

Algoritmos e Estruturas de Dados  
IF672cc – 2016.2 CIn – UFPE  
Pedro Tôrres phts@cin.ufpe.br

## Estrutura de dados

### Tipos primitivos de dados

- Tipos básicos para valores atómicos
- Fornecidos com a linguagem de programação

- int
- float
- boolean
- char
- ⋮

### Tipos abstratos de dados

- Modelo matemático para dados estruturados
- "Encapsulam" informações primitivas
- Semântica e/ou operações associadas
- Classificação

↳ Unidimensional  
1 coordenada para  
localizar componente

Multidimensional  
 $n$ -coordenadas

↳ Homogênea  
componentes do  
mesmo tipo

Heterogênea  
tipos diferentes

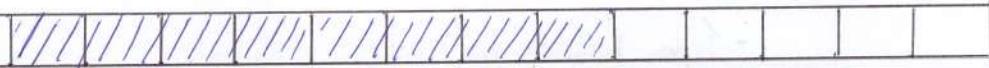
↳ Estática  
quantidade fixa  
de memória

Dinâmica  
quantidade variável

• Estrutura unidimensional homogênea estática de dados

$$d = \{ d_0, d_1, \dots, d_{n-1} \}$$
$$d_i = d[i]$$

- Representados na memória por bloco de posições consecutivas



Array de 2 ints

1 int = 4 bytes

posição inicial  $\downarrow$

$$\text{posição}(a[i]) = \text{posição}_{\text{inicial}} + i \times s$$

$\hookrightarrow$  espaço de alocação unitário

- + Cada elemento pode ser acessado instantaneamente
  - $\hookrightarrow$  tempo constante

- Algoritmo média ponderada

Entrada:  $V = (v_0, \dots, v_{n-1}) \in \mathbb{R}^n_+$  valores  
 $w = (w_0, \dots, w_{n-1}) \in \mathbb{R}^n_+$  pesos

Saída: A média dos valores em  $V$  ponderados pelos pesos em  $w$

inicio

$sv \leftarrow \emptyset$

$sw \leftarrow \emptyset$

para  $i \in \emptyset, \dots, n-1$  faça

$sv \leftarrow sv + (v[i] * w[i])$

$sw \leftarrow sw + w[i]$

Fim-faça

retorna  $(sv/sw)$

Fim

- Estrutura homogênia estática multidimensional de dados

- Representados na memória por bloco de posições consecutivas

$$\text{posição}(v[i,j]) = \text{posição}_{\text{inicial}} + (i \times nx s) + (j \times s)$$

• Algoritmo multiplicação de matrizes

Entrada:  $A_{m \times s}$ ,  $B_{s \times n}$

Saída:  $C_{m \times n} = AB$

Início

para  $i \leftarrow 0, \dots, m-1$  faça

    para  $j \leftarrow 0, \dots, n-1$  faça

$C[i, j] \leftarrow \emptyset$

        para  $k \leftarrow 0, \dots, s-1$  faça

$C[i, j] \leftarrow C[i, j] + A[i, k] \times B[k, j]$

    Fim-faça

  Fim-faça

  retorna  $C$

Fim

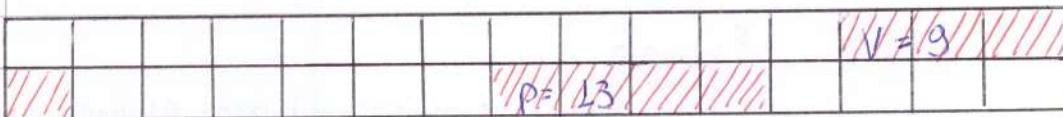
## A ponteiros / Ponteiros e Alocação Dinâmica

- Ponteiro

Tipo de dado usado para variáveis que armazenam um endereço de memória

+ Em sistemas de arquitetura 32-bits, os ponteiros são inteiros de 32 bits (4 bytes), limitando a memória do sistema a aproximadamente 4 GB

• Exemplo



g - variável inteira

o valor de g é 9

o endereço de g é 13

p - variável ponteiro

p = 13

&p = 23

p = &g

\*p = g

+ Ponteiro nulo =  $\perp$

## • Estruturas Lineares Dinâmicas

- Semântica básica

Colégio dinâmica linear de elementos do mesmo tipo

- Listas Simplesmente Encadeadas

- Referência externa

1º elemento

- Referência interna (encadeamento)

elemento posterior

- Acesso

irrestrito

- Representação explícita com ponteiros

- Tipo de Dados Nó (registrar, estrutura, ...)

- Conjunto finito de campos

heterogêneo

- Cada campo tem um nome (identificador)

value	next
-------	------

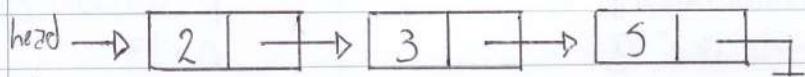
|                          ↳ ponteiro para elemento posterior

↳ valor do nó

conteúdo armazenado na lista

• Exemplo

Lista com 3 primeiros números primos



|                          ↳ ponteiro para o 1º elemento  
referência externa

+ Notação

$N.x$

valor do campo  $x$  no nó  $N$

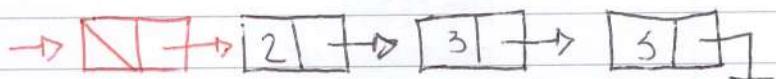
$P \rightarrow x$

valor do campo  $x$  do nó apontado por  $P$

• Operações

- Localizar elemento pela posição

+ Representação alternativa (por conveniência)



↳ elemento sentinel

valor não interessa

não pertence logicamente à lista

• Algoritmo get\_element

Entrada: head ponteiro para a cabeça da lista (sentinel)  
 $pos \geq 0$

Saída: referência para o nó na posição  $pos$ , se existir, ou  $\perp$

início

$i \leftarrow \emptyset$

$cur \leftarrow head$

enquanto  $i < pos$  e  $cur \neq \perp$  faça

$cur \leftarrow (cur \rightarrow next)$

$i \leftarrow i + 1$

fim-faça

retorna  $cur$

fim

+ a referência para um elemento da lista é fisicamente

- Localizar elemento pelo valor

• Algoritmo find element

Entrada: head

∨ valor procurado

Saída: referência para o 1º elemento de valor, se existir, ou  $\perp$

início

cur  $\leftarrow$  head

enquanto cur  $\rightarrow$  next  $\neq \perp$  e cur  $\rightarrow$  next  $\rightarrow$  value  $\neq$  v faça

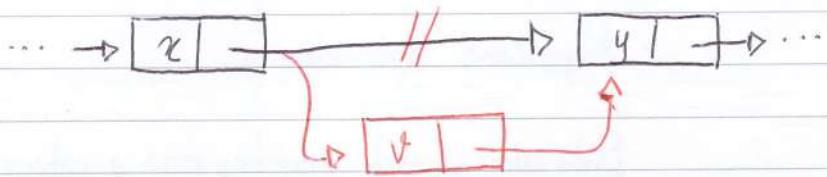
    cur  $\leftarrow$  cur  $\rightarrow$  next

fim-faça

retorna cur

fim

- Inserir elemento



• Algoritmo insert element

Entrada: cur referência para o elemento corrente  
v valor a inserir

Saída: referência para "novo" elemento corrente

início

N  $\leftarrow$  novo nó

N.value  $\leftarrow$  v

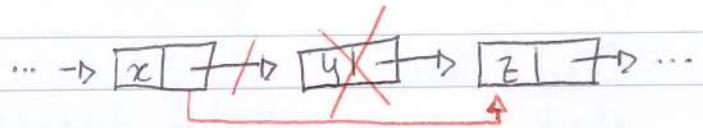
N.next  $\leftarrow$  cur  $\rightarrow$  next

cur  $\rightarrow$  next  $\leftarrow$  & N

retorna cur

fim

- Remover elemento



• Algoritmo remove element

Entrada: cur referência para posição corrente ( $cur \rightarrow next \neq \perp$ )

Saída: é "nova" referência para posição corrente após remoção  
o valor removido

início

$p \leftarrow cur \rightarrow next$

$v \leftarrow p \rightarrow value$

$cur \rightarrow next \leftarrow p \rightarrow next$

$dispose(p)$  // libera memória do nó apontado por p

retorna (cur, v)

fim

- Pilhas

- Referência externa

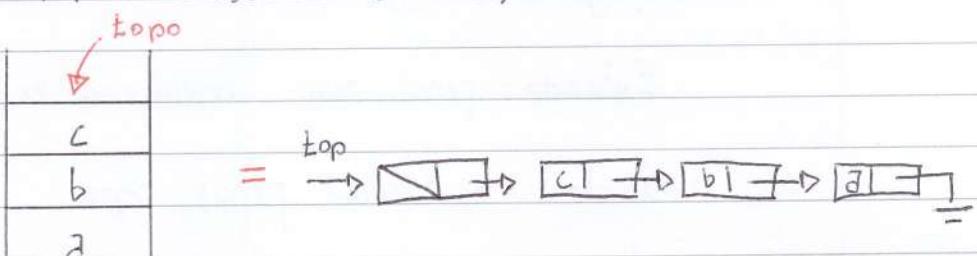
último elemento

- Encadeamento

referência ao anterior

- Política de acesso

LIFO (Last In First Out)



- Operações

- Desempilhar

stack-pop(top)  $\equiv$  list-delete(top)

- Empilhar

stack-push(top, v)  $\equiv$  list-insert(top, v)

- Filas

- Referências externas

primeiro elemento

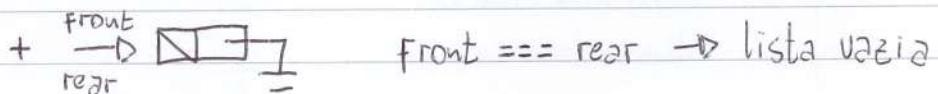
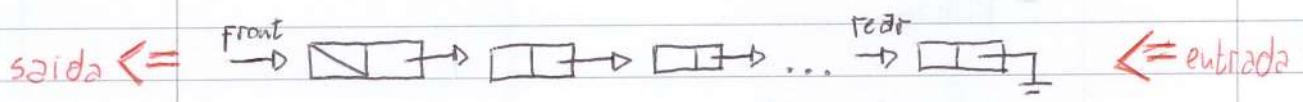
último elemento

- Encadeamento

referência ao posterior

- Política de acesso

FIFO (First In First Out)



- Operações

- Enfileirar

- Algoritmo enqueue

Entrada: front, rear referências externas da fila

Saída: os "novos" front, rear

início

$r \leftarrow \text{list-insert}(\text{rear}, v) \rightarrow \text{next}$   
retorna (front, r)

- Desenfileitar

• Algoritmo de dequeque

Entrada: front, rear ( $\text{front} \neq \text{rear}$ )

Saída: os "novos" front, rear após o desenfileiramento  
o valor desenfileirado

Início

$f, v \leftarrow \text{list-delete}(\text{front})$

se  $f \rightarrow \text{next} = \perp$  então

$r \leftarrow f$

senão

$r \leftarrow \text{rear}$

Fim-se

retorna  $(f, r, v)$

Fim

# Introdução à Análise de Algoritmos

## Critérios

- Tempo de execução
- Espaço de memória
- ⋮

## Abordagem analítica

Tempo (espaço, ...) definido matematicamente como função das entradas

$$T: \mathcal{J} \rightarrow \mathbb{N}$$
$$x \mapsto T(x)$$

onde:

$\mathcal{J}$  - espaço de parâmetros que definem o tamanho da entrada

$T(x)$  - tempo de execução para entradas de tamanho  $x$  em número de operações básicas

## Tamanho da entrada

Definido pelos parâmetros preponderantes para o número de operações, depende do problema

## Exemplo 1

### Algoritmo factorial

Entrada:  $n \in \mathbb{N}$

Saída:  $n!$

inicio

fat  $\leftarrow 1$

m  $\leftarrow n$

enquanto  $m > 0$  Faça

    fat  $\leftarrow$  fat  $\ast$  m

    m  $\leftarrow$  m - 1

fim - Faça

retorna fat

fim

} 2 atribuições

} 1 comparação  $\ast (n + 1)$

} (2 atribuições  $\ast n$ )  $\ast n$

$$T(n) = 5n + 2$$

Tamanho da entrada = magnitude de n

## Exemplo 2

### Algoritmo média

Entrada :  $V = \{v_0, v_1, \dots, v_{n-1}\}$

Saída :  $\bar{V} = \sum_{i=0}^{n-1} v_i / n$

início

$s \leftarrow \emptyset$

} 1 atribuição

para  $i \leftarrow \emptyset, \dots, n-1$  faça

} 1 comparação, 1 operação aritmética, 1 atribuição } n

$s \leftarrow s + v[i]$

} 1 operação aritmética, 1 atribuição

fim-faça

retorna ( $s/n$ )

} 1 operação aritmética

Fim

$$T(n) = 5n + 2$$

Tamanho de entrada = n° elementos do vetor

+ O número exato de operações e o custo de cada uma delas depende do modelo de computação

### Custos típicos

$$T(n) = C$$

mais eficiente

$$C \log_2(n)$$

$$Cn$$

$$Cn \log_2(n)$$

$$Cn^k, k \geq 2$$

$$CK^n$$

menos eficiente

### Casos medidos

#### Exemplo

list-find (head, v) numa lista de n elementos

Qual o tempo de execução?

↳ depende

3 casos

1 - Melhor caso  $\rightarrow$  menor valor possível para  $T$   
exemplo :  $T(n) = c$

2 - Pior caso  $\rightarrow$  maior valor possível para  $T$   
exemplo :  $T(n) = cn$

3 - Caso médio  $\rightarrow$  valor médio sobre todas as entradas  
+ depende da distribuição das entradas

### Complexidade Assintótica

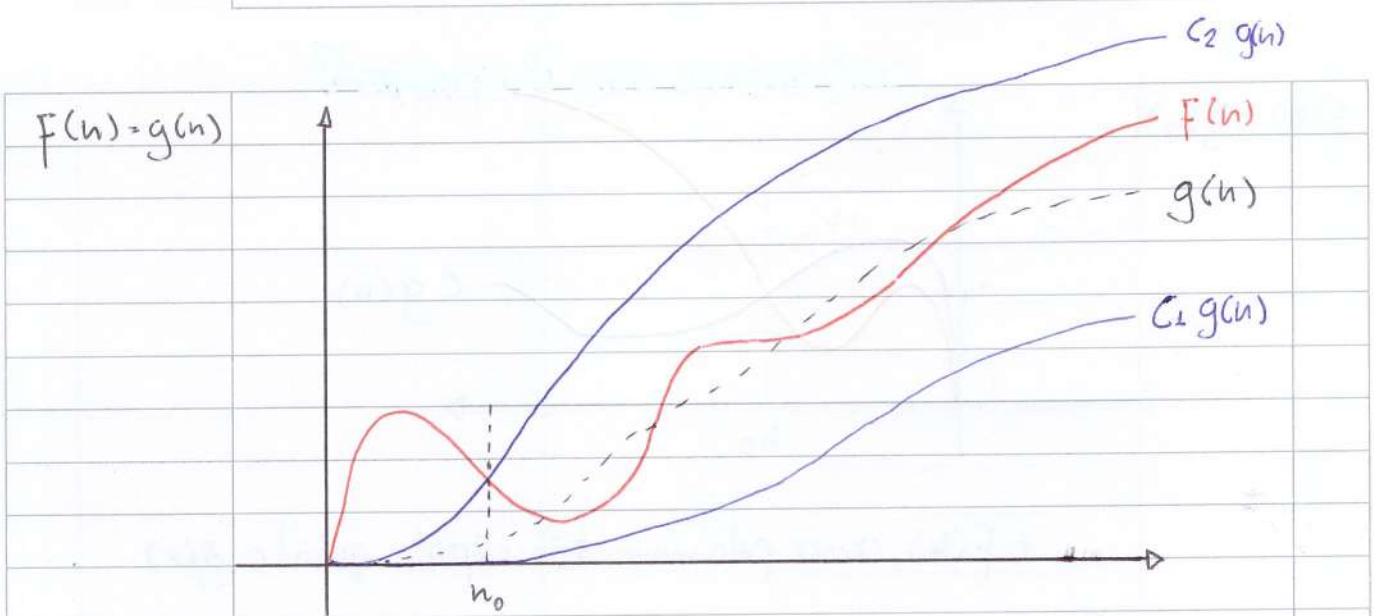
Descrever a taxa de crescimento do tempo de computação à medida que a entrada cresce indefinidamente

$T(n)$	$\alpha = T(Kn) / T(n)$
$Cf(n)$	$\alpha = \cancel{C} \frac{T(Kn)}{T(n)} = \frac{T(Kn)}{T(n)}$ + constantes multiplicativas não importam
$f(n) + g(n)$	$\alpha = \frac{f(Kn) + g(Kn)}{f(n) + g(n)} = \frac{f(Kn)}{f(n) + g(n)} + \frac{g(Kn)}{f(n) + g(n)}$ Suponha $f(n) + g(n) \xrightarrow{n \rightarrow \infty} f(n)$ e.g.: $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0$ $\alpha \approx \frac{f(K)}{f(n)}$ + termos aditivos de menor ordem não importam

Queremos caracterizar a função de custo a menos de constantes e termos de menor ordem

Definição (ordem exata) seja  $g(n) > 0$

$$\Theta(g(n)) = \{f(n) \mid \exists \text{ constantes } c_1, c_2, n_0 > 0, \forall n > n_0, (c_1 g(n) \leq f(n) \leq c_2 g(n))\}$$



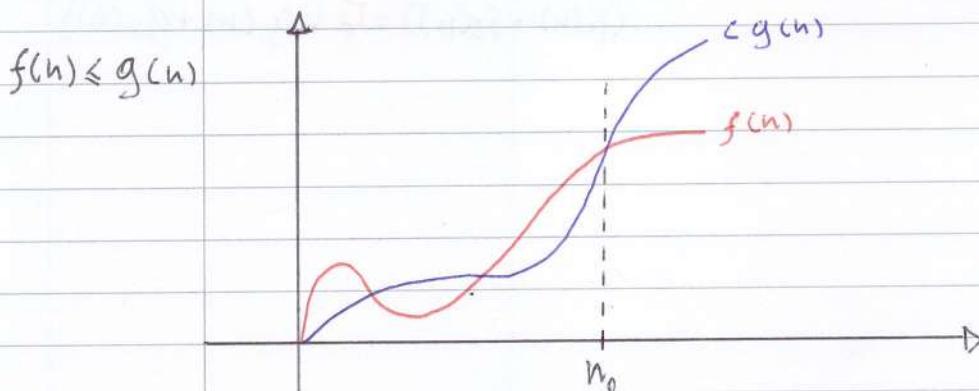
+ correto  $f(n) \in \Theta(g(n))$

usual  $f(n) = \Theta(g(n))$

O crescimento de  $f(n)$  acompanha o crescimento de  $g(n)$

Definição (cota superior) seja  $g(n) > 0$

$$\mathcal{O}(g(n)) = \{f(n) \mid \exists \text{ constantes } c, n_0 > 0 \ (\forall n > n_0 \ (f(n) \leq c g(n)))\}$$

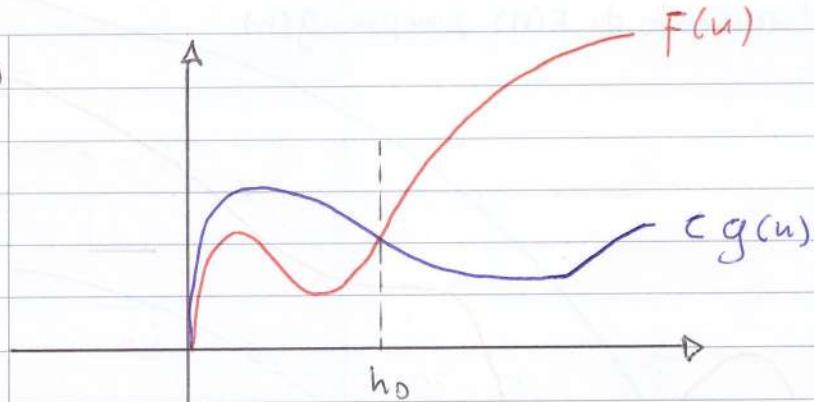


+ O crescimento de  $f(n)$  é mais lento do que o crescimento de  $g(n)$

Definição (cota inferior) seja  $g(n) > 0$

$$\Omega(g(n)) = \{f(n) \mid \exists \text{ constantes } c, n_0 > 0 \ (\forall n > n_0 \ (f(n) \geq c g(n)))\}$$

$$f(n) > g(n)$$



+  $F(n)$  cresce pelo menos tão rápido quanto  $g(n)$

### Propriedades

$$1 - f(n) \geq \Omega(g(n)) \Leftrightarrow g(n) \in \mathcal{O}(f(n))$$

$$2 - f(n) = \Theta(g(n)) \Leftrightarrow f(n) = \Omega(g(n)) \wedge g(n) = \mathcal{O}(f(n))$$

$$3 - f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$$

$$\square = \Omega, \mathcal{O}, \Theta$$

$$4 - \text{se } \square(f(n))$$

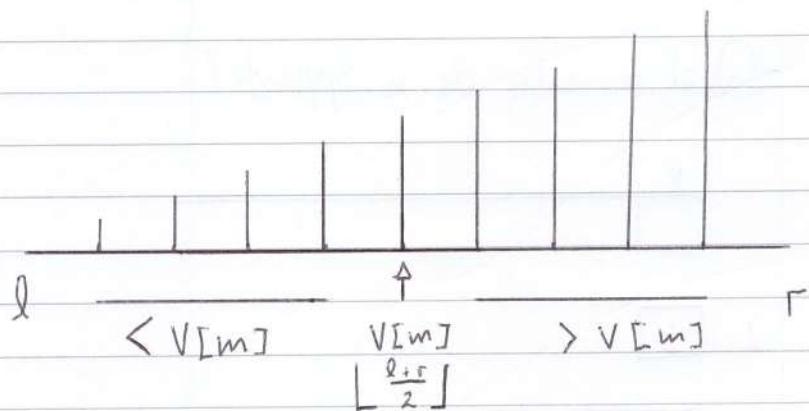
$$5 - f(n) = \square(g(n)) \wedge g(n) = \square(h(n)) \rightarrow f(n) = \square(g(n))$$

$$6 - f(n) = \square(Kg(n)) \rightarrow f(n) = \square(g(n))$$

$$7 - f_1(n) = \square(g_1(n)) \wedge f_2(n) = \square(g_2(n)) \rightarrow$$

$$(f_1(n) + f_2(n)) = \square(g_1(n) + g_2(n))$$

## Busca Binária



Algoritmo binary-search

Entrada:  $V = \{v_0, \dots, v_{n-1}\}$ , ordenado e sem repetição

$x$  - valor procurado

Saída: posição de  $x$  em  $V$ , se existir, ou  $-1$ , caso contrário.

início

$l, r \leftarrow 0, n-1$

repita

$m \leftarrow \lfloor (l+r)/2 \rfloor$

se  $x = V[m]$  então

retorna  $m$

senão -se  $x < V[m]$  então

$r \leftarrow m-1$

senão

$l \leftarrow m+1$

fim -se

enquanto  $l \leq r$

retorna  $-1$

Fim

+ A cada passagem, o  
tamanho do intervalo consi-  
derando reduz-se à metade

$n \rightarrow \frac{n}{2} \rightarrow \dots \rightarrow 2 \rightarrow 1$

$T(n) = \Theta(\log_2 n)$  no pior caso

Exemplo

$V =$	0	1	2	3	4	5	6	7	8	9
	2	4	6	8	10	12	14	16	18	20
$x = 14$	$l$				$m$					$r$
						$l$		$m$		
							$m$	$r$		
								$l/m$		

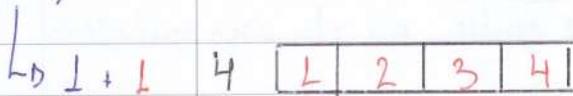
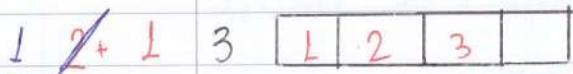
Custo no pior caso

$$T(n) = C * t$$

$t = \text{número de execuções do loop}$

## Arrays Dinâmicos vs Listas

- Qual o custo de  $n$  appends?



Já se irá no final de um array dinâmico tem custo amortizado constante. Com vantagem da localidade

- ADyn rules

$$T(n) = 2n$$

## Algoritmos de Ordenação

### - Mergesort

- Dividir recursivamente o conjunto em 2 metades
- Ordenar individualmente
- Combinar as duas metades ordenadas
- + Dividir para conquistar

4	2	8	6	10		5	1	7	3	9
---	---	---	---	----	--	---	---	---	---	---

4 2 | 8 6 0

4 | 2      8 | 6 0

4    2    8    6 | 0 | 5    1    7 |

2 | 4    8    6    0 |

0 | 6

0 | 6 | 8

0 2 4 6 8

### Algoritmo mergesort

Entrada:  $V = (v_0, \dots, v_{n-1})$  vetor a ordenar

Saída:  $V$  ordenado

início

$A \leftarrow$  novo vetor ( $z_0, \dots, z_{n-1}$ )  
msort ( $V, A, 0, n-1$ )

Fim

## Algoritmo msort

Entrada:  $V = (v_0, \dots, v_{n-1})$  vetor a ordenar

$A = (a_0, \dots, a_{n-1})$  vetor memória auxiliar

$l, r$  limites do intervalo a ordenar

Saída:  $V[l, \dots, r]$  ordenado

Início

se  $l = r$  então

retorna

Fim-se

$m \leftarrow \lfloor (l+r)/2 \rfloor$

msort( $V, A, l, m$ )

msort( $V, A, m+1, r$ )

merge( $V, A, l, r$ )

Fim

## Algoritmo merge

Entrada:  $V = (v_0, \dots, v_{n-1})$

$A = (a_0, \dots, a_{n-1})$

$l, r \mid V[l, \dots, (l+r)/2] \text{ e } V[(l+r)/2+1, \dots, r]$  ordenados

Saída:  $V[l, \dots, r]$  ordenado

Início

$A[l, \dots, r] \leftarrow V[l, \dots, r]$

$m \leftarrow \lfloor (l+r)/2 \rfloor$

$i, j \leftarrow l, m+1$

para  $K \leftarrow l, \dots, r$  faça

se  $i = m+1$  então

$V[K] \leftarrow A[i]$

$j \leftarrow j+1$

senão - se  $j = r+1$  então

$V[K] \leftarrow A[j]$

$i \leftarrow i+1$

senão - se  $A[i] < A[j]$  então

$V[K] \leftarrow A[i]$

$i \leftarrow i+1$

senão

$V[K] \leftarrow A[j]$

$j \leftarrow j+1$

Fim - se

Fim - faça

Fim

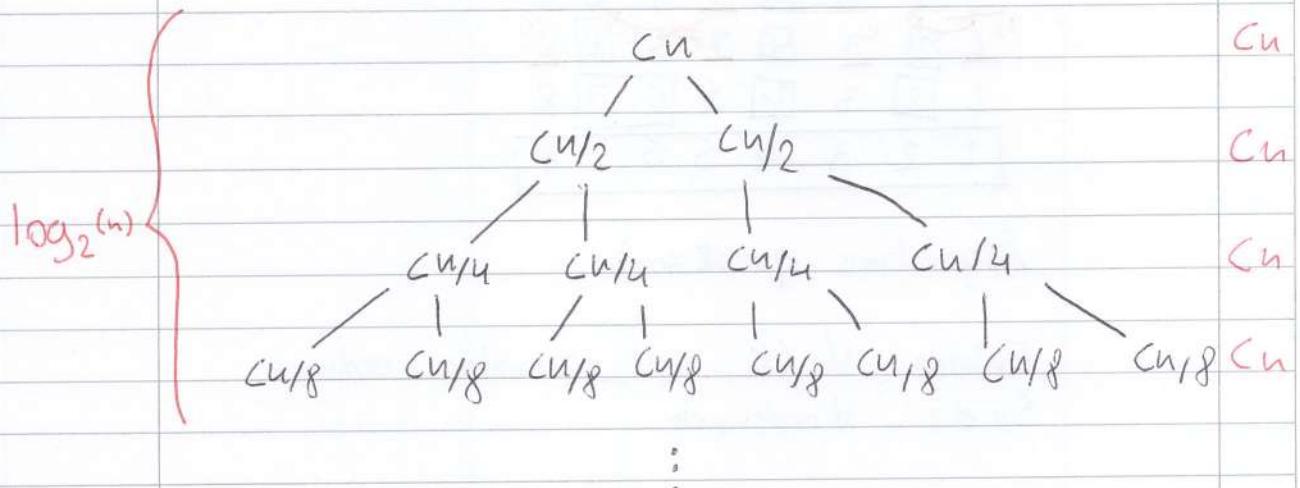
Análise

$$T(n) = T(n/2) + T(n/2) + \Theta(n)$$

Tempo de ordenação  
das 2 metades

Tempo do merge

$$T(n) = \begin{matrix} Cn \\ / \quad \backslash \\ T(n/2) \quad T(n/2) \end{matrix} = \begin{matrix} Cn \\ / \quad \backslash \\ Cn/2 \quad Cn/2 \\ / \quad \backslash \quad / \quad \backslash \\ T(n/4) \quad T(n/4) \quad T(n/4) \quad T(n/4) \end{matrix}$$



CCCCCCCC ... CCCCCCCC

$Cn$

$$T(n) = Cn \log_2(n) = \Theta(n \log_2(n))$$

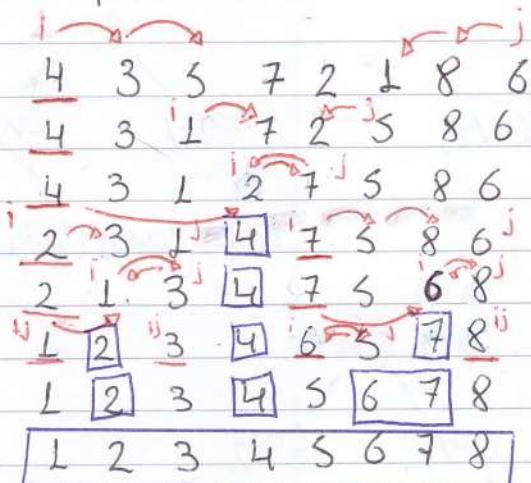
## - Quick sort (Sir C. A. Hoare, 1960)

- Escolher um elemento **pivo**
- Particionar o vetor de entrada

$\leftarrow$  pivo      pivo       $\rightarrow$  pivo

- Ordenar (recursivamente) os trechos à esquerda e à direita do pivo

Exemplo



Algoritmo quick sort

Entrada:  $V = (V_0, \dots, V_{n-1})$  vetor a ordenar

Saída:  $V$  ordenado

início

qsort( $V, 0, n-1$ )

fim

Algoritmo qsort

Entrada:  $V = (V_0, \dots, V_{n-1})$

$l, r$  limites do trecho a ordenar

Saída  $V[l, \dots, r]$  ordenado

```

início
    se  $l > r$  então
        retorna
    Fim-se
     $p \leftarrow \text{partition}(V, l, r)$  // retorna posição final do pivô
     $qsort(V, l, p-1)$ 
     $qsort(V, p+1, r)$ 
Fim

```

### Algoritmo partition

Entrada:  $V = (v_0, \dots, v_{n-1})$   
 $l, r$

Saída: posição final do pivô com o trecho  $V[l \dots r]$  particionado

```

início
     $j \leftarrow \text{pivot}(V, l, r)$  // escolhe o pivô
    permuta  $V[j] \leftrightarrow V[l]$ 
     $i, j \leftarrow l, r$ 
    enquanto  $i < j$  faça
        enquanto  $i < j$  faça
             $i \leftarrow i + 1$ 
        Fim-faça
        enquanto  $V[i] > V[r]$  faça
             $j \leftarrow j - 1$ 
        Fim-faça
        se  $i \leq j$  então
            permuta  $V[i] \leftrightarrow V[j]$ 
        Fim-se
    Fim-faça
    permuta  $V[l] \leftrightarrow V[j]$ 
    retorna  $j$ 
Fim

```

Análise

$$T_{QS}(V) = T_{part}(V) + T_{QS}(\text{esquerda}) + T_{QS}(\text{direita})$$

- Melhor caso

partição equilibrada (duas partes  $\approx$ )

$$T_{QS}(n) \approx 2T_{QS}(n/2) + \Theta(n) = \Theta(n \log_2(n))$$

- Pior caso

partição trivial (pivô é o maior / menor elemento)

$$\begin{aligned} T_{QS}(n) &= cn + T_{QS}(n-1) \\ &= cn + c(n-1) + T_{QS}(n-2) \\ &\vdots \\ &= cn + c(n-1) + (n-2) + \dots + c \\ &= c \frac{n^2}{2} = \Theta(n^2) \end{aligned}$$

- Caso médio supondo permutações equiprováveis

$$\Theta(n \log_2(n))$$

Escolha do pivô

Objetivo

partição equilibrada

Heurística

- elemento do meio

- mediana de três

mediana entre os extremos  $V[0], V[n]$  e o elemento do meio

- escolha aleatória

Randomized quicksort

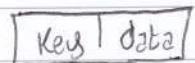
## Tabelas de dispersão (Hash Table)

### Problema

Mantener coleção dinâmica de registros com busca (exata e unívoca) muito eficiente

Supomos que

Cada registro associado a uma chave numérica num conjunto  
 $K = \{0, \dots, s-1\}$



### Ideia Ø

- Usar array  $T = (t[0], \dots, t[s-1])$

- Registro com chave  $K$  na posição  $t[K]$

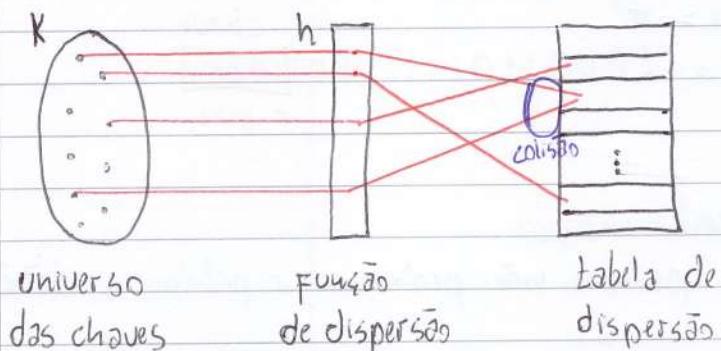
+ acesso, inserção, deleção  $\rightarrow \Theta(1)$

\* Problemas

- chaves únicas

- número de chaves =  $s \gg$  número de registros =  $n$   
↳ ineficiência de memória

### Ideia L



$$h: K \rightarrow \{0, \dots, m-1\}$$
$$K \mapsto h(k)$$

### Problema

$h$  não-injetiva  $\rightarrow$  colisão

### Tarefas

- Definir função de dispersão  $h$
- Tratar colisões

## Definição da função de dispersão

### Requisitos

- 1 - dispersar uniformemente as chaves pela tabela
- 2 - ser robusto quanto a regularidades e viéses nas chaves

### Problemas

- 1 - distribuição de chaves desconhecidas
- 2 - regularidade e viéses frequentes

Não existe regra única  $\rightarrow$  Heurísticas

### Exemplos

#### Heurística da divisão

$$h(K) = K \bmod m$$

- eficácia depende de  $m$

- sensível à regularidade

$$m = 2^r$$

$$K = 1001101 \dots 101 \boxed{10001} \text{ chave}$$

$r$  bits

- regra mágica

$m$  primo não próximo a potências de 2 ou de 10

#### Heurística da multiplicação

Suponha  $m = 2^r$  com palavras de  $w$  bits

$$h(K) = (A * K \bmod 2^w) \gg (w-r)$$

Inteiro ímpar  
 $2^{w-1} < A < 2^w$

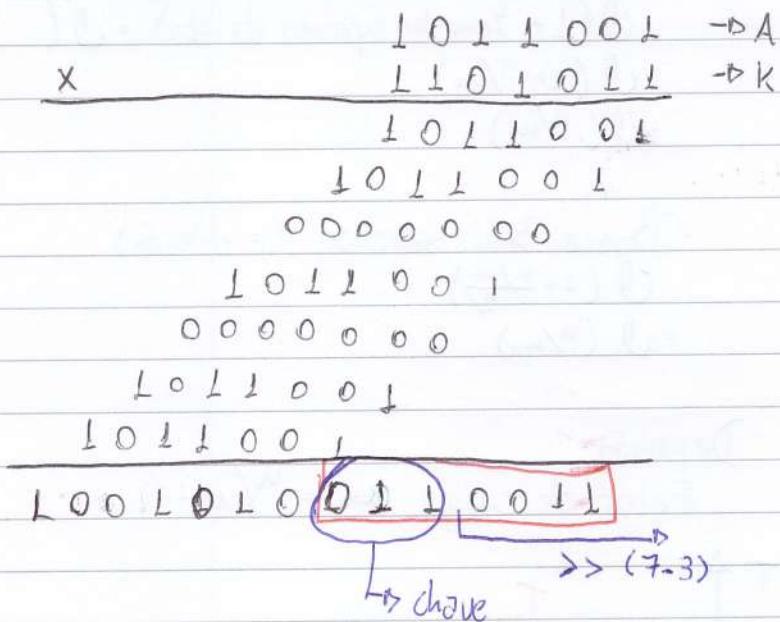
$\Rightarrow$  right shift  
deslocamento de bits à direita  
exemplo:  $1100111 \ll 2$   
 $001100$

$$n \gg k = n/2^k$$

$+ \ll$  (left shift)  
 $n \ll k = 2^k n$

### Exemplo

$$m = 8 = 2^3 \text{ (r=3)}, \quad w = 7 \text{ bits}$$

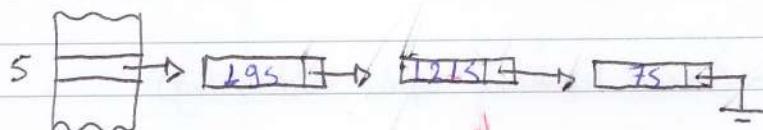


## Resolução de colisões

## 1 - Open hashing

Várias chaves na mesma posição

e.g.: lista en adenda



## Análise

### - Prior case

Todas as chaves na mesma posição  
acesso, inserção, deleção = O(n)

- Melhor caso  
sem colisões  
acesso, inserção, deleção =  $\mathcal{O}(1)$

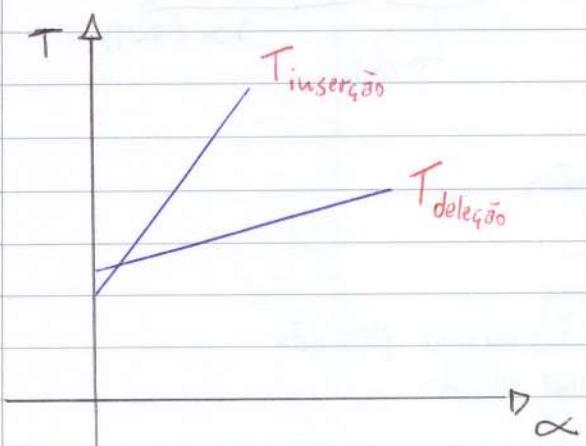
- Caso médio  
sob hipótese da dispersão uniforme simples  
 $P(h(K)=i) = \frac{1}{m}$ ,  $\forall i$  independente das outras chaves

- Busca mal-sucedida (= inserção)  
 $\mathcal{O}(1 + \text{tamanho esperado da lista})$   
 $= \mathcal{O}(1 + \frac{w}{m})$   
 $= \mathcal{O}(\frac{w}{m})$

- Busca bem-sucedida (= deleção)  
 $\mathcal{O}(1 + \frac{w}{2})$   
 $= \mathcal{O}(\frac{w}{m})$

Definição

Fator de carga  $\alpha = \frac{w}{m}$



## 2- Closed hashing (open addressing)

- Máximo de 1 chave por posição
- Tentar posição inicial
- Se ocupada, sondar sistematicamente outras posições até encontrar a vaga

chave ↴

↓  
Entablima

posição  
↓

$$h: K \times \{0, \dots, m-1\} \rightarrow \{0, \dots, m-1\}$$

$$(k, i) \mapsto (h_0(k) + p(k, i)) \bmod m$$

posição  
inicial ↗

↑ valor do  $i$ -ésimo "salto"

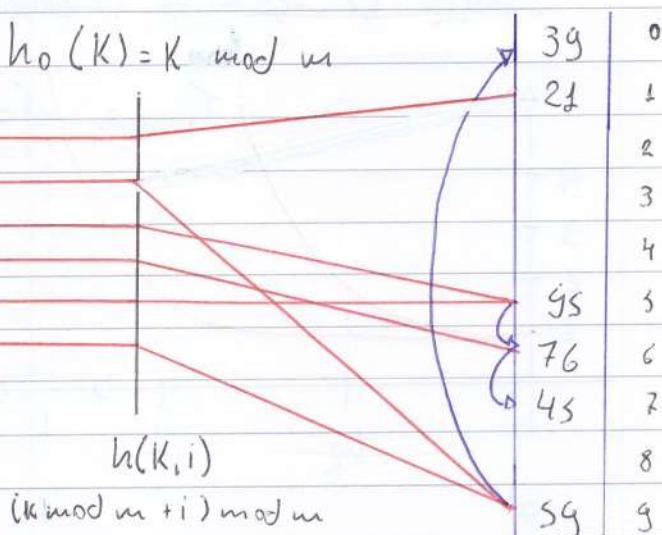
- Sondagem linear (linear probing)

$$p(k, i) = i \rightarrow h(k, i) = (h_0(k) + i) \bmod m$$

Exemplo

$$m = 10, h_0(k) = k \bmod m$$

21  
59  
95  
76  
45  
39



$$+ P(h=3) = 4/10$$

$$P(h=8) = P(h_0 \in \{5, 6, 7, 8\}) = 4/10$$

- Primary clustering

posições vizinhas a regiões densas têm maior chance de serem escolhidas

Alternativas:

Sondagem quadrática

Sondagem aleatória

+ Secondary clustering

se  $h_0$  gera um clustering, então  $h$  também gera

Problema: A função do salto não considera  $K$

Solução

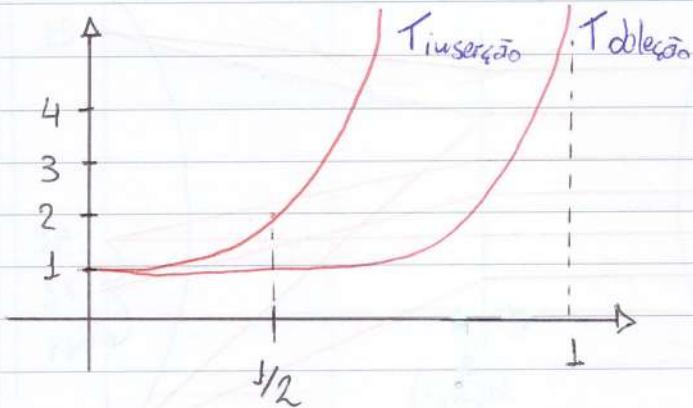
Double hashing

$$h(K, i) = [h_0(K) + i \cdot h_1(K)] \bmod m$$

Usualmente  $m = 2^r$ ,  $h_1(K)$  é ímpar

Análise

- Busca mal-sucedida ( $=$  inserção) =  $\mathcal{O}(\frac{1}{1-\alpha})$
- Busca bem-sucedida ( $=$  deleção) =  $\mathcal{O}(\frac{1}{\alpha} \ln \frac{1}{1-\alpha})$

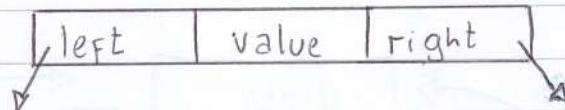


# Árvores binárias

"Trees are Computer Scientists' best friend"

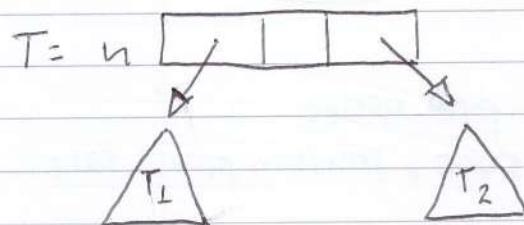
- Donald E Knuth

Estrutura de dados dinâmica não-linear com nós

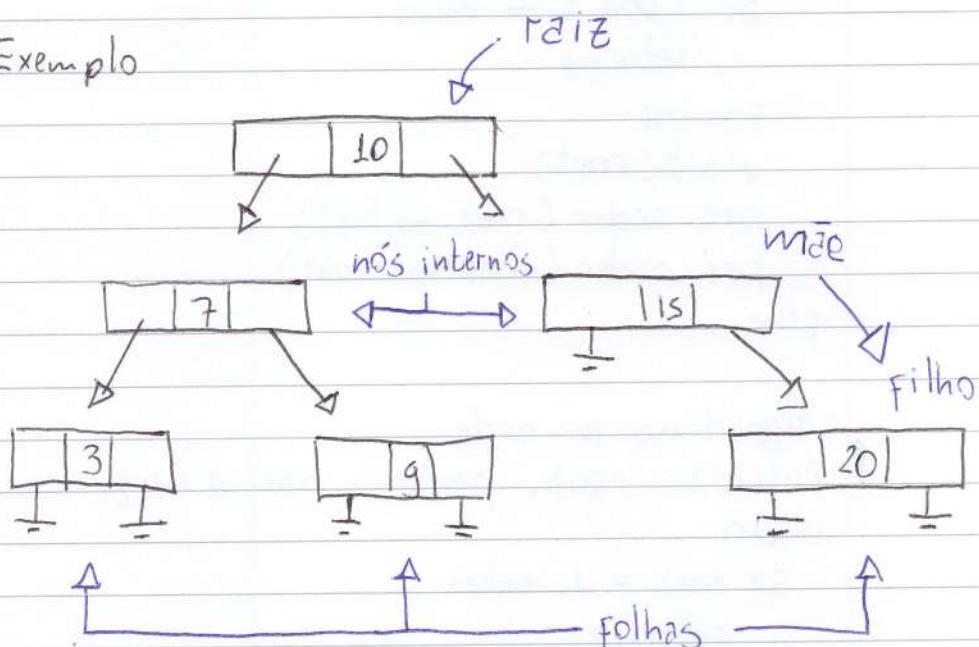


Definição

- 1 - Um conjunto vazio de nós é uma binary tree
- 2 - Se  $n$  é um nó e  $T_1, T_2$  são binary trees disjuntas e  $n \notin T_1 \wedge n \notin T_2$ , então  $T$  é uma árvore binária



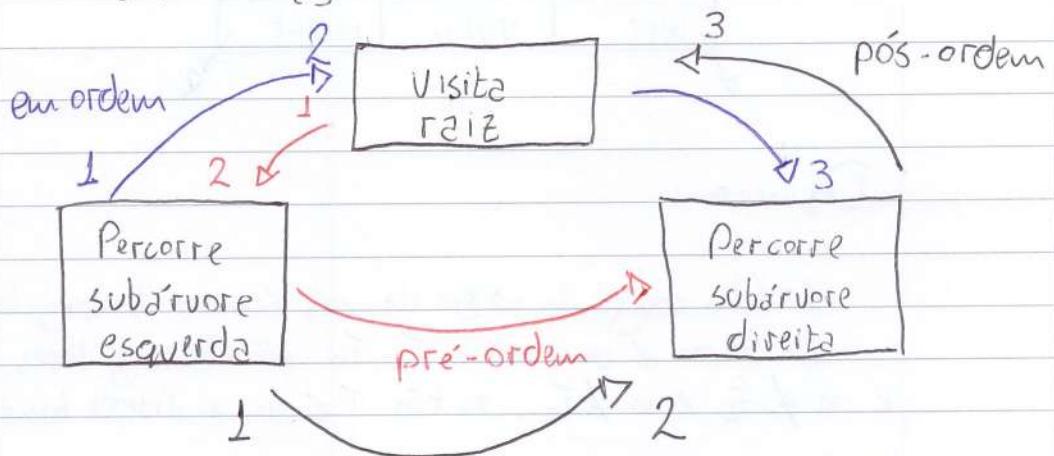
Exemplo



## Percorso de uma binary tree

Objetivo: enumerar (visitar) os nós numa ordem sistemática

3 alternativas



Algoritmo pre-order

Entrada: root, ponteiro para a raiz  
início

se root =  $\perp$  então  
retorna

Fim-se  
visit(root)  
pre-order (root  $\rightarrow$  left)  
pre-order (root  $\rightarrow$  right)

fim

Algoritmo in-order

Entrada: root, ponteiro para a raiz  
início

se root =  $\perp$  então  
retorna

Fim-se  
in-order (root  $\rightarrow$  left)  
visit(root)  
in-order (root  $\rightarrow$  right)

fim

## Algoritmo post-order

Entrada: root, ponteiro para a raiz

início

se root = 1 então

retorna

Fim-se

post-order(root->left)

post-order(root->right)

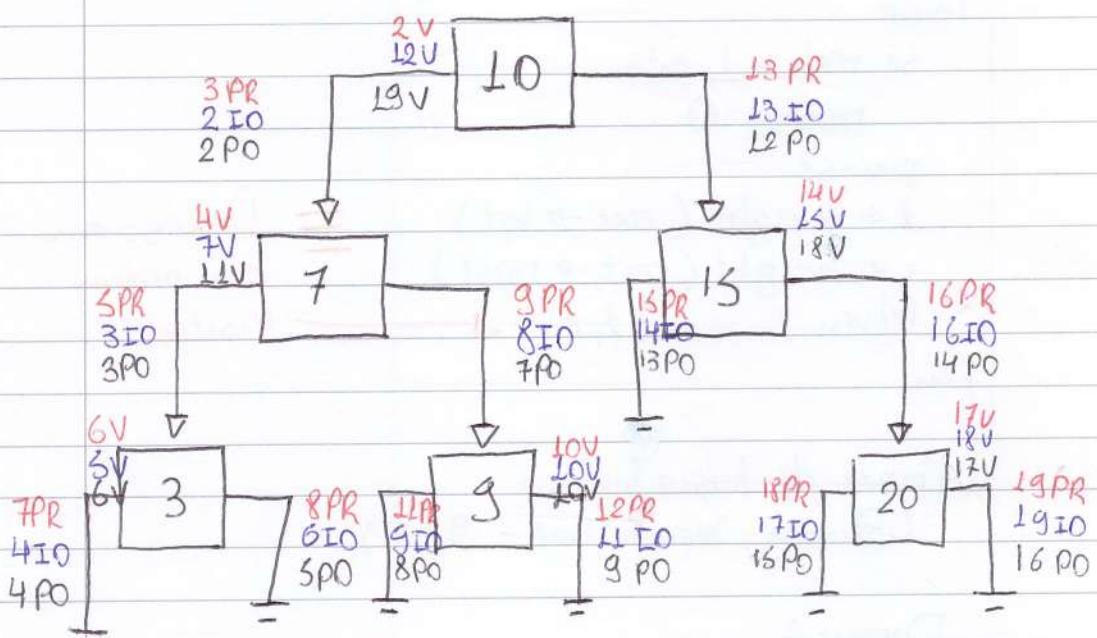
visit(root)

fim

+ A função visit depende da aplicação

Exemplo

1 PR 1 IO 1 PO

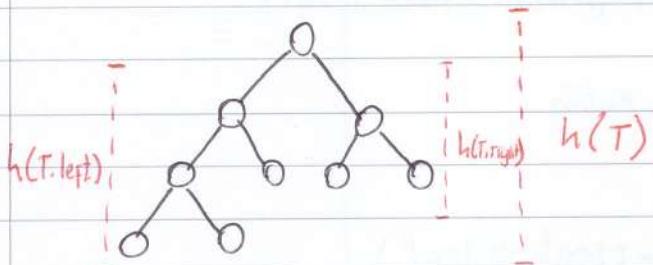


PR: 10, 7, 3, 9, 15, 20

IO: 3, 7, 9, 10, 15, 20

PO: 3, 9, 7, 20, 15, 10

## Altura de uma árvore binária



$$h(N) = \begin{cases} 0, & \text{se } N = \text{None} \\ 1 + \max\{h(n.\text{left}), h(n.\text{right})\}, & \text{caso contrário} \end{cases}$$

Algoritmo height

Entrada: root

Saída: a altura da árvore enraizada em root

inicio

se root = None então

retorna 0

fim-se

$l \leftarrow \text{height}(\text{root} \rightarrow \text{left})$

$r \leftarrow \text{height}(\text{root} \rightarrow \text{right})$

return  $l + \max(l, r)$

= Percurso em  
pós-ordem  
visited

Fim

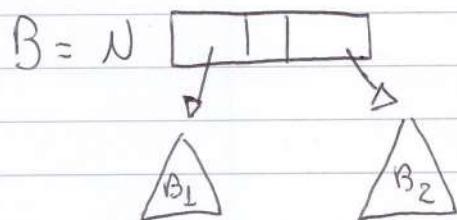
Árvore de busca binária

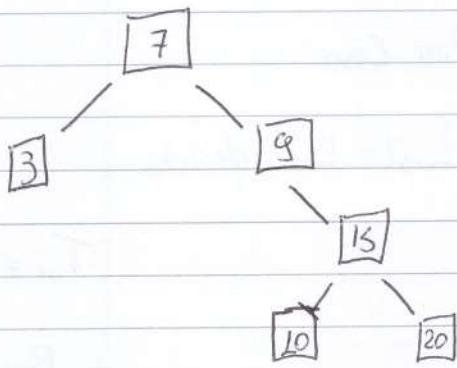
(Binary Search Tree - BST)

Definição

1 - Uma BT vazia é uma BST

2 - Se  $N$  é um nó  $B_1, B_2$  são BSTs disjuntas e  $N \notin B_1, B_2$  e  $N.\text{value} > N_1.\text{value} \wedge N_1 \in B_1$  e  $N.\text{value} \leq N_2.\text{value} \wedge N_2 \in B_2$ .





Pre-ordem 7, 3, 9, 15, 10, 20

Em-ordem 3, 7, 9, 10, 15, 20

Pós-ordem 3, 10, 20, 15, 9, 7

### Teoria

Um percurso em ordem de uma BST visita os nós em ordem crescente de valor

### Algoritmo bst-search

Entrada: root ponteiro para raiz

v valor procurado

Saída: ponteiro para o nó com valor v, se existir, ou L

início

se root = L então

retorna L

senão-se v = root  $\rightarrow$  val então

retorna root

senão-se v < root  $\rightarrow$  val então

retorna bst-search (root  $\rightarrow$  left, v)

senão

retorna bst-search (root  $\rightarrow$  right, v)

Fim-se

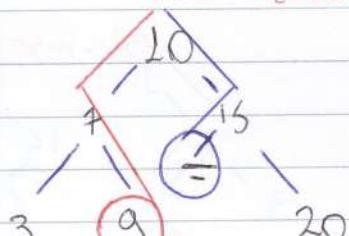
fim

$$\text{bst-search}(\&10, 13) = \text{L}$$

$$\text{bst-search}(\&10, 9) = \&9$$

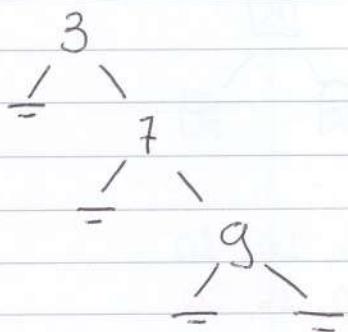
Exemplo

$$v = 9$$



Análise Pior Caso

BST  $\equiv$  Lista Encadeada



$$T_n = \mathcal{O}(n)$$

+ Busca Linear

$\neq$

Busca Binária

$\hookrightarrow$  Solução

Arvores Balanceadas

Algoritmo bst-insert

Entrada : root

v valor a inserção

Saida : ponteiro para "nova" raiz após inserção

início

se root =  $\perp$

n  $\leftarrow$  novo nó

n.value  $\leftarrow$  v

n.left  $\leftarrow$   $\perp$

n.right  $\leftarrow$   $\perp$

retorna &n

senão - se  $v < root \rightarrow$  value então

root  $\rightarrow$  left  $\leftarrow$  bst-insert (root  $\rightarrow$  left, v)

retorna root

senão

root  $\rightarrow$  right  $\leftarrow$  bst-insert (root  $\rightarrow$  right, v)

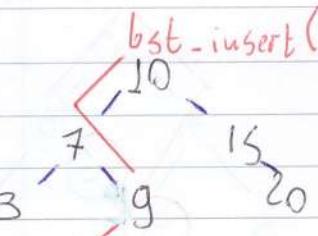
retorna root

Fim-SP

Fim

Exemplo

v = 8



bst-insert (&10, 8)

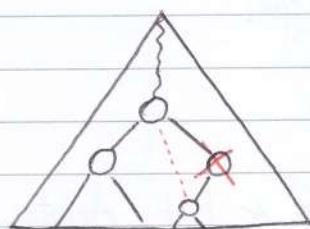
+ inserção sempre  
na folha

Remoção

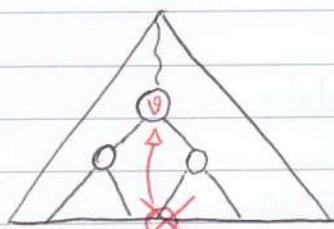
3 casos

(I) Nó a remover é uma folha  
remove e pronto

(II) Nó a remover tem apenas 1 filho

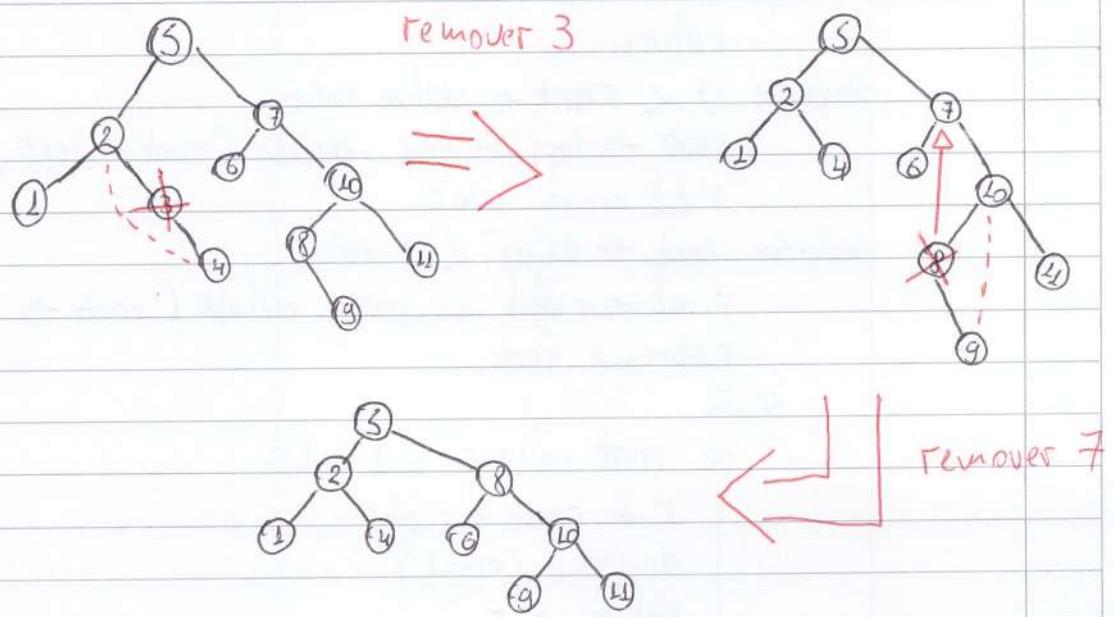


(III) Nó a remover tem 2 filhos



↳ elemento mínimo da subárvore à direita

Exemplo



### Algoritmo bst-delete-min

Entrada:  $\text{root} \neq \perp$  ponteiro para raiz da bst

Saida: ponteiro para "nova" raiz após remoção  
valor minimo removido

início

se  $\text{root} \rightarrow \text{left} = \perp$  então

$v \leftarrow \text{root} \rightarrow \text{value}$

$r \leftarrow \text{root} \rightarrow \text{right}$

$\text{dispose}(\text{root})$

$\text{return}(r, v)$

Senão

$\text{root} \rightarrow \text{left}, v \leftarrow \text{bst-delete-min}(\text{root} \rightarrow \text{left})$

$\text{return}(\text{root}, v)$

Fim-se

fim

### Algoritmo bst-delete

Entrada:  $\text{root}$

$v$  valor a remover

Saida: ponteiro para "nova" raiz após remoção

início

se  $\text{root} = \perp$  então

$\text{return } \perp$

senão se  $v < \text{root} \rightarrow \text{value}$  então

$\text{root} \rightarrow \text{left} \leftarrow \text{bst-delete}(\text{root} \rightarrow \text{left}, v)$

$\text{return} \text{root}$

senão se  $\text{root} \rightarrow \text{value} < v$  então

$\text{root} \rightarrow \text{right} \leftarrow \text{bst-delete}(\text{root} \rightarrow \text{right}, v)$

$\text{return} \text{root}$

Senão

se  $\text{root} \rightarrow \text{left} = \perp$  então

$r \leftarrow \text{root} \rightarrow \text{right}$

$\text{dispose}(\text{root})$

$\text{return} r$

sendo-se  $\text{root} \rightarrow \text{right} = \text{L}$  então

$\text{r} \leftarrow \text{root} \rightarrow \text{left}$

$\text{dispose}(\text{root})$

$\text{returna r}$

Senão

$\text{root} \rightarrow \text{right}, \text{root} \rightarrow \text{value} \leftarrow \text{bst\_delete\_min}(\text{root} \rightarrow \text{right})$

$\text{returna root}$

Fim-se

Fim-se

Fim

## Árvores Balanceadas (AVL)

Objetivo

BST com busca  $\mathcal{O}(\log_2 n)$

Definição árvore AVL

BST tal que para cada nó  $N$ , o valor absoluto da diferença entre as alturas das subárvore esquerda e direita (nesta ordem) é  $\leq 1$

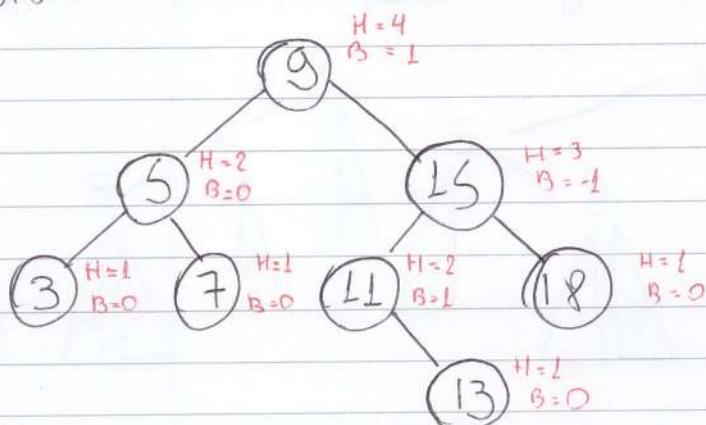
$$B(N) \stackrel{\text{def}}{=} H(R) - H(L)$$

$\Rightarrow$  fator de平衡amento

AVL

$$-1 \leq B(N) \leq 1, \forall N$$

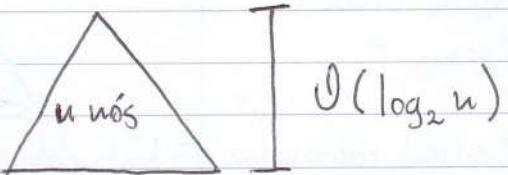
Exemplo



É AVL

Teorema

Uma árvore AVL com  $n$  nós tem altura  $\mathcal{O}(\log_2 n)$



$$\mathcal{O}(\log_2 n)$$

Corolário

Uma busca numa árvore AVL requer tempo  $\mathcal{O}(\log_2 n)$  no pior caso

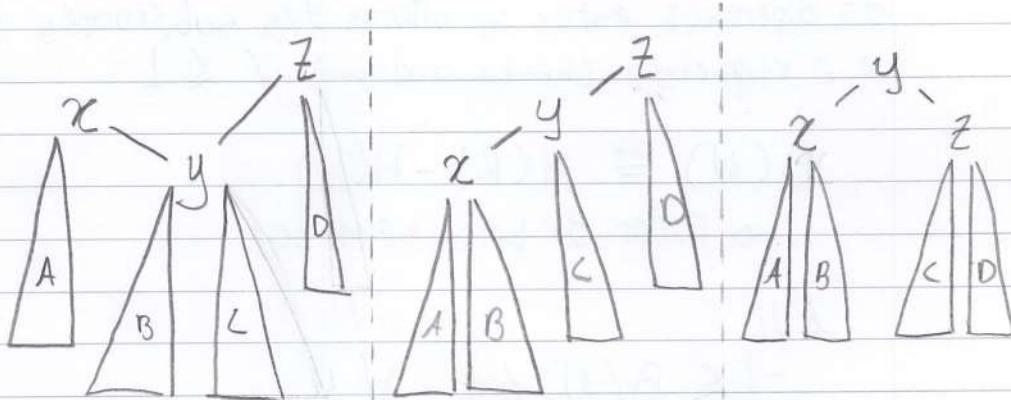
Inserção em AVL

Problema

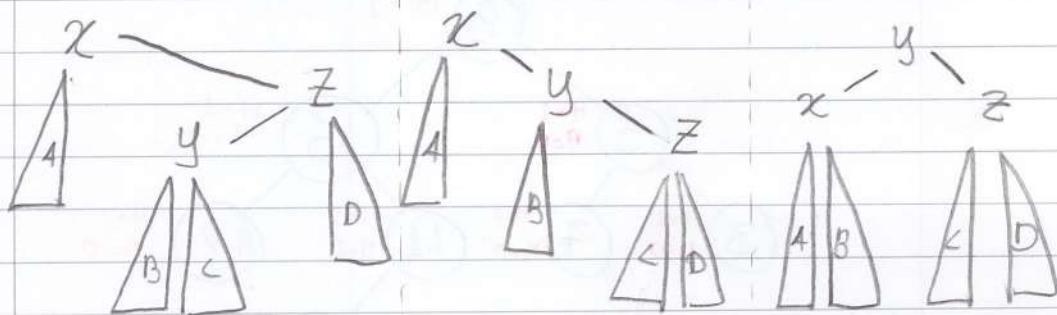
Inserções podem causar desbalanceamento

Solução

Restaurar balanceamento através de rotações



left-right  $\xrightarrow{\text{left}}$  left-left  $\xrightarrow{\text{right}}$  balanced



right-left  $\xrightarrow{\text{right}}$  right-right  $\xrightarrow{\text{left}}$  balanced

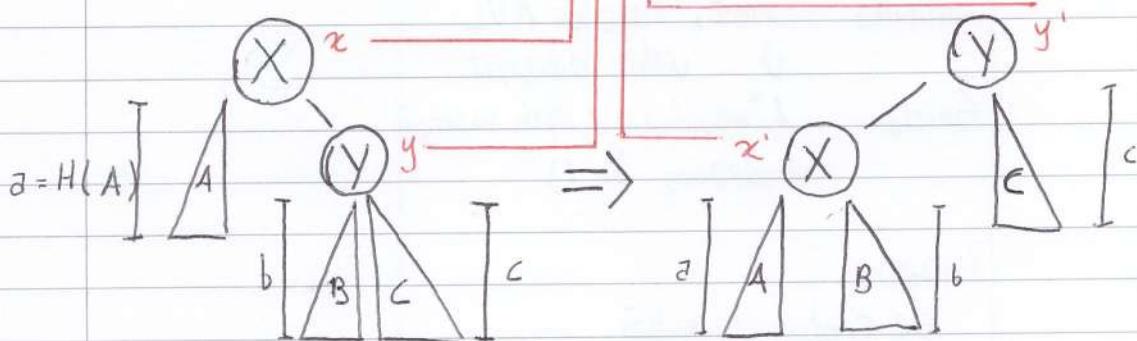
$$x = (L + \max(b, c)) - z$$

$$y = c - b$$

$$x' = b - z$$

$$y' = c - (L + \max(z, b))$$

Rotação à Esquerda



$$\cdot x' - x = b - L - \max(b, c)$$

$$x' = x - L - (\max(b, c) - b)$$

$$x' = x - L - \begin{cases} y = c - b, & \text{se } \max(b, c) = c \equiv y \geq 0 \\ \emptyset, & \text{se } \max(b, c) = b \equiv y < 0 \end{cases}$$

$$\cdot y' - y = b - (L + \max(z, b))$$

$$y' = y - L + (b - \max(z, b))$$

$$y' = y - L + \begin{cases} x' = b - z, & \text{se } \max(z, b) = z \equiv x' \leq 0 \\ \emptyset, & \text{se } \max(z, b) = b \equiv x' \geq 0 \end{cases}$$

+ Se mantivermos as informações dos fatores de平衡amento  
temos atualização em tempo  $O(1)$

### Algoritmo avl-rotate-kft

Entrada : root, ponteiro para raiz da rotação

Saída : a nova raiz após a rotação

início

$R \leftarrow \text{root.right}$

$RL \leftarrow \text{root} \rightarrow \text{right} \rightarrow \text{left}$

$R \rightarrow \text{left} \leftarrow \text{root}$

$\text{root} \rightarrow \text{right} \leftarrow RL$

se  $R \rightarrow \text{bf} \geq 0$  então

$\text{root} \rightarrow \text{bf} \leftarrow \text{root} \rightarrow \text{bf} - L - R \rightarrow \text{bf}$

sendo

$\text{root} \rightarrow \text{bf} + \text{root} \rightarrow \text{bf} - 1$

fim-se

se  $\text{root} \rightarrow \text{bf} < 0$  então

$R \rightarrow \text{bf} \leftarrow R \rightarrow \text{bf} - L + \text{root} \rightarrow \text{bf}$

senão

$R \rightarrow \text{bf} \leftarrow R \rightarrow \text{bf} + 1$

fim-se

return R

fim

## Algoritmo AVL-insert

Entrada : root raiz da AVL  
v valor a inserir

Saida : 1 "nova" raiz após inserção  
booleano

início

se root = 1 então

u ← novo nó

u.left = 1

u.right = 1

u.val = v

u.bf = 0

retorna (&u, 1)

senão-se root → value = v então *// não inserir repetidos*

retorna (root, 0)

senão-se v < root → value então

root → left, hc ← AVL-insert (root → left, v)

root → bf ← root → bf - hc

senão

root → right, hc ← AVL-insert (root → right, v)

root → bf ← root → bf + hc

Fim-se

se !hc então

retorna (root, 0)

senão-se root → bf = 0 então

retorna (root, 0)

senão-se root → bf = ± 1 então

retorna (root, 1)

senão-se root → bf = -2 então

se root → left → bf = 1 então

root → left ← AVL-rotate-left (root → left)

fim-se

retorna (AVL-rotate-right (root), 0)

senão-se root → bf = 2 então

se root → right → bf = -1 então

root → right ← AVL-rotate-right (root → right)

retorna (avl-rotate-left(root), 0)

fim-se

fim

# Heaps Binários

## Objetivo

Armazenar coleção de itens a serem processados por ordem de prioridades

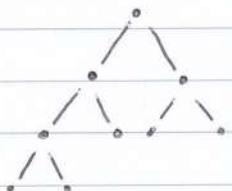
## Exemplo

Tipo abstrato de dados, fila com prioridades (priority queue)

## Definição

Seja  $(A, \succeq)$  conjunto ordenado. Um heap binário sobre  $(A, \succeq)$  é uma árvore binária completa, tal que para cada nó não-folha  $N$ , temos  $N.\text{value} \succeq N' .\text{value}$ ,  $\forall N'$  filho de  $N$

+ árvore binária completa



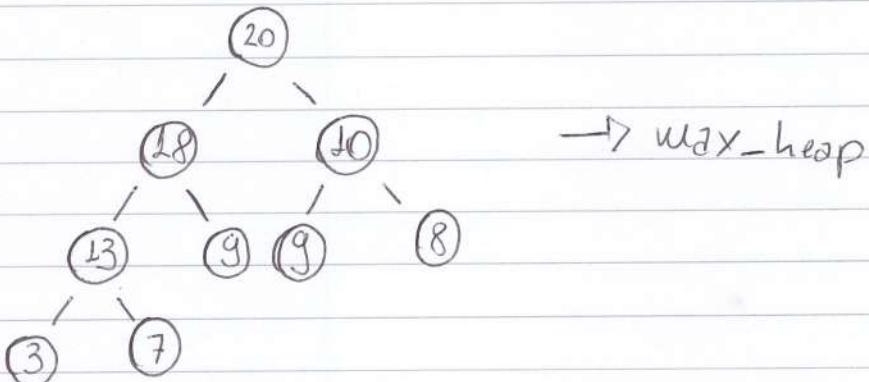
Preenchida um nível por vez  
da esquerda para direita

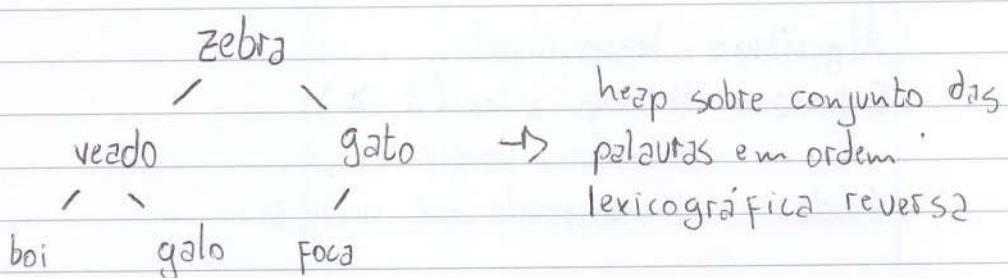
## - Nomenclatura

$\succeq \equiv \geq \rightarrow \text{max\_heap}$

$\succeq \equiv \leq \rightarrow \text{min\_heap}$

## Exemplos





Representação

Heap de altura  $h$



Array com  $2^h - 1$  elementos  
sendo:

- Raiz na posição  $\emptyset$
- Filhos do nó da posição  $i$  em
 
$$\begin{cases} \text{left}(i) = 2i + 1 \\ \text{right}(i) = 2i + 2 \end{cases}$$

Exemplo

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
20	18	10	13	9	9	8	3	7							

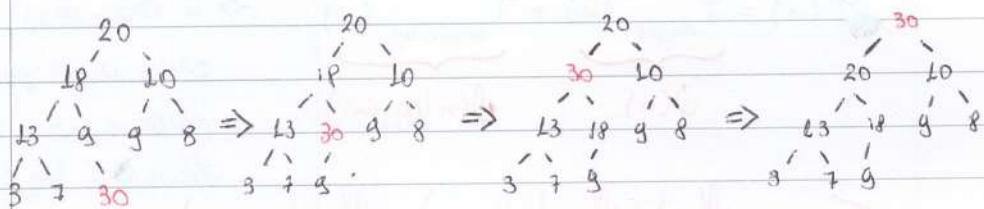
heap-size( $h$ ) = número de elementos do heap

size( $h$ ) = capacidade do array

Inserção

- Inserir elemento na próxima posição disponível
- Fazê-lo "subir" até o nível adequado

Exemplo



### Algoritmo heap-insert

Entrada : H heap sobre  $(A, \leq)$

✓ valor  $v$  a inserir

Saída : H modificado após inserção

início

se heap-size(H) = size(H) então

double(H)

Fim-se

H[heap-size(H)]  $\leftarrow v$

bubble-up(H)

Fim

### Algoritmo bubble-up

Entrada H heap sobre  $(A, \leq)$  exceto pelo último elemento

Saída H "aheapiado"

início

i  $\leftarrow$  heap-size(H) - 1

enquanto  $i > 0$  e  $H[i] \leq H[\lceil i/2 \rceil - 1]$  faça

permute  $H[i] \leftrightarrow H[\lceil i/2 \rceil - 1]$

$i \leftarrow \lceil i/2 \rceil - 1$

Fim-se

Fim

### Análise

#### Teoria

heap-insert de  $n$  elementos requer tempo  $\mathcal{O}(n \log_2 n)$  no pior caso

#### Prova

$$T(n) = \underbrace{T_{\text{double}}(n)}_{\mathcal{O}(n)} + \underbrace{T_{\text{bubble-up}}(n)}_{\mathcal{O}(n \log_2(n))} \rightarrow n \text{ chamadas, sendo que}$$

cada uma precisa, no

pior caso, subir

altura  $\leq \log_2(n)$

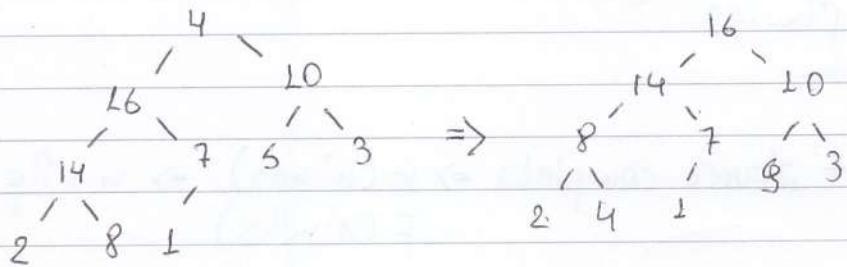
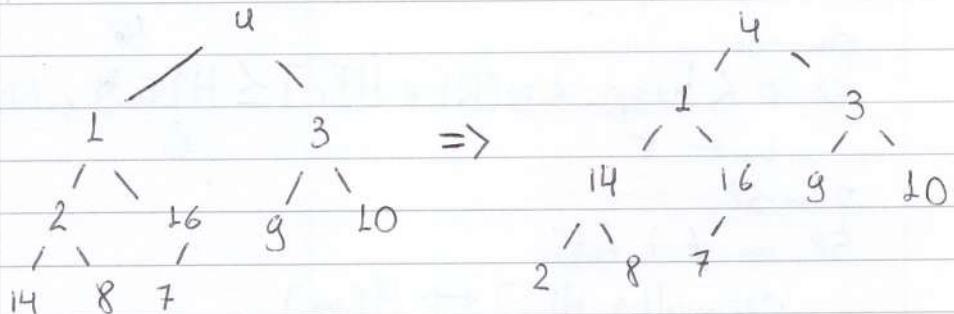
$$T(n) = \mathcal{O}(n) + \mathcal{O}(n \log_2(n)) = \mathcal{O}(n \log_2(n))$$

## Construção offline

- JÁ SÃO CONHECIDOS OS  $n$  VALORES
- + SE  $H_1$  E  $H_2$  SÃO HEAPS, ENTÃO PODE SER TRANSFORMADO EM HEAP FAZENDO-SE X "DESCER" ATÉ O NÍVEL ADEQUADO

Exemplo

$$H = (4, 1, 3, 2, 16, 9, 10, 14, 8, 7)$$



Algoritmo build\_heap

Entrada : H array

Saída : H como heap sobre ( $\leq$ ,  $\geq$ )

início

para  $i \leftarrow \lfloor \text{heap-size}(H)/2 \rfloor - 1$  até 0 fazer  
    heapify( $H, i$ )

final-faça

final

Algoritmo heapify

Entrada:  $H$  array

$i$  tal que subárvore enraizada em  $\text{left}(i)$  e  $\text{right}(i)$   
s o heap

Sa da:  $H$  tal que a sub rvore enraizada em  $i$   heap sobre  $(\leq, \leq)$

in cio

$m, l, r \leftarrow i, 2i+1, 2i+2$

se  $l < \text{heap-size}(H)$  e  $H[l] \leq H[m]$  ent o

$m \leftarrow l$

fim-se

se  $r < \text{heap-size}(H)$  e  $H[r] \leq H[m]$  ent o

$m \leftarrow r$

fim-se

se  $m \neq i$  ent o

permute  $H[i] \leftrightarrow H[m]$

heapify( $H, m$ )

fim-se

fim

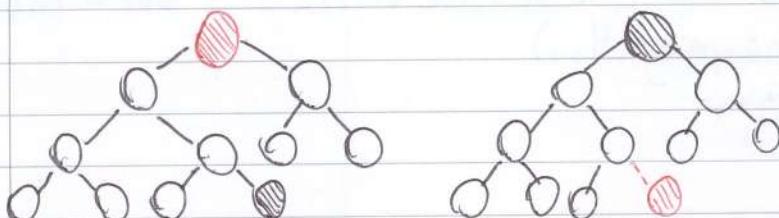
+  rvore completa  $\Rightarrow n(\text{n}^{\circ} \text{nos}) \Rightarrow n = 2F - 1$   
 $F(n^{\circ} \text{folhas})$

An lise

T eoria

A constru  o bottom-up a partir de um array com  $n$  elementos requer tempo  $O(n)$  no pior caso

Extrac o do elemento dominante (raiz)



Algoritmo heap-extract

Entrada:  $H$  heap não-vazia sobre  $\leq$

Saída: O valor do elemento removido

início

$r \leftarrow H[0]$

permute  $H[0] \leftrightarrow H[\text{heap-size}(H) - 1]$

$\text{heap-size}(H) \leftarrow \text{heap-size}(H) - 1$

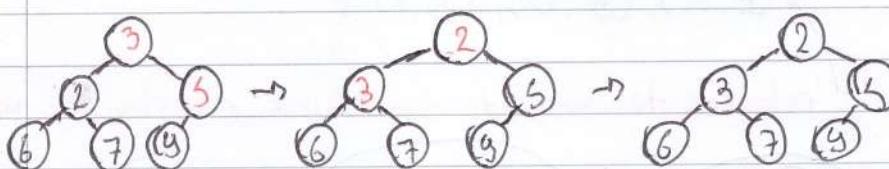
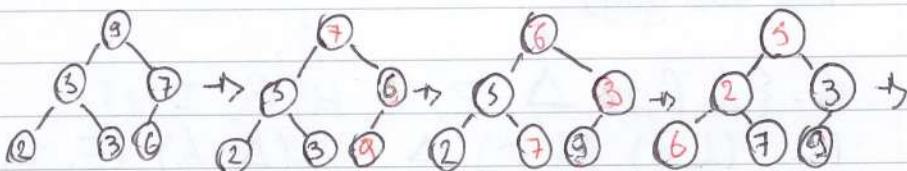
heapsify  $(H, 0)$

return  $r$

fim

Exemplo

$$H = \{9, 5, 7, 2, 3, 6\}$$



$$H = \{2, 3, 5, 6, 7, 4\}$$

Heapsort

Extrações sucessivas de uma max-heap remove os elementos em ordem decrescente

Algoritmo heapsort

Entrada:  $H = \{h_0, \dots, h_{n-1}\}$  array

Saída:  $H$  ordenado

início

build-max-heap( $H$ ) //  $O(n)$

enquanto  $\text{heap-size}(H) > 0$  faça

    heap-extract( $H$ )

fim-faça

fim.  $N_1, N_2, N_3, T(n) = O(n \log n)$

# Conjuntos Disjuntos (Union-Find)

## Objetivo

representar relações de equivalência

## Teorema

uma relação de equivalência define uma partição do conjunto em classes de equivalência e viceversa

$$\forall a, b \in \mathbb{A}, a \sim b \Leftrightarrow [a] = [b]$$

$[x]$  = classe de equivalência de  $x$

$$+ [a] \stackrel{\text{def}}{=} \{ b \in \mathbb{A} \mid b \sim a \}$$

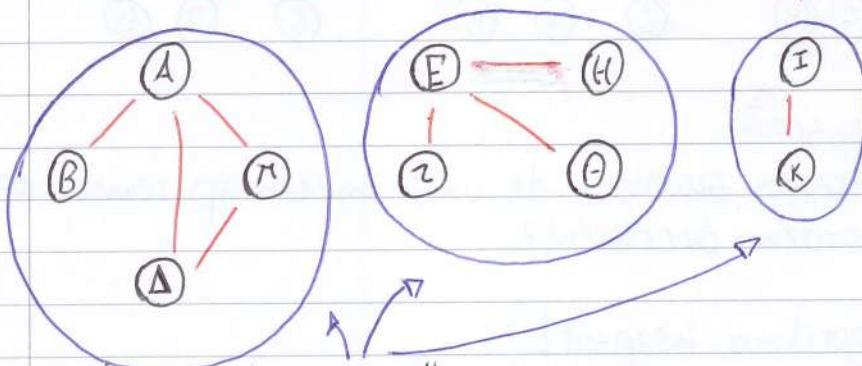
## Exemplo

Rede social grega

$$\mathbb{A} = \{A, B, \Gamma, \Delta, E, \beth, H, \Theta, I, K\}$$

$$F = \{(A, B), (A, \Gamma), (\Delta, \Gamma), (\Delta, A), (E, \beth), \\ (E, H), (E, \Theta), (I, K)\}$$

relações de amizade simétricas, reflexivas, não transitivas



Seja a relação  $C =$  "no círculo de amizade de"  
 $\equiv$  fecho transitivo de  $F$

$$C = \{(A, B, \Gamma, \Delta), (E, H, \beth, \Theta), (I, K)\}$$

Para determinar  $C$

- 1 - Para cada  $a \in \mathbb{A}$ , cria a classe de equivalência unitária  
 $[a] \leftarrow \{a\}$

2- Para cada par  $(a, b) \in F$ , atualiza  
 $[a] = [b] \cup [a] \cup [b]$

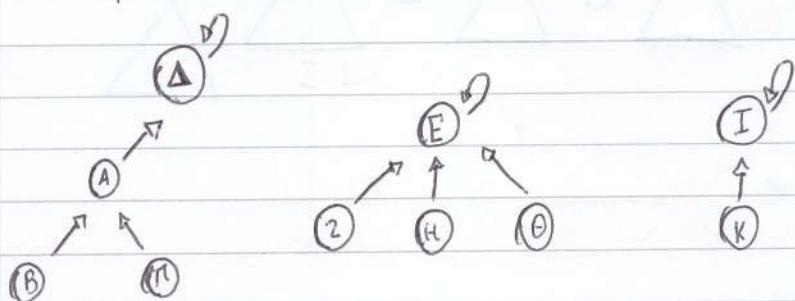
Precisamos de um tipo abstrato de dados com operações

- make-set( $a$ ) cria  $[a] = \{a\}$
- find( $a$ ) determina  $[a]$
- union( $a, b$ ) computa  $[a] \cup [b]$

### Implementação

- Conjuntos disjuntos representados por árvores enraizadas (não necessariamente binárias)
- Cada nó (elemento) aponta para sua raiz
- A raiz é a representante da classe

### Exemplo



+ Representação não é única

### Operações

- make-set( $a$ ) = cria árvore unitária (nó)  
 $\hookrightarrow \Theta(1)$

- find( $a$ ) = encontra raiz de  $[a]$   
 $\hookrightarrow \Theta(\text{altura de } [a])$

- union( $a, b$ ) = encontra as raízes de  $[a], [b]$  e liga uma delas à outra  
 $\hookrightarrow \Theta(\max \{\text{altura } [a], \text{altura } [b]\})$

+ manter alturas o menor possível

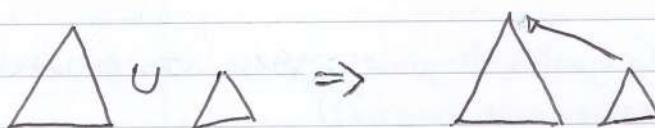
## Heurística da união ponderada

1 - manter em cada nó uma estimativa (superior) da sua altura

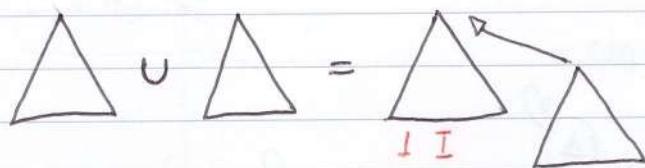
nó [value | parent | height]

2 - ligar a raiz de mais baixa à mais alta

2.1 - Alturas diferentes  $\rightarrow$  cotas das alturas não mudam



2.2 - Alturas iguais  $\rightarrow$  cota da nova raiz += 1

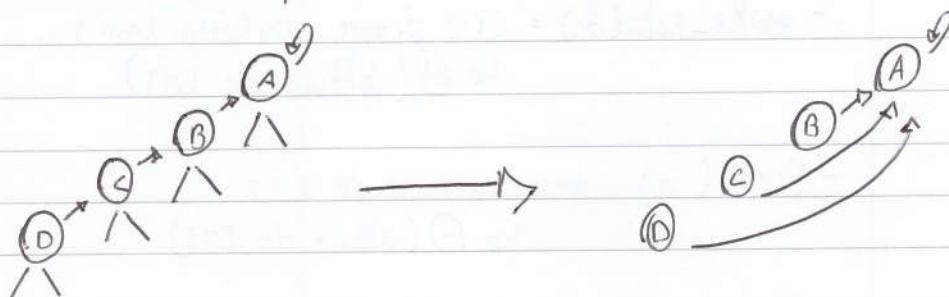


## Análise

### Teoria

Uma sequência de  $m \geq n$  operações find / union usando união ponderada sobre um conjunto com  $n$  elementos requer tempo  $O(n \lg(n))$  no pior caso

## Heurística da compreensão de caminhos



## Algoritmo

```
find (node)
    if node.parent == node
        return node
    else
        return find (node.parent)
```

Algoritmo find-path compression ( $x$ )

se  $x.parent \neq x$  então

$x.parent \leftarrow \text{find-path compression}(x.parent)$

fim-se

return  $x.parent$

Análise

Teoria

O custo (amortizado) de  $m$  operações union/find com compressão de caminhos + união ponderada sobre um conjunto de  $n$  elementos é  $\mathcal{O}(m \lg^* n)$

+  $\lg^* n = \log_2$  iterado de  $n$

= número de vezes que precisa aplicar  $\lg$  até que o resultado  $\leq 1$

Exemplo

$\rightarrow \lg k$  dígitos

$$\lg 2^{65536} = 65536$$

$$\lg 65536 = 16$$

$$\lg 16 = 4$$

$$\lg 4 = 2$$

$$\lg 2 = 1$$

---

$$\lg^* 2^{65536} = 5$$

+ tempo constante por operação, na prática

# Grafos

- Conjunto de elementos (nós / vértices)
- Conjunto de ligações binárias entre esses elementos (arestas)
- Estrutura de dados flexível para relações binárias

## Exemplos

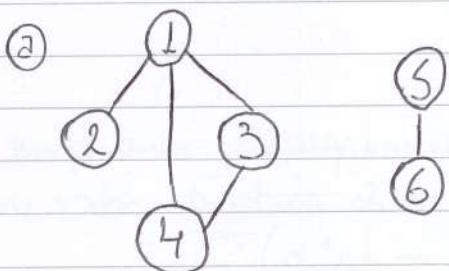


Gráfico simples

- 6 vértices

- 5 arestas

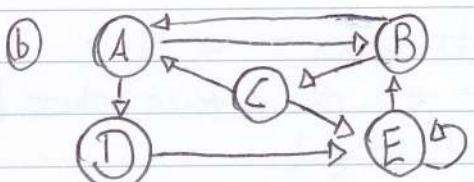


Gráfico dirigido

- 5 vértices

- 9 arestas

## Definição

Gráfico  $G = (V, E)$  onde

$V = \{v_0, \dots, v_{n-1}\}$  vértices

$E = \{\{v_{i_k}, v_{j_k}\} \mid 0 \leq k < \frac{n(n-1)}{2}; 0 \leq i_k \neq j_k < n\}$  arestas

## Exemplo

$$V = \{1, 2, 3, 4, 5, 6\}$$

$$E = \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{3, 4\}, \{5, 6\}\}$$

Gráfico dirigido  $G = (V, E)$  onde

$V = \{v_0, \dots, v_{n-1}\}$  vértices

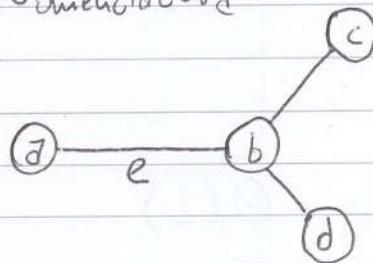
$E = \{(v_{i_k}, v_{j_k}) \mid 0 \leq k < n^2; 0 \leq i_k, j_k < n\}$

## Exemplo

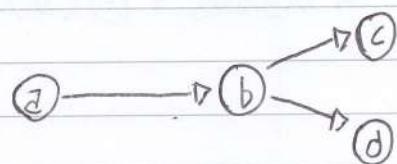
$$V = \{A, B, C, D, E\}$$

$$E = \{(A, B), (A, D), (B, A), (B, C), (C, A), (C, E), (D, E), (E, B), (E, E)\}$$

## Nomenclatura



$a, b$  - vértices adjacentes  
 $e$  - aresta incidente a  $a, b$   
 grau -  $\text{gr}(v) = n^{\circ}$  de arestas incidentes a  $v$   
 e.g.:  $\text{gr}(b) = 3$



grau de entrada  
 $\text{gr}^+(v) = n^{\circ}$  de arestas que "chegam" a  $v$   
 (aférentes)

e.g.:  $\text{gr}^+(b) = 1$

grau de saída

$\text{gr}^-(v) = n^{\circ}$  de arestas que "saem" de  $v$   
 (eférentes)

e.g.:  $\text{gr}^-(b) = 2$

## Representação

### (I) Matriz de adjacências

$$A_{n \times n} = \begin{bmatrix} d_{00} & \dots & d_{0n} \\ \vdots & \ddots & \vdots \\ d_{n0} & \dots & d_{nn} \end{bmatrix}$$

$$d_{ij} = \begin{cases} 1, & \text{se } (v_i, v_j) \in E \\ 0, & \text{se } (v_i, v_j) \notin E \end{cases}$$

### Exemplos

+ A	③	1	2	3	4	5	6
$\text{diag}(A) = \vec{0}$	2	0	1	1	1	0	0
matriz simétrica	3	1	0	0	1	0	0
A =	4	1	0	1	0	0	0
	5	0	0	0	0	1	0
	6	0	0	0	0	1	0

	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	0	0
C	0	1	0	0	1
D	0	0	0	0	1
E	0	1	0	0	1

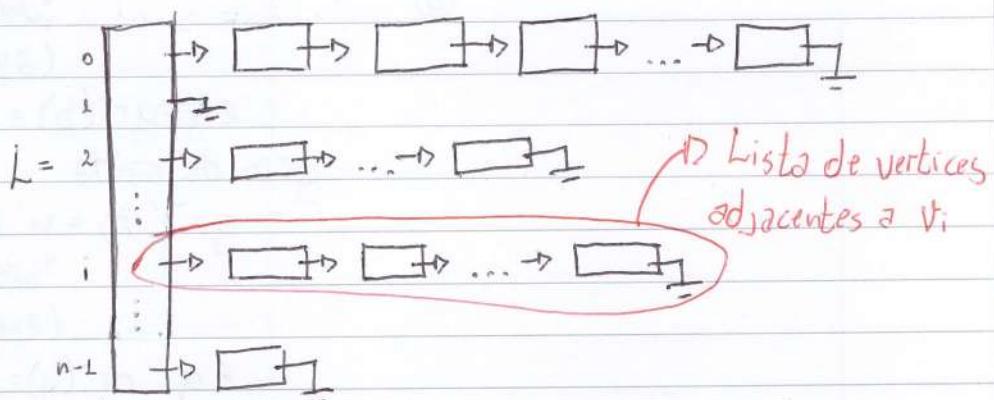
## Análise

- Espaço =  $\Theta(v^2)$

- Tempo consulta à aresta =  $\Theta(1)$

- Tempo para percorrer os vizinhos =  $\Theta(v)$

## II Listas de adjacências



### Exemplos

+ Grafo simples

$n$ º de nós

das listas

$$= 2E$$

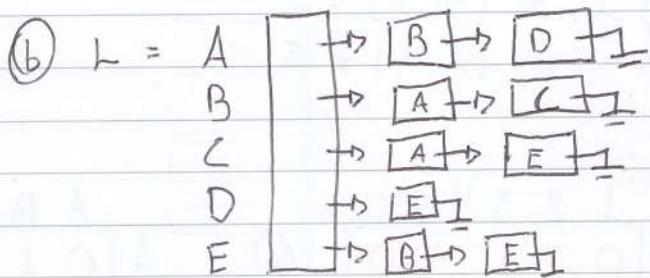
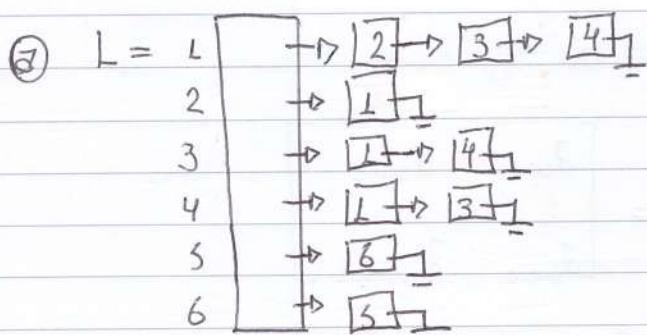
+ Convenção

nós das listas

ordenadas

pelos rótulos

dos vértices



## Análise

- Espaço =  $\Theta(V + E)$  = linear no tamanho do grafo

↳ Entrada no array h

↳ 1 ou 2 vezes números de elementos nas listas de adjacências

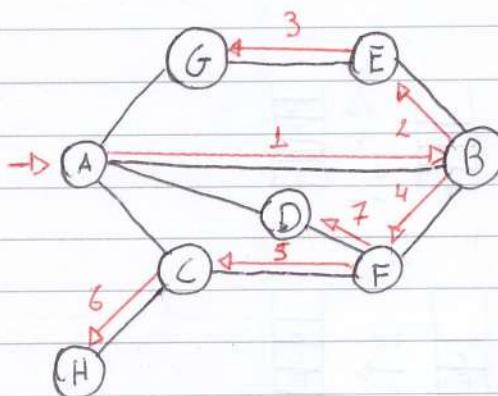
- Tempo para consultar 1 aresta =  $\Theta(1)$  no pior caso

- Tempo para percorrer todos os vizinhos em um vértice  
=  $\Theta(E)$  no pior caso

## Percursos

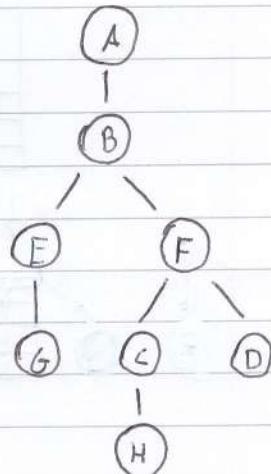
(I) Em profundidade  
Depth-First Search (DFS)

Exemplo



Ordem DFS  
A, B, E, F, C, H, D

Árvore DFS



Algoritmo DFS

Entrada:  $G = (V, E)$

início  $n$

$P \leftarrow (0, \dots, 0)$  // marca nós visitados

para  $s \leftarrow 0, \dots, n-1$  faça

se  $\neg P[s]$  então

dfs-visit( $G, s, P$ )

Fim-se

Fim-Faça

Fim

Algoritmo dfs-visit

Entrada:  $G = (V, E)$

$s = \text{vertice de origem}$

$P = \text{marcadores de nós visitados}$

início

$P[s] \leftarrow 1$

pre-visit ( $G, s$ )

$e \leftarrow G[s]$

enquanto  $e \neq \perp$  faça

se  $\top P[e \rightarrow \text{value}]$  então

dfs-visit ( $G, e \rightarrow \text{value}, P$ )

Fim-se

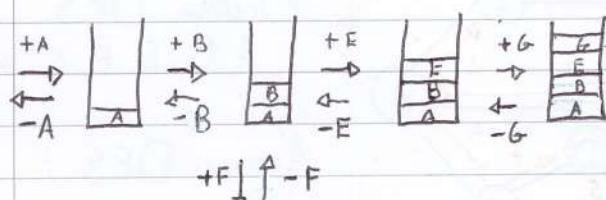
$e \leftarrow e \rightarrow \text{next}$

Fim-faça

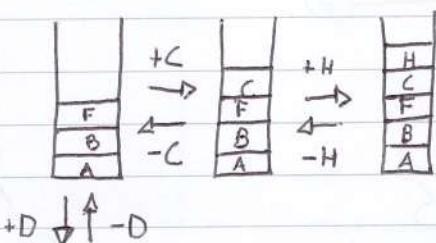
post-visit ( $G, s$ )

Fim

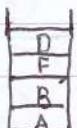
Pilha Recursiva



+F ↓ ↑ -F



+D ↓ ↑ -D



## Análise

- Cada vértice é visitado exatamente uma vez
  - Para cada vértice visitado, `dfs-visit` considera uma das suas arestas incidentes / eferentes
    - { duas vezes, se  $G$  simples
    - uma vez, se  $G$  dirigido
- +  $O(V + E)$

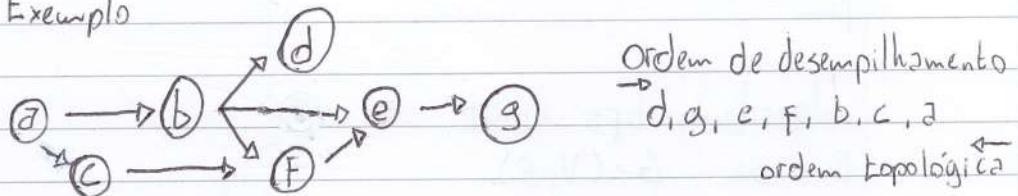
## Ordenação Topológica

- Codificar relações de dependências com grafo dirigido

$$a \rightarrow b \Rightarrow \begin{cases} b \text{ depende de } a \\ a \text{ precede } b \end{cases}$$

- Ordenar vértices respeitando as dependências, i.e.: cada vértice só aparece depois de todos os vértices dos quais ele depende

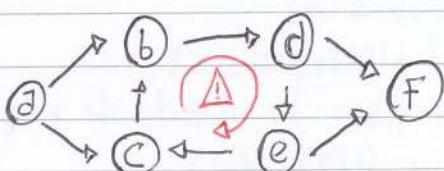
Exemplo



Ordem Topológica

a, b, c, d, f, e, g

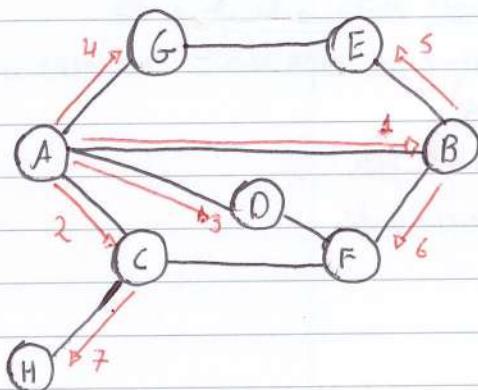
+ ordem topológica não é única



+ gráfico cíclico não possui ordem topológica

- no DFS, o nó só é desempilhado após todos os nós que dependem dele terem sido visitados. Os nós são desempilhados

II Em largura  
(Breadth-First Search (BFS))



Ordem BFS

A, B, C, D, G, E, F, H

Algoritmo

Entrada  $G = (V, E)$

início

$P \leftarrow (0, \dots, 0)$

para  $s = 0, \dots, n-1$  faça

se  $T[s] = 0$  então

bfs-visit( $G, s, P$ )

Fim-se

Fim-faça

fim

Algoritmo bfs-visit

Entrada :  $G = (V, E)$

$S$  vértice origem

$P = (p_0, \dots, p_{n-1})$  indicador nós marcados para visita

início

$P[S] \leftarrow L$

front, rear  $\leftarrow$  NOVA fila( $V, \dots, V$ )

front, rear  $\leftarrow$  enqueue(front, rear, S)

enquanto front  $\neq$  rear faça // enquanto fila não vazia

front, rear, u  $\leftarrow$  dequeue(front, rear)

previsit( $G, u$ )

e  $\leftarrow G[u]$

enquanto e  $\neq \perp$  faça

se  $T[e] \rightarrow \text{val}$  então

| front, rear  $\leftarrow$  enqueue(front, rear, e  $\rightarrow$  val)

PL eval & L

fin-se

$e \rightarrow e \& \text{next}$

fin-fa2

post-visit( $G, v$ )

fin-fa3

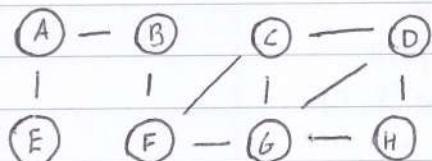
fin

# Caminhos Mínimos em Grafos

## Grafos Simples

Qual o caminho mínimo (menor número de arestas) de um vértice  $s$  a todos os vértices do grafo?

Exemplo



$$\delta(A, A) = 0$$

$$\delta(A, B) = 1$$

$$\delta(A, D) = 4$$

+ Na BFS iniciada em  $s$ , os vértices são marcados por ordem de distância a  $s$ .

u	Filha(s)	Distância								From
		A	B	C	D	E	F	G	H	
B		∞	0	∞	∞	∞	∞	∞	∞	? B ? ? ? ? ? ?
B	A, F	1				1				B B
A	F, E				2					A
F	E, C, G			2		2				F F
E	C, G									
C	G, D				3					C
G	D, H						3			G
D	H									
H										
		1	0	2	3	2	1	2	3	BBFCABA BFG

Algoritmo shortest-path

Entrada:  $G = (V, E)$

$s$  vértice origem

Saída:  $D = (d_0, \dots, d_{n-1})$  tal que  $D[j] = \delta(s, j)$

$F = (f_0, \dots, f_{n-1})$  tal que  $F[j]$  = precursor de  $j$  no caminho mínimo  $s \rightarrow j$

inicio

$$D \leftarrow (\infty, \dots, \infty)$$

$$F \leftarrow (-L, \dots, -L)$$

$$D[S] \leftarrow 0$$

$$F[S] \leftarrow S$$

$F, r \leftarrow$  nova fila (vazia)

$F, r \leftarrow$  enqueue ( $F, r, S$ )

enquanto  $F \neq r$  faça

$F, r, u \leftarrow$  dequeue ( $F, r$ )

$e \leftarrow G[u]$

enquanto  $e \neq L$  faça

se  $D[e \rightarrow \text{value}] \neq \infty$  então

$F, r \leftarrow$  enqueue ( $F, r, e \rightarrow \text{value}$ )

$D[e \rightarrow \text{value}] \leftarrow D[u] + 1$

$F[e \rightarrow \text{value}] \leftarrow u$

Fim-se

$e \leftarrow e \rightarrow \text{next}$

Fim-faça

Fim-faça

return  $D, F$

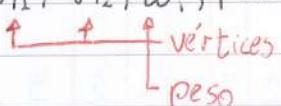
Fim

## Grafo Ponderados

Definição

Cada aresta possui peso associado

$$E = \{(v_{i1}, v_{i2}, w_i) | \dots\}$$

 vértices  
peso

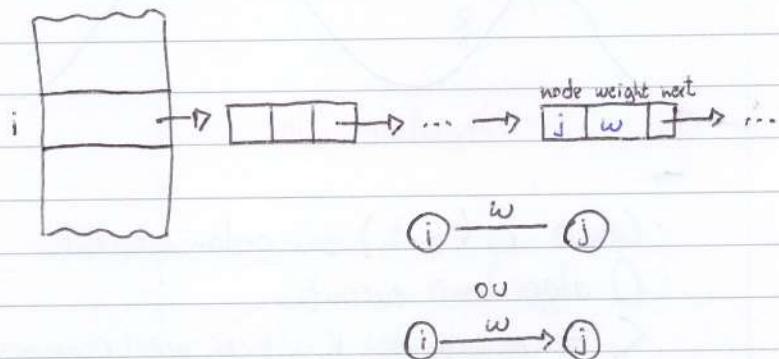
Representação

Matriz de adjacências

$$\begin{matrix} & 0 & j & n-1 \\ \begin{matrix} 0 \\ i \\ n-1 \end{matrix} & \left[ \begin{matrix} & & \\ & \vdots & \\ & w_{ij} & \end{matrix} \right] & \end{matrix}$$

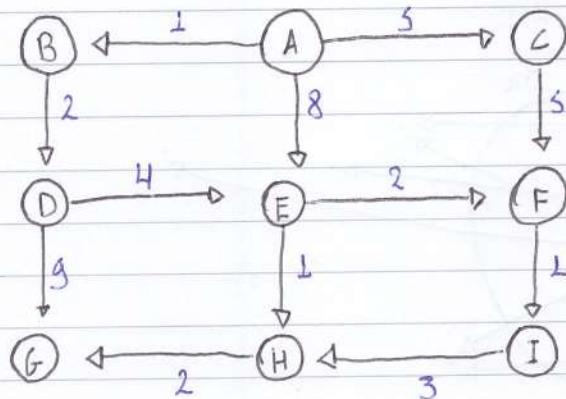
$w_{ij} =$  peso da aresta  $(i, j)$

## Listas de adjacências



- Comprimento do caminho  $\stackrel{\text{def}}{=} \text{Soma dos pesos das arestas no caminho}$
- $\delta(s, t) \stackrel{\text{def}}{=} \text{Comprimento do caminho mínimo}$
- +  $s \rightarrow t = \text{aresta } (s, t)$
- +  $s \rightsquigarrow t = \text{caminho de } s \text{ a } t$

### Exemplo



$$\delta(A, E) = 7$$

caminho mínimo =  $A \rightarrow B \rightarrow D \rightarrow E$

### Algoritmo de Dijkstra

Calcula as distâncias mínimas a partir de origem comum  $s$

#### + Algoritmo Gúloso

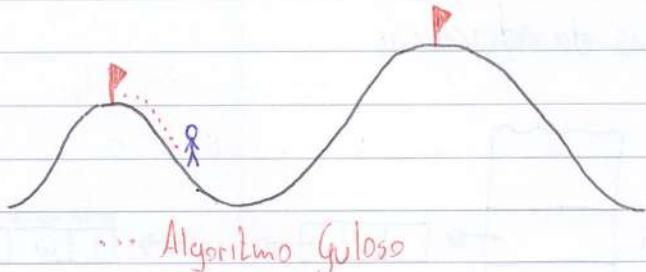
- Heurística de otimização

- Processo iterativo

A cada passo, escolhe o min/máx local na esperança de alcançar min/máx global

- Problema potencial

... para min/máx local



- Calcula  $\delta(s, t)$  em ordem crescente
- O algoritmo encontra  
 $S_K \stackrel{\text{def}}{=} \begin{cases} \text{conjunto dos } K \text{ vértices mais próximos à } s \\ \text{para } K=1, \dots, n=|V| \end{cases}$

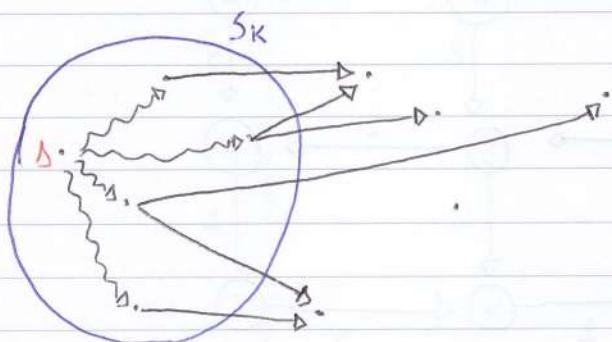
- Início

$$\delta(s, s) = 0$$

$$S = \{s\}$$

- Atualização

$$S_K \xrightarrow{?} S_{K+1}$$



Lema

O  $(K+1)$ -ésimo vértice mais próximo à origem  $s$ ,  $t \in V - S_n$   
tem seu precursor em  $S_K$

Prova

Seja  $\pi: s \rightsquigarrow t$  caminho mínimo de  $s$  a  $t$

Temos que  $\pi: s \rightsquigarrow u \xrightarrow{w} t$  para algum  $u \in V \setminus \{s, t\}$

Suponha  $u \notin S_K$ . Como  $w > 0$  então  $\delta(s, u) < \delta(s, t)$ .

Como  $u \notin S_K$  então  $t$  não pode ser o  $(K+1)$ -ésimo nó mais próximo a  $s$ , por que pelo menos  $u \notin S_K$  está mais próximo

$v \in V - S_k$  é escolhido a partir de  $S_k$  minimizado  
 $d(s, v) = \min_{u \in S_k} \{ \delta(s, u) + w(u, v) \}$

onde  $w(u, v) = \begin{cases} \kappa, & \text{se } u \xrightarrow{\kappa} v \in E \\ \infty, & \text{caso contrário} \end{cases}$

- O algoritmo mantém estimativas  $\Delta[t]$  (cota superior) para  $\delta(s, t)$  e, a cada iteração, escolhe o próximo vértice de maneira gulosa:  $v = \arg\min \{\Delta[t]\}, t \in S_k$

- Para cada vértice  $v$  adicionando à fronteira  $S_k$ , apenas as estimativas dos seus vizinhos precisão ser atualizadas

$\Delta$	A	B	C	D	E	F	G	H	I
$k=0$	$0_A^*$	$\infty$							
1	$0_A$	$1_A^*$	$5_A$		$8_A$				
2		$1_A$		$3_B^*$					
3			$5_A^*$	$3_B$	$7_0$		$12_0$		
4			$5_A$		$7_0^*$	$10_c$			
5					$7_0$	$9_E$		$8_E^*$	
6						$9_E^*$	$10_H$	$8_E$	
7						$9_E$	$10_H^*$		$10_F$
8							$10_H$		$10_F^*$
9								$8_E$	$10_F$
	$0_A$	$1_A$	$5_A$	$3_B$	$7_0$	$9_E$	$10_H$	$8_E$	$10_F$

Algoritmo Dijkstra

Entrada  $G = (V, E)$  grafo ponderado com pesos  $> 0$

1 vértice origem

Saída  $D = (D[0], \dots, D[n-1])$  tal que  $D[t] = \delta(s, t)$

$F = (F[0], \dots, F[n-1])$   $F[t] = \text{precursor de } t \text{ no caminho mínimo } s \rightarrow t$

início

$D \leftarrow (\infty, \dots, \infty)$

$F \leftarrow (-1, \dots, -1)$

$D[b] \leftarrow 0$

$F[\Delta] \leftarrow \Delta$

$H \leftarrow$  nova min-heap

min-heap-insert ( $H, \{0, \Delta\}$ )

para  $h \leftarrow 0, \dots, n-1$  faz

$(d, u) \leftarrow$  min-heap-extract ( $H$ )

$e \leftarrow G[u]$

enquanto  $e \neq \perp$  faz

se  $D[e \rightarrow \text{node}] > D[u] + e \rightarrow \text{weight}$  então

$D[e \rightarrow \text{node}] \leftarrow D[u] + e \rightarrow \text{weight}$

$F[e \rightarrow \text{node}] \leftarrow u$

min-heap-update ( $H, \{D[e \rightarrow \text{node}], e \rightarrow \text{node}\}$ )

final-se

$e \leftarrow e \rightarrow \text{next}$

final-faz

final-faz  
retorna  $\{D, F\}$

final

Análise

- A heap possui  $\mathcal{O}(V)$  elementos
- heap-extract é executado  $V$  vezes
- heap-update é executado  $\leq E$  vezes

$$T(G = (V, E)) = \mathcal{O}((V+E)\lg(V))$$

Árvores Geradoras de Custo Mínimo  
(Minimum Cost Spanning Trees - MST)

Problema

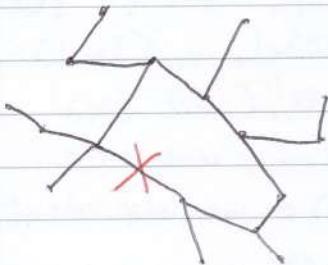
Dado um grafo conexo ponderado  $G = (V, E)$  com função peso  $w: E \rightarrow \mathbb{R}_+$  encontrar subgrafo conexo  $T = (V, E' \subseteq E)$  tal que a soma dos pesos das arestas  $C(T) = \sum_{e \in E'} w(e)$  seja mínima

Teorema

T é uma árvore = grafo conexo acíclico

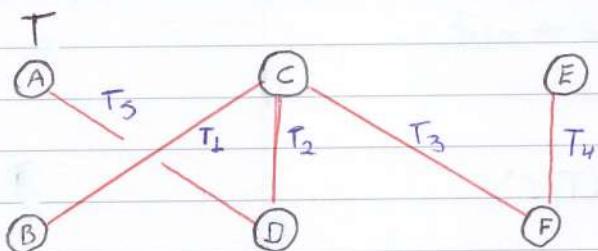
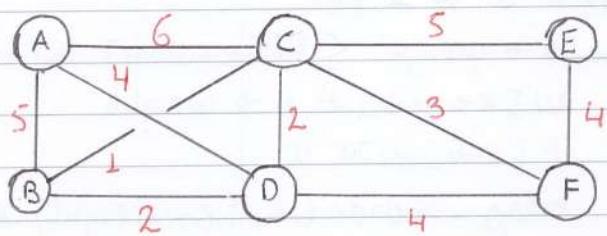
Prova

Retirar arestas de ciclos não quebra a conexidade e diminui o custo total



Algoritmo de Prim

- Construir T progressivamente de modo guloso. A cada passo adiciona à árvore de menor peso  $(u, v, w)$  tal que  $u \in T$ ,  $v \in T'$



$$C(T) = 14$$

+ A MST não é única

Algoritmo prim

Entrada:  $G = (V, E)$  grafo conexo com pesos  $> 0$

Saída:  $T = \text{MST}(G)$

$$C(T)$$

início

$W \leftarrow (\infty, \dots, \infty)$

$F \leftarrow (-1, \dots, -1)$

$r \leftarrow$  vértice qualquer  $\in V$

$W[r] \leftarrow 0$

$H \leftarrow$  nova min-heap

min-heap-insert ( $H, \{O, r\}$ )

$T \leftarrow [T[0]=1, \dots, T[n-1]=1]$  // lista de adjacências com todos os vértices e sem arestas

$C \leftarrow 0$  //  $C(T)$

para  $K=0, \dots, n-1$  faça

$(w, v) \leftarrow$  min-heap-extract ( $H$ )

$C \leftarrow C + w$

se  $K > 0$  então

list-append ( $T[v], \{v, F[v], w\}$ )

list-append ( $T[F[v]], \{F[v], v, w\}$ )

fim-se

$e \leftarrow G[v]$

enquanto  $e \neq \perp$  faça

se  $e \rightarrow weight < W[e \rightarrow node]$  então

$W[e \rightarrow node] \leftarrow e \rightarrow weight$

$F[e \rightarrow node] \leftarrow v$

heap-update ( $H, \{e \rightarrow weight, e \rightarrow node\}$ )

fim-se

$e \leftarrow e \rightarrow next$

fim-faça

fim-faça

retorna ( $T, C$ )

fim

Análise

Identico ao Dijkstra

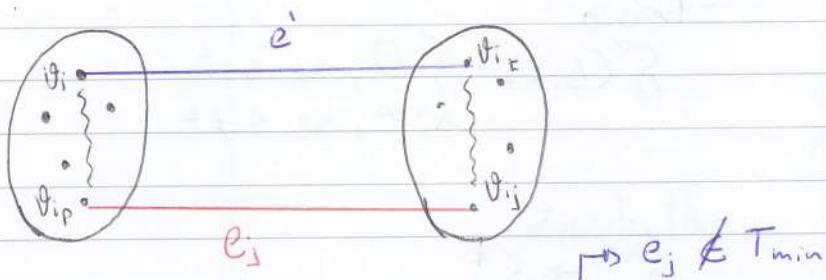
$O((V+E) \lg(V))$

Teorema

O Algoritmo de Prim produz uma MST

P prova

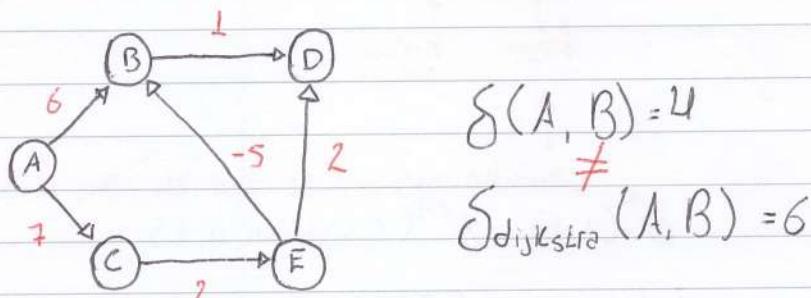
Seja  $T_{\text{prim}}$  a árvore gerada pelo algoritmo de Prim e  $T_{\text{min}}$  MST tal que  $C(T_{\text{min}}) < C(T_{\text{prim}})$ . Seja  $v_{i_0}, v_{i_1}, \dots, v_{i_{n-1}}$  a ordem de escolha dos vértices pelo algoritmo de Prim. Seja  $e_j = (v_{i_p}, v_{i_q})$   $p < q$  a primeira aresta  $\notin T_{\text{min}}$  escolhida pelo algoritmo. Por conexidade,  $T_{\text{min}}$  possui caminho  $v_{i_p} \rightarrow v_{i_q}$ . Como  $e_j \notin T_{\text{min}}$  esse caminho possui aresta  $e' = (v_{i_s}, v_{i_t})$  para algum  $s < p$  e  $t > q$  que cruza o corte  $V = \{v_{i_0}, \dots, v_{i_p}\} \cup \{v_{i_{q+1}}, \dots, v_{i_{n-1}}\}$ .



- Considere o grafo  $T' = T_{\text{min}} \cup \{e_j\}$  e defina  $T'' = T' - \{e'\}$ . Como, por definição  $e_j$  é a menor aresta que cruza o corte ( $w(e_j) < w(e')$ ) temos  $C(T'') = C(T') - w(e') = C(T_{\text{min}}) + w(e_j) - w(e')$ .  
 $\therefore C(T'') < C(T_{\text{min}}) \rightarrow \text{contradiction}$

### Grafos com Peso Negativos

+ O Algoritmo de Dijkstra pode falhar com pesos  $\leq 0$



+ Se grafo possui ciclos negativos o problema não possui solução.

- Suponha (inicialmente)  $G$  sem ciclos negativos

Queremos computar

$d(s, t)$ ,  $\forall t \in G$  ( $s$  = origem em comum)

+ Um caminho mínimo  $s \rightsquigarrow t$  possui  $\leq n-1$  arestas.  
Do contrário,  $\pi$  possui ciclos  $> 0$  que podem ser removidos

- Defina

$\delta^k(s, t) \stackrel{\text{def}}{=} \text{distância mínima de } s \text{ a } t \text{ com } \leq k \text{ arestas}$

- Precisamos computar  $\delta^0(s, t), \delta^1(s, t), \dots, \delta^{n-1}(s, t)$

- Início

$$\delta^0(s, t) = \begin{cases} 0, & \text{se } s = t \\ +\infty, & \text{se } s \neq t \end{cases}$$

- Atualização

$$\delta^{k-1} \rightarrow \delta^k$$

Seja  $\pi: s \rightsquigarrow t$  caminho mínimo de  $s$  a  $t$  com  $\leq k$  arestas

2 casos

- Se  $\pi$  tem  $< k$  arestas ( $\leq k-1$  arestas)

$$\delta^k(s, t) = \delta^{k-1}(s, t)$$

- Se  $\pi$  tem  $= k$  arestas, então temos

$$\pi: \underbrace{s \rightsquigarrow u}_{K-1 \text{ arestas}} \xrightarrow{+} t$$

K-ésima aresta



caminho mínimo se  $s$  a  $u$  com  $\leq k-1$  arestas

$$\delta^k(s, t) = \delta^{k-1}(s, u) + w(u, t)$$

$$\delta^k(s, t) = \min \left\{ \begin{array}{l} \delta^{k-1}(s, t), \\ \delta^{k-1}(s, u) + w(u, t) \end{array} \right\}$$

E se o grafo possui ciclos negativos?

Então

$G = (V, E)$  possui ciclos negativos se e somente se  $\delta^{n-1}(s, t) > \delta^n(s, t)$  para algum  $t \in V$

Prova

Seja  $t \in V$  tal que  $\delta^n(s, t) < \delta^{n-1}(s, t)$  e seja  $\pi: s \rightarrow t$  caminho mínimo com  $< n$  arestas (i.e.  $|\pi| = \delta^n(s, t)$ ) se possui  $= n$  arestas pois, do contrário,  $\delta^{n-1}(s, t) = \delta^n(s, t)$ , e logo possui ciclo  $C$ . Se  $|C| > 0$ , podemos eliminá-lo obtendo  $\pi': s \rightarrow t$  com  $< n$  arestas e  $|\pi'| < \delta^n(s, t)$ .  $\rightarrow | \leftarrow$

Pela contrapositiva, suponha que  $\delta^n(s, t) = \delta^{n-1}(s, t) \forall t$ . Como  $\delta^n(s, t) = \min \{ \delta^{n-1}, \min \{ \delta^{n-1}(s, u) + w(u, t) \} \}, (u, t) \in E$  então  $\delta^n(s, t) = \delta^{n-1}(s, t) \leq \delta^{n-1}(s, u) + w(u, t) \forall (u, t) \in E$ . Seja  $C$  ciclo arbitrário. Temos  $|C| = \sum_{(u, v) \in C} w(u, v) \geq \sum_{(u, v) \in C} \delta^{n-1}(s, u) - \delta^{n-1}(s, u) = 0$ .  $\therefore |C| > 0$ .

Algoritmo bellman-ford

Entrada:  $G = (V, E)$  grafo ponderado

$s$  vértice de origem

Saída:  $w_C$  booleano indicando se  $G$  tem ciclos  $< 0$

$D = (\delta_0 = \delta(s, 0), \dots, \delta_{n-1} = \delta(s, n-1))$  distâncias

$F = (f_0, \dots, f_{n-1})$  precursores

início

$$D = \begin{bmatrix} d[0,0] = \delta^0(s,0) & \dots & d[0,n-1] = \delta^0(s,n-1) \\ \vdots & \ddots & \vdots \\ d[n-1,0] = \delta^{n-1}(s,0) & \dots & d[n-1,n-1] = \delta^{n-1}(s,n-1) \end{bmatrix} \leftarrow \begin{bmatrix} \infty & \dots & \infty \\ \vdots & \ddots & \vdots \\ \infty & \dots & \infty \end{bmatrix}$$

$$F = (-1, \dots, -1)$$

$$D[0, s] \leftarrow 0$$

para  $K \leftarrow 1, \dots, n-1$  faça

para  $t \leftarrow 0, \dots, n-1$  faça

$$D[K, t] \leftarrow D[K-1, t] \quad // \text{caso 1}$$

fim-faça

para  $u \leftarrow 0, \dots, n-1$  faça

$$e \in G[u]$$

enquanto  $e \neq s$  faça

$t, w \in e.hoc, e.weight \quad // \text{caso 2}$

se  $D[K, t] > D[K-1, u] + w$  então

$$D[K, t] \leftarrow D[K-1, u] + w$$

$$F[t] \leftarrow u$$

fim-se

$e \leftarrow e \rightarrow \text{next}$

Fim-Fase

Fim-Fase

Fim-Fase

para  $u \leftarrow 0, \dots, n-1$  Fase

$e \leftarrow G[u]$

enquanto  $e \neq \perp$  Fase

$t, w \in e.\text{node}, e.\text{weight}$

se  $D[n-1, t] < D[n-1, u] + w$  então

retorna ( $\text{True}, t, l$ )

Fim-se

$e \leftarrow e \rightarrow \text{next}$

Fim-Fase

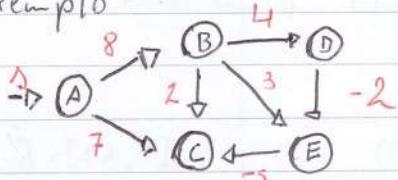
Fim-Fase

retorna ( $\text{False}, \underline{D[n-1, \perp]}, F$ )

Fim

última linha  
de  $D$

Exemplo



R	t	A	B	C	D	E
0		$0_A$	$\infty$	$\infty$	$\infty$	$\infty$
1			$8_A$	$7_A$		
2					$12_B$	$11_B$
3				$6_E$		$10_D$
4				$5_E$		
5		$0_A$	$8_A$	$5_E$	$12_B$	$10_D$

→ Não tem ciclo negativo

Análise

Tempo  $T(G = (V, E)) = \mathcal{O}(VE)$

Espaço  $\mathcal{O}(V) \rightarrow$  matriz só precisa de 2 linhas

Caminhos mínimos entre todos os grafos

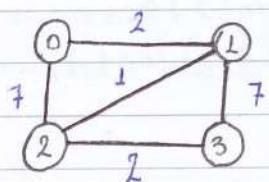
Dado: Grafo ponderado  $G = (V, E)$

Calcular:  $\delta(s, t)$  distâncias mínimas entre todos os pares  $s, t \in V$

+ Usar Bellman-Ford iniciando em cada vértice  
 $O(V^2E)$

Definição  
 $\delta^k(s, t) \stackrel{\text{def}}{=} \text{distância mínima de } s \text{ a } t \text{ passando apenas por}$   
vértices em  $\{0, \dots, k-1\}$  excluindo as extremidades  
os  $k$  primeiros vértices

Exemplo



$$\begin{aligned}\delta^2(0,3) &= 9 & \pi: 0 \rightarrow 1 \rightarrow 5 \\ \delta^3(0,3) &= 5 & \pi: 0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \\ \delta^2(3,1) &= 7 \\ \delta^3(3,1) &= 3\end{aligned}$$

$$\delta(s, t) = \delta^{n-1}(s, t)$$

Início

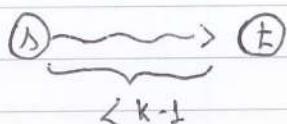
$$\delta^0(s, t) = \begin{cases} 0, & \text{se } s = t \\ w(s, t), & \text{se } s \neq t \wedge (s, t) \in E \\ \infty, & \text{caso contrário} \end{cases}$$

Atualização

$$\delta^k(s, t), k > 0$$

1º caso

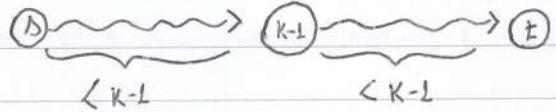
O caminho mínimo não passa por  $k-1$



$$\delta^k(s, t) = \delta^{k-1}(s, t)$$

• 2º caso

O caminho mínimo passa por  $k-l$



$$\delta^k(s, t) = \delta^{k-l}(s, k-l) + \delta^{k-l}(k-l, t)$$

• Caso geral

$$\delta^k(s, t) = \min \{ \delta^{k-l}(s, t), \delta^{k-l}(s, k-l) + \delta^{k-l}(k-l, t) \}$$

Algoritmo floyd-warshall

Entrada:  $G = (V, E)$  grafo ponderado

Saída:  $D_{n \times n} = [D[s, t] = \delta(s, t)]_{0 \leq s, t \leq n-1}$

$F_{n \times n} = [F[s, t] = F^n(s, t)]_{0 \leq s, t \leq n-1}$

Início

$$D^0_{n \times n} = [D^0[s, t]]_{0 \leq s, t \leq n} \leftarrow [\delta^0(s, t)]_{0 \leq s, t \leq n}$$

$$P^0_{n \times n} = [P^0[s, t]]_{0 \leq s, t \leq n}$$

para  $K = 1, \dots, n$  faça

para  $s = 0, \dots, n-1$  faça

$$D^K[s, t] \leftarrow \min \left\{ D^{K-1}[s, t]; D^{K-1}[s, k-l] + D^{K-1}[k-l, t] \right\}$$
$$P^K[s, t] \leftarrow \begin{cases} P^{K-1}[s, t], & \text{se } D^K[s, t] = D^{K-1}[s, t] \\ P^{K-1}[k-l, t], & \text{se } D^K[s, t] = D^{K-1}[s, k-l] + D^{K-1}[k-l, t] \end{cases}$$

fim-faça

se  $D[n, n] < 0$  então

exit(1, "Erro: grafo com ciclos negativos")

fim-se

fim-faça

fim-faça

retorna  $(D^n, P^n)$

Fim

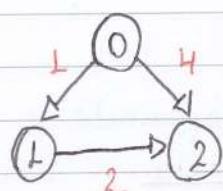
Análise

$$\text{Tempo: } T(G = (V, E)) = \mathcal{O}(V^3)$$

$$\text{Espaço: } S(G = (V, E)) = \mathcal{O}(V^2) \quad \text{?}$$

Só precisa armazenar as matrizes  $D^K$  e  $D^{K-L}$

Exemplo



$$D^0 = \begin{bmatrix} 0 & L & 4 \\ \infty & 0 & 2 \\ \infty & \infty & 0 \end{bmatrix}$$

$$D^L = D^0$$

$$D^2[0, 2] = \min \{$$

$$D^L[0, 2] = 4,$$

$$D^L[0, L] + D^L[L, 2] = 3$$

$$\} \quad \begin{matrix} " \\ L \end{matrix} \quad \begin{matrix} " \\ 2 \end{matrix}$$

$$D^2 = \begin{bmatrix} 0 & L & 3 \\ \infty & 0 & 2 \\ \infty & \infty & 0 \end{bmatrix}$$

$$D^3 = D^2$$

Reconstrução dos caminhos

Definição

$F^K(s, t)$  = precursor de  $t$  no caminho mínimo de  $s \rightarrow t$  podendo passar apenas pelos  $K$  primeiros vértices ( $0, \dots, K-1$ )

• 1º caso

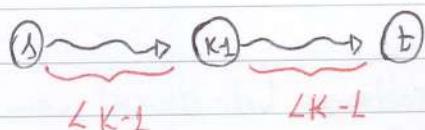
Caminho mínimo não passa por  $K-1$



$\underbrace{s \rightsquigarrow}_{\leq K-1} \underbrace{K-1 \rightsquigarrow t}_{\leq K-1}$  precursor = mesmo do caminho com  $K-1$

• 2º caso

Caminho mínimo passa por  $K-1$



precursor é o mesmo do caminho de  $K-1 \rightarrow t$  com  $K-1$  intermediações

# Introdução à Computabilidade e Complexidade Computacional

## Pergunta Ø

Todos os problemas têm solução algorítmica?  
"Resposta" nos anos 1930

- Alonzo Church
- Kurt Gödel
- Alan Turing

## I) Tese de Church - Turing

Um problema pode ser resolvido se e somente se pode ser resolvido por uma Máquina de Turing.  
↳ Modelo "geral" de computação

## II) Problema da Parada

Dado o código fonte de um programa, decidir se sim ou não, ele sempre termina qualquer que seja a entrada.

### Turing (1936)

Não existe programa que "decide" o problema da parada

+ A "infinita" maioria dos problemas não são computáveis

## Pergunta 1

Todos os problemas computáveis têm solução eficiente?

### Teorema de Cobham - Edmonds (1965)

Eficiente = Polinomial

#### Definição

$P \stackrel{\text{def}}{=} \{ \text{problemas (de decisão) com solução polinomial} \}$

Problemas sem solução polinomial são chamados intratáveis

#### Definição

$\text{EXPTIME} = \{ \text{problemas (de decisão) com solução exponencial no tamanho da entrada} \}$

## Teoria da Hierarquia

P ⊂ EXPTIME ⊂ R

↳ problemas computáveis

Exemplo

Xadrez ∈ (EXPTIME - P) (por volta de 1984)

+ Existem muitos problemas práticos sem solução polinomial conhecida mas que também não foram provados intratáveis

Exemplo

SAT

Entrada:  $\varphi$  fórmula booleana FNC

Saída:  $\varphi$  é satisfatório

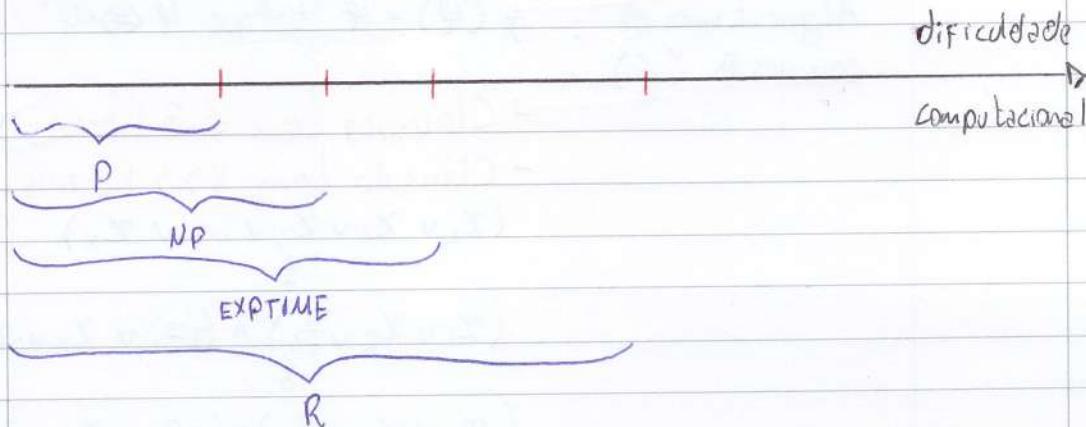
$$\varphi = (x \vee y \vee z) \wedge (\neg y \vee \neg z) \in \text{SAT}$$

$$\varphi' = (x \vee y) \wedge (\neg y \vee z) \wedge \neg z \wedge \neg z \notin \text{SAT}$$

+ Não se conhece nenhum algoritmo polinomial para SAT.  
Entretanto, dada uma valoração das variáveis é possível verificar se ela satisfaaz a fórmula em tempo polinomial

Definição

NP = {problemas (de decisão) com provas (de solução) verificáveis em tempo polinomial}

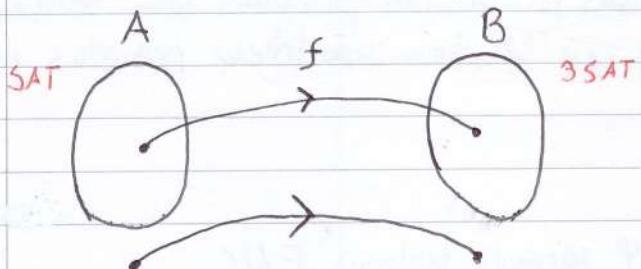


## Redução polinomial e NP - Completeness

Estudar a diferença relativa entre problemas NP e caracterizar os "mais difíceis"

### Definição Redução Polinomial

Uma redução de um problema de decisão A para um problema de decisão B é um algoritmo f que mapeia cada instância x de A numa instância f(x) de B tal que  $A(x) = \text{L} \leftrightarrow B(f(x)) = \text{L}$



Se f tem custo polinomial no tamanho de x, então f é dita redução polinomial ( $A$  é polinomialmente reduzível a  $B$ :  $A \leq_p B$ )

### Exemplo

3SAT: Testar se uma fórmula 3CNF (CNF com  $\leq 3$  literais por cláusula) é satisfatível

Afirmação

$SAT \leq_p 3SAT$

↳ notação

equivalente ( $\equiv$ )

↓

Algoritmo de:  $f(\varphi) = \varphi'$  tal que  $\varphi \leftrightarrow \varphi'$   
conversão (f)

- Cláusula com  $\leq 3$  literais ✓

- Cláusula com  $k > 3$  literais

$(x_1 \vee x_2 \vee x_3 \vee \dots \vee x_k)$

↑

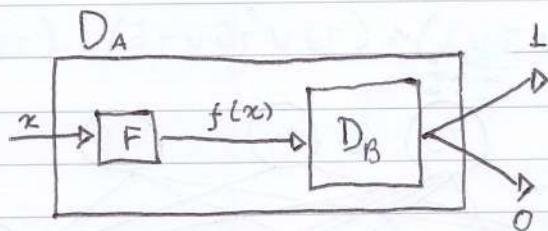
$(x_1 \vee x_2 \vee z_1) \wedge (z_1 \vee x_3 \vee x_4 \vee \dots \vee x_k)$

$(x_1 \vee x_2 \vee z_1) \wedge (z_1 \vee x_3 \vee z_2) \wedge (z_2 \vee x_4 \vee \dots \vee x_k)$

↓

$(x_1 \vee x_2 \vee x_3) \wedge (x_3 \vee x_4 \vee \dots \vee x_k) \vee \dots \vee (x_{k-1} \vee x_k)$

+  $A \leq_p B$  significa

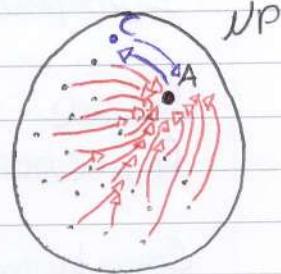


- $A$  não é mais difícil do que  $B$  ex eto por um fator polinomial
- $B$  é pelo menos tão difícil quanto  $A$  a menos de fator polinomial

Definição

Um problema  $A \in NP$  é dito  $NP$ -completo se  $\forall B, B \leq_p A$

+ Os problemas  $NP$ -C são pelo menos tão difíceis (a menos de fator polinomial) quanto qualquer outro problema  $NP$ .



Teorema de Cook-Levin (1971)  
 $SAT$  é  $NP$  completo

$$C \leq_p A$$
$$A \leq_p C$$

+ Se algum problema  $NP$ -C é também  $P$ , então  $P=NP$

+ Para provar que  $C$  é  $NP$ -Completo é suficiente provar  $A \leq_p C$  para algum  $A \in NP$ -C

Logo  $3SAT$  é  $NP$ -C

Exemplo

Clique

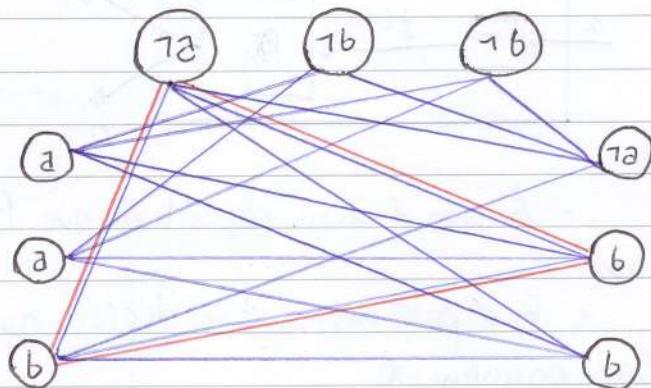
Entrada:  $G = (V, E)$  grafo,  $K \in \mathbb{R}$

Propriedade:  $G$  possui um  $K$ -clique  $\rightarrow$  subgrafo completo de  $K$  vértices

Clique é  $NP$  ✓

## Sub exemplo

$$q = (\alpha \vee \alpha \vee b) \wedge (\neg \alpha \vee \neg b \vee \neg b) \wedge (\neg \alpha \vee b \vee b)$$



## Exemplos

## Independent Set

Entrada:  $G = (V, E)$ ,  $K \in \mathbb{N}$

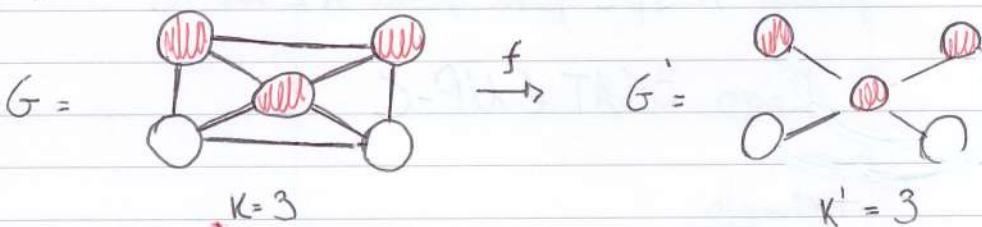
Propriedade:  $G$  possui conjunto independente de  $K$  vértices  
↳ Subconjunto de vértices sem  
nenhuma aresta entre eles

IS e NP ✓

Redução Clique  $\leq_p$  IS

$$\langle G, k \rangle \xrightarrow{f} \langle G', k' \rangle$$

instância de Clique      instância de IS



$F = \text{sobre } \langle G = (V, E), K \rangle$   
 $\text{output } \langle G' = (V, \neg E), K' = K \rangle$

Exemplo

### Vertex Cover

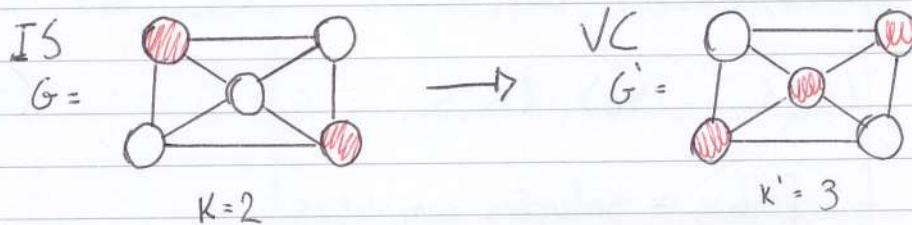
Entrada:  $G = (V, E)$ ,  $K$

Propriedade:  $G$  possui cobertura por  $K$  vértices

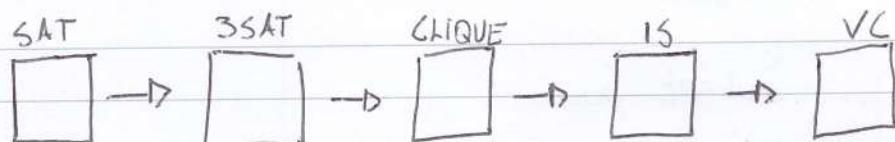
↳ subconjunto de vértices que  
"toca" em todos as arestas

VC é NP ✓

Redução: IS  $\leq_p$  VC



F = sobre  $\langle G = (V, E), K \rangle$   
output  $\langle G, |V| - K \rangle$



$SAT \leq_p VC$

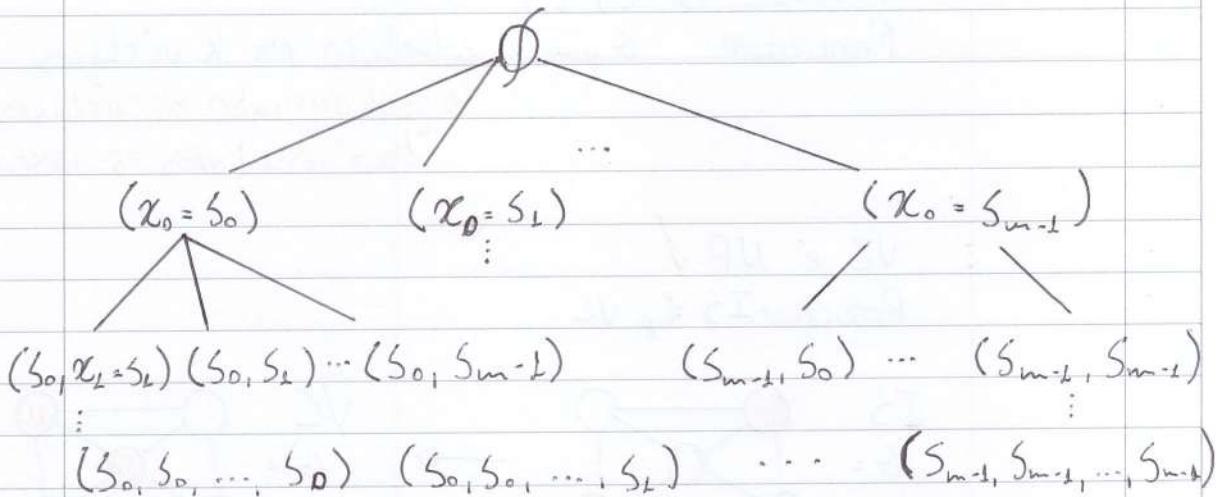
## Backtracking

Percorso sistemático do espaço de soluções  $S^u$

Cada possível "solução" codificada como um vetor  $X = (x_0, \dots, x_{n-1})$

Tal que  $x_i \in \{S_0, \dots, S_{m-1}\}$

$S^u$  organizado como uma árvore de soluções parciais



Folhas = soluções completas

Backtracking = percurso em profundidade da árvore  $S^u$ , interrompendo assim que chegar a um nó incompatível

Exemplo

Subset Sum

Entrada:  $A = \{a_0, \dots, a_{m-1}\}$  conjunto de naturais  
 $S \in \mathbb{N}$

Propriedade:  $\exists B \subseteq A \mid \sum b \in B = S$

Exemplo

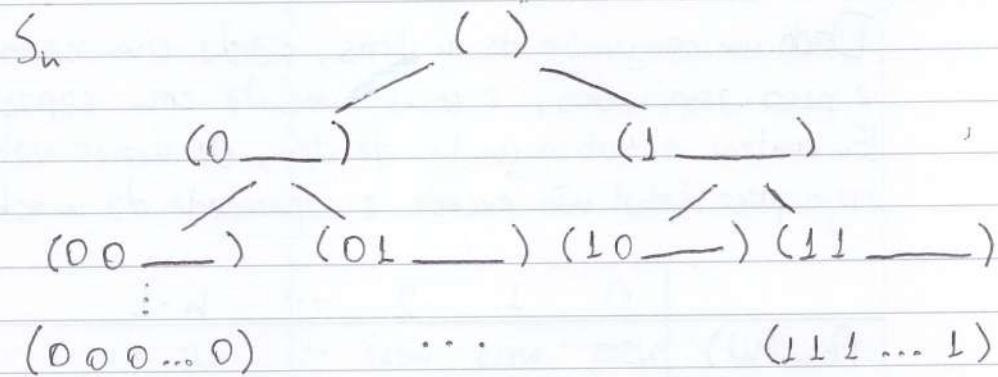
$$A = \{1, 2, 4, 6, 8, 10\}$$

$$S = 18 \quad \text{e.g. } B = \{8, 10\}$$

Subset Sum é NP-C

$$B = (0, 0, 0, 1, 1)$$

Codificação:  $B = (b_0, \dots, b_{m-1}) \in \{0, 1\}^m \mid b_j = 1 \Leftrightarrow a_j \in B$



Algoritmo subset-sum

Entrada:  $A = (a_0, \dots, a_{m-1}) \in \mathbb{N}^m$ ,  $S \in \mathbb{N}$

Saída:  $B = (b_0, \dots, b_{m-1}) \in \{0, 1\}^m | \sum_{i=0}^{m-1} b_i a_i = S \text{ ou } \perp \text{ se } \nexists B$

início

retorna ssbacktrack( $A, (0, 0, \dots, 0), S, 0, 0$ )

fim

Algoritmo ssbacktrack

Entrada:  $A = (a_0, \dots, a_{m-1})$

$B = (b_0, \dots, b_{m-1})$

$S$

$P = \text{soma parcial}$

$i = \text{número de elementos já considerados}$

Saída: solução final ou  $\perp$

início

se  $P = S$  então

retorna  $B$

senão-se  $P > S$  então

retorna  $\perp$

senão-se  $P + \sum_{j=i}^{m-1} a_j < S$  então

retorna  $\perp$

senão

para  $\theta \in \{0, 1\}$ , o faça

$B[i] \leftarrow \theta$

$X \leftarrow \text{ssbacktracking}(A, B, S, P + (B[i] \cdot a[i]), i+1)$

se  $X \neq \perp$  então

retorna  $X$

... fim-se

## Problema da Mochila (Knapsack)

Dado um conjunto de  $n$  itens, cada um com valor e peso associados, e uma mochila com capacidade  $K$  (peso). Encontrar o subconjunto de itens de maior valor total, cujo peso total não excede a capacidade da mochila.

	0	1	2	...	$n-1$
Peso ( $W$ )	$w[0]$	$w[1]$	$w[2]$	...	$w[n-1]$
Valor ( $V$ )	$v[0]$	$v[1]$	$v[2]$	...	$v[n-1]$

$$\max \left\{ \sum_{i=0}^{n-1} b_i V_i \mid \sum_{i=0}^{n-1} b_i w_i \leq K \right\}$$

$b_i = \begin{cases} 1, & \text{se item } i \text{ é escolhido} \\ 0, & \text{caso contrário} \end{cases}$

Teorema

O-1 Knapsack é NP-Completo

Prova

Estratégia

O-1 Knapsack



Knapsack-decisão



Subset-Sum



NP-C

Knapsack-decisão

Entrada:  $V = (v_0, \dots, v_{n-1})$

$W = (w_0, \dots, w_{n-1})$

$K$

$S$

Propriedade:  $\exists$  subconjunto de itens cuja  $\sum w \leq K$  e  $\sum v \geq S$

Exemplo

i	0	1	2	3
w	4	3	1	2
..	..	..	..	..

Knapsack-decisão ( $V, W, K, S=100$ ) = True  
 Knapsack-decisão ( $V, W, K, S=200$ ) = False

+ Se conseguirmos resolver problema de otimização encontrando valor total ótimo  $S^*$ , então  $\forall S \leq S^*$  a resposta do problema decisão é True e  $\forall S > S^*$ , a resposta é False  
 $\therefore$  Knapsack - Decisão  $\leq_p$  Knapsack

+

- Knapsack - Decisão é NP

- Subset-Sum  $\leq_p$  Knapsack - Decisão

$$\langle A = (z_0, \dots, z_{n-1}), B \rangle \xrightarrow{f} \langle V = A(z_0, \dots, z_{n-1}), W = A(z_0, \dots, z_{n-1}), K = B, S = B \rangle$$

tal que

$A$  tem subconjunto  
cuja soma é  $B$

$\exists$  subconjunto de itens tal  
que  $\sum w \leq K$  e  $\sum v \geq S$

## Programação Dinâmica

- 1 - Identificar subproblemas
- 2 - Resolvê-los do menor para o maior (bottom-up)
  - Armazena soluções dos subproblemas numa tabela
  - Solução para problema (maior) obtida a partir de soluções de subproblemas consultadas na tabela
  - + Não usa recursão

## PD para O-L Knapsack

Definição  $S(m, K) \stackrel{\text{def}}{=} \text{Valor máximo transportável com capacidade } K, \text{ escolhendo dentre os } m \text{ primeiros itens}$

Solução do problema original  
 $S(n, K)$

Base

$$S(0, K) = 0$$

$$S(m, 0) = 0$$

Judugão

$$S(m, k) \quad m, k > 0$$

2 casos

1 - Solução ótima não inclui item ( $m-1$ )

$$S(m, k) = S(m-1, k)$$

2 - Solução ótima inclui item ( $m-1$ )

$$S(m, k) = V[m-1] + S(m-1, k - W[m-1])$$

$L+2$

$$S(m, k) = \max \left\{ \begin{array}{l} S(m-1, k), \\ S(m-1, k - W[m-1]) \end{array} \right\} + V[m-1]$$

Exemplo

$m \backslash k$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	0	120	120
*2	0	0	0	80	120	120
3	0	20	20	80	120	140
*4	0	20	70	90	120	150

i	0	*1	2	*3	K=5
w	4	3	1	2	
v	120	80	20	70	

Algoritmo - 01-Knapsack

Entrada:  $V = (V_0, \dots, V_{n-1})$

$W = (W_0, \dots, W_{n-1})$

K

Saida:  $S(n, K)$

início

$$D^{(n+1)(K+1)} = \begin{bmatrix} d_{00} & & \\ & \ddots & \\ & & d_{nK} \end{bmatrix} \leftarrow 0_{(n+1)(K+1)}$$

para  $m \leftarrow 1, \dots, n$  faça

para  $k \leftarrow 1, \dots, K$  faça

se  $W[m-1] \leq k$  então

$$D[m, k] \leftarrow \max \left\{ D[m-1, k], V[m-1] + D[m-1, k - W[m-1]] \right\}$$

senão

$$D[m, k] \leftarrow D[m-1, k]$$

fim

fim-se

fim-faça

fim-faça

retorna  $D[n, K]$

fim

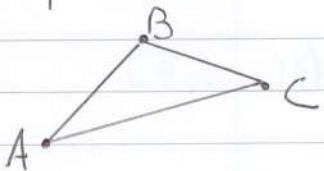
$$T(n, K) = O(nK)$$

Pseudo-polynomial

## Travelling Salesperson (TSP)

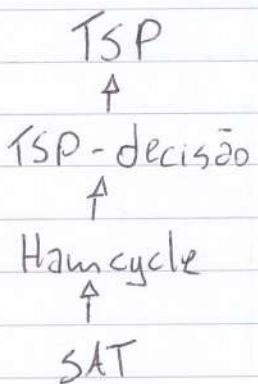
TSP - Grafo dirigido  $G = (V, E, w)$  completo com pesos  
 $w: E \rightarrow \mathbb{R}_+$

Objetivo - Encontrar circuito (começa e termina no mesmo vértice e passa exatamente uma vez por cada vértice) de comprimento mínimo



Teorema - TSP é NP-hard

Provô



Hamcycle

Entrada: Grafo dirigido  $G = (V, E)$

Propriedade:  $G$  possui circuito hamiltoniano

$\Leftrightarrow$  começa e termina no mesmo vértice e passa 1 vez por cada vértice

Lema: Hamcycle é NP-completo

NP ✓

NP-hard

Redução SAT  $\leq_p$  Hamcycle

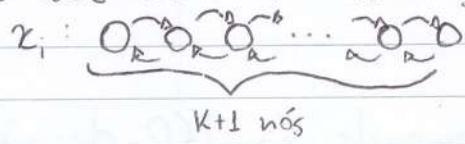
$\varphi \xrightarrow{f} G = (V, E)$

Tal que  $\varphi$  é satisfatível  $\Leftrightarrow G$  é hamiltoniano

Exemplo

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

- Suponha  $\varphi$  possui  $n$  variáveis e  $K$  cláusulas
- Para cada variável, construir gadget

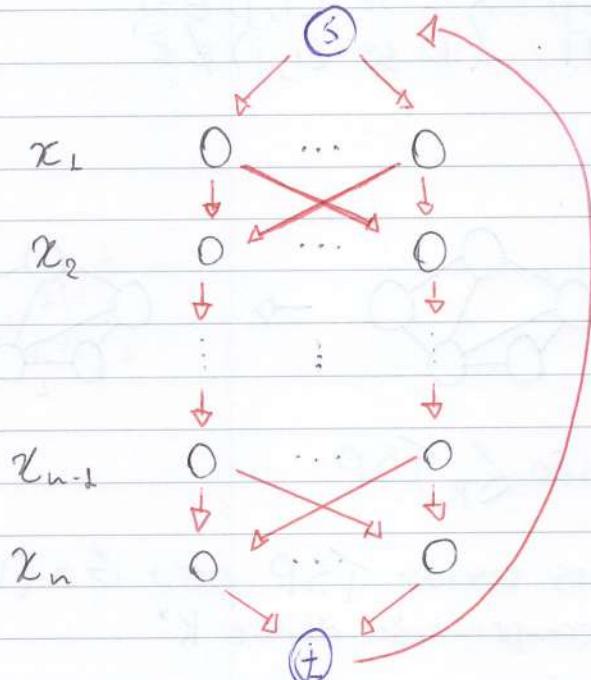


- Para cada cláusula, criar nó  $C_j$
- Conectar as variáveis às cláusulas

$$x_i \in C_j: O \overset{\curvearrowright}{\sim} \dots \overset{\curvearrowright}{\sim} O \overset{\curvearrowright}{\sim} O \overset{\curvearrowright}{\sim} \dots \overset{\curvearrowright}{\sim} O$$

$$\neg x_i \in C_j: O \overset{\curvearrowright}{\sim} \dots \overset{\curvearrowright}{\sim} O \overset{\curvearrowright}{\sim} O \overset{\curvearrowright}{\sim} \dots \overset{\curvearrowright}{\sim} O$$

- Conectar os gadgets variáveis



Afirmção:  $G$  tem circuito hamiltoniano  $\Leftrightarrow \varphi$  é satisfatível

## TSP-Decisão

Entrada:  $G = (V, E, w)$   
 $K \in \mathbb{R}_+$

Propriedade:  $G$  possui um circuito de comprimento  $\leq K$

NP ✓

NP-Difícil

Redução Hamcycle  $\leq_p$  TSP-decisão

$G' = (V', E')$   $\xrightarrow{f} G = (V, E, w); K$

tal que  $G'$  tem circuito hamiltoniano se, e somente se,

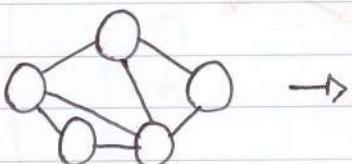
$G$  tem constante de comprimento  $\leq K$

defina  $G = (V, E, w)$  tal que

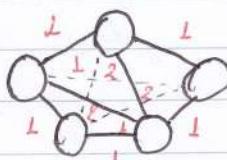
$$\begin{cases} V = V' \\ E = V \times V \\ w(i, j) = \begin{cases} 1 & \text{se } (i, j) \in E \\ 2 & \text{se } (i, j) \notin E \end{cases} \end{cases}$$

$$K = |V|$$

$$n=5$$



→

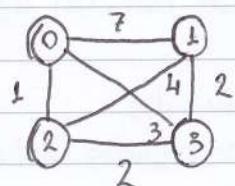


TSP-Decisão  $\leq_p$  TSP

Se soubermos resolver TSP para  $G = (V, E)$  e encontrarmos o comprimento ótimo  $K^*$

Então  $\begin{cases} \forall_k < K^* \rightarrow \text{TSP-decisão } (G, k) = \text{False} \\ \forall_k \geq K^* \rightarrow \text{TSP-decisão } (G, k) = \text{True} \end{cases}$

Exemplo



Algoritmo Guloso

Sempre vai para cidade (nó) mais próxima

Solução gulosa

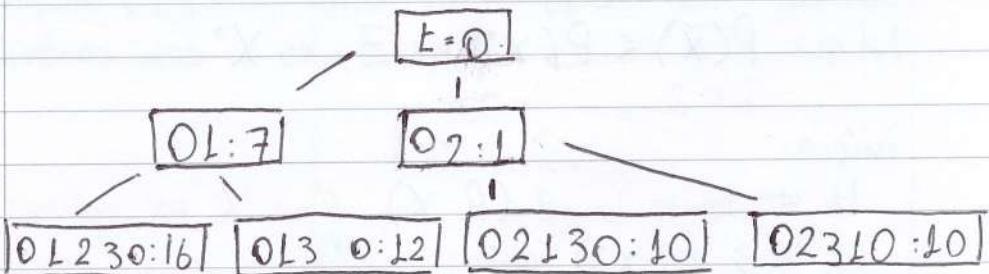
$$t = 0 - 2 - 3 - 1 - 0$$

$$c(t) = 1 + 2 + 3 + 4 = 10$$

Outra solução

$$t' = 0 - 3 - 1 - 2 - 0$$

$$c(t') = 3 + 2 + 4 + 1 = 10$$



## Branch & Bound

- + Extensão do backtracking para otimização
- Interruma o ramo assim que solução parcial não pode melhorar a solução já conhecida
  - Guardar melhor solução até o momento
  - Estima cota superior e inferior para todas extensões da solução parcial corrente

Algoritmo branch-bound

Entrada : P problema de minimização

$X = (x_0, \dots, x_i)$  solução parcial

$S = (s_0, \dots, s_{n-1})$  possíveis valores para  $x_j$

$X^*$  = solução ótima corrente

Saída :  $\tilde{X} = (x_0, \dots, x_i, x_{i+1}, \dots, x_{n-1})$  solução total  
tal que  $P(\tilde{X}) < P(X^*)$  se  $\exists$ , ou  $X^*$  caso contrário

início

$lb \leftarrow lower\_bound(P, X)$  // se  $X$  for inviável, então  $lb = \infty$

se  $lb \geq P(X^*)$  então

retorna  $X^*$

senão

se  $i = n-1$  então

se  $P(X) < P(X^*)$  então

retorna  $X$

senão

retorna  $X^*$

Fim-se

senão

$R \leftarrow X^*$

para  $\vartheta \in S$

$R \leftarrow branch\_bound(P, \langle x_0, \dots, x_i, x_{i+1} = \vartheta \rangle, S, R)$

Fim-faça

retorna  $R$

Fim-se

Fim-se

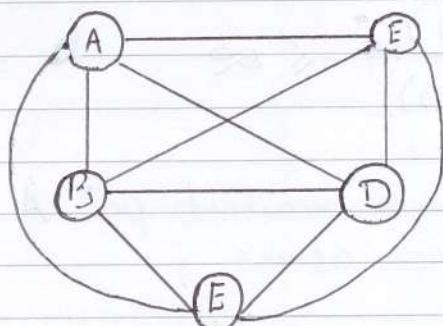
Fim

## TSP B&B (grafo não-dirigido)

+ Cada circuito  $C$  tem uma aresta entrando e uma saíndo de cada vértice

$$|C| = \sum_{v \in V} \frac{e^v + e''_v}{2}, \text{ onde } e^v \text{ e } e''_v \text{ correspondem aos custos das duas menores arestas adjacentes a } v$$

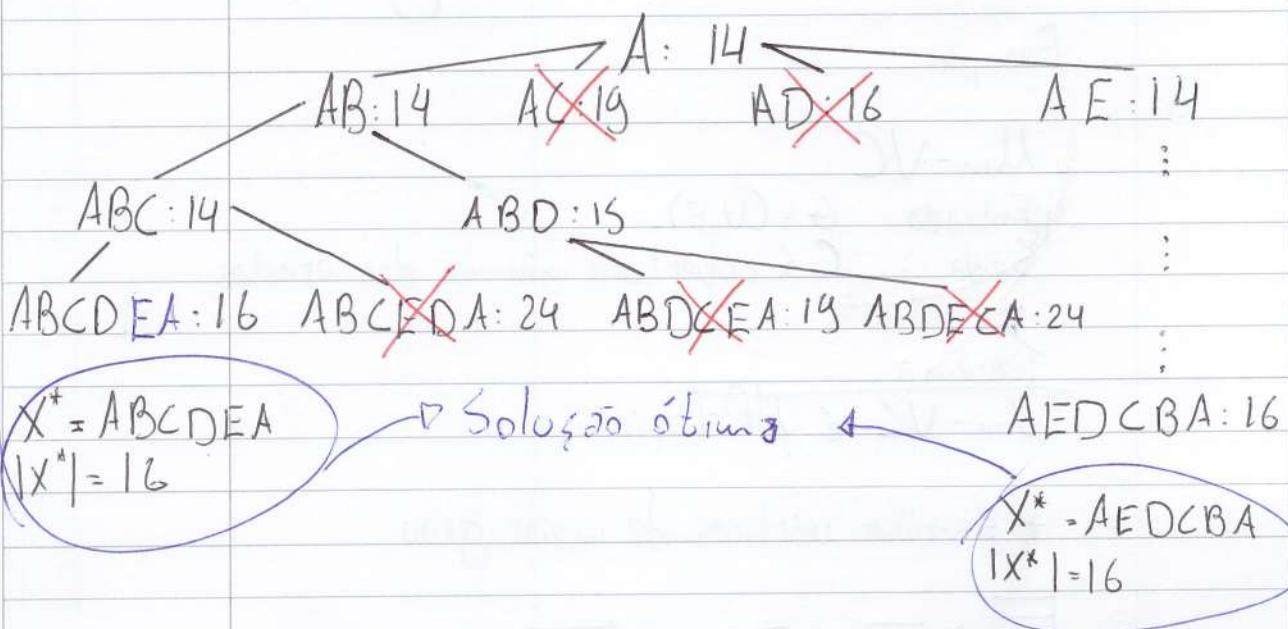
Exemplo



	A	B	C	D	E
A	-	1	8	5	3
B	1	-	2	4	6
C	8	2	-	3	9
D	5	4	3	-	7
E	3	6	9	7	-

Cota inferior:  $\frac{28}{2} = 14$

+ Construir solução progressivamente 1 vértice por vez  
+ A cota inferior é estimada de maneira similar a essa, fixando-se as arestas já incluídas na solução parcial



# Algoritmos de Aproximação

+ Soluções subótimas com "garantia" de desempenho

Definição

Seja um problema de minimização

$$P(I) = \begin{cases} \min f(x) & \text{função objetivo} \\ \text{sujeto a } Q(I, X) & \text{restrições} \end{cases}$$

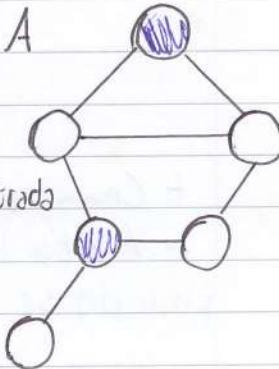
Dizemos que um algoritmo  $A$  para  $P$  é uma  $\alpha$ -aproximação se

$$\max_I \frac{f(A(I))}{f(OPT(I))} \leq \alpha$$

onde  $\begin{cases} A(I) & \text{solução encontrada por } A \\ OPT(I) & \text{solução ótima} \end{cases}$

$\alpha$  é dito fator de aproximação de  $A$

+  $\alpha$  pode ser  $\begin{cases} \text{constante} \\ \text{função do tamanho da entrada} \end{cases}$



Algoritmo de aproximação guloso

Exemplo:

Min-VC

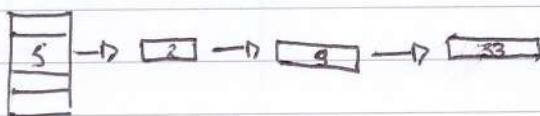
Entrada:  $G = (V, E)$

Saída:  $C \leq$  cobertura mínima das arestas

Teorema

Min-VC é NP-Difícil

+ Escolher vértices de maior grau



## Algoritmo greed-vc

Entrada:  $G = (V, E)$

Saída:  $C \subseteq V$  cobertura

início

$C \leftarrow \emptyset$

$G' = (V', E') \leftarrow G$

enquanto  $E' \neq \emptyset$  faça

$v \leftarrow$  vértice de maior grau de  $G'$

$C \leftarrow C \cup \{v\}$

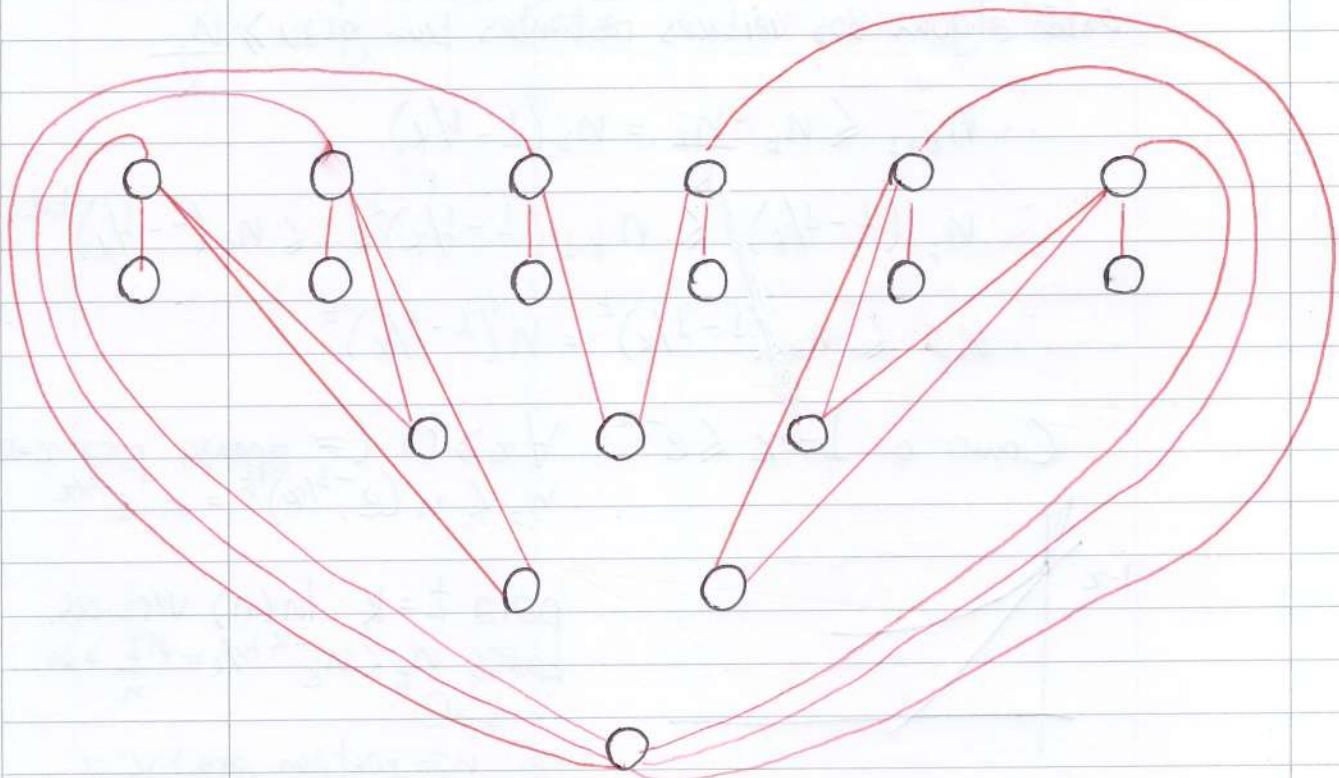
$V' \leftarrow V' - \{v\}$

$E' \leftarrow E' - \{(v, w) \in E'\}$

fim-faça

retorna  $C$

fim



### Teorema

Seja  $G$  grafo com  $n$  arestas e cobertura mínima de  $K$  vértices, então a cobertura produzida por Greedy-VC possui  $\leq K \ln(n)$  vértices

### Algoritmo Greedy-VC

- Enquanto houver arestas não cobertas, escolha o vértice de maior grau e retire-o do grafo, junto com suas arestas

### Teorema

Greedy-VC é uma  $\ln(n)$ -aproximação

Seja  $n_t \stackrel{\text{def}}{=} \text{número de arestas não-cobertas após } t \text{ escolhas}$   
em particular  $n^0 = n$

Seja  $K$  tamanho da cobertura ótima

Como essas  $n_t$  arestas são cobertas por  $K$  vértices,  
então algum dos vértices restantes tem grau  $\geq \frac{n_t}{K}$

$$\therefore n_{t+1} \leq n_t - \frac{n_t}{K} = n_t \left(1 - \frac{1}{K}\right)$$

$$\therefore n_t \left(1 - \frac{1}{K}\right) \leq n_{t-1} \left(1 - \frac{1}{K}\right)^2 \leq \dots \leq n_0 \left(1 - \frac{1}{K}\right)^{t+1}$$

$$\therefore n_t \leq n_0 \left(1 - \frac{1}{K}\right)^t = n \left(1 - \frac{1}{K}\right)^t$$

Como o  $1-x \leq e^{-x} \quad \forall x > 0$  ( $=$  apenas para  $x=0$ )  
 $n_t \leq n \left(e^{-\frac{1}{K}}\right)^t = n e^{-\frac{t}{K}}$



para  $t = K \ln(n)$  vértices,  
temos  $n_t \leq n e^{-K \ln(n)/K} = n e^{-1} = n$   
 $\therefore t = 0$

i.e.: não restam arestas a serem cobertas

+ Escolher uma aresta ao invés de vértices

## Algoritmo greedy-vc 2

Entrada:  $G = (V, E)$

Saída:  $C \subseteq V$  cobertura

início

$$C \leftarrow \emptyset$$

$$G' = (V', E') \leftarrow G$$

enquanto  $E' \neq \emptyset$  faça

$(u, v) \leftarrow$  aresta qualquer de  $E'$

$$C \leftarrow \{u, v\}$$

$$V' \leftarrow V' - \{u, v\}$$

$$E' \leftarrow E' - (\{(u, w) \in E' \mid u \in \{v, w\} \in E'\})$$

fim-faça

retorna  $C$

fim

Teorema

Greedy-VC2 é uma 2-aproximação

Prova

Seja  $K$  o tamanho da cobertura ótima e que o número de arestas escolhidas pelo algoritmo

Como cada aresta escolhida está coberto por pelo menos 1 vértice diferente da cobertura ótima (as arestas escolhidas são todas disjuntas então  $q \leq K \therefore |C| = 2q \leq 2K$ )

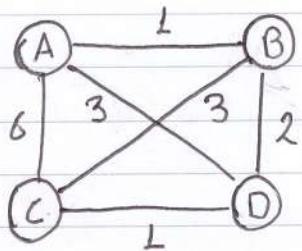
Aproximação gulosa para TSP

Algoritmo greedy-tsp

· Inicia num vértice qualquer (e.g.:  $v_0$ )

· A cada passo, vai para o nó mais próximo ainda não visitado

Exemplo



Solução gulosa

A - B - C - D - A

$$|Sg| = 10$$

Solução ótima

A - B - D - C - A

$$|S^*| = 8$$

$$\text{Fator de aproximação} \frac{|Sg|}{|S^*|} = \frac{10}{8} = 1,25$$

+ Esse fator não é constante nem sequer limitado pela entrada

Por exemplo: fazendo o custo  $\omega$  da aresta  $AD \rightarrow \infty$

$$\text{Fator de aproximação} = \frac{1+2+\infty}{8} = \infty$$

Teorema

Se  $P \neq NP$  então  $\nexists$  algoritmo polinomial de  $\alpha$ -aproximação para TSP com  $\alpha$  constante

Prova

Suponha que existe algoritmo  $A$  de  $\alpha$ -aproximação para TSP ( $\alpha$  constante)

+ Usar  $A$  para resolver Hamilton cycle  $\rightarrow 1/\alpha$

Seja  $G = (V, E)$ , defina  $G' = (V', E')$  tal que

$$V' = V$$

$$E' = \{(u, v, l) \mid (u, v) \in E \} \cup \{(u_0, u_{i-1}, \alpha_{i-1}) \mid (u_i, u) \in E\}$$

Se  $G$  possui circuito hamiltoniano,  $C$ , então  $|C|$ , então  $|C'| = n$  e esse é o custo ótimo para TSP  $C'$

Por outro lado, por hipótese  $A$  produz um circuito com custo  $\leq \alpha C^*$

Então  $A$  produz circuito com comprimento  $\leq \alpha n$ , se e somente se, esse circuito possui apenas arestas constantes comprimento 1, se e somente se, esse circuito é circuito hamiltoniano em  $G$ .

Aproximação para TSP via MST Euclidiano

Algoritmo MST-TSP ( $G = (V, E, w)$ )

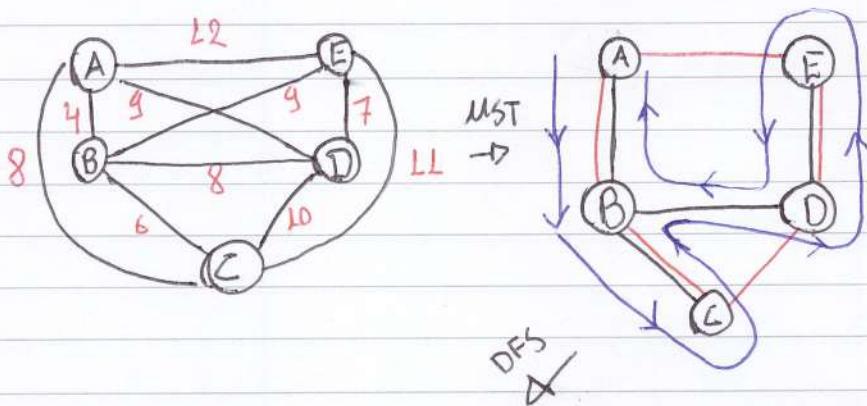
$\rightarrow$  distância simétrica  
e desigualdade triangular

①  $T \leftarrow \text{MST}(G)$

②  $C \leftarrow \text{DFS}(T)$

③  $\tilde{C} \leftarrow \text{Remove as repetições de } C$

Exemplo



$$C: A-B-C-B-D-E-D-A$$

$$\tilde{C}: A-B-C \quad D-E \quad A$$

$$|\tilde{C}| = 39$$

Teorema

MST-TSP é uma 2-aproximação

P prova

$$|C| = 2|T| \quad (T \text{ é MST})$$

Se  $S^*$  é solução ótima do TSP, então retirando-lhe  
qualquer aresta obtemos uma ST cujo  $\epsilon \geq |T|$

$$\therefore |T| \leq |S^*|$$

$$\therefore |C| = 2|T| \leq 2|S^*|$$

Portanto pelo desigualdade triangular,  $|C| \leq |C| \leq 2|S^*|$