Aula 14

Algoritmos Gulosos

Prof. Marco Aurélio Stefanes
marco em dct.ufms.br

www.dct.ufms.br/ ~ marco

Algoritmos Gulosos

- Problema de Otimização: Queremos computar uma solução que maximiza ou minimizar algum objetivo - A solução ótima
- Algoritmos Gulosos são eficientes em alguns problemas de otimização
- Fazem a escolha que parece melhor no momento
- São mais simples que o uso de programação dinâmica
- Problemas precisam ter “característica gulosa”

Problema de seleção de atividades

- Intervalo: par ordenado de números
- $[s[i], f[i])$: início e fim do intervalo semi-aberto i
- Supomos $s[i] < f[i]$

Problema da seleção de atividades: Dada uma coleção de atividades, digamos S, dadas em intervalos, determinar um subconjunto sem sobreposição (compatíveis) máximo de atividades de S.

Dois intervalos x e y são compatíveis se

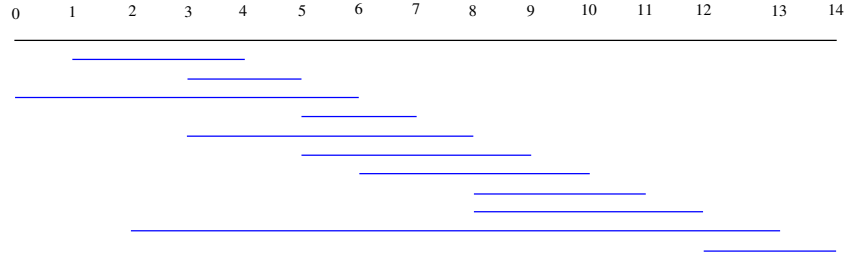
$$f[x] \leq s[y] \text{ ou } f[y] \leq s[x]$$

- Simplificação: Dado um conjunto de atividades S, encontrar o tamanho de um subconjunto sem sobreposição máximo de S.

Exemplo

Conjunto de atividades

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

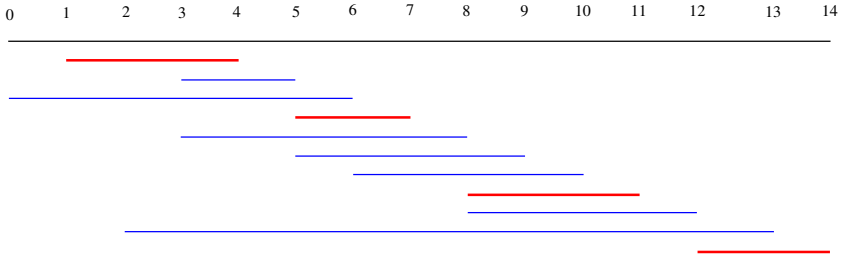


Subconjunto de atividades mutuamente compatíveis máximo

Exemplo

Conjunto de atividades

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

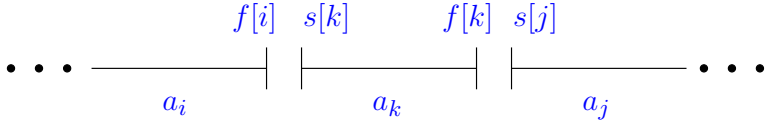


Uma subcoleção disjunta máxima:$\{a_1, a_4, a_8, a_{11}\}$

Substrutura Ótima

Vamos formular uma solução com programação dinâmica para o problema

- Seja $S_{ij} = \{a_k \in S : f_i \leq s[k] < f[k] \leq s[j]\}$
- Atividade a_k que começa após a_i terminar e termina antes de a_j começar



- Atividades em S_{ij} são compatíveis com
  - a_i e a_j
  - Atividades que não terminam depois de $f[i]$
  - Atividades que não começam antes de $s[j]$

Substrutura Ótima

- Vamos acrescentar as atividades fictícias a_0 e a_{n+1}
  - $f[0] = 0$
  - $s[n+1] = \infty$
- Problema todo $S = S_{0,n+1}$
- Para facilitar, vamos supor que as atividades estão em ordem crescente de tempo de término:

$$f[0] \leq f[1] \leq f[2] \leq \cdots \leq f[n] \leq fn + 1$$

- Como encontrar solução ótima a partir de Sij e assim para $S_{0,n+1}$

Substrutura Ótima

- Afirmação Se $i \geq j$ então $S_{i,j} = \emptyset$
  - se existir
    $a_k \in S_{ij} \Rightarrow f_i \leq s[k] < f[k] \leq s[j] < f[j] \Rightarrow f[i] < f[j]$
  - Mas $i \geq j \Rightarrow f[i] \geq f[j]$. Contradição!
- Espaço de subproblemas S_{ij} $0 \leq i < j \leq n + 1$
- Propriedade da substrutura ótima
- Suponha a que solução para S_{ij}, A_{ij}, inclua a_k então
  - temos a solução A_{ik} e A_{kj}
  - A solução para S_{ij} é $A_{ik} \cup \{a_k\} \cup A_{kj}$

Solução Recursiva

- Seja $c[i,j]$ o tamanho de A_{ij}
  - $c[i,j] = 0$ para $i \geq j$, pois $S_{ij} = \emptyset$
- Caso $S_{ij} \neq \emptyset$ e a_k seja usada na solução ótima de S_{ij}, então também usamos soluções ótimas para os subproblemas S_{ik} e S_{kj}
- Temos então a recorrência

$$c[i,j] = c[i,k] + c[k,j] + 1$$

- De fato há $j - i - 1$ possíveis valores para k. A saber, $k = i+1, \cdots, j-1$,
- Recorrência completa

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

Solução Gulosa

- Com a recorrência completa podemos de forma tabular escrever um algoritmo button-up que encontra o valor de uma solução para o problema
- Podemos também construir também a solução ótima propriamente dita
- Qual a complexidade deste algoritmo?
- Há duas observações que permitem simplificar a solução (no teorema a seguir)
- Com estas observações podemos escrever um algoritmo guloso mais eficiente

Solução Gulosa

Teorema Seja S_{ij} não vazio e a_m a atividade em S_{ij} com término mais antigo: $f[m] = \min\{f[k] : a_k \in S_{ij}\}$. Então

1. a_m é usada em algum subconjunto de atividades compatíveis de tamanho máximo de S_{ij}
2. $S_{im} = \emptyset$, tal que a escolha de a_m deixa S_{mj} como única escolha possivelmente não vazia

Prova

- 2. (mais fácil) Suponha $S_{im} \neq \emptyset$, então existe a_k tal que $f[i] \leq s[k] < f[k] \leq s[m] < f[m]$. Assim, a_k também está em S_{ij} e termina antes de a_m. Contradição. Concluímos que $S_{im} = \emptyset$

Teorema

- 1. Seja A_{ij} a solução para S_{ij}. Considere A_{ij} ordenada por tempo de término. Seja a_k a primeira atividade de A_{ij}.
  - Se $a_k = a_m$, Feito!
  - Se $a_k \neq a_m$, construa o subconjunto $A'_{ij} = A_{ij} - a_k \cup a_m$. As atividades em A'_{ij} são compatíveis, pois as em A_{ij} elas eram compatíveis, a_k é a primeira atividade em S_{ij} a terminar e $f[m] < f[k]$. Veja que $|A'_{ij}| = |A_{ij}|$.
- Portanto temos um conjunto de atividades compatíveis máximo contendo a_m.

Solução Gulosa

- Com programação dinâmica temos $j - i - 1$ escolhas para resolver S_{ij}
- O Teorema reduz para apenas um subproblema (o de menor término) para resolver S_{ij}
- forma top-down: Para resolver S_{ij}, escolhemos a_m e depois resolvemos S_{mj}
- padrão dos subproblemas: Solução de $S_{0,n+1}$ é a_{m_1} mais a solução para $S_{m_1,n+1}$, e assim por diante.
- padrão das escolhas: as atividades escolhidas desta forma possuem ordem crescente de término.

Algoritmo Recursivo

A partir destas observações podemos escrever um algoritmo recursivo para o problema.
Vamos considerar a entrada ordenada por tempo de término

Algoritmo RECURSIVE_SELECTOR(s, f, i, n)

1: $m = i + 1$
2: **while** $m \leq n$ **e** $s[m] < f[i]$ **do**
3: $\quad m = m + 1$
4: **if** $m \leq n$ **then**
5: $\quad$ devolva $\{a_m\} \cup$ RECURSIVE_SELECTOR(s, f, m, n)
6: **else**
7: $\quad$ devolva $emptyset$

Chamada Inicial: RECURSIVE_SELECTOR($s, f, 0, n$)
Complexidade: tempo $\Theta(n)$

Algoritmo Iterativo

- A conversão do algoritmo recursivo para um iterativo é Direto
- Novamente, consideramos a entrada ordenada por tempo de término

Algoritmo GREDDY_SELECTOR(s, f, n)

1: $A = \{a_1\}$
2: **for** $m = 2$ to n **do**
3: $\quad$ **if** $s[m] \geq f[i]$ **then**
4: $\quad\quad A = A \cup \{a_m\}$
5: $\quad\quad i = m$

Complexidade: tempo $\Theta(n)$

Estratégia Gulosa

- Em geral, temos os seguintes passos
  - Modelar o problema de forma a ficarmos com apenas uma escolha e temos apenas um subproblema a resolver
  - Provar que existe uma solução ótima que contenha a escolha gulosa, assim fazer esta escolha é sempre segura
  - Mostrar que a escolha gulosa e a resolução do subproblema geram a solução ótima para o problema
- Ingredientes chaves de um algoritmo guloso
  - subestrutura ótima
  - característica gulosa: Solução ótima global pode ser produzida a partir de uma escolha ótima local.

Método Guloso $\times$ Prog. Dinâmica

Ambos precisam de substruturas ótimas

- Programação Dinâmica
  - Um passo de cada vez
  - Escolhe dependem de soluções dos subproblemas: resolva os subproblemas primeiro
  - Resolva de forma Button-up
- Algoritmo guloso
  - Um passo de cada vez
  - Faça a escolha antes de resolver os subproblemas
  - Resolva de forma Top-Down

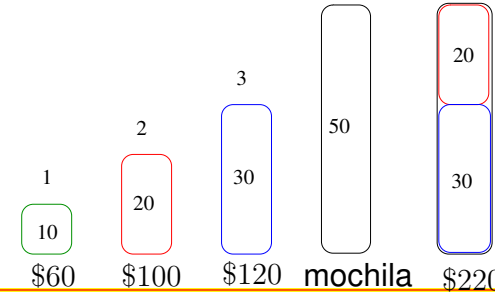
Problema da Mochila

Problema clássico tem duas variações:

- Problema da mochila 0-1
  - Um ladrão rouba uma loja que contém n itens: o item i tem peso w_i e vale $v_i, \in N$. Ele quer levar o maior valor possível em uma mochila de carga máxima W. Quais itens escolher?
  - Pense em ouro em barras
- Problema da mochila fracionada
  - Mesmo formulação anterior, mas agora ele pode carregar frações dos itens, ao invés da escolha binária (0-1).
  - Pense em ouro em pó

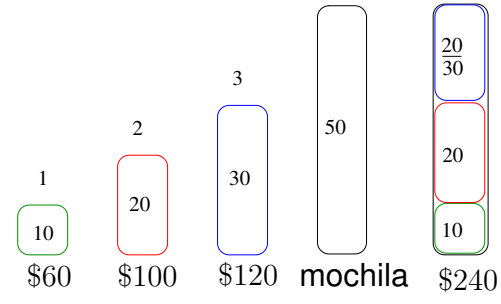
Problema da Mochila

- Ambos tem sub-estrutura ótima: (0-1) Se removermos j de uma solução ótima, a carga restante deve ser a de maior valor para carga máxima $W - w_j$ tomando $n - 1$ itens.
- O Problema da mochila fracionada tem solução gulosa e o Problema da mochila 0-1 não.



Problama da Mochila Fracionada

- Para resolver o Problema da Mochila Fracionada ordene os itens por valor/peso v_i/w_i decrescentemente
- começando em $i = 1$ coloque na mochila o máximo do ítem i que estiver disponível e for possível e se puder levar mais passe para o próximo item.



Problama da Mochila Fracionada

Formulação Geral

- Seja S um conjunto de n itens cada um contendo
  - v_i: o valor do item i
  - w_i: o peso do item i
- Objetivo: Escolher itens com valor total máximo, de forma que o peso total não exceda W
- Podemos pegar frações de cada i
  - Seja $0 \leq x_i \leq 1$ a quantidade escolhida do item i
  - Maximizar
    $$\sum_{i \in S} x_i v_i$$
  - Sujeito a restrição
    $$\sum_{i \in S^*} w_i \leq W$$

Problema da Mochila Fracionada

Algoritmo MOCHILA_FRACIONADA
Entrada: Conj. ordenado S de itens de valor v_i e peso w_i cada e capacidade máxima W
Saída: x_i de cada item i que maximiza o valor sem exceder W

1: $load = 0$
2: $i = 1$
3: **while** $load < W$ e $i \leq n$ **do**
4: **if** $w_i \leq W - load$ **then**
5: Pegue todo o item i
6: **else**
7: Pegue $(W - load)/w_i$ do item i
8: Adicione a $load$ o peso que foi pego
9: $i = i + 1$

Problema da Mochila

- Complexidade do MOCHILA_FRACIONADA $O(n)$
- O Problema da mochila 0-1 pode ser resolvido usando programação dinâmica em tempo $O(nW)$. Algoritmo Pseudo-polinomial.
- Prove que o Problema da Mochila Fracionária tem a propriedade de escolha gulosa.
- O método guloso está associado a teoria dos matróides. Estas estruturas combinatórias buscam determinar quando o método produz soluções ótimas.