Relatório Trabalho Prático II - Problema de Empacotamento com Conflitos (*bin packing problem*)

Hailton Fernando de Carvalho
Universidade Federal de Ouro Preto

Relatório técnico do segundo Trabalho Prático da disciplina Projeto e Análise de Algoritmos, do Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Ouro Preto

Relatório Trabalho Prático II - Problema de Empacotamento com Conflitos (*bin packing problem*)

Resumo

O problema de empacotamento com conflitos descreve-se como segue: dado um conjunto de item, em que cada componente do conjunto tenha um peso e uma lista de itens os quais não podem ser alocados no mesmo recipiente, o objetivo é minimizar o número de recipientes necessários de modo que todos os itens sejam empacotados. Cada recipiente possui uma capacidade máxima de peso, sendo assim, o somatório do peso de todos os itens alocados a ele não deve excedê-la. Existem muitas variações deste problema, e as restrições variam com a aplicação. Neste trabalho foi feito um estudo sobre as algoritmos capazes de oferecer uma solução factível de boa qualidade. Foram desenvolvidas Heurísticas Construtivas e algoritmos baseados em Metaheurísticas conhecidas, como o GRASP. Os resultados obtidos demonstram que as abordagens foram efetivas para o tratamento do problema.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1.

Introdução

Este trabalho apresenta algoritmos para resolução do Problema de Empacotamento com conflitos (*bin packing problem with conflicts(BPPC)*, dado um conjunto $V = 1, 2, ..., n$ de itens, cada item i possui um peso não negativo w_i, e um conjunto infinito de contêineres idênticos, (*bins*) de capacidade C. De outro modo, de acordo com [1], podemos definir o problema como: dado um grafo de conflito $G = (V, E)$, em que E é um conjunto de arestas tal que $(i, j) \in$ E quando o item i e j são conflitantes. Também foi observado por Muritiba o BPCC é importante porque existem um grande número de problemas com aplicações reais no cotidiano, devido a generelização de muitos problemas de otimização combinatória. Algumas destas aplicações incluem escalonamento de consultas como dito por [3], atribuição de processos à processadores, e balanceamento de tarefas para computação paralela [4].

A formulação matemática do problema se dá pelo conjunto de equações:

Em que (1) é função objetivo, onde busca-se minimizar a quantidade de contêineres, (2) restrição de que cada item deve ser alocado a um contêiner, (3) é a restrição de capacidade do contêiner, e as restrições seguintes são clássicas do problema de empacotamento [4].

A definição mais simples para Limites Inferiores para o problema de empacotamento com conflitos, de acordo com a literatura, consiste na soma do peso de todos os itens, e divide pela capacidade máxima c do contêiner. O próximo inteiro maior que este valor será o limite inferior do problema. Esta definição foi utilizada neste trabalho para cálculo do limite inferior, como pode ser observado abaixo:

$$LB = \lceil \sum_{i=1}^{n} \frac{w_i}{C} \rceil$$

$$\min \sum_{h=1}^{n} y_h \tag{1}$$

$$\sum_{h=1}^{n} x_{ih} = 1 \quad i = 1, 2, \ldots, n, \tag{2}$$

$$\sum_{i=1}^{n} w_i x_{ih} \leq C y_h \quad h = 1, 2, \ldots, n, \tag{3}$$

$$x_{ih} + x_{jh} \leq y_h \quad (i, j) \in E,\ h = 1, 2, \ldots, n, \tag{4}$$

$$y_h \in \{0, 1\} \quad h = 1, 2, \ldots, n, \tag{5}$$

$$x_{ih} \in \{0, 1\} \quad i = 1, 2, \ldots, n,\ h = 1, 2, \ldots, n. \tag{6}$$

Figura 1.1.: Formulação *bin packing* [4]

1.1. Enunciado

Neste trabalho desenvolvido por [5] o aluno deve implementar algoritmos para a produção de soluções factíveis, de alta qualidade, para um problema NP-Difícil. O aluno deve implementar vários algoritmos com o intuito de conseguir melhores resultados possíveis, ou seja, a produção de soluções de alta qualidade (a solução ótima ou soluções com custo próximo do ótimo) em tempos computacionais restritos. Entre as técnicas vistas em aula podem ser utilizadas então:

- *branch & bound* com tempo limitado
- algoritmos construtivos e de busca local
- metaheurísticas
  - algoritmos evolutivos
  - algoritmos de reinício
  - *simulated annealing*

1.1.1. Problema

O problema é o problema de empacotamento com conflitos (*bin packing*). Considere os seguintes dados de entrada:

I conjunto de itens $1, ..., n$
$c \in \mathbb{Z}^+$ capacidade de contêiner (*bin*), considera-se que o número de contêineres é ilimitado
$w_i \in \mathbb{Z}^+$ peso do item i
F_i conjunto de item compatível com o item i.
Uma solução é um agrupamento de itens por contêiner, usando o menor número possível de contêineres, sendo que cada contêiner deve ter sua capacidade c respeitada, dois itens incompatíveis não podem ficar no mesmo contêiner. Formalmente, procura-se o menor k tal que exista uma solução S com k conjuntos disjuntos $B_1, ..., B_k : B_1 \cup B_2 \cup, ..., B_k = I$, respeitando as seguintes restrições:

capacidade contêiner: o peso dos itens incluídos em cada contêiner não deve ultrapassar a capacidade c do contêiner, ou seja: $\sum_{i \in B_j}^{k} w_i \leq c \forall j \in 1, ..., k$;

conflitos: cada contêiner somente pode incluir itens não conflitantes, ou seja: se para dois itens i_1 e i_2 temos que $i_2 \in F_{i1}$ ou $i_1 \in F_{i2}$ (se i_2 *in* F_{i2}), então esses dois itens não podem ser alocados no mesmo contêiner.

1.1.2. Instancias

50 problemas teste foram disponibilizados no seguinte endereço:http://www.decom.ufop.br/haroldo/paa/bpinstances.tar.gz.

Os problemas teste estão no seguinte formato: na primeira linha temos o número de itens e a capacidade de cada contêiner. Em cada linha seguinte, para cada item, temos o peso do mesmo e quais itens que o mesmo conflita. Importante: para não desperdiar espaço no arquivo o conflito entre os itens 2 e 3 aparece uma vez só, ou seja, o 3 aparece como conflitante do 2. A implementação de cada um deve garantir que ao consultar sobre conflito a pesquisa $(2, 3)$ e $(3, 2)$ retorne verdadeiro nesse caso. Considere que o núemro de conflito do i-ésimo item é f_i e que a_{ij} indica o j-ésimo conflito do item i. O formato do arquivo é:

		n	c
1	w_1	a_{11},a_{12},..., a_{if1}	
2	w_2	a_{11},a_{12},..., a_{if2}	
		...	
n	w_n	a_{n1},a_{n2},..., a_{nfn}	

1.1.3. Solucões

A solução deve ser salva em um arquivo de texto no seguinte formato. Considere que foram usados k contêineres na solução e cada contêiner tem um número de itens especificado em $m_1, ..., m_k$ e que a_{ji} indica o j-ésimo item do contêiner j. O formato de saída então é:

	k
m_1	a_{11},a_{12},..., a_{if1}
m_2	a_{11},a_{12},..., a_{if2}
	...
m_k	a_{k1},a_{k2},..., a_{kmk}

1.1.4. Entrada e Saída

O programa deve receber argumentos no seguinte formato:
`programa` `arquivoInstancia` `arquivoSolucao` `maxSeconds`[parametros de configuração do método].

Onde programa é o nome do executável, `arquivoInstancia` e `arquivoSolucao` são o nome do arquivo de entrada e saída, respectivamente. `maxSeconds` é um parâmetro que o programa deve receber indicando o tempo limite de execução em segundos. Parâmetros adicionais devem ser incluídos para configurações adicionais do programa, configurando por exemplo o algoritmo utilizado e seus parâmetros.

1.1.5. Material a ser entregue

- Código fonte: Qualquer linguagem de programação pode ser utilizada, mas o código fonte deve ser organizado e comentado. (O Código e a documentação estão disponíveis neste link: https://github.com/hailtoncarvalho/TP2)
- Relatório técnico descrevendo o problema, métodos de solução implementados, resultados detalhados de experimentos computacionais (tabelas e gráficos) e conclusões.

Capítulo 2.

Desenvolvimento

Neste capítulo serão definidos os conceitos base para o desenvolvimento da abordagem adotada, e as dificuldades enfrentadas durante desenvolvimento. O trabalho foi desenvolvido com a Linguagem Java, e a documentação do código pode ser acessada em https://github.com/hailtoncarvalho/TP2.

2.1. Algoritmos

Como foi citado na seção de Introdução, o problema de empacotamento com conflito se enquadra na classe dos NP-difíceis, desse modo, ainda não existem algoritmos que os resolva em tempo polinomial. [2].

Problemas da classe NP-Completos são conhecidos pela dificuldade em resolver de modo exato em razão da quantidade possível de soluções enumeráveis. Dado que a enumeração de soluções é o número total de soluções que satisfazem o problema, para provar que uma solução é ótima, é necessário verificar todas que todas as outras possuam valor de utilidade inferior. À medida em que aumenta o número de itens em uma instância, eleva-se exponencialmente a quantidade de soluções possíveis.

Entretanto, em aplicações reais dificilmente é possível analisar todo o espaço de soluções de um problema de otimização combinatória, sendo necessário utilizar métodos que reduzam a quantidade de soluções analisadas, ou que produzam soluções satisfazíveis, não necessariamente ótimas.

De acordo com [6] é possível dar uma certa "inteligência"ao método de enumeração, aplicando técnicas como *brancha-and-bound* ou *brancha-and-cut*, de forma a reduzir o

número de soluções a analisar no espaço de soluções. Através disto, é possível resolver problemas com dimensões mais elevadas. Contudo, dada a natureza combinatória do problema, pode ser que, na pior das hipóteses, todas as soluções precisem ser verificadas. Isto impede que seja utilizado apenas métodos exatos, dado o tempo proibitivo de se encontrar a melhor solução ([6].

Portanto, em problemas desta natureza, o uso de métodos exatos se torna bastante restrito, e há a necessidade de dedicar esforços na pesquisa por aplicação de heurísticas eficientes para solucionar problemas deste nível de complexidade.

O desafio é, portanto, produzir em tempo computacional relativamente baixo, soluções tão próximas quanto possível da solução ótima. Muitos esforços têm sido desenvolvidos neste sentido, porém a maioria das heurísticas são específicas para um problema, e não é necessariamente eficiente ou aplicável a uma classe mais abrangente de problemas [6].

Do esforço em produzir soluções computacionais satisfatórias, as metaheurísticas surgiram da reunião de conceitos das áreas de Otimização e Inteligência Artificial, que tornou possível a obtenção de solução através de métodos "inteligentemente flexíveis" [6].

De acordo com [6], dentre os procedimentos que se enquadram como metaherísticas, destacam-se: Algoritmos Genéticos, Reds Neurais, *Simulated Annealing*, Busca Tabu, GRASP (*Greed and Randomized Adaptative Search Procedure*), VNS, Colônia de Formigas, etc..

Conforme será visualizado na próxima seção, neste trabalho foram desenvolvidos algoritmos Heurísticos para construção de uma solução, e metaheurística GRASP.

2.1.1. Heurísticas Construtivas

Uma Heurística Construtiva objetiva construir uma solução, elemento a elemento. Nas heurísticas mais conhecidas, os elementos candidatos são geralmente ordenados segundo uma função gulosa, que estima o benefício de inserção de cada elemento, e somente o "melhor"elemento é inserido a cada passo. O pseudocódigo da figura mostra o funcionamento básico de uma Heurística Construtiva:

O problema utiliza uma função gulosa $g(.)$. Na figura, t_{melhor} indica o membro do conjunto de elementos candidatos com o valor mais favorável da função de avaliação g.

procedimento *ConstrucaoGulosa*$(g(.), s)$;
1 $s \leftarrow \emptyset$;
2 Inicialize o conjunto C de elementos candidatos;
3 enquanto $(C \neq \emptyset)$ faça
4 $\quad g(t_{melhor}) = melhor\{g(t) \mid t \in C\}$;
5 $\quad s \leftarrow s \cup \{t_{melhor}\}$;
6 $\quad$ Atualize o conjunto C de elementos candidatos;
7 fim-enquanto;
8 Retorne s;
fim *ConstrucaoGulosa*;

Figura 2.1.: Heurística Construtiva
[6]

Um outro modo comum de obter uma solução inicial é escolher aleatoriamente entre elementos candidatos. Um ponto positivo desta abordagem é a facilidade de implementação. Abaixo pode ser visualizado um pseudocódigo de uma Heurística Construtiva Aleatória.

procedimento *ConstrucaoAleatoria*$(g(.), s)$;
1 $s \leftarrow \emptyset$;
2 Inicialize o conjunto C de elementos candidatos;
3 enquanto $(C \neq \emptyset)$ faça
4 $\quad$ Escolha aleatoriamente $t_{escolhido} \in C$;
5 $\quad s \leftarrow s \cup \{t_{escolhido}\}$;
6 $\quad$ Atualize o conjunto C de elementos candidatos;
7 fim-enquanto;
8 Retorne s;
fim *ConstrucaoAleatoria*;

Figura 2.2.: Heurística Construtiva Aleatória
[6]

A grande desvantagem desta abordagem é a baixa qualidade das soluções que podem ser geradas, e consequentemente, para buscar uma solução final melhor, pode ser necessário grande esforço computacional.

2.1.2. Heurísticas de Refinamento

De acordo com [6], as Heurísticas de Refinamento em problemas de otimização, também conhecidos como técnicas de busca local, representa um conjunto de técnicas baseadas na noção de vizinhança. Seja S o espaço de pesquisa de um problema de otimização e f a função objetivo a minimizar. A função N (*Neighbohood*), associa a cada solução de $s \in S$, sua vizinhança $N(S) \subseteq S$. Cada solução $s' \in N(s)$ é chamada de vizinho de s, que esteja em sua vizinhança.

De modo resumido, este tipo de heurística parte de uma solução inicial qualquer, e se modifica de iteração a iteração a partir da vizinhança adotada.

Método da Descida/Subida(*Descent/Uphill Method*)

A principal estrutura por trás do método da Descida/Subida consiste na análise de todos os possíveis vizinhos da solução inicial, promovendo movimentos apenas se este representar uma melhora no valor atual da função de avaliação. Em razão de toda vizinhança escolher o melhor candidato, esta técnica é reconhecida na literatura como *Best Improvement Method*. O método é executado até que um ótimo local seja encontrado.

Abaixo pode ser conferido um pseudocódigo da Heurística:

procedimento *Descida*$(f(.), N(.), s)$;
1 $V = \{s' \in N(s) \mid f(s') < f(s)\}$;
2 enquanto $(|V| > 0)$ faça
3 Selecione $s' \in V$, onde $s' = \arg\min\{f(s') \mid s' \in V\}$;
4 $s \leftarrow s'$;
5 $V = \{s' \in N(s) \mid f(s') < f(s)\}$;
6 fim-enquanto;
7 Retorne s;
fim *Descida*;

Figura 2.3.: Heurística de Refinamento Descida/Subida
[6]

Método da Descida/Subida Randômica

Enquanto o método da Descida/Subida explora toda a vizinhaça, o método da Descida/Subida Randômica (*Random Descent/Uphill Method*) considera analisar um vizinho qualquer, e aceita o movimento somente se ele resultar em um ganho, de acordo com a função objetivo.

```
procedimento DescidaRandomica(f(.), N(.), IterMax, s);
1  Iter ← 0;      {Contador de iterações sem melhora }
2  enquanto (Iter < IterMax) faça
3      Iter ← Iter + 1;
4      Selecione aleatoriamente s' ∈ N(s);
5      se (f(s') < f(s)) então
6          Iter ← 0;
7          s ← s' ;
8      fim-se;
9  fim-enquanto;
10 Retorne s;
fim DescidaRandomica;
```

Figura 2.4.: Descida/Subida Randômica
[6]

2.1.3. Metaheurística GRASP

O algoritmo desenvolvido em alternativa às Heurísticas puramente construtivas, foi baseado metaheurística GRASP (Procedimento de busca adaptativa gulosa e aleatória), que foi proposta em Feo e Resende (1995). A ideia principal do GRASP consiste em uma fase construtiva, e uma fase de busca local. Na fase construtiva, a solução é construída elemento por elemento adicionado randomicamente a partir de uma Lista Restrita de Candidatos (LRC). Na fase de busca local, são realizados movimentos na solução construtiva a fim de encontrar soluções melhores.

Como parâmetro, o GRASP recebe um número de iterações, e quando este número for atingido, o algoritmo retorna a melhor solução encontrada. De acordo com [6], na literatura é demonstrado que a variação do parâmetro α, que determina a aleatoriedade da escolha de novos elementos, pode influenciar na qualidade da solução gerada, de acordo com o problema. Em variações mais sofisticadas do GRASP, é comum

a utilização de múltiplos α's Sendo assim, neste trabalho o parâmetro α também é variável, e foi feito um estudo sobre o impacto da variação deste parâmetro nas soluções geradas.

2.2. Algoritmos Implementados

Nesta seção serão apresentados os algoritmos que foram implementados. Foram desenvolvidos algoritmos de Heurística Construtiva, que possui variação na função de avaliação e modificação no método de escolha dos candidatos. Foram realizados diversos testes até a escolha do método final, que consiste em uma metaheurística com fase de busca local híbrida, e refinamento também híbrido. Todos os algoritmos Heurísticos foram baseados na estratégia *First Fit*, ou seja, o item será alocado no primeiro contêiner que o permita ser alocado sem inflingir as restrições do problema.

Tabela 2.1.: Tipos de Algoritmos

Tabela de Algoritmos			
Algoritmo	Tipo de Heurística	Parâmetros	Retornos
binPGRandom	Construtiva	Instância	Solução Aleatória
binPGSorted	Construtiva	Instância	Solução Gulosa
binPGSReverse	Construtiva	Instância	Solução Gulosa
binPHAscending	Construtiva	Instância	Solução Híbrida
binPHDecreasing	Construtiva	Instância	Solução Híbrida
binPHF	Construtiva	Instância	Solução Híbrida
binPHFTRandom	Construtiva	Instância	Solução Híbrida
binPGrasp	Metaheurística	Instância,α	Solução Híbrida Refinada
binPGraspLRC	Metaheurística	Instância α	Solução Híbrida Refinada
binPLS	Refinamento	Solucão	Solução Refinada

2.2.1. bPGRandom -*Bin Packing Random*

Este algoritmo gera uma solução de modo completamente aleatório, sem nenhuma função de avaliação. Uma função determina aleatoriamente, entre os itens não alo-

cados, qual será o próximo item a ser inserido na solução s. Seja i o item sorteado, ele é alocado no primeiro contêiner que disponha de capacidade remanescente suficientemente grande para comportá-lo, e que nenhum item alocado anteriormente no contêiner esteja na lista de conflitos. Caso o item i não possa ser alocado, é incluído um novo contêiner.

2.2.2. bPGSorted -*Bin Packing Sorted*

Este algoritmo gera uma solução de modo guloso, cuja função determina que os itens inseridos primeiro devam ser os mais leves. A lista de itens é ordenada de modo ascendente, através do parâmetro peso, de cada item $i \in I$.

2.2.3. bPGSReverse -*Bin Packing Sorted Reverse*

A Heurística deste algoritmo também é gulosa, entretanto a ordenação dos itens pelo atributo peso é inversa. Sendo assim, os itens mais pesados são alocados com prioridade. A ideia do desenvolvimento desta heurística parte do pressuposto de que se os itens mais pesados forem alocados nos primeiros contêineres, o espaço remanescente deve ser suficiente para alocar uma quantidade maior de itens mais leves, fazendo com que a restrição de capacidade do contêiner não seja violada a menos que seja pela compatibilidade dos itens.

2.2.4. bPHAscending -*Bin Packing Half Ascending*

Neste algoritmo, instância é ordenada de modo crescente, e o item i adicionado no contêiner k deve ser selecionado por um sorteio correspondente à metade dos itens a serem alocados nos contêineres. Enquanto houver espaço e não conflitar com os demais itens já alocados nele, itens podem ser adicionados.

2.2.5. binPHDecreasing- *Bin Packing Half Decreasing*

Seguindo a ideia do algoritmo anterior, instância é ordenada de modo decrescente, e o item i adicionado no contêiner k deve ser selecionado por um sorteio correspondente

à metade dos itens a serem alocados nos contêineres. Enquanto houver espaço e não conflitar com os demais itens já alocados nele, itens podem ser adicionados.

2.2.6. binPHF - *Bin Packing Heavy First*

Algoritmo Guloso ordenado por ordem decrescente com prioridade: A instância é ordenada de forma decrescente, e os itens mais pesados são adicionados nos primeiros $\frac{n}{2}$ contêineres, e o item subsequente é adicionado no primeiro contêiner que couber, enquanto houver espaço e não conflitar com os demais itens já alocados no contêiner k_i.

2.2.7. binPHFTRandom - *Bin Packing Heavy First Then Random*

Baseado no algoritmo anterior, a diferença desta Heurística é que ao verso de os $\frac{n}{2}$ itens mais leves serem adicionados de modo guloso, ele são alocados de modo aleatório.

2.2.8. binPGrasp - *Bin Packing Grasp*

Esta Heurística é baseada na fase construtiva da metaheurística GRASP (*Greed Adaptative Search Procedure*), proposta por Feo e Resende (1995). Os componentes básicos desta heurística consiste em um parâmetro $\alpha \in [0, 1]$, e uma lista de candidatos, nomeada no método como Lista Restrita de Candidatos (LRC). A Lista de Candidatos contém todos os elementos que podem ser alocados à solução, ordenados com base em uma função $f(.)$ que gera um benefício imediato. Neste contexto, a LRC foi construída com base em ordenação pelo menor peso. O parâmetro α restringe a quantidade de itens que serão sorteados quando $\alpha \rightarrow 0$, as soluções tendem a ser mais gulosas. Quando $\alpha \rightarrow 1$, as soluções tendem a um maior grau de aleatoriedade. Como pode existir benefícios nas duas abordagens, o GRASP fornece um bom conjunto de soluções, entretanto é necessário utilizar Heurísticas de Refinamento.

2.2.9. binPLS - *Bin Packing Local Search*

Este método é baseado em uma Heurística de Refinamento Descida/Subida Randômica. Conforme mencionado em seções anteriores, o método aceita apenas movimentos de melhora, possui caráter aleatório, e não garante ótimo local.

Capítulo 3.

Experimentos Computacionais e Análise dos Resultados

Nesta seção serão expostos os resultados obtidos com a aplicação das Heurísticas desenvolvidas.

O experimento foi realizado a partir de vários conjuntos de instâncias. O teste de cada instância dos conjuntos foi executado por parâmetros que variam de acordo com o método utilizado. Os testes possuem caráter qualitativo e quantitativo, em que os valores considerados de boa qualidade são os próximos ao Limite Inferior de cada instância testada, e a quantidade consiste em um intervalo de tempo em que o algoritmo foi executado para chegar à solução descrita.

Para os métodos sem nenhum fator de aleatoriedade, como é o caso das Heurísticas Gulosas, os algoritmos foram executados apenas uma vez, pois não há nenhuma chance de a solução ser diferente, dado que a ordem de entrada e escolha dos itens não são alteradas.

Apenas para os métodos com alguma natureza de aleatoriedade, foram executados os testes quantitativos e qualitativos. Estes algoritmos serão comparados com a Metaheurística desenvolvida.

Os testes foram feitos no Sistema Operacional Windows 7 Ultimate 64 bits, Processador Intel(R) Core(TM) i3-2350M CPU @2.30GHz 2.30GHz, 4Gigabytes.

3.1. Algoritmos Heurísticos Construtivos

Nesta seção, serão expostos os resultados obtidos com a execução dos algoritmos gulosos. Foram implementadas duas versões básicas deste tipo de Heurística: Guloso Aleatório Crescente, que ordena todos os itens de modo crescente, a partir do menor peso; e o Guloso Aleatório Decrescente, que ordena de forma inversa.

Considerando que os itens possuem restrição de capacidade, poderia ser que a ordem de ordenação afetasse na forma como cada item fosse distribuído, gerando assim, soluções diferentes, visto que o item é inserido no primeiro contêiner possível. Entretanto, os resultados obtidos ficaram bem próximos. Conforme pode ser observado na tabela abaixo. Vale ressaltar que as soluções apenas foram geradas, e não passaram por nenhum algoritmo de refinamento.

	Heurísticas Construtivas								
Instancia	**bPS**	**bPR**	**bPHA**	**bPHD**	**bPHR**	**bPHF**	**bPHR**	**bPHFTR**	**Grasp**
d-BPWC_2_2_5.txt	58	59	59	60	59	60	58	59	59
d-BPWC_2_3_1.txt	89	89	89	90	88	90	89	88	89
d-BPWC_3_6_2.txt	308	308	309	310	309	310	309	309	309
d-BPWC_2_8_5.txt	202	203	202	201	202	201	201	202	201
d-BPWC_3_6_9.txt	292	292	292	292	293	292	292	293	293
d-BPWC_3_5_3.txt	248	248	248	248	248	248	247	248	247
d-BPWC_3_6_6.txt	304	306	305	306	304	306	304	304	305
d-BPWC_3_7_7.txt	359	359	358	358	359	358	359	359	358
d-BPWC_4_5_6.txt	504	507	504	505	505	505	505	505	504
d-BPWC_4_2_5.txt	180	207	180	192	182	192	184	182	181

3.2. Metaheurística GRASP

O primeiro teste consiste em na análise da qualidade da solução. Foi escolhido um conjunto de instâncias como amostra do estudo (GRASP, 10s, α aleatório). Neste teste foram consideradas o UB =Limite Superior e LB = Limite Inferior, a distância o quão

a Solução obtida está do Limite Inferior. O cálculo da distância é dado pela fórmula: $\frac{UB-LB}{UB}$.

Tabela 3.1.: Tabela de Distância da Solução e Limite

Teste Qualitativo			
Algoritmo	**LB**	**UB**	**Distância**
d-BPWC_2_2_5.txt	39	58	0,33
d-BPWC_2_3_1.txt	38	88	0,57
d-BPWC_3_6_2.txt	77	308	0,75
d-BPWC_2_8_5.txt	39	201	0,81
d-BPWC_3_6_9.txt	77	292	0,74
d-BPWC_3_5_3.txt	76	247	0,69
d-BPWC_3_6_6.txt	73	304	0,76
d-BPWC_3_7_7.txt	76	358	0,79
d-BPWC_4_5_6.txt	151	504	0,70
d-BPWC_4_2_5.txt	149	179	0,17

Como o cálculo do limite inferior é feito relaxando a restrição de conflito dos itens, o cálculo da distância não se mostrou eficiente. Entretanto, comparando os resultados obtidos com estas mesma Instancia, fornecidas pelo professor, as soluções estão iguais, próximas ou melhores.

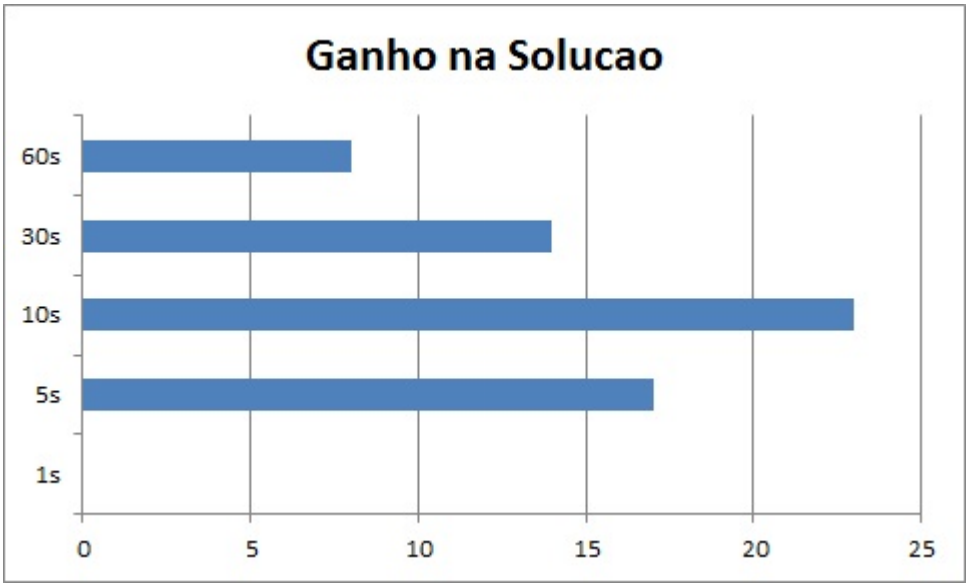
A tabela a seguir mostra a partir deu conjunto de instâncias, a variação dos resultados do Limite Superior, mesmo após ao submetimento da solução à uma Heurística de Refinamento. Neste primeiro experimento coma metaheurística, o algoritmo foi testado sem fixação do parâmetro α, e a quantidade de vezes que o algoritmo executou consistiu em um intervalo de tempo, que variou de 1 a 60 segundos, conforme tabela.

Tabela 3.2.: α aleatório, tempo de tempo incremental

GRASP - α Aleatório					
Instancia	1s	5s	10s	30s	60s
d-BPWC_2_2_5.txt	58	58	58	58	58
d-BPWC_2_3_1.txt	89	88	88	88	88
d-BPWC_3_6_2.txt	309	309	308	308	308
d-BPWC_2_8_5.txt	201	201	201	201	201
d-BPWC_3_6_9.txt	292	292	292	292	292
d-BPWC_3_5_3.txt	247	248	247	247	247
d-BPWC_3_6_6.txt	305	304	304	304	305
d-BPWC_3_7_7.txt	359	359	358	358	358
d-BPWC_4_5_6.txt	505	506	504	504	505
d-BPWC_4_2_5.txt	180	184	179	181	180

Mesmo com baixo tempo de processamento o algoritmo é capaz de gerar boas soluções. Após observar o comportamento do algoritmo, foi realizado uma verificação quanto a variação das soluções, e a partir da observação do somatório da solução ótima obitida de todas as instâncias, noutou-se que a partir de 30s os ganhos com o incremento do tempo de execução do algoritmo são cada vez menores. A relação de ganho por aumento de tempo pode ser melhor visualizada no gráfico abaixo:

Figura 3.1.: Relação de melhoria por tempo de execução



Sendo assim, o teste foi realizado novamente, considerando a fixação do parâmetro alfa, todos executados por 30 segundos. O resultado pode ser conferido na tabela a seguir.

Tabela 3.3.: α fixado, 30s de execução.

GRASP - α Aleatório						
Instancia	$\boldsymbol{\alpha \to 0.1}$	$\boldsymbol{\alpha \to 0.3}$	$\boldsymbol{\alpha \to 0.5}$	$\boldsymbol{\alpha \to 0.7}$	$\alpha \to 0.9$	$\boldsymbol{\alpha \to 1}$
d-BPWC_2_2_5.txt	58	58	58	58	58	58
d-BPWC_2_3_1.txt	88	88	88	89	89	89
d-BPWC_3_6_2.txt	308	309	308	308	309	309
d-BPWC_2_8_5.txt	201	201	201	201	201	201
d-BPWC_3_6_9.txt	292	292	292	292	292	292
d-BPWC_3_5_3.txt	247	247	247	247	248	247
d-BPWC_3_6_6.txt	304	304	304	304	304	304
d-BPWC_3_7_7.txt	358	358	358	358	358	359
d-BPWC_4_5_6.txt	504	505	504	504	504	505
d-BPWC_4_2_5.txt	181	181	181	181	185	184

Capítulo 4.

Conclusões

O problema de empacotamento com conflito é um problema de otimização combinatória de interesse prático porque existem diversas aplicações em situações reais, como problema de corte e empacotamento, escalonamento, entre outras. Neste trabalho foi apresentado uma série de algoritmos que proporcionam soluções factíveis, e de boa qualidade. A Metaheurística Grasp apresentou bons resultados mesmo em baixo tempo de execução.

Apêndice A.

Apêndice

A.1. Dados utilizados para os testes

Todos os arquivos referentes ao trabalho estão disponíveis no repositório: https://github.com/hailtoncarvalho/TP2

Referências Bibliográficas

[1] L. Epstein and A. Levin. On bin packing with conflicts. *Hebrew University*, 5(1):303–305.

[2] C. A. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[3] K. Jansen. Approximation algorithms for time constrained scheduling.

[4] A. E. F. Muritiba et al. Algorithms for the bin packing problem with conflicts. *INFORMS Journal on Computing*, 5(22):14–29.

[5] H. G. Santos. Trabalho prático 2 - paa 2018.1.

[6] M. J. d. S. Souza. Notas de aula: Inteligência computacional para otimização - decom - ufop.