L.EIC
#2.2

Desenho de Algoritmos

Resumos

— Brute - force : Explorar todas as combinações

1. Greedy
2. Divide and Conquer
3. Dynamic Programming
4. Branch and Bound
5. NP e NP - Complete
6. Algoritmos de aproximação
7. Programação linear
8. Programação linear inteira

1. Greedy

Propriedades

- "Optimal greedy choice" | A solução global pode ser derivada de várias soluções locais
- "Optimal sub-structure" | A solução ideal contém soluções ideais para subproblemas

"Fractional Knapsack"

```
fn (v, w, m) {
// values <list>, weights <list>, maxWeight <int>
p = {}  // itemsPicked <list>
curr = 0  // currentWeight <float>
while curr < w:
    i = maxValuePerWeight (v, w, p)
    if (w[i] + curr ≤ w):
        p[i] = 1
        weight += w[i]
    else:
        p[i] = (m - curr) / w[i]
        weight = m
return p }
```

Item com melhor $\frac{v[i]}{w[i]}$ que não tenha sido escolhido

- **Otimalidade: Escolha gananciosa**
  A solução ideal contém o máximo possível de items com o maior valor por peso $\left(\frac{v[i]}{w[i]}\right)$.

- **Otimalidade: Subproblema ideal**
  Assumindo uma solução ideal X para um problema, um subproblema sem o item i terá como solução ideal $X' = X - X[i]$.

- "**Integer knapsack**" não dá uma solução ideal

Ex//

$v = [10, 6, 6]$
$w = [3, 2, 2]$
$m = 4$

Resultado		Solução	
[1, 0, 0]	value = 10	[0, 1, 1]	value = 12

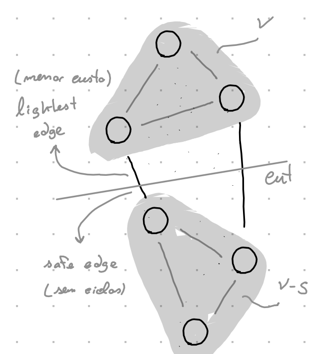
"Minimum-Cost Spanning Tree"

- **Propriedades**
  → A aresta mais pesada de um **ciclo** nunca faz parte de uma MST.

  → A aresta mais leve que une dois conjuntos separados por um **corte** faz sempre parte de uma MST.

- **Algoritmo genérico**

```
fn (g) {
// graph with weights
tree = ∅  // Same vertices
while !isST(tree):   → acyclic, connected
    edge = findSafeEdge (tree)
    tree.addEdge (edge)
return tree }
```



DA

Prim

```
fn (g, r) { // Graph, root <vertex>
q = priorityQueue (g.vertexs ) // Comparing keys
for v ∈ q :
    v.key = ∞ // Lowest edge weight connecting
r.key = 0            v to the tree
r.pred = NIL // Node predecessor
while ! q.empty () :
   u = q.top () // And removes from q
   for v ∈ u.adjVertexs :
      if v ∈ q & g.weight(u,v) < v.key :
         v.pred = u
         v.key = g.weight (u,v) // And update q
// Answer inside "pred" of the vertexs }
```

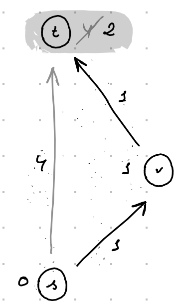
- $O(E \log(V))$
  usando uma binary heap em q

Kruskal

```
fn ( g ) {
A = ∅ // Same vertexs
sets = ∅ // Ex: Union-Find Disjoint Sets
for v ∈ g.vertexs :
   sets.newSet (v)
sort ( g.edges ) // By weight
For (u, v) ∈ g.edges :
   if sets.differentSets (u, v) :
      A.addEdge (u, v)
      sets.merge (u, v)
return A }
```

- $O(E \log(V))$
  usando listas para representar conjuntos

"Single-Source Shortest Path"



- **Relaxamento de acestas**

```
fn g.relax (v, t) {
if t.dist > v.dist + g.weight (v,t):
   t.dist = v.dist + g.weight (v,t )
   t.pred = v }
```

-
```
fn init (g, s) {
for v ∈ g.vertexs :
   v.dist = ∞
   v.pred = NIL
s.dist = 0 }
```

Dijkstra

```
fn (g, s) {
init (g, s)
q = priorityQueue (g.vertexs) // Comparing dist
while !q.empty () :
   u = q.top () // And removes from q
   u.visited ()
   For v ∈ u.adjVertexs :
      g.relax (u, v) // And updates q
// Answer inside "pred" of the vertexs }
```

⚠ Não funciona em acestas com pesos negativos

Bellman - Ford - Moore

```
fn (g, s) {
init (g, s)
repeat g.vertexs.size() -1 :
   for (u, v) ∈ g.edges :
      g.relax ( u, v)
for (u, v) ∈ g.edges :
   if v.dist > u.dist + g.weight (u,v) :
      return false // Negative cycle
return true // Answer inside "pred" of the vertexs }
```

⚠ Não funciona com ciclos negativos

Dijkstra

- $O((V+E)\log(V))$, usando uma binary heap em q

- O algoritmo irá escolher o vértice v com menor distância de s e relaxar as arestas adjacentes a v até visitar todos os vértices.

- Todas as arestas são relaxadas, no máximo, uma vez

Bellman - Ford - Moore

- $O(VE)$, devido ao relaxamento das arestas #V vezes

- O algoritmo irá relaxar #V - 1 vezes todos os caminhos de s até um vértice v, isto é, considerar todos os caminhos de s até v, desde o direto até aquele que passe por todos os vértices. Se for possível relaxar ainda mais, estamos perante um ciclo.

DAGs

```
fn (g, s) {
init (g, s)
sortByTopologicalOrder (g.vertexs)
for u ∈ g.vertexs :
    for v ∈ u.adjVertexs :
        g.relax (u, v)
// Answer inside "pred" of the vertexs }
```

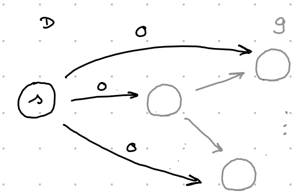
⚠ Só funciona com grafos direcionados e acíclicos

- $O(V+E)$, devido à ordenação

- Ao ordenar por ordem topológica, só é necessário relaxar cada aresta no máximo uma vez. Raciocínio semelhante a Dijkstra.

"All-Pairs Shortest Path"

```
fn (g) {
d = derive (g) // An auxilior graph with a
                  new vertex s connected to all
                  vertexs via edges with no weight
s = d.vertexs[0] // Vertex s from before
if !Bellman-Ford (d, s):
    return False // Negative cycle detected
for e ∈ d.edges:
    e.weight += e.orig().dist - e.dest().dist
for v ∈ d.vertexs:
    Dijkstra (d, v)
    for u ∈ d.vertexs:
    d.edge(u,v).weight += v.dist - u.dist
return d }
```



- **Método de Johnson**

$w'(u,v) = w(u,v) + h(u) - h(v)$

$w(u,v) = w'(u,v) + h(v) - h(u)$

$\approx$

e.weight += u.dist - v.dist

e.weight += v.dist - u.dist

- **Complexidade temporal**

$O(V(V+E)\log(V))$

"Max - Flow"

- **Método de Ford - Fulkerson**

```
fn (g, s, t) { // Graph, source, sink
 For e ∈ g.edges :
   e.flow = 0
 P = ∅   // Augmenting Path
 while isAnAugmentingPath (g, &P) :
   flow = flow Along (P)  // Max Flow From P
   augmentFlow (s, t, flow)
 return g.flow  // Max Flow / Min cut
```

- $O(E\,|f^*|)$, sendo $|f^*|$ o nº de caminhos limitados pelo flow máximo

- Para valores racionais, basta converter para inteiros pelo mínimo múltiplo comum.

- Para valores irracionais, pode nunca terminar ou convergir para a solução

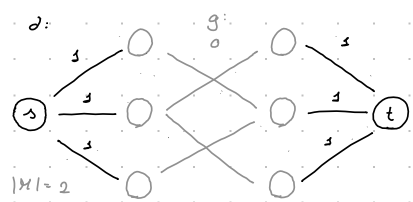
- **Algoritmo de Edmonds - Karp**

```
Fn isAnAugmentingPath (g, &P){
 // BFS search, looking for:
 // - v.adjVertices with residual > 0
 // - v.incoming with flow > 0 }

fn augmentFlow (s, t, flow){
 while (s ≠ t) :
   if (s.path is incoming) :
     e.flow += flow
     s = s.path.origin
   else :
     e.flow -= flow
     s = s.path.dest }

fn Flow Along (P) {
  return min (P.edges.flow) }
```

- Em casos extremos, $|f^*|$ pode ser enorme:

"Maximal Bipartite Matching"

```
fn (g) {
d = copyGraph (g)
d.setweights (0) // For every vertex
d.addVertex (s)
for v ∈ d.leftVertices() :
   d.addEdge (s, v, 1)
d.addVertex (t)
for v ∈ d.rightVertices () :
   d.addEdge (v, t, 1)
return MaxFlow (d) }
```



- $O(VE)$, pois $|f^*| = V$ ao usar o algoritmo de Ford - Fulkerson.

- Assumindo um "matching" M em G e um "flow" f em D:
  - → Todas as arestas em M têm um "flow" de 1 e as restantes têm um "flow" de 0;
  - → Todos os vértices de G são disjuntos.

  Logo, existem M caminhos que contribuem em 1 para o "flow" total do grafo D, isto é, $|M| = |f|$

2. Divide and Conquer

Teorema Mestre

$$T(n) = \begin{cases} d, & \text{se } n \text{ for constante} \\ aT\left(\frac{n}{b}\right) + f(n) \end{cases}$$

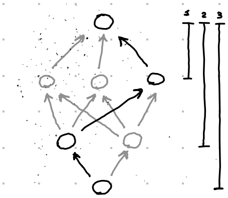
(*)
- $a \geq 1$
- $b > 1$
- f assintoticamente positiva
- $f \in O(n^c)$

Ex// $T(n) = 2T\left(\frac{n}{2}\right) + n$
↳ $O(n \log n)$ [Merge Sort]
$T(n) = 2T(n-1) + 1$
↳ $O(2^n)$ [Hanoi towers]

- $\log_b a < c \mid T(n) \in O(n^c)$
- $\log_b a = c \mid T(n) \in O(n^c \cdot \log n)$
- $\log_b a > c \mid T(n) \in O(n^{\log_b a})$

3. Programação Dinâmica

Princípio de otimalidade



Numa solução ótima, qualquer porção dela é uma solução ótima em relação ao espaço de escolhas correspondente

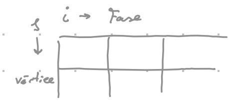
Corolário: Uma solução parcial subótima não precisa de ser mais explorada

Relação de recorrência

Ex// Shortest Path

$$cost(i, j) = \min_{\substack{k \in V_{i+1} \\ (j,k) \in E}} \{ c(j,k) + cost(i+1, k) \}$$

Cálculo do estado atual, com base nos resultados anteriores.



A implementação desta "memorização" é feita usando tabelas:

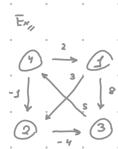
All-Pairs Shortest Path

$d_{ij}^{(k)} = \min($
$\quad d_{ij}^{(k-1)},$
$\quad d_{ik}^{(k-1)} +$
$\quad d_{kj}^{(k-1)}$
$)$

$O(V^3)$

(*) ij: Caminho de i a j
$D^{(k)}$: Tabela em que k é um vértice intermédio entre o caminho ij
Π: Tabela de precedentes

Ex//



$D^{(0)}$ (i →, j ↓)

	1	2	3	4
1	0	3	8	∞
2	∞	0	-4	∞
3	∞	∞	0	5
4	2	-1	∞	0

...

$D^{(4)}$

	1	2	3	4
1	0	3	-1	13
2	2	0	-4	1
3	7	4	0	5
4	2	-1	-5	0

0/1 Knapsack

$val[i, j] = \max($
$\quad val[i-1, j],$
$\quad val[i-1, j-w_i] + v_i$
$)$

$O(n \cdot W)$ **Pseudo-polinomial!**

(*) i: Item a incluir
j: Peso limite
v_i: Valor do item
w_i: Peso do item

Ex//

i	v_i	w_i
1	8	3
2	5	2
3	5	2

$W = 4$ Stock = 1

(j →, i ↓)

		1	2	3	4
	0	0	0	0	0
1	0	0	0	8	x
2	0	0	5	8	x
3	0	0	5	8	$\boxed{10}$

Longest Common Sequence

$$len[i, j] = \begin{cases} 0 & \text{se } i = 0 \vee j = 0 \\ len[i-1, j-1] + 1 & \text{se } x_i = y_j \quad \nwarrow \\ \max(len[i-1, j], len[i, j-1]) & \text{se } x_i \neq y_j \quad \leftarrow\uparrow \end{cases}$$

(*) i: Posição de X
j: Posição de Y
X, Y: Sequências

$O(n \cdot m)$

Ex//

X = ABCB

Y = BDCA

(j →, i ↓)

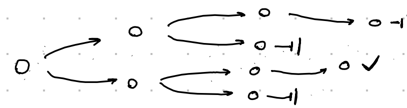
		B	D	C	A
	0	0	0	0	0
A	0	0 ←↑	0 ←↑	0 ←↑	1 ↖
B	0	1 ↖	1 ←	1 ←	1 ←↑
C	0	1 ↑	1 ←	2 ↖	2 ←
B	0	1 ↖	1 ←↑	2 ↑	$\boxed{2 ←↑}$

∴ Z = BC

DA

4. Brand - And - Bound

Backtracking



Metodicamente experimentar várias sequências de decisões até encontrar a melhor.

Bounding

Evitar explorar caminhos desnecessários e que garantidamente não darão um melhor resultado

Torna "backtracking" eficiente com as funções apropriadas.

Sum of subsets

- **Boundings:**
  - → Quando a soma de todos os elementos restantes for inferior a M

    $$\sum_{i=1}^{k} W(i)\,X(i) + \sum_{i=k+1}^{n} w(i) < M$$

  - → Quando M é ultrapassado ao adicionar o próximo elemento
    (Os valores devem estar ordenados)

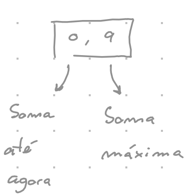
    $$\sum_{i=1}^{k} w(i)\,x(i) + W(k+1) > M$$

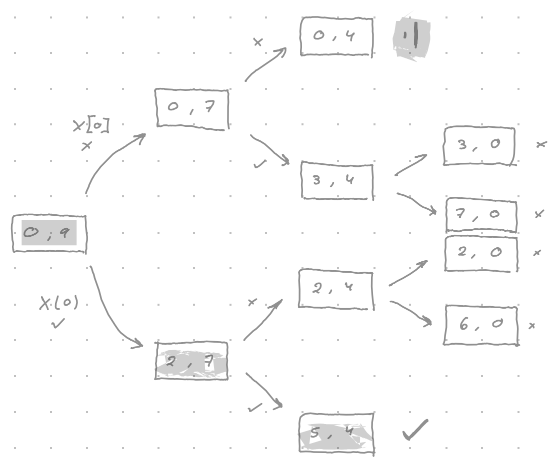
⊛ W : Conjunto de elementos
k : Nº de elementos
M : Soma exigida
X : Conjunto booleano que indica se um elemento foi escolhido

Ex:

W = [2, 3, 4]
k = 3
M = 5

⊛





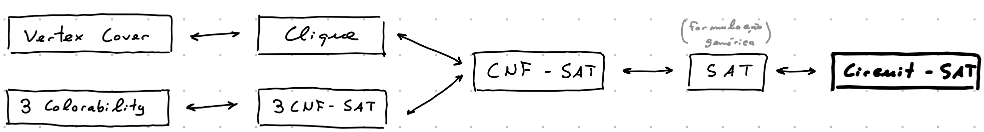
5. NP e NP-Complete

Classes de problemas

- **Trackable** | Resolvido em tempo polinomial
- **(NP) Non-Deterministic Polynomial Time** | Possível confirmar a resposta em tempo polinomial
- **NP-Hard** | Um problema que é possível ser reduzido de um problema NP-Hard já existência
- **NP-Complete** | NP e NP-Hard

"recast"
  - **Otimização** | Encontrar o máximo / mínimo, tendo em conta um número de condições
  - **Decisão** | Impor restrições e perguntar se o problema ainda é viável

- Um problema de otimização não é mais fácil que o seu problema de decisão
- O problema **Circuit-SAT** é o problema NP-Complete base
- É possível usar qualquer um dos problemas NP-Hard para reduzir
- Basta encontrar um algoritmo polinomial para um dos problemas NP-Complete para provar que P = NP



Provar que um problema é NP-Complete

1. Mostrar que o problema (M) é NP (verificável em tempo polinomial)
2. Escolher L: um problema NPC apropriado
3. Descrever uma função f que mapeie qualquer instância x de L para uma instância $f(x)$ de M
4. Provar a correção de f ($x \in L$ se e só se $f(x) \in M$)
5. Provar a complexidade polinomial do algoritmo de f, em função do tamanho da instância de L

Circuit-SAT $\leq_p$ SAT (27)

Gerar expressão booleana com base no circuito dado, em que cada fio é uma variável.

CNF SAT $\leq_p$ 3CNF SAT (34)

Converter condições para conter exatamente 3 literais. P.e. $(l_1 \vee l_2) \rightarrow (l_1 \vee l_2 \vee y_1) \wedge (l_1 \vee l_2 \vee \neg y_1)$

CNF SAT $\leq_p$ Clique (40)

Grafo com grupos de condições, em que cada variável é um nó. Cada nó está ligado a todos, exceto o complemento.

Clique $\leq_p$ Vertex Cover (47)

Gerar um grafo auxiliar com o complemento das arestas e esperar uma cobertura de $|V| - k$.

3CNF SAT $\leq_p$ 3 Colorability (52)

Grafo com o triângulo "TBF", triângulo "$x\ \bar{x}\ B$" para cada variável e um sub-grafo ligado a "F" por cada condição.

6. Algoritmas de Aproximação

$\rho(n)$ - Approximation

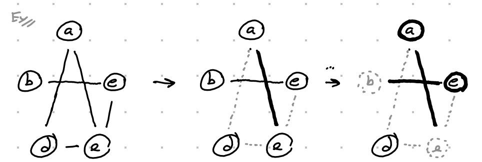
$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n)$$

Notação usada para representar o rácio do custo aproximado e do custo ótimo.
Idealmente, $\rho(n)$ aproxima-se a 1

Vertex - Cover : 2-Approximation Algorithm

- Selecionar uma aresta aleatória e remover as já cobertas até ficar sem arestas disponíveis.
- Heurística: Remover nós redundantes, isto é, que todos os nós vizinhos já estejam presentes na resposta.

Ex//



0/1 knapsack : 2 - Approximation Algorithm

- Selecionar o melhor de dois algoritmos:
  - → Escolher os items com a maior densidade (valor / peso);
  - → Escolher os items com o maior valor.

Ex// $W = 5$

$v = \{5, 8, 9\}$

$w = \{1, 2, 3\}$

$\frac{v}{w} = \{5, 4, 3\}$

Maior densidade:

Ⓐ $\{1, 2\} \rightarrow 13$

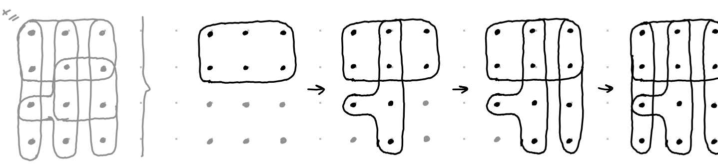
Maior valor:

Ⓑ $\{3, 2\} \rightarrow 17$ ✡

Set - Cover : Polynomial-time Approximation Algorithm

- Selecionar o conjunto que cubra o maior nº de novos pontos
- "Touch-up" disponível

Ex//



TSP : 2 - Approximation Algorithm

- Selecionar um vértice como a raíz
- Calcular a MST a partir da raíz
- Gerar um "pre-order walk" da MST
- Gerar o circuito a partir do caminho anterior, mas saltando para o próximo vértice por visitar, se necessário

⚠ O grafo tem de ser completo.

⚠ A desigualdade triangular tem de ser válida para todas as arestas:
custo $(u, v) \leq$ custo (u, t) + custo (t, v)

- Não existe uma aproximação para instâncias gerais do TSP.

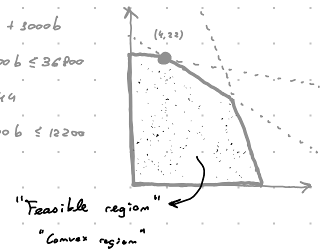
DA

7. Programação Linear

Dado uma função que represente o objetivo e uma lista de restrições, é possível resolver o problema geometricamente:

- Assume-se que todas as variáveis $\in \mathbb{R}$
- Se existir uma solução, ocorre sempre na borda e pode não ser única
- A "convex region" pode ser **vazia** devido às restrições ou "**unbounded**" por falta destas

Ex//
$c = 3000a + 3000b$
$400a + 1600b \leq 36800$
$2a + b \leq 44$
$300a + 500b \leq 12200$
$a, b \geq 0$



Conversão para "Standard Form"

- A função é sempre maximizante
  - ↳ Multiplicar coeficientes por -1

Ex// $\min 2x_1 - 3x_2 \rightarrow \max -2x_1 + 3x_2$

- Todas as variáveis têm restrições
  - ↳ Substituir por duas variáveis com restrições ≤ 0, em que uma delas tem o sinal oposto

Ex//
$$\left.\begin{array}{l} \max 2x_1 - 3x_2 \\ x_1 - 2x_2 \leq 4 \\ x_1 \geq 0 \end{array}\right\} \quad \begin{array}{l} \max 2x_1 - 3x'_2 + 3x''_2 \\ x_1 - 2x'_2 + 2x''_2 \leq 4 \\ x_1, x'_2, x''_2 \geq 0 \end{array}$$

- Todas as restrições são $\leq$
  - ↳ Substituir $=$ por duas restrições: $\leq$ e $\geq$
  - ↳ Substituir $\geq$ por $\leq$ ao multiplicar coeficientes por -1

Ex// $x_1 + x_2 = 7 \rightarrow x_1 + x_2 \geq 7$; $x_1 + x_2 \leq 7$

Ex// $x_1 + x_2 \geq 7 \rightarrow -x_1 - x_2 \leq 7$

Conversão para "Slack Form"

- Todas as restrições são de igualdade
  - ↳ Criar uma nova variável, cujo valor é igual ao da restrição

Ex//
$$\left.\begin{array}{l} \max 2x_1 - 3x_2 \\ x_1 + x_2 \leq 7 \\ x_1, x_2 \geq 0 \end{array}\right\} \quad \begin{array}{l} z = 2x_1 - 3x_2 \\ x_3 = 7 - x_1 - x_2 \\ x_1, x_2, x_3 \geq 0 \end{array}$$

"Basic variable" "Non-Basic Variable"

- **"Slack Form"**
  - → z | Variável a maximizar
  - → N | Nº de "Non-Basic variables" $= 2$
  - → B | Nº de "Basic variables" $= 1$
  - → A | Matriz de coeficientes $= \{\{-1, -1\},\}$
  - → b | Vetor de constantes $= \{7\}$
  - → c | Vetor de coeficientes da função $= \{2, -3\}$
  - → v | Constante da função $= 0$

Algoritmo Simplex (Exemplo)

$z = 18x_1 + 12{,}5\,x_2$
$x_3 = 20 - x_1 - x_2$
$x_4 = 12 - x_1$
$x_5 = 16 - x_2$
$x_1, x_2, x_3, x_4, x_5 \geq 0$

Initial Solution:

$(0, 0, 20, 12, 16)$

$\downarrow$ x_3 "leaving" x_4 "entering" $\begin{array}{ccc} x_3 & x_4 & x_5 \\ (20, & 12, & \infty) \end{array}$

$z = -18x_4 + 12{,}5\,x_2 + 216$
$x_3 = 8 - x_4 - x_2$
$x_3 = 12 - x_4$
$x_5 = 16 - x_2$
...

$\downarrow$ x_2 "leaving" x_3 "entering" $\begin{array}{ccc} x_1 & x_3 & x_5 \\ (\infty, & 8, & 16) \end{array}$

$z = -18x_4 + 12{,}5\,(8 - x_4 - x_3) + 216 \Rightarrow z = 316 - 12{,}5\,x_3 - 30{,}5\,x_4$
$x_2 = 8 - x_4 - x_3$
$x_1 = 12 - x_4$
$x_5 = 8 + x_4 + x_3$

$\therefore$ Solution:

$\begin{array}{ccccc} x_1 & x_2 & x_3 & x_4 & x_5 \\ (12, & 8, & 0, & 0, & 8) \end{array}$

8. Programação Linear Inteira

- A solução já pode não estar nas margens
- ILP é um problema NP-Complete (redução a 3-CNF-SAT, p.e.)

Branch and bound

- Resolver inicialmente o problema LP
- Pegar na variável com maior fração e gerar dois problemas LP: "upper-bound" e "lower-bound"
- Explorar recursivamente esta árvore binária de problemas até: **(Fathoming)**
  → Todas as variáveis do resultado são inteiras (possível solução)
  → Não resultar em nenhuma resposta válida
  → A solução for mais baixa que a melhor solução encontrada até então

Ex: Slides 23 a 27