Algoritmos Exatos

F. K. Miyazawa

Instituto de Computação/Unicamp

2021

Sumário I

Introdução

Porque desenvolver Algoritmos Exatos para Problemas NP-Difíceis?

- Há muitos problemas NP-Difíceis precisando ser resolvidos
- Por vezes, para entradas pequenas, é preferível executar um algoritmo $O(1{,}1^n)$ que um $O(n^5)$.
- Algumas aplicações precisam de soluções exatas.
- Algoritmos exatos com complexidade de tempo melhor, podem levar a um aumento no tamanho das instâncias resolvidas.

Notação O^*

Para a análise de algoritmos com complexidade de tempo exponencial, vamos preferir usar a notação O^*, que remove polinômios multiplicativos.

Definição

Função $f(n) \in O^*(g(n))$ *se* $f(n) \in O(n^k g(n))$ *para alguma constante* k.

Exemplo

- $n^2 3^n \in O^*(3^n)$
- $n^{1000} 2^n \in O^*(2^n)$
- $2^n \notin O^*(n^{1000}(2-\varepsilon)^n)$ *para qualquer constante* $\varepsilon > 0$

Recorrências Lineares Homogêneas

Vamos nos restringir a recorrências lineares com coeficientes constantes. Não entraremos muito nas técnicas de resolução de recorrências. O leitor interessado, poderá obter mais sobre isso nas seguintes referências:

- *Gilles Brassard and Paul Bratley, Fundamentals of Algorithmics 1ed, 1995.
- *Kenneth H. Rosen, Discrete Mathematics and Its Applications, 7ed, McGraw Hill, 2011.
- Ronald L. Graham, Donald E. Knuth and Oren Patashnik, Concrete Mathematics, 2ed, Addison-Wesley, 1994.
- George Lueker, Some techniques for solving recurrences, ACM Computing Surveys 12(4):419-436, 1980.

Para mais detalhes, veja as referências marcadas com *.

Recorrências Lineares Homogêneas

Definição

Uma relação de recorrência para uma sequência $\{T_n\}$, é uma equação que expressa T_n através dos termos anteriores, $T_{n-1}, T_{n-2}, \ldots, T_0$, para todos inteiros n com $n \geq n_0$, onde n_0 é um inteiro não negativo.

Estamos interessados em recorrências lineares com coeficientes constantes.

Definição

Recorrências lineares homogêneas com coeficientes constantes: relação de recorrência é dada pela

- *Recorrência:* $c_0 T_n + c_1 T_{n-1} + \cdots + c_k T_{n-k} = 0$
- *Primeiros k termos* $T_0, \ldots, T_{k-1} \in \mathbb{R}$
- *Coeficientes* $c_0, \ldots, c_k \in \mathbb{R}$.

Vamos mencionar as recorrências lineares com coeficientes constantes apenas por recorrência.

Recorrências Lineares Homogêneas

Exemplo: Recorrência da Torre de Hanoi

$$\begin{aligned} T_0 &= 0 \\ T_n &= 2T_{n-1} + 1, \quad \text{para} \quad n \geq 1. \end{aligned}$$

Ao substituirmos os valores para encontrar os primeiros termos, obtemos

$$\begin{aligned} T_1 &= 2T_0 + 1 = 2 \cdot 0 + 1 = 1 \\ T_2 &= 2T_1 + 1 = 2 \cdot 1 + 1 = 3 \\ T_3 &= 2T_2 + 1 = 2 \cdot 3 + 1 = 7 \\ &\vdots \end{aligned}$$

Com isso, podemos 'chutar' que $T_n = 2^n - 1$. O que pode ser provado por indução.

Recorrências Lineares Homogêneas

Fato (Linearidade)

Se f_n e g_n são soluções da recorrência $\sum_{i=0}^{k} c_i T_{n-i} = 0$, então $a \cdot f_n + b \cdot g_n$ e $c \cdot f_n$ também são soluções, para quaisquer constantes a, b, c.

Prova. Considere $h_n = a \cdot f_n + b \cdot g_n$, com a e b constantes. Substituindo na recorrência, temos

$$\begin{aligned}
\sum_{i=0}^{k} c_i h_{n-i} &= \sum_{i=0}^{k} c_i (a \cdot f_{n-i} + b \cdot g_{n-i}) \\
&= a \sum_{i=0}^{k} c_i \cdot f_{n-i} + b \sum_{i=0}^{k} c_i \cdot g_{n-i} \\
&= a \cdot 0 + b \cdot 0 \\
&= 0.
\end{aligned}$$

O outro caso é análogo. □

Recorrências Lineares Homogêneas

Exemplo: Recorrência de Fibonacci

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2}, \quad \text{para} \quad n \geq 2. \end{aligned}$$

O método da substituição sugere que $c\,2^n$ é muito grande para F_n, pois

$$c2^n > c2^{n-1} + c2^{n-2}, \quad \text{para } n \geq 0$$

Vamos estimar que $F_n = c \cdot a^n$ para alguma constante a.

Recorrências Lineares Homogêneas

Substituindo $c \cdot a^n$ na recorrência $F_n = F_{n-1} + F_{n-2}$ temos

$$c \cdot a^n = c \cdot a^{n-1} + c \cdot a^{n-2}.$$

Simplificando, obtemos uma equação, chamada *equação característica*:

$$a^2 = a + 1.$$

Resolvendo esta equação do segundo grau, obtemos duas raízes:

$$a_1 = \frac{1+\sqrt{5}}{2} \quad \text{e} \quad a_2 = \frac{1-\sqrt{5}}{2}.$$

Assim, $c_1 a_1^n + c_2 a_2^n$ também é solução, para constantes c_1 e c_2.

Segue que $F_n \in O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$, uma vez que a_1^n é o termo dominante.

Recorrências Lineares Homogêneas

Podemos calcular F_n exatamente, utilizando valores para $n = 0$ e $n = 1$.

Aplicando os valores da recorrência nos termos da base, temos

$$\begin{aligned} F_0 &= c_1(\tfrac{1+\sqrt{5}}{2})^0 + c_2(\tfrac{1-\sqrt{5}}{2})^0 = 0 \quad \text{e} \\ F_1 &= c_1(\tfrac{1+\sqrt{5}}{2})^1 + c_2(\tfrac{1-\sqrt{5}}{2})^1 = 1. \end{aligned}$$

Resolvendo este sistema, com incógnitas c_1 e c_2, obtemos

$$c_1 = \frac{1}{\sqrt{5}} \quad \text{e} \quad c_2 = \frac{-1}{\sqrt{5}}.$$

Assim, temos que

$$F_n = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^n.$$

Recorrências Lineares Homogêneas

A técnica utilizada na resolução da recorrência de Fibonacci é parecida com a utilizada na resolução de outras recorrências lineares homogêneas.

Definição

Seja

$$c_0 T_n + c_1 T_{n-1} + \cdots + c_k T_{n-k} = 0,$$

onde $c_i's$ *são constantes. Sua equação característica é dada por*

$$c_0 x^k + c_1 x^{k-1} + \cdots + c_k = 0,$$

obtida substituindo T_m *por* x^m *e dividindo tudo por* x^{n-k}.

O seguinte lema é valido:

Lema

Se a é uma raiz da equação característica com multiplicidade m, também são soluções para a recorrência: $a^n,\ na^n,\ n^2a^n, \ldots, n^{m-1}a^n$.

Recorrências Lineares Homogêneas

Combinando as soluções com multiplicidade, temos

Teorema

Seja T_n uma recorrência linear homogênea e $p(n)$ sua equação característica. Se $p(n)$ tem raízes $a_1, \ldots, a_t$, cada raiz a_i com multiplicidade m_i, então

$$T_n = \sum_{i=1}^{t} q_{i,n}\ a_i^n,$$

onde

$$q_{i,n} = \sum_{j=0}^{m_i-1} c_{i,j}\ n^j$$

e os termos $c_{i,j}$ são constantes.

Fato

Seja $a > 1$ a maior raiz da equação característica de uma recorrência linear homogênea T_n. Então $T_n = O^*(a^n)$.

Recorrências Lineares Homogêneas

Exemplo: Considere a recorrência:
$T_0 = 1$, $T_1 = 2$, $T_2 = 3$ e
$T_n - 5T_{n-1} + 7T_{n-2} - 3T_{n-3} = 0$, para $n \geq 3$.

A equação característica é dada por: $x^3 - 5x^2 + 7x - 3 = 0$
que tem duas raízes distintas: $x_1 = 1$ com multiplicidade 2 e $x_2 = 3$.

Assim, $T_n = c_{1,0}n^0 1^n + c_{1,1}n^1 1^n + c_{2,0}n^0 3^n = c_{1,0} + c_{1,1}n + c_{2,0}3^n$.

Pelos valores iniciais da recorrência, T_0, T_1 e T_2, temos

$$\begin{array}{rcccl} T_0 & = & c_{1,0} + c_{1,1}0 + c_{2,0}3^0 & = & 1 \\ T_1 & = & c_{1,0} + c_{1,1}1 + c_{2,0}3^1 & = & 2 \\ T_2 & = & c_{1,0} + c_{1,1}2 + c_{2,0}3^2 & = & 3 \end{array}$$

Resolvendo este sistema, obtemos $c_{1,0} = 1$, $c_{1,1} = 1$ e $c_{2,0} = 0$.

Portanto $T_n = 1 + 1 \cdot n + 0 \cdot 3^n = n + 1$
(uma análise preliminar nos teria dado $T_n = O(3^n)$)

Recorrências Lineares Homogêneas

Definição

Dada recorrência T_n na forma $T_n \leq T_{n-t_1} + T_{n-t_2} + \cdots + T_{n-t_k}$, onde $k \geq 1$, denotamos por $\mathbf{b} = (t_1, t_2, \ldots, t_k)$ o vetor de ramificação de T_n.

Corolário

Se T_n é recorrência com vetor de ramificação $b = (t_1, t_2, \ldots, t_k)$, então, $T_n \in O^*(\alpha^n)$, onde α é a única raiz positiva real da equação característica

$$x^n - x^{n-t_1} - x^{n-t_2} - \cdots - x^{n-t_k} = 0. \quad (1)$$

Definição

Dado vetor de ramificação $b = (t_1, t_2, \ldots, t_k)$ de uma recorrência, chamamos sua única raiz positiva por fator de ramificação e a denotamos por $\tau(t_1, t_2, \ldots, t_k)$

Recorrências Lineares Homogêneas

Lema

Seja $r \geq 2$ *e* $t_i > 0$ *para todo* $i = 1, \ldots, r$. *Então vale que*

1. $\tau(t_1, t_2, \ldots, t_k) > 1$.
2. $\tau(t_1, t_2, \ldots, t_k) = \tau(t_{\pi(1)}, t_{\pi(2)}, \ldots, t_{\pi(k)})$ *para qualquer permutação* π *de* $(1, \ldots, k)$.
3. *Se* $t_1 > t_1'$ *então* $\tau(t_1, t_2, \ldots, t_k) < \tau(t_1', t_2, \ldots, t_k)$.

Lema

Seja i, j, k *números reais positivos. Então*

1. $\tau(k, k) \leq \tau(i, j)$ *para todo vetor de ramificação* (i, j) *tq.* $i + j = 2k$.
2. $\tau(i, j) > \tau(i + \varepsilon, j - \varepsilon)$ *para todo* $0 < i < j$ *e* $0 < \varepsilon < \frac{j-i}{2}$.

Recorrências Lineares Não-Homogêneas

Definição

Recorrências lineares não-homogêneas com coeficientes constantes:

- *Recorrência:* $c_0 T_n + c_1 T_{n-1} + \cdots + c_k T_{n-k} = h_n$
- *Primeiros k termos* $T_0, \ldots, T_{k-1} \in \mathbb{R}$
- *Coeficientes* $c_0, \ldots, c_k \in \mathbb{R}$ *e termo* h_n *não-nulo.*

Vamos considerar recorrências não-homogêneas para certas funções de h_n.

Seja T_n uma recorrência linear não-homogênea dada por

$$c_0 T_n + c_1 T_{n-1} + \cdots + c_k T_{n-k} = b_1^n q_1(n) + b_2^n q_2(n) + \cdots + b_t^n q_t(n), \quad (2)$$

onde
(i) $c_0, \ldots, c_k$ são constantes;
(ii) $b_1, \ldots, b_t$ são constantes distintas e
(iii) $q_1, \ldots, q_t$ são polinômios, cada polinômio q_i com grau d_i.

Recorrências Lineares Não-Homogêneas

Teorema

O polinômio característico da recorrência T_n (em (2)) é dado por

$$p(n) = (c_0 x^k + c_1 x^{k-1} + \cdots + c_k)(x - b_1)^{d_1+1}(x - b_2)^{d_2+1} \ldots (x - b_t)^{d_t+1}.$$

Se $p(n)$ tem raízes $a_1, \ldots, a_t$, cada raiz a_i com multiplicidade m_i, então

$$T_n = \sum_{i=1}^{t} q_{i,n}\ a_i^n,$$

onde

$$q_{i,n} = \sum_{j=0}^{m_i-1} e_{i,j}\ n^j$$

e os termos $e_{i,j}$ são constantes.

Recorrências Lineares Não-Homogêneas

Exemplo: (Brassard e Bratley'95)
Considere a recorrência $T_n = 2T_{n-1} + n + 2^n$ quando $n \geq 1$ e $T_0 = 0$.

Podemos reescrevê-la como

$$T_n - 2T_{n-1} = n + 2^n. \tag{3}$$

A recorrência homogênea obtida só pelo lado esquerdo ($T_n - 2T_{n-1} = 0$), possui equação característica $x - 2 = 0$, que possui raiz 2.

O lado direito de (3) pode ser escrito como

$$n + 2^n = 1^n n + 2^n 1 = b_1^n p_1(n) + b_2^n p_2(n),$$

onde $b_1 = 1$, $p_1(n) = n$, $b_2 = 2$ e $p_2(n) = 1$.

O grau do polinômio $p_1(n)$ é $d_1 = 1$ e do polinômio $p_2(n)$ é $d_2 = 0$.

Recorrências Lineares Não-Homogêneas

A equação característica é dada por $(x-2)(x-1)^{1+1}(x-2)^{0+1}=0$, que simplificando, nos dá

$$(x-2)^2(x-1)^2=0.$$

Como temos duas raízes, uma com valor 1 e outra com valor 2, ambas com multiplicidade 2, podemos escrever a recorrência na forma

$$\begin{aligned} T_n &= c_1 1^n + c_2 n 1^n + c_3 2^n + c_4 n 2^n \\ &= c_1 + c_2 n + c_3 2^n + c_4 n 2^n. \end{aligned}$$

Com isso, já podemos concluir que $T_n \in O(n2^n)$ (ou que $T_n \in O^*(2^n)$).

Recorrências Lineares Não-Homogêneas

Para calcular a recorrência exata, devemos calcular os valores de c_1, c_2, c_3 e c_4, que podem ser obtidos resolvendo-se um sistema obtido com os quatro primeiros valores da recorrência: $T_0, \ldots, T_3$.

$$\begin{array}{rcrrrrrr} T_0 & = & c_1 + & & c_3 & & = & 0 \\ T_1 & = & c_1 + & c_2 + & 2c_3 + & 2c_4 & = & 3 \\ T_2 & = & c_1 + & 2c_2 + & 4c_3 + & 8c_4 & = & 12 \\ T_3 & = & c_1 + & 3c_2 + & 8c_3 + & 24c_4 & = & 35 \end{array}$$

Tal sistema nos dá $c_1 = -2$, $c_2 = -1$, $c_3 = 2$ e $c_4 = 1$. Portanto, a recorrência é dada por

$$T_n = -2 - n + 2^{n+1} + n2^n.$$

Recorrências Lineares Não-Homogêneas

Na análise de muitos algoritmos, temos que analisar a seguinte recorrência:

$$c_0 T_n + c_1 T_{n-1} + \cdots + c_k T_{n-k} = q_n, \quad \text{para um polinômio } q_n.$$

Pelo teorema anterior, o termo q_n acrescenta na equação característica (em relação a da correspondente recorrência homogênea) uma raiz adicional de valor 1, o que acrescenta um termo $e\, n^t 1^n$ na fórmula fechada de T'_n, com e e t constantes.

Fato

Se o polinômio característico da recorrência

$$c_0 T_n + c_1 T_{n-1} + \cdots + c_k T_{n-k} = 0$$

tem maior raiz $\alpha > 1$ *então* $T_n \in O^*(\alpha^n)$. *E da recorrência não-homogênea*

$$c_0 T'_n + c_1 T'_{n-1} + \cdots + c_k T'_{n-k} = q_n$$

temos que $T'_n \in O^*(\alpha^n)$, *para qualquer polinômio* q_n.

Recorrências Lineares Não-Homogêneas

Muitos algoritmos dividem o problema em duas partes satisfazendo a seguinte desigualdade:

$$T_n \leq T_{n-i} + T_{n-j}, \quad \text{para } j \geq i \geq 1$$

Podemos simplificar e limitar a recorrência por cima, considerando

$$T_n = T_{n-i} + T_{n-j}, \quad \text{para } j \geq i \geq 1$$

A equação característica fica $x^j - x^{j-i} - 1 = 0$.
Se esta possui maior raiz $\alpha_{i,j} > 1$, temos $T_n \in O^*(\alpha_{i,j}^n)$

$\alpha_{i,j}$	$j=1$	$j=2$	$j=3$	$j=4$
$i=1$	2,0000	1,6181	1,4656	1,3803
$i=2$		1,4143	1,3248	1,2721
$i=3$			1,2560	1,2208
$i=4$				1,1893

(valores de $\alpha_{i,j}$ arredondados para cima na quarta casa decimal)

Algoritmos de Força Bruta

Geram todas configurações candidatas a solução, com pouco ou sem critério.

Para cada configuração candidata, verificam se de fato é uma solução.

Vantagens

- Fácil de programar;
- tanto para encontrar uma solução;
- ou uma solução ótima, para otimização;
- ou para encontrar todas soluções viáveis

Desvantagens

- Pode levar muito tempo

Algoritmos de Força Bruta

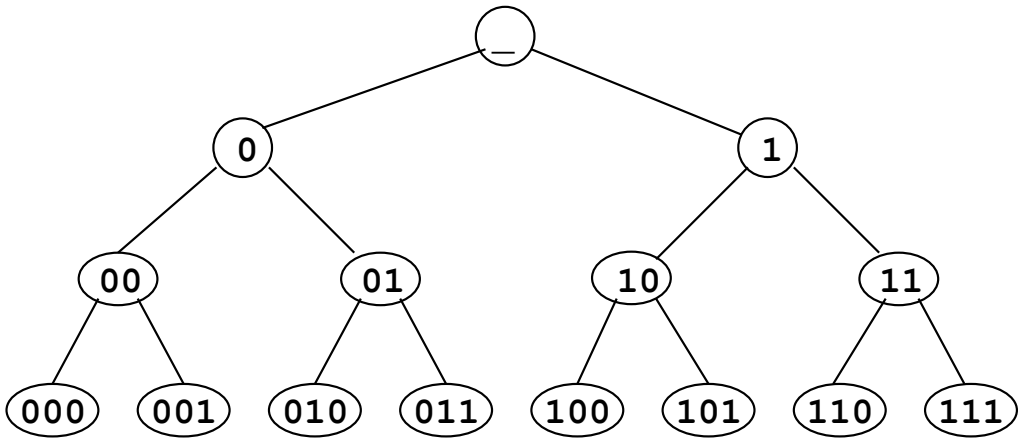
Exemplo: Considere o problema da Satisfatibilidade:

- Dada fórmula booleana $\phi(x_1, \ldots, x_n)$ encontrar todas as atribuições para $x_1, \ldots, x_n$ que satisfazem $\phi(x_1, \ldots, x_n)$.
- A rotina seguinte é chamada inicialmente por SAT-FORÇA-BRUTA $(\phi, x, 1, n)$

SAT-FORÇA-BRUTA (ϕ, x, k, n), onde $x = (x_1, \ldots, x_n)$

```
1   Se k = n + 1 e φ(x) = 1
2       // Obteve uma enumeração válida
3       imprima x
4   Senão
5       Para v ← 0 até 1 faça
6           x_k ← v
7           SAT-FORÇA-BRUTA (φ, x, k + 1, n)
```

Algoritmos de Força Bruta



Fato

O algoritmo SAT-FORÇA-BRUTA $(\phi, x, 1, n)$ *testa todas as* 2^n *possíveis valores de* $x_1, \ldots, x_n$.

Algoritmos de Força Bruta

Estruturas comuns das soluções de alguns problemas NP-difíceis

Subconjunto: Encontrar um subconjunto S com certas propriedades
Complexidade de algoritmo força-bruta: $O^*(2^n)$
Exemplos: Mochila, subset-sum, vertex-cover, conjunto independente.

Permutação: Encontrar uma sequência de elementos com certas propriedades
Complexidade de algoritmo força-bruta: $O^*(n!)$
Exemplos: Caixeiro viajante, alguns problemas de escalonamento, etc.

Partição: Encontrar uma partição dos elementos com certas propriedades
Complexidade de algoritmo força-bruta: $O^*(n^n)$
Exemplos: Coloração de grafos, classificação de objetos, etc.

Algoritmos de Força Bruta

Exercício

Descreva algoritmos exatos por força-bruta para os seguintes problemas:

- *Problema do Caixeiro Viajante*
- *Problema da Clique de peso máximo*
- *Problema da Mochila 0/1 de Valor Máximo*
- *Problema de escalonamento em uma máquina com prioridades e precedência.*
- *Problema da coloração mínima dos vértices de um grafo*

Branch and Bound

Branch and Bound:

- Método que combina enumeração (branch) e
- poda da enumeração (bound)

Estratégia:

- Percorrer uma árvore de enumeração que particiona o conjunto de soluções do problema.
- Sempre que percorrer um ramo que não é promissor ou inviável, podar o ramo sem percorrê-lo.

Branch and Bound

Pontos importantes em um algoritmo Branch and Bound

1. Como fazer a enumeração.
   Que tipo de condições/partição usaremos para ramificar ?
2. Como percorrer a árvore de enumeração.
   Quais ramos iremos percorrer primeiro ?
3. Como podar a árvore eficientemente.
   Que condições de inviabilidade iremos testar ?
   Como avaliar se um ramo não irá levar a soluções melhores ?
   - Uso de limitantes superiores e inferiores para podar a árvore.

Na maioria dos problemas,

- a enumeração gera uma quantidade exponencial de soluções;
- é importante investir em um planejamento adequado de cada um dos itens acima, principalmente a poda

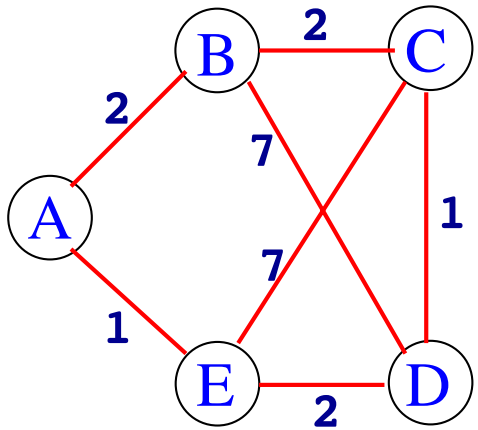
Branch and Bound

Exemplo: Branch and Bound para o TSP

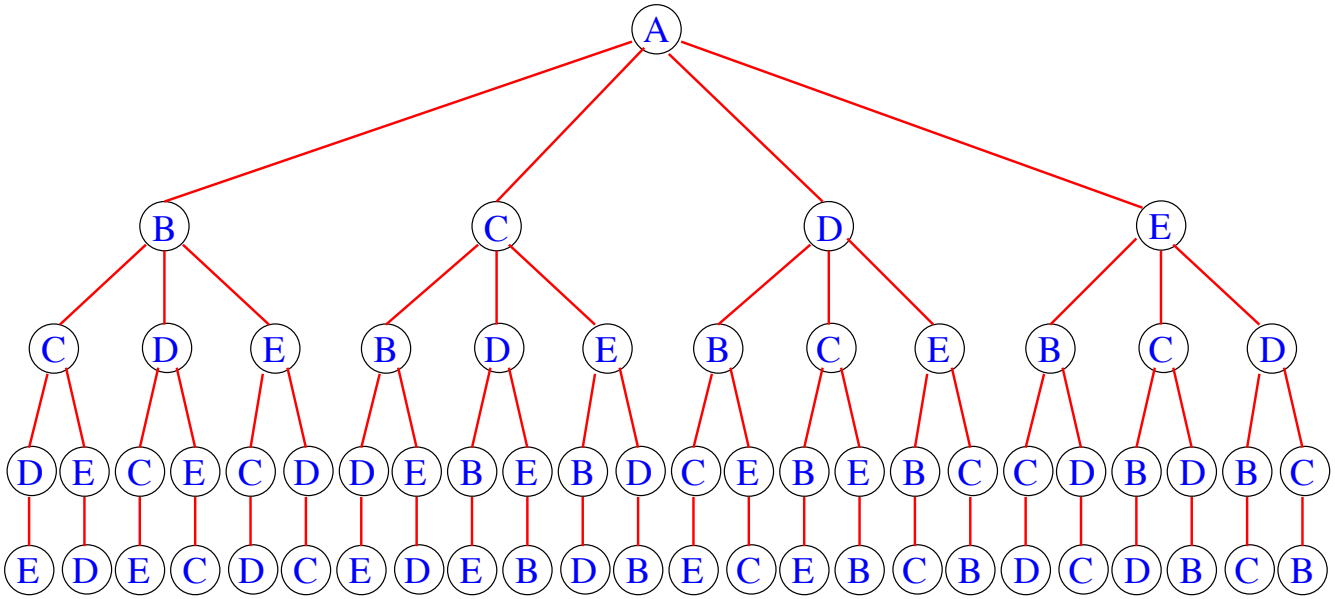
Ramificação da árvore:

- Circuito do TSP é representado como sequência de n vértices distintos
- Enumerar todos os circuitos possíveis por sequências de vértices.
- Cada nó da árvore de enumeração representa um vértice
- Cada ramificação de um nó para outro representa uma aresta percorrida ligando os dois nós

Vamos fazer a árvore de ramificação do seguinte grafo:



Branch and Bound

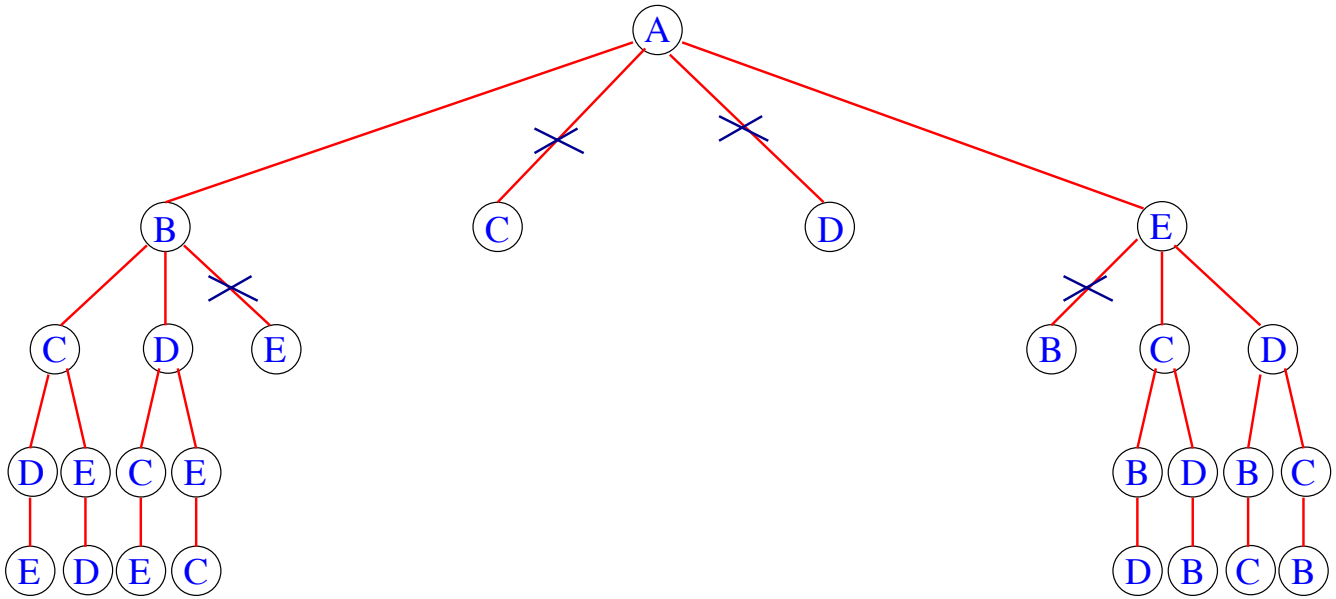


Todas as sequências possíveis dos vértices A, B, C, D e E

Branch and Bound

Poda da árvore por inviabilidade:

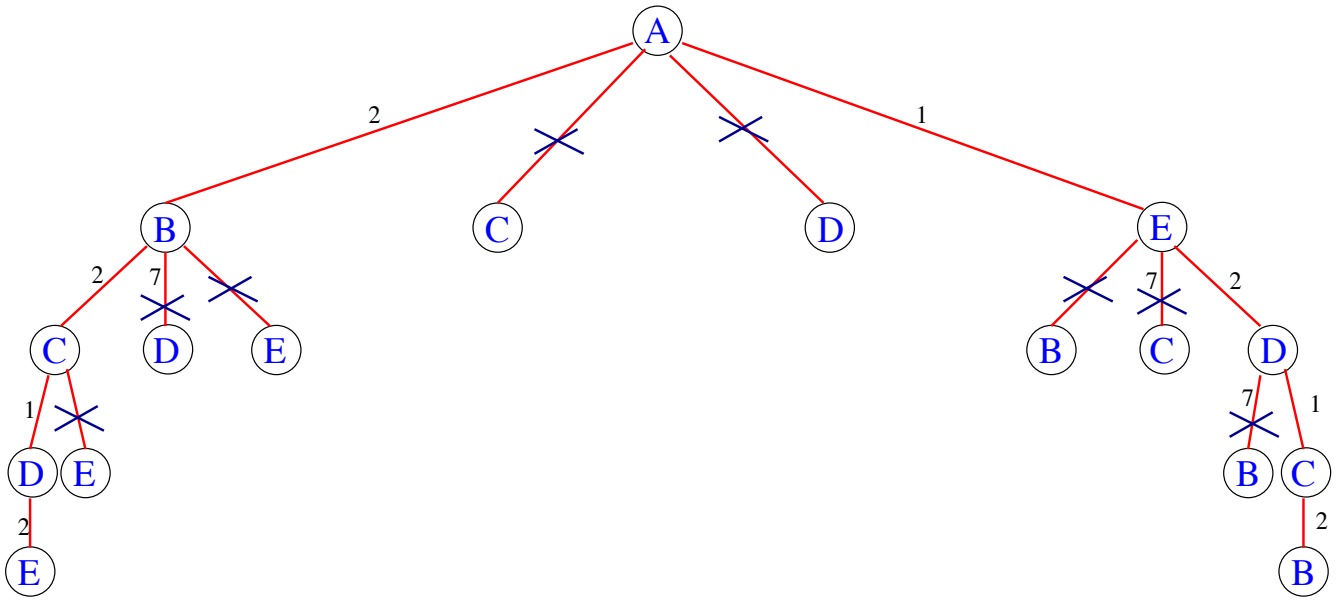
- Algumas arestas da árvore de enumeração não existem. A figura seguinte mostra as podas que podemos fazer considerando apenas a falta de arestas:



Branch and Bound

Poda da árvore por não ser promissor:

- Suponha que o algoritmo escolheu, de maneira gulosa, um ramo promissor, digamos pela sequência A, B, C, D e E de custo 8. Se em outro percurso a soma das arestas possui custo 8, podamos este ramo.



Árvore de branch and bound efetivamente percorrida

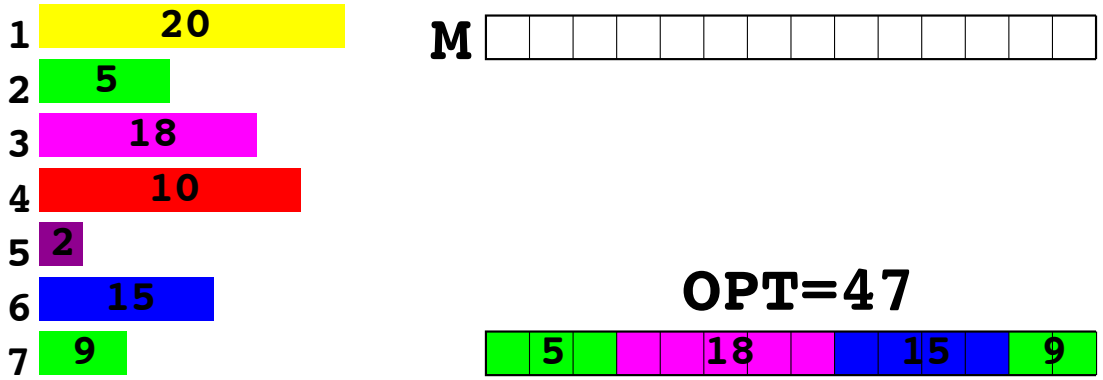
Note a importância de se começar com boas soluções iniciais. O valor da melhor solução encontrada é sempre usado para podar ramos não promissores

Branch and Bound

Problema (Problema da Mochila Ilimitada)

Dados itens $\{1, \ldots, n\}$, *cada item* i *com valor* v_i *e tamanho* s_i, *e capacidade da mochila* B, *encontrar quantidades* $x_i \geq 0$ *do item* i *que maximiza* $\sum_{i \in [n]} v_i x_i$ *tal que* $\sum_{i \in [n]} s_i x_i \leq B$. *Todos os valores são inteiros não negativos.*

Exemplo: de instância para o problema da mochila:



Branch and Bound

Problema da Mochila
Árvore de ramificação com estratégia gulosa

- Nós mais próximos da raiz indicam decisão dos mais 'valiosos' primeiro.
- Primeiros ramos tentam preencher com maior custo relativo primeiro.

Itens ordenados por valor relativo $(1, 2, \ldots, n)$:

$$\frac{v_1}{s_1} \geq \frac{v_2}{s_2} \geq \cdots \geq \frac{v_n}{s_n}$$

Nesta ordenação, objetos de ouro vêm antes dos de prata que vêm antes dos de bronze...

Estratégia gulosa:

1. Tentar instâncias com máximo de ouro,
2. depois completar com máximo de prata,
3. depois completar com máximo de bronze,...

Problema da Mochila

Exemplo [Chvatal'83]:

Capacidade da mochila:	120			
Valor do item:	$v_1 = 4$	$v_2 = 5$	$v_3 = 5$	$v_4 = 2$
Pesos do item:	$s_1 = 33$	$s_2 = 49$	$s_3 = 51$	$s_4 = 22$

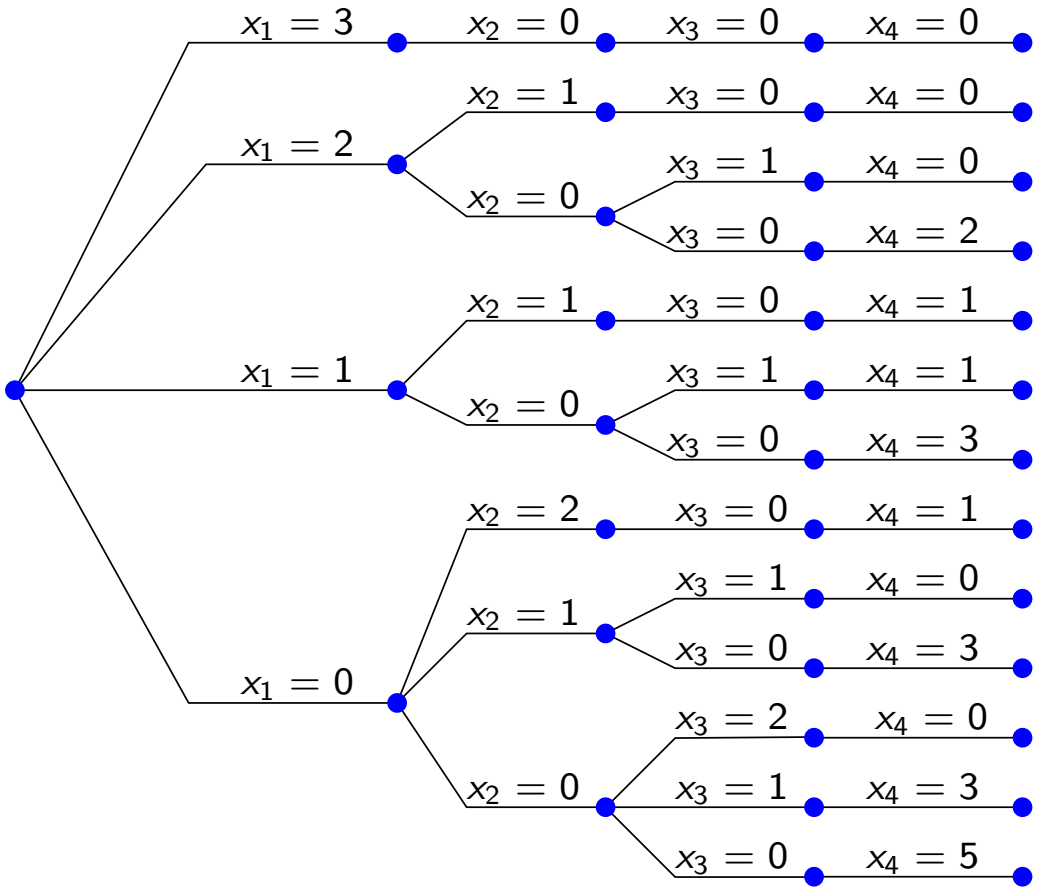
Note que os valores possíveis de x_1 são 0, 1, 2, 3.

Estratégia gulosa para construir e percorrer a árvore de enumeração:

- com os itens de maior valor relativo mais próximos da raiz
- percorrer a árvore de maneira que ramos mais promissores (estratégia gulosa) sejam percorridos primeiro.

Problema da Mochila

Árvore de enumeração completa. Itens de maior valor relativo mais próximos da raiz e os ramos mais promissores mais ao alto.



Note que o primeiro ramo mais ao alto é exatamente o algoritmo guloso.

Problema da Mochila

Percurso na árvore

- obedecendo a ordem de percurso gulosa (ramos de cima para baixo);
- sempre atualizando a melhor solução encontrada;
- podando ramos que não levem a soluções melhores do que temos.

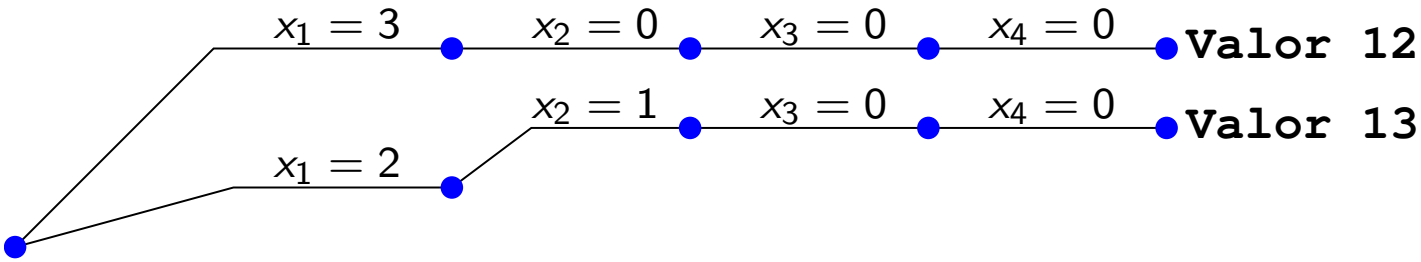
Mantemos a melhor solução encontrada em (x^*, V^*),
onde x^* é o vetor solução e V^* é seu valor.
Sempre que encontrarmos uma solução melhor, atualizamos (x^*, V^*)

Problema da Mochila

Vejamos um momento que atualizamos a melhor solução:

Inicialmente, melhor solução foi obtida no ramo superior (estratégia gulosa), de valor 12.

O próximo ramo a ser percorrido (segundo ramo de cima para baixo) nos dá uma solução de valor 13.



Neste ponto atualizamos as variáveis (x^*, V^*).

Problema da Mochila

Poda por limitante (ramo não é promissor)

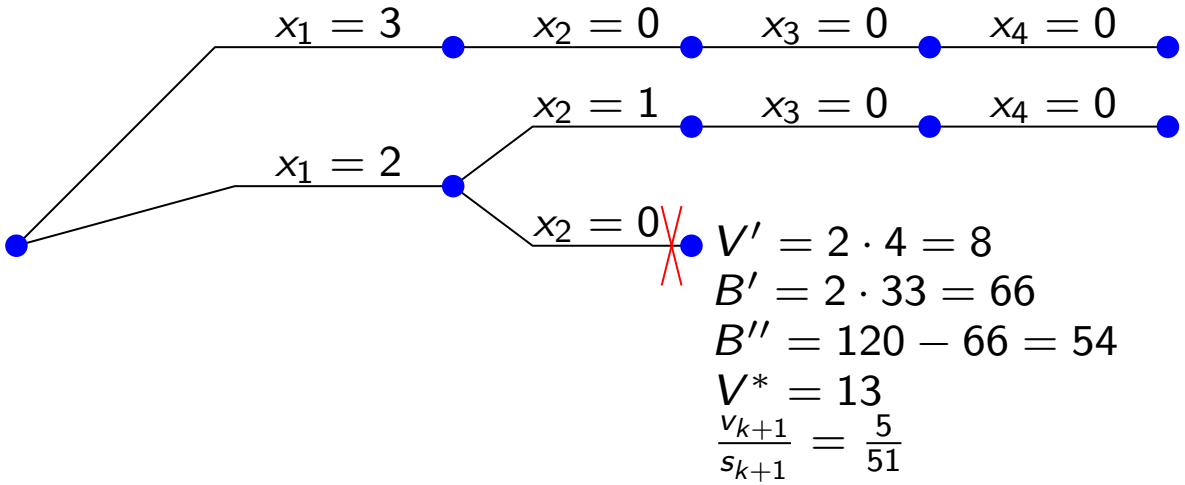
Nesta enumeração consideramos apenas instâncias viáveis

Suponha que estamos em um nó interno da árvore com atribuições de $x_1, \ldots, x_k$ (faltam atribuir $x_{k+1}, \ldots, x_n$).

- seja V' e B' o valor e o espaço preenchido pela atribuição $x_1, \ldots, x_k$
- O espaço não ocupado da mochila, de $B'' := B - B'$ deve ser completado com a melhor atribuição em $x_{k+1}, \ldots, x_n$.
- O maior valor relativo dos itens restantes é de v_{k+1}/s_{k+1}.
- Na melhor das hipóteses, todo o espaço restante da mochila, B'', será completado com este valor relativo.
- Se $V' + B'' \frac{v_{k+1}}{s_{k+1}} \leq V^*$ então não podemos ter solução melhor que a que temos em (x^*, V^*) e podemos descartar este ramo sem percorrê-lo.
- Se os valores dos itens são inteiros, podemos podar quando ocorrer $V' + B'' \frac{v_{k+1}}{s_{k+1}} < V^* + 1$.

Problema da Mochila

Vamos analisar porque a ramificação apresentada abaixo foi podada:



Note que a melhor solução atual tem valor $V^* = 13$ e o ramo irá produzir solução de valor no máximo

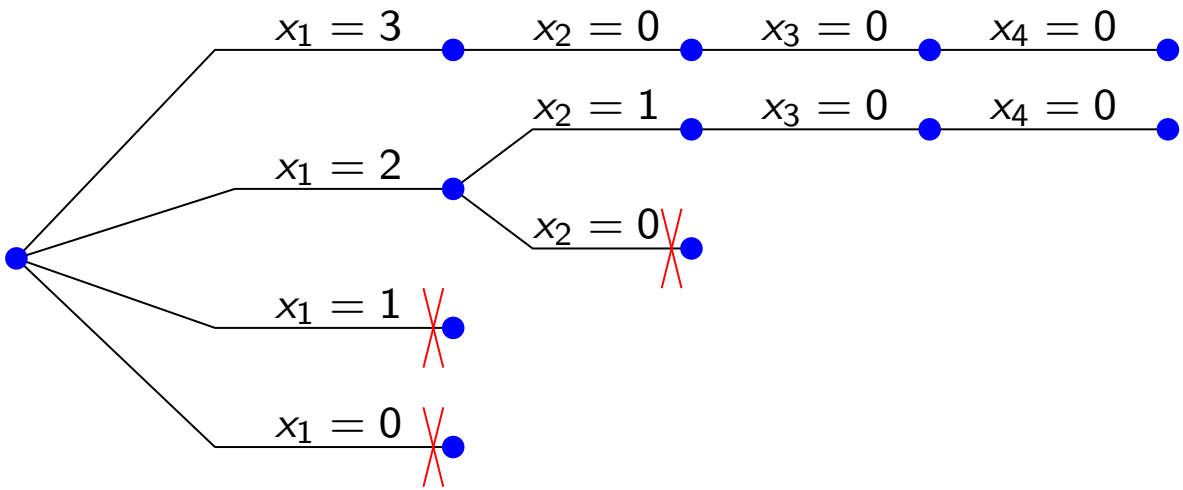
$$V' + B'' \frac{v_{k+1}}{s_{k+1}} = 8 + 54 \cdot \frac{5}{51} = 13.29412$$

Como os valores dos itens são inteiros, só iremos melhorar o valor de V^* se tivermos valor pelo menos 14. I.e., vale que

$$13.29412 = V' + B'' \frac{v_{k+1}}{s_{k+1}} < V^* + 1 = 14$$

Logo podemos podar este ramo.

Problema da Mochila



Árvore de branch and bound efetivamente percorrida

Importante em um algoritmo branch and bound

- Ter boas soluções viáveis (p.ex., usar heurísticas)
- Ter bons limitantes para as soluções de cada nó
- Balanço entre custo computacional para obter limitante e sua qualidade

Problema da Mochila

BRANCH-BOUND-MOCHILA(n, s, v, B) % Itens ordenados por custo relativo

1 $V^* \leftarrow 0$
2 $k \leftarrow 0$
3 para $j \leftarrow k+1$ até n faça $x_j \leftarrow \left\lfloor \left(B - \sum_{i=1}^{j-1} s_i x_i\right) / s_j \right\rfloor$
4 $V \leftarrow \sum_{i=1}^{n} v_i x_i$
5 se $V > V^*$ então
6 $x^* \leftarrow x$
7 $V^* \leftarrow V$
8 $k \leftarrow n$
9 enquanto $(k \geq 1)$ e $(x_k = 0)$ faça $k \leftarrow k - 1$
10 se $k \geq 1$
11 $x_k \leftarrow x_k - 1$
12 $V' \leftarrow \sum_{i=1}^{k} v_i x_i$
13 $B'' \leftarrow B - \sum_{i=1}^{k} s_i x_i$
14 se $V' + \frac{v_{k+1}}{s_{k+1}} B'' \leq V^*$ então
15 vá para o passo 9
16 senão
17 vá para o passo 3
18 retorne x^*

Branch and Bound

Podas na Árvore de Branch and Bound

- Poda por inviabilidade
  O ramo não levará a soluções viáveis.
- Poda por limitante
  O ramo não levará a soluções melhores que a que temos.
- Poda por otimalidade
  Foi obtido uma solução ótima (restrita) para o nó em questão.

Branch and Bound

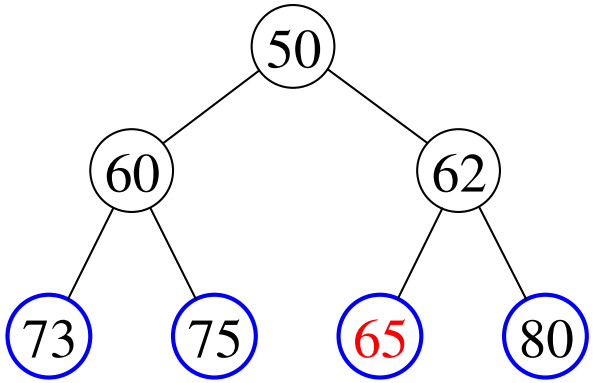
Percorrendo a Árvore de Branch and Bound

- Por busca em profundidade
  (exemplo da árvore de branch and bound do problema da Mochila)
  Possibilitou que a memória consumida na enumeração seja pequena.
  Muitas vezes implementada de maneira recursiva (backtracking)
  Pode fazer com que a solução ótima (ou soluções boas) sejam obtidas após muito processamento.
- Por busca em largura
  Mantém certa igualdade entre ramos, permitindo manter ramos que contenham soluções boas.
  Pode levar a um consumo excessivo de memória.
- Combine ideias das duas estratégias.
  P. ex., explore um ramo que tem o pior limitante para uma solução viável: Diminui a diferença do valor viável entre todos limitantes.
  Possível implementação por Heap.

Branch and Bound

Exemplo: Escolha do nó de pior limitante em Problema de Minimização
Suponha que há solução viável de valor UB $= 100$

Árvore de branch and bound atual com limitante em cada nó:

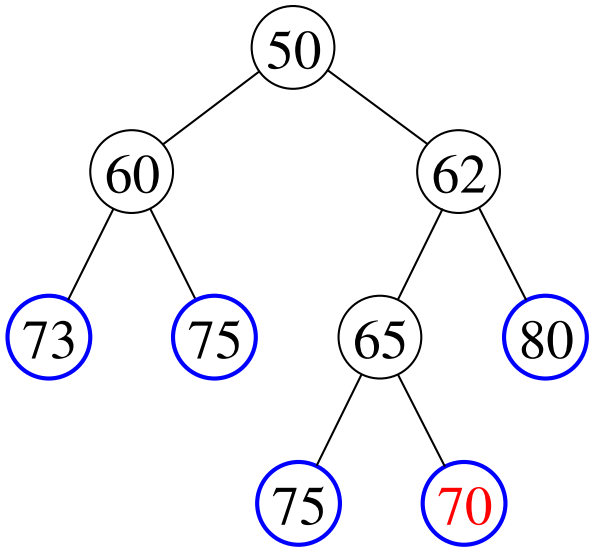


Nós em azul são nós ativos.
Valor em vermelho = pior limitante.
Solução ótima no intervalo $[65, 100]$ (razão de UB/LB$= 100/65 \approx 1,54$)
A ramificação no nó de limitante 65, pode diminuir razão de pior caso

Branch and Bound

Exemplo: Escolha do nó de pior limitante em Problema de Minimização
Suponha que há solução viável de valor $UB = 100$

Árvore de branch and bound atual com limitante em cada nó:



Nós em azul são nós ativos.
Valor em vermelho = pior limitante (LB).
Solução ótima no intervalo $[70, 100]$ (razão de $UB/LB = 100/70 \approx 1,43$)
Razão de pior caso diminuiu de $1,54$ para $1,43$.

Branch and Bound

Exercício:

Projete algoritmos branch and bound para os seguintes problemas

- Problema da Mochila Binária.
- Problema do Caixeiro Viajante.
- Problema do Conjunto Independente de Peso Máximo.

Split and List

Ideia:

- Cada solução pode ser dividida em partes
- Liste soluções para uma ou mais partes do problema.
- Combine soluções parciais com algoritmos rápidos.

Problema SUBSET-SUM

Notação: Dada função $s(.)$ e conjunto X, denote por $s(X)$ a soma $\sum_{i \in X} s_i$

Problema (SUBSET-SUM)

Dada tupla (n, s, K), onde temos n itens, cada item $i \in [n]$ com tamanho inteiro $s_i > 0$ e capacidade inteira $K > 0$, encontrar subconjunto $A \subseteq [n]$ tal que $s(A) = K$.

Ideia:

- Dividir os itens em duas partes iguais (ou quase), L_1 e L_2
- Ordene L_2 pelo tamanho total dos itens no conjunto
- Para cada $A \in L_1$, procure conjunto $B \in L_2$ que some K.

Problema SUBSET-SUM

SUBSET-SUM-SPLITLIST (n, s, K)

1 Seja $L_1 = \{1, \ldots, \lfloor n/2 \rfloor\}$ e $L_2 = \{\lfloor n/2 \rfloor + 1, \ldots, n\}$.
2 Seja $\mathcal{S}_1 = \{S \subseteq L_1 : s(S) \leq K\}$ e $\mathcal{S}_2 = \{S \subseteq L_2 : s(S) \leq K\}$.
3 Ordene $\mathcal{S}_2 = (S_1, \ldots, S_m)$ t.q. $s(S_i) \leq s(S_{i+1})$, para $i = 1, \ldots, m-1$
4 Para cada $A \in \mathcal{S}_1$ faça
5 Procure $B \in \mathcal{S}_2$ de tamanho $K - s(A)$ em $\mathcal{S}_2$, por busca binária.
6 Se encontrou conjunto B no passo anterior, devolva $A \cup B$.
7 Devolva $\emptyset$

Problema SUBSET-SUM

Teorema

SUBSET-SUM-SplitList *resolve* SUBSET-SUM *em tempo* $O^*(2^{n/2})$ *(i.e.,* $O^*(1{,}4143^n)$*).*

Prova.

- Seja O^* uma solução ótima e $O_i^* = O^* \cap L_i$, para $i = 1, 2$.
- Em algum momento O_1^* será o conjunto A do passo 4.
- A busca binária deve encontrar $B = O_2^*$ no passo 5
- e $A \cup B$ é devolvida como solução.
- Os conjuntos $\mathcal{S}_1$ e $\mathcal{S}_2$ são gerados em tempo $O^*(2^{n/2})$.
- A ordenação de $\mathcal{S}_2$ gasta tempo $O^*(2^{n/2} \cdot \log 2^{n/2})$ que é $O^*(2^{n/2})$.
- Uso do espaço em $O^*(2^{n/2})$.

□

Problema Mochila-Binária

Problema (Mochila-Binária)

Dada tupla (n, s, v, K)*, onde temos* n *itens, cada item* $i \in [n]$ *com tamanho inteiro* $s_i > 0$ *e valor inteiro* $v_i > 0$*, capacidade* $K > 0$*, encontrar subconjunto* $A \subseteq [n]$ *tal que* $s(A) \leq K$ *e* $v(A)$ *é máximo.*

Teorema

Existe algoritmo para o Problema Mochila-Binária *com complexidade de tempo* $O^*(2^{n/2}) < O^*(1{,}4143^n)$.

Prova. Exercício.

Dica: Utilize a ideia do algoritmo exato para SUBSET-SUM, com os subconjuntos de L_1 e L_2 ordenados pelo tamanho (crescente) e mantendo para um mesmo tamanho, apenas um conjunto não dominado. □

Problema 3-Coloração

Definição

Dado grafo $G = (V, E)$, uma k-Coloração de G é uma partição de V em k partes $(S_1, \ldots, S_k)$, onde S_i é um conjunto independente em G.

Problema (3-Coloração)

Dado grafo G, encontrar (se existir) uma 3-coloração dos vértices de G.

Algoritmo força bruta: $O^*(3^n)$.

Podemos ter um algoritmo com complexidade $O^*(1.89^n)$

Vamos usar o seguinte resultado:

Lema

O problema 2-Coloração de um grafo pode ser resolvido em tempo polinomial.

Prova. Exercício. □

Problema 3-Coloração

Considerações:

- Seja (S_1, S_2, S_3) uma 3-Coloração de G.
- SPG. considere $|S_1| \leq |S_2| \leq |S_3|$.
- Note que $|S_1| \leq n/3$.
- $G - S_1$ é 2-Colorível.

Ideia:

- Liste todos os conjuntos (independentes) possíveis de S_1
- Para cada S_1 do passo anterior, verificamos se $G - S_1$ é 2-colorível.

Problema 3-Coloração

3Cores(G)

1 Seja $\mathcal{S} = \{S \subseteq V : |S| \leq n/3$ e S é conjunto independente$\}$

2 Para cada $S \in \mathcal{S}$ faça

3 Se $G - S_1$ possui 2-Coloração (S_2, S_3), devolva (S_1, S_2, S_3)

4 Devolva $\emptyset$.

Teorema

O algoritmo 3Cores está correto e pode ser implementado para executar em tempo $O^*(1.89^n)$.

Prova. Temos $|\mathcal{S}| \leq \binom{n}{n/3} + \binom{n}{n/3-1} + \cdots + \binom{n}{1}$. Isto é, $|\mathcal{S}| \leq \frac{n}{3}\binom{n}{n/3}$.
Passo 1 pode ser feito em tempo $O^*(\binom{n}{n/3})$ (exercício).
Usando aproximação de Stirling, temos
$\binom{n}{n/3} \leq 3^{n/3}(3/2)^{2n/3} = (27/4)^{n/3} \leq O^*(1.89^n)$.
Corretude: Exercício. □

Problema 3-Coloração

Algoritmo melhor baseado em conjuntos independentes maximais.

Notação: Dado grafo G simples e não-dirigido com n vértices, denote por $\text{MIS}(G)$ o conjunto de todos os conjuntos independentes maximais de G.

Teorema (Miller e Muller'60 e Moon e Moser'65)

Seja G um grafo simples. Então, $MIS(G) \leq g(n)$, onde

$$g(n) = \begin{cases} 3^{n/3} & \text{se } n \equiv 0 (\mod 3) \\ 4 \cdot 3^{(n-4)/3} & \text{se } n \equiv 1 (\mod 3) \\ 2 \cdot 3^{(n-2)/3} & \text{se } n \equiv 2 (\mod 3) \end{cases}$$

Teorema

Existe algoritmo que obtém $MIS(G)$ em tempo $O^*(MIS(G))$.

Teorema

Há algoritmo $O^*(3^{n/3})$ (i.e., $O^*(1{,}4423^n)$), para o problema 3-Coloração.

Prova. Exercício. □

Branch and Reduce

Ideia: Dividir o espaço de soluções e reduzir subproblemas

- Dividir um problema (ramificar) em subproblemas menores (em geral, com esforço de tempo polinomial.)
- Resolver os subproblemas menores pela mesma abordagem.
- Esforço para reduzir o tamanho dos subproblemas.
- O tamanho dos subproblemas pode não diminuir de um fator (mas possivelmente por uma boa diferença.)

Problema 3-CNF-SAT

Definição (fórmula em CNF)

Uma fórmula ϕ está na forma CNF (forma normal conjuntiva) se

- *ϕ pode ser expressa como uma conjunção ($\wedge$) de cláusulas, onde*
- *uma cláusula é uma disjunção ($\vee$) de uma ou mais literais, e*
- *uma literal é uma variável ou sua negação.*

Nesta seção, vamos considerar uma definição mais ampla para k-CNF-SAT.

Definição (fórmula em k-CNF-SAT)

Uma fórmula ϕ está em k-CNF se ϕ está na forma CNF e cada cláusula possui no máximo k literais distintas.

Exemplo:[Fórmula em 3-CNF-SAT]

$$\phi(x_1, x_2, x_3) = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1) \wedge (x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee \neg x_3).$$

Problema 3-CNF-SAT

Problema (3-CNF-SAT)

Dada fórmula $\phi(x_1, \ldots, x_n)$ em 3-CNF, *encontrar (se existir) uma atribuição para as variáveis de maneira a tornar ϕ verdadeira.*

Teorema

A versão de decisão de 3-CNF-SAT *pertence a* NP-Completo.

Considerações:

- Todas literais de uma cláusula são de variáveis distintas.
- Fórmula de entrada em k-CNF não apresenta cláusulas redundantes para k constante (número de cláusulas é polinomial)

Problema 3-CNF-SAT

Casos Particulares:

Fato

Se ϕ é fórmula sem cláusulas, então qualquer atribuição satisfaz ϕ (todas as cláusulas estão satisfeitas).

Fato

Se ϕ possui uma cláusula vazia (sem literais) não é possível torná-la verdadeira.

Lema (2-CNF-SAT $\in$ P)

Se ϕ é fórmula em 2-CNF (cada cláusula tem no máximo duas literais), há algoritmo, que denominamos por Exato-2SAT, de tempo polinomial, para encontrar atribuição que satisfaz ϕ ou decide que não é satisfatível.

Prova. Exercício. □

Problema 3-CNF-SAT

Primeira ideia: Escolher cláusula $C = (\ell_1 \vee \ell_2 \vee \ell_3)$ e dividir em casos que a tornam verdadeira

$$C \text{ é verdadeira} \Leftrightarrow \begin{cases} \ell_1 = 1 & & & \text{ou} \\ \ell_1 = 0 & \ell_2 = 1 & & \text{ou} \\ \ell_1 = 0 & \ell_2 = 0 & \ell_3 = 1 & \end{cases}$$

Notação: Dada fórmula ϕ, literais $\ell_1, \ldots, \ell_q$ e valores lógicos $v_1, \ldots, v_q$, denotamos por $(\phi|\ell_1 = v_1, \ldots, \ell_q = v_q)$ a fórmula ϕ trocando as aparições das correspondentes variáveis em ℓ_i por seu valor lógico correspondente quando $\ell = v_i$, já simplificada removendo as cláusulas verdadeiras e literais falsas.

Exemplo: Se tivermos a atribuição $(\neg x_3 = 1)$ e

$$\phi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1) \wedge (x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee \neg x_3)$$

então

$$\begin{aligned} (\phi|\neg x_3 = 1) &= (x_1 \vee x_2 \vee 0) \wedge (\neg x_1) \wedge (x_2 \vee 1) \wedge (x_1 \vee x_2 \vee 1) \\ &= (x_1 \vee x_2) \wedge (\neg x_1) \end{aligned}$$

Problema 3-CNF-SAT

EXATO-3SAT-1 (ϕ)

1 Se ϕ é vazia então devolva 1.

2 Se ϕ possui cláusula vazia então devolva 0.

3 Se ϕ está na forma 2-CNF-SAT devolva Exato-2SAT(ϕ).

4 Seja $C = (\ell_1 \vee \ell_2 \vee \ell_3)$ uma cláusula de ϕ com 3 literais.

5 Se EXATO-3SAT-1($\phi|\ell_1 = 1$) devolva 1;

6 senão se EXATO-3SAT-1($\phi|\ell_1 = 0, \ell_2 = 1$) devolva 1;

7 senão se EXATO-3SAT-1($\phi|\ell_1 = 0, \ell_2 = 0, \ell_3 = 1$) devolva 1;

8 senão devolva 0.

Problema 3-CNF-SAT

Teorema

Exato-3SAT-1 *resolve* 3-CNF-SAT *em tempo* $O^*(1{,}8393^n)$.

Prova.

Exercício: prove que o algoritmo devolve a resposta correta.

Complexidade de tempo: No pior caso, cada chamada recursiva faz outras três chamadas, dos passos 5, 6 e 7. Em cada uma das chamadas, uma ou mais variáveis ficam fixas.

A recorrência é dada por

$$T(n) \leq T(n-1) + T(n-2) + T(n-3) + p(n),$$

onde $p(n)$ é um polinômio relativo ao tempo adicional da rotina, além das chamadas recursivas (dividir/simplificar fórmulas/combinar soluções/...).

Problema 3-CNF-SAT

Continuação da prova.

Como $p(n)$ é um polinômio, o fator de ramificação é o mesmo dado pela recorrência homogênea

$$T'(n) = T'(n-1) + T'(n-2) + T'(n-3),$$

que possui equação característica

$$x^3 - x^2 - x - 1 = 0$$

e sua única raiz real positiva é $1{,}83928\ldots$.

Portanto, $T(n) \in O^*(1{,}8393^n)$. □

Humm... pode ser interessante pegar para ramificar cláusulas com poucas literais... se garantirmos pegar sempre cláusulas com no máximo 2 literais, poderemos ter um fator de ramificação menor!

Problema k-CNF-SAT

Melhoria: analisando melhor as atribuições da ramificação (e generalizando para k-CNF-SAT).

Definição: Uma atribuição parcial $a = (x_{i_1} = v_1, x_{i_2} = v_2, \ldots, x_{i_t} = v_t)$ é uma atribuição de valores lógicos para um subconjunto $V = \{x_{i_1}, \ldots, x_{i_t}\}$ das variáveis, onde a variável x_{i_j} recebe valor v_j.

Definição: Uma variável x_i intercepta uma cláusula C se a variável x_i aparece em C (a variável ou sua negação). Um conjunto de variáveis V intercepta uma cláusula C se há variável $x_i \in V$ que intercepta C.

Definição: Uma atribuição parcial a com variáveis V é uma autark se todas as cláusulas interceptadas por V são satisfeitas pela atribuição parcial.

Problema k-CNF-SAT

Lema

Dada fórmula ϕ, uma atribuição parcial a com variáveis V que é uma autark, então

ϕ *é satisfatível se e somente se* $(\phi|a)$ *é satisfatível.*

Prova.

Seja V um subconjunto das variáveis e a uma atribuição parcial que é autark.

$$\phi = \overbrace{C_1 \wedge \ldots \wedge C_k}^{\text{intercep. por } V} \wedge \underbrace{C_{k+1} \wedge \ldots \wedge C_m}_{\text{não intercep. por } V}$$

Note que a atribuição parcial a torna verdadeira $C_1 \wedge \ldots \wedge C_k$.

Note que $(\phi|a) = C_{k+1} \wedge \ldots \wedge C_m$

Variáveis que aparecem em $C_{k+1} \wedge \ldots \wedge C_m$ não estão em V. □

Problema k-CNF-SAT

Lema

Dada fórmula ϕ, uma atribuição parcial a com variáveis V que não é uma autark, então em $(\phi|a)$ há pelo menos uma cláusula que diminuiu o número de literais.

Prova.
Seja a uma atribuição com variáveis V e que não é uma autark.

Por não ser autark, a não consegue satisfazer todas as cláusulas interceptadas por V. Então, há cláusula C interceptada por V que não fica verdadeira em $(\phi|a)$.

Isto é, C contém pelo menos uma literal ℓ cuja correspondente variável pertence a V que não fica verdadeira.

Portanto, a cláusula C em $(\phi|a)$ tem pelo menos a literal ℓ a menos. □

Problema k-CNF-SAT

Ideia: A cada chamada, escolher uma cláusula de menor tamanho (para limitar o número de chamadas recursivas)

EXATO-kSAT (ϕ)

1 Repita
2 Se ϕ é vazia então devolva 1.
3 Se ϕ possui cláusula vazia então devolva 0.
4 Seja $C = (\ell_1 \vee \ldots \vee \ell_q)$ uma cláusula de ϕ com menor tamanho.
5 Seja a_1=(ℓ_1=1), a_2=(ℓ_1=0, ℓ_2=1) ... a_q=(ℓ_1=0, ℓ_2=0, ..., ℓ_q=1).
6 Se (a_i é autark, para algum $i \in \{1, \ldots, q\}$), então $\phi \leftarrow (\phi|a_i)$
7 senão saia do loop Repita.
8 Se EXATO-kSAT$(\phi|a_1)$ devolva 1;
9 senão se EXATO-kSAT$(\phi|a_2)$ devolva 1;
10 $\vdots$
11 senão se EXATO-kSAT$(\phi|a_q)$ devolva 1;
12 senão devolva 0.

Problema k-CNF-SAT

Lema

Exceto pela primeira vez que a rotina Exato-kSAT *é chamada, as chamadas recursivas feitas nas linhas 8–11, serão feitas para fórmulas que sempre terão pelo menos uma cláusula com no máximo* $k - 1$ *literais.*

Prova. Note que as q chamadas recursivas nas linhas 8–11 só serão feitas se a_i não for autark para todo $i = 1, \ldots, q$.

Para cada uma destas chamadas (que não é a primeira), o lema anterior garante que há pelo menos uma cláusula com uma literal a menos.

Com isso, na escolha da correspondente cláusula C na linha 4, temos que C temo no máximo $k - 1$ literais. □

Vamos ver como fica a análise para o 3-CNF-SAT.

Problema k-CNF-SAT

Teorema

EXATO-3SAT *resolve* 3-CNF-SAT *em tempo* $O^*(1{,}6181^n)$.

Prova.

Exercício: prove que o algoritmo devolve a resposta correta.

Complexidade de tempo: A cada chamada, temos duas possíveis situações:

A correta verificação na atribuição de ϕ na linha 6 é válida pois logo antes da atribuição, temos que ϕ é verdadeira sse $(\phi|a_i)$ é verdadeira.

A verificação nas linhas 6-7 só permite seguir para as chamadas recursivas das linhas 8–11 se nenhuma das atribuições $a_1, \ldots, a_q$ for autark.

Tirando a primeira chamada, para todas as outras, temos:

$$T(n) \leq T(n-1) + T(n-2) + p(n),$$

onde $p(n)$ é um polinômio relativo ao tempo adicional da rotina, além das chamadas recursivas (dividir/simplificar fórmulas/combinar soluções/...).

Problema 3-CNF-SAT

Continuação da prova.

A recorrência

$$T(n) \leq T(n-1) + T(n-2) + p(n),$$

é não-homogênea e seu fator de ramificação é o mesmo da recorrência homogênea

$$T'(n) - T'(n-1) - T'(n-2) = 0,$$

que possui equação característica

$$x^2 - x - 1 = 0$$

e sua única raiz real positiva é 1,6180 . . ..

Portanto, $T(n) \in O^*(1{,}6181^n)$. □

Exercício: Faça a análise do algoritmo Exato-4SAT.

Problema Conjunto Independente Máximo

Definição (Conjunto Independente)

Dado grafo $G = (V, E)$, um conjunto $S \subseteq V$ é um conjunto independente de G se não há aresta $e \in E$ com ambas extremidades em S.

Problema (Conjunto-Independente-Máximo)

Dado grafo $G = (V, E)$, encontrar conjunto independente $S \subseteq V$ de cardinalidade máxima.

Teorema

A versão de decisão do Problema Conjunto-Independente-Máximo *pertence a* NP-Completo.

Notação: Denotamos por $\text{grau}(v)$ o grau do vértice v e por $\text{adj}(v)$ o conjunto dos vértices adjacentes a v.

Problema Conjunto Independente Máximo

Considere o seguinte algoritmo força-bruta para Conjunto-Independente-Máximo.

ConjInd-1 (G), onde $G = (V, E)$

1 se $|V| \leq 1$ então devolva V
2 senão
3 seja $v \in V$ um vértice qualquer.
4 $S_0 \leftarrow$ ConjInd1$(G - v)$.
5 $S_1 \leftarrow$ ConjInd1$(G - \mathrm{adj}(v) - v) \cup \{v\}$.
6 devolva $S \in \{S_0, S_1\}$, com $|S|$ máximo.

Como o vértice v não necessariamente tem um vizinho, o desempenho do algoritmo pode ser dado pela recorrência

$$T(n) = T(n-1) + T(n-1) + p(n)$$

Que nos dá uma complexidade de tempo $O^*(2^n)$.

Problema Conjunto Independente Máximo

A análise do algoritmo ConjInd-1 nos diz para escolher vértice de grau não nulo.

ConjInd-2 (G), onde $G = (V, E)$

1 se $E = \emptyset$ então devolva V

2 senão

3 seja $v \in V$ um vértice de grau não nulo.

4 $S_0 \leftarrow$ ConjInd2$(G - v)$.

5 $S_1 \leftarrow$ ConjInd2$(G - \mathrm{adj}(v) - v) \cup \{v\}$.

6 devolva $S \in \{S_0, S_1\}$, com $|S|$ máximo.

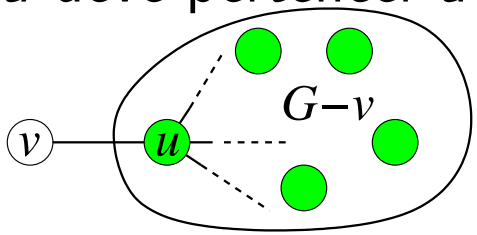
Como o vértice v tem pelo menos um vizinho, o grafo da segunda chamada tem pelo menos dois vértices a menos. O que nos dá a recorrência

$$T(n) = T(n-1) + T(n-2) + p(n)$$

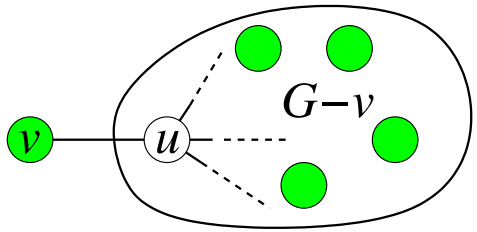
Que nos dá uma complexidade de tempo $O^*(1{,}6181^n)$.

Problema Conjunto Independente Máximo

Se pudermos pegar vértices de grau alto, podemos ter desempenho melhor.
Mas pior caso do algoritmo$\textsc{ConjInd}$-2 é quando v tem grau 1.
Considere v de grau 1 e não pertencente a uma solução ótima S^*
Note que seu único vizinho u deve pertencer a S^*:



Por outro lado, $S' = S^* - u + v$ também é conjunto independente ótimo!



Lema

Seja G um grafo e v um vértice de grau 1. Então há um conjunto independente ótimo que contém v.

Prova. Exercício. □

Problema Conjunto Independente Máximo

Usando esta ideia, podemos ter o seguinte algoritmo:

ConjInd-3 (G), onde $G = (V, E)$

```
1   I ← ∅
2   Repita
3       se V(G) = ∅ então devolva I
4       seja v ∈ V(G) de grau máximo
5       se grau_G(v) ≤ 1 então
6           S ← S ∪ {v}
7           G ← G − v − adj(v)
8       senão interrompa o loop 'Repita'
9   S_0 ← ConjInd3(G − v) ∪ I.
10  S_1 ← ConjInd3(G − adj(v) − v) ∪ {v} ∪ I.
11  devolva S ∈ {S_0, S_1}, com |S| máximo.
```

Problema Conjunto Independente Máximo

Como o vértice v tem grau pelo menos 2, o grafo da segunda chamada tem pelo menos três vértices a menos. O que nos dá a recorrência

$$T(n) = T(n-1) + T(n-3) + p(n)$$

cujo fator de ramificação é dado pela maior raiz da seguinte equação característica:

$$x^3 - x^2 - 1 = 0$$

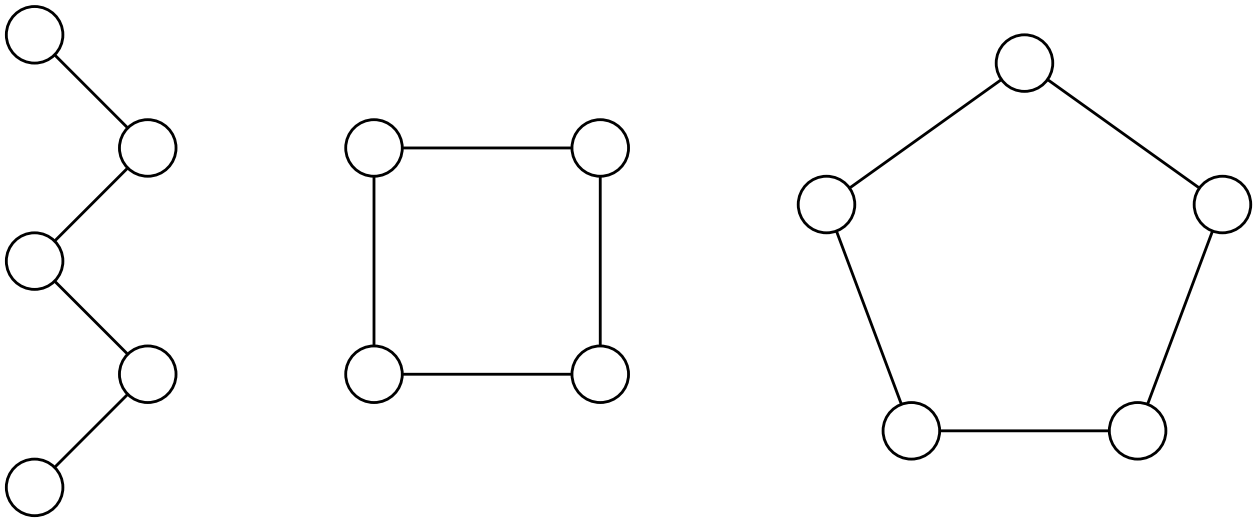
Que tem uma única raiz real positiva 1,46557 . . . .

Lema

O algoritmo ConjInd-3 *tem complexidade de tempo* $O^*(1{,}4656^n)$.

Problema Conjunto Independente Máximo

Podemos continuar melhorando!
Agora o gargalo são vértices de grau 2.
Note que grafos com grau máximo 2 são bem simples.



Lema

Existe algoritmo, ConjInd-$\Delta 2$*, que obtém conjunto independente máximo em grafos de grau máximo 2 em tempo polinomial.*

Prova. Exercício. □

Problema Conjunto Independente Máximo

Notação: Denote por $\Delta(G)$ o grau máximo de um vértice em G.

Usando esta ideia, podemos ter o seguinte algoritmo:

ConjInd-4 (G), onde $G = (V, E)$

1 Se $|V| \leq 1$ devolva V.
2 Se $\Delta(G) \leq 2$ devolva(ConjInd-Δ2(G)).
3 Seja $v \in V(G)$ de grau máximo.
4 $S_0 \leftarrow$ ConjInd4$(G - v)$.
5 $S_1 \leftarrow$ ConjInd4$(G - \mathrm{adj}(v) - v) \cup \{v\}$.
6 Devolva $S \in \{S_0, S_1\}$, com $|S|$ máximo.

Problema Conjunto Independente Máximo

Como o vértice v tem grau pelo menos 3, o grafo da segunda chamada tem pelo menos quatro vértices a menos. O que nos dá a recorrência

$$T(n) = T(n-1) + T(n-4) + p(n)$$

cujo fator de ramificação é dado pela maior raiz da seguinte equação característica:

$$x^4 - x^3 - 1 = 0$$

Que tem uma única raiz real positiva 1,38027 . . ..

Lema

O algoritmo ConjInd-4 *tem complexidade de tempo* $O^*(1{,}3803^n)$.

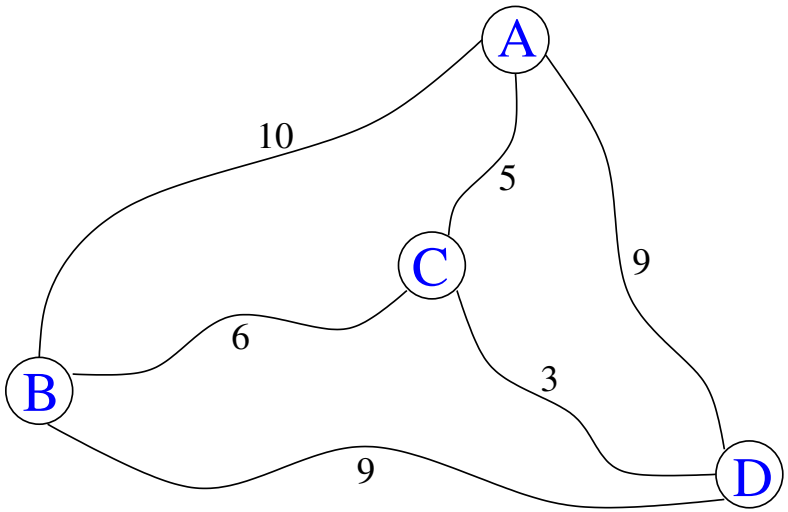
Programação Dinâmica

- Projeto de algoritmos por indução
- Subestrutura ótima
  Soluções ótimas obtidas por soluções ótimas de subproblemas
- Ocorrência de subproblemas repetidos
- Memorização das soluções de subproblemas
  muitas vezes por tabelas que facilitam acesso às soluções

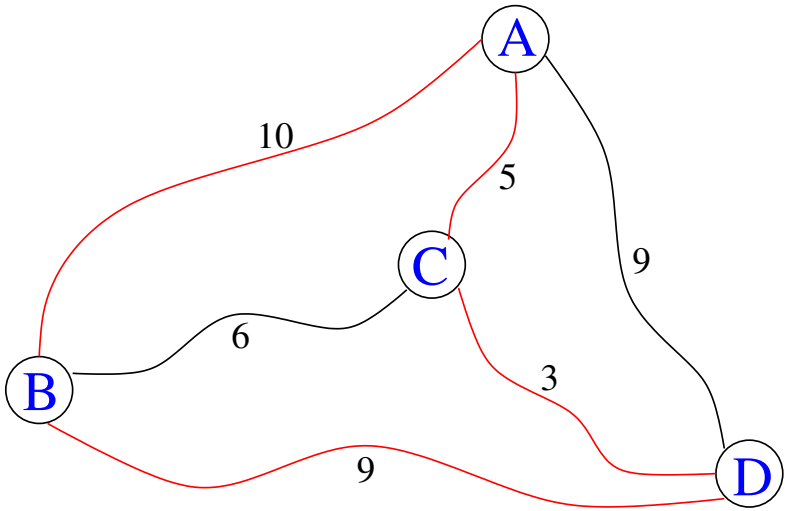
Problema do Caixeiro Viajante (TSP)

- Entrada:
  Grafo completo não-orientado com custos nas arestas
- Objetivo:
  Encontrar um *tour* (circuito hamiltoniano) de custo mínimo que visita cada vértice exatamente uma vez.

Entrada



Solução Ótima de custo 27



Problema do Caixeiro Viajante (TSP)

Ideia: Programação Dinâmica

- Projeto por indução no tamanho de caminhos
- Parecida com a ideia de caminhos mínimos de Bellman-Ford
- Usando subconjuntos para especificar vértices visitados.

Caminhos e projeto por indução:

- Indução nos subconjunto dos vértices
- Caminhos começando no vértice 1
- Caminhos terminando em um vértice $v \in V \setminus \{1\}$

Problema do Caixeiro Viajante (TSP)

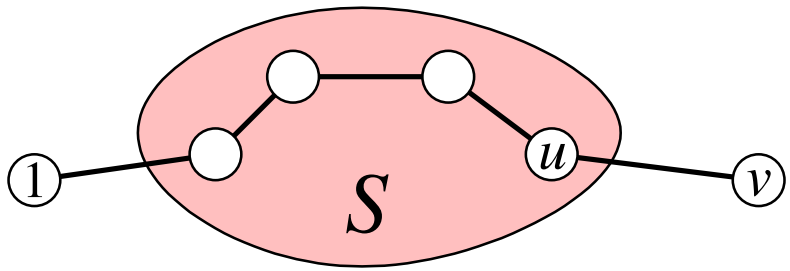
Notação:

$A[S, v]$ = Custo do caminho saindo de 1, passando por cada vértice de S exatamente uma vez terminando no vértice v, onde $1, v \notin S$.

V' = $V \setminus \{1\}$.

Programação Dinâmica:

- Nos subconjuntos de vértices.
- $A[S, v] = \min_{u \in S}\{A[S - u, u] + c(u, v)\}$



- Solução em $\min_{u \in V'} \left\{A[V' \setminus \{u\}, u] + c(u, 1)\right\}$.

Problema do Caixeiro Viajante (TSP)

TSP-PD (G, c), onde $G = (V, E)$ e $V = \{1, \ldots, n\}$.

1 $V' \leftarrow V \setminus \{1\}$

2 $A[\{\}, v] \leftarrow c_{1,v}$ para todo $v \in V'$.

3 Para $t \leftarrow 1$ até $|V'| - 1$ faça

4 Para cada $S \subseteq V'$ tal que $|S| = t$ faça

5 Para cada $v \in V' \setminus S$ faça

6 $A[S, v] \leftarrow \min_{u \in S}\{A[S \setminus \{u\}, u] + c_{u,v}\}$

7 $\varrho^* \leftarrow \min_{u \in V'} \left\{A[V' \setminus \{u\}, u] + c(u, 1)\right\}$.

8 Devolva ϱ^*

Problema do Caixeiro Viajante (TSP)

Lema

O algoritmo TSP-PD computa $A[S, v]$ como o caminho mínimo que sai de 1, passa por cada vértice de S exatamente uma vez e termina em v.

Prova. Por indução no tamanho de S.

Base: Para $|S| = 0$ o passo 2 computa corretamente $A[S, v]$.

H.I.: Suponha que $A[S, v]$ está correto para todo $|S| = t - 1$ e $v \in V' \setminus S$.

Seja $S \subseteq V'$ tal que $|S| = t$ e $v \in V' \setminus S$.

Seja P caminho mínimo ótimo saindo de 1, passando por cada vértice de S exatamente uma vez e terminando em v.

Seja P' o caminho P removendo v e seja u a extremidade final de P'.

Por H.I., temos $A[S \setminus \{u\}, u] \leq c(P')$. Portanto

$$c(P) = c(P') + c_{u,v} \geq A[S \setminus \{u\}, u] + c_{u,v} \geq A[S, v]$$

Pela otimalidade de P, temos $c(P) = A[S, v]$ e $A[S, v]$ está correto. □

Problema do Caixeiro Viajante (TSP)

Teorema

O algoritmo TSP-PD computa corretamente o custo de um circuito hamiltoniano de custo mínimo.

Prova. Seja $C^* = (1, v_{i_1}, \ldots, v_{i_{n-1}})$ circuito hamiltoniano de custo mínimo. Pelo lema anterior, $A[\{v_{i_1}, \ldots, v_{i_{n-2}}\}, v_{i_{n-1}}]$ é computado corretamente.

$$A[\{v_{i_1}, \ldots, v_{i_{n-2}}\}, v_{i_{n-1}}] \leq c(C^* - (v_{i_{n-1}}, 1))$$

Pela escolha de v no passo 7, o circuito ótimo tem custo ϱ^* tal que

$$\begin{aligned} \varrho^* &= \min_{v \in V'} \left\{ A[\, V' \backslash \{v\}\,,\, v] + c(v, 1) \right\} \\ &\leq A[\{v_{i_1}, \ldots, v_{i_{n-2}}\}, v_{i_{n-1}}] + c(v_{i_{n-1}}, 1) \\ &\leq c(C^* - (v_{i_{n-1}}, 1)) + c(v_{i_{n-1}}, 1) \\ &= c(C^*) \end{aligned}$$

Como C^* é otimo, temos $\varrho^* = c(C^*)$. □

Problema do Caixeiro Viajante (TSP)

Lema

A complexidade de tempo do algoritmo TSP-PD é $O^*(2^n)$.

Prova.

- Passos 3 e 4: percorre todos os subconconjuntos com cardinalidade 1 a $|V| - 1$: $O(2^n)$
- Passos 5 e 6: $O(n^2)$

Portanto, temos uma complexidade de $O(n^2 2^n)$ □

Problema do Caixeiro Viajante (TSP)

Exercício

Descreva um algoritmo que não apenas encontra o valor, mas também encontra um circuito hamiltoniano de menor custo.

Exercício

Um conjunto de tarefas $J = \{1, \ldots, n\}$ deve ser escalonado em uma máquina. Cada tarefa $j \in J$ tem um tempo de processamento $t_j > 0$ e um custo $c_j > 0$. A máquina não processa mais de uma tarefa por vez, e não interrompe uma tarefa em execução até sua finalização. Se uma tarefa j termina sua execução no tempo τ, seu custo é $\tau \cdot c_j$. O problema é encontrar um escalonamento das tarefas de maneira a minimizar o custo total das tarefas escalonadas. Projete um algoritmo com complexidade de tempo $O^*(2^n)$.

Coloração de Grafos

Definição

Dado grafo $G = (V, E)$, uma k-coloração de G é uma atribuição $\psi : V \to \{1, \ldots, k\}$ tal que $\psi(u) \neq \psi(v)$ para toda aresta $\{u, v\} \in E$.

Definição

Dado grafo G, o índice cromático de G é o menor inteiro k tal que G possui uma k-coloração.

Teorema

O problema de decidir se um dado grafo G possui uma k-coloração é um problema NP-Completo, *mesmo para $k = 3$.*

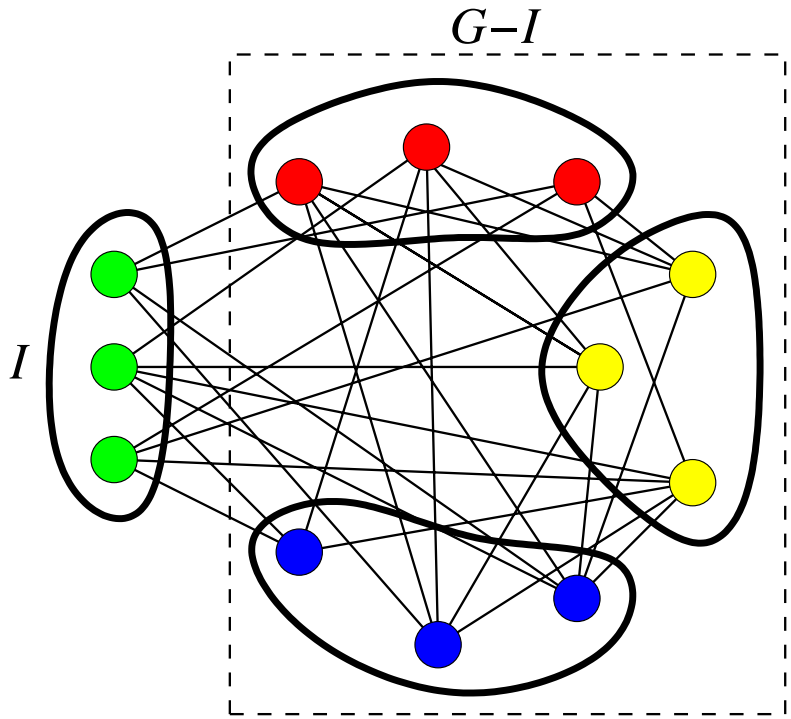
Problema (Coloração de Grafo)

Dado grafo G, encontrar uma k-coloração de G que minimiza k.

Coloração de Grafos

Ideia: Usando conjuntos independentes

Se $G = (V, E)$ é k-colorível e temos um conjunto independente I de uma das cores, aplique projeto por indução no grafo restante $G - I$.
I.e., $G[V \setminus I]$ é $(k-1)$-colorível.



Como não sabemos qual I, testamos todos possíveis.

Coloração de Grafos

Ideia: Usando conjuntos independentes e projeto por indução

COLORAÇÃO-PD-1(G), onde $G = (V, E)$ e $V = \{1, \ldots, n\}$.

1 $\mathcal{P} \leftarrow \{0,1\}^V$ (i.e., conjunto das partes de V)
2 $c[S] \leftarrow \infty$ para todo $S \in \mathcal{P} \setminus \{\emptyset\}$.
3 $c[\emptyset] \leftarrow 0$
4 Para $t \leftarrow 1$ até $|V|$ faça
5 Para cada $S \in \mathcal{P}$ faça
6 Para cada $I \subseteq S$ onde I é independente faça
7 $c[S] \leftarrow \min\{c[S], 1 + c[S \setminus I]\}$
8 Devolva $c[V]$

Coloração de Grafos

Fato

Algoritmo Coloração-PD-1 *devolve resposta correta em tempo* $O^*(3^n)$

Prova. Corretude: Exercício (após término da iteração t, do loop 5, se $G[S]$ tem índice cromático t, então $c[S] = t$.)
Loop do passo 5 percorre todos os subconjuntos S de vértices e do passo 6 percorre todos os conjuntos independentes de S.
A combinação dos passos 5 e 6 repete $T = \sum_{S \subseteq V} 2^{|S|}$ vezes:

$$\begin{aligned} T &= \sum_{S \subseteq V} 2^{|S|} \\ &= \sum_{s=0}^{n} \binom{n}{s} 1^{n-s} 2^s \\ &= (1+2)^n = 3^n \end{aligned}$$

O loop do passo 4 e o do passo 7 agregam fator polinomial e a complexidade computacional do algoritmo é $O^*(3^n)$. □

Coloração de Grafos

Ideia: Usar conjuntos independentes maximais: Existe coloração ótima com uma cor que é um conjunto independente maximal

Notação: Dado grafo G simples e não-dirigido com n vértices, denote por MIS(G) o conjunto de todos os conjuntos independentes maximais de G.

Teorema (Miller e Muller'60 e Moon e Moser'65)

Seja G um grafo simples. Então, $MIS(G) \leq g(n)$, onde

$$g(n) = \begin{cases} 3^{n/3} & \text{se } n \equiv 0 (\mod 3) \\ 4 \cdot 3^{(n-4)/3} & \text{se } n \equiv 1 (\mod 3) \\ 2 \cdot 3^{(n-2)/3} & \text{se } n \equiv 2 (\mod 3) \end{cases}$$

Teorema

Existe algoritmo que obtém $MIS(G)$ em tempo $O^*(MIS(G))$ (i.e., em $O^*(1{,}4423^n)$.)

Coloração de Grafos

Ideia: Por conjuntos independentes maximais.

COLORAÇÃO-PD (G), onde $G = (V, E)$ e $V = \{1, \ldots, n\}$.

1 $\mathcal{P} \leftarrow \{0,1\}^V$ (conjunto das partes de V)
2 $c[S] \leftarrow \infty$ para todo $S \in \mathcal{P} \setminus \{\emptyset\}$.
3 $c[\emptyset] \leftarrow 0$
4 Para $t \leftarrow 1$ até $|V|$ faça
5 Para cada $S \in \mathcal{P}$ faça
6 Para cada $I \subseteq S$ onde I é independente maximal em $G[S]$ faça
7 $c[S] \leftarrow \min\{c[S], 1 + c[S \setminus R]\}$
8 Devolva $c[V]$

Coloração de Grafos

Teorema (Lawler'76)

Coloração-PD *devolve a resposta correta em tempo* $O^*(2{,}4423^n)$.

Prova. No passo 7, o número de conjuntos independentes maximais em $G[S]$ é no máximo $3^{|S|/3}$.
Assim, os passos 6 e 7 levam a seguinte quantidade T de iterações:

$$\begin{aligned} T &\leq \sum_{s=0}^{n} \binom{n}{s} 3^{s/3} \\ &= \sum_{s=0}^{n} \binom{n}{s} 1^{n-s} (3^{1/3})^s \\ &= (1+3^{1/3})^n \\ &\leq 2{,}4423^n. \end{aligned}$$

Obs.: $(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k$ □