Problema da Mochila 01

Pedro P. L. Sousa[1]

[1]Curso de Bacharelado em Sistemas de Informação – Universidade Federal do Pará (UFPA) – Belém – PA – Brasil

{pedro.sousa@ig.ufpa.br}

1. Estratégias de Resolução

1.1 Estratégia gulosa

É uma técnica de algoritmos para resolver problemas de otimização, ela escolhe a cada iteração, o objeto mais "apetitoso" que vê pela frente. E este objeto passa a fazer parte da solução que o algoritmo constrói.

Ele toma decisões com base na iteração corrente, sem olhar as consequências futuras. Sua característica é jamais voltar atrás em suas decisões, cada escolha é definitiva. Ele escolhe uma decisão ótima local esperando que leve até uma solução ótima global. Possui uma implementação relativamente fácil e é muito rápido, mas não tem prova de correção simples.

Uma das consequências disso é que às vezes ele pode ter um resultado distante do ótimo global.

O algoritmo guloso geralmente é utilizado em problemas de otimização, como: problema da mochila, problema do troco, caixeiro viajante e etc…

Existe um outro tipo de técnica para resolver problemas de otimização, chamada de programação dinâmica.

Diferente da estratégia gulosa essa técnica explora todas as alternativas, podendo prever a próxima decisão e assim ele pode se arrepender e voltar nas decisões. Em consequência disso é um algoritmo muito mais lento que o guloso, mas tem prova de correção simples.

Neste documento será mostrado de forma sucinta algumas das soluções do problema da mochila 01.

1.2 Estratégia dinâmica

Diferente da estratégia gulosa essa técnica explora todas as alternativas, podendo prever a próxima decisão e assim ele pode se arrepender e voltar nas decisões. Em consequência disso é um algoritmo muito mais lento que o guloso, mas tem prova de correção simples.

Neste documento será mostrado de forma sucinta algumas das soluções do problema da mochila 01.

2. Mochila 01

É um problema de otimização combinatória. No cenário proposto, temos uma mochila com uma capacidade determinada onde deve ser colocados dentro dela itens com determinado valor e peso. Sendo que no final a mochila deve estar preenchida com o maior valor possível sem extrapolar a capacidade da mochila. Normalmente é resolvido com programação dinâmica, algoritmos guloso e algoritmos genéticos.

Este problema pode ser aplicado em diversas situações reais como: investimento de capital, corte e empacotamento, carregamento de veículos, orçamento e etc.

No caso da mochila 01, inteira ou binária, o item pode ter 2 estados: estar na solução ou não estar na solução.

3. Problema Proposto e Implementação

O problema proposto foi implementar 2 algoritmos para resolver o problema da mochila inteira, e explicá-los em sala.

Foi disponibilizado um arquivo .txt para ser a entrada do algoritmo. Nesse arquivo, as 2 primeiras linhas são o total de itens disponíveis (3500) e a capacidade da mochila (20000). O restante das linhas contém valor e peso respectivamente.

A implementação dos algoritmos foi feita em JAVA, pois tenho certa familiaridade. Inicialmente o programa lê os dados do arquivo .txt, extraindo a capacidade da mochila e os valores das outras linhas criando objetos “Item” a partir de cada uma delas.

Posteriormente é chamado cada classe de implementação das estratégias: Gulosa e Dinâmica.

3.1 Estratégia gulosa

Abaixo segue sucintamente a lógica da parte gulosa:

- Ordena os valores em ordem decrescente
- pesoacumulado = 0
- Inicia loop em itens
    - se (pesoacumulado + pesodoitem <= capacidade mochila) então
        - pesoacumulado += pesodoitem
        - valordasolucao += valordoitem
        - adiciona item a solucao
    - senao se(pesoacumulado>capacidade)
        - sai
    - fim senaose
    - fim se
- fim loop
- imprime solução
- imprime númerodeintruçoes

3.1 Estratégia dinâmica

Abaixo segue sucintamente a lógica da parte dinâmica:

- i e w inteiros
- n = tamanhodalistadeitens
- K[n+1][capacidade+1]
- inicia loop i de 0 até n
    - inicia loop w de 0 até capacidade
        - se (i ou w = 0) então
            - k[i][w] = 0

            - senao se(peso do item anterior <= w
                - k[i][w] = maximo entre(valor do item anterior + k[i-1][w-pesodoitemanterior, k[i-1][w]
            - senao
                - k[i][w]=k[i-1][w]
            - fim se
    - fim loop
- fim loop
- adiciona k[n][capacidade] a solucao
- imprime solucao