

1

Técnicas de Concepção de Algoritmos (1ª parte): algoritmos gananciosos

R. Rossetti, A.P. Rocha, J. Pascoal Faria
CAL, MIEIC, FEUP
Fevereiro de 2014

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2014

2

Algoritmos gananciosos (*greedy algorithms*)

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2014

3

Algoritmos Gananciosos

- ◆ É qualquer algoritmo que aplica uma heurística de solução em que se tenta aplicar uma escolha óptima local em todo e cada estágio da solução.
- ◆ Aplicável a problemas de optimização (*maximização* ou *minimização*)
- ◆ Em diversos problemas, a optimização local garante também a optimização global, permitindo encontrar a solução óptima de forma eficiente
- ◆ **Subestrutura óptima:** um problema tem subestrutura óptima se uma solução óptima p/ problema contém soluções óptimas para os seus subproblemas!

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2013

4

Estratégia Gananciosa

- ◆ Um algoritmo ganancioso funciona em fases. Em cada fase verifica-se a seguinte estratégia:
 1. Pega-se o melhor que se pode obter no exacto momento, sem considerar as consequências futuras ou para o resultado final
 2. Por se ter escolhido um **ótimo local** a cada passo, espera-se por acabar a encontrar um **ótimo global**!
- ◆ Portanto, a escolha que parece ser a melhor no momento é a opção escolhida! Assim,
 - Prova-se que quando há uma escolha a fazer, uma das escolhas possíveis é a “gananciosa.” Portanto, é sempre seguro optar-se por esta escolha
 - Demonstra-se que todos os subproblemas resultantes de uma alternativa gananciosa são vazios, excepto o resultado

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2013

5

Premissas

- ◆ Cinco principais características que suportam essa solução:
 1. Um **conjunto de candidatos**, de onde a solução é criada
 2. Uma **função de selecção**, que escolhe o melhor candidato a ser incluído na solução
 3. Uma **função de viabilidade**, que determina se o candidato poderá ou não fazer parte da solução
 4. Uma **função objectivo**, que atribui um valor a uma solução, ou solução parcial
 5. Uma **função solução**, que determinará se e quando se terá chegado à solução completa do problema

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2013

6

Algoritmo abstracto

- ◆ Inicialmente o conjunto de itens está vazio (i.e. conjunto solução)
- ◆ A cada passo:
 - Um item será adicionado ao conjunto solução, pela função de selecção
 - SE o conjunto solução se tornar inviável, ENTÃO rejeita-se os itens em consideração (não voltando a seleccioná-los)
 - SENÃO o conjunto solução ainda é viável, ENTÃO adiciona-se os itens a serem considerados

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2013

7

Problema do troco



extrair 8 cêntimos
(com nº mínimo de moedas)

Saco / depósito / stock de moedas

extrair(8, {1, 1, 1, 2, 2, 2, 2, 5, 5})
(com nº mínimo de moedas)

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2013

8

Resol. c/ algoritmo ganancioso

extrair(8, {1, 1, 1, 2, 2, 2, 2, 5, 5})

5

extrair(3, {1, 1, 1, 2, 2, 2, 2, 5})

2

extrair(1, {1, 1, 1, 2, 2, 2, 5})

1

extrair(0, {1, 1, 2, 2, 2, 5})

FIM

Escolhe-se a moeda de valor mais alto que não excede o montante em falta (pois com moedas de valor mais alto o nº de moedas necessário será mais baixo)

Sub-problema do mesmo tipo

Dá a solução ótima, se o sistema de moedas tiver sido concebido apropriadamente (caso do euro) e não existirem problemas de stock!

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2013

9

Implementação iterativa

```
static final int moedas[] = {1,2,5,10,20,50,100,200};

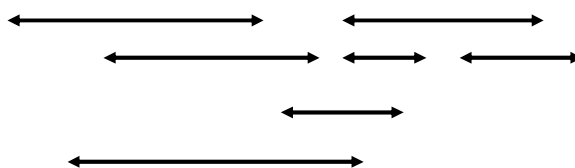
// stock[i] = nº de moedas de valor moedas[i]
public int[] select(int montante, int[] stock) {
    int[] sel = new int[moedas.length];
    for (int i=moedas.length-1; montante>0 && i>=0; i--)
        if (stock[i] > 0 && moedas[i] <= montante) {
            int n_moed=Math.min(stock[i],montante/moedas[i]);
            sel[i] += n_moed;
            montante -= n_moed * moedas[i];
        }
    if (montante > 0)
        return null;
    else
        return sel;
}
```

Técnicas de Conceção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2013

10

Escalonamento de actividades

- ◆ Problema: dado um conjunto de actividades, encontrar um subconjunto com o maior número de actividades não sobrepostas!
- ◆ Input: Conjunto S de n actividades, a_1, a_2, \dots, a_n .
 - s_i = instante de início da actividade i .
 - f_i = instante de fim da actividade i .
- ◆ Output: Subconjunto A de número máximo de actividades compatíveis (i.e. não sobrepostas)



Técnicas de Conceção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2013

11

Escalonamento de actividades

Subestrutura óptima:

- ◆ Assume-se que as actividades estão ordenadas.
 - $f_1 \leq f_2 \leq \dots \leq f_n$.
- ◆ Supondo-se q/ uma solução óptima inclua actividade a_k .
 - Isso gera dois subproblemas:
 - Seleccionar de a_1, \dots, a_{k-1} , actividades compatíveis entre si, e que terminam antes de a_k começar (compatíveis com a_k).
 - Seleccionar de a_{k+1}, \dots, a_n , actividades compatíveis entre si, e que iniciam depois de a_k terminar.
 - A solução para os dois subproblemas deve ser óptima.
 - * Fica como exercício provar esta condição!

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2013

12

Escalonamento de actividades

Abordagem recursiva:

- ◆ Seja S_{ij} = subconjunto de actividades em S q/ iniciam depois de a_i terminar e terminam antes de a_j começar.
- ◆ Subproblemas: Seleccionar o máximo número de actividades mutuamente compatíveis de S_{ij} .
- ◆ Seja $c[i, j]$ = tamanho do subconjunto de tamanho máximo de actividades mutuamente compatíveis em S_{ij} .

Solução
recursiva

$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max_{i < k < j} \{c[i, k] + c[k, j] + 1\} & \text{if } S_{ij} \neq \emptyset \end{cases}$$

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2013

13

Escalonamento de actividades

Abordagem gananciosa:

- ◆ Considerar as actividades numa ordem específica
- ◆ Escolher a “melhor opção” de actividade
- ◆ Descartar todas as actividades incompatíveis com a actividade seleccionada
- ◆ Estratégias
 - “Earliest starting time” -> ascendente em S_i
 - “Earliest finishing time” -> ascendente em F_i
 - “Shortest interval” -> ascendente em $F_i - S_i$
 - “Fewest conflicts” -> para cada actividade, contar o número de conflitos e ordenar segundo este número.

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2013

14

Escalonamento de actividades

Abordagem gananciosa:

$A = \{a_1, a_2, \dots, a_i, \dots, a_n\}$

$R = \emptyset$

While $A \neq \emptyset$

$a \leftarrow a_i \mid \text{earliest finishing time}$

$R \leftarrow R \cup \{a\}$

$A \leftarrow A - \forall a_j \mid a_j \text{ não é compatível com } a_i$

EndWhile

Return R

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2013

15

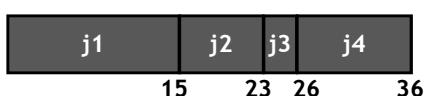
* Escalonamento de actividades

Variação do problema de escalonamento de actividades:

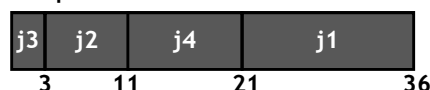
- ◆ Dados: tarefas (*jobs*) e tempo (duração)
- ◆ Objectivo: sequenciar tarefas minimizando o tempo médio de conclusão
- ◆ Método: tarefas mais curtas (que acabam mais cedo) primeiro

Tarefa	Tempo
j1	15
j2	8
j3	3
j4	10

Tempo médio: 25



Tempo médio: 17.75



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2013

16

Outros Exemplos de Problemas

- ◆ Problemas em que garante uma solução óptima:
 - Problema do troco, desde que não haja falta de stock
 - Problema de escalonamento
 - Árvores de expansão mínima
 - Codificação de Huffman
 - Dijkstra, para cálculo do caminho mais curto num grafo
- ◆ Problemas em que não garante uma solução óptima
 - Problema da mochila (mas pode dar boas aproximações ...)

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2013

17

Referências

- ◆ Mark Allen Weiss. Data Structures & Algorithm Analysis in Java. Addison-Wesley, 1999
- ◆ Steven S. Skiena. The Algorithm Design Manual. Springer 1998
- ◆ Robert Sedgewick. Algorithms in C++. Addison-Wesley, 1992
- ◆ Slides de Maria Cristina Ribeiro

Técnicas de Conceção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2013

18

*Demonstração de optimalidade

- ◆ Tarefas: $j_{i1}, j_{i2}, \dots, j_{in}$ (ordenadas)
- ◆ Durações: $t_{i1}, t_{i2}, \dots, t_{in}$
- ◆ Instantes de conclusão: $t_{i1}, t_{i1}+t_{i2}, \dots$
- ◆ Custo total da solução

$$\sum_{k=1}^n (n-k+1) t_{ik} = (n+1) \sum_{k=1}^n t_{ik} - \sum_{k=1}^n k t_{ik}$$

- ◆ Se existe $x > y$ tal que $t_{ix} < t_{iy}$, troca de j_{ix} e j_{iy} diminui custo

Técnicas de Conceção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2013

* Escalonamento de actividades

```
// event-schedule
// schedule as many non-conflicting events as possible
function event-schedule(events array of s[1..n], j[1..n]): set
  if n == 0: return {} fi
  if n == 1: return {events[1]} fi
  let event := events[1]
  let S1 := union(event-schedule(events -
                    set of conflicting events), event)
  let S2 := event-schedule(events - {event})
  if S1.size() >= S2.size():
    return S1
  else
    return S2
  fi
end
```

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2013