UNIVERSIDADE DO ESTADO DE SANTA CATARINA

CENTRO DE EDUCAÇÃO SUPERIOR DO ALTO VALE DO ITAJAÍ (CEAVI)

ENGENHARIA DE SOFTWARE

GABRIEL NAOTO YMAI PEREIRA

JANDIR LUIZ HABITZREUTER

LUMA TURATTO HOSCHER

MARCOS RUFINO DE CAMARGO

SILA GEORGES AGIRÚ JUDICK SIEBERT

PROJETO INTEGRADOR II: BAÚ DA FELICIDADE ABORDADO PELO PROBLEMA DA MOCHILA DE MÚTIPLA ESCOLHA

IBIRAMA

2016

1 Sumário

2 O Cenário

Nossa empresa foi procurada por um grande empresário que deseja lançar um produto inovador, uma cesta de produtos surpresa denominada provisoriamente de "Baú da Felicidade". Segundo o empreendedor Senhor Abravanel, os clientes pagam uma mensalidade e a cada trimestre eles recebem uma cesta com produtos surpresa. O cliente ainda não tem claro o funcionamento de todo o processo. Contudo já obtivemos algumas premissas extraídas de uma conversa informal.

2.1 OS PRODUTOS

- A compra dos produtos será através de um processo composto por uma tomada de preços e a escolha dos produtos. A tomada de preço é similar a um leilão fechado. Será lançado uma intenção de compra e os fornecedores submetem seu preço a cada um dos produtos. Havendo concordância com o valor apresentado pelo fornecedor a compra é efetuada.
- Os produtos são separados por categorias.
- A cesta só pode conter produtos que não tenham sido utilizados em outras cestas por um período de 1 ano.

2.2 A CESTA

- A princípio, uma cesta não pode conter mais que um produto de uma categoria.
- A cesta deve ter o menor custo possível.
- Deve ser levado em consideração a satisfação que cada produto proporciona aos clientes.
- Desconsiderar a quantidade de produtos da cesta, apenas certificar-se de que o valor total da cesta não ultrapasse o valor máximo estipulado para a cesta do trimestre e o conjunto de produtos selecionados para a cesta resulte no maior índice de satisfação dos clientes.
- Para resolução do problema deve-se arredondar os preços dos produtos para cima, caso os mesmos possuam casas decimais, para que os preços sejam tratados como números inteiros e garantir que o domínio matemático do problema seja discreto.
- O problema deve ser tratado como um Problema da Mochila de Múltipla Escolha.

2.3 A ENTREGA

- O entregador tem que fazer viagens otimizadas e gastar o mínimo de tempo possível. Ao final o mesmo deve regressar ao depósito da empresa.
- A entrega será sempre realizada no terceiro dia após o pagamento da terceira mensalidade.
- Cada entregador pode trabalhar no máximo por 6 horas nas entregas.

3 Problema da Mochila aplicado ao Baú da Felicidade

Segundo (Luila, 2008), o Problema da Mochila caracteriza uma classe de problema de programação linear inteira e são classificados na literatura, segundo a sua complexidade de resolução, como problemas NP-Difícil. O problema é um dos mais importantes em programação linear inteira e tem sido estudado intensivamente nos últimos anos por vários investigadores.

O problema da mochila (em inglês, Knapsackproblem) é um problema de otimização combinatória. O nome dá-se devido ao modelo de uma situação em que é necessário preencher uma mochila com objetos de diferentes pesos e valores. O objetivo é que se preencha a mochila com o maior valor possível, não ultrapassando o peso máximo.

O problema da mochila é um dos 21 problemas NP-completos de Richard Karp, exposto em 1972. A formulação do problema é extremamente simples, porém sua solução é mais complexa. Este problema é a base do primeiro algoritmo de chave pública (chaves assimétricas).

> Um cenário onde se tem um conjunto de itens a serem colocados em uma mochila é denominado de problema da mochila. Nesse problema, existe uma determinada quantidade de itens, cada qual com o seu peso e valor, onde deseja-se colocar esses itens em uma mochila com uma capacidade predefinida. O objetivo é colocar os itens na mochila de modo a se obter o maior valor (composto pela soma de valores de cada item inserido na mochila), desde que não ultrapasse o peso total suportado pela mochila. O problema da mochila 0-1 (também chamado de problema da mochila binária) é considerado o mais simples deles (MARTELLO; TOTH, 1990).

Comparando o problema do "Baú da Felicidade" com o Problema NP da Mochila, pode-se afirmar que ambos possuem três variáveis, a mochila possui o peso total da mochila e cada objeto a ser colocado nela possui um peso e valor específicos, busca-se potencializar o valor final da mochila dentro do peso estipulado. No "Baú da Felicidade" temos o valor máximo da Cesta (equivalente ao peso máximo da Mochila) e os objetos possuem valor e satisfação, o que deve ser potencializada é a satisfação do cliente, logo, o valor de cada objeto a ser colocado na Cesta, equivale ao peso do objeto da Mochila, e a satisfação dos objetos da Cesta equivalem ao preço dos objetos da Mochila.

Os métodos abordados nesta pesquisa são: solução usando Programação Dinâmica, solução usando o Método Guloso e solução usando relaxação Lagrangeana.

3.1 PROBLEMA DA MOCHILA DE MÚLTIPLA ESCOLHA

O problema da Mochila com Múltipla Escolha (Multiple-choice Knapsack Problem) ocorre quando os itens devem ser escolhidos de classes disjuntas, e se várias Mochilas são preenchidas simultaneamente temos o problema da Mochila Múltiplo (Multiple Knapsack Problem).

O problema de múltiplas mochilas é muito similar ao problema da mochila simples. A diferença está no número de mochilas utilizadas: enquanto no problema simples só utiliza uma mochila, no de múltiplas mochilas são utilizadas várias mochilas. Esse problema foi proposto para ser aplicado em problemas de tolerância a falhas (SINHA; ZOLTNERS, 1979), esquema de criptografia pública (DIFFIE; HELLMAN, 1976), problemas de alocação e escalonamento, entre outros.

3.2 PROBLEMAS NP-COMPLETOS

Na teoria da complexidade computacional, a classe de complexidade é o subconjunto dos problemas NP de tal modo que todo problema em NP se pode reduzir, com uma redução de tempo polinomial, a um dos problemas NP-completo. Pode-se dizer que os problemas de NP-completo são os problemas mais difíceis de NP e muito provavelmente não formem parte da classe de complexidade P.

A razão é que se conseguisse encontrar uma maneira de resolver qualquer problema NP-completo rapidamente (em tempo polinomial), então poderiam ser utilizados algoritmos para resolver todos problemas NP rapidamente. Na prática, o conhecimento de NP-completo pode evitar que se desperdice tempo tentando encontrar um algoritmo de tempo polinomial para resolver um problema quando esse algoritmo não existe.

4 Possíveis Soluções

4.1 PROGRAMAÇÃO DINÂMICA

Programação dinâmica é um método para a construção de algoritmos para a resolução de problemas computacionais, em especial os de otimização combinatória. Ela é aplicável a problemas nos quais a solução ótima pode ser computada a partir da solução ótima previamente calculada e memorizada - de forma a evitar recálculo - de outros subproblemas que, sobrepostos, compõem o problema original.

O que um problema de otimização deve ter para que a programação dinâmica seja aplicável são duas principais características: subestrutura ótima e superposição de subproblemas. Um problema apresenta uma subestrutura ótima quando uma solução ótima para o problema contém em seu interior soluções ótimas para subproblemas. A superposição de subproblemas acontece quando um algoritmo recursivo reexamina o mesmo problema muitas vezes.

Quadro 1: Programação Dinâmica para resolver o Knapsack Problem

```
9 public class Knapsack {
20
21     public static void main(String[] args) {
22         int N = Integer.parseInt(args[0]);
23         int W = Integer.parseInt(args[1]);
24         int[] profit = newint[N + 1];
25         int[] weight = newint[N + 1];
26         for (int n = 1; n <= N; n++) {
27             profit[n] = (int) (Math.random() * 1000);
28             weight[n] = (int) (Math.random() * W);
29         }
30         int[][] opt = new int[N + 1][W + 1];
31         boolean[][] sol = new boolean[N + 1][W + 1];
32         for (int n = 1; n <= N; n++) {
33             for (int w = 1; w <= W; w++) {
34                 int option1 = opt[n - 1][w];
35                 int option2 = Integer.MIN_VALUE;
36                 if (weight[n] <= w) {
37                     option2 = profit[n] + opt[n - 1][w - weight[n]];
38                 }
39                 opt[n][w] = Math.max(option1, option2);
40                 sol[n][w] = (option2 > option1);
41             }
42         }
43         boolean[] take = new boolean[N + 1];
44         for (int n = N, w = W; n > 0; n--) {
45             if (sol[n][w]) {
46                 take[n] = true;
47                 w = w - weight[n];
48             }else {
49                 take[n] = false;
```

```
50              }
51          }
52          System.out.println("item" + "\t" + "profit" + "\t" + "weight" +
"\t" + "take");
53          for (int n = 1; n <= N; n++) {
54              System.out.println(n + "\t" + profit[n] + "\t" + weight[n]
+ "\t" + take[n]);
55          }
56      }
57  }
```

Fonte: Robert Sedgewickand Kevin Wayne[1]

4.2 MÉTODO GULOSO

Técnica utilizada para problemas de otimização. Sempre faz a escolha que parece melhor no momento. Sugere construir uma solução através de uma sequência de passos, cada um expandindo uma solução parcialmente construída até o momento, até ser obtida uma solução completa para o problema.

Em cada passo, a escolha deve ser feita:

- Possível - Deve satisfazer as restrições do problema.
- Localmente ótima – Deve ser a melhor escolha local dentre todas as disponíveis.
- Irreversível – Uma vez feita, ela não pode ser alterada nos passos seguintes do algoritmo.

Há expectativas de que escolhas locais ótimas levem a uma solução ótima global para o problema como um todo. Por ser um algoritmo que usa estratégia gananciosa, faz sempre escolhas que, naquele instante, parecem excelentes. Isto pode levar a uma solução ótima, ou não, mas provavelmente não vai levar a uma solução insatisfatória.

Quadro 2: Método Guloso para resolver o Knapsack Problem

```
 5 package knapsack;
 6
 7 public class GreedyKnapsack {
12
13      double[] profit;
14      double[] weight;
15      double[] take;
16
17      public GreedyKnapsack(int n) {
18
19          profit = new double[n];
20          weight = new double[n];
21          take = new double[n];
```

[1] Disponível em: <http://introcs.cs.princeton.edu/java/96optimization/Knapsack.java.html>. Acesso em: 14 mar. 2016.

```
22          for (int i = 0; i < n; i++) {
23              profit[i] = (int) Math.round(Math.random() * 96 + 44);
24              weight[i] = (int) Math.round(Math.random() * 89 + 15);
25          }
26      }
27
28      public void unitPriceOrder() {
29          for (int i = 0; i < profit.length; i++) {
30              for (int j = 1; j < (profit.length - i); j++) {
31                  double x = profit[j - 1] / weight[j - 1];
32                  double y = profit[j] / weight[j];
33                  if (x <= y) {
34                      double temp = profit[j - 1];
35                      profit[j - 1] = profit[j];
36                      profit[j] = temp;
37
38                      double temp1 = weight[j - 1];
39                      weight[j - 1] = weight[j];
40                      weight[j] = temp1;
41                  }
42              }
43          }
44      }
45
46      public void Knapsack(int m) {
47          unitPriceOrder();
48          int j;
49          for (j = 0; j < profit.length; j++) {
50              take[j] = 0;
51          }
52          double total = m;
53          for (j = 0; j < profit.length; j++) {
54              if (weight[j] <= total) {
55                  take[j] = 1.00;
56                  total = total - weight[j];
57              } else {
58                  break;
59              }
60          }
61          if (j < profit.length) {
62              take[j] = (double) (total / weight[j]);
63          }
64      }
65
66      public void print() {
67          System.out.println("item" + " |  " + "profit" + "  |   " +
"weight"
68                  + "   |     " + "Unit Price" + "      |" + "  Take
weight");
69          for (int n = 0; n < profit.length; n++) {
70              System.out.println(n + "\t" + profit[n] + "\t" + weight[n]
+ "\t"
71                      + (profit[n] / weight[n]) + "\t" + take[n]);
72          }
73      }
74
75      public static void main(String args[]) {
76          GreedyKnapsack G = new GreedyKnapsack(10);
77          System.out.println("Optimal soluation to knapsack instance "
78                  + "with values given as follows : m=35");
79          G.Knapsack(35);
```

```
80          G.print();
81
System.out.println("=======+============+=======+============+="
82                  + "============");
83          System.out.println("Optimal soluation to knapsack instance with
"
84                  + "values given as follows : m=120");
85          G.Knapsack(120);
86          G.print();
87      }
88  }
```

Fonte: Achuchuthan[2]

Quadro 2: Saída da execução do Método Guloso

```
run:
Optimal soluation to knapsack instance with values given as follows : m=35
item |  profit  |   weight   |     Unit Price      |  Take weight
0         92.0         17.0         5.411764705882353         1.0
1         113.0        61.0         1.8524590163934427      0.295081
2         65.0         41.0         1.5853658536585367        0.0
3         139.0        96.0         1.4479166666666667        0.0
4         101.0        70.0         1.4428571428571428        0.0
5         66.0         59.0         1.11864406779661          0.0
6         79.0         76.0         1.0394736842105263        0.0
7         64.0         74.0         0.8648648648648649        0.0
8         84.0         98.0         0.8571428571428571        0.0
9         47.0         93.0         0.5053763440860215        0.0
=======+============+=======+============+=======+============
Optimal soluation to knapsack instance with values given as follows :
m=120
item |  profit  |   weight   |     Unit Price      |  Take weight
0         92.0         17.0         5.411764705882353         1.0
1         113.0        61.0         1.8524590163934427        1.0
2         65.0         41.0         1.5853658536585367        1.0
3         139.0        96.0         1.4479166666666667        0.01041
4         101.0        70.0         1.4428571428571428        0.0
5         66.0         59.0         1.11864406779661          0.0
6         79.0         76.0         1.0394736842105263        0.0
7         64.0         74.0         0.8648648648648649        0.0
8         84.0         98.0         0.8571428571428571        0.0
9         47.0         93.0         0.5053763440860215        0.0
BUILD SUCCESSFUL (total time: 0 seconds)
```

Fonte: Achuchuthan[2]

4.3 RELAXAÇÃO LAGRANGEANA

Existe uma técnica denominada relaxação lagrangeana, que consiste em remover algumas das restrições da formulação original, mas tenta embutir essas desigualdades na função objetivo. A ideia é penalizar a função objetivo quando as restrições removidas forem violadas. O "peso" dessas penalidades é controlado por coeficientes chamados multiplicadores lagrangeanos.

---

[2] Disponível em <http://www.java.achchuthan.org/2012/02/knapsack-problem-in-java.html>. Acesso em: 15 mar. 2016.

A relaxação Lagrangeana é uma técnica poderosa para se obter limitantes duais de problemas combinatórios que podem ser modelados como programas lineares inteiros. Esses limitantes podem ser usados na tentativa de melhorar o desempenho de algoritmos branch-and-bound.

Além do mais, para certos problemas, existe a possibilidade de ajustar a solução obtida via relaxação de modo a obter uma solução viável para o problema original, esperando que a qualidade da solução não seja muito pior.

Figura 1: Heurística Lagrangeana

Método do Subgradiente(π, T_π, N, u_0)

1. $\bar{x}^*$ *//Melhor solução primal*
2. Enquanto condição de parada não for satisfeita, faça:
3. $\quad x^* \leftarrow$ *Dual_lagrangeano*()
4. $\quad$ Se $f'(x^*) \geq z_{UB}$, faça:
5. $\quad\quad z_{LB} \leftarrow f'(x^*)$;
6. $\quad\quad n_{stuck} \leftarrow 0$;
7. $\quad$ Caso contrário faça:
8. $\quad\quad n_{stuck} \leftarrow n_{stuck} + 1$;
9. $\quad \bar{x} \leftarrow$ *Heurística_Lagrangeana()*; *//Solução primal viável*
10. $\quad$ Se o valor de $f(\bar{x}) \leq z_{LB}$, faça:
11. $\quad\quad z_{LB} \leftarrow f(\bar{x})$;
12. $\quad\quad \bar{x}^* \leftarrow \bar{x}$;
13. $\quad$ Se $z_{LB} = z_{UB}$ *//Encontramos a solução ótima*:
14. $\quad\quad$ PARE;
15. $\quad u_l^{k+1} \leftarrow$; *Atualiza_multiplicadores*$(\pi, z_{LB}, z_{UB}, u_l^k, x^*)$;
16. $\quad$ Se $n_{stuck} = N$, faça:
17. $\quad\quad n_{stuck} \leftarrow 0$;
18. $\quad\quad \pi \leftarrow \pi/2.0$;
19. $\quad\quad$ Se $\pi \leq T_\pi$:
20. $\quad\quad\quad$ PARE;
21. $\quad k \leftarrow k + 1$;
22. Retorne $\bar{x}^*$:

Onde a função `Atualiza_multiplicadores` é dada por:

Atualiza_multiplicadores$(\pi, z_{LB}, z_{UB}, u_l^k, x^*)$

1. $G_\ell = b_\ell - \sum_{j=1}^{m} a_{\ell j} x_{\ell j}^* \qquad \forall \ell \in R$
2. $T = \dfrac{\pi(1.05 \cdot z_{UB} - z_{LB})}{\sum_{\ell \in R} G_\ell^2}$
3. $u_\ell^{k+1} = \max\{0, u_\ell^k + TG_\ell\} \qquad \forall \ell \in R$
4. Retorne u^{k+1}

Fonte: Kunigami[3]

[3] Disponível em: <https://kuniga.wordpress.com/2012/03/11/relaxacao-lagrangeana-pratica/>. Acesso em: 15 mar. 2016.

5 Considerações Finais

A princípio, para testes iniciais, utilizaremos o Método Guloso, mas, como apresentado, este método não é tão eficiente comparado a outros, como por exemplo, o Método da Programação Dinâmica, que será nossa segunda tentativa de abordagem do problema.

A pesquisa ainda é muito recente para afirmar com precisão que utilizaremos o Método da Programação Dinâmica para solucionar o problema, apesar de nos parecer o mais viável, ainda realizaremos outras pesquisas para nos aprofundar em cada um dos métodos apresentados.

6 Referências Bibliográficas

WIKIPEDIA. **Problema da Mochila.** Disponível em: <https://pt.wikipedia.org/wiki/Problema_da_mochila>. Acesso em: 03 mar. 2016

CALDAS, R. **Projeto e Análise de Algoritmos.** Belo Horizonte, 2004. 44p. Disponível em: <http://homepages.dcc.ufmg.br/~nivio/cursos/pa04/tp2/tp22/tp22.pdf>. Acesso em: 03 mar. 2016.

BARBOSA, F. R. S; SOUSA, F. R. H; VILELA, L, R. **Problema da Mochila**. Uberlândia, 11p. Disponível em: <http://www2.ic.uff.br/~jsilva/artigo_problema_da_mochila.pdf>. Acesso em: 15 mar. 2016.

KUNIGAMI. **Relaxação Lagrangeana:** Teoria. Disponível em <http://www2.ic.uff.br/~jsilva/artigo_problema_da_mochila.pdf>. Acesso em: 15 mar. 2016.

KUNIGAMI. **Relaxação Lagrangeana:** Prática. Disponível em <http://www2.ic.uff.br/~jsilva/artigo_problema_da_mochila.pdf>. Acesso em: 15 mar. 2016.

CARVALHO, E. C. R; SILVA M. M. **Otimização do Problema da Mochila**. Barbacena, 14p. Disponível em: <http://www.unipac.br/site/bb/tcc/tcc-33fc0129a58e28afaa0283f0a0af2f1d.pdf>. Acesso em: 15 mar. 2016.