ABORDAGENS PARA RESOLVER O PROBLEMA DA MOCHILA 0/1

Érik Alexander Landim RAFAEL(1); Éfren Lopes de SOUZA (2);
(1) CEFET-AM, Av. Sete de Setembro, 1975 – Centro, Manaus-AM, (92)3621-6700, e-mail: erikrafael@cefetam.edu.br
(2) UFAM, Av. Gen. Rodrigo Octávio Jordão Ramos, 3000 Coroado I, Manaus-AM, (92)9100-1776, e-mail: efren_lopes@hotmail.com

RESUMO

O *Problema da Mochila* (PM) não é somente um problema clássico no contexto de otimização combinatória, mas também apresenta diversas aplicações práticas no mundo real que, muitas vezes, a maioria dos estudantes dos cursos tecnológicos não tem oportunidade de conhecer. Resolver este problema, não somente para instâncias de tamanho pequeno, mas também para instâncias de tamanho grande, tem se revelado um desafio dentro da área de algoritmos e pesquisa operacional. Neste artigo o *Problema da Mochila 0/1* (PM0/1) e algumas de suas variações são apresentadas. Além disso, quatro técnicas para resolver problemas de otimização combinatória são discutidas: *Força Bruta*, *Programação Dinâmica*, *Algoritmo Guloso e Branch and Bound*. O PM0/1 é então abordado segundo as quatro técnicas e um comparativo da complexidade de tempo de execução assintótico, do tempo de execução de máquina, da memória requerida e do esforço de programação necessário para cada algoritmo formulado como solução também é apresentado. Esta pesquisa compilou três das principais técnicas de projeto de algoritmos e sua aplicação a um clássico problema de otimização procurando destacar a criatividade e o uso da matemática na resolução do problema. A metodologia utilizada neste trabalho consistiu das seguintes etapas: levantamento de técnicas de resolução de problemas de otimização, identificação da aplicação das técnicas na resolução do PM0/1, estudo das abordagens, implementação de cada abordagem, projeto e implementação de experimentos para comparação dos tempos de execução de cada solução e geração de gráficos resultantes dos experimentos. Como contribuição oferecida por este trabalho, é possível destacar: a organização deste conhecimento na forma de artigo científico combinando a explicação das técnicas com sua aplicação ao PM0/1, o desenvolvimento de aplicações em Java que implementam as soluções, a comparação dos resultados obtidos com a execução dos experimentos projetados através de gráficos e por fim uma discussão destes resultados.

Palavras-chave: problema da mochila, otimização combinatória, pesquisa operacional, complexidade, projeto de algoritmos.

1. INTRODUÇÃO

O Problema da Mochila não é somente um problema clássico no contexto de otimização combinatória, mas também apresenta diversas aplicações práticas no mundo real. Resolvê-lo, não somente para instâncias de tamanho pequeno, mas também para instâncias de tamanho grande, tem se revelado um desafio dentro da área de algoritmos e pesquisa operacional. Neste artigo, o Problema da Mochila 0/1 e algumas de suas variações são apresentadas e técnicas de otimização combinatória como Força Bruta, Programação Dinâmica, Algoritmo Guloso e Branch and Bound são utilizadas para resolver este problema.

Este artigo está organizado da seguinte forma: a seção 2 formaliza o Problema da Mochila 0/1, apresenta algumas de suas variações e aplicações; a seção 3 resume cada uma das abordagens selecionadas para tratar problemas de otimização combinatória; na seção 4 são apresentadas as características dos algoritmos que resolvem o problema da mochila usando cada técnica mostrada; na seção 5 é fornecido o mecanismo para construção das instâncias de teste e gráficos do tempo de execução de cada algoritmo; a seção 6 apresenta uma breve conclusão da pesquisa; por fim, as referências bibliográficas usadas são fornecidas.

2. DESCRIÇÃO DO PROBLEMA DA MOCHILA

O Problema da Mochila é um dos mais importantes e mais intensamente estudados em Programação Discreta e Otimização Combinatória [Martello e Toth 1990]. O problema pode ser declarado da seguinte forma [Pisinger 1993]: são dados *n* itens para serem colocados em uma mochila de capacidade *c*. Cada item *j* tem um valor p_j e peso w_j associado. Deseja-se encher a mochila obtendo o maior valor possível sem exceder a capacidade $\boldsymbol{c}$. Formalmente o Problema da Mochila 0/1 pode assim ser definido:

$$\text{maximize } \sum_{j=1}^{n} p_j x_j$$

$$\text{sujeito a } \sum_{j=1}^{n} w_j x_j \leq c\,, x_j \in \{0,1\}, j = 1, \ldots, n$$

Onde x_j é igual a 1 se o item *j* está incluído na mochila e 0 caso contrário. Sem perda de generalidade assume-se que $w_j \leq c$, para assegurar que todos os itens cabem na mochila e $\sum_{j=1}^{n} w_j > c$ para evitar soluções triviais.

O problema da mochila apresenta ainda várias aplicações práticas no mundo real, como por exemplo: *carregamento de cargas, seleção de projetos, corte de peças, controle orçamentário* entre outras.

2.1. Variações do Problema da Mochila

[Goldbarg e Luna 2005] e [Martello e Toth 1990] apresentam diversas variações do Problema da Mochila e suas respectivas aplicações. Algumas dessas variações são:

Mochila Limitada: Cada tipo de item que pode se incluído na mochila tem sua quantidade limitada.

Mochila Ilimitada: Cada tipo de item tem quantidade infinita.

Mochila de Múltipla Escolha: Nesse problema, dado um conjunto de classes que contém itens, é necessário escolher exatamente um item de cada classe.

Soma do Subconjunto: Ocorre quando no Problema da Mochila 0/1 o valor de cada item disponível é igual ao seu peso.

Mochila Múltipla 0/1: Nesse problema, existem várias mochilas, cada qual com sua capacidade definida. Os pesos dos itens e seus valores são os mesmos para todas as mochilas, sendo necessário incluir uma restrição adicional que evite a inclusão de um mesmo produto em mais de uma mochila.

Mochila Multidimensional 0/1: Neste problema um pagamento é exigido a cada item adquirido e existe uma limitação no capital disponível para as aquisições, ou seja, deseja-se levar o maior valor possível atendendo a disponibilidade de orçamento.

3. TÉCNICAS PARA RESOLUÇÃO DE PROBLEMAS DE OTIMIZAÇÃO

3.1. Força Bruta

É uma técnica de projeto de algoritmos de fácil implementação, mas que na grande maioria dos casos possui complexidade elevada tendo como importante característica a aplicabilidade em uma grande variedade de problemas. Utiliza abordagem simples para resolver um determinado problema baseando-se diretamente na declaração do mesmo e nas definições dos conceitos envolvidos [Hristakeva e Shrestha 2005].

3.2. Programação Dinâmica

É uma técnica de projeto de algoritmos para resolução de problemas de otimização que tira proveito da existência de soluções ótimas descritas através de relações de recorrência com superposição de subproblemas [Hristakeva and Shrestha 2005]. [Cormen 2002] aponta a existência de quatro etapas no desenvolvimento de um algoritmo usando programação dinâmica: caracterizar a estrutura de uma solução ótima, definir recursivamente o valor da solução ótima em função de uma subestrutura ótima, calcular o valor da solução ótima em um processo *bottom-up* e construir a solução ótima a partir de informações calculadas em subproblemas superpostos. Esta última etapa possibilita que cálculos já realizados sejam utilizados no processo de baixo para cima no cômputo da solução ótima do problema.

3.3. Algoritmo Guloso

Assim como a Programação Dinâmica, os Algoritmos Gulosos também são aplicados a problemas de otimização. De forma diferente da técnica anterior, a abordagem Gulosa, primeiramente realiza uma escolha, aquela que pareça ser a melhor em um determinado momento (a melhor escolha local) e depois resolve um subproblema resultante dessa escolha. Considera-se que a escolha local do melhor possa levar a solução ótima global do problema. Este tipo de técnica é de fácil implementação para problemas de modo geral, porém nem sempre conduzem a soluções ótimas globais. [Cormem 2002] lembra que tais algoritmos usam subestrutura ótima de cima para baixo (*top-down*) e, portanto, podem repetir cálculos. Os algoritmos produzidos por está técnica usam heurísticas ou conhecimento de senso comum para gerar a seqüência de subótimos que poderá convergir ou não para um valor ótimo global [Hristakeva and Shrestha 2005].

3.4. Branch and Bound (B&B)

B&B é um método geral, útil e confiável para encontrar soluções ótimas para problemas de programação inteira e discreta aplicado a diversos problemas de otimização combinatória. O método foi proposto por [Land and Doig, 1960] e consiste em enumerar soluções viáveis, consideradas promissoras, em uma árvore de busca explícita ou implícita. Cada nó na árvore pode ser visto como uma possível solução ou uma solução parcial do problema. Utilizando funções matemáticas e/ou algoritmos que determinam limites superiores (*upper bound*) e inferiores (*lower bound*) do problema com respeito ao valor ótimo da função objetivo, uma grande quantidade de nós da árvore podem ser descartados, reduzindo assim, não só o espaço de armazenamento para resolver o problema, como também o tempo de execução do algoritmo para a instância fornecida. No pior caso, o algoritmo obtido pela aplicação do método é *exponencial*. O termo *branching*, indica que a partir de um nó selecionado, deve ser possível explorar outros nós como alternativas de solução para o problema. O termo *bound*, sugere a identificação de dois limites com relação ao valor ótimo do problema: o inferior e o superior. O valor obtido pela solução ótima do problema de otimização, se existir, é encontrado em um intervalo delimitado por estes dois limites. O método vai reduzindo este intervalo até que estes dois limites sejam iguais quando, portanto, encontra-se o ótimo. [Chinneck 2007] apresenta mais informações sobre a técnica e exemplifica sua aplicação.

A eficiência do método depende criticamente da efetividade na determinação dos limites e no estabelecimento da forma de branching, ou seja, na fixação de valores as variáveis dentro do espaço de restrições do problema original.

Por fim, este método possibilita sacrificar a garantia da otimalidade de um problema muito grande (*very large problem*) em razão de restrições de memória principal ou tempo de resposta sem perder sua efetividade, visto que o mesmo indica que a solução encontrada durante o tempo de busca está em um intervalo. Desta forma, o algoritmo pode ser parado quando uma distância aceitável da otimalidade tiver sido alcançada.

4. ABORDAGENS PARA O PROBLEMA

4.1. Força Bruta

No Problema da Mochila 0/1, os *n* itens podem estar ou não na mochila, ou seja, cada item possui dois estados. Sendo assim, teremos 2^n combinações possíveis, das quais a melhor deve ser escolhida. Inicialmente, considera-se que todos os itens estão fora da mochila, a partir daí gera-se todas as demais combinações, sendo que, a cada nova combinação gerada, compara-se o valor obtido a partir da combinação com o melhor valor até então obtido. Caso a nova combinação produza um valor superior e o seu peso não ultrapasse a capacidade da mochila, esta combinação passa a ser considerada a melhor solução até então obtida. O processo continua até que todas as combinações tenham sido avaliadas. Visto que, se faz necessário a geração de todas as combinações possíveis e para cada combinação é avaliado o valor produzido por ela, a complexidade de tempo deste algoritmo é $O(2^n*n)$ e a complexidade de espaço é $O(n)$, pois utiliza dois vetores: um para guardar a combinação corrente e o outro, a melhor combinação.

4.2. Programação Dinâmica

Segundo [Hristakeva and Shrestha 2005] a representação deste problema pode ser feita em uma tabela de $n+1$ linhas e $c+1$ colunas. Seja $T[i,j]$ a solução ótima de uma instância de i itens, $0 \leq i \leq n$, e capacidade j, $0 \leq j \leq c$, para a mochila. É possível dividir todos os primeiros i itens que cabem na mochila de tamanho j, em duas categorias: aquela que não inclui o *i-ésimo* item e aquela que inclui. Isto permite definir a recorrência que expressa a solução do problema em termos de subproblemas:

$$T_{i,j} = \begin{cases} T_{i-1,j} & ,j < w_i(i) \\ \max\left(T_{i-1,j},\ p_i + T_{i-1,j-w_i}\right) & ,j \geq w_i(ii) \end{cases}$$

A expressão *(i)* indica que o *i-ésimo* item não cabe na mochila de capacidade j e a expressão *(ii)* indica respectivamente que o *i-ésimo* item não será colocado na mochila ou será colocado. Esta decisão, colocar ou não o item na mochila, será baseada no valor proporcionado pela inclusão ou não do item.

As duas condições limites para a mochila são: a mochila não tem valor e nenhum item foi selecionado, $T_{0,j} = 0 \ \forall \ j \geq 0$; e a mochila não tem valor quando sua capacidade é zero, visto que nenhum item pode ser incluído nela, $T_{i,0} = 0 \ \forall \ i \geq 0$. Por fim, a solução ótima pode ser obtida em $T_{n,c}$. Neste algoritmo a complexidade de tempo é $O(n*c)$, pois todos os arranjos de itens são testados em todas as possibilidades das capacidades. A complexidade de espaço também é $O(n*c)$, visto que é necessário registrar as soluções ótimas dos subproblemas em uma matriz para computar a solução ótima.

4.3. Algoritmo Guloso

Foram consideradas três estratégias de seleção gulosa nesta abordagem: a primeira estratégia consiste em selecionar os itens de maior valor com a expectativa de aumentar o valor da mochila rapidamente; a segunda estratégia consiste em selecionar os itens de menor peso buscando levar a maior quantidade de itens possíveis; por fim, a terceira estratégia consiste em selecionar o item com maior razão valor/peso, com a expectativa de levar os itens mais valiosos e leves.

Segundo [Ziviane 2006] este algoritmo pode ser implementado em um processo iterativo que faz a melhor escolha de acordo com uma das estratégias apresentadas acima, desde que essa escolha atenda a capacidade da mochila. A complexidade desse algoritmo é $O(n * \log n)$, pois se faz necessária a ordenação dos itens de acordo com a estratégia de seleção gulosa e a inserção de cada item até que a mochila não suporte nenhum outro item. A complexidade de espaço é $O(n)$, um vetor usado para guardar a solução.

4.4. Branch and Bound (B&B)

[Martello and Toth, 1990] apresentam diversas abordagens para solucionar o problema da Mochila 0/1 utilizando métodos exatos e aproximados. Em 1957, Dantzig apresentou um método eficiente para determinar a relaxação contínua para o problema e, portanto estabeleceu um *upper bound* que foi usado por vinte anos em quase todos os estudos e abordagens para resolver este problema. Em 1967, Kolesar apresentou a primeira abordagem *B&B*. Esta solução requeria uma imensa quantidade de recursos de memória e tempo. Em 1974, Horowitz e Sahni apresentaram uma solução *B&B* mais efetiva, estruturada e

fácil de implementar que posteriormente foi melhorada em outras propostas de solução. Na década de 70, o método *B&B* revelou-se ser o único método capaz de resolver o problema da Mochila 0/1 para uma grande quantidade de variáveis garantindo a otimalidade da solução obtida.

A abordagem *B&B* utilizada por Horowitz and Sahni baseia-se nas seguintes premissas:

1. Assume-se que os items estão ordenados de acordo com os valores não-crescentes de benefícios por unidade de peso (weight): $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \cdots \geq \frac{p_n}{w_n}$
2. Os items ordenados, de acordo com a premissa anterior, são consecutivamente inseridos na mochila até o primeiro item, *s*, não caber mais – chamado de item crítico ou *break item*: $s = min\{j: \sum_{i=1}^{j} w_i > c\}$
3. O *upper bound* (*UB*) para o problema é obtido pela relaxação da restrição inteira proposto por Dantzig, onde $\bar{x_s} = \frac{\bar{c}}{w_s}$, sendo $\bar{c} = c - \sum_{j=1}^{s-1} w_j$ a capacidade residual da mochila. Assim, temos: $UB = \sum_{j=1}^{s-1} p_j + \left\lfloor \bar{c} \frac{p_s}{w_s} \right\rfloor$

O algoritmo produz uma enumeração implícita das soluções viáveis para o problema utilizando para isto as seguintes operações:

1. *Forward move (fm)*: inserir o maior conjunto possível de novos itens consecutivos em uma solução parcial até obtermos uma solução viável (todas as variáveis do problema foram fixadas, ou seja, foi-se atribuído um valor a cada variável) para o problema;
2. *Backtracking move (bm)*: remover o último item inserido em uma solução viável ou parcial;

Sempre que a operação *fm* não puder ser aplicada, o *UB* correspondente a corrente solução é calculado e comparado com a melhor solução obtida até o momento. Se a nossa melhor solução for maior que o *UB* obtido, isto significa que não vale a pena prosseguir naquela direção, ou seja, tendo fixado a última variável com o valor 1. De outra forma, isto significa que o último item colocado na mochila não é uma boa decisão no sentido de encontrar uma solução melhor do que aquela já encontrada e, portanto um *backtracking* deve ser realizado dando oportunidade a um novo arranjo de itens ser explorado. Quando o último item da lista de itens for considerado pelo algoritmo, uma solução viável foi estabelecida. O algoritmo termina quando não houver mais possibilidade de executar *backtracking*.

Com relação a complexidade de tempo assintótico deste algoritmo, pode-se dizer que ele é $\Omega(n * logn)$ considerando a primeira premissa citada. Com relação ao limite superior, não é simples estabelecer esse valor, mas é possível perceber, analisando o algoritmo, que uma grande quantidade de soluções viáveis são descartadas, o que reduz de maneira substancial as possibilidades a serem avaliadas.

A versão completa do algoritmo escrito em notação matemática e pseudo-código pascal está disponível em [Martello and Toth 1990]. Uma árvore de decisão produzida pela execução do algoritmo aplicada a mochila de capacidade 50 com 7 itens tendo P_j *= (70, 20, 39, 37, 7, 5, 10)* e W_j *= (31, 10, 20, 19, 4, 3, 6)* é também fornecida.

5. EXPERIMENTOS E RESULTADOS

Os algoritmos apresentados foram implementados em Java em um processador Core2Duo 2.0GHz com 2GB de memória. O código fonte está disponível sob requisição.

Avaliou-se o comportamento dos algoritmos para entradas de diferentes tamanhos e faixa de dados. Como sugerido por [Pisinger 1993], a faixa de dados é limitada a *R=100, 1.000, 10.000* e os valores e pesos dos itens foram gerados randomicamente de forma não correlacionada, ou seja, ambos distribuídos aleatoriamente no intervalo *[1,R]*. A capacidade da mochila é correspondente a metade do somatório dos pesos, sendo assim, $c = \frac{\sum_{j=1}^{n} w_j}{2}$.

Foi observado que das três estratégias gulosas implementadas, a última (razão valor/peso) sempre apresenta soluções com maior resultado. Com base nisso, somente ela é considerada para efeito de comparação com as outras técnicas.

O comportamento de cada algoritmo com relação ao tamanho da entrada fornecida e o tempo gasto de processamento na escala dos nano segundos são mostrados na *Figura 1*.

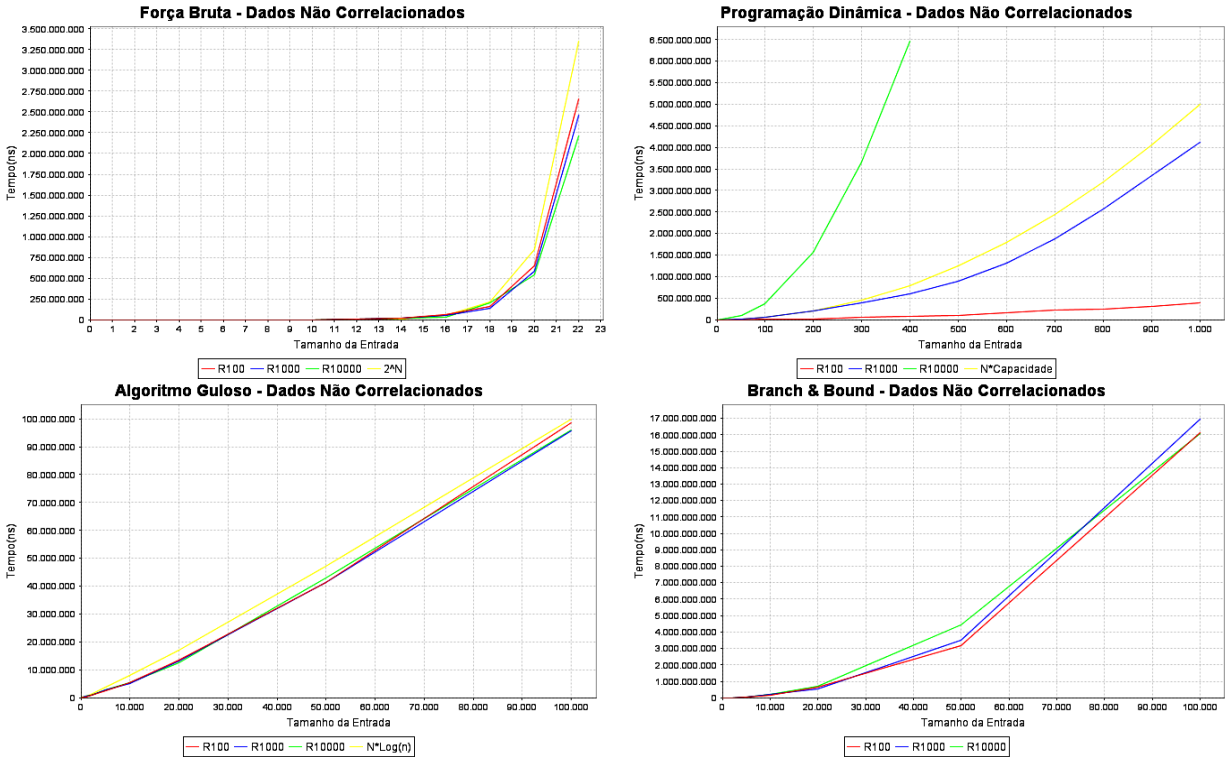

Figure 1. Tempo de execução de cada técnica

A *Tabela 1* mostra os resultados da execução de cada técnica. A solução ótima e seus respectivos valores são encontrados pelas técnicas de *Força Bruta*, *Programação Dinâmica* e *Branch and Bound*. O resultado obtido pela técnica Gulosa, utilizando a estratégia razão valor/peso, apesar de não apresentar a solução ótima, fornece um valor próximo a ela. A qualidade desta solução é 3,22% inferior ao ótimo.

Tabela 1. Resultado de cada técnica

Técnica	Nº Itens	Capacidade	Valor da Solução	Peso da Solução
Força Bruta	22	5712	8540	5704
Prog. Dinâmica	22	5712	8540	5704
Guloso	22	5712	8273	5572
Branch and Bound	22	5712	8540	5704

6. CONCLUSÕES

O algoritmo de *Força Bruta* foi testado com no máximo 22 itens, pois gera obrigatoriamente todas as possibilidades para chegar a um resultado ótimo. É útil apenas para pequenas instâncias e fins acadêmicos, devido a sua fácil implementação.

O algoritmo de *Programação Dinâmica* possui um custo de tempo bem menor, porém utiliza muita memória para registrar o resultado da solução ótima dos subproblemas, sendo que boa parte desse uso exagerado de memória, não se deve apenas a quantidade de itens, mas também a capacidade da mochila. Por isso, para evitar estouros de memória foi necessário inicializar a JVM com espaço extra de memória e mesmo assim, só foi possível verificar os resultados de 400 itens com *R=10.000*, bem abaixo dos 1.000 itens verificados com *R=100, 1.000*.

O algoritmo *Guloso* é bem mais simples de ser implementado e bem mais barato em termos de tempo e memória, porém não garante encontrar a solução ótima, mas sim uma solução aproximada. É excelente para

se trabalhar com grandes entradas em que a solução ótima não é necessária e há restrições rígidas de tempo de resposta para a instância do problema.

Por fim, o *Branch and Bound* é o que exige maior esforço na implementação e no entendimento dos seus conceitos. Entretanto, este algoritmo consome pouco recurso de memória e, apesar de ter, teoricamente, um custo assintótico de tempo exponencial, ele forneceu a solução ótima para uma instância de 100.000 itens em um tempo razoável. Esta quantidade de itens, até então, só tinha sido resolvida pelo Algoritmo Guloso, que infelizmente não alcançou o ótimo. Assim, *Branch and Bound* é indicado para situações em que a quantidade de itens é grande e deseja-se obter a solução ótima.

7. REFERÊNCIAS

Chinneck, J.W., (2007). **Practical Optimization: A Gentle Introduction**. Disponível em: <http://www.sce.carleton.ca/faculty/chinneck/po.html>

Cormen H. (2002). **Introduction to Algorithms**. Second Edition.

Goldbarg, M. e Luna, H. (2005). **Otimização Combinatória e Programação Linear: Modelos e Algoritmos**. 2ª Edição.

Hristakeva, M. e Shrestha, D. (2005). **Different Approaches to Solve the 0/1 Knapsack Problem**. Editora Campus. Rio de Janeiro.

Land, A.H. and Doig, A.G. (1960). **An automatic method for solving discrete programming problems Econometrica**. 28, Julho, Number 3.

Martello, Silvano e Toth, P. (1990). **Knapsack Problems – Algorithms and Computer Implementations**. John Wiley & Sons ltd., England.

Pisinger, D. (1993). **An expanding-core algorithm for exact 0-1 Knapsack Problem**. European Journal of Operational Research, 87:175-187, 1995.

Ziviani, N. (2006). **Projeto de Algoritmos com implementações em Java e C++**. Thomson Learning. Primeira Edição.