Dener Vieira Ribeiro (3872), Germano Barcelos dos Santos (3873)

Avaliando o problema da Mochila usando algoritmo de combinação

Brasil

2019, 07/11

Dener Vieira Ribeiro (3872), Germano Barcelos dos Santos (3873)

Avaliando o problema da Mochila usando algoritmo de combinação

Prof. Thais R. M. Braga Silva
Algoritmos e Estruturas de Dados I

Universidade Federal de Viçosa - Campus Florestal – UFV-CAF

Ciência da Computação

Graduação

Brasil

2019, 07/11

Sumário

Introdução

O problema da mochila é um problema simples porém de difícil solução ótima. Podemos exemplificar o problema com um jogo, por exemplo: um jogador precisa carregar vários objetos para vendê-los a um comerciante, porém cada objeto é caracterizado por um peso e valor. O objetivo do jogador é maximizar o lucro que ele vai obter quando vender os objetos dado que existe um limite de peso que ele pode carregar.

Como é um problema que tem um alto nível de complexidade, foi nos proposto que abordássemos o problema usando um algoritmo de combinação para determinar a solução ótima. No caso, o algoritmo deveria fazer a combinação de NxN elementos, ou seja, 1x1, 2x2, 3x3 até NxN, depois escolheria a combinação de elementos que teria o maior valor agregado, o critério desempate era o tamanho da combinação. Dentre as especificações do projeto, existia 2 entradas 'C' - capacidade de peso da mochila -, 'N' - quantidade de objetos -, com isso foi nos pedido para determinar a solução do problema da mochila.

1 Desenvolvimento

1.1 Estruturação do Problema

Antes de desenvolver a solução do problema em código, pensamos na organização do código e discutimos o que deveria ser utilizado para melhorar a leitura e o entendimento do código por outras pessoas. Para isso, separamos em ações cada etapa da solução (entrada, modelagem do problema, uso do algoritmo de combinação). Assim, para cada ação há uma biblioteca para ficar mais fácil a manutenção do código.

A entrada do código é feita via arquivo-texto que tem o seguinte padrão: 1º linha é a quantidade N itens; após a 1º linha, há N linhas compostas de tuplas (peso, valor).

Na modelagem, desenvolvemos uma estrutura que chamamos de `tuple_t`; no caso, ela faz o mapeamento da linha do arquivo (peso, valor) para uma variável do tipo `tuple_t`. Fizemos esse mapeamento para ficar mais fácil a manipulação de variáveis, precisamos de um vetor e uma matriz de `tuple_t` para resolver o problema, sem essa estrutura precisaríamos de uma matriz para peso e uma matriz para valor, o que iria ficar mais difícil manter o programa consistente. Se a entrada, porventura, não for mais uma tupla, basta algumas alterações em partes mínimas do código para que esteja ajustado.

Para determinar a solução do problema foi utilizado um algoritmo de combinação como determinado pela especificação do projeto. O algoritmo era composto por vetor de inteiros, porém a função utiliza de troca de posições para calcular as combinações, portanto não foi necessário ajustes grandes no algoritmo para que funcionasse com a estrutura `tuple_t`. Além do uso do algoritmo de combinação, desenvolvemos algumas funções, como somar o peso/valor dos objetos pertencentes a grupo de combinação.

Para calcular a medida do tempo foi utilizado a biblioteca `time.h` e a função de clock do sistema.

1.2 Explicação do Algoritmo

O algoritmo de combinação utilizado está publicado no site (GEEKS,) e funciona da seguinte forma. Cria-se uma árvore de recursão a toda iteração do for, pois há uma chamada da própria função dentro desse for. Quando se cria uma árvore de recursão, alguns dados podem ser calculados mais de uma vez, fazendo com que o algoritmo fique ineficiente o bastante para que entradas muito grandes [1]. Um exemplo clássico de prova é o fibonacci

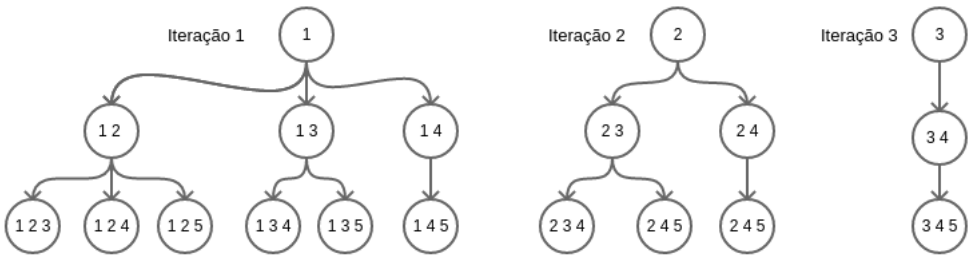
[1] No caso, do problema da mochila, 35 já é uma quantidade de objetos considerável em relação ao tempo de execução para determinar a solução ótima

recursivo, a partir do tamanho da entrada o algoritmo demora para calcular a solução, isso porque se baseia em uma árvore de recursão.

```
for (int i = start; i <= end && end - i + 1 >= r - index; i++) {
  data[index] = arr[i];
  combination_util(arr, data, maxSubsetArr, i+1,
                end, index+1, r, knapsackSize);
}
```

Código: 1.1 – Função que calcula as combinações usando recursão

Por exemplo: se considerarmos que queremos calcular as combinações 3x3 do vetor composto pelos números 1, 2, 3, 4, 5 é gerado uma árvore de recusão como abaixo:



Na primeira iteração o vetor que possui a combinação é preenchido com o número 1, depois, a chamada recursiva acontece criando o nó a esquerda do 1, criando essa árvore de recursão. Os nós que estão nos mesmos níveis são criados a partir de iterações e a seta representa a chamada recursiva.

Em resumo, nosso código calcula NxN combinações a partir de iteração de 1 a N, determina a melhor combinação dentro da iteração gerando uma matriz que terá as melhores combinações. Depois disso, o algoritmo avalia qual é a melhor combinação dentre essa matriz, assim, expondo para o usuário qual é a solução do problema.

1.3 Tempo de Execução

Para medir o tempo do algoritmo utilizamos a biblioteca `time.h` e a função `clock()` usando os ciclos do closk do sitema para medir com precisão quantos segundos o algoritmo gastou determinando a solução do problema[2]. Testamos o algoritmo de acordo com a tabela abaixo:

[2] O tempo de execução medido engloba toda a solução, desde determinar as combinações até achar a melhor entre todas.

Tamanho da Entrada	Tempo (segundos)
10	0.000350
20	0.135317
23	1.192091
25	5.106476
30	193.342603

Tabela 1 – Tempo de Execução do algoritmo

Para estimar os valores pedidos: 50, 80, 100; fizemos uma regressão exponencial no *GeoGebra*, o que gerou a seguinte função:

$$f(x) = 0.000000378424e^{0.6578631760196x} \tag{1.1}$$

Substituindo os valores, na função:

Tamanho da Entrada	segundos	hora	ano(s)
50	$7.29 * 10^7$	$2.02765 * 10^4$	2.3
80	$2.72 * 10^{16}$	$7.5541205 * 10^{12}$	$8.6 * 10^8$
100	$1.41 * 10^{22}$	$3.9 * 10^{18}$	$4.46 * 10^{14}$

Tabela 2 – Tempo de Execução do algoritmo usando a função

Concluindo, para valores maiores que 40 ficaria inviável calcular a solução ótima usando esse algoritmo de combinação.

1.4 Configuração do Hardware

Para executar os testes utilizamos a seguinte configuração:

- Processador: Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz x 4
- Memória RAM: 7887MB
- Sistema Operacional: Unix/Ubuntu
- Caches:
    1. tamanho: 32KB
    2. tamanho: 256KB
    3. tamanho: 4MB

2 Conclusão

Portanto devemos ter cuidado quando temos um problema computacional para ser resolvido. Muitas vezes o problema parece simples porém é difícil determinar a solução ótima do problema. Devemos avaliar o que é mais importante em cada caso, se é ter uma solução correta sempre ou o tempo gasto para encontrar tal solução.

Quando usamos algoritmos de força bruta para resolver problemas, uma pequena variação no tamanho da entrada irá influenciar fortemente o tempo gasto e por isso, devemos utilizar outras técnicas para resolver esses problemas.

O projeto foi de grande importância para entendermos e aplicarmos o conceito de complexidade de algoritmo.

Referências

GEEKS, G. for. *Print all possible combinations of r elements in a given array of size n*. Disponível em: <https://www.geeksforgeeks.org/print-all-possible-combinations-of-r-elements-in-a-given-array-of-size-n/>. Citado na página 4.