Aula 29: Algoritmos gulosos

Greedy

David Déharbe
Programa de Pós-graduação em Sistemas e Computação
Universidade Federal do Rio Grande do Norte
Centro de Ciências Exatas e da Terra
Departamento de Informática e Matemática Aplicada

Download me from `http://DavidDeharbe.github.io`

Plano

Introdução

Um exemplo simples

Considerações sobre algoritmos gulosos

Exemplo avançado: códigos de Huffman

Referência: Cormen, cap 17.

Introdução

- Otimização
- Alternativa à programação dinâmica
- Em geral, mais simples
- Nem sempre possível
- Exemplos:
  - algoritmo de menor caminho (Dijkstra)
  - árvore geradora mínima

Princípios

- a solução é calculada de forma incremental
- cada incremento é realizado com a melhor escolha naquele momento
- a sequência de decisões localmente ótimas é globalmente ótima
- adequado para problemas exibindo certas propriedades
  - sub-estrutura ótima
  - propriedade da escolha gulosa

Problema de escalonamento de atividades

Exemplo introdutório

- Temos n atividades
- As atividades necessitam de um recurso compartilhado
- Cada atividade i tem uma hora de início s_i e uma hora de término f_i
- A cada t, $s_i \leq t < f_i$, a atividade i precisa do acesso exclusivo ao recurso para funcionar.
- Problema:
  - Escalonar a maior quantidade possível de atividades.

GREEDY-SCHEDULER(s, f)

// $f_1 \leq f_2 \leq \ldots f_n$

$n = s.length$

// A: atividades escalonadas

$A = \{1\}$ // atividade terminando mais cedo é escalonada

// j: última atividade escalonada

$j = 1$

for $i = 2$ **to** n // escolhe se a atividade i é escalonada

    if $s_i \geq f_j$

        $A = A \cup \{i\}$

        $j = i$

return A

Ilustração

i	1	2	3	4	5	6	7	8	9	10	11
s	1	3	0	5	3	5	6	8	8	2	12
f	4	5	6	7	8	9	10	11	12	13	14

Ilustração

i	1	2	3	4	5	6	7	8	9	10	11
s	1	3	0	5	3	5	6	8	8	2	12
f	4	5	6	7	8	9	10	11	12	13	14

Por qual razão este algoritmo é qualificado de guloso?

Complexidade

Escalonamento de atividades

```
GREEDY-SCHEDULER(s, f)
n = s.length
A = {1}
j = 1
for i = 2 to n // Θ(n) repetições
    if s_i ≥ f_j // O(1) por repetição
        A = A ∪ {i}
        j = i
return A
```

Complexidade

Escalonamento de atividades

GREEDY-SCHEDULER(s, f)

$n = s.length$
$A = \{1\}$
$j = 1$
for $i = 2$ **to** n // $\Theta(n)$ repetições
  if $s_i \geq f_j$ // $O(1)$ por repetição
    $A = A \cup \{i\}$
    $j = i$
return A

$\Theta(n)$

- O algoritmo sempre tenta maximizar o tempo que o recurso ficará disponível
- Será que é correto? Justifique.

Teorema

O algoritmo GREEDY-SCHEDULER *produz soluções de tamanho máximo para o problema do escalonamento de atividades.*

Por indução.

- Seja $A = \{A_1, \ldots A_k\}$ uma solução ótima, ordenada por tempo de término crescente.
- Queremos mostrar que existe um escalonamento ótimo com 1
- Se $A_1 \neq 1$, seja $B = A - \{A_1\} \cup \{1\}$
- B é um escalonamento correto das atividades, pois $f_1 \leq f_{A_1}$
- Logo B é um escalonamento ótimo que começa com 1.
- Sempre existe um escalonamento ótimo começando por 1.
- Uma vez 1 escolhido, deve-se escalonar a maior quantidade possível de tarefas a partir de f_1.
  - Instância menor do mesmo problema!
  - Podemos repetir a escolha gulosa.
  - Por indução a solução é correta.

Exercícios

1. Desenvolver um algoritmo para o problema do escalonamento da atividades utilizando programação dinâmica.
   - Dica: o algoritmo deve calcular m_i, para $1 \leq i \leq n$, onde m_i é a solução para as atividades $\{1, 2, \ldots i\}$.
   - Compare a complexidade do algoritmo obtido com a de GREEDY-SCHEDULE
2. A abordagem gulosa nem sempre funciona:
   - Encontre um exemplo que mostra por que não funciona se for escolhida a atividade de menor duração no problema do escalonamento de atividades.
   - Encontre um exemplo que mostra por que não funciona se for escolhida a atividade com menos conflitos no problema do escalonamento de atividades.

Estratégias gulosas

- Como determinar se uma estratégia gulosa é correta?
- Deve-se demonstrar que uma solução globalmente ótima pode ser calculada a partir de soluções localmente ótimas
  - Exemplo: o teorema sobre GREEDY-SCHEDULER
  - Considera uma solução globalmente ótima (A).
  - Mostra que pode se construir uma solução globalmente ótima a partir da escolha gulosa ($A - \{A_1\} \cup \{1\}$).
  - Mostrar por indução que a escolha gulosa pode ser repetida.
  - uma escolha localmente ótima produz uma instância menor do mesmo problema de otimização.
  - sub-estrutura ótima

- Compartilham a propriedade de sub-estrutura ótima
- Porém não são equivalentes!
- Exemplos:
  - problema da mochila 0-1 (ver aula 27)
  - problema da mochila: itens tem pesos e valores, mas é possível fracionar cada item.
- Uma instância:
- $K = 50$, $v_1 = 60$, $w_1 = 10$, $v_2 = 100$, $w_2 = 20$, $v_3 = 120$, $w_1 = 30$,
  - Qual a solução ótima no caso da mochila $0 - 1$?
  - Qual a solução ótima no caso da mochila?
- Qual pode ser resolvido apenas com programação dinâmica, por que?

- Compartilham a propriedade de sub-estrutura ótima
- Porém não são equivalentes!
- Exemplos:
  - problema da mochila 0-1 (ver aula 27)
  - problema da mochila: itens tem pesos e valores, mas é possível fracionar cada item.
- Uma instância:
- $K = 50$, $v_1 = 60$, $w_1 = 10$, $v_2 = 100$, $w_2 = 20$, $v_3 = 120$, $w_1 = 30$,
  - Qual a solução ótima no caso da mochila $0 - 1$?
  - Qual a solução ótima no caso da mochila?
- Qual pode ser resolvido apenas com programação dinâmica, por que?
- Como resolver o problema da mochila com um algoritmo guloso?

Exercícios

1. Demonstrar que o problema da mochila pode ser resolvido de forma gulosa.
2. Projetar um algoritmo guloso para solucionar o problema da mochila.

- Utilizado para compactar dados
- Dados: sequências de símbolos
- Necessita conhecer a frequência de cada símbolo
- Atribui código de tamanho menor para símbolos mais frequentes
- Alocação de código: algoritmo guloso

Ilustração

Um exemplo avançado

- arquivo de tamanho 100.000 símbolos
- seis símbolos diferentes
- hipótese: codificação binária
- 3 bits por símbolo
- custo total (sem compactação): 300 kbits

Ilustração

Um exemplo avançado

- arquivo de tamanho 100.000 símbolos
- seis símbolos diferentes
- hipótese: codificação binária
- 3 bits por símbolo
- custo total (sem compactação): 300 kbits

	a	b	c	d	e	f
frequência	45	13	12	16	9	5
código de tamanho fixo	000	001	010	011	100	101
código de tamanho variável	0	101	100	111	1101	1100

Custo com código de tamanho variável: 224 kbits ($\approx$ 25%)

Códigos livres de prefixo

	a	b	c	d	e	f
código de tamanho variável	0	101	100	111	1101	1100

Definição (Código livre de prefixo)

Um código é livre de préfixo se, para qualquer par de símbolos s_1, s_2 a codificação de s_1 não é prefixo da codificação de s_2.

- O código na tabela é livre de prefixo
- Qual texto representa 0101100? e 101001101?
- Por que é possível decodificar?

Códigos livres de prefixo

- Decodificação é simples
- Qual estrutura de dados utilizar para uma decodificação eficiente?

Códigos livres de prefixo

- Decodificação é simples
- Qual estrutura de dados utilizar para uma decodificação eficiente?
- Árvore binária
  - sem sub-árvore: símbolo decodificado no vértice
  - 0: selecionar sub-árvore esquerda
  - 1: selecionar sub-árvore direita
- Exemplo:

	a	b	c	d	e	f
frequência	45	13	12	16	9	5
código de tamanho fixo	000	001	010	011	100	101
código de tamanho variável	0	101	100	111	1101	1100

Códigos livres de prefixo

- codificação ótima: árvore binária cheia
  - cada vértice ou é uma folha, ou tem duas sub-árvores não vazias
- Seja T uma árvore de codificação ótima. Dada a frequência $f(A, c)$ de cada caractere c em um arquivo A, qual o tamanho da representação de A com T?

Códigos livres de prefixo

- codificação ótima: árvore binária cheia
  - cada vértice ou é uma folha, ou tem duas sub-árvores não vazias
- Seja T uma árvore de codificação ótima. Dada a frequência $f(A, c)$ de cada caractere c em um arquivo A, qual o tamanho da representação de A com T?

$$B(A, T) = \sum_{c} f(A, c) d_T(c)$$

  onde $d_T(c)$ é a profundidade de c em T
  $\Longrightarrow$ o custo de T.

Códigos livres de prefixo

- codificação ótima: árvore binária cheia
  - cada vértice ou é uma folha, ou tem duas sub-árvores não vazias
- Seja T uma árvore de codificação ótima. Dada a frequência $f(A, c)$ de cada caractere c em um arquivo A, qual o tamanho da representação de A com T?

$$B(A, T) = \sum_c f(A, c) d_T(c)$$

  onde $d_T(c)$ é a profundidade de c em T
  $\Longrightarrow$ o custo de T.
- Dada $f(A, c)$ para cada c, como construir uma árvore de codificação ótima?

Códigos livres de prefixo

- codificação ótima: árvore binária cheia
  - cada vértice ou é uma folha, ou tem duas sub-árvores não vazias
- Seja T uma árvore de codificação ótima. Dada a frequência $f(A, c)$ de cada caractere c em um arquivo A, qual o tamanho da representação de A com T?

  $$B(A, T) = \sum_{c} f(A, c) d_T(c)$$

  onde $d_T(c)$ é a profundidade de c em T
  $\Longrightarrow$ o custo de T.
- Dada $f(A, c)$ para cada c, como construir uma árvore de codificação ótima? Códificação de Huffman

- David Huffman inventou a codificação de Huffman em 1952
- doutorando no M.I.T.
- avaliação da disciplina *Teoria da Informação*:
  - exame escrito, ou
  - escrever um artigo original sobre o tema “codificação binária ótima”
- desenvolveu a ideia, que era melhor que as existentes (inclusive do docente da disciplina)

- construção incremental da árvore
- abordagem ascendente
- inicia com uma floresta de $|C|$ folhas
- realiza $|C| - 1$ etapas de "junção" para criar a árvore final:
  - escolher duas árvores T_i, T_j da floresta
  - criar uma nova sub-árvore que tem T_i e T_j como sub-árvores.

Construção do código de Huffman

Princípios

- construção incremental da árvore
- abordagem ascendente
- inicia com uma floresta de $|C|$ folhas
- realiza $|C| - 1$ etapas de "junção" para criar a árvore final:
  - escolher duas árvores T_i, T_j da floresta
  - criar uma nova sub-árvore que tem T_i e T_j como sub-árvores.
- como escolher?

Construção do código de Huffman

Algoritmo

- $v.f$ frequência dos caracteres da sub-árvore enraizada em v
- $v.c$ caractere no vértice-folha v.

```
HUFFMAN-CODE(C)
    // Q: fila de prioridade com chave v.f
    for c ∈ C
        v = ALLOC-VERTEX()
        v.f = c.f
        v.val = c
        PUSH-BACK(Q, v)
    MAKE-HEAP(Q, v)
    for i = 1 to |C| − 1
        v = ALLOC-VERTEX()
        l = v.left = EXTRACT-MIN(Q)
        r = v.right = EXTRACT-MIN(Q)
        v.f = l.f + r.f
        INSERT(Q, v)
    return EXTRACT-MIN(Q)
```

Ilustração

	a	b	c	d	e	f
frequência	45	13	12	16	9	5

quadro

Complexidade

Construção do código de Huffman

```
Huffman-Code(C)
    // Q: fila de prioridade com chave v.f
    for c ∈ C // Θ(|C|) iterações
        v = Alloc-Vertex()
        v.f = c.f
        v.val = c
        Push-Back(Q, v)
    Make-Heap(Q, v) // heap binário Θ(|C|)
    for i = 1 to |C| − 1 // |C| iterações
        v = Alloc-Vertex()
        l = v.left = Extract-Min(Q) // Θ(lg |C|)
        r = v.right = Extract-Min(Q) // Θ(lg |C|)
        v.f = l.f + r.f
        Insert(Q, v) // Θ(lg |C|)
    return Extract-Min(Q) // Θ(lg |C|)
```

Complexidade

Construção do código de Huffman

$$\Theta(|C| \lg |C|)$$

Lema

Seja C um alfabeto, e $c.f$ a frequência de cada caractere em C. Se x e y são os caracteres com a menor frequência em C, então existe um código livre de prefixo ótimo para C, onde a códificação de x e a de y tem mesmo comprimento e diferem apenas no último bit.

Roteiro da prova

- Considerar uma árvore T representando um código livre de prefixo ótimo qualquer
- Construir uma árvore T' a partir de T onde as codificações de x e y têm as propriedades enunciadas.
- x e y devem ser irmãos de profundidade máxima em T'.

Demonstração.

- Seja uma árvore T representando um código livre de prefixo ótimo qualquer.
- Seja b e c dois vizinhos na profundidade máxima, $b.f \leq c.f$
- Seja x e y os dois caracteres com a menor frequência, $x.f \leq y.f$
- Temos $x.f \leq b.f$ e $y.f \leq c.f$
- T' é obtido a partir de T trocando x e b.

Demonstração.

- Seja b e c dois vizinhos na profundidade máxima, $b.f \leq c.f$
- Seja x e y os dois caracteres com a menor frequência, $x.f \leq y.f$
- Temos $x.f \leq b.f$ e $y.f \leq c.f$
- T' é obtido a partir de T trocando x e b.

$$
\begin{aligned}
B(T) - B(T') & \\
&= \sum_c c.f \cdot d_T(c) - \sum_c c.f \cdot d_{T'}(c) \\
&= x.f \cdot d_T(x) + b.f \cdot d_T(b) - x.f \cdot d_{T'}(x) + b.f \cdot d_{T'}(b) \\
&= x.f \cdot d_T(x) + b.f \cdot d_T(b) - x.f \cdot d_T(b) + b.f \cdot d_T(x) \\
&= (b.f - x.f) \cdot (d_T(b) - d_T(x)) \\
&= \underbrace{(b.f - x.f)}\geq 0 \cdot \underbrace{(d_T(b) - d_T(x))}\geq 0 \\
&\geq 0
\end{aligned}
$$

Demonstração.

- Seja b e c dois vizinhos na profundidade máxima, $b.f \leq c.f$
- Seja x e y os dois caracteres com a menor frequência, $x.f \leq y.f$
- Temos $x.f \leq b.f$ e $y.f \leq c.f$
- T' é obtido a partir de T trocando x e b.
- T'' é obtido a partir de T' trocando y e c.
- Similarmente, mostramos que $B(T'') \leq B(T') \leq B(T)$.
- Mas B tem custo mínimo, logo $B(T) \leq B(T'')$.
- Conclusão: $B(T'') = B(T)$ e T'', onde a códificação de x e a de y tem mesmo comprimento e diferem apenas no último bit, representa um código livre de prefixo ótimo.

Lema

Seja T uma árvore binária cheia representando um código livre de prefixo ótimo para um alfabeto C, com frequência $c.f$ para cada c em C.

Se x e y são duas folhas vizinhas em T, e z o ascendente imediato dessas folhas. Então, considerando z como um caractere de frequência $x.f + y.f$, a árvore $T' = T - \{x, y\}$ representa código livre de prefixo ótimo para o alfabeto $C' = C - \{x, y\} \cup \{z\}$.

Roteiro da prova

- Relacionar o custo de T com o custo de T'
- Mostrar, por contradição, que T' é representa um código livre de prefixo ótimo.

Demonstração.

- hipótese inicial: T representa um código livre de prefixo ótimo;
- Relacionar o custo de T com o custo de T':

$$
\begin{aligned}
B(T) &= \textstyle\sum_{c \in T} c.f \cdot d_T(c) \\
&= \textstyle\sum_{c \in T - \{x,y\}} c.f \cdot d_T(c) + x.f \cdot d_T(x) + y.f \cdot d_T(y) \\
&= (\textstyle\sum \cdots) + x.f \cdot (d_{T'}(z) + 1) + y.f \cdot (d_{T'}(z) + 1) \\
&= (\textstyle\sum \cdots) + (x.f + y.f) \cdot d_{T'}(z) + x.f + y.f \\
&= (\textstyle\sum \cdots) + x.f + y.f \\
&= \textstyle\sum_{c \in T'} c.f \cdot d_{T'}(c) + x.f + y.f \\
&= B(T') + x.f + y.f
\end{aligned}
$$

$\square$

Demonstração.

- hipótese inicial: T representa um código livre de prefixo ótimo;
- Relacionar o custo de T com o custo de T':
  $B(T) \quad = \quad B(T') + x.f + y.f$
- hipótese adicional: T' não representa um código livre de prefixo ótimo:
  - Logo, existe T'' ótimo para C', tal que $B(T'') < B(T')$.
  - z é uma folha em T' e em T''.
  - Substituindo z por x e y em T'', obtemos uma nova árvore para C, que tem custo menor que T. Contradição

$\square$

Teorema

O algoritmo HUFFMAN-CODE *produz uma árvore que representa um código livre de prefixo ótimo para* C*.*

Demonstração.

A partir dos lemas anteriores. $\square$

Exercícios

1. Explique por quais motivos HUFFMAN-CODE é um algoritmo guloso.
2. Demonstrar que uma árvore binária não cheia não pode representar um código livre de prefixo ótimo.
3. Construir um código livre de prefixo ótimo para o alfabeto e as frequências seguintes:

$$a : 1, b : 1, c : 2, d : 3, e : 5, f : 8, g : 13, h : 21$$

4. Construir um código livre de prefixo ótimo para um alfabeto cujas frequências correspondem à sequência de Fibonacci.
5. Caracterizar instâncias do problema tal que todas as codificações do código livre de prefixo ótimo tenham o mesmo comprimento.