Algoritmos gulosos

Esta página é uma introdução à *estratégia gulosa* (= gananciosa = *greedy*) de solução de problemas. Poderíamos também apresentar o assunto como *paradigma guloso* de concepção de algoritmos.

Para resolver um problema, um algoritmo guloso escolhe, em cada iteração, o objeto mais "apetitoso" que vê pela frente. (A definição de "apetitoso" é estabelecida a priori.) O objeto escolhido passa a fazer parte da solução que o algoritmo constrói.

Um algoritmo guloso é "míope": ele toma decisões com base nas informações disponíveis na iteração corrente, sem olhar as consequências que essas decisões terão no futuro. Um algoritmo guloso jamais se arrepende ou volta atrás: as escolhas que faz em cada iteração são definitivas.

Embora algoritmos gulosos pareçam óbviamente corretos, a prova de sua correção é, em geral, muito sutil. Para compensar, algoritmos gulosos são muito rápidos e eficientes. (É preciso dizer, entretanto, que os problemas que admitem soluções gulosas são um tanto raros.)

Veja o capítulo 16 do CLRS. Veja também o verbete *Greedy algorithm* na Wikipedia.

Máximo *versus* maximal

Suponha que **S** é uma coleção de subconjuntos de $\{1,\ldots,n\}$. Um elemento X de **S** é *máximo* se não existe Y em **S** tal que $|Y| > |X|$, ou seja, se nenhum elemento de **S** é maior que X. Um elemento X de **S** é *maximal* se não existe Y em **S** tal que $Y \supset X$, ou seja, se nenhum elemento de **S** é superconjunto próprio de X. É evidente que todo máximo é maximal. Mas a recíproca longe está de ser verdadeira.

EXEMPLO: Sejam {1,2}, {2,3}, {4,5}, {1,2,3}, {1,2,4}, {2,3,4,5} e {1,3,4,5} os elementos da coleção **S**. Então **S** tem dois elementos máximos: {2,3,4,5} e {1,3,4,5}. Há quatro elementos maximais: {1,2,3}, {1,2,4}, {2,3,4,5} e {1,3,4,5}.

Procurar por um elemento máximo de **S** é, em geral, uma tarefa computacionalmente pesada (pois exige que todos os elementos de **S** sejam examinados). Já encontrar um elemento maximal de **S** é muito fácil. Basta aplicar o seguinte algoritmo guloso, que abocanha o primeiro Y aceitável que vê pela frente:

```
escolha algum X em S
enquanto X ⊂ Y para algum Y em S
    faça  X ← Y
devolva X
```

É ainda mais fácil encontrar um elemento maximal se a coleção **S** tiver caráter *hereditário*, ou seja, se tiver a seguinte propriedade: para cada X em **S**, todos os subconjuntos de X também estão em **S**. Nesse caso, basta executar o seguinte algoritmo guloso:

```
X ← { }
para cada k em {1,...,n} faça
    Y ← X ∪ {k}
    se Y está em S
        então  X ← Y
devolva X
```

Exercícios

1. Refaça a discussão acima com *minimal* e *mínimo* no lugar de *maximal* e *máximo* respectivamente.

Problemas de otimização combinatória

"A *greedy algorithm* starts with a solution to a very small subproblem
and augments it successively to a solution for the big problem.
The augmentation is done in a 'greedy' fashion, that is, paying attention to short-term or local gain,
without regard to whether it will lead to a good long-term or global solution.
As in real life, greedy algorithms sometimes lead to the best solution, sometimes lead to pretty good solutions,
and sometimes lead to lousy solutions.
The trick is to determine when to be greedy."

[Most greedy algorithms are deceivingly simple.]
"One thing you will notice about greedy algorithms is that they are usually easy to design,
easy to implement, easy to analyze, and they are very fast,
but they are almost always difficult to prove correct."

— Ian Parberry, *Problems on Algorithms*

Um problema de *otimização combinatória* consiste em encontrar um elemento de valor ótimo num conjunto finito. Dependendo do problema, *ótimo* pode significar *máximo* ou *mínimo*. No caso da maximização, por exemplo, a estratégia de um algoritmo guloso é calcular um máximo "local" na esperança de encontrar um máximo "global". (É como um montanhista que anda sempre "para cima" na esperança de assim chegar ao pico mais alto da montanha.)

Suponha que n é um número grande e que **S** é uma enorme coleção de subconjuntos de $\{1,\ldots,n\}$. (Nos casos concretos, a coleção **S** é dada de maneira indireta, e não como uma lista explícita de conjuntos.) Queremos encontar um elemento máximo de **S**. Se todos os elementos maximais de **S** têm o mesmo tamanho, então o problema admite solução gulosa, pois basta encontrar um elemento maximal.

Mesmo que nem todo maximal seja máximo, entretanto, um algoritmo de caráter guloso pode, às vezes, ter sucesso. Para certas coleções **S**, o algoritmo guloso abaixo produz um conjunto máximo se o universo $\{1,\ldots,n\}$ for examinado numa certa ordem "mágica":

```
X ← { }
para cada k em {1,…,n} na ordem "mágica" faça
    Y ← X ∪ {k}
    se Y está em S
        então  X ← Y
devolva X
```

Suponhamos, por exemplo, que a ordem "mágica" para um determinado **S** é a ordem decrescente. Para mostrar que o algoritmo produz um elemento máximo de **S** é preciso verificar que

1. algum elemento máximo de **S** contém n e
2. se X é máximo e contém n então $X-\{n\}$ é um elemento máximo de $\mathbf{S}/n$, onde $\mathbf{S}/n$ é a coleção de todos os conjuntos da forma $S-\{n\}$ tais que S está em **S** e contém n.

No CLRS, a propriedade 1 é denominada Propriedade da Escolha Gulosa e a propriedade 2 é denominada Propriedade da Subestrutura Ótima.

Exemplos de algoritmos gulosos

Eis uma pequena lista de problemas admitem soluções gulosas (ou seja, podem ser resolvidos por algoritmos gulosos):

- problema da mochila fracionária

- problema do escalonamento de intervalos
- problema da árvore de Huffman
- problema da arborescência maximal de peso mínimo num digrafo simétrico (algoritmos de Prim e Kruskal)

Exercícios

1. [Problema do pen drive] Tenho um grande número de arquivos digitais no meu computador. Cada arquivo ocupa um certo número de kilobytes. Quero gravar o maior número possível de arquivos num pendralvo — oops, pen drive — com capacidade para c kilobytes. O problema pode ser modelado assim: dados números naturais $p_1, p_2, \ldots, p_n$ e c, encontrar o maior subconjunto X de $\{1,2,\ldots,n\}$ que satisfaça a restrição $\sum_{i\in X} p_i \le c$. Mostre que um algoritmo guloso apropriado resolve o problema.

2. Prove ou desprove as seguintes afirmações sobre o problema do pen drive: (1) se p_i é mínimo então i faz parte de toda solução do problema; (2) se p_i é mínimo então i faz parte de alguma solução do problema.

3. É dado um conjunto $\{a_1,a_2,\ldots,a_n\}$ de números naturais. Quero encontrar uma permutação $i_1,i_2,\ldots,i_n$ de $1,2,\ldots,n$ que minimize a soma $1a_{i_1} + 2a_{i_2} + \ldots + (n-1)a_{i_{n-1}} + na_{i_n}$.

4. [Pares de livros] Suponha dado um conjunto de livros numerados de 1 a n. Suponha que cada livro i tem um peso $p[i]$ que é maior que 0 e menor que 1. Problema: Acondicionar os livros no menor número possível de envelopes de modo que
   - cada envelope tenha ≤ 2 livros e
   - o peso do conteúdo de cada envelope seja ≤ 1.

   Escreva um algoritmo guloso que resolva o problema em tempo $O(n \log n)$ no pior caso. Aplique seu algoritmo a um exemplo interessante. Mostre que seu algoritmo está correto.

5. [CLRS 16.2-4] Quero dirigir um carro de uma cidade A a uma cidade B ao longo de uma redovia. O tanque de combustível do carro tem capacidade suficiente para cobrir n quilometros. O mapa da rodovia indica a localização dos postos de combustível. Dê um algoritmo que garanta uma viagem com número mínimo de reabastecimentos.

6. [CLR 17-3, CLRS 16-4: Escalonamento de tarefas] Seja $1,\ldots,n$ um conjunto de *tarefas*. Cada tarefa consome um dia de trabalho. Durante um dia de trabalho somente uma das tarefas pode ser executada. Os dias de trabalho são numerados de 1 a n. A cada tarefa t está associado um *prazo* $d[t]$: a tarefa deveria ser executada em algum dia do intervalo $1..d[t]$. A cada tarefa t está associada uma *multa* $m[t] \ge 0$. Se uma dada tarefa t é executada depois do prazo $d[t]$, sou obrigado a pagar a multa $m[t]$ (mas a multa não depende do número de dias de atraso).
   Problema: Programar as tarefas (ou seja, estabelecer uma bijeção entre as tarefas e os dias de trabalho) de modo a minimizar a multa total.

   Escreva um algoritmo guloso para resolver o problema. Prove que seu algoritmo está correto. Analise o consumo de tempo.

7. [CLRS 16.2-7] Suponha que A e B são dois conjuntos, cada um contendo n números naturais. Problema: encontrar uma ordenação $a_1,a_2,\ldots,a_n$ de A e uma ordenação $b_1,b_2,\ldots,b_n$ de B que maximize o produto $\prod_{1\le i\le n} a_i^{b_i}$. Dê um algoritmo que resolva o problema. Qual o consumo de tempo do seu algoritmo?

Gula *versus* programação dinâmica

Às vezes é difícil distinguir um algoritmo guloso de um algoritmo de programação dinâmica. A seguinte lista grosseira de características pode ajudar. Um algoritmo guloso

- abocanha a alternativa mais promissora (sem explorar as outras),
- é muito rápido,
- nunca se arrepende de uma decisão já tomada,
- não tem prova de correção simples.

Um algoritmo de programação dinâmica

- explora todas as alternativa (mas faz isso de maneira eficiente),
- é um tanto lento,
- a cada iteração pode se arrepender de decisões tomadas anteriormente (ou seja, pode rever o "ótimo corrente"),
- tem prova de correção simples.

---

http://www.ime.usp.br/~pf/analise_de_algoritmos/
Last modified: Mon Apr 13 07:08:05 BRT 2015
Paulo Feofiloff
Departamento de Ciência da Computação
Instituto de Matemática e Estatística da USP