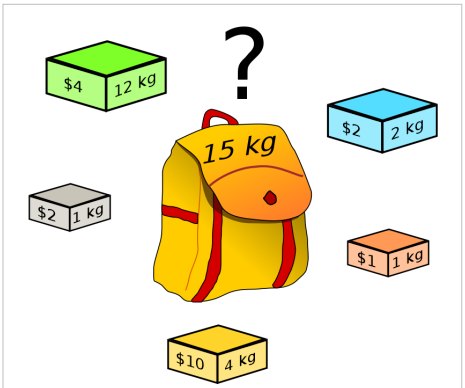
Problema da mochila



Problema da mochila: Como maximizar o valor com um peso máximo?

O **problema da mochila** (em inglês, *Knapsack problem*) é um problema de optimização 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).

Normalmente este problema é resolvido com programação dinâmica, obtendo então a resolução exata do problema, mas também sendo possível usar PSE (procedimento de separação e evolução). Existem também outras técnicas, como usar algoritmo guloso, meta-heurística (algoritmos genéticos) para soluções aproximadas.

História

Também conhecido como "knapsack problem", o Problema da Mochila está relacionado com um grande número de outros modelos de programação. Sua importância está associada exatamente a esse fato.

Foi possivelmente reportado pela primeira vez na literatura por Dantzig (1957) e constitui um marco das técnicas de programação inteira, otimização combinatória e programação dinâmica.

Metaforicamente podemos entendê-lo como o desafio de encher uma mochila sem ultrapassar um determinado limite de peso, otimizando o valor do produto carregado.

O problema da mochila possui alguns variantes que são:

- O Problema simples da mochila: Onde o ladrão possui uma mochila e vários objetos cada tipo com o seu valor.
- O Problema múltiplo da mochila: Onde o ladrão possui mais de uma mochila ou uma mochila com vários bolsos e vários objetos de cada tipo com o seu valor.

O problema simples da mochila foi resolvido por Ronald Rivest, Adi Shamir e Len Adleman em 1982 e o problema dinâmico da mochila foi resolvido por Ernmie Brickell em 1984.

Definição

Podemos definir o problema matematicamente da seguinte forma:

Dados vetores x[1..n] e w[1..n], vamos denotar por x·w o produto escalar de x por w:

```
x·w  =  x[1]w[1] + ... + x[n]w[n] .
```

Suponha dado um número positivo ou nulo W e vetores w[1..n] e v[1..n] com componentes positivos ou nulos. Uma mochila é qualquer vetor x[1..n] tal que

```
x·w  ≤ W     e     0 ≤ x[i] ≤ 1  para todo i
```

O valor de uma mochila é o número x·v. Uma mochila é ótima se tem valor máximo. Consideremos então os dois problemas a seguir:

Problema Fracionário da Mochila (para solução aproximada):

```
Dados w, v, n e W,  encontrar uma mochila ótima.
```

Problema Booleano da Mochila (para solução exata):

```
Dados w, v, n e W,  encontrar uma mochila ótima dentre as que satisfazem a seguinte restrição adicional:   para cada i,  x[i] = 0  ou  x[i] = 1.
```

Motivação

Imagine que você possui um contêiner de certo tamanho e vários produtos com seus valores para serem colocados dentro dele. Porém os produtos devem ser colocados de um jeito que o contêiner tenha o maior valor.

Suponha que você tenha em sua máquina uma pasta cheia de arquivos e deseja gravá-los em CD, só que você já sabe de antemão que não vai caber tudo num CD só. Podem ser dois, três... dez CD's. Como proceder para encaixar o máximo de arquivos em cada CD desperdiçando o mínimo espaço em cada disco?

Desses exemplos podemos entender que a motivação do problema da mochila é colocar dentro de um compartimento uma quantidade de objetos com determinadas importâncias para que esse compartimento tenha a maior importância possível.

Aplicações Práticas

O modelo em si pode ser aplicado, além nos exemplos citados acima, em casos como: Investimento de capital, corte e empacotamento, carregamento de veículos, orçamento.

Foi também utilizado para a construção do algoritmo de Criptografia de chave pública, onde no contexto do problema da mochila a chave pública seria o peso total que a mochila pode carregar e a partir daí, por essa palavra a encriptação é gerada.

Resoluções

Por ser um problema combinatório é possível resolve-lo por enumeração exaustiva, ou seja tentar todas as soluções possíveis e comparando-as para identificar a melhor solução. Porém isso se torna completamente inviável uma vez que, mesmo em pequenas dimensões como um problema com apenas 20 itens, haveria um número enorme de respostas. Em um problema com mais de 80 itens, o algoritmo levaria bilhões de anos para ser concluido. Assim, o método de resolução por enumeração exaustiva é de utilidade muito reduzida, senão mesmo nula, para problemas de grande dimensão. Descreveremos aqui três soluções para este algoritmo: **1. Solução usando Backtracking**
Backtracking é um algoritmo refinado da busca por força bruta (ou enumeração exaustiva), no qual boa parte das soluções podem ser eliminadas sem serem explicitamente examinadas. É uma estratégia que se aplica em problemas cuja solução pode ser definida a partir de uma seqüência de n decisões, que podem ser modeladas por uma árvore que representa todas as possíveis seqüências de decisão. De fato, se existir mais de uma disponível para cada uma das n decisões, a busca exaustiva da árvore é exponencial. A eficiência desta estratégia depende da possibilidade de limitar a busca, ou seja, podar a árvore, eliminando os ramos que não levam a solução desejada. Para tanto é necessário definir um espaço de solução para o problema. Este espaço de solução deve incluir pelo menos uma solução ótima para o problema. Depois é preciso organizar o espaço de solução de forma que seja facilmente pesquisado. A organização típica é uma árvore. Só então se pode realizar a busca em profundidade.

2. Solução usando o Método Guloso

O Método Guloso é um dos algoritmos mais simples que pode ser aplicado a uma grande variedade de problemas, que na maioria possuem n entradas e é necessário obter um subconjunto que satisfaça alguma restrição, como no Problema da Mochila. Qualquer subconjunto que satisfaça esta restrição é chamado de solução viável. Então o que queremos é uma solução viável que maximize ou minimize uma dada função objetivo. Essa função viável que

satisfaz a função objetivo é chamada de Solução Ótima. No Método Guloso construimos um algoritmo que trabalha em estágios, considerando uma entrada por vez. Em cada estágio é tomada uma decisão considerando se uma entrada particular é uma solução ótima. Isto é feito considerando as entradas em uma ordem determinada por um processo de seleção, que é baseado em alguma medida de otimização que pode ou não ser a função objetivo. Na maioria das vezes, porém, essas medidas de otimização resultarão em algoritmos que gerarão soluções sub-ótimas.

3. Solução usando Programação Dinâmica

Programação Dinâmica, ou Função Recursiva, foi a primeira técnica mais inteligente que foi usada para resolver esse problema, na década de 50. É um método aprimorado de usar recursividade que ao invés de chamar uma função várias vezes ele, na primeira vez que é chamado, armazena o resultado para que cada vez que a função for chamada novamente volte o resultado e não uma requisição para ser resolvida. No Método Guloso, uma solução ótima é definida por uma sequencia de decisões ótimas locais. Quando esse método não funciona, uma possível saída seria gerar todas as possíveis sequencias de decisões e escolher a melhor sequencia, como no Backtracking. Porém essa solução é ineficiente, por ser de ordem exponencial. Na programação dinâmica é possível avaliar todas as soluções, garantido assim que a resposta final estará correta, e armazenar resultados anteriores impedindo dessa forma contas repetidas, o que deixa esse algoritmo mais eficente.

4. Solução usando Algoritmo de Aproximação de Greedy

Foi proposto por George Dantzig, em 1957, um algoritmo de aproximação de greedy para resolver o problema da mochila. A versão dele dispõe os itens em ordem decrescente de valor por unidade de peso. Em seguida, começa a inseri-los na mochila com tantas cópias quanto possível do primeiro tipo de item, até que não haja mais espaço na mochila. Caso o problema seja delimitado, ou seja, a oferta para cada tipo de item tenha um limite, o algoritmo pode ficar muito custoso.

Exemplo prático (JAVA)

```java
import java.util.Random;
/**
*
*
* Compilation:  javac Knapsack.java
*  Execution:    java Knapsack N P W
*
*  Generates an instance of the 0/1 knapsack problem with N items,
*  profits between 0 and P-1, weights between 0 and W-1,
*  and solves it in O(NW) using dynamic programming.
*
*  %  java Knapsack 6 1000 2000
*  item    profit  weight  take
*  1       874     580     true
*  2       620     1616    false
*  3       345     1906    false
*  4       369     1942    false
*  5       360     50      true
*  6       470     294     true
*
**************************************************************************/

public class Knapsack {
```

```
    private static final Random rnd = new Random();
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);   // number of items
        int P = Integer.parseInt(args[1]);   // maximum profit
        int W = Integer.parseInt(args[2]);   // maximum weight
        int[] profit = new int[N+1];
        int[] weight = new int[N+1];

        // generate random instance, items 1..N
        for (int n = 1; n <= N; n++) {
            profit[n] = rnd.nextInt(P);
            weight[n] = rnd.nextInt(W);
        }

        // opt[n][w] = max profit of packing items 1..n with weight
limit w
        // sol[n][w] = does opt solution to pack items 1..n with weight
 limit w include item n?
        int[][] opt = new int[N+1][W+1];
        boolean[][] sol = new boolean[N+1][W+1];

        for (int n = 1; n <= N; n++) {
            for (int w = 1; w <= W; w++) {

                // don't take item n
                int option1 = opt[n-1][w];

                // take item n
                int option2 = Integer.MIN_VALUE;
                if (weight[n] <= w) option2 = profit[n] + opt[n-1][w-weight[n]];

                 // select better of two options
                opt[n][w] = Math.max(option1, option2);
                sol[n][w] = option2 > option1;
            }
        }

        // determine which items to take
        boolean[] take = new boolean[N+1];
        for (int n = N, w = W; n > 0; n--) {
            if (sol[n][w]) { take[n] = true;  w = w - weight[n]; }
            else           { take[n] = false;                    }
        }

        // print results
        System.out.println("item" + "\t" + "profit" + "\t" + "weight" +
 "\t" + "take");
```

```
        for (int n = 1; n <= N; n++)
            System.out.println(n + "\t" + profit[n] + "\t" + weight[n]
+ "\t" + take[n]);
        System.out.println();
    }
}
```

Outro Exemplo (JAVA)

```
/**
* Entidade para resolução do problema da mochila
* @author Lucas Rocco
*/
public class Mochila {

    private int W; //capacidade da mochila
    private int j; //itens
    private int[] w = {2,1,5,6,8,10,3,5,4,1}; //tamanho dos itens
    private int[] v = {0,100,200,300,400,500,600,700,800,900}; //valor
dos itens
    private int[] x; //mochila

    public Mochila(){

    }
    public Mochila(int capacidade, int itens){
        this.W = capacidade;
        this.j = itens;
    }
    /**
     * Retorna um vetor com o tamanho de cada item disponível
     * @return
     */
    public int[] getw() {
        return w;
    }
    /**
     * Pega o valor definido como tamanho da mochila
     * @return
     */
    public int getW() {
        return W;
    }
    /**
     * Atribui o valor como tamanho da mochila
     * @param W
     */
    public void setW(int W) {
```

```
        this.W = W;
    }
    /**
     * Pega o valor da quantidade de itens
     * @return
     */
    public int getJ() {
        return j;
    }
    /**
     * Atribui quantidade de itens
     * @param j
     */
    public void setJ(int j) {
        this.j = j;
    }
}
```

```
/**
* Utiliza a classe Mochila para composição do algoritmo guloso
* @author Lucas Rocco
*/
public class algGuloso {
    public static void main(String[] args) {
        Mochila mochila = new Mochila(10, 9);
        int W = mochila.getW();
        int j = mochila.getJ();
        int[] w = mochila.getw();
        int[] x = new int[W];

        while (j >= 1 && w[j] <= W) {
            x[j] = 1; //insere o item na mochila
            W = W - w[j]; //decrementa a capacidade
            j--; //decrementa itens
            System.out.println("Peso do item "+j+": "+w[j]);
            System.out.println("Espaço disponivel: "+W);

            //invariante
            if (j >= 1) {
                x[j] = W / w[j]; //avalia se o item cabe na mochila
                for (int k = j - 1; k < j && k > 0; k--) {
                    x[k] = 0;//os espaços não preênchidos recebem 0
                }
            }
        }
        int total = 0;
        for (int m : x) {
            total += m;
```

```
            }
            System.out.println("Total de itens: "+total);
        }
}
```

Referências

[1] http://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/mochila.html

[2] http://en.wikipedia.org/wiki/Knapsack_problem

[3] http://fr.wikipedia.org/wiki/Probl%C3%A8me_du_sac_%C3%A0_dos

[4] http://mathworld.wolfram.com/KnapsackProblem.html

[5] http://homepages.dcc.ufmg.br/~nivio/cursos/pa04/tp2/tp22/tp22.pdf

[6] http://www.ime.usp.br/~regis/Publications/wscad2002.pdf

[7] http://acm.uva.es/archive/nuevoportal/data/problem.php?p=2172

[8] http://oglobo.globo.com/blogs/cat/default.asp?a=5&periodo=200403

Veja também

- Problema de Roteamento de Veículos
- Problema do caixeiro viajante

Fontes e Editores da Página

Problema da mochila *Fonte*: http://pt.wikipedia.org/w/index.php?oldid=22298305 *Contribuidores*: Burmeister, Faga, Hermógenes Teixeira Pinto Filho, Humbertobrandao, Jonas alves, Leandroalex, Lusitana, Marcosrk, Salgueiro, ThiagoRuiz, 18 edições anónimas

Fontes, Licenças e Editores da Imagem

Imagem:Knapsack.svg *Fonte*: http://pt.wikipedia.org/w/index.php?title=Ficheiro:Knapsack.svg *Licença*: Creative Commons Attribution-Sharealike 2.5 *Contribuidores*: User:Dake

Licença