Algoritmos *Greedy*

Pedro Ribeiro

DCC/FCUP

2021/2022

Algoritmos Greedy

- Vamos falar de algoritmos ***greedy***.
  Em português são conhecidos como:
  - Algoritmos **ávidos**, **gananciosos**, ou **gulosos**

Estratégias *Greedy* (uma paradigma algorítmico)

- Em cada passo fazer a **"melhor"** escolha **local** (*"imediata"*)
- Nunca olhar *"para trás"* ou mudar decisões tomadas
- Nunca olhar *"para a frente"* para verificar se a nossa decisão tem consequências negativas.

- Esta escolhas locais são feita na expectativa de conduzirem a uma *"boa"* solução **global**

O problema do troco (problema do *cashier*)

Input: Um conjunto de valores de moedas S e uma quantia K a criar com as moedas

Output: O menor número de moedas que fazem a quantia K
(podemos repetir moedas)

Exemplo de Input/Output

Input: $S = \{1, 2, 5, 10, 20, 50, 100, 200\}$
(temos infinitas moedas de cada tipo)
$K = 42$

Output: 3 moedas $(20 + 20 + 2)$

O Problema do Troco

Um algoritmo greedy

Em cada passo escolher a maior moeda que não faz passar da quantia k

Exemplos (com $S = \{1, 2, 5, 10, 20, 50, 100, 200\}$):

- $K = \mathbf{35}$
  - **20** (total: 20) + **10** (total: 30) + **5** (total: 35) [3 moedas]
- $K = \mathbf{38}$
  - 20 + 10 + 5 + 2 + 1 [5 moedas]
- $K = \mathbf{144}$
  - 100 + 20 + 20 + 2 + 2 [5 moedas]
- $K = \mathbf{211}$
  - 200 + 10 + 1 [3 moedas]

O Problema do Troco

- Este algoritmo resulta sempre no **mínimo** número de moedas?

- Para os sistemas de moedas comuns (ex: euro, dólar)... sim!
- Para um sistema de moedas qualquer... não!

Exemplos:

- $S = \{1, 2, 5, 10, 20, 25\}$, $K = 40$
  - Greedy dá 3 moedas $(25 + 10 + 5)$, mas é possível 2 moedas $(20 + 20)$
- $S = \{1, 5, 8, 10\}$, $K = 13$
  - Greedy dá 4 moedas $(10 + 1 + 1 + 1)$, mas é possível 2 moedas $(5 + 8)$

(Será que basta que uma moeda seja $\geq$ que o dobro da anterior?)

- $S = \{1, 10, 25\}$, $K = 40$
  - Greedy dá 7 moedas $(25 + 10 + 1 + 1 + 1 + 1 + 1)$, mas é possível 4 moedas $(10 + 10 + 10 + 10)$

Algoritmos Greedy

- Ideia **"simples"**, mas nem sempre funciona
  - Dependendo do problema, pode não dar resposta **ótima** (pode no entanto dar resposta aproximada que seja *"boa"* o suficiente: irão falar disso noutras UCs, mas em DAA queremos soluções ótimas)
- Normalmente a **complexidade temporal é baixa** (ex: linear ou linearítmica)
- Um **contra-exemplo** prova que um greedy está errado...
- ...o **difícil** é provar a otimalidade!
- Tipicamente é aplicado em **problemas de optimização**
  - Encontrar a "melhor" solução entre todas as soluções possíveis, segundo um determinado critério (função objectivo)
  - Geralmente descobrir um máximo ou ou mínimo
- Uma passo de pré-processamento muito comum é... **ordenar**!

Propriedades para um algoritmo greedy funcionar

Subestrutura Ótima

Quando a solução óptima de um problema contém nela própria soluções óptimas para subproblemas do mesmo tipo

Exemplo

Seja $min(k)$ o menor número de moedas para fazer a quantia k. Se essa solução usar uma moeda de valor v, então o resto das moedas a usar é precisamente $min(k - v)$.

- Se um problema apresenta esta característica, diz-se que respeita o **princípio da optimalidade**.

Propriedades para um algoritmo greedy funcionar

Propriedade da Escolha Greedy

Uma solução ótima é consistente com a escolha greedy que o algoritmo faz.

Exemplo

No caso das moedas de euro, existe uma solução ótima que usa a maior moeda que ainda é menor ou igual à quantia a fazer.

- **Provar** esta propriedade é o mais complicado

Problema do Troco: Prova

- Seja $H = \{h_1, h_2, h_5, h_{20}, h_{50}, h_{100}, h_{200}\}$ uma solução ótima com h_v moedas de cada valor v
- Se $h_{100} > 1$, H não seria ótima (poderíamos simplesmente substituir duas moedas de 100 por uma de 200). Portanto, $h_{100} \leq 1$
- Usando o mesmo raciocínio, $h_{50} \leq 1$, $h_{10} \leq 1$, $h_5 \leq 1$ e $h_1 \leq 1$
- Se $h_{20} > 2$, H não seria ótima (poderíamos substituir três moedas de 20 por uma de 50 e outra 10). Portanto, $h_{20} \leq 2$ (e $h_2 \leq 2$)
- $h_2 = 2$ e $h1 = 1$ não pode acontecer ao mesmo tempo (caso contrário poderíamos simplesmente usar uma moeda de 5). Portanto, $2h_2 + h_1 \leq 4$ (e $20h_{20} + 10h_{10} \leq 40$)

Problema do Troco: Prova

- Temos que:
  - $h_1 \leq 1$
  - $h_2 \leq 2$ (e $2h_2 + h_1 \leq 4$)
  - $h_5 \leq 1$
  - $h_{10} \leq 1$
  - $h_{20} \leq 2$ (e $20h_{20} + 10h_{10} \leq 40$)
  - $h_{50} \leq 1$
  - $h_{100} \leq 1$
- Combinando isto temos que:
  - $5h_5 + 2h_2 + h_1 \leq 9$
  - $10h_{10} + 5h_5 + 2h_2 + h_1 \leq 19$
  - $20h_{20} + 10h_{10} + 5h_5 + 2h_2 + h_1 \leq 49$
  - $50h_{50} + 20h_{20} + 10h_{10} + 5h_5 + 2h_2 + h_1 \leq 99$
  - $100h_{100} + 50h_{50} + 20h_{20} + 10h_{10} + 5h_5 + 2h_2 + h_1 \leq 199$
- Seja $V = \{1, 2, 5, 10, 20, 50, 100\}$.
- Temos então que $\sum_{i=1}^{k} v_i h_i < v_{k+1}$. Portanto, H tem o mesmo número de moedas que a nossa solução *greedy*! $\square$

Exemplos de Algoritmos Greedy

Componentes de um algoritmo greedy

- Um conjunto de soluções possíveis, do qual escolhemos a nossa solução
  (ex: conjunto de moedas, possivelmente com repetições)
- Um função de seleção local, que escolhe o melhor candidato a ser adicionado à solução
  (ex: em cada passo escolher maior moeda $\leq$ quantia restante)
- Uma função objectivo que mede a qualidade da nossa solução
  (ex: quantidade de moedas)
- Uma função de finalização, que indica quando completamos a solução
  (ex: soma das moedas já é igual à quantia desejada)

Fractional Knapsack

Problema da Mochila Fracionada (fractional knapsack)

Input: Uma mochila com capacidade C
Um conjunto de n materiais, cada um com peso w_i e valor v_i

Output: A alocação de materiais para a mochila que maximize o valor transportado.

Os materiais podem ser "partidos" em pedaços mais pequenos, ou seja, podemos decidir levar apenas quantidade x_i do objecto i, com $0 \leq x_i \leq 1$.

O que queremos é portanto respeitar o seguinte:

- Os materiais cabem na mochila ($\sum_i x_i w_i \leq C$)
- O valor da mochila é o maior possível (maximizar $\sum_i x_i v_i$)

Fractional Knapsack

Exemplo de Input

Input: 5 objectos e $C = 100$

i	1	2	3	4	5
w_i	10	20	30	40	50
v_i	20	30	66	40	60

Qual é a resposta ótima neste caso?

- Escolher sempre o material de maior valor:

i	1	2	3	4	5
x_i	0	0	1	0.5	1

Isto daria um peso total de 100 e um valor total de 146.

Fractional Knapsack

Exemplo de Input

Input: 5 objectos e $C = 100$

i	1	2	3	4	5
w_i	10	20	30	40	50
v_i	20	30	66	40	60

Qual é a resposta ótima neste caso?

- Escolher sempre o material mais leve:

i	1	2	3	4	5
x_i	1	1	1	1	0

  Isto daria um peso total de 100 e um valor total de 156.

Fractional Knapsack

Exemplo de Input

Input: 5 objectos e $C = 100$

i	1	2	3	4	5
w_i	10	20	30	40	50
v_i	20	30	66	40	60

Qual é a resposta ótima neste caso?

- Escolher sempre o material com maior rácio *valor/peso*:

i	1	2	3	4	5
v_i/w_i	2	1.5	2.2	1.0	1.2
x_i	1	1	1	0	0.8

Isto daria um peso total de 100 e um valor total de 164.

Fractional Knapsack

Teorema

Escolher sempre a maior quantidade possível do material com maior rácio *valor*/*peso* é uma estratégia *greedy* que dá valor ótimo

1) Subestrutura Ótima

Considere uma solução ótima e o seu material m com melhor rácio.

Se o retirmos da mochila, então o restante tem de conter a solução ótima para os outros materiais que não m e para uma mochila de capacidade $C - w_m$.

Caso assim não seja, então a solução inicial também não era ótima!

Fractional Knapsack

Teorema

Escolher sempre a maior quantidade possível do material com maior rácio *valor*/*peso* é uma estratégia que dá valor ótimo

2) Propriedade da Escolha Greedy

Queremos provar que a máxima quantidade possível do material m com maior rácio (v_i/w_i) deve ser incluida na mochila.

O valor da mochila: $valor = \sum_i x_i v_i$.

Seja $q_i = x_i w_i$ a quantidade de material i na mochila: $valor = \sum_i q_i v_i / w_i$

Se ainda temos material m disponível, então substituir um outro qualquer material i por m vai dar um melhor valor total:

$q_i v_m / w_m \geq q_i v_i / w_i$ (por definição de m)

Fractional Knapsack

Algoritmo greedy para *Fractional Knapsack*

- Ordenar materiais por ordem decrescente de rácio *valor*/*peso*
- Processar o próximo material na lista ordenada:
  - Se o elemento couber na totalidade na mochila, incluir todo e continuar para o próximo material
  - Se o elemento não couber na totalidade na mochila, incluir o máximo possível e terminar

Complexidade:

- Ordenar: $\mathcal{O}(n \log n)$
- Processar: $\mathcal{O}(n)$
- Total: $\mathcal{O}(n \log n)$

Interval Scheduling

Problema do Planeamento de Intervalos (interval scheduling)

Input: Um conjunto de n actividades, cada uma com início no tempo s_i e final no tempo f_i.

Output: Descobrir o maior subconjunto de actividades que não tenham sobreposições

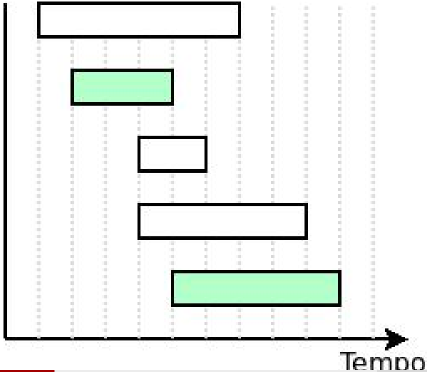
Dois intervalos i e j têm uma sobreposição se existe um tempo k no qual ambos estão activos.

Interval Scheduling

Exemplo de Input

Input: 5 actividades:

i	1	2	3	4	5
s_i	1	2	4	4	5
f_i	7	5	6	9	10



Interval Scheduling

"Padrão" greedy: estabelecer uma ordem segundo um determinado critério e depois ir escolher actividades que não sejam sobrepostas com as já escolhidas

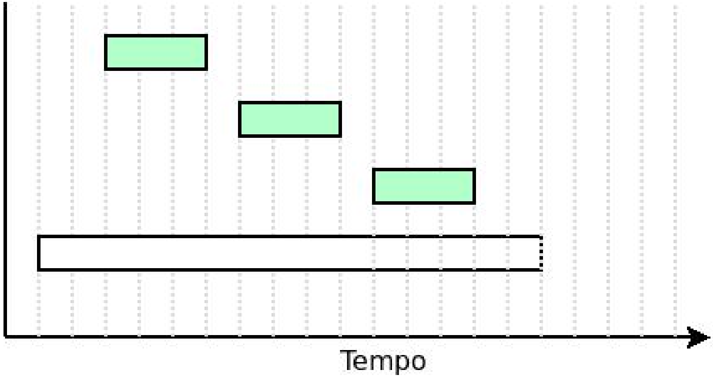
Algumas possíveis ideias:

- [Início mais cedo] Alocar por ordem ascendente de s_i
- [Final mais cedo] Alocar por ordem ascendente de f_i
- [Intervalo mais pequeno] Alocar por ordem ascendente de $f_i - s_i$
- [Menos conflitos] Alocar por ordem ascendente do número de outras actividades que estão sobrepostas

Interval Scheduling

[Início mais cedo] Alocar por ordem ascendente de s_i

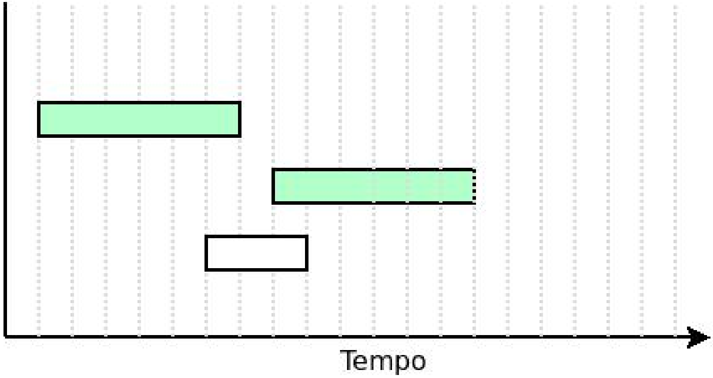
Contra-Exemplo:



Interval Scheduling

[Intervalo mais pequeno] Alocar por ordem ascendente de $f_i - s_i$

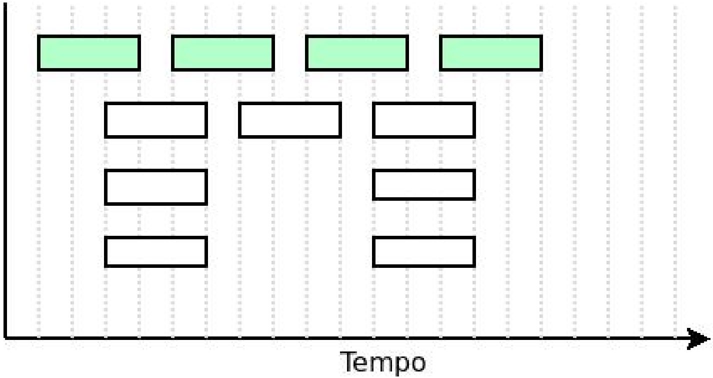
Contra-Exemplo:



Interval Scheduling

[Menos conflitos] Alocar por ordem ascendente do número de outras actividades que estão sobrepostas

Contra-Exemplo:



Interval Scheduling

[Final mais cedo] Alocar por ordem ascendente de f_i

Contra-Exemplo: Não existe!
De facto esta estratégia greedy produz solução ótima!

Teorema

Escolher sempre a actividade não sobreposta com as já escolhidas que tenha o menor tempo de finalização produz uma solução ótima.

1) Subestrutura Ótima

Considere uma solução ótima e actividade m com menor f_m.

Se retirarmos essa actividade então o restante tem de conter a solução ótima para as outras actividades que começam depois de f_m.

Caso assim não seja, então a solução inicial também não era ótima!

Teorema

Escolher sempre a actividade não sobreposta com as já escolhidas que tenha o menor tempo de finalização produz uma solução ótima.

2) Propriedade da Escolha Greedy

Vamos assumir que as actividades estão ordenadas por ordem crescente de tempo de finalização

Seja $G = \{g_1, g_2, \ldots, g_m\}$ a solução criada pelo algoritmo greedy.

Vamos mostrar por **indução** que dada qualquer outra solução ótima H, podemos modificar as primeiras k actividades de H para corresponderem às primeiras k actividades de G, sem introduzirmos nenhuma sobreposição.

Quando $k = n$, a solução H corresponde a G e logo $|G| = |H|$.

Interval Scheduling

Caso base: $k = 1$

- Seja outra solução ótima $H = \{h_1, h_2, \ldots, h_m\}$
- Temos de mostrar que g_1 podia substituir h_1
- Por definição, temos que $f_{g_1} \leq f_{h_1}$
- Sendo assim, g_1 podia ficar no lugar de h_1 sem criar nenhuma sobreposição
- Isto prova que g_1 pode ser o início de qualquer solução ótima!

Interval Scheduling

Passo Indutivo (assumindo que é verdade até k)

- Assumimos que outra solução ótima $H = \{g_1, \ldots, g_k, h_{k+1}, \ldots h_m\}$
- Temos de mostrar que g_{k+1} podia substituir h_{k+1}
- $s_{g_{k+1}} \geq f_{g_k}$ (não existe sobreposição)
- Logo, $f_{g_{k+1}} \leq f_{h_{k+1}}$ (o algoritmo greedy escolhe desse modo)
- Sendo assim, g_{k+1} podia ficar no lugar de h_{k+1} sem criar nenhuma sobreposição
- Isto prova que g_{k+1} pode ser ser escolhido para extender a solução greedy!

Interval Scheduling

Algoritmo greedy para *Interval Scheduling*

- Ordenar actividades por ordem crescente de tempo de finalização
- Começar por iniciar $G = \emptyset$
- Ir adicionando a G a próxima actividade da lista (com menor f_i, portanto) que não esteja sobreposta com nenhuma actividade de G

Complexidade:

- Ordenar: $\mathcal{O}(n \log n)$
- Processar: $\mathcal{O}(n)$
- Total: $\mathcal{O}(n \log n)$

Cobertura Mínima

Problema da cobertura mínima

Input: Um conjunto de n segmentos de linha com coordenadas não negativas $[l_i, r_i]$, e um número M.

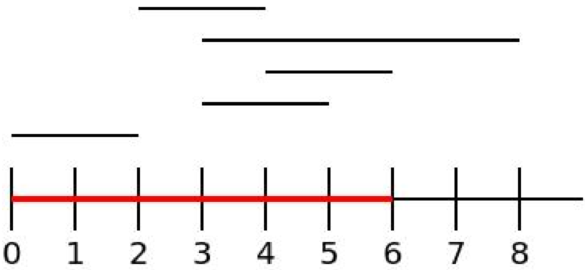
Output: Descobrir a menor quantidade possível de segmentos que cobrem o segmento $[0, M]$.

Cobertura Mínima

Exemplo de Input

Input: 5 segmentos, M=6 :

i	1	2	3	4	5
l_i	0	3	4	3	2
r_i	2	5	6	8	4



Cobertura Mínima

"Padrão" greedy: estabelecer uma ordem segundo um determinado critério e depois ir escolher segmentos cubram zona ainda não coberta

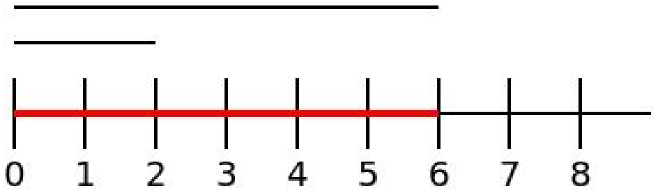
Algumas possíveis ideias:

- [Início mais cedo] Alocar por ordem ascendente de l_i
- [Final mais cedo] Alocar por ordem ascendente de r_i
- [Tamanho maior] Alocar por ordem descendente de $r_i - l_i$

Cobertura mínima

[Final mais cedo] Alocar por ordem ascendente de r_i

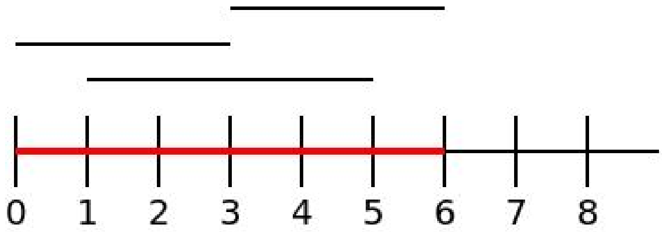
Neste problema não faz sentido!
Contra-Exemplo:



Cobertura mínima

[Tamanho maior] Alocar por ordem descendente de $r_i - l_i$

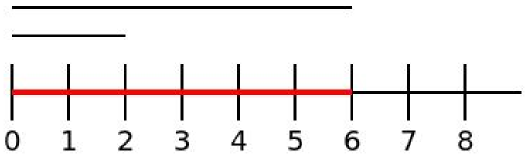
Contra-Exemplo:



Cobertura mínima

[Início mais cedo] Alocar por ordem ascendente de l_i

Parece ser uma boa ideia, porque precisamos de alocar o espaço desde início....
Mas o que acontece se existirem empates?

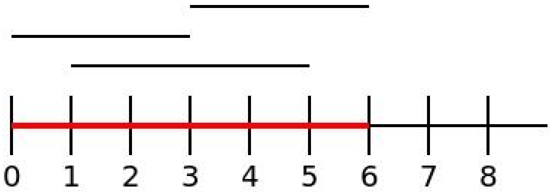


Em caso de empate escolhemos o maior! (o que termina depois)
E será isto suficiente?

Cobertura mínima

[Início mais cedo] Alocar por ordem ascendente de l_i e em caso de empate escolher o maior

O que acontece neste caso?



Se já temos coberto até ao ponto *end*, temos de escolher o segmento que começa em ponto inferior ou igual a *end* e termina o mais para a frente possível!

Intuição: temos sempre de cobrir a partir de *end*. Logo, o melhor que podemos fazer é com um único segmento cobrir até o mais longe possível!

Cobertura mínima

Algoritmo *greedy* para cobertura mínima

- Ordenar actividades por ordem crescente do seu início (l_i).
- Começar por iniciar $end = 0$ (sendo que vamos sempre tendo coberto o segmento $[0, end]$)
- Processar na lista todos os segmentos que têm início pelo menos em end ($l_i \leq end$), e escolher destes o que termina depois (maior r_i).
- Actualizar end para o sítio onde termina o segmento escolhido e repetir o passo anterior até que $end \geq M$

Complexidade:

- Ordenar: $\mathcal{O}(n \log n)$
- Processar: $\mathcal{O}(n)$
- Total: $\mathcal{O}(n \log n)$

Algoritmos Greedy

- Uma ideia muito **poderosa** e **flexível**
- O difícil é **provar** que dá origem a **resultado ótimo**
  - Optimalidade não é garantida porque não explora de forma completa todo o espaço de procura
  - Geralmente é mais fácil provar a incorrecção (via contra-exemplo)
  - Uma maneira de analisar é pensar num caso onde existem empates na condição greedy: o que escolhe o algoritmo nesse caso?
- Quando funcionam, costumam ter **complexidade baixa**
- Não existe "receita mágica" para todos os *greedy*: **experiência é necessária**!
- Vamos falar de vários algoritmos *greedy* durante o resto desta UC:
  - Árvores de Suporte de Custo Mínimo: Prim e Kruskal
  - Distâncias: Dijkstra (também usa ideias de programação dinâmica)
  - ...