Algoritmos Gulosos

Prof. Leandro Fernandes

Baseado no material de Paulo Feofiloff (IME - USP) e no livro
"Algoritmos: Teoria e Prática" - Cormen, Leiserson, Rivest, C. Stein

Algoritmos Gulosos

- Um algoritmo guloso (ou *Greedy Algorithm*) é uma técnica de otimização que sempre realiza a escolha que parece ser a melhor no momento, na esperança de que esta escolha leve até a solução ótima global.
  - Para resolver um problema um algoritmo guloso escolhe, em cada iteração, o objeto mais "apetitoso" a sua frente; e este passa a fazer parte da solução que o algoritmo constrói.

Especificidades

Em geral, os algoritmos gulosos apresentam 5 componentes:

- Um **conjunto candidato**, a partir do qual a solução é criada;
- Uma **função de seleção**, que escolhe qual o melhor candidato a ser adicionado a solução;
- Uma **função de viabilidade**, que é usada para determinar se um candidato pode ser usado para contribuir para a solução;
- Uma **função objetivo**, que atribui um valor para a solução ou para a solução parcial; e
- A **função solução**, que indica quando uma solução completa foi descoberta.

Propriedades do problema

- A maioria dos problemas em que esta técnica funciona apresentam duas características:
- **Propriedade da Escolha Gulosa**
  - Podemos fazer qualquer escolha que nos pareça melhor em um momento e então resolver os subproblemas que surgirem depois.
  - A escolha realizada por um algoritmo guloso pode depender das escolhas feitas até o momento, mas não das escolhas futuras ou de todas as soluções dos subproblemas. Ele realiza escolhas gulosas sucessivamente, reduzindo cada dado problema em um menor e nunca reconsidera suas escolhas.
- **Subestrutura ótima**
  - *"Um problema apresenta uma subestrutura ótima se uma solução ótima para o problema contém a solução ótima dos subproblemas".*

Guloso *vs* Programação Dinâmica

Às vezes é difícil distinguir um algoritmo guloso de um algoritmo de programação dinâmica, mas podemos notar alguns pontos:

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.

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.

Resumo das características

- **Imediatista**: toma decisões com base nas informações disponíveis na iteração corrente, sem olhar as consequências que essas decisões terão no futuro.
- **Jamais se arrepende ou volta atrás**: as escolhas que faz em cada iteração são definitivas.
- Embora algoritmos gulosos pareçam obviamente corretos, a **prova de sua correção** é, em geral, muito **sutil**.
- Para compensar, são muito **rápidos** e **eficientes**.
- Vale ressaltar que os problemas que admitem soluções gulosas são um tanto raros.

Problemas com Algoritmos Gulosos

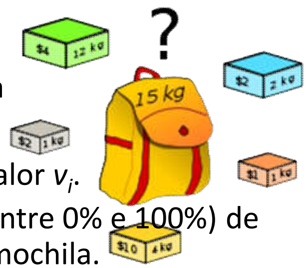
- Mochila fracionária
- Escalonamento de intervalos
- Grafos (Coloração e arborescência)
  - Os algoritmos de Kruskal, Prim e de Expansão de Árvore Mínima são gulosos.
- Roteamento em redes
- Árvore de Huffman

(Knapsack Problem)

PROBLEMA DA MOCHILA FRACIONÁRIA

Problema da Mochila fracionária

- Imagine que tenho n objetos que gostaria de colocar numa mochila de capacidade c.
- Cada objeto i tem peso p_i e valor v_i.
- Posso escolher uma fração (entre 0% e 100%) de cada objeto para colocar na mochila.



Problema: Deseja-se fazer isso respeitando a capacidade da mochila e maximizando o seu valor.

Problema da Mochila fracionária

Exemplo:

- Suponha $c = 50$ e $n = 4$.
- A tabela abaixo fornece os valores de p e v.

p	40	30	20	10	20
v	840	600	400	100	300
x	1	1/3	0	0	0

O valor dessa solução é $x \cdot v = 1040$

O algoritmo exige que os dados estejam em ordem crescente de valor específico:

$v_1/p_1 \leq v_2/p_2 \leq \ldots \leq v_n/p_n$

(Isto é, considerar o valor por unidade de peso)

Problema da Mochila fracionária

Exemplo:

- Suponha $c = 50$ e $n = 4$.
- A tabela abaixo fornece os valores de p e v.

p	40	30	20	10	20
v	840	600	400	100	300
x	1	1/3	0	0	0

O valor dessa solução é $x \cdot v = 1040$

```
MOCHILA-FRACIONÁRIA (p, v, n, c)
j ← n
enquanto j ≥ 1 e p_j ≤ c faça
   x_j ← 1
   c ← c − p_j
   j ← j − 1
se j ≥ 1 então
   x_j ← c/p_j
   para i ← j−1 decres. até 1 faça
      x_i ← 0
devolva x
```

Considerações e comentários

- O algoritmo é guloso porque, em cada iteração, abocanha o objeto de maior valor específico dentre os disponíveis, sem se preocupar com o que vai acontecer depois e jamais se arrepende do valor atribuído a um componente de x.
- É nesta ordem "mágica" que está o segredo do funcionamento do algoritmo.

PROBLEMA DO ESCALONAMENTO DE INTERVALOS

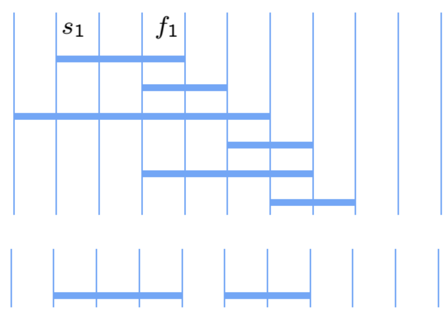
Escalonamento de Intervalos

- Um *intervalo* é um conjunto de números naturais consecutivos, assim um intervalo como { s, s+1, ..., f−1, f } será denotado por (s, f).
  - O primeiro número do par é o *início* do intervalo e o segundo é o *término*.
- **Problema:** Dados intervalos $[s_1, f_1), \ldots, [s_n, f_n)$ encontrar uma coleção máxima de intervalos disjuntos dois a dois.

Escalonamento de Intervalos

Instância:

Solução:



Obs.: Os intervalos i e j são disjuntos se $f_i \leq s_j$ ou $f_j \leq s_i$

A estrutura "gulosa" do problema

- Propriedade da **escolha gulosa**:
  - se f_m é mínimo então m está em alguma solução ótima
- Propriedade da **subestrutura ótima**:
  - se A é solução ótima e $m \in$ A é tal que f_m é mínimo então A − {m} é solução ótima de $\{ j : s_j \geq f_m \}$ (todos os intervalos "posteriores" a f_m)
- Propriedade mais geral da subestrutura ótima:
  - se A é solução ótima e $m \in$ A então A − {m} é solução ótima de $\{ i : f_i \leq s_m \} \cup \{ j : s_j \geq f_m \}$ (intervalos "anteriores" a s_m ou "posteriores" a f_m)

Algoritmo guloso, versão recursiva:

dado L ⊆ {1, . . . , n}
o algoritmo devolve subconjunto máximo A de L tal que os intervalos $[s_a, f_a)$, $a \in$ A, são disjuntos dois a dois.

INTERV-DISJ-REC (s, f, L) ⇒ supõe L ≠ 0
se |L| = 1
então devolva L
senão escolha m em L tal que $f_m = \min\{f_i : i \in L\}$
J ← { j ∈ L : $s_j \geq f_m$ }
A ← INTERV-DISJ-REC (s, f, J)
devolva {m} ∪ A

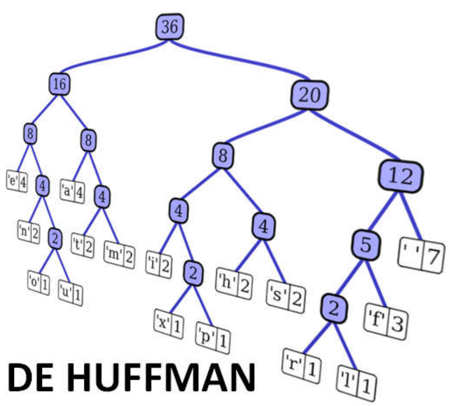
prova da correção: escolha gulosa + subestrutura ótima

Algoritmo guloso, versão iterativa:

supõe $f_1 \leq f_2 \leq \ldots \leq f_n$

INTERVALOS-DISJUNTOS (s, f, n) ⇒ supõe n ≥ 1
A ← {1}
i ← 1
para m ← 2 até n faça
se $s_m \geq f_i$
então A ← A ∪ {m}
i ← m
devolva A

Desempenho:
- tamanho de instância: n
- consumo de tempo: $\Theta(n)$



ALGORITMO DE HUFFMAN PARA COMPRESSÃO DE DADOS

Algoritmo de *Huffman*

- O algoritmo de *Huffman* recebe um fluxo de bits, correspondente aos caracteres, e devolve um fluxo de bits comprimido que representa o fluxo original.
  - Em geral, o fluxo comprimido é mais curto que o original. Reduções no tamanho dos arquivos dependem das características dos dados.
  - Valores típicos oscilam entre 20 e 90%.
- **Problema:** Encontrar uma codificação de caracteres em sequencia de bits que minimize o comprimento do arquivo codificado.

A ideia por trás do algoritmo

- O fluxo de bits original é lido de 8 em 8 bits, como se fosse um fluxo de caracteres:

01000001	01000010	01010010	01000001
A	B	R	A

- Tudo se passa como se o algoritmo transformasse uma *string* numa cadeia de bits.
- Cada caractere da *string* original é convertido em uma pequena cadeia de bits, que é o seu *código*.
  - Por exemplo, B é convertido em 111.

Algoritmo de *Huffman*

Ideia do algoritmo de *Huffman*:

- Usar códigos curtos para os caracteres que ocorrem com frequência e deixar os códigos mais longos para os caracteres mais raros.
- Os códigos são, portanto, de *comprimento variável*.

caractere *c*	*a*	*b*	*c*	*d*	*e*	*f*
frequência *f*[*c*]	45	13	12	16	9	5
cód. compr. fixo	000	001	010	011	100	101
cód. compr. variável	0	101	100	111	1101	1100

- Se texto original tem 100 caracteres, o texto codificado terá:
  - 300 bits se código de comprimento fixo
  - 224 bits se código de comprimento variável

Algoritmo de *Huffman*

- A tabela de código é representada por árvore binária cheia.
- Códigos: "0" se desce para esquerda, "1" se desce para direita.
- $d(c)$ representa a profundidade da folha *c* na árvore e, consequentemente, o número de bits do caractere.
- Custo da árvore: $\sum_c f[c]d(c)$
  - Assim o texto codificado terá $\sum_c f[c]d(c)$ bits

Problema reformulado:

- Encontrar árvore de custo mínimo!

Árvores binárias cheias

Definição (mais abstrata):

- Uma árvore binária cheia, ou *ábch*, sobre o conjunto {1, 2, ..., *n*} é qualquer coleção B de subconjuntos de {1, 2, ..., *n*} que tenha as seguintes propriedades:
  - para cada X e cada Y em B, tem-se X ∩ Y = { } ou X ⊆ Y ou X ⊇ Y;
  - {1,2,...,n} está em B;
  - { } não está em B;
  - todo elemento não minimal de B é união de dois outros elementos de B.

- Os elementos de B são chamados nós.
  - O nó {1,2,...,n} é a raiz de B.
  - Os nós minimais são folhas e os demais nós são internos.

Árvores binárias cheias

- Se X, Y e X ∪ Y são elementos de B, dizemos que:
  - X e Y são os **filhos** de X ∪ Y; e também
  - que X ∪ Y é o **pai** de X e de Y.
- Um **ancestral** de X é qualquer nó W tal que W ⊇ X.
- A **profundidade** de um nó X é o número de ancestrais de X que são diferentes de X.
  - A profundidade da raiz, por exemplo, é 0.

Árvores binárias cheias

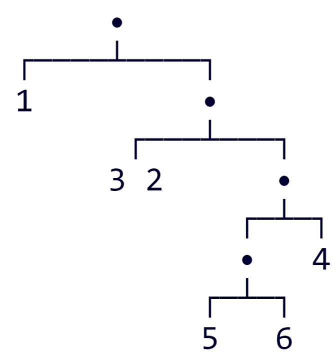
- Toda *ábch* com mais de dois nós tem pelo menos duas folhas X e Y tais que X ∪ Y é também um nó. Dizemos que X e Y são **folhas irmãs**.
- Se X e Y são folhas irmãs de uma *ábch* B então B − {X,Y} também é uma *ábch*.
- O conjunto de todas as folhas de uma *ábch* é uma partição de {1, 2, ..., *n*}.
  - Reciprocamente, dada qualquer partição F de {1, 2, ..., *n*} existe pelo menos uma *ábch* que tem F como conjunto de folhas.
- Diremos que uma *ábch* sobre {1, 2, ..., *n*} tem folhas unitárias se suas folhas são {1}, {2}, ..., {*n*}.

Árvores binárias cheias

EXEMPLO:
{ {1}, {2,3}, {4}, {5}, {6}, {5,6}, {4,5,6}, {2,3,4,5,6}, {1,2,3,4,5,6} }
é uma *ábch* com:

- raiz {1,2,3,4,5,6} e
- folhas {1}, {2,3}, {4}, {5} e {6}.
- O nó {4,5,6} é pai dos nós {4} e {5,6}.
- As folhas {5} e {6} são irmãs; ambas têm profundidade 4.

Essa *ábch* pode ser representada graficamente conforme a figura dada ao lado:



O custo de uma *ábch*

- Dados números $p_1, \ldots, p_n$ e uma parte X de {1, 2, ..., *n*}, assumiremos que $p(X)$ é a soma de todos os p_i com i em X.
  - Diremos que $p(X)$ é o peso de X, dado por $\mathrm{p(X)} = \sum_{i \in X} p(i)$.
- Seja B uma *ábch* sobre {1, 2, ..., *n*}, o custo de B em relação a uma família $p_1, \ldots, p_n$ de números é a soma dos pesos de todos os nós exceto a raiz:

$$\mathrm{custo(B,p)} = \sum_{X \in B - \{\{1,2,\ldots,n\}\}} p(X)$$

- O custo de B pode ser calculado a partir dos pesos de suas folhas:

$$\mathrm{custo(B,p)} = \sum_{f \in F} p(f) d(f)$$

sendo F o conjunto das folhas de B e $d(f)$ a profundidade da folha f.

A estrutura recursiva do problema

- Nosso problema tem uma propriedade estrutural simples e natural.
- Se B é uma *ábch* ótima para uma família de pesos $p_1, \dots, p_n$, supondo X e Y duas folhas irmãs de B e que B' a *ábch* B − {X,Y}, temos que B' também é ótima.

Propriedade recursiva:

- A *ábch* B' é ótima para $p_1, \dots, p_n$.

A estrutura gulosa do problema

- Podemos supor, sem perder generalidade, que em qualquer solução do nosso problema as duas folhas mais leves são também as mais profundas.

Propriedade gulosa:

- Dados números $p_1, \dots, p_n$ e uma partição F de {1, 2, …, n}, sejam X e Y dois elementos de F tais que $p(X) \leq p(\underline{Y})$ e $p(Y) \leq p(U)$ para todo U em F − {X, Y}.
- Então existe uma *ábch* ótima B para $p_1, \dots, p_n$, com folhas F, na qual X e Y são folhas irmãs de profundidade máxima.

O Algoritmo de *Huffman*

```
HUFFMAN (n, p1,…,pn, F)
se |F| = 1
   então devolva F e pare
   senão seja X um elemento de F que minimiza p(X)
         F ← F − {X}
         seja Y um elemento de F que minimiza p(Y)
         F ← F − {Y}
         F ← F ∪ {X ∪ Y}
         B ← HUFFMAN (n, p1,…,pn, F)
         devolva B ∪ {X,Y}
```