo não-linear para minimizar o número de objetos processados e o setup num problema de corte unidimensional. SOBRAPO (Ed.), *XXXIX Simpósio Brasileiro de Pesquisa Operacional – SBPO*, p. 1679 – 1688, 2005.

Souza, M. J. F. Inteligência computacional para otimização. techreport, Universidade Federal de Ouro Preto, 2011.

Temponi, E. C. C. and Santos, F. A. Uma metaheurística híbrida grasp-ils aplicada à solução do problema de corte bi-dimensional guilhotinado. SOBRAPO (Ed.), *XXXIX Simpósio Brasileiro de Pesquisa Operacional – SBPO*, p. 1708 – 1717, Fortaleza - CE, 2007.