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.

Algoritmo Exato para o TSP

- Dado um grafo não-direcionado $G = (V, E)$, encontrar o custo mínimo de ciclo que visita exatamente uma vez todos os vértice de V.
- Uma solução força bruta:
- Testar todos os $n!$ percursos possíveis.
- Resolve 12, 13, talvez 14 vértices.
- Podemos fazer melhor? Vamos tentar fazer um algoritmo de Programação Dinâmica!

Tentativa 1

- Vamos entender o ciclo como um caminho que começa no vértice 1 vai até algum vértice j e depois viaja direto de j para 1, fechando o ciclo.
- Podemos usar a ideia do algoritmo de Bellman-Ford, para encontrar o caminho de 1 até j que usa no máximo i arestas.
- Para todo $i \in \{0, 1, \ldots, n\}$ e todo destino $j \in \{1, 2, ..., n\}$, temos L_{ij} o custo de um caminho mínimo de 1 até j que usa no máximo i arestas.

- Podemos então pegar todos os $L_{n-1,j}$ e somar com o custo de c_{j1}.
- Isso vai formar um ciclo de caixeiro viajante? Infelizmente não!
- O Bellman-Ford apenas indica o máximo de arestas que podem ser usadas.

Tentativa 2

- Podemos modificar um pouco a ideia do algoritmo de Bellman-Ford, para encontrar o caminho de 1 até j que usa **EXATAMENTE** i arestas.
- Para todo $i \in \{0, 1, \ldots, n\}$ e todo destino $j \in \{1, 2, ..., n\}$, temos L_{ij} o custo de um caminho mínimo de 1 até j que usa **exatamente** i arestas.

- Novamente, podemos então pegar todos os $L_{n-1,j}$ e somar com o custo de c_{j1}. Certo?
- Infelizmente também não. Note que o caminho mínimo de $n-2$ arestas até um vértice k pode passar por j,
- Dessa forma estaríamos repetindo vértices (e deixando alguns de fora).

Tentativa 3

- Podemos tentar modificar mais um pouco a ideia para encontrar o caminho de 1 até j que usa no **exatamente** i arestas e **NÃO REPETE** vértices.
- Para todo $i \in \{0, 1, \ldots, n\}$ e todo destino $j \in \{1, 2, ..., n\}$, temos L_{ij} o custo de um caminho mínimo de 1 até j que usa **exatamente** i arestas e **não repete** vértices.

- Note que ainda o caminho mínimo de $n-2$ arestas até um vértice k pode passar por j.
- Como garantir que não vai haver repetições de vértices?
- O único jeito é resolver mais subproblemas, para cada possível conjunto de vértices.

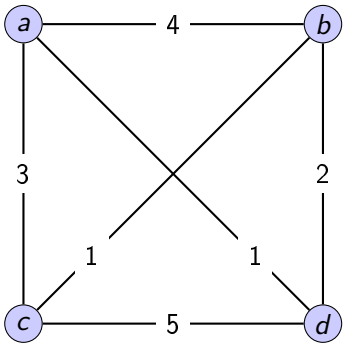
Subestrutura Ótima

- Para todo destino $j \in \{1, 2, ..., n\}$ e todo subconjunto $S \subseteq V$ que contêm 1 e j.
- L_{Sj} é o custo de um caminho mínimo de 1 até j que visita todos os vértices de S (e somente eles).

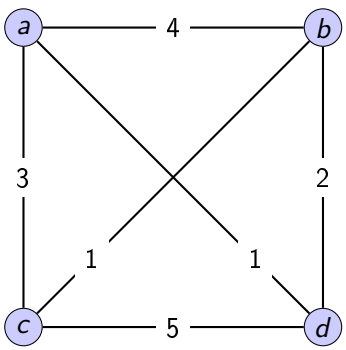
- Considere o seguinte exemplo:
- Queremos descobrir $L_{\{1,3,7,8,9\},7}$ ou seja queremos o caminho de 1 até 7 que visita exatamente uma vez os vértices 1, 3, 7, 8 e 9. Quais opções temos?
- Podemos ir de 1 até 3 visitando $\{1, 3, 8, 9\}$ depois para 7.
- Podemos ir de 1 até 8 visitando $\{1, 3, 8, 9\}$ depois para 7.
- Podemos ir de 1 até 9 visitando $\{1, 3, 8, 9\}$ depois para 7.

Dessa forma podemos escrever a seguinte recorrência:

$$L_{S,j} = \begin{cases} c_{1j}, \text{ se } S = \{1, j\} \\ \min_{k \in S, k \neq j} \left\{ L_{S \setminus \{j\}, k} + c_{kj} \right\} \end{cases}$$

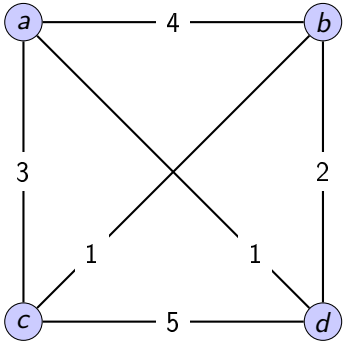


S	b	c	d
$\{a, b\}$	4	-	-
$\{a, c\}$	-	3	-
$\{a, d\}$	-	-	1



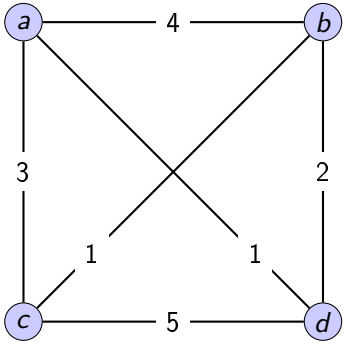
S	b	c	d
$\{a, b\}$	4	-	-
$\{a, c\}$	-	3	-
$\{a, d\}$	-	-	1

S	b	c	d
$\{a, b, c\}$	4	5	-
$\{a, b, d\}$	3	-	6
$\{a, c, d\}$	-	6	8



S	b	c	d
$\{a, b, c\}$	4	5	-
$\{a, b, d\}$	3	-	6
$\{a, c, d\}$	-	6	8

S	b	c	d
$\{a, b, c, d\}$	7	4	6



S	b	c	d
$\{a, b, c, d\}$	7	4	6

Por fim a ultima aresta,
$(b, a) = 7 + 4 = 11$
$(c, a) = 4 + 3 = 7$
$(d, a) = 6 + 1 = 7$

Algoritmo 1: BellmanHeldKarp

Entrada: $G(V, E)$, $V = \{1, 2, ..., n\}$, custos c_{ij} para $(i, j) \in E$
Saída: Custo mínimo de um percurso de caixeiro viajante em G

```
1 //A[S][j] indica o custo mínimo de chegar em j passando uma vez por cada vértice de S ⊆ V
2 A[2^{n-1} − 1][n − 1];
3 para j = 2 até n faça
4     A[{1, j}][j] = c_{1j};
5 para s = 3 até n faça
6     para todo S com |S| = s e 1 ∈ S faça
7         para j ∈ S\{1} faça
8             A[S][j] = min_{k∈S\{1,j}}(A[S\{j}][k] + c_{kj});
9 devolve min_{j∈S\{1}}(A[V][j] + c_{j1});
```


Complexidade.

- Temos $O(2^n)$ possíveis escolhas de S.
- Para cada S temos $O(n)$ problemas (um para cada membro de S)
- O total de subproblemas é $O(n2^n)$
- Para cada subproblema temos que procurar o mínimo em $O(n)$ subproblemas.
- Dessa forma a complexidade total de BellmanHeldKarp é $O(n^2 2^n)$

Considere que vamos utilizar um computador de 4Ghz

n	$n!$	$n^2 2^n$
10	0 s	0s
15	300 s	0s
18	18 dias	0s
20	19 anos	0,1 s
23	200 milênios	1 s
35	?	3 horas
40	?	5 dias

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 $W = 10$:

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

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 2: GreedyKnap(I, W)

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

1 Ordenar os itens tal que $\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \frac{v_3}{w_3} \geq \ldots \geq \frac{v_n}{w_n}$;
2 $S = \emptyset$;
3 **para** $i = 1, 2, \ldots, n$ **faça**
4 **se** *item i cabe em S* **então**
5 $S = S \cup \{i\}$;
6 **senão**
7 Pare;
8 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 3: ApproxKnap(I, W)

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

1 $S = \mathit{GreedyKnap}(I, W)$;
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:

$$Ap \geq \sum_{i=1}^{K} v_i$$

$$Ap \geq v_{k+1}$$

$$2 * Ap \geq \sum_{i=1}^{K+1} v_i \geq F \geq OPT$$

$$Ap \geq \frac{1}{2} OPT$$

- 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.

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