Uma análise do framework OptaPlanner aplicado ao problema de empacotamento unidimensional
An analysis of the OptaPlanner framework applied to the one-dimensional bin packing problem

Victor Rios de Souza
Instituto de Ciência e Tecnologia – Universidade Federal Fluminense (UFF)
28.895-532 – Rio das Ostras – RJ – Brasil
vrios@id.uff.br

Carlos Bazilio Martins
Instituto de Ciência e Tecnologia – Universidade Federal Fluminense (UFF)
28.895-532 – Rio das Ostras – RJ – Brasil
carlosbazilio@id.uff.br

ABSTRACT

Diversos problemas de otimização combinatória foram alvo de estudos nas últimas décadas. Em sua maioria, as soluções apresentadas estão diretamente ligadas a especificidade de cada problema, ou seja, essas soluções acabam não sendo reutilizáveis. Em paralelo a isso, solucionadores genéricos surgiram com o propósito de simplificar e facilitar a busca por soluções viáveis. O presente trabalho pretende explorar diferentes soluções, utilizando o framework OptaPlanner aplicado ao problema de empacotamento unidimensional e comparar os resultados obtidos entre si. Além disso, medir a performance das soluções para diferentes conjuntos de dados, disponibilizados pela OR-Library.

ABSTRACT

Several problems of combinatorial optimization have been studied in recent decades. For the most part, the solutions presented are directly linked to the specificity of each problem, that is, these solutions end up being not reusable. In parallel to this, generic solvers have emerged with the purpose of simplifying and facilitating the search for feasible solutions. The present work intends to explore different solutions, using the OptaPlanner framework applied to the one-dimensional bin packing problem and compare the obtained results among themselves. In addition, measure the performance of solutions for different data sets, made available by OR-Library.

CCS Concepts

•**Theory of computation** → *Models of computation;* •**Software and its engineering** → *Software notations and tools;*

Keywords

Otimização combinatória; OptaPlanner; empacotamento unidimensional; OR-Library
Combinatorial optimization; OptaPlanner; one-dimensional packaging; OR-Library

1. INTRODUÇÃO

O problema de empacotamento é considerado um problema clássico na otimização combinatória. As primeiras pesquisas envolvendo variações desse problema surgiram no início da década de 70 [7] [11]. Ainda assim, publicações recentes podem ser encontradas com novas abordagens e técnicas aplicadas a variações do mesmo problema [5] [18]. A demanda por novos estudos são fomentadas por inúmeras aplicações nas áreas de engenharia e computação, como por exemplo: corte de materiais[12]; alocação de carga em transportes [16]; balanceamento de carga para cluster de computadores na nuvem [19].

Apesar de ser um problema conhecido na área de pesquisa operacional, é possível encontrar na literatura diversos trabalhos que também relacionam esse tipo de problema com o campo de inteligência artificial, estabelecendo uma forte relação entre elas [17] [9] [3].

Em sua essência, o problema clássico de empacotamento unidimensional consiste em minimizar o número de pacotes ou caixas utilizadas para alocar uma certa quantidade de itens. Os pacotes ou caixas possuem capacidade restrita e igualitária, os itens possuem diferentes pesos ou tamanhos e um item deve estar alocado em apenas um pacote. Na Figura 1 é possível visualizar uma representação do problema de empacotamento unidimensional.

Apesar da facilidade em descrevê-lo, trata-se de um problema de difícil resolução. Considerando a hipótese que `P` $\neq$ `NP`, o problema do empacotamento é classificado como `NP-Difícil`, ou seja, é improvável que haja um algoritmo de tempo polinomial capaz de resolvê-lo de forma otimizada [8].

1.1 Motivação

Nos últimos anos diversos trabalhos na área de pesquisa operacional e inteligência computacional têm surgido com o propósito de resolver problemas complexos de natureza `NP-Difícil`. Essas pesquisas, em sua maioria, propõe novas soluções para problemas específicos, visando obter o melhor

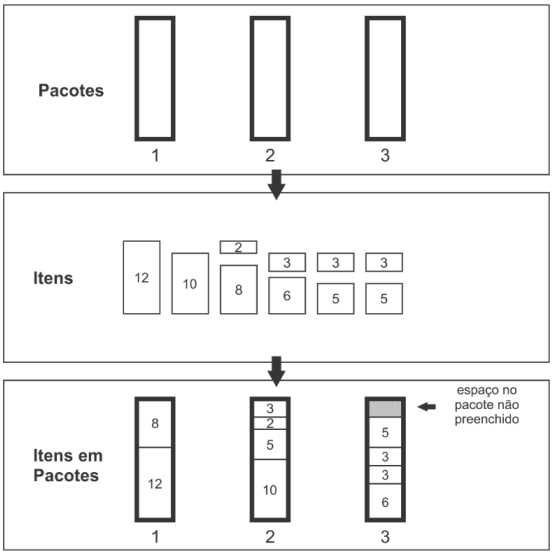

Figure 1: Representação do problema de empacotamento unidimensional

resultado possível, seja ele de minimização ou maximização. Porém, desenvolver soluções dessa natureza pode ser algo custoso e demorado. Produtos de software, normalmente chamados de "solucionadores", surgiram com o objetivo de fornecer soluções mais generalistas. Alguns são produtos comerciais e só estão disponíveis através da aquisição de uma licença ou obtenção de uma licença acadêmica por meio de outra instituição. Algumas dessas ferramentas podem ser observadas na Tabela 1.

Dentre eles destaca-se o OptaPlanner, um *framework open source* em Java, sob a licença Apache. O OptaPlanner é parte do projeto Knowledge Is Everything (KIE) da Red Hat, que engloba outros projetos, como por exemplo, o Drools, um sistema de gerenciamento de regras de negócios com mecanismo de regras baseado em inferência [10]. Na Figura 2 é possível visualizar os subprojetos do projeto KIE e como eles se relacionam.

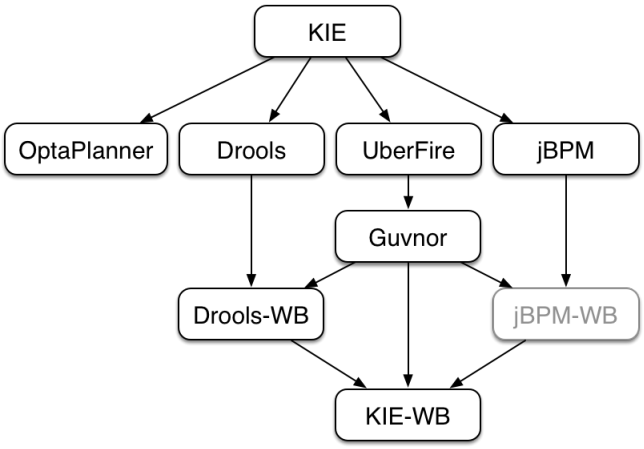

Figure 2: Subprojetos do projeto KIE

O OptaPlanner pode ser definido como um *framework* de

Table 1: Ferramentas de otimização

Classificação	Nome	Linguagem / Licença
Constraint Programming Solvers	CHOCO	java library, open source
	Gecode	c++, free
	ILog	binary, free com licença acadêmica
	JACOP	java , open source
	MiniZinc/G12	binary, free para estudantes
	or-tools	c++, open source, APIs em Java, Python e .NET
Mixed Integer Programming Solvers	BCP	c++, open source
	CBC	c++, open source
	CPlex	binary, free com licença acadêmica
	GLPK	c, open source
	gurobi	binary, free com licença acadêmica
	LPSolve	c, open source
	SCIP	binary, free para uso acadêmico
Linear Programming Solvers	CLP	c++, open source
	MINOS	fortran 77, proprietário
	SimplexSolver	java, open source
Local Search Solvers	Local Solver	binary, free com licença acadêmica
	OptaPlanner	java, open source
SAT Solvers	cryptominisat	c++, open source
	Lingeling	c, open source
	MiniSat	binary, free
	UBCSAT	c, open source
Hybrid Solvers	SCIP	binary, free para uso acadêmico

inteligência artificial, capaz de solucionar problemas de otimização com restrições. Apesar de estar classificado na Tabela 1 como um solucionador de busca local, além de utilizar essa técnica, sua configuração permite o uso de outras estratégias que vão desde força bruta, até heurísticas [4].

Na literatura foram encontrados trabalhos onde o OptaPlanner é citado e também objeto de estudo. Em Čimbora [20], foi realizada uma proposta de melhoria do *benchmark* existente no *framework*. Em outro, foi utilizada uma implementação preexistente do *framework* para solucionar o problema de roteamento de veículos [13]. O OptaPlanner também foi utilizado para solucionar parte do problema de roteamento de veículos envolvendo o mecanismo de coleta de lixo [15].

Já outro trabalho encontrado, refere-se a uma proposta de solução apresentada para o problema de gerenciamento de atribuição de tarefas [14]. Este sendo o que mais se aproxima do objetivo que pretende-se alcançar com o presente trabalho, se levada em consideração a comparação dos resultados obtidos através das configurações de *benchmark* do *framework*. Contudo, não foi encontrado um trabalho no qual essa comparação se dá aplicando a mesma solução em diferentes conjuntos de dados, tampouco analisar o quanto essas soluções podem escalar em termos de performance.

1.2 Objetivos

Primordialmente, o presente trabalho objetiva:

- Explorar o framework OptaPlanner aplicado ao problema de empacotamento unidimensional;

- Comparar os resultados obtidos entre si, para as diferentes configurações da solução desenvolvida;
- Mensurar a performance a medida que o tamanho do problema cresce;

Outros aspectos secundários a serem explorados:

- Testar a pluralidade de configurações oferecidas pelo framework OptaPlanner;
- Traçar paralelos com outros trabalhos existentes, que utilizem o mesmo conjunto de dados;
- Identificar pontos de melhoria e evolução da ferramenta.

1.3 Metodologia de Pesquisa

Quanto a metodologia de pesquisa, pretende-se realizar uma investigação aplicada ao problema de empacotamento unidimensional, e exploratória em relação ao *framework* OptaPlanner. Ademais, visa adotar uma abordagem quantitativa sobre os resultados obtidos, demonstrando a qualidade da solução (número de restrições violadas ou não atendidas) e o tempo de execução de cada uma delas, para cada conjunto de dados.

2. FORMULAÇÃO DO PROBLEMA

Para um conjunto finito de pacotes com capacidade idêntica C, um conjunto de itens `V = {1,...,n}` com pesos `w1,...`,wn, o objetivo do problema de empacotamento unidimensional é alocar cada item a um pacote minimizando o número de pacotes utilizados. Esta alocação deve respeitar a capacidade máxima dos pacotes. O seguinte modelo matemático pode ser usado para representar o problema de empacotamento unidimensional:

Função objetivo:

$$Minimizarz = \sum_{i=1}^{m} y_i \tag{1}$$

Sujeito a:

$$\sum_{j=1}^{m} w_j x_{ij} \leq C y_i \; i \epsilon N = \{1, ..., n\} \tag{2}$$

$$\sum_{j=1}^{m} x_{ij} = 1, \; j \epsilon N \tag{3}$$

$$y_i \epsilon \{0, 1\}, \; j \epsilon N \tag{4}$$

$$x_{ij} \epsilon \{0, 1\}, \; i \epsilon N \;\; j \epsilon N \tag{5}$$

Considere a variável binária y_i igual a 1 se o pacote i está sendo utilizado e 0, caso contrário. A variável binária x_{ij} indica se o item j está alocado ao pacote i ($x_{ij} = 1$) ou não ($x_{ij} = 0$). Assume-se também que C > 0 e $w_j <$ C, ou seja, a capacidade do pacote deve ter valor positivo e o peso de um item não pode ser maior que a capacidade de um pacote.

O objetivo definido por (1) é minimizar o número de pacotes utilizados. Em (2) é definida a primeira restrição, onde a soma dos pesos dos itens alocados em um pacote deve ser menor ou igual a sua capacidade. Em (3) restringe que um item deve estar alocado em somente um pacote. As restrições (4) e (5), limitam os valores das variáveis de decisão.

3. DESENVOLVIMENTO DA SOLUÇÃO

O desenvolvimento da solução para o problema do empacotamento unidimensional usando o *framework* OptaPlanner pode ser dividido em 3 etapas distintas: modelagem de domínio e implementação; estratégia de pontuação da solução e regras de restrições; e configuração do *framework*.

3.1 Modelagem de domínio e implementação

Para problemas de planejamento ou otimização, como é o caso do problema de empacotamento unidimensional, é necessário identificar as entidades de domínio e propriedades que serão alvo do solucionador. Para isso, a compreensão de alguns conceitos do *framework* OptaPlanner se faz necessária:

- Entidade de Planejamento: Indica a classe, ou classes, que podem sofrer alterações durante o processo de busca da solução. Para o caso do problema de empacotamento, de acordo com a modelagem de domínio, a classe Item representa essa entidade;
- Fato Problema: São as classes que não podem sofrer alterações em sua entrada de dados pelo solucionador. No exemplo estudado, a classe Bin é a única que possui essa característica;
- Variável de Planejamento: Determina a propriedade da Entidade de Planejamento que pode sofrer alterações durante o processo de resolução do problema. Também pode ser aplicado a mais de uma propriedade, desde que a classe seja uma Entidade de Planejamento. No presente trabalho, essa propriedade é a bin localizada na classe Item;
- Solução de Planejamento: A classe que representa a solução do problema. Essa classe contém todo o conjunto de dados e todas as Entidades de Planejamento. No caso, a classe BinPackingSolver representa essa entidade.

O *framework* OptaPlanner utiliza um recurso da própria linguagem Java, chamado *annotations*, para "marcar" as classes e propriedades com as configurações supracitadas. Dessa forma, é possível determinar a Entidade de Planejamento, Variável de Planejamento e Solução do Planejamento, com as seguintes anotações respectivamente: `@PlanningEntity`, `@PlanningVariable` e `@PlanningSolution`. O Fato Problema é identificado por exclusão, caso a classe não seja anotada, então significa que esta é uma classe Fato Problema.

De acordo com as definições da Unified Model Language (UML) [2], podemos representar a modelagem de domínio para o problema de empacotamento unidimensional, através do diagrama de classes da Figura 3:

O mecanismo de funcionamento do *framework* consiste em modificar o valor das Variáveis de Planejamento, utilizando a estratégia de busca local configurada, de modo que todas as regras de restrições sejam atendidas, até que a condição de parada seja satisfeita.

Nas subseções posteriores, serão detalhados com mais clareza as etapas de definição das regras de restrições e configuração do *framework*.

3.2 Estratégia de pontuação da solução e regras de restrição

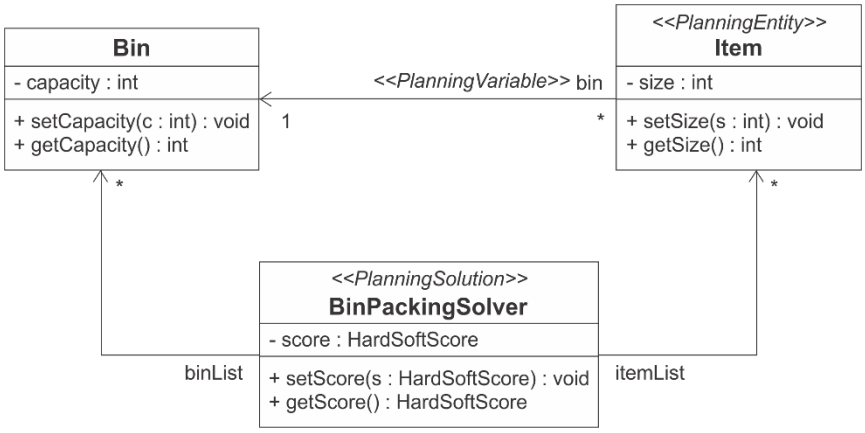

Figure 3: Diagrama de classes do problema de empacotamento unidimensional

A pontuação de uma solução define se uma solução é melhor que outra. Quanto maior a pontuação, melhor é a solução. Todas as técnicas de pontuação são baseadas em restrições e possuem aspectos que devem ser levados em consideração, como o sinal, peso e camada da pontuação.

O sinal da pontuação de uma restrição indica se a restrição é de minimização ou maximização. Restrições de minimização possuem pontuação com o sinal negativo e ficam melhores quando tendem a zero. Já as restrições de maximização possuem pontuação com o sinal positivo e ficam melhores quando se afastam do valor zero. Quando uma pontuação recebe um valor negativo, dizemos que ela foi penalizada, já quando recebe um valor positivo, dizemos que ela foi bonificada.

O peso da pontuação, determina se restrições em uma mesma camada possuem mais importância que outra. Isso se dá pelo fato de restrições diferentes serem penalizadas ou bonificadas com valores diferentes. Por exemplo, se uma restrição for penalizada com um valor maior que outra, evidencia que a primeira possui uma importância maior que a segunda.

Já as camadas subdividem as restrições em conjuntos distintos. Cada camada possui uma pontuação diferente. Por padrão, o OptaPlanner disponibiliza 3 camadas para serem utilizadas: rígida, média e suave.

Para escrever as regras de restrições foi utilizado o Drools, um sistema de gerenciamento de regras de negócios, que também é um projeto do KIE. O Drools está nativamente incorporado ao OptaPlanner. Entretanto, como alternativa, também é possível escrever as regras utilizando uma classe Java simples.

Para o problema de empacotamento unidimensional, foram desenvolvidas 3 restrições, sendo 2 restrições rígidas e 1 restrição suave. As restrições rígidas ou *hard constraints*, são aquelas que, quando violadas, inviabilizam a solução. As restrições suaves ou *soft constraints*, podem ser violadas que não inviabilizam a solução, embora o valor de seu resultado indique a qualidade da solução encontrada.

A primeira regra de restrição, que pode ser observada na Figura 4, serve para determinar que a soma dos valores dos itens de um pacote não exceda a capacidade máxima do pacote. A restrição é acionada quando para um dado pacote `$bin` com capacidade `$capacity`, a soma do tamanho dos itens `$sizeTotal` for maior que `$capacity`, se o pacote do item bin for igual a `$bin`. Então, a pontuação da camada de restrição rígida recebe uma penalização de `$capacity - $sizeTotal`. Por ser uma restrição rígida, qualquer valor menor que zero inviabiliza a solução, entretanto, esse regra utiliza o critério de peso para pontuar a restrição. A solução fica mais distante de um resultado viável a medida que o valor da soma dos itens for maior que a capacidade máxima do pacote.

Algorithm 1 Regra que verifica capacidade máxima do pacote

```
when :
    $bin : Bin($capacity : capacity)
    accumulate(
        Item(
            bin != null,
            bin == $bin,
            $size : size
        );
        $sizeTotal : sum($size);
        $sizeTotal > $capacity
    )
then :
    int $score = $capacity - $sizeTotal;
    scoreHolder.addHardConstraintMatch(kcontext, $score);
```

Figure 4: Primeira regra de restrição

A segunda regra de restrição também penaliza a pontuação localizada na camada rígida da solução. Quando um item possui um pacote `bin` igual a nulo, então a pontuação recebe uma penalização de -1. Em outras palavras, para se ter uma solução viável, todos os itens da solução devem possuir um pacote associado. Neste caso a penalização da restrição não possui um peso, já que o mesmo valor (-1) é atribuído na penalização. Essa regra pode ser visualizada na Figura 5.

A última regra definida, identificada na Figura 6, verifica a utilização completa do pacote. Esse critério não é uma premissa para a viabilidade da solução. Por essa razão, fica categorizada na camada suave. A penalização ocorre quando para um dado pacote `$bin` com capacidade `$capacity`, o somatório `$sizeTotal` do tamanho dos itens cujo pacote `bin` é igual a `$bin`, for menor que `$capacity`, então a pontuação será penalizada com o valor `$sizeTotal - $capacity`.

Algorithm 2 Regra que verifica a existência de item no pacote

```
when :
    Item(bin == null)
then :
    scoreHolder.addHardConstraintMatch(kcontext, -1);
```

Figure 5: Segunda regra de restrição

Algorithm 3 Regra que verifica utilização completa do pacote

```
when :
    $bin : Bin($capacity : capacity)
    accumulate(
        Item(
            bin != null,
            bin == $bin,
            $size : size
        );
        $sizeTotal : sum($size);
        $sizeTotal < $capacity
    )
then :
    int $score = $sizeTotal - $capacity;
    scoreHolder.addSoftConstraintMatch(kcontext, $score);
```

Figure 6: Terceira regra de restrição.

3.3 Configuração do framework

O *framework* OptaPlanner permite que sejam realizadas configurações que definem diversos aspectos da solução, como: caminho do arquivo contendo as regras de restrições, a condição de parada da solução, heurística de construção e algoritmo de busca local. Uma combinação dessas estratégias pode ser utilizada em um mesmo arquivo de configuração, a fim de se encontrar a melhor configuração para um determinado problema. Este tipo de configuração é chamado de configuração de *benchmark*.

Para o problema foram utilizadas 2 configurações, utilizadas em momentos distintos. A primeira, uma configuração de *benchmark*, para encontrar a melhor estratégia de busca local. Foram utilizadas 3 estratégias de busca local para comparação: *Tabu Search*, *Simulated Annealing* e *Late Acceptance*. Para esta configuração, foram utilizados 2 critérios na condição de parada: ou quando a pontuação rígida chegar ao valor zero (solução viável), ou quando extrapolado o tempo de 30 minutos após o início da busca.

Não foi realizada nenhuma customização nas configurações padrão de nenhum dos algoritmos de busca local. Além disso, foram utilizados os mesmos seletores de movimentação para todos eles (*swapMoveSelector* e *pillarSwapMoveSelector*). Também foi utilizado a heurística *first fit* para construção da solução inicial. Estes princípios tem por objetivo comparar os resultados obtidos considerando os mesmos critérios de partida.

Após executar a solução para o primeiro conjunto de dados, foi identificado que o algoritmo de busca local *Simulated Annealing*, apresentou os melhores resultados. Detalhes sobre os resultados serão apresentados adiante na subseção 4.3.

A segunda configuração utilizou somente o algoritmo de busca local Simulated Annealing. Foi utilizado como condição de parada o valor zero para a restrição rígida, ou seja, até uma solução viável ser encontrada. Essa configuração foi aplicada a todos os conjuntos de dados, sendo quatro no total. Detalhes sobre os conjuntos de dados testados podem ser vistos na subseção 4.2.

4. EXPERIMENTOS COMPUTACIONAIS

Os experimentos computacionais foram divididos em duas etapas. Na primeira, foi executada a configuração de *benchmark*, citada na seção anterior, no menor conjunto de dados (*binpack1*), a fim de identificar o melhor algoritmo de busca local, ou seja, a solução capaz de resolver o problema atendendo todas as restrições, no menor tempo possível. Na segunda, foi aplicada a solução escolhida na etapa anterior para os demais conjuntos de dados.

4.1 Ambiente Computacional

A máquina utilizada para o experimento possui em sua configuração um processador Intel Core i7-4500U 1.8GHz e 8GB de memória RAM. Todos os testes rodaram em um sistema operacional Windows 10 Pro de 64 bits, versão 1803 e compilação 17134.885. Como já mencionado anteriormente, o desenvolvimento da solução foi realizada com linguagem Java, versão 1.8.0, e *framework* OptaPlanner, versão 7.21.0.final.

4.2 Instâncias Testadas

Foram utilizadas 4 instâncias de conjuntos de dados para o problema do empacotamento unidimensional disponibilizadas na OR-Library [1]. Cada um deles contendo 20 problemas. Esses conjuntos de dados possuem itens de tamanho uniforme, que variam entre 20 a 100, com pacotes de capacidade 150. A diferença entre eles se dá na quantidade de itens e pacotes existentes em cada problema. Os problemas variam a partir de 120 itens e 46 pacotes, até 1000 itens e 412 pacotes, ou seja, a complexidade do problema cresce na escala que a quantidade de itens e pacotes aumenta. As informações dos conjuntos de dados podem ser vistas na Tabela 2.

Table 2: Conjunto de dados OR-LIBRARY

Instância	Qtd. Problemas	Cap. Pacote	Qtd. Itens
binpack1	20	150	120
binpack2	20	150	250
binpack3	20	150	500
binpack4	20	150	1000

4.3 Resultados da primeira etapa

Como mencionado anteriormente, os resultados da primeira etapa foram obtidos provenientes da configuração de *benchmark* do *framework* OptaPlanner explicado na subseção 3.3.

De acordo com as informações apresentadas na Tabela 3, em relação a pontuação, os algoritmos de busca local *Tabu Search* e *Simulated Annealing* obtiveram resultados idênticos. Ambos alcançaram soluções viáveis para todos os problemas do conjunto de dados. Em contrapartida, o algoritmo *Late Acceptance* falhou em 7 dos 20 problemas tes-

Table 3: Pontuação x tempo gasto entre os algoritmos de busca local

binpack1	*Tabu Search*		*Simulated Annealing*		*Late Acceptance*	
Problema	**Pontuação**	**Tempo**	**Pontuação**	**Tempo**	**Pontuação**	**Tempo**
0	0hard/-122soft	6.265ms	0hard/-122soft	1.727ms	0hard/-122soft	279.025ms
1	0hard/-145soft	679ms	0hard/-145soft	373ms	0hard/-145soft	34.421ms
2	0hard/-106soft	966 ms	0hard/-106soft	3.097ms	0hard/-106soft	172.998ms
3	0hard/-65soft	937 ms	0hard/-65soft	2.400ms	-3hard/-68soft	1.800.000ms
4	0hard/-146soft	3.129ms	0hard/-146soft	300ms	0hard/-146soft	37.280ms
5	0hard/-78soft	619ms	0hard/-78soft	418ms	0hard/-78soft	1.505.699ms
6	0hard/-63soft	985ms	0hard/-63soft	831ms	-1hard/-64soft	1.800.006ms
7	0hard/-55soft	5.000ms	0hard/-55soft	611ms	-3hard/-58soft	1.800.000ms
8	0hard/-172soft	654ms	0hard/-172soft	259ms	0hard/-172soft	27.796ms
9	0hard/-30soft	23.649ms	0hard/-30soft	1.698ms	-6hard/-36soft	1.800.001ms
10	0hard/-120soft	2.466ms	0hard/-120soft	642ms	0hard/-120soft	255.052ms
11	0hard/-103soft	2.882ms	0hard/-103soft	311ms	0hard/-103soft	1.202.570ms
12	0hard/-20soft	57.362ms	0hard/-20soft	2.473ms	-8hard/-28soft	1.800.000ms
13	0hard/-148soft	849ms	0hard/-148soft	464ms	0hard/-148soft	39.870ms
14	0hard/-127soft	1.067ms	0hard/-127soft	585ms	0hard/-127soft	79.600ms
15	0hard/-98soft	5.340ms	0hard/-98soft	969ms	-1hard/-99soft	1.800.000ms
16	0hard/-112soft	1.008ms	0hard/-112soft	488ms	0hard/-112soft	219.074ms
17	0hard/-97soft	32.370ms	0hard/-97soft	1.206ms	-1hard/-98soft	1.800.000ms
18	0hard/-95soft	716ms	0hard/-95soft	548ms	0hard/-95soft	110.804ms
19	0hard/-178soft	645ms	0hard/-178soft	310ms	0hard/-178soft	19.549ms
Média	0hard/-104soft	7.379ms	0hard/-104soft	985ms	-2hard/-106soft	829.187ms
Total	0hard/-2080soft	147.588ms	0hard/-2080soft	19.710ms	-23hard/-2103sof	16.583.745ms

tados, ou seja, obteve valor da restrição rígida menor que zero.

Em relação a performance, é possível observar na coluna "Tempo", o tempo gasto em milissegundos para cada problema, entre as 3 soluções. Os resultados observados na Tabela 3 indicam que o algoritmo *Simulated Annealing* foi capaz de solucionar o mesmo conjunto de dados e obtendo o mesmo resultado que o algoritmo *Tabu Search*, gastando 7,48 menos tempo. Em apenas 2 problemas o algoritmo *Tabu Search* foi superior ao algoritmo *Simulated Annealing* nesse quesito.

Outro critério de performance a observar é a velocidade do cálculo da pontuação da solução. Esse item indica o custo de se adicionar novas regras de restrição a solução. Outra vantagem de se obter um indicador superior nesse critério é a capacidade de se escapar do ótimo local, uma vez que mais tentativas podem ser realizadas por segundo. Nesse quesito o algoritmo *Tabu Search* alcançou o melhor resultado. Em média ele foi capaz de calcular 56.822 pontos por segundo, contra 23.703 pontos por segundo do algoritmo *Simulated Annealing*.

Apesar disso, foi adotado para a segunda etapa o algoritmo *Simulated Annealing* para ser aplicado como solução para as outras 3 instâncias do conjuntos de dados. Um dos critérios para a escolha desse algoritmo se deu pelo fato de suas regras não possuírem tendência de alteração. Além do mais, o resultados obtidos pelo algoritmo foram satisfatórios.

4.4 Resultados da segunda etapa

Na segunda etapa foram executadas as instâncias *binpack2*, *binpack3* e *binpack4*, utilizando a configuração com algoritmo de busca local *Simulated Annealing*. Assim como ocorreu na primeira etapa, para todos os problemas das outras instancias testadas, o solucionador obteve soluções viáveis ou ótimas.

Ao fim do processo de busca, o solucionador é capaz de mensurar a complexidade das instâncias dos conjuntos de dados testados. Na Tabela 4 é possível observar essa informação para as instâncias *binpack1*, *binpack2*, *binpack3* e *binpack4*, ao lado do nome da instância, sendo elas respectivamente: 5.898, 25.425, 100.600 e 400.550. Através desse valor, é possível realizar um comparativo entre a complexidade e tempo gasto para cada um deles. Tomando como base as instâncias dos extremos, pode-se observar que a instância *binpack4* é 67,91 mais complexa que a instância *binpack1*, e o tempo gasto pela solução na instância *binpack4* é 71,11 maior que a instância *binpack1*. Segundo a documentação oficial do *framework*, problemas com pontuação acima de 1.000 são considerados de complexidade alta [4].

Também é possível observar que o problema que levou mais tempo para ser solucionado foi o problema 3 da instância *binpack4*, com 4,11 minutos. Já o problema que foi solucionado mais rápido, foi o problema 8 da instância *binpack1*, com 0,259 segundos.

Para solucionar as 4 instâncias, usando o algoritmo de busca local *Simulated Annealing*, foram gastos no total 33,15 minutos. Uma média de 24,86 segundos para cada problema. Embora haja diferenças em termos de ambiente computacional e solução adotada, no trabalho "*A Hybrid Grouping Genetic Algorithm for Bin Packing*" de Falkenauer, por exemplo, para solucionar os mesmos conjuntos de dados, foram necessários 163 minutos. Uma média de 2,03 minutos por problema [6]. Neste caso, não há intenção de realizar uma

Table 4: Pontuação x tempo gasto entre as instâncias com *Simulated Annealing*

S. Annealing	*binpack1* (5.898)		*binpack2* (25.425)		*binpack3* (100.600)		*binpack4* (400.550)	
Problema	**Pontuação**	**Tempo**	**Pontuação**	**Tempo**	**Pontuação**	**Tempo**	**Pontuação**	**Tempo**
0	0hard/-122soft	1.727ms	0hard/-67soft	4.333ms	0hard/-63soft	21.305ms	0hard/-86soft	99.783ms
1	0hard/-145soft	373ms	0hard/-146soft	2.014ms	0hard/-23soft	18.985ms	0hard/-112soft	48.681ms
2	0hard/-106soft	3.097ms	0hard/-87soft	2.712ms	0hard/-84soft	13.248ms	0hard/-120soft	63.335ms
3	0hard/-65soft	2.400ms	0hard/-86soft	2.156ms	0hard/-28soft	50.126ms	0hard/-20soft	247.178ms
4	0hard/-146soft	300ms	0hard/-58soft	3.362ms	0hard/-133soft	7.502ms	0hard/-39soft	80.376ms
5	0hard/-78soft	418ms	0hard/-26soft	38.477ms	0hard/-137soft	6.220ms	0hard/-76soft	33.463ms
6	0hard/-63soft	831ms	0hard/-146soft	559ms	0hard/-17soft	74.653ms	0hard/-119soft	31.989ms
7	0hard/-55soft	611ms	0hard/-182soft	898ms	0hard/-3soft	102.276ms	0hard/-126soft	37.886ms
8	0hard/-172soft	259ms	0hard/-13soft	58.468ms	0hard/-48soft	13.443ms	0hard/-85soft	47.031ms
9	0hard/-30soft	1.698ms	0hard/-120soft	1.738ms	0hard/-141soft	5.958ms	0hard/-11soft	140.630ms
10	0hard/-120soft	642ms	0hard/-94soft	2.543ms	0hard/-140soft	5.741ms	0hard/-99soft	24.385ms
11	0hard/-103soft	311ms	0hard/-43soft	4.991ms	0hard/-86soft	8.544ms	0hard/-72soft	74.490ms
12	0hard/-20soft	2.473ms	0hard/-161soft	1.573ms	0hard/-57soft	10.499ms	0hard/-114soft	27.117ms
13	0hard/-148soft	464ms	0hard/-156soft	2.534ms	0hard/-62soft	20.385ms	0hard/-109soft	38.244ms
14	0hard/-127soft	585ms	0hard/-125soft	1.223ms	0hard/-146soft	7.447ms	0hard/-17soft	93.856ms
15	0hard/-98soft	969ms	0hard/-28soft	25.857ms	0hard/-130soft	9.990ms	0hard/-29soft	125.398ms
16	0hard/-112soft	488ms	0hard/-73soft	1.789ms	0hard/-149soft	4.931ms	0hard/-146soft	29.910ms
17	0hard/-97soft	1.206ms	0hard/-125soft	922ms	0hard/-86soft	8.020ms	0hard/-30soft	80.930ms
18	0hard/-95soft	548ms	0hard/-45soft	3.187ms	0hard/-106soft	7.823ms	0hard/-121soft	34.431ms
19	0hard/-178soft	310ms	0hard/-96soft	2.454ms	0hard/-55soft	9.108ms	0hard/-100soft	42.527ms
Média	0hard/-104soft	985ms	0hard/-94soft	8.089ms	0hard/-85soft	20.310ms	0hard/-82soft	70.082ms
Total	0hard/-2080soft	19.710ms	0hard/-1877soft	161.780ms	0hard/-1694soft	406.200ms	0hard/-1631soft	1.401.640ms

comparação entre os trabalhos, apenas evidenciar a capacidade e potencial do uso do *framework* OptaPlanner.

5. CONCLUSÕES

O objetivo principal do presente trabalho foi explorar o *framework* OptaPlanner aplicado ao problema de empacotamento unidimensional, bem como mensurar a performance e comparar os resultados obtidos entre si.

Levando em consideração esses aspectos, foi possível constatar a viabilidade do uso do *framework* OptaPlanner para problemas com grande volume de dados. Além do mais, vencida a etapa de compreensão do funcionamento do *framework*, destaca-se a simplicidade da solução desenvolvida. Para o problema de empacotamento unidimensional, a modelagem de domínio necessitou utilizar apenas 3 classes, fazendo uso das anotações disponibilizadas pelo *framework*. Além disso, foram utilizadas 3 regras de restrições para limitar o espaço de busca, sendo 2 rígidas e 1 suave. O restante do esforço foi direcionado para configuração. Há de se ressaltar a necessidade de algum conhecimento prévio na linguagem Java e Drools.

Por tratar de um nicho específico de problemas e não ser uma ferramenta difundida no mercado, ainda não possui uma grande comunidade para suporte, embora haja uma documentação detalhada que favorece o entendimento e auxilia na implementação de novas soluções. Também há um fórum mantido pelos responsáveis do projeto com esse propósito.

Apesar de terem sido utilizados 3 algoritmos de busca local, além das configurações de *benchmark* do *framework*, não foi possível explorar todos os algoritmos e configurações existentes. Outro item possível para ser considerado em trabalhos futuros é a realização de estudos comparativos com outros *frameworks* existentes no mercado.

Durante a pesquisa foi possível observar que, apesar da existência de diversos algoritmos e possibilidades de configuração, muitos outros algoritmos estudados pela comunidade científica ainda não são suportados pela ferramenta. O algoritmo de otimização colônia de formigas e o *variable neighborhood search* são exemplos desses algoritmos.

Como já exposto em momento anterior, conclui-se que o *framework* OptaPlanner é uma ferramenta que se mostrou eficaz para o problema de empacotamento unidimensional, tanto no aspecto dos resultados obtidos quanto no tempo computacional gasto para alcançá-los.

6. REFERENCES

[1] J. E. Beasley. OR-library: Distributing test problems by electronic mail. *Journal of the Operational Research Society*, 41(11):1069–1072, Nov. 1990.

[2] R. J. Booch, G. and I. Jacobson. *UML: guia do usuário*. Elsevier Brasil, 2006.

[3] N. G. Bourbakis. *Artificial Intelligence Methods And Applications*. Springer International Publishing, 2014.

[4] G. De Smet. Optaplanner user guide. In *Red Hat, Inc. or third-party contributors. OptaPlanner is an open source constraint solver in Java.*, 2006.

[5] M. Dell'Amico, F. Furini, and M. Iori. A branch-and-price algorithm for the temporal bin packing problem. In *. arXiv:1902.04925 [math]*, feb 2019.

[6] E. Falkenauer. A hybrid grouping genetic algorithm for bin packing. *Journal of Heuristics*, 2(1):5–30, 1996.

[7] M. R. Garey, R. L. Graham, and J. D. Ullman. Worst-case analysis of memory allocation algorithms. In *Proceedings of the fourth annual ACM symposium*

on Theory of computing - STOC 72, pages 143–150. ACM, ACM Press, 1972.

[8] M. R. Garey and D. S. Johnson. *Computers and intractability : a guide to the theory of NP-completeness*. New York [u.a] Freeman [ca. 2009], 2009.

[9] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers & operations research*, 13(5):533–549, 1986.

[10] R. Hat. Kie group - knowledge is everything. In *http://kiegroup.org/*, 2019.

[11] D. S. Johnson. *Near-optimal bin packing algorithms*. PhD thesis, Massachusetts Institute of Technology, 1973.

[12] G. Kłosowski, E. Kozłowski, and A. Gola. Integer linear programming in optimization of waste after cutting in the furniture manufacturing. In *Advances in Intelligent Systems and Computing*, pages 260–270. Springer International Publishing, Aug. 2017.

[13] S. Kosecka-Żurek et al. Application of it tools in optimization of logistics problems. *Czasopismo Techniczne*, 1:177–186, 2019.

[14] B. M. Macik. Case management task assignment using optaplanner. Master's thesis, Masaryk University Faculty of Informatics, 2016.

[15] Á. L. Murciego, G. V. González, A. L. Barriuso, D. H. de La Iglesia, and J. R. Herrero. Multi agent gathering waste system. In *DCAI 2015*, 2015.

[16] C. Paquay, S. Limbourg, and M. Schyns. A tailored two-phase constructive heuristic for the three-dimensional multiple bin size bin packing problem with transportation constraints. *European Journal of Operational Research*, 267(1):52–64, May 2018.

[17] R. I. Phelps. Artificial intelligence—an overview of similarities with o.r. *Journal of the Operational Research Society*, 37(1):13–20, Jan. 1986.

[18] L. F. Santos, H. T. Yoshizaki, and C. B. Cunha. Variable neighborhood search for the bin packing problem with compatible categories. *arXiv preprint arXiv:1905.03427*, 2019.

[19] A. Wolke, B. Tsend-Ayush, C. Pfeiffer, and M. Bichler. More than bin packing: Dynamic resource allocation strategies in cloud data centers. *Information Systems*, 52:83–95, 2015.

[20] M. ČIMBORA. Usability improvements of optaplanner benchmarker. Master's thesis, Masaryk University, Faculty of Informatics, Brno, 2015.