1

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

R. Rossetti, A. P. Rocha, L. Ferreira, J. P. Fernandes, F. Ramos, G. Leão
CAL, MIEIC, FEUP

2

Algoritmos gananciosos (*greedy algorithms*)

3

Algoritmos Gananciosos

- É qualquer algoritmo que aplica uma heurística de solução em que se tenta realizar 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

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 para o resultado final
  2. Por se ter escolhido um **óptimo local** a cada passo, espera-se por acabar a encontrar um **óptimo global**!
- Portanto, a opção que parece ser a melhor no momento é a escolhida! Assim,
  - Quando há uma escolha a fazer, uma das opções possíveis é a "gananciosa." Portanto, é sempre seguro optar-se por esta escolha
  - Todos os subproblemas resultantes de uma alternativa gananciosa são vazios, excepto o resultado

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP

5

Premissas

- As 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

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 considerados

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

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, 10})

(com nº mínimo de moedas)

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP

8

Resol. c/ algoritmo ganancioso

extrair(8, {1, 1, 1, 2, 2, 2, 2, 5, 5, **10**})

X 10

extrair(8, {1, 1, 1, 2, 2, 2, 2, 5, **5**, 10})

5

extrair(3, {1, 1, 1, 2, 2, 2, **2**, 5, 10})

2

extrair(1, {1, 1, **1**, 2, 2, 2, 5, 10})

1

FIM! extrair(0, {1, 1, 2, 2, 2, 5, 10})

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 óptima, 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

9

Implementação iterativa (Java)

```
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;
}
```


10

Prova de optimalidade

- Definição: Um sistema de moedas diz-se *canónico*, se o algoritmo ganancioso encontra sempre uma solução ótima para o problema do troco (com stock ilimitado).[1]
- A maioria dos sistemas de moedas são canónicos (USA, EU, etc.).
- Teorema: Sendo $C = \{1, c_2, \cdots, c_n\}$ as denominações do sistema de moedas, se o sistema for não canónico, o menor contra-exemplo situa-se na gama $c_3 + 1 < x < c_{n-1} + c_n$. [1]
  - Logo basta fazer pesquisa exaustiva nesta gama para determinar se é canónico.
- Exemplo: Seja o sistema de moedas $C = \{1, 4, 5\}$.
  - Basta procurar contra-exemplos na gama $6 < x < 9$.
  - No caso $x = 7$, o algoritmo ganancioso dá a solução ótima (5,1,1).
  - No caso $x = 8$, o alg. ganancioso dá {5, 1, 1, 1} mas o ótimo é {4,4}.
  - Logo o sistema não é canónico.

[1] Xuan Cai (2009). "Canonical Coin Systems for CHANGE-MAKING Problems". *Proc. of the Ninth International Conference on Hybrid Intelligent Systems.*

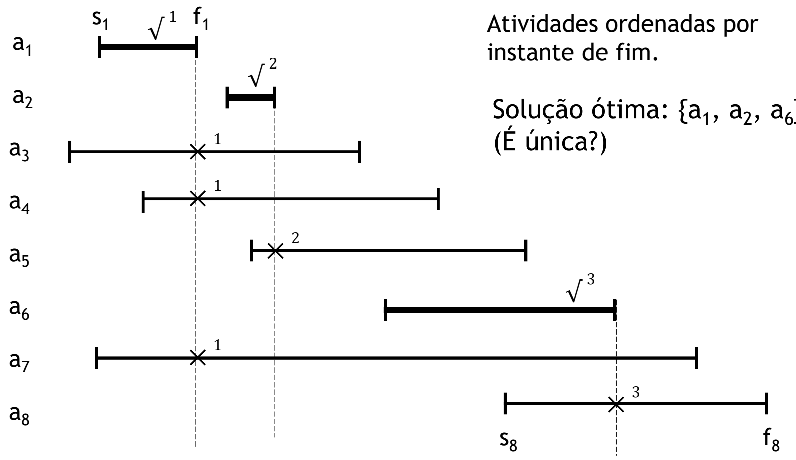
Escalonamento de atividades

- Problema: dado um conjunto de atividades, encontrar um subconjunto com o maior número de atividades não sobrepostas!

- <u>Input:</u> Conjunto ***A*** de *n* atividades, a_1, a_2, …, a_n.
  - s_i = instante de início (*start*) da atividade *i*.
  - f_i = instante de fim (*finish*) da atividade *i*.

- <u>Output:</u> Subconjunto *R* com o número máximo de atividades compatíveis (i.e. não sobrepostas)

Escalonamento de atividades: exemplo



Atividades ordenadas por instante de fim.

Solução ótima: {a_1, a_2, a_6}
(É única?)

13

Escalonamento de atividades: abordagem gananciosa

- Passos:
  - Considerar as atividades numa ordem específica
  - Escolher a "melhor opção" de atividade.
  - Descartar as atividades incompatíveis com a atividade escolhida.
  - Proceder da mesma forma para as atividades restantes.
- Estratégias:
  - "Earliest finishing time" -> ascendente em f_i
  - "Earliest starting time" -> ascendente em s_i
  - "Shortest interval" -> ascendente em $f_i - s_i$
  - "Fewest conflicts" -> para cada atividade, contar o número de conflitos e ordenar segundo este número.

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

14

Escalonamento de atividades:
algoritmo ganancioso por fim mais cedo

Baseado na intuição de que, para realizar o maior nº de atividades sequencialmente, devemos começar pela que termina mais cedo!

Inputs: $A = \{a_1, a_2, \ldots, a_i, \ldots, a_n\}$
$R \leftarrow \emptyset$
While $A \neq \emptyset$
  $a \leftarrow a_i$ | earliest finishing time
  $R \leftarrow R \cup \{a\}$
  $A \leftarrow A \setminus \{a_j \in A \mid a_j \text{ overlaps } a_i\}$ (includes a_i)
EndWhile
Return R

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

15

Escalonamento de atividades:
prova de optimalidade do algoritmo

- No algoritmo e exemplo dados, sejam:
  - *A* – conjunto inicial de atividades
  - *a* – atividade selecionada com fim mais cedo (a_1)
  - *I* – conj. de atividades incompatíveis com *a* ($\{a_3, a_4, a_7\}$)
  - *C* – conj. de atividades restantes ($\{a_2, a_5, a_6, a_8\}$)
- Do conjunto $\{a\} \cup I$, só pode ser selecionada no máximo uma atividade, pois são mutuamente incompatíveis *

  * C/outro critério de ordenação (p.e. início mais cedo), podia não ser assim!
- Desse conjunto, escolhemos uma, que é o máximo possível
- A atividade escolhida (*a*) não tem incompatibilidade com as restantes (*C*), logo a escolha de *a* permite maximizar o nº de atividades que se podem escolher de *C*

16

Escalonamento de atividades:
verificação experimental

- O programa anexo "scheduling.cpp" gera instâncias aleatórias do problema (listas de atividades), aplica vários algoritmos de escalonamento gananciosos e compara com o resultado ótimo (obtido por algoritmo de pesquisa exaustiva)
- Se encontrar contra-exemplos, prova que o algoritmo em causa não garante o ótimo (sem termos de pensar muito na análise teórica …)
- De facto, só os algoritmos gananciosos por fim mais cedo (cf. prova) e início mais tarde (simétrico do anterior) garantem o ótimo!

```
counter-example found for shortest interval: {[10, 14], [12, 18], [5, 12]}; expected result size=2; actual result size=1
counter-example found for earliest start: {[12, 25], [16, 16], [22, 28]}; expected result size=2; actual result size=1
counter-example found for latest finish: {[10, 27], [16, 25], [2, 14]}; expected result size=2; actual result size=1
counter-example found for fewest conflicts: {[5, 12], [89, 127], [25, 32], [80, 117], [31, 45], [48, 67], [44, 52], [68, 125], [54, 81], [47, 79], [44, 70], [27, 83], [27, 91], [11, 81], [5, 107]}; expected result size=5; actual result size=4
no counter-example found for earliest finish with up to 20 activities (1000 samples for each number of activities)
no counter-example found for latest start with up to 20 activities (1000 samples for each number of activities)
```

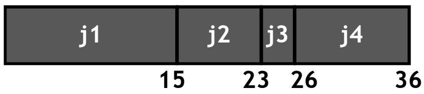
Escalonamento de atividades:
minimizar tempo médio de conclusão

Variação do problema de escalonamento de atividades:

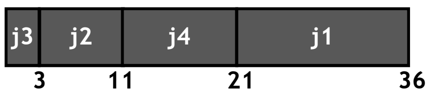
- Dados: tarefas (*jobs*) e tempo (duração)
- Objectivo: sequenciar tarefas minimizando o tempo médio de conclusão
- Método: **tarefas mais curtas primeiro!**

Tarefa	Tempo
j1	15
j2	8
j3	3
j4	10

Tempo médio: 25



Tempo médio: 17.75



Prova de optimalidade

- Tarefas: $j_1, j_2, \ldots, j_n$, ordenadas por ordem de execução
- Durações: $d_1, d_2, \ldots, d_n$
- Instantes de conclusão (fim): $f_1=d_1$, $f_2=d_1+d_2$, …
- Tempo médio de conclusão das tarefas (custo):

$$\frac{\sum_{i=1}^{n} f_i}{n} = \frac{\sum_{i=1}^{n}(n-i+1)d_i}{n} = \frac{(n+1)\sum_{i=1}^{n} d_i - \sum_{i=1}^{n} i\, d_i}{n}$$

- Se existe x > y tal que $d_x < d_y$, troca de j_x e j_y diminui custo da solução
- Assim, custo é minimizado se tarefas forem ordenadas tal que $d_1 \le d_2 \le \ldots \le d_n$

21

Outros Exemplos de Problemas

- Problemas em que se garante uma solução óptima:
  - Problema do troco, desde que não haja falta de stock e o sistema de moedas esteja bem concebido
  - Problema de escalonamento
  - Árvores de expansão mínima (a ver mais tarde)
  - Dijkstra, para encontrar caminho mais curto num grafo (a ver mais tarde)
  - Codificação de Huffman (a ver mais tarde)

- Problemas em que não garante uma solução óptima
  - Problema da mochila (mas pode dar boas aproximações …)

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP

22

Referências

- T.H. Cormen, C. E. Leiserson, R. L. Rivest , C. Stein. Introduction to Algorithms, 3rd Edition. MIT Press, 2009
  - Capítulo 16 (Gready Algorithms)
- 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

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP