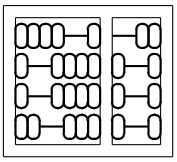


Universidade Estadual de Campinas - UNICAMP
Instituto de Computação - IC

Relatório Final - MO824

Metaheurísticas BRKGA para o Problema do Empacotamento Bidimensional Ortogonal

RAFAEL GRISOTTO E SOUZA

Campinas
2017

Abstract

Este projeto computacional descreve o Problema da Mochila Bidimensional (2KP), que é um problema NP-Difícil. Uma mochila bidimensional é usada para empacotar itens de mesma dimensão e estes itens tem valores agregados. No projeto foi utilizado um solver exato *Gurobi* e a metaheurísticas RKGA para resolver o 2KP. Com o BRKGA utilizamos 4 decodificadores diferentes e dois deles se mostraram bastante eficientes. Concluindo que a abordagem exata para este problema não se mostrou boa o suficiente e a metaheurística teve resultados bons para dois decodificadores dos quatro avaliados.

Keywords: problema empacotamento bidimensional, BRKGA, knapsack

Sumário

1 Introdução

Os problemas de corte e empacotamento são geralmente problemas onde se deseja que objetos pequenos sejam empacotados em outros maiores, podendo ser feito cortes nestes pequenos objetos para satisfazer restrições do empacotamento. Os problemas gerais e suas variedades tem sidos estudados por mais de seis décadas. Estes são considerados problemas de otimização combinatória e possuem em sua grande maioria variantes que são NP-Difíceis. Além do interesses teóricos destes problemas, existem demandas práticas. Podemos destacar aplicações como: escalonamento de processos, industrias de metais e placas, propaganda, telecomunicações e industrias de tecido.

Devido aos diferentes tipos de problemas presentes na área de corte e empacotamento, Dyckhoff's [7] caracterizou características comuns e propriedades para problemas de corte e empacotamento. As quatro categorias definidas por ele são:

Dimensionalidade: É o número de dimensões necessárias para descrever a geometria dos objetos.

Tipo de atribuição: Descreve se todos os itens ou apenas uma seleção deles devem ser utilizados.

Variedade dos recipientes: Pode ser apenas um ou vários recipientes e estes podem ser iguais ou de variadas formas.

Variedade dos itens: A forma como são e sua quantidade. Estes podem ser poucos itens ou muitos com diferentes formas e similaridades.

Neste trabalho, foi investigado O problema de empacotamento bidimensional ortogonal que pode ser chamado de **Problema da Mochila Bidimensional Ortogonal**, do inglês *Two-Dimensional Orthogonal Knapsack Problem* (2KP) quando temos apenas um recipiente para os itens serão empacotados. No problema temos retângulos com largura, comprimento e um valor agregado a eles, não é permitido a rotação dos retângulos e devem ser empacotados de forma ortogonal aos lados da mochila. Neste problema é fundamental que os retângulos não extrapolem as dimensões da mochila e nem ocupem o mesmo espaço dentro do recipiente. O objetivo é maximizar o valor empacotado na mochila.

O problema 2KP pode ser descrito assim: Dado um conjunto de retângulos $L = \{1, 2, \ldots, n\}$, para cada i com comprimento l_i, largura w_i e um valor v_i não negativo, onde $V : L \to \mathbb{R}^+$, e temos uma mochila bidimensional R com comprimento L e largura W, tal que $0 < l_i \leq L$ e

$0 < w_i \leq W$. Para cada item, existe a quantidade máxima de cópias permitida e temos como objetivo encontrar um subconjunto $S \leq L$ e empacotar S em R, tal que $\sum_{i \in L} v_i$ é máximo.

Uma grande quantidade de autores propuseram procedimentos para a resolução do 2KP. Como algoritmos exatos [5, 6], também heurísticas híbridas [11] e metaheuristicas como GRASP [1], busca Tabu [2] e BRKGA [8]. Para uma completa lista de abordagens, veja Lodi, Andrea et al. [12].

2 Metodologias

No projeto computacional, foram utilizadas a metaheurística BRKGA e o solver de programação linear inteira Gurobi. As instâncias que foram utilizadas são da literatura, sendo oito do Beasley [5] e um pacote contendo 630 instâncias geradas por Beasley [4], estas instâncias estão classificadas em três níveis de dificuldade, cada nível contendo 210 instâncias. Para efeito de comparação, utilizamos apenas oito do primeiro nível e 5 de cada um dos demais níveis. No total utilizamos 26 instâncias que abrangem vários tamanhos de mochilas e quantidade entre 40 e 2000 de itens.

2.1 Modelo de Programação de linear

Para a solução da programação linear, precisamos definir a região da mochila para empacotar os retângulos e garantir todas as restrições. Assim, utilizamos pontos de discretização proposto por Herz [10], que pode ser descrito da seguinte forma: Um conjunto P de todos os pontos de discretização do comprimento da mochila, onde para cada $i \in P$ é um valor $i \leq R$ que pode ser obtido por uma combinação cônica inteira de $l_1, \ldots, l_m$. De maneira similar é gerado um conjunto Q para a largura.

O algoritmo DDP 1 (Discretization using Dynamic Programming) é basicamente resolver o problema de knapsack em que cada item i tem peso e valor d_i ($i = 1, \ldots, n$), e a knapsack tem capacidade D. A partir disso, utilizamos uma técnica de programação dinâmica para o problema knapsack que encontra os valores ótimos para as knapsacks com capacidades entre 1 até D. Veja que é fácil observar que j é um ponto de discretização se e somente se a knapsack com capacidade j tem valor ótimo j.

Algorithm 1 DDP

Require: $D, d_1, \ldots, d_n$.

Ensure: Um conjunto de pontos de discretização P.

$P \leftarrow \emptyset$
for $j = 0$ até D **do**
    $c_j = 0$
for $i = 0$ até n **do**
    for $j = 0$ até D **do**
        if $c_j < c_{j-d_i} + d_i$ **then**
            $c_j = c_{j-d_i} + d_i$
for $j = 0$ até D **do**
    if $c_j = j$ **then**
        $P = P \bigcup j$

O problema 2KP pode ser formulado com o seguinte programa linear inteiro:

$$\text{Maximizar} \quad \sum_i \sum_p \sum_q v_i x_{i,p,q} \tag{1}$$

$$\text{Sujeito a} \quad \sum_i \sum_p \sum_q (w_i \cdot l_i) x_{i,p,q} \leq W \cdot L \tag{2}$$

$$\sum_{(p,q) \in wSet \times hSet} x_{i,p,q} \leq Q_i \qquad \forall i \in Items \tag{3}$$

$$\sum_{(i,p,q) \in E(s,t)} x_{i,p,q} \leq 1 \qquad \forall (s,t) \in wSet \times hSet \tag{4}$$

$$x_{i,p,q} \leq Q_i \qquad \text{para } i = 1, \ldots, n \tag{5}$$

$$x_{i,p,q} \geq 0, \qquad \text{inteiro, para } i = 1, \ldots, n \tag{6}$$

onde:

- n é a quantidade total de itens;

- v_i representa o valor do item i;
- Q_i representa a quantidade máxima de copias do item i;
- l_i e w_i representam o comprimento e a largura do item i, respectivamente;
- $x_{i,p,q}$ indica se o item i será empacotado na mochila ($x_{i,p,q} > 0 <= Q_i$) ou não ($x_{i,p,q} = 0$).

$$E(s,t) = \begin{cases} (i,p,q) : (p,q) \in wSet \times hSet \\ \qquad i \in Itens \\ \qquad p \leq s \leq p + w \\ \qquad q \leq t \leq q + h \end{cases}$$

2.2 BRKGA

O BRKGA (biased random-key genetic algorithm)[9] é uma metaheuristica baseada no algoritmo de chaves aleatórias de Bean [3] para problemas de otimização. A solução é representada por um vetor de chaves aleatórias em um intervalo de números reais continuo $(0, 1]$. O decodificador calcula o custo da solução mapeando um vetor de chaves aleatórias em um uma solução do problema. No inicio o algoritmo gera uma serie de populações iniciais de p vetores de n chaves aleatórias. A cada geração é copiado um conjunto dos melhores indivíduos, conjunto chamado elite, também é gerado um conjunto de mutantes e para completar a população desta nova geração são gerados aleatoriamente novos indivíduos.

Para conseguir novos resultados, adicionamos ao BRKGA duas variações: diversidade de população e reinício por não aprimoramento. Para manter os indivíduos diversos e evitar a convergência muito cedo, foi usada uma função que diversifica as populações após o processo seleção do conjunto elite na geração atual. Os indivíduos são então comparados 2 a 2 e se houver menos de uma porcentagem, que é um parâmetro, de locus diferentes entre eles, o primeiro indivíduo selecionado tem n locus mutados aleatoriamente, onde n é o tamanho da instância de entrada. No entanto, indivíduos que são uma porcentagem, definida por um parâmetro, do fitness da solução incumbente são mantidos para a próxima geração e não chegam a ser comparados. Para conseguir tentar evitar mínimos locais, foi aplicado a estratégia de reinício por não aprimoramento, onde a cada uma quantidade definida de gerações não houver melhoria na solução, então é gerado populações novas.

2.2.1 Conjunto de parâmetros

Para o BRKGA utilizado, temos os seguintes parâmetros:

- `n`: Indica o número de indivíduos existentes em cada geração do processo. Esses indivíduos serão avaliados de acordo com sua qualidade (*fitness*).
- `pe`: Define uma fração da população que será o conjunto elite.
- `pm`: Define uma fração da população que será trocada por mutantes para a próxima geração.
- `rhoe`: Define a probabilidade de que os descendentes herdem um alelo de um parente de elite.
- `K`: Indica quantas populações independentes serão utilizadas no algoritmo.
- `X_INTVL`: Este parâmetro define a quantidade de gerações para a troca dos melhores indivíduos entre as populações.
- `X_NUMBER`: Quando for feito a troca dos melhores indivíduos, este parâmetros define quantos serão trocados.
- `SizeRate`: Utilizando *Diversity Maintenance*, é necessário diversificar a população. Neste caso, realizam-se testes 2 a 2 em cada indivíduo e se os indivíduos possuírem um número maior que o valor pré-definido (0.5%, por exemplo) de alelos iguais, o primeiro dos dois sofre uma mutação aleatória em todo o locus.
- `FitRate`: Dada a solução incumbente, se indivíduos possuírem um *fitness* com porcentagem `FitRate` da solução incumbente, esses indivíduos serão mantidos para a próxima geração e não serão sequer comparados no processo de *SizeRate*.
- `reset`: Define a quantidade de gerações sem melhoria na solução para a aplicação da estratégia de reinício por não aprimoramento.

2.2.2 Decodificadores para o problema 2KP

Para o calcular o custo de solução, foram implementados quatro decodificadores que resolvem a restrição de cópias dos items, tornando os items unitários. Para cada item i e $Q_i > 1$, adiciona as

cópias do item i no conjunto L, incrementa a quantidade total de items a cada inclusão e subtraí Q_i. Assim, para cada item i do conjunto L, $Q_i = 1$. O problema 2KP tem uma entrada representada por uma tupla $I = (R, l, h, w, v)$, onde $R = (H, W)$ é um retângulo, $L = (1, \ldots, n)$ lista de retângulos, cada retângulo com dimensões (h_i, w_i) e valor v_i.

Decodificador 1: O decodificador 1 utiliza o método de BL (Bottom-Left) ou LB (Left-Bottom), nele temos a preferência de empacotar os items primeiramente pela esquerda ou por baixo. Assim, os n primeiros elementos do vetor de números reais representam a sequência de items e os $n + 1$ até $2n$ representam o empacotamento pelo BL ou LB. Assim, $2n$ chaves, sendo $[(o_i, t_i), (o_2, t_2), \ldots, (o_n, t_n)]$.

O Algoritmo 2 inicialmente cria um vector de cantos com o canto inicial $(0, 0)$, define se o empacotamento é pelo BL ou LB e o item que será empacotado. Após, verificamos se ele não ultrapassa as dimensões da mochila, se não, então empacotamos, removemos o canto que foi utilizado, além disso sendo as coordenadas do item $(x_1, y_1), (x_2, y_2)$, removemos todos os cantos onde no primeiro caso $x \leq x_1$ e $y \leq y_2$ ou no segundo caso $x \leq x_2$ e $y \leq y_1$, onde (x, y) são coordenadas de um canto. Por fim, adicionamos um canto a esquerda da altura do item empacotado, um a direita à baixo e outro na extremidade direita e acima do item.

Algorithm 2 Decodificador 1

Require: $I = (R, L, h, w, v)$ e a sequência de $2n$ chaves aleatórias $K = (o_1, \ldots, o_n, t_1, \ldots, t_n)$.

Ensure: Somatório de valores dos items empacotados.

1: $P \leftarrow [(0, 0)]$ // P é a lista de pontos de canto vigente
2: $A \leftarrow 0$ // pesos dos itens empacotados
3: $tam \leftarrow \text{SIZE}(K)/2$
4: **for** cada $j \leftarrow 1$ to tam **do**
5: $type \leftarrow K[j + tam]$
6: **if** $type > 0.5$ **then** // BL
7: **for** $c \in P$ **do** Pontos de canto pelo início da lista
8: **if** $(L_j.h + c.first) \leq H$ AND $(l_j.w + c.second) \leq W$ **then**
9: $A \leftarrow A + L_i.v$
10: $P \leftarrow P \setminus c$
11: **for** $c2 \in P$ **do**
12: **if** $c2.first \leq c.first$ AND $c2.second \leq (L_j.w + L_j.h + c.second)$ **then**
13: $P \leftarrow P \setminus c2$
14: **if** $c2.first \leq (L_j.w + c.first)$ AND $c2.second \leq c.second$ **then**
15: $P \leftarrow P \setminus c2$
16: Adiciona um ponto com x = x do primeiro ponto de canto a esquerda e y = altura do item
17: Adiciona um ponto com x = x do primeiro ponto de canto a direita e y = comprimento do iten
18: Adiciona um ponto com x = $c.first + L_j.w$ e y = $c.second + L_j.h$
19: **else** // LB
20: **for** $c \in \text{RESERVE}(P)$ **do** Pontos de canto pelo fim da lista

Repete todo o processo da linha 8-18, apenas pegando do final da lista de cantos

21: **return** A

Decodificador 2: Decodificador 2 utiliza a ideia de bandeja, onde é empacotado os itens até o limite da largura da mochila, após é feito um limite acima do item mais alto destes empacotados, uma bandeja. O vetor de números reais representam a sequência de items a serem empacotados. O algoritmo soma os valores da largura dos itens até o limite da largura da mochila e define a altura pelo altura como a altura do item que tem altura maior. Os próximos items a serem empacotados começam a partir desta altura e o processo se repete até atingir a altura da mochila.

Decodificador 3: Decodificador 3 utiliza a mesma ideia do decodificador 2. As diferenças são que escolhemos aleatoriamente os pontos de cantos, e se no ponto de canto escolhido o item não couber, então ele é descartado e começa o processo para o próximo item.

Decodificador 4: Decodificador 4 use como caixa preta o algoritmo do Flávio Miyazawa. Nele é usado a estratégia de escada, onde existe um parâmetro, *alpha*, que define em qual região da escada o item será empacotado. A escada é uma região onde item mais a esquerda tem altura maior ou igual ao item à esquerda e assim por diante. O vetor de números reais representam a sequência de items e os $n + 1$ até $2n$ representam o valor do *alpha*.

3 Experimentos Computacionais

Os experimentos computacionais foram executados em um máquina com oito cores na plataforma Azure. Na primeira etapa dos experimentos executamos os quatro decodificadores para identificar os mais promissores a terem os seus parâmetros aprimorados pelo *irace*.

3.1 BRKGA

Através de experimentos anteriores e intuição, escolhemos a seguinte configuração de parâmetros como promissora para os decodificadores. São mostrados na coluna *Sem Tunnig* da Tabela 1. Após os experimentos com os decodificadores, escolhemos os dois que tiveram os melhores resultados entre o conjunto de instâncias para serem *tunados* pelo Irace. Os melhores foram o Deco1 e o Deco4.

Lista de parâmetros utilizados para o *tunning* pelo Irace do Deco1 e Deco2 e seus devidos intervalos:

- `n`: valor de a no intervalo $(3, 12)$, onde $a * \log l$;
- `pe`: intervalo $(0.07, 0.20)$;
- `pm`: intervalo $(0.07, 0.20)$;
- `rhoe`: intervalo $(0.40, 0.90)$;
- `K`: intervalo $(1, 5)$;

Tabela 1: Parâmetros dos decodificadores sem *tunning* e com *tunning* pelo Irace

	Sem Tunnig	Deco1 & Deco3	Deco4
`n`	7 * $\log l$	7 * $\log l$	8 * $\log l$
`pe (%)`	10	8	11
`pm (%)`	10	18	12
`rhoe`	0.70	0.77	0.74
`K`	3	5	4
`X_INTVL`	100	444	284
`X_NUMBER`	2	4	3
`SizeRate (%)`	98	99	84
`FitRate (%)`	5	5	4
`reset`	400	1026	409

- `X_INTVL`:intervalo $(50, 500)$;
- `X_NUMBER`: intervalo $(1, 5)$;
- `SizeRate`: intervalo $(0.80, 0.99)$;
- `FitRate`: intervalo $(0.01, 0.10)$;
- `reset`: intervalo $(300, 1500)$.

Utilizamos todas as instâncias exceto *gcut1, type1-1, type2-1 e type3-1* e definimos por 8 horas o tempo que o Irace teve para o *tuning*, tivemos problemas pela quantidade e domínio dos parâmetros. Na Tabela 1 nas colunas Deco1 & Deco3 e Deco4 estão os valores para os parâmetros definidos pelo Irace.

3.1.1 Critérios de Parada

Consideramos como critério de parada para nosso algoritmo 30 minutos de execução.

3.1.2 Resultados

Nesta primeira Tabela 2, temos os resultados para todas as instâncias obtidos pelo *gurobi* e pelos decodificadores sem ajuste dos parâmetros. Os melhores resultados estão destacados. A instância

type2-5 o número de items é superior a 200, com isso o solver *gurobi* não conseguiu resolver.

O conjunto de instâncias *gcut* basicamente o solver e os decodificadores conseguiram resolver bem, exceto o decodificador 2. O melhor decodificador foi o 4, tendo 19 instâncias com os melhores resultados e 7 destas melhores como resultados únicos. O Decodificador 1 também teve resultado considerável com 13 instâncias, conseguindo os melhores resultados e o tempo para atingir bem inferior ao Decodificador 4.Os decodificadores 2 e 3 não tiveram resultados bons. Deste modo, escolhemos os decodificadores 1 e 4 para serem *tunados* pelo Irace.

Na Tabela 3 temos os resultados para os decodificadores 1, 3 e 4 que tiveram os melhores resultados na etapa anterior. Podemos observar que o decodificador 1 teve 21 melhores resultados contra 20 do decodificador 4 e além disso, os tempos para encontrar a solução do decodificador 1 são bem menores. As instâncias com mais de 2000 itens os decodificadores se mostraram eficientes e como não sabemos a solução ótima, pois o solver não conseguiu resolver, acreditamos que estes valores provavelmente são os ótimos.

Tabela 2: Comparação do modelo exato com os decodificadores

Instância	Modelo				Decode 1			Decode 2			Decode 3			Decode 4		
	LI	LS	GAP	$t(s)$	LS_1	GAP	$t_1(s)$	LS_2	GAP	$t_2(s)$	LS_3	GAP	$t_3(s)$	LS_4	GAP	$t_4(s)$
gcut1	**48368**	48368	0	0.41	**48368**	0	0.42	43024	11.05	0.01	**48368**	0	0.01	**48368**	0	0.03
gcut2	**59798**	60478	1.12	440	59563	1.51	0.44	57996	4.10	0.01	59335	1.89	8.11	**59798**	1.12	37.17
gcut3	**61275**	61785	0.83	1675	**61275**	0.83	1.24	59895	3.06	1.57	60663	1.82	29.56	**61275**	0.83	83.37
gcut4	48479	-	-	2121	**61305**	-	4.20	60504	-	10.63	**61305**	-	27.84	61191	-	3.94
gcut5	**195582**	195720	0.07	8.63	**195582**	0.07	0.16	193379	1.20	0.01	**195582**	0.07	0.33	**195582**	0.07	0.02
gcut6	**236305**	241639	2.26	175	**236305**	2.21	0.07	224399	7.13	0.05	**236305**	2.21	1.71	**236305**	2.21	0.06
gcut7	**240143**	240289	0.06	885	**240143**	0.06	10.34	238974	0.55	0.05	**240143**	0.06	0.73	**240143**	0.06	1.65
type1-1	27897	28963	3.82	1746	27897	3.68	42.20	27852	3.84	0.11	27852	3.84	0.34	**28032**	3.21	0.53
type1-2	11362	1,44E+08	99.99	1800	**28494**	99.98	15.40	27528	99.98	0.12	28014	99.98	16.65	**28494**	99.98	05.08
type1-3	**28677**	29907	4.11	1542	**28677**	4.11	0.05	**28677**	4.11	0.08	**28677**	4.11	0.02	**28677**	4.11	0.20
type1-4	**29271**	29804	1.79	1527	28488	4.42	6.65	28104	5.70	1.84	28488	4.42	0.96	28836	3.25	843.08
type1-5	19668	175770000	99.99	1800	29354	99.98	196.07	28947	99.98	55.96	28845	99.98	8.91	**29386**	99.98	172.32
type1-6	28944	29742	2.68	1750	28746	3.35	54.55	28395	4.53	35.52	28746	3.35	0.81	**29434**	1.04	147.69
type1-7	20634	212510000	99.99	1800	29460	99.99	7.25	29403	99.99	0.28	29403	99.99	0.73	**29508**	99.99	168.61
type1-8	27357	200640000	99.99	1800	**28956**	99.99	3.11	28143	99.99	0.75	28842	99.99	3.99	28842	99.99	14.68
type2-1	10224	136080000	99.99	1800	**29730**	99.98	1.67	**29730**	99.98	2.31	29610	99.98	0.15	**29730**	99.98	0.25
type2-2	10721	406680000	100.0	1800	**29708**	99.98	8.42	28359	99.98	12.17	29215	99.98	73.85	29646	99.98	40.37
type2-3	23207	-	-	1800	29881	-	5.48	29574	-	08.05	29664	-	129.68	**29944**	-	1569.02
type2-4	21815	-	-	1800	29973	-	29.98	29788	-	40.91	29976	-	58.56	**30000**	-	3.28
type2-5	-	-	-	-	**29988**	-	49.85	29709	-	2.28	29973	-	7.95	**29988**	-	476.64
type3-1	20098	189920000	99.99	1800	28752	99.98	209.05	29512	99.98	4,5	**29736**	99.98	52.50	29530	99.98	22.36
type3-2	12067	190100000	99.99	1800	27955	99.98	131.35	**29430**	99.98	101.94	28453	99.98	79.40	27868	99.98	1576.15
type3-3	19294	-	-	1800	**30000**	-	20.58	**30000**	-	1648.75	**30000**	-	535.40	**30000**	-	55.90
type2-4	13466	-	-	1800	29957	-	644.50	29928	-	106.38	29959	-	158.84	**29994**	-	1742.3
type2-5	19675	-	-	1800	29972	-	100.30	29757	-	700.197	29928	-	538.30	**29988**	-	121.31

Tabela 3: Resultados dos decodificadores após o *tunnig*.

Instância	Decode 1			Decode 3			Decode 4		
	LS_1	GAP	$t_1(s)$	LS_3	GAP	$t_3(s)$	LS_4	GAP	$t_4(s)$
gcut1	**48368**	0,00	0.03	**48368**	0.00	0.01	**48386**	0.00	0.11
gcut2	59563	1,51	2.55	59335	1.89	65.68	**59798**	1,12	138.20
gcut3	**61275**	0.83	0.92	60663	1.82	11.42	**61275**	0.83	49.09
gcut4	**61380**	-	100.33	**61380**	-	3.42	**61380**	-	293.90
gcut5	**195582**	0,07	24.98	**195582**	0.07	3.10	**195582**	0.07	0.11
gcut6	**236305**	2.21	0.10	**236305**	2.21	0.01	**236305**	2.21	0.17
gcut7	**240143**	0.06	2.58	**240143**	0.06	45.40	**240143**	0.06	13.53
type1-1	**28032**	3.21	3.60	27852	3.84	0.22	**28032**	3.21	3.10
type1-2	**28494**	99.98	0.43	28014	99.98	16.10	**28494**	99.98	0.95
type1-3	**28677**	4.11	0.20	**28677**	4.11	0.31	**28677**	4.11	0.27
type1-4	**29271**	1.79	192.41	28488	4.42	1.20	28836	3.25	22.90
type1-5	29481	99.98	344.81	29386	99.98	353.57	**29610**	99.98	1062.94
type1-6	**29434**	1.04	114.35	29029	2.40	180.77	**29434**	1.04	452.30
type1-7	**29700**	99.99	481.48	29403	99.99	12.18	29508	99.99	48.9
type1-8	**28956**	99.99	6.78	28842	99.99	17.21	28842	99.99	4.28
type2-1	**29730**	99.98	0.30	**29730**	99.98	122.5	**29730**	99.98	22.45
type2-2	**29928**	99.99	20.62	29442	99.99	758.37	29850	99.99	690.41
type2-3	**29944**	-	106.50	**29944**	-	306.8	**29944**	-	482.14
type2-4	29976	-	759.35	**30000**	-	690.60	**30000**	-	1410.56
type2-5	**30000**	-	320.70	**30000**	-	942.26	**30000**	-	751.73
type3-1	**29736**	99.98	89.52	29512	99.98	162.13	**29736**	99.98	216.52
type3-2	**29191**	99.98	108.60	28848	99.98	50.14	**29191**	99.98	990.78
type3-3	**30000**	-	18.43	**30000**	-	696.77	**30000**	-	1423.38
type3-4	**30000**	-	424.22	**30000**	-	1596.5	29994	-	816.22
type3-5	29979	-	54.08	29928	-	245.46	**29988**	-	400.86

3.2 *Time-to-Target Plots*

Para analisar os decodificadores utilizamos o *Time-to-Target Plots* nas versões padrão e com *tunning*. Escolhemos as instâncias com a maior de quantidade de itens, *gcut7, type-1-8, type2-5, type3-5* com os respectivos *targets*, 238974, 28842, 29973, 29928. Executamos cada decodificador por 200 vezes com tempo limite para encontrar o *target* de 300 segundos.

A Figura 1 é mostrado os gráficos para os decodificadores com a configuração padrão. Podemos observar que nas duas primeiras figuras, todos os decodificadores conseguiram resolver facilmente as instâncias. Já na Figura 1c, o decodificador 2 praticamento não conseguiu resolver e os decodificadores 2 e 3 não conseguiram resolver todas as vezes com 300 segundos de limite. Na Figura 1d mostra a superioridade dos decodificadores 1 e 4, que foram os únicos que resolveram acima de 80% das execuções.

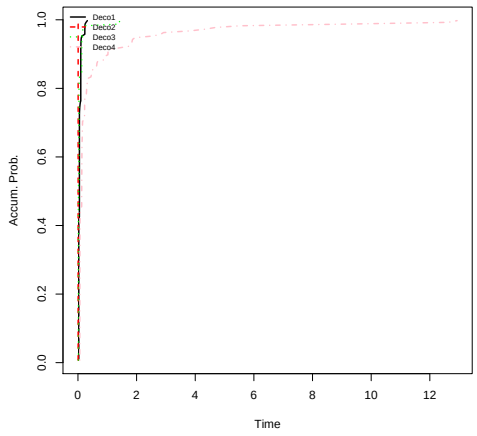
Na Figura 2 temos os gráfios de TTT-plots para os decodificadores 1 e 4. Podemos observar a completividade deles, uma leve vantagem do decodificador 1 no tempo para resolver as instâncias. Porém, na Figura 2d fica nítido uma superioridade do decodificador 4, que resolveu em torno de 90% das vezes contra 80% do decodificador 1.

4 Conclusões

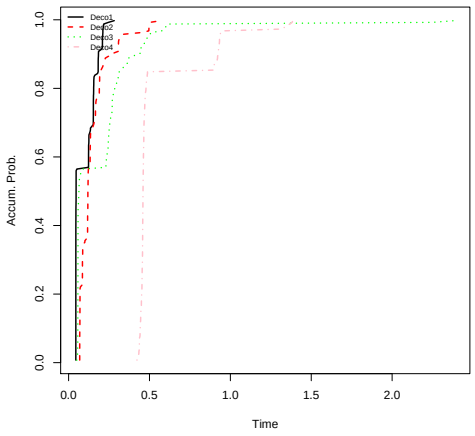
A utilização do BRKGA para resolução do 2KP se mostrou bastante eficiente, com os decodificadores 1 e 4 se destacando. Outro fator foi a importância da utilização do Irace, que mostrou que uma configuração adequada de parâmetros, com isso o decodificador 1 conseguiu superar os demais e ter melhores resultados de 11 para 21 instâncias. O solver *Gurobi* teve dificuldades em resolver a maiorias das instâncias e uma delas nem conseguiu solução, mostrando a importância de resolver o problemas 2KP com outras abordagens.

Referências

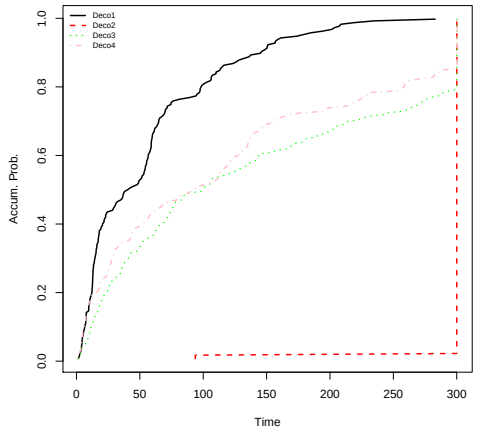
[1] Ramon Alvarez-Valdes, Francisco Parreño e Jose M Tamarit. “A GRASP algorithm for constrained two-dimensional non-guillotine cutting problems”. Em: *Journal of the Operational Research Society* 56.4 (2005), pp. 414–425.



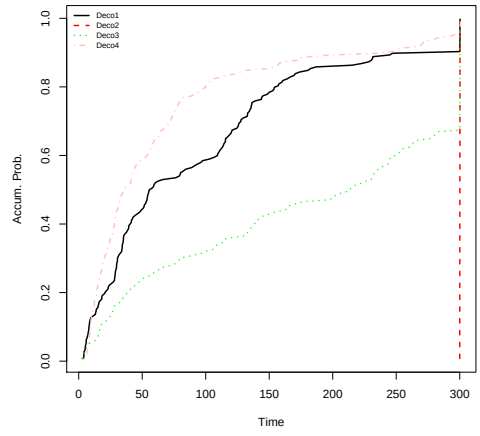
(a) Instância *gcut7*



(b) Instância *type1-8*

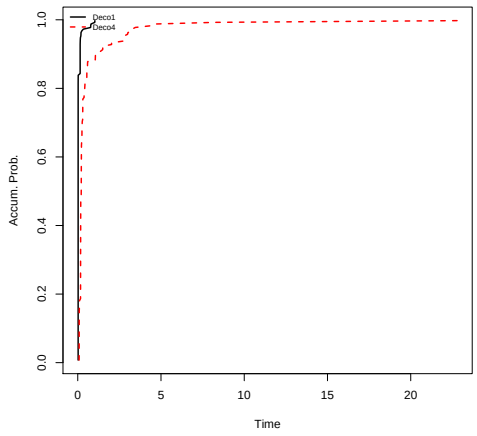


(c) Instância *type2-5*

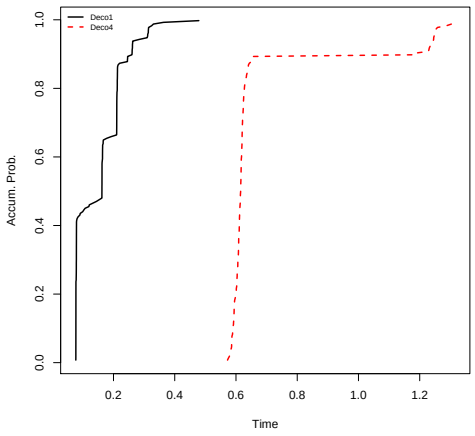


(d) Instância *type3-5*

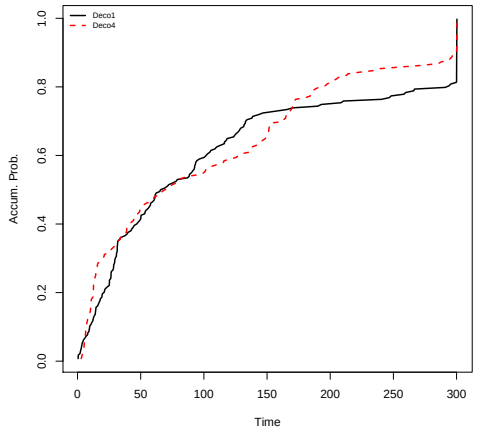
Figura 1: Gráfico TTT-plots comparando os decodificadores com configurações padrões.



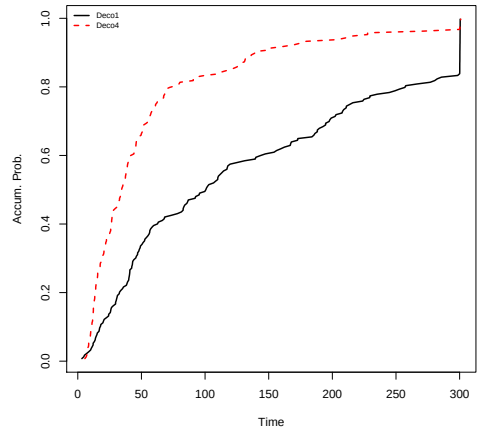
(a) Instância *gcut7*



(b) Instância *type1-8*



(c) Instância *type2-5*



(d) Instância *type3-5*

Figura 2: Gráfico TTT-plots comparando os decodificadores tunados.

[2] Ramón Alvarez-Valdés, Francisco Parreño e José Manuel Tamarit. “A tabu search algorithm for a two-dimensional non-guillotine cutting problem”. Em: *European Journal of Operational Research* 183.3 (2007), pp. 1167–1182.

[3] James C Bean. “Genetic algorithms and random keys for sequencing and optimization”. Em: *ORSA journal on computing* 6.2 (1994), pp. 154–160.

[4] JE Beasley. “A population heuristic for constrained two-dimensional non-guillotine cutting”. Em: *European Journal of Operational Research* 156.3 (2004), pp. 601–627.

[5] JE Beasley. “An exact two-dimensional non-guillotine cutting tree search procedure”. Em: *Operations Research* 33.1 (1985), pp. 49–64.

[6] Nicos Christofides e Charles Whitlock. “An algorithm for two-dimensional cutting problems”. Em: *Operations Research* 25.1 (1977), pp. 30–44.

[7] Harald Dyckhoff. “A typology of cutting and packing problems”. Em: *European Journal of Operational Research* 44.2 (1990), pp. 145–159.

[8] José Fernando Gonçalves e Mauricio GC Resende. “A parallel multi-population genetic algorithm for a constrained two-dimensional orthogonal packing problem”. Em: *Journal of Combinatorial Optimization* 22.2 (2011), pp. 180–201.

[9] José Fernando Gonçalves e Mauricio GC Resende. “Biased random-key genetic algorithms for combinatorial optimization”. Em: *Journal of Heuristics* 17.5 (2011), pp. 487–525.

[10] JC Herz. “Recursive computational procedure for two-dimensional stock cutting”. Em: *IBM Journal of Research and Development* 16.5 (1972), pp. 462–469.

[11] Stephen CH Leung et al. “A hybrid simulated annealing metaheuristic algorithm for the two-dimensional knapsack packing problem”. Em: *Computers & Operations Research* 39.1 (2012), pp. 64–73.

[12] Andrea Lodi, Silvano Martello e Michele Monaci. “Two-dimensional packing problems: A survey”. Em: *European journal of operational research* 141.2 (2002), pp. 241–252.