Algoritmos

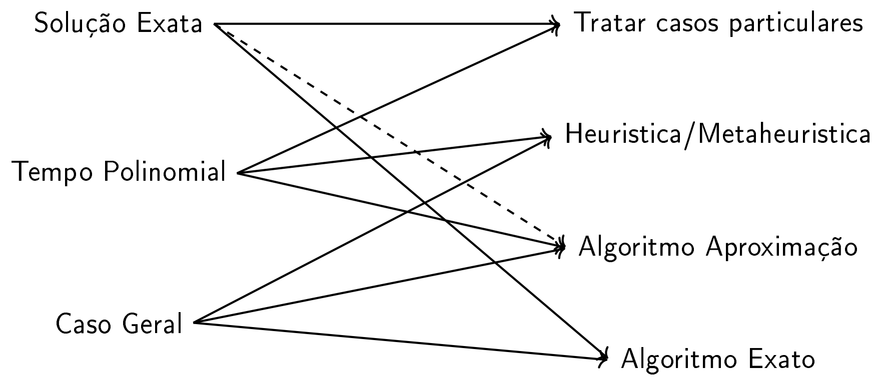
Pedro Hokama

Fontes

- [*clrs*] Algoritmos: Teoria e Prática (Terceira Edição) Thomas H. Cormen, Charles Eric Leiserson, Ronald Rivest e Clifford Stein.
- [*timr*] Algorithms Illuminated Series, Tim Roughgarden
- Desmistificando Algoritmos, Thomas H. Cormen.
- Algoritmos, Sanjoy Dasgupta, Christos Papadimitriou e Umesh Vazirani
- Stanford Algorithms
  `https://www.youtube.com/playlist?list=PLXFMmlk03Dt7Q0xr1PIAriY5623cKiH7V`
  `https://www.youtube.com/playlist?list=PLXFMmlk03Dt5EMI2s2WQBsLsZl7A5HEK6`
- Conjunto de Slides dos Professores Cid C. de Souza, Cândida N. da Silva, Orlando Lee, Pedro J. de Rezende
- Conjunto de Slides do Professores Cid C. de Souza para a disciplina MO420

Qualquer erro é de minha responsabilidade.

Se $P \neq NP$ não conseguiremos para um problema NP-Difícil:



Algoritmos

- Casos Particulares
  - PD para Conjunto Independente de Peso Máximo no Grafo Caminho.
- Heurísticas
- Algoritmos de Aproximação
- Algoritmos Exatos
  - PD para o Knapsack, BackTracking para o Vertex-Cover, PD o TSP.

Problema da Mochila

Dado uma coleção I de n itens e uma capacidade inteira W. Cada item $i \in I$ tem:

- Um valor v_i (não negativo)
- Um peso w_i (não negativo e inteiro)

Encontrar $S \subseteq I$ cujo peso não ultrapasse W, ou seja,

$$\sum_{i \in S} w_i \leq W$$

e que maximiza $\sum_{i \in S} v_i$.

Heurísticas Gulosas

- Estamos interessados em algoritmos rápidos.
- Mas que não necessariamente chegam na solução ótima.
- Podemos então voltar a usar o paradigma de algoritmos gulosos.

Heurística Gulosa para o Knapsack

- Ordenar os itens seguindo algum critério.
- Colocar os itens na solução seguindo essa ordenação até que algum item não caiba.

Tentativa 1

- Ordenar os itens pelos mais valiosos
- Exemplo para $W = 10$:

$v_1 = 3$	$v_2 = 2$	$v_3 = 2$	...	$v_{11} = 2$
$w_1 = 10$	$w_2 = 1$	$v_3 = 1$	...	$w_{11} = 1$

Tentativa 2

- Ordenar pelo valor proporcional ao peso.

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \frac{v_3}{w_3} \geq \ldots \geq \frac{v_n}{w_n}$$

- Colocar os itens na solução até que um não caiba. (Na prática você pode continuar analisando a lista e continuar colocando os itens que couberem na solução)

Algoritmo 1: GreedyKnap(I, W)

Entrada: Um conjunto de itens I e uma capacidade W
Saída: Solução S

Ordenar os itens tal que $\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \frac{v_3}{w_3} \geq \ldots \geq \frac{v_n}{w_n}$;
$S = \emptyset$;
para $i = 1, 2, \ldots, n$ **faça**
    se *item i cabe em S* **então**
        $S = S \cup \{i\}$;
    senão
        Pare;
devolva S;

Quiz

- Considere uma instância da mochila com W = 1000.

$v_1 = 2$	$v_2 = 1000$
$w_1 = 1$	$w_2 = 1000$

- Qual seria o valor de solução dada pelo algoritmo guloso, e qual seria a solução ótima?

a. 2 e 1000
b. 2 e 1002
c. 1000 e 1002
d. 1002 e 1002

- Nessa instância a solução gulosa é 0.004% da solução ótima.
- De fato ela pode ser arbitrariamente ruim em relação a solução ótima.
- Ou seja, é possível encontrar uma instância que faça a solução gulosa ser X% da ótima, para um X tão pequeno quanto você queira.

Algoritmo de Aproximação para Knapsack

- Podemos melhorar a Heurística Gulosa adicionando o seguinte passo:

Algoritmo 2: ApproxKnap(I, W)

Entrada: Um conjunto de itens I e uma capacidade W
Saída: Solução S

1 $S = GreedyKnap(I, W)$;
2 $S' =$ o item mais valioso;
3 devolva o melhor entre S e S';

Teorema

O valor da solução de ApproxKnap é maior ou igual a 50% da solução ótima.

- Um algoritmo com essa caracteristica é dito um Algoritmo de $\frac{1}{2}$ aproximação, ou $\frac{1}{2}$-aproximado.

Para provar o Teorema, vamos considerar o seguinte:

- No GreedyKnap paramos de empacotar no item K, Suponha que pudêssemos completar a mochila com uma fração do item $K + 1$.
- Vamos chamar essa de uma solução Gulosa Fracionária.
- Exemplo: $W = 3$, $v_1 = 3$, $v_2 = 2$, $w_1 = w_2 = 2$.
- Solução fracionária: 4

Quiz

Seja F o valor da solução Gulosa Fracionária. Seja OPT o valor da solução ótima. Qual das alternativas é verdade:

a. $F = OPT$
b. $F > OPT$
c. $F \leq OPT$
d. $F \geq OPT$

- Seja Ap o valor da solução devolvida por *ApproxKnap*. F o valor da solução Gulosa Fracionária e OPT o valor da solução ótima. Então:

$$
\begin{aligned}
Ap &\geq \sum_{i=1}^{K} v_i \\
Ap &\geq v_{k+1} \\
2 * Ap &\geq \sum_{i=1}^{K+1} v_i \geq F \geq OPT \\
Ap &\geq \frac{1}{2} OPT
\end{aligned}
$$

- Será que não poderíamos fazer uma análise mais forte e provar que a solução encontrada não é melhor que 50%?
- Será que poderíamos encontrar características na instância que nos permitiriam mostrar que na verdade a solução é melhor? (Por exemplo: todos os items tem no máximo peso $W/10$)
- Modificar o algoritmo para conseguir uma aproximação maior.

A análise de ApproxKnap é justa

- Considere a seguinte instância do problema da mochila com $W = 1000$

$v_1 = 502$	$v_2 = 500$	$v_3 = 500$
$w_1 = 501$	$w_2 = 500$	$w_3 = 500$

- A solução de ApproxKnap é 502.
- A solução ótima é 1000.
- De fato é possível construir instâncias que a solução de ApproxKnap é de fato 50% da ótima.

- Suponha então que todo item i tem $w_i \leq 10\% W$
- Se ApproxKnap (ou GreedyKnap) falharem em colocar todos os itens na solução, então a mochila está pelo menos 90% cheia.

$$
\begin{aligned}
Ap &\geq 90\% F \\
&\geq 90\% OPT
\end{aligned}
$$

Aproximação Arbitrariamente Boa

- Dado um parâmetro $0 < \epsilon < 1$ (por exemplo, $\epsilon = 0.01$). Garantir uma $(1 - \epsilon)$-Aproximação.
- Parece bom demais. Entretanto o tempo de execução aumenta quando ϵ diminui.
- Pelo lado bom, podemos calibrar ϵ para uma boa troca entre qualidade de solução e tempo de execução.
- Esse é o melhor cenário para problemas NP-Difíceis em relação a aproximação.

- Para vários problemas não existe um algoritmo com aproximação arbitrariamente boa (de tempo polinomial) a menos que $P = NP$.
- Por exemplo: Vertex-Cover.

Arredondamento dos Valores dos Itens

- Ideia: Resolver de forma exata uma instância da Mochila ligeiramente incorreta, porém mais fácil que a instância original.
- Obs: Se os w_i e W são inteiros, podemos resolver a Mochila por Programação Dinâmica em tempo $O(nW)$. (note que no caso particular que W é polinomial em n, o algoritmo também é polinomial)
- Alternativamente: Se os v_i são inteiros, podemos resolver usando programação dinâmica em tempo $O(n^2 \max\{v_i\})$.

- Se todos os v_i forem pequenos (polinomiais em n), então usamos esse algoritmo para obter tempo polinomial.
- Plano: Jogar fora os bits menos significativos dos v_i's.

O algoritmo

- Passo 1: Arredondar para baixo os v_i para o múltiplo de m mais próximo. m depende de ϵ. Quanto Maior o m mais informação é jogada fora, e portanto menos acurado será a solução.

$$\hat{v}_i = \left\lfloor \frac{v_i}{m} \right\rfloor$$

- Passo 2: Resolva a mochila com valores $\hat{v}_i$, pesos w_i e capacidade W.

Knapsack: PD nos Valores

Nosso primeiro algoritmo de PD (nos pesos) para o Knapsack

- Pesos w_i e capacidade W eram inteiros.
- Tempo de execução $O(nW)$
- Uma das dimensões da tabela era W

Algoritmo de PD (nos valores) para o Knapsack

- Valores v_i são inteiros.
- Tempo de execução $O(n^2 \max\{v_i\})$
- Uma das dimensões da tabela é $n \max\{v_i\}$

- Ideia: Uma das dimensões da tabela será i que indica o prefixo $1, \ldots, i$ que é permitido usar.
- O segundo parâmetro x será o valor que desejamos obter (ou maior). E vamos procurar o menor peso que consegue obter aquele valor. $x = 0, 1, \ldots, n \cdot v_max$.
- $A[i][x]$ indicará o menor peso necessário para obter valor pelo menos x usando apenas os itens $1, \ldots, i$.

$$A[i][x] = \begin{cases} A[i-1][x] \\ w_i + A[i-1][x - v_i] \end{cases}$$

- $A[i-1][x - v_i]$ é zero se $v_i \geq x$

Algoritmo 3: KnapsackPDV(I, W)

Entrada: Um conjunto de Itens I e uma capacidade W
Saída: Valor de uma solução ótima

$A[n][n \max\{v_i\}$;
$A[0][0] = 0$;
para $x = 1, 2, \ldots, n \max\{v_i\}$ **faça** $A[0][x] = +\infty$;
para $i = 1, 2, \ldots, n$ **faça**
    para $x = 1, 2, \ldots, n \max\{v_i\}$ **faça**
        $A[i][x] = \min\{A[i-1][x]; w_i + A[i-1][x - v_i]\}$;
devolva o maior x tal que $A[n][x] \leq W$;

- Tempo de Execução $O(n^2 \max\{v_i\})$

Algoritmo $(1-\epsilon)$-aproximado

- Passo 1: Calcular $\hat{v}_i = \lfloor \frac{v_i}{m} \rfloor$ para todo item.
- Passo 2: Resolver o problema com $\hat{v}$ usando KnapsackPDV.

Plano:

- Quão grande pode ser m, que ainda garanta uma $(1-\epsilon)$-aproximação
- Dado esse m qual é o tempo de execução do algoritmo?

Quiz

Suponha que transformamos v_i em $\hat{v}_i$. Qual das seguintes alternativas é verdade?

a. $\hat{v}_i$ está entre $v_i - m$ e v_i
b. $\hat{v}_i$ está entre v_i e $v_i + m$
c. $m \cdot \hat{v}_i$ está entre $v_i - m$ e v_i
d. $m \cdot \hat{v}_i$ está entre $v_i - 1$ e v_i

Análise da Acurácia

Concluimos que:

- 1) $v_i \geq m \cdot \hat{v}_i$
- 2) $m \cdot \hat{v}_i \geq v_i - m$

Seja S^* a solução ótima para o problema original, e S a solução para o problema com $\hat{v}_i$. Como resolvemos de forma ótima o problema usando $\hat{v}_i$ obtemos:

- 3)

$$\sum_{i \in S} \hat{v}_i \geq \sum_{i \in S^*} \hat{v}_i$$

$$\sum_{i \in S} \hat{v}_i \geq \sum_{i \in S^*} \hat{v}_i$$

$$m \cdot \sum_{i \in S} \hat{v}_i \geq m \cdot \sum_{i \in S^*} \hat{v}_i$$

$$\sum_{i \in S} v_i \geq m \cdot \sum_{i \in S} \hat{v}_i \geq m \cdot \sum_{i \in S^*} \hat{v}_i \geq \sum_{i \in S*} (v_i - m)$$

$$\sum_{i \in S} v_i \geq (\sum_{i \in S*} v_i) - nm$$

$$\sum_{i \in S} v_i \geq (\sum_{i \in S*} v_i) - nm$$

Queremos obter

$$\sum_{i \in S} v_i \geq (1 - \epsilon) \sum_{i \in S*} v_i = \sum_{i \in S*} v_i - \epsilon \sum_{i \in S*} v_i$$

Para isso precisamos escolher um m pequeno o bastante tal que:

$$mn \leq \epsilon \sum_{i \in S*} v_i$$

$$mn \leq \epsilon \sum_{i \in S*} v_i$$

Não sabemos qual a solução ótima S^*, mas podemos pegar um m ainda menor.

$$mn = \epsilon \max\{v_i\}$$

$$m = \frac{\epsilon \max\{v_i\}}{n}$$

Algoritmo $(1 - \epsilon)$-aproximado

Algoritmo 4: $(1-\epsilon)$-ApproxKnap(I, W, ϵ)

Entrada: Um conjunto de itens I, uma capacidade W e um fator ϵ
Saída: Valor de uma solução ótima

Calcule v_{max};
$m = \frac{\epsilon v_{max}}{n}$;
para $i = 1, 2, \ldots, n$ **faça**
    $\hat{v}_i = \lfloor \frac{v_i}{m} \rfloor$;
devolva $KnapsackPDV(I$ com valores $\hat{v}, W)$;

Complexidade

- Escolhendo $m = \frac{\epsilon v_{max}}{n}$ garantimos que o valor da nossa solução é $\geq (1-\epsilon)\cdot$ valor do ótimo
- Como o calculo dos $\hat{v}_i$ é linear, a complexidade total do algoritmo é a mesma do *KnapsackPDV*.

$$O(n^2 \hat{v}_{max})$$

$$\hat{v}_{max} \leq \frac{v_{max}}{m} = \frac{v_{max}}{\frac{\epsilon v_{max}}{n}} = \frac{v_{max} n}{\epsilon v_{max}} = \frac{\cancel{v_{max}} n}{\epsilon \cancel{v_{max}}} = \frac{n}{\epsilon}$$

Dessa forma a complexidade do nosso algoritmo $(1-\epsilon)$-aproximado é

$$O(n^2 \hat{v}_{max}) = O\left(n^2 \cdot \frac{n}{\epsilon}\right) = O\left(\frac{n^3}{\epsilon}\right)$$