

Apontamentos de Algoritmos

1-Aspectos introdutórios

2-Funções

3-Algoritmos de ordenação e de pesquisa

4-Pesquisa linear e pesquisa binária

5-Algoritmos sobre Grafos e Digrafos

6-Extensões: noções básicas sobre heurísticas.

Algoritmo (Definição)

Procedimento computacional bem definido que aceita uma dada entrada e produz uma dada saída, i.e., é uma ferramenta para resolver um problema computacional bem definido, como por exemplo:

- . Calcular a média de um conjunto de valores
- . Ordenar uma sequência de valores
- . Determinar os caminhos mais curtos em grafos orientados
- . etc.

Exemplo: mudar maiúsculas para minúsculas

Entrada: vetor de caracteres com o texto

Objetivo: mudar todas as letras maiúsculas do texto de entrada para as respectivas letras minúsculas

Saída: vetor de entrada alterado

Algoritmo (Características)

- _ **Rigoroso** – cada instrução do algoritmo deve expressar exta e rigorosamente o que deve ser feito
- _ **Eficaz** – cada instrução deve ser suficientemente básica e bem compreendida de modo a poder ser executada num intervalo de tempo finito, com uma quantidade de esforço finita
- _ **Deve terminar** – deve levar a uma situação em que o objetivo tenha sido atingido e não existam mais instruções para ser executadas

Programa - Um algoritmo escrito de modo a poder ser executado por um computador.

Linguagem de programação – usada para escrever programas de computador

Sintaxe – frases da linguagem (determina qual a constituição das frases que podem ser fornecidas ao computador)

Semântica – significado associado às frases (determina o que o computador vai fazer ao seguir as indicações apresentadas em cada uma dessas frases)

1.1 O que é o Python?

- ① É uma linguagem de programação;
- ② É uma VHLL (Very High Level Language);
- ③ É interpretada;
- ④ Criada por Guido van Rossum (Universidade de Amesterdam) em 1991;
- ⑤ Nas suas próprias palavras: Python is an interpreted, interactive, object-oriented programming language. Python combines remarkable power with very clear syntax. It has modules, classes, exceptions, very high level dynamic data types, and dynamic typing;
- ⑥ Curiosidade: O nome Python vem do grupo de humor inglês Monty Python.

1.2 Como usar o Python

Em ambientes Linux, Mac OS, Windows. PyCharm: contém editor para a linguagem

<http://www.python.org>; <https://www.jetbrains.com/pycharm/>

Anaconda

Anaconda is a free and open-source distribution of the Python and R programming languages for scientific computing, that aims to simplify package management and deployment. Includes data-science packages suitable for Windows, Linux, and MacOS. Could use Jupyter Notebook or Spyder.

<https://www.anaconda.com/distribution/>

1.3 A função *print*

É muito comum, ao apresentar uma nova linguagem, começar com um exemplo simples que mostra as palavras Hello World.

Hello World

```
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct 5 2020, 15:34:40) [MSC v.1927 64 bit (AMD64)] on win32 Type "help", "copyright", "credits" or "license" for more information. >>> print("Hello, World!")  
Hello, World!
```

Hello "World"

```
>>> print('Hello, "World"!')  
Hello, "World"!
```

Print ('Hello "World"!')

```
>>> Print('Hello, "World"!')  
Traceback (most recent call last): File <stdin>, line 1, in  
<module> NameError: name 'Print' is not defined
```

1.4 Operações aritméticas básicas, constantes e comentários

Operações aritméticas básicas

2 + 3

```
>>> 2+3  
5
```

2 × 3

```
>>> 2 * 3  
6
```

2 ÷ 3

```
>>> 2/3  
0.6666666666666666
```

2 ÷ 0

```
>>> 2/0
Traceback (most recent call last): File <<stdin>>, line 1, in <module>
ZeroDivisionError: division by zero
```

2 – 3

```
>>> 2-3
-1
```

Divisão Inteira

```
>>>10//3
3
```

Resto da divisão

```
>>> 10% 2
0
```

10%3

```
>>> 10% 3
1
```

$\sqrt{4}$

```
>>> 4**0.5
2
```

$\sqrt{9}$

```
>>> import math
>>> math.sqrt(9)
3.0
```

ESA01

Expressões numéricas

2 + 3 * 4

```
>>>2+3*4
14
```

7 + 3 * 6 - 4 ** 2

```
>>> 7 + 3 * 6 - 4 ** 2
9
```

(3 + 4) * 2

```
>>> (3 + 4) * 2
14
```

(9/3) **(5 – 3)

```
>>> (9/3)**(5-3)
9
```

Notação científica

10e3

```
>>> 10e3  
10000.0
```

1e-5

```
>>> 1e-5  
1e-5
```

10000000e-5

```
>>> 10000000e-5  
100.0
```

Números decimais

0.1

```
>>> 0.1  
0.1
```

0.1 :valor real da máquina

```
>>> format(0.1, '.50f')  
'0.100000000000000055511512312578270211815834045410'
```

0.1 + 0.7

```
>>> 0.1+0.7  
0.7999999999999999
```

1.5 Operadores lógicos e de comparação

Operadores de comparação

<code>x == y</code>	Igualdade. Verdadeiro se $x = y$.
<code>x != y</code>	Diferença. Verdadeiro se $x \neq y$.
<code>x < y</code>	Verdadeiro se $x < y$.
<code>x > y</code>	Verdadeiro se $x > y$.
<code>x <= y</code>	Verdadeiro se $x \leq y$.
<code>x >= y</code>	Verdadeiro se $x \geq y$.

Operadores lógicos

Negação de x	$\sim x$.
Conjunção de x e y	$x \wedge y$.
Disjunção de x e y	$x \vee y$.

1.6 Atribuições e variáveis

Atribuições

```
>>> numero = 31  
>>> numero  
>>> 31
```

Atenção

Nomes de variáveis podem ter o tamanho que achar necessário e podem conter tanto letras como números, porém não podem começar com números. É possível usar letras maiúsculas, porém a convenção é utilizar somente letras minúsculas para nomes de variáveis.

Formatação

%s	str(objeto)
%d	Número inteiro
%x	Inteiro no formato hexa
%d	Número inteiro
%f	Float (muito usado com o número de casas: para duas casas decimais)

1.7 Estruturas de controle

Instrução if...else

```
x = -4;
if x < 0:
    x = -x
else:
    x=x
print("Valor absoluto: ", x)
```

Instrução if...elif...else

```
% Calcular o logaritmo do valor absoluto de x
import math
x=100
if x > 0:
    y = math.log10(x)
    print(y)
elif: x < 0:
    y = math.log10(-x)
    print(y)
else:
    print("Nao existe logaritmo de zero.")
```

ESA02

1.8 Estruturas de repetição

Ciclo for

Este ciclo executa um dado conjunto de instruções um número pre-determinado de vezes. O seguinte programa usa um ciclo `for` para calcular a potência de expoente n de um número real x .

```
n=4
x=2
potencia = 1
for i in range(0,n):
    potencia = potencia * x
print(potencia);

lista=[1,2,3,4,10]
for numero in lista:
    print(numero **2)

\begin{verbatim}
palavra = "escola"
for letra in palavra:
    print(letra):
```

Ciclo while

O ciclo `while` permite a execução repetida de um dado conjunto de instruções, mas agora baseado numa condição lógica. O seguinte excerto de programa multiplica x por 2 enquanto o valor obtido não ultrapassar 10000.

```
x = 1.0
while x <= 10000:
    x = x*2
    print(x)
```

break e continue

O `break`: É usado para sair de um loop, não importando o estado em que se encontra.

O `continue`: Funciona de maneira parecida com a do `break`, não encerrar o loop, ele faz com que todo o código que esteja abaixo (porém ainda dentro do loop) seja ignorado e avança para a próxima iteração.

```
while True:
    string_digitada = input("Digite uma palavra: ")
    if string_digitada.lower() == "algoritmos":
        print("Fim! Esta é a unidade curricular certa")
        break
```

1-Aspetos introdutórios

2-Funções

3-Algoritmos de ordenação e de pesquisa

4-Pesquisa linear e pesquisa binária

5- Algoritmos sobre Grafos e Digrafos

6-Extensões: noções básicas sobre heurísticas.

2.0 Revisões de funções

Matemática: Função (de uma variável) é um conjunto de pares ordenados que não contém dois pares distintos com o mesmo primeiro elemento

Domínio = conjunto dos primeiros elementos dos pares

Contradomínio = conjunto dos segundos elementos dos pares

Ex: $G_f=\{(1;3), (2;4), (3,5)\}$

Definição por extensão ou enumeração (lista de todos os seus elementos)

Definição por compreensão ou abstração (apresenta uma propriedade comum aos seus elementos $f(x)=x^2$ e $f(y)=y^2$ definem a mesma função em que x e y são variáveis mudas porque dão origem a uma expressão equivalente

Nome da função= f

Nome dos argumentos da função = x

Regra para calcular o valor da função para um dado valor de x : multiplicar esse valor por ele próprio

Expressão designatória versus designação

Funções em Python

Para além das funções intrínsecas em Python, podemos criar as nossas próprias funções. Assim, a seguinte função adiciona dois números:

```
def minhasoma(a,b):
    y = a + b
    return y
Para usar esta função para somar $4$ e $-5$ basta fazer

minhasoma(4,-5)
-1

def soma_valor(x):
    return 3+x

def tabuadas():
    for i in range(1, 11):
        for j in range(1, 11):
            print("{} * {} = {}".format(i, j, i * j))
```

Funções:continuação

```
def save(x,L=[]):
    L.append(x)
    print L

def union(*args):
    res = []
    for seq in args:
        for x in seq:
            if x not in res:
                res.append(x)
    return res
```

Desafio: Criar uma função que determine os zeros de um polinómio de grau dois.

2.1 Função recursiva

Definição

Uma função diz-se *recursiva* se na sua definição ela se invoca a si mesma.

Estrutura de uma função recursiva

Uma função recursiva é composta por:

- uma ou mais *condições recursivas*;
- uma ou mais *condições de paragem*, que impedem que a função entre em ciclo infinito.

A) Fatorial

Definição

Dado um número natural n , o seu factorial é definido por

$$n! = 1 \times 2 \times \dots \times (n-1) \times n$$

Além disso, convenciona-se que $0! = 1$.

Aplicação

Desta definição podemos constatar que

$$\begin{aligned} 1! &= 1 \\ 2! &= 1 \times 2 = 1! \times 2 \\ 3! &= 1 \times 2 \times 3 = 2! \times 3 \\ 4! &= 1 \times 2 \times 3 \times 4 = 3! \times 4 \\ 5! &= 1 \times 2 \times 3 \times 4 \times 5 = 4! \times 5 \\ &\vdots \end{aligned}$$

De um modo geral, $n! = (n-1)! \times n$.

Expressão da função

$$f(n) = \begin{cases} 1 & \text{se } n = 0 \\ f(n-1) \times n & \text{se } n > 0 \end{cases}$$

Fatorial recursiva

Ficheiro factorial11.py

```
def factorial(n)
    if (n==0):
        y=1
    else:
        y=factorial(n-1)*n
    return y
```

Nota: `factorial(3)`: fica em memória $3 * \text{factorial}(2)$; $2 * \text{factorial}(1)$; $\text{factorial}(1)$ é 1; retrocede e calcula o que ficou pendente.

Será possível construir uma versão *não recursiva* da função factorial?

Fatorial não recursiva

Ficheiro factorial2.py

```
def factorial2(n)
    y=1
    for i in range(1,n+1):
        y=y*i
    return y
```

Proposta

Testar o desempenho das duas funções. Para tal ver: import timeit

B) Sucessão de Fibonacci

Definição

A sucessão de Fibonacci é uma sequência de números inteiros em que os dois primeiros são 0 e 1 e cada um dos seguintes é igual à soma dos dois anteriores.

Construção da função de Fibonacci

Se $\text{fib}(n)$ é o n -ésimo termo da sucessão de Fibonacci, então

$$\text{fib}(n) = \begin{cases} 0 & \text{se } n = 1 \\ 1 & \text{se } n = 2 \\ \text{fib}(n - 1) + \text{fib}(n - 2) & \text{se } n > 2 \end{cases}$$

Desafio

Construir uma função recursiva `fib` que calcule o n -ésimo termo da sucessão de Fibonacci.

Será eficiente?

Cada chamada da função `fib` desencadeia **duas** chamadas da mesma função...

Desafio

Implementar uma versão não recursiva da função `fib`.

Fibonacci recursiva

Ficheiro fibonacci1.py

```
def fibonacci1(n)
    if (n==1):
        return 0
    elif (n==2):
        return 1
    else:
        return (fibonacci1(n-1)+fibonacci1(n-2))
```

Fibonacci não recursiva

Ficheiro fibonacci2.py

```
def fibonacci2(n)
    if n==1:
        y=0
    elif n==2
        y=1
    else
        a=0
        b=1
        for i in range(3,n+1):
            y=a+b
            a=b
            b=y
    return y
```

Desafio

Comparar o desempenho das funções fib e fib2.

C) Função McCarthy91

Definição

$$f(n) = \begin{cases} n - 10 & \text{se } n > 100 \\ f(f(n + 11)) & \text{se } n \leq 100 \end{cases}$$

Desafio I

Implementar esta função em Python.

McCarthy91 recursiva

Ficheiro MacCarthy91.py

```
def MacCarthy91(n):
    if n > 100:
        return n-10
    else:
        return (MacCarthy91(MacCarthy91(n+11)))
```

Desafio II

O que fará esta função?

Ficheiro MacCarthy91S.py

```
def MacCarthy91S(n):
    if n > 100:
        return n-10
    else:
        return 91
```

D) Função Ackermann

Definição

A função de Ackermann $A(m, n)$ é dada por

$$A(m, n) = \begin{cases} n + 1 & \text{se } m = 0 \\ A(m - 1, 1) & \text{se } m > 0 \text{ e } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{se } m > 0 \text{ e } n > 0 \end{cases}$$

Desafio

Implementar esta função em Python.

Perigo!

Esta função tem um crescimento **explosivo** e requerimentos de memória **devoradores...**

Ackermann recursiva

Ficheiro Ackermann.py

```
def Ackermann(m, n):
    if (m == 0):
        return n + 1
    elif (m > 0 and n == 0):
        return Ackermann(m - 1, 1)
    elif (m > 0 and n > 0):
        return Ackermann(m - 1, Ackermann(m, n-1))
```

E) Máximo divisor comum (m.d.c.)

Definição

Como é sabido, o *máximo divisor comum* (ou mdc) de dois números é o maior número que é simultaneamente divisor de ambos. Por exemplo, $\text{mdc}(12, 18) = 6$.

Função recursiva $\text{mdc}(x, y)$

$$\text{mdc}(x, y) = \begin{cases} x & \text{se } y = 0 \text{ e } x > 0 \\ \text{mdc}(y, \text{resto da divisão de } x \text{ por } y) & \text{se } x > 0 \text{ e } y > 0 \end{cases}$$

Desafio

Implementar a função mdc em Python.

Nota

O resto da divisão de x por y pode ser calculado em Python como?

m.d.c. recursiva

Ficheiro mdcRecursivo.py

```
def mdcRecursivo(x, y):
    if (y == 0 and x>0):
        return x
    elif (x>0 and y>0):
        return mdcRecursivo(y, (x%y))
    else :
        print("Supõe-se que x e y são positivos")
```

m.d.c. não recursiva

Ficheiro mdcNaoRecursivo.py

```
def mdcNaoRecursivo(x,y):  
    if (x<0 or y>0):  
        print("Supõe-se que x e y são positivos")  
    else:  
        while y != 0:  
            r = x % y  
            x = y  
            y = r  
    return x
```

F) Cálculo do número de dígitos de um número inteiro positivo

Definição

$$\text{ndigitos}(n) = \begin{cases} 1 & \text{se } n < 10 \\ \text{ndigitos}(\lfloor \frac{n}{10} \rfloor) + 1 & \text{caso contrário} \end{cases}$$

Desafio

Implementar a função em Python.

Número de dígitos recursiva

Ficheiro ndigitosRecursivo.py

```
def ndigitosRecursivo(n):  
    if n < 10:  
        return 1  
    else:  
        return ndigitosRecursivo(n/10) + 1
```

Número de dígitos não recursiva

Ficheiro ndigitosNaoRecursivo.py

```
def ndigitosNaoRecursivo(n):  
    if n < 10:  
        return 1  
    else:  
        nd=0  
        while n!=0:  
            nd+=1  
            n=n//10  
    return nd
```

ESA03

ESA04

1-Aspetos introdutórios

2-Funções

3-Algoritmos de ordenação e de pesquisa

4-Pesquisa linear e pesquisa binária

5-Algoritmos sobre Grafos e Digrafos

6-Extensões: noções básicas sobre heurísticas.

(https://www.youtube.com/watch?v=0IAPZzGSbME&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=1)

(https://www.youtube.com/watch?v=-JTq1BFBwmo&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=2)

(https://www.youtube.com/watch?v=FbYzBWdhMb0&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=3)

(https://www.youtube.com/watch?v=xGYsEqe9Vl0&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=4)

(https://www.youtube.com/watch?v=1U3Uwct45IY&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=5)

(https://www.youtube.com/watch?v=9TlHvipP5yA&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=6)

(https://www.youtube.com/watch?v=9SgLBjXqwd4&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=7)

(https://www.youtube.com/watch?v=p1EnSvS3urU&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=8)

(https://www.youtube.com/watch?v=w7t4_JUUTeg&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=9)

(https://www.youtube.com/watch?v=5v-tKX2uRAk&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=10)

(https://www.youtube.com/watch?v=A03oI0znAoc&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=11)

(https://www.youtube.com/watch?v=Nd0XDY-jVHs&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=12)

(https://www.youtube.com/watch?v=NI4OKSvGAgM&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=13)

(https://www.youtube.com/watch?v=mwN18xfwNhk&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=14)

(https://www.youtube.com/watch?v=WIBBTSL0ZRc&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=15)

(https://www.youtube.com/watch?v=lj3E24nnPjI&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=16)

(https://www.youtube.com/watch?v=wU6udHRIkcc&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=17)

(https://www.youtube.com/watch?v=2Rr2tW9zvRg&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=18)

(https://www.youtube.com/watch?v=4V30R3I1vLI&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=19)

(https://www.youtube.com/watch?v=IawM82BQ4II&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=20)

(https://www.youtube.com/watch?v=MhT7XmxhaCE&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=21)

(https://www.youtube.com/watch?v=JvcqtZk2mng&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=22)

(https://www.youtube.com/watch?v=CyknhZbfMqc&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=23)

(https://www.youtube.com/watch?v=8gt0D0IqU5w&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=24)

(https://www.youtube.com/watch?v=XcZw01FuH18&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=25)

[\(https://www.youtube.com/watch?v=1K9ebQJosvo&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=26\)](https://www.youtube.com/watch?v=1K9ebQJosvo&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=26)
[\(https://www.youtube.com/watch?v=OvnWkEj0S-s&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=27\)](https://www.youtube.com/watch?v=OvnWkEj0S-s&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=27)
[\(https://www.youtube.com/watch?v=kGcO-nAm9Vc&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=28\)](https://www.youtube.com/watch?v=kGcO-nAm9Vc&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=28)
[\(https://www.youtube.com/watch?v=9rVuyjxzwgM&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=29\)](https://www.youtube.com/watch?v=9rVuyjxzwgM&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=29)

3.1 Análise de algoritmos

A) Medidas de Complexidade

Eficiência

- _ Escolha do algoritmo certo
- _ Mapear um algoritmo numa determinada classe de eficiência/complexidade

Medidas de eficiência e complexidade

- _ Tempo (tempo de execução)
- _ Espaço (memória ocupada)

Medidas de Complexidade

- Tempo necessário (execução)
- Espaço necessário

- Tanto o tempo como o espaço dependem do tamanho da entrada
- Tamanho da entrada
 - Fibonacci - o parâmetro n
 - Algoritmos de ordenação - o tamanho do array
 - Algoritmos que operam sobre grafos - número de vértices e número de arcos

Comparação de algoritmos diferentes

- _ Não basta correr o programa, já que este depende da linguagem, da máquina, do *input*, etc.; depende de como o programa escala nos dados maiores
- _ Alternativa: encontrar uma relação matemática que dependa do tamanho de entrada; isto é, determinar o número de “passos” básicos” (passo que toma um tempo constante, independentemente do tamanho do problema) que o algoritmo requer, em função do *input size* (tamanho) do programa.

_ Exemplos:

- . Num problema em que se pretende mudar os caracteres de maiúsculas para minúsculas, o número de caracteres da string de entrada é uma medida razoável
- . Nos algoritmos de ordenação, uma medida razoável é o número de elementos a ordenar
- . Num grafo/digrafo, as medidas utilizadas são o número de vértices e o número de arestas/arcos.

Crescimento de Funções

Padrões típicos do tempo de execução

1	Constante: quando o número de instruções do programa é executado um número limitado/constante de vezes
Log N	Logarítmicos: quando se divide continuamente o <i>input</i> em duas partes (<i>e.g.</i> , pesquisa binária, supondo que o vetor já foi lido e ordenado previamente)
N	Linear: quando existe algum processamento para cada elemento de entrada
NlogN	Quando um problema é resolvido através da resolução de um conjunto de sub-problemas e combinando, posteriormente, as suas soluções

Atenção: um programa só pode ter uma complexidade sublinear, se ignorar uma parte do *input*

Tempos de execução típicos

N^2	Quadrático: quando a dimensão da entrada duplica, o tempo aumenta 4 vezes
N^3	Cúbico: quando a dimensão da entrada duplica, o tempo aumenta 8 vezes
2^N	Exponencial: quando a dimensão da entrada duplica, o tempo aumenta para o quadrado

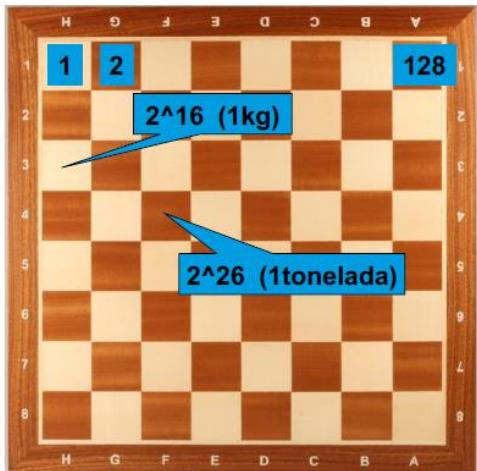
Comparação (1 passo = 1 segundo)

Ordem	$N=10$	10s
N^2	10^2	1,7 minutos
N^4	10^4	2,8 horas
N^5	10^5	1,1 dias
N^6	10^6	1,6 semanas
N^7	10^7	3,8 meses
N^8	10^8	3,1 anos
N^9	10^9	3,1 décadas
N^{10}	10^{10}	3,1 séculos

Comparação

$\log N$	$N^{1/2}$	N	$N \log N$	N^2
3	3	10	33	100
7	10	100	664	1000
10	32	1000	9966	1000000
13	100	10000	132877	100000000
20	1000	100000	19931569	10000000000000

Exemplo: Arroz e funções exponenciais



- 1 grão de arroz na 1ª casa
- 2 grãos na 2ª
- Continua dobrando

Número de átomos no
universo observável
 2^{82}

$$2^{64} = \\ 1.8446744e+19 \\ (3e+17 toneladas)$$

Conta a lenda que o xadrez foi inventado na Índia, há mais de 1500 anos. O rei ficou tão fascinado com a invenção e as infinitas variações de movimentos, que resolveu recompensar o inventor.

O rei perguntou: O que você quer de recompensa?

Inventor: Quero um grão de arroz para a primeira casa, dois grãos para a segunda casa, 4 para a terceira, e assim sucessivamente.

“Só isso?”, respondeu o rei.

Então, o rei pediu para os matemáticos do reino fazerem as contas: na primeira casa, 1 grão = 2^0 ; na segunda casa, 2 grãos = 2^1 ; na terceira casa, 4 grãos = 2^2 ; ...

Trata-se de uma progressão geométrica. Sendo o tabuleiro é um quadriculado de 8×8 , portanto tem 64 casas e, na vigésima primeira casa já tem mais de 1 milhão de grãos de arroz! A casa 41 corresponde a mais de 1 trilhão de grãos de arroz!

1 :	1
2 :	2
3 :	4
4 :	8
5 :	16
6 :	32
7 :	64
8 :	128
9 :	256
10 :	512
11 :	1024
12 :	2048
13 :	4096
14 :	8192
15 :	16384
16 :	32768
17 :	65536
18 :	131072
19 :	262144
20 :	524288
21 :	1048576
22 :	2097152
23 :	4194304
24 :	8388608
25 :	16777216
26 :	33554432
27 :	67108864
28 :	134217728
29 :	268435456
30 :	536870912
31 :	1073741824
32 :	2147483648
33 :	4294967296
34 :	8589934592
35 :	17179869184
36 :	34359738368
37 :	68719476736
38 :	137438953472
39 :	274877906944
40 :	549755813888
41 :	1099511627776
42 :	219902325552
43 :	4398046511104
44 :	8796093022208
45 :	17592186044416
46 :	35184372088832
47 :	70368744177664
48 :	140737488355328
49 :	281474976710656
50 :	562949953421312
51 :	1125899906842624
52 :	2251799813685248
53 :	4503599627370496
54 :	9007199254740992
55 :	18014398509481984
56 :	36028797018963968
57 :	72057594037927936
58 :	144115188075855872
59 :	288230376151711744
60 :	576460752303423488
61 :	1152921504606846976
62 :	2305843009213693952
63 :	4611686018427387904
64 :	9223372036854775808

A soma de todas das casas é igual a $2^{64}-1 = 1,8 \times 10^{19}$ grãos de arroz.

Curiosidades:

- . O maior avião cargueiro até o presente momento é o “Antonov AN-225 Mriya”, de fabricação Russa e o peso máximo de carga que ele suporta é de 250 tons: *O número de grãos de arroz equivale a 20143844528 aviões “AN-225” (mais de 20 biliões de “AN-225”).*
- . Sendo a produtividade média do arroz de 6,2tons de arroz por hectare, seriam necessários 812251795503 hectares para produzir esta quantidade de arroz, i.e., *seria necessária uma área equivalente a 2471 vezes o tamanho da Índia para que o rei pudesse cultivar o arroz necessário para pagar o inventor.*

Um último detalhe técnico: o Excel perde um pouco de precisão nas últimas casas decimais, pelo sistema de ponto flutuante; o Python utiliza uma espécie de *Big Int*, então é mais preciso para fazer este cálculo com rigor.

B) Notação assintótica

(https://www.youtube.com/watch?v=ddsP7NecEBk&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=78)

Que inputs devemos usar para calcular a complexidade de um algoritmo?

- _ Uso do **pior caso** como valor para a complexidade
 - . Representa um limite superior no tempo de execução (ocorre numerosas vezes)
 - . O valor médio é, muitas vezes, próximo do pior caso
 - . É, geralmente, mais fácil de calcular
 - . Evita surpresas
- _ Uso do **melhor caso** como valor para a complexidade
- _ Uso do **caso médio** como valor para a complexidade
 - . Importante em algoritmos probabilísticos
 - . É necessário saber a distribuição dos problemas
- Objectivo é caracterizar tempos de execução dos algoritmos para tamanhos arbitrários das entradas
- A notação assintótica permite estabelecer taxas de crescimento dos tempo de execução dos algoritmos em função dos tamanhos das entradas
- Constantes multiplicativas e aditivas tornam-se irrelevantes
 - Nota: Tempo de execução de cada instrução não é essencial para o comportamento assintótico de um algoritmo
- Símbolos da notação assintótica
 - Θ, O, Ω
 - o, ω

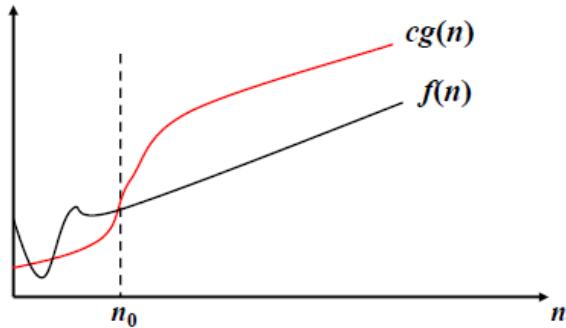
B1) Limite Assintótico Superior

Permite aferir a complexidade no **pior caso**

Notação **O** : Limite Assimptótico Superior

- $O(g) = \{f \mid \exists c, n_0 : \forall n \geq n_0 : f(n) \leq c \cdot g(n)\}$
- $f = O(g)$, significa $f \in O(g)$
- Analogia: $f \leq g$

- $O(g(n)) = \{f(n) : \text{existem constantes positivas } c \text{ e } n_0, \text{ tal que } 0 \leq f(n) \leq cg(n), \text{ para } n \geq n_0\}$
- $f(n) = O(g(n)), \text{ significa } f(n) \in O(g(n))$



Exemplo: Procura em vetor

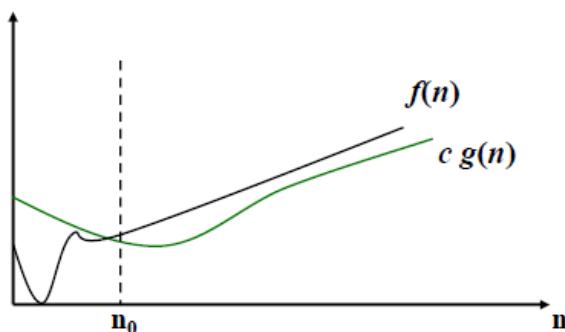
- _ **Solução:** analisar sequencialmente todo o vetor até encontrar o elemento
 - . No **pior caso**, o vetor é analisado até ao fim (i.e., a última posição do vetor é o elemento que procuramos)
 - . Complexidade: Limite assintótico superior = $O(N)$
- _ Outra forma de pensar: se $f(N)$ for a função que descreve o pior caso para o crescimento do tempo de execução do algoritmo, $f(N) = O(N)$, significa que $f(N)$ pertence ao conjunto de funções $O(N)$

B2) Limite Assintótico Inferior

Permite aferir a complexidade no **melhor caso**

Notação Ω : Limite Assimptótico Inferior

- $\Omega(g) = \{f \mid \exists c, n_0 : \forall n \geq n_0 : f(n) \geq c \cdot g(n)\}$
- $f = \Omega(g)$, significa $f \in \Omega(g)$
- Analogia: $f \geq g$
 - $\Omega(g(n)) = \{f(n) : \text{existem constantes positivas } c \text{ e } n_0, \text{ tal que } 0 \leq cg(n) \leq f(n), \text{ para } n \geq n_0\}$
 - $f(n) = \Omega(g(n)), \text{ significa } f(n) \in \Omega(g(n))$



Exemplo: Procura em vetor

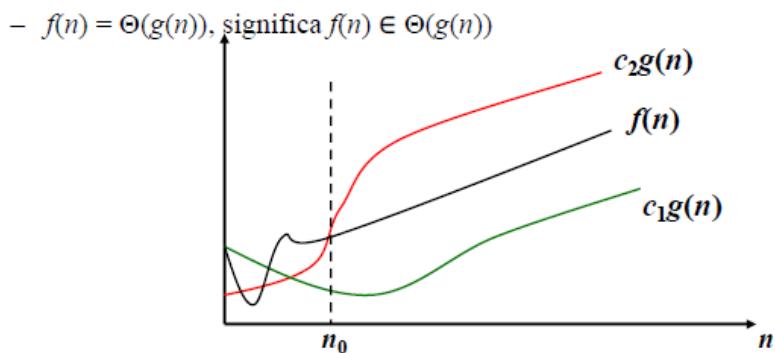
- _ **Solução:** analisar sequencialmente todo o vetor até encontrar o elemento
 - . No **melhor caso**, a primeira posição do vetor é o elemento que procuramos
 - . Complexidade constante no melhor caso: Limite assintótico inferior = $\Omega(1)$, i.e., o melhor caso pertence ao conjunto de funções descritas por $\Omega(1)$

B3) Limite Assintótico Apertado

Uma função $f(n)$ diz-se $\Theta(g(n))$ se e só se $f(n)$ for simultaneamente $O(g(n))$ e $\Omega(g(n))$

Notação Θ : Limite Assimptótico Apertado

- $\Theta(g) = \{f \mid \exists c_1, c_2, n_0 : \forall n \geq n_0 : c_1.g(n) \leq f(n) \leq c_2.g(n)\}$
- $f = \Theta(g)$, significa $f \in \Theta(g)$
- Analogia: $f = g$
 - $\Theta(g(n)) = \{f(n) : \text{existem constantes positivas } c_1, c_2, \text{ e } n_0, \text{ tal que } 0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n), \text{ para } n \in n_0\}$
 - $f(n) = \Theta(g(n))$, significa $f(n) \in \Theta(g(n))$



Exemplo: Vamos admitir que sabemos exatamente como o algoritmo se comporta:

- $f(n) = 0.1n^3 + 1000n^2 + 10^9$

- | | | |
|---------------------|-----------------------|-----------------|
| - É $O(n^3)$? | ... e $O(n^7)$? | É $\Theta(n^3)$ |
| - É $\Omega(n^3)$? | ... e $\Omega(n^2)$? | |

- Não é $O(n^\alpha)$ para $\alpha < 3$. Porquê?
- Não é $\Omega(n^\alpha)$ para $\alpha > 3$. Porquê?
- Não é $\Theta(n^\alpha)$ para $\alpha \neq 3$. Porquê?

Lema 1 - $\Theta \Leftrightarrow \Omega + O$

$$f = \Theta(g) \iff f = \Omega(g) \wedge f = O(g)$$

Lema 2 - Transitividade

- $f = O(g) \wedge g = O(h) \Rightarrow f = O(h)$
- $f = \Omega(g) \wedge g = \Omega(h) \Rightarrow f = \Omega(h)$
- $f = \Theta(g) \wedge g = \Theta(h) \Rightarrow f = \Theta(h)$

Lema 3 - $O - \Omega$ Correspondence

$$f = O(g) \iff g = \Omega(f)$$

Teorema Mestre

- Permite resolver recorrências da forma

$$T(n) = aT(n/b) + f(n) \quad , \quad a \geq 1 , \quad b > 1$$

Problema é dividido em a subproblemas, cada um com dimensão n/b

- Problema simplificado:

$$T(n) = aT(n/b) + n^d \quad , \quad a \geq 1 , \quad b > 1 , \quad d \geq 0$$

Se $T(n) = aT(n/b) + n^d \quad , \quad a \geq 1 , \quad b > 1 , \quad d \geq 0$, então:

$$T(n) = \begin{cases} O(n^d) & \text{se } d > \log_b a \\ O(n^d \log_b n) & \text{se } d = \log_b a \\ O(n^{\log_b a}) & \text{se } d < \log_b a \end{cases}$$

Interpretação:

- Caso 1: custo da raiz domina o custo total do problema
- Caso 2: custo do problema está uniformemente distribuído
- Caso 3: custo das folhas domina o custo total do problema

Exercício (verdadeiro ou falso)

- Se um algoritmo é $O(N^2)$ então também é $O(N^3)$.

Verdadeiro (embora, evidentemente, seja uma afirmação de pouca utilidade).

- Se o tempo de execução de um algoritmo, no pior caso, escala com $3N^2$ então é $O(N^2)$.

Verdadeiro.

- O tempo de execução do algoritmo G , escala com $2N^3+N$ no pior caso e, no melhor caso, apenas com N^3 .

Logo, G é $\Theta(N^3)$.

Verdadeiro.

3.2 Tipos de algoritmos

Síntese de Algoritmos

- Dividir para conquistar
- Programação Dinâmica
- Algoritmos gananciosos (greedy)

Dividir para Conquistar

- Dividir o problema num conjunto de sub-problemas do mesmo tipo
- Resolver cada sub-problema (com uma chamada recursiva)
- Combinar as soluções dos sub-problemas para obter a solução do problema original

3.3 Listas em Python

Uma lista em python (`list`) é uma sequências de elementos.

Por exemplo

`[]` – lista vazia

`[1, 2, 3]`

`[2, [1, 2]]`

`['a']`

Ao contrário dos tuplos, as listas de 1 elemento não contêm vírgula

Tal como nos tuplos, os elementos de uma lista podem ser outras listas, como por exemplo, é uma lista, a seguinte entidade: `[1, [2], [[3]]]`

Operação	Tipo de argumentos	Valor
<code>$l_1 + l_2$</code>	Listas	Concatenação das listas l_1 e l_2
<code>$l * i$</code>	Lista e inteiro	Repetição i vezes da lista l
<code>$l[i_1:i_2]$</code>	Lista e inteiros	A sublista de l entre os índices i_1 e i_2-1
<code>del(els)</code>	Lista e inteiro(s)	Em que els pode ser da forma $l[i]$ ou $[i_1:i_2]$ Remove os elementos especificados da lista l
<code>e in l</code>	Universal e lista	<code>True</code> se o elemento e pertence à lista <code>False</code> em caso contrário
<code>e not in l</code>	Universal e lista	Negação do resultado da operação <code>e in l</code>
<code>list(a)</code>	Tuplo ou dicionário ou cadeia de caracteres	Transforma o seu argumento numa lista Se não forem fornecidos elementos, o seu argumento é a lista vazia
<code>len(l)</code>	Lista	Número de elementos da lista l

A seguinte interação mostra a utilização de algumas operações sobre listas, onde é possível verificar que Sendo as listas entidades mutáveis, é possível alterar qualquer dos seus elementos.

```
>>> lst1 = [1,2,3]
>>> lst2 = [[4,5]]
>>> lst = lst1 + lst2
>>> lst
[1,2,3,[4,5]]
>>> len(lst)
4
>>> lst[3]
[4,5]
>>> lst[3][0]
4
>>> lst[2] = 'a'
>>> lst
[1,'a',[4,5]]
>>> del(lst[1])
>>> lst
[1,'a',[4,5]]
>>> del(lst[1:])
>>> lst
[1]
```

Vejamos outra interação, com alteração indireta de uma lista a partir de outra:

```
>>> lst1 = [1,2,3,4]
>>> lst2 = lst1  (define novo nome de lst2 como sendo lst1, pseudóminos para a mesma
                  entidade – situação que não se verifica em tuplos ou em cadeias de
                  caracteres, pois estes são estruturas imutáveis)
>>> lst1
[1,2,3,4]
```

```

>>> lst2
[1,2,3,4]
>>> lst2[1] = 'a'
>>> lst2
[1,'a',3,4]
>>> lst1
[1,'a',3,4]
>>> lst1 = 10

```

Vejamos a diferença entre alterar o elemento de uma lista que é partilhada por várias variáveis e alterar uma dessas variáveis

```

>>> lst1
10
>>> lst2
[1,'a',3,4]

```

Passagem de parâmetros por valor – o valor do parâmetro concreto é calculado independentemente de ser uma constante, uma variável ou uma expressão mais complicada e esse valor é associado com o parâmetro formal correspondente (i.e., utilizando a passagem por valor, a função recebe o valor de cada um dos parâmetros e nenhuma informação adicional).

Um parâmetro informal em que seja utilizada a passagem por valor comporta-se, dentro da função, como um nome local que é inicializado com o início da avaliação da função.

Exemplo

```

def troca(x, y)
    print('antes da troca: x=', x, 'y=', y)
    x, y =y, x #os valores são trocados
    print('depois da troca: x=', x, 'y=', y)           (altera o ambiente local)

```

Interação

```

>>> x = 3      (criação de nomes no ambiente global)
>>> y = 10
>>> troca(x, y)      (invoca a função no ambiente local, associa os parâmetros concretos aos
                        parâmetros formais e não altera o ambiente global)
antes da troca : x=3 y=10
depois da troca : x=10 y=3
>>> x
3
>>> y
10

```

3.4 Apresentação do problema

Objectivo

Ordenar por *ordem crescente* as componentes de um vector.

Categorias dos algoritmos de ordenação

- Ordenação por inserção
- Ordenação por selecção
- Ordenação por troca

Um conjunto de elementos é normalmente ordenado para facilitar a procura. Em programação, a utilização de listas ordenadas permite a escrita de algoritmos de procura mais eficientes, por exemplo, a procura binária em vez da procura sequencial.

Podemos agrupar os algoritmos de ordenação em dois grandes grupos (veremos apenas os primeiros): **Algoritmos de ordenação interna** (ordenam um conjunto de elementos que estão simultaneamente armazenados em memória, por exemplo, numa lista) e **algoritmos de ordenação externa** (ordenam elementos que, devido à sua quantidade, não podem estar simultaneamente em memória, estando parte deles armazenados algures no computador – i.e. num ficheiro, para ser mais preciso).

3.5 Algoritmos de Inserção

A) Inserção Linear

Objectivo

Inserir cada uma das n componentes do vector $a = (a_1, a_2, \dots, a_n)$ (começando na segunda e terminando na última) na posição correcta na porção do vector que está para trás dessa mesma componente.

Pseudo-algoritmo

```
for  $i = 2, \dots, n$  do
     $x = a_i;$ 
    "Inserir  $x$  no local apropriado na lista  $a_1, \dots, a_{i-1}$ ;";
end
```

Exemplo

Suponhamos que pretendemos ordenar o vector $a = (44, 55, 12, 42, 94, 18, 6, 67)$.

Ilustração



Como inserir x no local apropriado

Se se encontrar um a_j tal que $x < a_j$, deve-se mover a_j para a direita, e continuar pela esquerda.

Como termina a inserção?

- Ou se encontrou um elemento a_j inferior a x ;
- Ou se chegou ao extremo esquerdo do vector.

A técnica da sentinela

Consiste em copiar o vector a para o vector b , que tem **mais uma componente**, da seguinte forma: $b_2 = a_1$, $b_3 = a_2$, e assim sucessivamente. À componente b_1 dá-se o nome de *sentinela*.

Algoritmo

```
Copiar a para b;  
for i = 3, ..., n + 1 do  
    x = bi;  
    b1 = x;  
    j = i - 1;  
    while x < bj do  
        bj+1 = bj;  
        j = j - 1;  
    end  
    bj+1 = x;  
end  
Copiar b para a;
```

B) Inserção Binária

Constatação

Quando no algoritmo de inserção linear se coloca o elemento x na posição a_i , a sequência a_1, \dots, a_{i-1} já se encontra ordenada. Daqui podemos criar um método **mais rápido** de obter o ponto de inserção de x . → Pesquisa binária

Inserção binária

Divide-se a sequência a_1, \dots, a_{i-1} ao meio o número de vezes suficientes para encontrar o ponto de inserção de x .

Algoritmo

```
for i = 2, ..., n do
    x = ai;
    l = 1;
    r = i - 1;
    while l ≤ r do
        m = (l + r) div 2;
        if x < am then
            r = m - 1;
        else
            l = m + 1;
        end
    end
    for j = i - 1, i - 2, ..., l do
        aj+1 = aj;
    end
    al = x;
end
```

Desafio

Implementar este algoritmo em Python.

Desempenho

- Gerar um vector de 1000 componentes com valores aleatórios;
- Medir o tempo de execução.

3.6 Algoritmos de Procura

A) Procura Linear

Problema: Procurar elemento em vetor (ou lista)

- . O vetor (ou lista) podem não estar ordenados
- . Complexidade: $O(n)$

Procura Linear

```
LinearSearch(A, key)
1  for i = 1 to length[A]
2      do if A[i] = key
3          then return i
4  return 0
```

Desafio

Implementar este algoritmo em Python.

Desempenho

- Gerar um vector de 1000 componentes com valores aleatórios;
- Medir o tempo de execução.

B) Procura binária

(https://www.youtube.com/watch?v=C2apEw9pgtw&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=30)

(https://www.youtube.com/watch?v=uEUXGcc2VXM&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=31)

Problema: Procurar elemento em vetor (ou lista)

. O vetor tem que estar ordenado

. Complexidade: $T(n)=T(n/2)+O(1)$.

Parâmetros do Teorema Mestre: $a=1$, $b=2$, $d=0$

Conclui-se que a complexidade do algoritmo é $O(\log n)$ – caso 2

Procura Binária

BinarySearch(A, l, r, key)

```
1 if l <= r
2   then m = ⌊ (l + r) / 2 ⌋
3   if A[m] = key
4     then return m
5   if A[m] < key
6     then return BinarySearch(A, m+1, r, key)
7   else return BinarySearch(A, l, m-1, key)
8 return 0
```

Procura Binária

BinarySearch(A, l, r, key)

```
1 if l <= r
2   then m = ⌊ (l + r) / 2 ⌋
3   if A[m] = key
4     then return m
5   if A[m] < key
6     then return BinarySearch(A, m+1, r, key)
7   else return BinarySearch(A, l, m-1, key)
8 return 0
```

Elemento de ordem k

Problema: Encontrar o k-ésimo menor valor num dado vetor A

Solução 1

- Ordernar o vector A
- Retornar $A[k]$

- O custo da ordenação domina o custo do problema

- Complexidade: $O(n \cdot \lg n)$

- Aplicar a estratégia dividir-para-conquistar

Função de Partição - Dividir

$$\text{Partition}(A) = (A_L, A_v, A_R)$$

Onde v é um *qualquer* valor em A e:

- A_L contém todos os elementos de A menores que v
- A_v contém todas as ocorrências de v em A
- A_R contém todos os elementos de A maiores que v

- Aplicar a estratégia dividir-para-conquistar

Função de Seleção - Conquistar

$$\text{Select}(A, k) = \begin{cases} \text{Select}(A_L, k) & \text{se } k \leq |A_L| \\ v & \text{se } |A_L| < k \leq |A_L| + |A_v| \\ \text{Select}(A_R, k - |A_L| - |A_v|) & \text{se } k > |A_L| + |A_v| \end{cases}$$

Pseudo-código

Função de Partição Esquerda

```

LeftPartition(A, l, r, v)
1 i = l
2 for j = l to r
3     do if A[j] < v
4         then Swap(A, i, j); i++
5 return i

```

Função de Partição Direita

```

RightPartition(A, l, r, v)
1 i = l
2 for j = l to r
3     do if A[j] == v
4         then Swap(A, i, j); i++
5 return i

```

Função de Partição

Partition(A, l, r)

- 1 $v = A[r]$
- 2 $i = \text{LeftPartition}(A, l, r, v)$
- 3 $j = \text{RightPartition}(A, i, r, v)$
- 4 **return** (i, j, v)

Função de Seleção

Select(A, l, r, k)

- 1 $(i, j, v) = \text{Partition}(A, l, r)$
- 2 **if** $(k \leq (i-l))$
3 **then return** Select(A, l, i-1, k)
- 4 **elseif** $(k \leq (j-l))$
5 **then return** v
- 6 **else return** Select(A, j, r, k-(j-l))

Complexidade

- **Melhor caso:** $T(n) = T(n/2) + O(n)$. Parâmetros do Teorema Mestre: $a = 1$, $b = 2$, $d = 1$. Complexidade: $O(n)$ (Caso 1).
- **Caso médio:** $T(n) = T(3n/4) + O(n)$. Parâmetros do Teorema Mestre: $a = 1$, $b = 4/3$, $d = 1$. Complexidade: $O(n)$ (Caso 1).
- **Pior Caso:** $T(n) = T(n - 1) + O(n)$. Complexidade: $O(n^2)$ (método a utilizar: expansão da equação).

3.7 Seleção Linear

Estratégia

Em cada passo,

- ① Seleccionar o elemento com menor valor;
- ② Trocá-lo com o primeiro;
- ③ Eliminar o primeiro elemento da lista.

Exemplo



Algoritmo

```
for  $i = 1, \dots, n - 1$  do
     $k = i;$ 
     $x = a_i;$ 
    for  $j = i + 1, \dots, n$  do
        if  $a_j < x$  then
             $k = j;$ 
             $x = a_j;$ 
        end
    end
     $a_k = a_i;$ 
     $a_i = x;$ 
end
```

3.8 Algoritmos de Ordenação Elementar

Motivação

- Porquê estudar algoritmos elementares ?
- Razões de ordem prática
 - Fáceis de codificar e por vezes suficientes
 - Rápidos/eficientes para problemas de dimensão média e por vezes os melhores em certas situações
- Razões pedagógicas
 - Bom exemplo para aprender terminologia e contexto dos problemas a codificar e por vezes suficiente
 - Alguns são fáceis de generalizar para algoritmos mais eficientes ou para melhorar o desempenho de outros algoritmos

Análise de Desempenho

- Parâmetro de interesse - tempo de execução
 - regra: tempo $O(N^2)$ para ordenar N items
 - mas se N for pequeno podem ser os melhores
- Mas desempenho em memória também interessa
 - ordenação *in-place*
 - utilizando memória adicional

Algoritmo Estável

- Um algoritmo de ordenação é dito **estável** se preserva a ordem relativa dos items com chaves repetidas
 - ex: ordenar lista de alunos, já previamente ordenada por nome, por ano de graduação
- Algoritmos elementares são normalmente estáveis, mas poucos algoritmos avançados o são

Algoritmo Interno

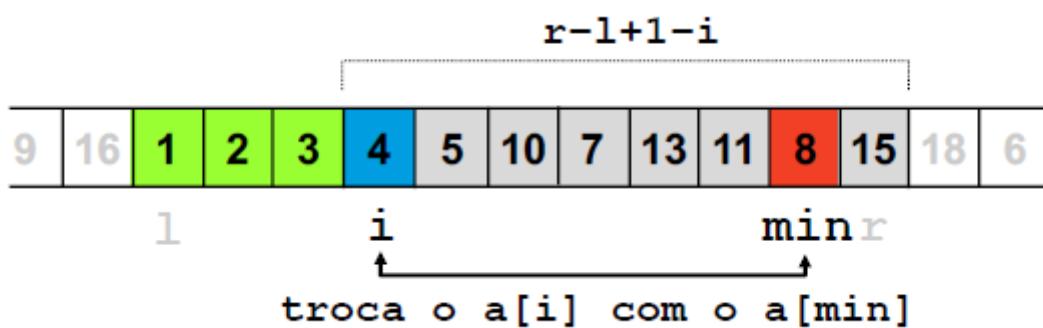
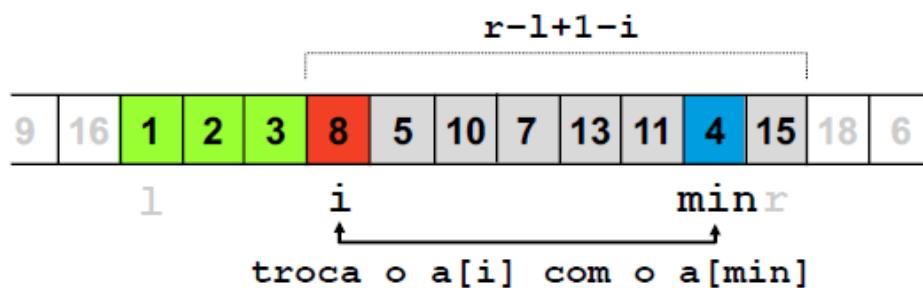
- Um algoritmo de ordenação é dito **interno** se o conjunto de todos os dados a ordenar couber em memória RAM; caso contrário é dito externo
- Distinção muito importante:
 - **ordenação interna** pode aceder a qualquer dado com um custo muito pequeno
 - **ordenação externa** tem de aceder aos dados de forma sequencial (ou em blocos)
- Vamos estudar apenas algoritmos de ordenação interna

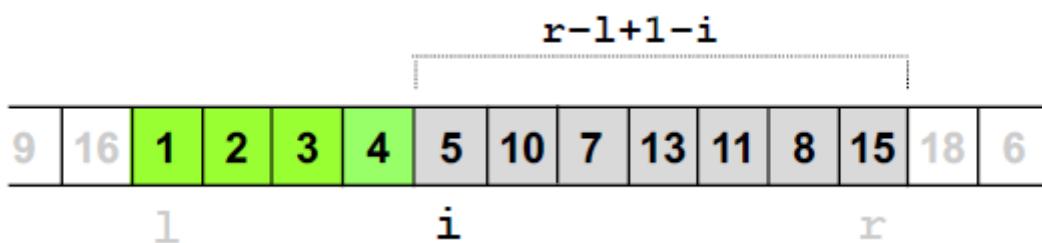
(A) Selection Sort

- Para cada elemento i entre as posições l e r
 - Procura o menor elemento entre i e r
 - Se o menor valor — guardado na posição min — for menor que o valor guardado na posição i ,
 - troca o $a[i]$ com o $a[\text{min}]$

		1
8		
5		
2		
6		
9		
3		
1		
4		
0		
7		r

- A cada passo, escolher o menor entre os $r-l+1-i$ maiores elementos
- Para cada i , os primeiros i elementos ficam já na sua posição final
- Para cada valor de i do primeiro ciclo, segundo ciclo é executado $r-i$ vezes





Exercício

- Considere o seguinte vector

$v[] = \{72, 29, 38, 22, 60, 2\}$

Indique o conteúdo de v no final de cada passo do algoritmo **selection sort**.

Início: 72 29 38 22 60 2
 Passo 1: 2 29 38 22 60 72
 Passo 2: 2 22 38 29 60 72
 Passo 3: 2 22 29 38 60 72
 Passo 4: 2 22 29 38 60 72
 Final: 2 22 29 38 60 72

Exercício

- Considere o seguinte vector

$a[] = \{22, 33, 1, 10, 11, 2\}$

Indique o conteúdo de a depois de 3 iterações do algoritmo **selection sort**.

Início: 22 33 1 10 11 2
 1 33 22 10 11 2
 1 2 22 10 11 33
 R: 1 2 10 22 11 33

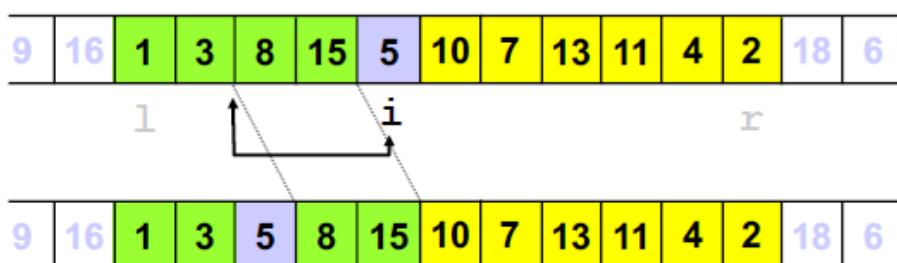
- Tempo de execução:
 - Comparações: $N^2 / 2$, trocas: N
 - No **pior caso** é $O(N^2)$, com $N = r-1$
 - No **melhor caso**, é $O(N^2)$, com $N = r-1$
- O algoritmo **não é estável**
 - i.e. ordem relativa de chaves duplicadas não é mantida
 - Podemos alterar o algoritmo para que este fique estável.



(B) Insertion Sort

(<https://www.youtube.com/watch?v=gbJzL6IJig0>)

- Para cada i , os primeiros i elementos ficam ordenados, embora possam ainda não ficar na sua posição final
- Isso significa que, se o vector já estiver ordenado, fazemos apenas $N-1$ comparações!



Exercício

- Considere o seguinte vector

$v[] = \{72, 29, 38, 22, 60, 2\}$

Indique o conteúdo de v no final de cada passo do algoritmo **insertion sort**.

Início: * 72 29 38 22 60 2

Passo 1: 29 * 72 38 22 60 2

Passo 2: * 29 38 72 22 60 2

Passo 3: 22 29 38 * 72 60 2

Passo 4: * 22 29 38 60 72 2

Final: 2 22 29 38 60 72

- Tempo de execução:

- No **pior caso** é $O(N^2)$, com $N = r-1$, i.e. vector já ordenado por ordem inversa
- No **melhor caso** é $O(N)$, com $N = r-1$, i.e. vector já ordenado

- Algoritmo é **estável**

- i.e. ordem relativa de chaves duplicadas é mantida

Exercício (Verdadeiro ou Falso)

- O tempo de execução do InsertionSort é $\Omega(N)$.

Verdadeiro.

O tempo de execução no melhor caso é $\Omega(N)$, logo o Insertion Sort é $\Omega(N)$.

- O tempo de execução do InsertionSort é $\Theta(N^2)$.

Falso.

$\Theta(N^2)$ sse $\Omega(N^2)$ e $O(N^2)$. Como o insertionSort não é $\Omega(N^2)$ não pode ser $\Theta(N^2)$.

- O número de trocas no algoritmo InsertionSort é $O(N)$.

Falso.

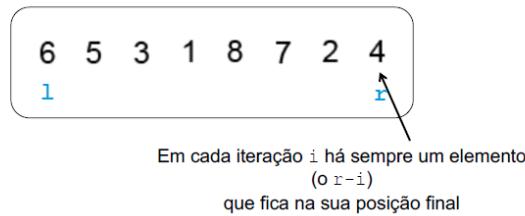
O número de trocas é $O(N^2)$.

- O número de trocas no algoritmo SelectionSort é $O(N)$.

Verdadeiro.

(C) Bubble Sort

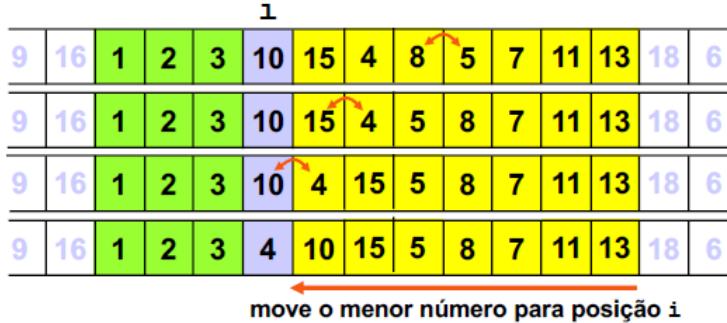
(https://www.youtube.com/watch?v=p_ETf2CKY4)



- Para cada valor de i no primeiro ciclo, o segundo ciclo é executado $r-i$ vezes
- Para cada i , os **últimos** i elementos ficam já na sua posição final
- Só são efectuadas trocas entre elementos adjacentes

Bubble Sort (direita para a esquerda)

- Para cada valor de i no primeiro ciclo, o segundo ciclo é executado $r-i$ vezes
- Para cada i , os **primeiros** i elementos ficam já na sua posição final
- Só são efectuadas trocas entre elementos adjacentes



Exercício

- Considere o seguinte vector

$v[] = \{72, 29, 38, 22, 60, 2\}$

Indique o conteúdo de v no final 2 passagens do algoritmo **bubble sort (esquerda para a direita)**.

Primeira passagem: 72 29 38 22 60 2

29 72 38 22 60 2

29 38 72 22 60 2

29 38 22 72 60 2

29 38 22 60 72 2

29 38 22 60 2 **72**

Segunda passagem: 29 38 22 60 2 **72**

29 38 22 60 2 72

29 22 38 60 2 72

29 22 38 60 2 72

29 22 38 2 **60 72**

Comparação

- Pior Caso

	Selection	Insertion	Bubble
Comparações	$N^2 / 2$	$N^2 / 2$	$N^2 / 2$
Trocas Chaves	N	$N^2 / 2$	$N^2 / 2$

- Caso Médio

	Selection	Insertion	Bubble
Comparações	$N^2 / 2$	$N^2 / 4$	$N^2 / 2$
Trocas Chaves	N	$N^2 / 4$	$N^2 / 2$

- Tabelas com poucos elementos fora de ordem
 - **Insertion e Bubble Sort** são quase lineares
 - os melhores algoritmos de ordenação podem ser quadráticos
- Contexto onde elementos são grandes e chaves pequenas
 - **Selection Sort** é linear no número de dados
 - N dados com tamanho M palavras
 - custo comparação: 1; custo troca: M
 - $N^2 / 2$ comparações e NM custo de trocas
 - termo NM domina: custo proporcional ao tempo necessário para mover os dados
- Alternativa: uso de ponteiros

Avaliação Experimental

N	Selection	Insertion	Bubble
1000	5	4	11
2000	21	15	45
4000	85	62	182

- Bubble/Insertion Sort são lentos: trocas ocorrem apenas entre items adjacentes
 - se o menor item está no final da tabela, serão precisos N passos para o colocar na posição correcta
- **Shellsort:**
 - acelerar o algoritmo permitindo trocas entre elementos que estão afastados
 - bubble/insertion sort, mas com elementos distanciados de h

Estratégia

Para cada componente i do vector:

- ① Localizar a menor componente ainda não colocada do vector;
- ② Por trocas entre componentes consecutivas, colocar essa componente na posição i .

Exemplo

44	55	12	42	94	18	6	67
↓ trocas entre elementos consecutivos							
6	44	55	12	42	94	18	67
↓ trocas entre elementos consecutivos							
6	12	44	55	42	94	18	67
↓ trocas entre elementos consecutivos							
6	12	18	44	55	42	94	67
↓ trocas entre elementos consecutivos							
6	12	18	42	44	55	94	67
↓ trocas entre elementos consecutivos							
6	12	18	42	44	55	94	67
↓ trocar entre elementos consecutivos							
6	12	18	42	44	55	67	94

Algoritmo

```
for i = 2, ..., n do
    for j = n, n - 1, ..., i do
        if aj-1 > aj then
            x = aj-1;
            aj-1 = aj;
            aj = x;
        end
    end
end
```

(D) Shell Sort ($h=2$)

- Vector diz-se h -ordenado se qualquer sequência de números separados por h posições está ordenada



- Vector é equivalente a h sequências ordenadas entrelaçadas



- O resultado de h -ordenar um vector que está k -ordenado, é um vector que está h -ordenado e k -ordenado

Shell Sort - Ideia

- Rearranjar os dados de forma a que estejam h -ordenados
 - Usando valores de h grandes é possível mover elementos na tabela distâncias grandes
 - Ordenar primeiro para valores de h grandes e depois para valores de h pequenos
 - facilita as últimas ordenações, para valores de h pequenos
 - Utilizando este procedimento para qualquer sequência de h 's que termine em 1 produz no final uma tabela ordenada
 - Cada passo torna o próximo mais simples

Exercício

$$h=3$$

19	2	15	18	4	11	17	18	5	1	3	22	16
				i								

18	2	15	19	4	11	17	18	5	1	3	22	16
----	---	----	----	---	----	----	----	---	---	---	----	----

18	2	15	19	4	11	17	18	5	1	3	22	16
----	---	----	----	---	----	----	----	---	---	---	----	----

18	2	11	19	4	15	17	18	5	1	3	22	16
----	---	----	----	---	----	----	----	---	---	---	----	----

18	2	11	19	4	15	17	18	5	1	3	22	16
----	---	----	----	---	----	----	----	---	---	---	----	----

i

18	2	11	19	4	15	17	18	5	1	3	22	16
----	---	----	----	---	----	----	----	---	---	---	----	----

i

17	2	11	18	4	15	19	18	5	1	3	22	16
----	---	----	----	---	----	----	----	---	---	---	----	----

i

17	2	11	18	4	15	19	18	5	1	3	22	16
----	---	----	----	---	----	----	----	---	---	---	----	----

i

17	2	11	18	4	15	19	18	5	1	3	22	16
----	---	----	----	---	----	----	----	---	---	---	----	----

i

17	2	11	18	4	15	19	18	5	1	3	22	16
----	---	----	----	---	----	----	----	---	---	---	----	----

i

17	2	5	18	4	11	19	18	15	1	3	22	16
----	---	---	----	---	----	----	----	----	---	---	----	----

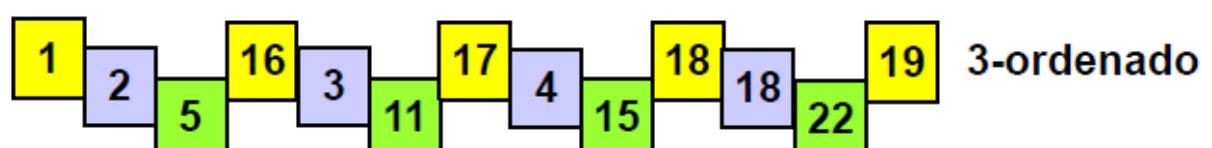
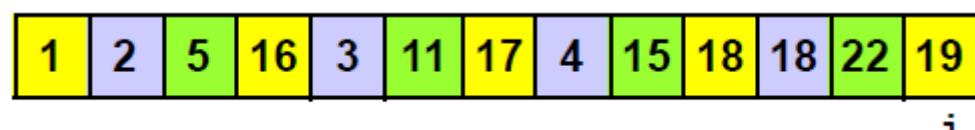
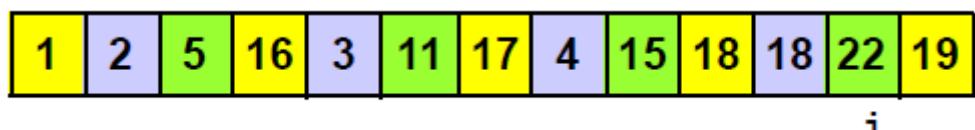
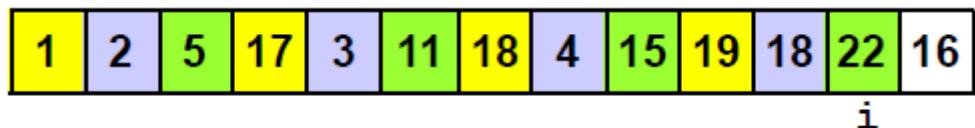
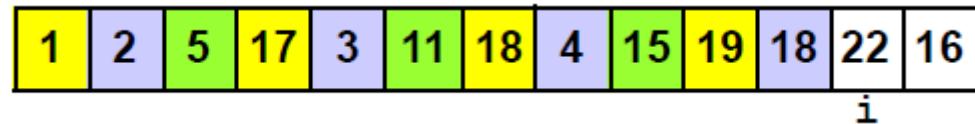
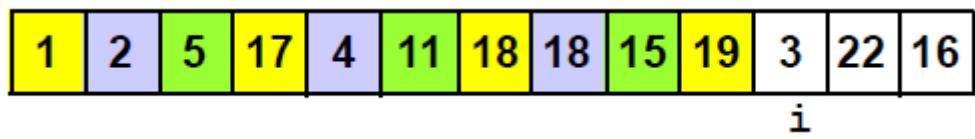
i

17	2	5	18	4	11	19	18	15	1	3	22	16
----	---	---	----	---	----	----	----	----	---	---	----	----

i

1	2	5	17	4	11	18	18	15	19	3	22	16
---	---	---	----	---	----	----	----	----	----	---	----	----

i



Shell Sort - Funcionamento

- Operação (vector com tamanho 150):
 - Para cada valor de h , 40, 13, 4, 1:
 - Utilizar Insertion sort para criar h sub-vectores ordenados dentro de vector com tamanho 150
 - Vector fica h -ordenado
 - Para $h = 40$ existem 40 sub-vectores ordenados, cada um com 3/4 elementos
 - Para $h = 13$ existem 13 sub-vectores ordenados, cada um com 11/12 elementos
 - ...
 - Para $h = 1$ existe 1 (sub-)vector ordenado, com 150 elementos

Escolha da Sequência de Ordenação

- Questão difícil de responder
- Propriedades de muitas sequências já foram estudadas
- Possível provar que umas melhores que outras
 - 1, 4, 13, 40, 121, 364, 1093, 3280, ... (Knuth, $\frac{3N+1}{2}$)
 - melhor que 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, ... (Shell, $\frac{2^i}{4^{i+1} + 3 \times 2^i + 1}$)
Porquê?
 - mas pior (20%) que 1, 8, 23, 77, 281, 1073, ... ($\frac{4^{i+1} + 3 \times 2^i + 1}{4^{i+1}}$)
- Na prática utilizam-se sequências que decrescem geometricamente para que o número de incrementos seja logarítmico
- A sequência óptima ainda não foi descoberta

elementos nas posições pares não são comparados com elementos nas posições ímpares até ao passo final

- Questão difícil de responder
- Propriedades de muitas sequências já foram estudadas
- Possível provar que umas melhores que outras
 - 1, 4, 13, 40, 121, 364, 1093, 3280, ... (Knuth, $\frac{3N+1}{2}$)
 - melhor que 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, ... (Shell, $\frac{2^i}{4^{i+1} + 3 \times 2^i + 1}$)
Porquê?
 - mas pior (20%) que 1, 8, 23, 77, 281, 1073, ... ($\frac{4^{i+1} + 3 \times 2^i + 1}{4^{i+1}}$)
- Na prática utilizam-se sequências que decrescem geometricamente para que o número de incrementos seja logarítmico
- A sequência óptima ainda não foi descoberta

Shell Sort - Complexidade

- Análise rigorosa da complexidade do algoritmo é desconhecida
- Complexidade depende da sequência de valores h utilizada
 - Sequência 1, 4, 13, 40, 121, 364, ... $\sim O(N^{3/2})$ comparações
 - Sequência 1, 8, 23, 77, 281, 1073, ... $\sim O(N^{4/3})$ comparações
 - Sequência 1, 2, 3, 4, 6, 9, 8, 12, 18, ... $\sim O(N(\log N)^2)$ comparações

Shell Sort - Avaliação Experimental

N	O	K	G	S	P	I
12500	16	6	6	5	6	6
25000	37	13	11	12	15	10
50000	102	31	30	27	38	26
100000	303	77	60	63	81	58
200000	817	178	137	139	180	126

- O: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, ...
- K: 1, 4, 13, 40, 121, 364, ...
- G: 1, 2, 4, 10, 23, 51, 113, 249, 548, ...
- S: 1, 8, 23, 77, 281, ...
- P: 1, 7, 8, 49, 56, 64, 343, 392, 448, 512, ...
- I: 1, 5, 19, 41, 109, 209, 505, 929, ...

Exercício

a = { 16, 8, 4, 19, 20, 5, 13, 11, 6, 12 }

h=4

16, 8, 4, 19, 20, 5, 13, 11, 6, 12

Resultado:

16, 8, 4, 19, 20, 5, 13, 11, 6, 12

6, 5, 4, 11, 16, 8, 13, 19, 20, 12

O Shell Sort é estável? NÃO!!

3.9 Algoritmos de Ordenação Eficiente

(A) Quick Sort

(<https://www.youtube.com/watch?v=7h1s2SojIRw>)

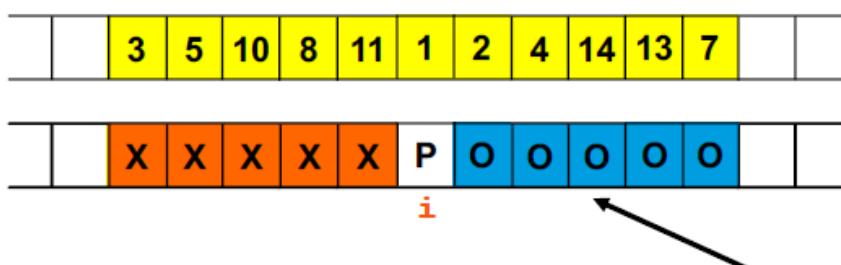
(<https://www.youtube.com/watch?v=-qOVVRIZZao>)

(https://www.youtube.com/watch?v=7h1s2SojIRw&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=36)

(https://www.youtube.com/watch?v=-qOVVRIZZao&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=37)

- Inventado nos anos 60 por A. R. Hoare
- Vantagens
 - popular devido à facilidade de implementação e eficiência
 - $O(N \lg N)$, em **média**, para ordenar N objectos
 - ciclo interno muito simples e conciso
- Inconvenientes
 - não é estável; $O(N^2)$ no **pior caso!**
 - frágil: qualquer pequeno erro de concretização pode não ser detectado mas levar a ineficiência
- Biblioteca C fornece uma concretização: `qsort()`
- Aplica método **dividir para conquistar** para ordenar (“divide and conquer”)
- Ideia chave: efectuar **partição dos dados** e **ordenar as várias partes independentemente** (de forma recursiva)
 - particionar os dados: menores para um lado, maiores para outro
 - usar recursão e aplicar algoritmo a cada uma das partes
 - processo de partição é crítico para evitar partições degeneradas

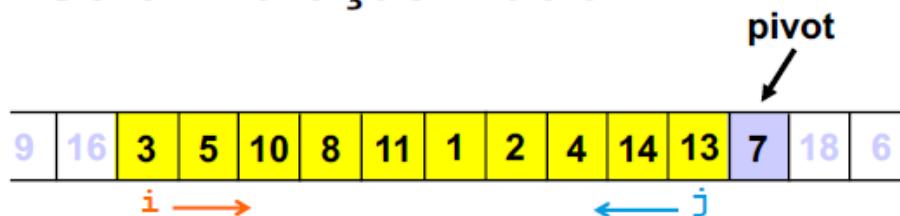
Quick Sort - ideia



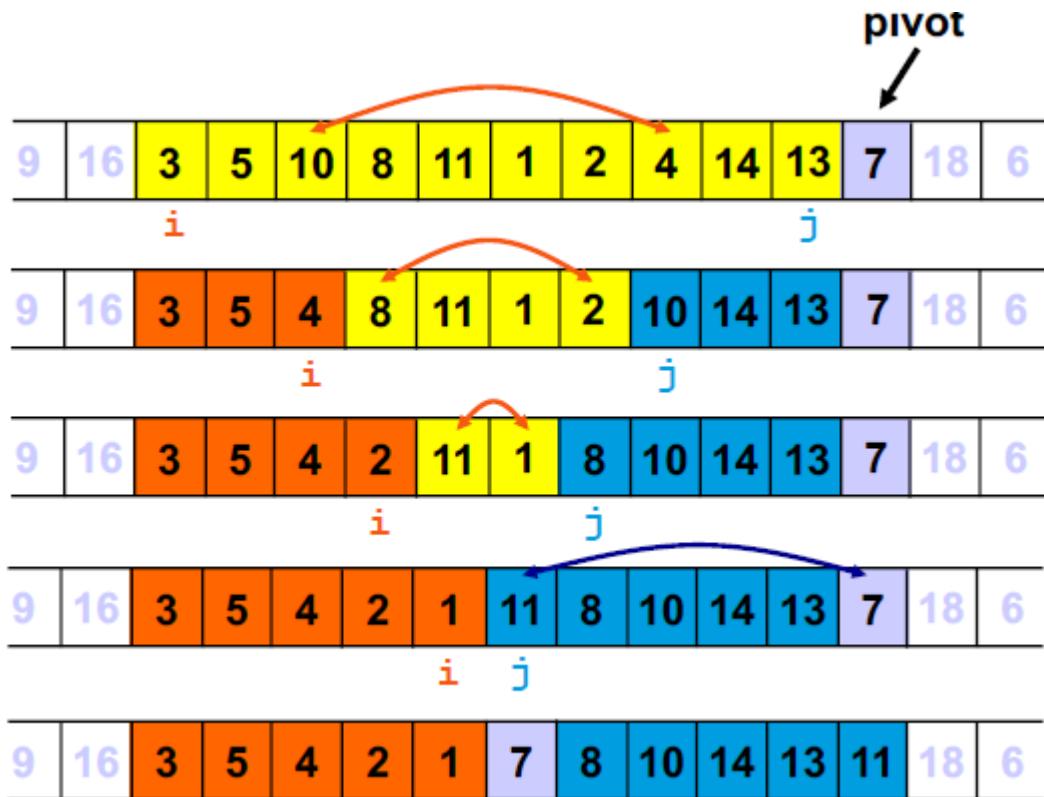
Todos os X's < P e
Todos os O's > P
P fica na posição final

- Volto a aplicar o mesmo algoritmo aos XX e aos OO's
- Quando não pudermos dividir mais, paramos

Quick Sort – Partição - ideia



- 2 iteradores: o **i** (esquerda) e o **j** (direita, mas antes do pivot)
- Avanço com **i** (para a direita) até encontrar um elemento **maior que o pivot**, ou seja, que deveria estar **no lado direito do vector**.
- Avanço com **j** (para a esquerda) até encontrar um elemento **menor que o pivot**, ou seja, que deveria estar **no lado esquerdo do vector**.
- Troco **a[i]** com **a[j]**
- Repito estes passos até que **i** cruze com **j**.
- Nessa altura, troco o **pivot** com o **a[i]**, colocando-o na divisão entre o **“lado” esquerdo** e o **lado “direito”** do vector.



- Recebe vector *a* e os limites da parte a ordenar, *l* e *r*
- Rearranja os elementos do vector de forma a que as **quatro condições** seguintes sejam válidas
 - o elemento *a[i]*, para algum *i*, fica na sua posição final (i.e., o pivot)
 - nenhum dos elementos de *a[l]* a *a[i - 1]* é maior do que *a[i]*
 - nenhum dos elementos de *a[i + 1]* a *a[r]* é menor do que *a[i]*
 - processo coloca pelo menos um elemento na sua posição final
- Após partição, a tabela fica sub-dividida em duas partes que podem ser ordenadas independentemente aplicando o mesmo processo

Quick Sort - Recursão

- Ordenação realizada através de partição e aplicação recursiva do algoritmo aos dois subconjuntos de dados daí resultantes

9	16	3	5	4	2	1	7	8	10	14	13	11	18	6
9	16	1	5	4	2	3	7	8	10	14	13	11	18	6
9	16	1	2	3	5	4	7	8	10	14	13	11	18	6
9	16	1	2	3	4	5	7	8	10	14	13	11	18	6
9	16	1	2	3	4	5	7	8	10	11	13	14	18	6
9	16	1	2	3	4	5	7	8	10	11	13	14	18	6
9	16	1	2	3	4	5	7	8	10	11	13	14	18	6
9	16	1	2	3	4	5	7	8	10	11	13	14	18	6

Quick Sort | função partição | Exercício

- Pivot = $v = a[right] = 6$
 $10, 2, -2, 13, 14, -1, 7, 5, 0, 6$
 $i \qquad \qquad \qquad j$
- Troco o 10 com o 0
 $0, 2, -2, 13, 14, -1, 7, 5, 10, 6$
 $i \qquad \qquad \qquad j$
- Troco o 13 com o 5
 $0, 2, -2, 5, 14, -1, 7, 13, 10, 6$
 $i \qquad \qquad \qquad j$
- Troco o 14 com o -1

Quick Sort | função partição | Exercício

0, 2, -2, 5, -1, 14, 7, 13, 10, 6
 i j

- Como $i \geq j$, nada acontece e saímos do ciclo while. Agora trocamos o elemento na posição i (o 14) com o pivot (`exch(a[i], a[right])`), ficando

0, 2, -2, 5, -1, **6**, **7, 13, 10, 14**
 i

- Retorno o i (neste caso o índice 1+5)

Quick Sort - Complexidade

- **Pior caso:** pivot é sempre o maior/menor elemento
 - Cerca de $N^2 / 2$ comparações
 - Partições degeneram e a função chama-se a si própria N vezes;
 - O número de comparações é
$$N + (N-1) + (N-2) + \dots + 2 + 1 = \frac{(N+1)N}{2}$$
 - Na nossa implementação se o vector estiver ordenado
- Não apenas o tempo necessário para a execução do algoritmo cresce quadraticamente como o espaço necessário para o processo recursivo é de cerca de N o que é inaceitável para vectores grandes. É possível modificar para obter espaço $O(\log N)$

- Melhor caso: quando cada partição divide o vector de entrada em duas metades iguais
 - Número de comparações usadas por `quicksort` satisfaz a recursão de dividir para conquistar: $C_N = 2C_{N/2} + N$
 - Solução: $C_N \approx N \lg N$
- Propriedade: QuickSort efectua cerca de $C_N \approx 2N \lg N$ comparações **em média**

Quick Sort - Melhorias

- Algoritmo pode ainda ser melhorado com alterações triviais
- Ordenação de sub-vectores de pequenas dimensões pode ser efectuada de forma mais eficiente
 - Natureza recursiva de QuickSort garante que uma fracção grande dos sub-vectores terão tamanho pequeno
- Como escolher correctamente o elemento de partição?
 - Aleatoriamente
 - Média de vários elementos
- Como melhorar o desempenho se os dados tiverem um grande número de chaves repetidas?



- QuickSort é garantido instanciar-se a si próprio múltiplas vezes para vectores pequenos!
- Conveniente utilizar o melhor método possível nesta situação: insertion sort

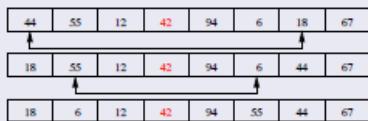
Estratégia

Consideremos um elemento x do vector, escolhido aleatoriamente.

- 1 Percorremos o vector da esquerda para a direita até encontrarmos um elemento a_i tal que $a_i > x$;
- 2 Percorremos o vector da direita para a esquerda até encontrarmos um elemento a_j tal que $a_j < x$;
- 3 Trocamos os dois elementos a_i e a_j ;
- 4 Repetimos os três pontos anteriores até que os dois percursos se encontrem sensivelmente a meio do vector.

O resultado final desta estratégia é um vector constituído por dois *subvectores*, em que o da esquerda contém os elementos inferiores a x e o da direita os elementos superiores a x .

Ilustração



Algoritmo do processo de partição

```
i = 1;  
j = n;  
"escolher aleatoriamente um elemento x";  
repeat  
    while  $a_i \leq x$  do  
        i = i + 1;  
    end  
    while  $x \leq a_j$  do  
        j = j - 1;  
    end  
    if  $i \leq j$  then  
        w =  $a_i$ ;  
         $a_i = a_j$ ;  
         $a_j = w$ ;  
        i = i + 1;  
        j = j - 1;  
    end  
until  $i > j$ ;
```

Objectivo

Considerando x sensivelmente o ponto médio do vector, gerar as duas partições como indicado anteriormente, repetir o processo em cada uma das partições, às partições das partições, e assim sucessivamente, até que cada partição tenha apenas um elemento.

Subalgoritmo qsort(l,r)

```
Função qsort(l,r);
i = l;
j = r;
x = a(l+r)/2;
repeat
    while ai ≤ x do
        i = i + 1;
    end
    while x ≤ aj do
        j = j - 1;
    end
    if i ≤ j then
        w = ai;
        ai = aj;
        aj = w;
        i = i + 1;
        j = j - 1;
    end
until i > j;
if l > j then
    qsort(l,j);
end
if i < r then
    qsort(i,r);
end
```

Quicksort = qsort(1,n)

Desempenho

- Gerar um vector de 1000 componentes com valores aleatórios;
- Medir o tempo de execução para o Bubblesort e Quicksort;
- Comparar com o desempenho dos algoritmos anteriores.

Pseudo-código

QuickSort

```
QuickSort(A, l, r)
1  if l < r
2      then m = Partition(A, l, r)
3          QuickSort(A, l, m-1)
4          QuickSort(A, m+1, r)
```

QuickSort - Função de Partição

```
Partition(A, l, r)
1  v = A[r]
2  i=l
3  for j = l to r-1
4      do if A[j] ≤ v
5          then swap(A, i, j); i++
6
7  swap(A, i, r)
8  return i
```

Complexidade

- **Melhor caso:** $T(n) = 2T(n/2) + O(n)$. Parâmetros do Teorema Mestre: $a = 2$, $b = 2$, $d = 1$. Complexidade: $O(n \log n)$ (Caso 2).
- **Caso médio:** $T(n) = T(3n/4) + T(n/4) + O(n)$. Complexidade: $O(n \log n)$ (solução obtida expandindo a equação).
- **Pior Caso:** $T(n) = T(n - 1) + O(n)$. Complexidade: $O(n^2)$ (solução obtida expandindo a equação).

Exercício

8, 1, -2, 10, 12, -7, 7, 11, 0, 5

(o pivot é o 5!)

o iterador i começa por parar no 8 e o j no 0:
troco o 8 com o 0

0, 1, -2, 10, 12, -7, 7, 11, 8, 5

o iterador i pára no 10 e o j no -7: troco o 10 com o -7
0, 1, -2, -7, 12, 10, 7, 11, 8, 5

o iterador i pára no 12 e o j no -7: como o $i > j$ já não troco

0, 1, -2, -7, 12, 10, 7, 11, 8, 5

Resta-me portanto trocar o a[i] com o pivot, ficando:

resultado: 0, 1, -2, -7, 5, 10, 7, 11, 8, 12

...e a função partição deverá retornar o inteiro 4

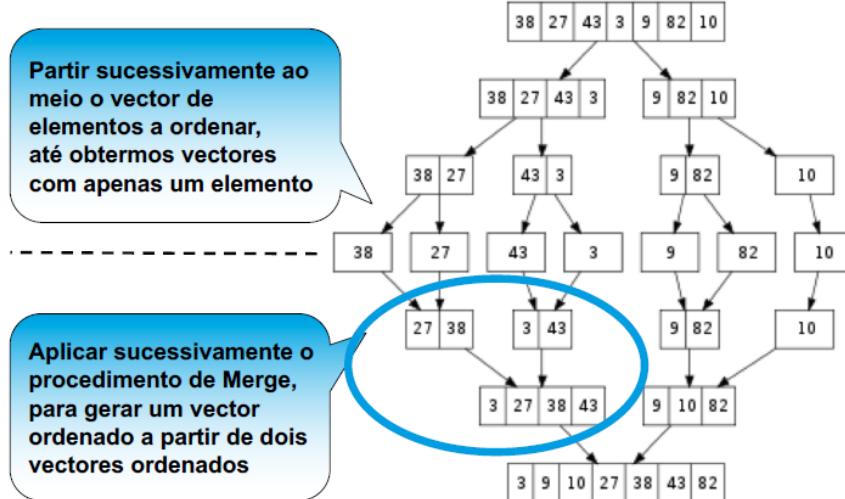
(B) Merge Sort

(https://www.youtube.com/watch?v=6pV2IF0fgKY&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=33)

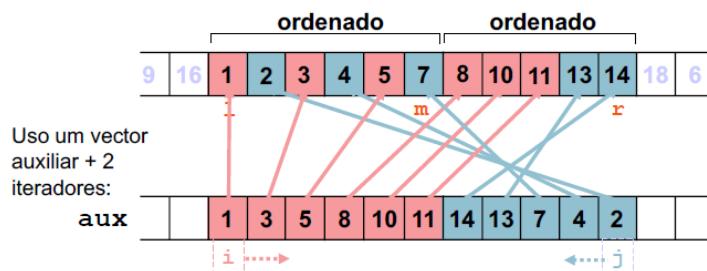
(https://www.youtube.com/watch?v=mB5HXBb_HY8&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=34)

(https://www.youtube.com/watch?v=ak-pz7tS5DE&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=35)

Merge Sort - ideia



- Gerar um vector ordenado a partir de dois vectores ordenados é simples!



Exercício

0	1	2	3	4	5	6	7	8	9	10
3	5	10	8	11	1	2	4	14	13	7
3	5	10	8	11	1	2	4	14	13	7
3	5	10	8	11	1	2	4	14	13	7
3	5	10	8	11	1	2	4	14	13	7
3	5	10	1	8	11	2	4	14	13	7
1	3	5	8	10	11	2	4	14	13	7
1	3	5	8	10	11	2	4	14	13	7
1	3	5	8	10	11	2	4	14	7	13
1	3	5	8	10	11	2	4	7	13	14
1	2	3	4	5	7	8	10	11	13	14

Main Algorithm

```
MergeSort(A, l, r)
1  if l < r
2    then m = ⌊(l + r)/2⌋
3      MergeSort(A, l, m)
4      MergeSort(A, m+1, r)
5      Merge(A, l, m, r)
```

Merge

```
Merge(A, l, m, r)
1  let L[1 .. (m-l)+2] be a new array
2  let R[1 .. (r-m)+1] be a new array
3  for i = 1 to (m-l)+1
4    do L[i] = A[l+i-1]
5  for j = 1 to (r-m)
6    do R[j] = A[m+j]
7  L[(m-l)+2] = ∞; R[(m-l)+1] = ∞
8  i = 1; j = 1
9  for k = l to r
10   do if L[i] ≤ R[j]
11     then A[k] = L[i]; i++
12     else A[k] = R[j]; j++
```

Merge Sort - Complexidade

- Tempo de execução:

$$T_N = T_{\lfloor N/2 \rfloor} + T_{\lceil N/2 \rceil} + O(N) = O(N \lg N)$$

- Fácil de verificar quando N é potência de 2, e no caso geral recorrendo a indução
- Complexidade do pior caso é $O(N \lg N)$

- $T(n) = 2T(n/2) + O(n)$
- Aplicando o Teorema Mestre ($a = 2$, $b = 2$, $d = 1$) obtemos: $O(n \lg n)$ (caso 2).

QuickSort *versus* MergeSort

QuickSort

- Vector não necessariamente dividido em 2 partes iguais
- Constantes menores
- Pior caso (vector ordenado): $O(n^2)$

MergeSort

- Vector dividido em 2 partes iguais
- Necessário fazer Merge (constantes maiores)
- Pior caso: $O(n \lg n)$

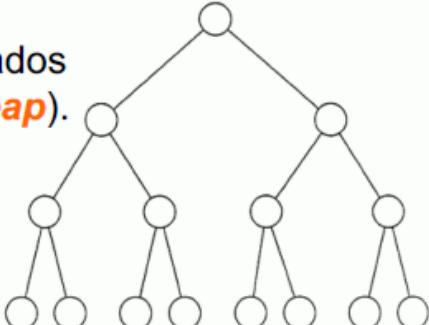
Na prática: QuickSort (aleatorizado) é normalmente mais rápido

(C) Heap Sort

(<https://www.youtube.com/watch?v=HqPJF2L5h9U>)

(https://www.youtube.com/watch?v=HqPJF2L5h9U&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=32)

HeapSort (ideia base)

- Podemos pensar no **heapsort** como um selection sort mais eficiente.
- o **heapsort** mantém os dados organizados numa estrutura de dados (chamada **heap**).
 - A raiz dessa estrutura, contém sempre o maior elemento.
 - Os elementos podem ser sucessivamente removidos da raiz da **heap**, na ordem desejada, tendo o cuidado de manter as propriedades dessa estrutura

1 Amontoados (Heaps)

- Operações sobre Amontoados
- Algoritmo Heap-Sort
- Outras Operações

Problema

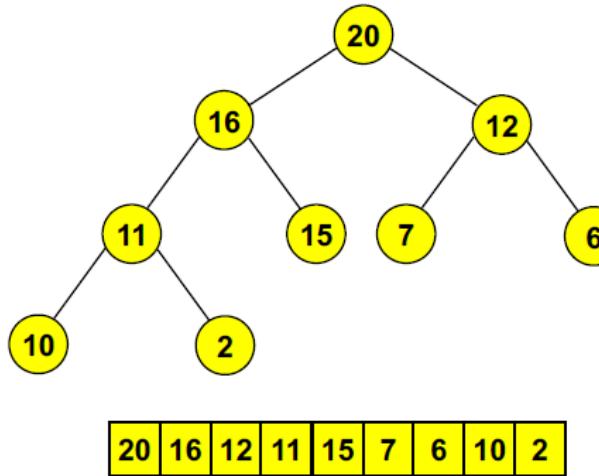
Suponha que pretende gerir a execução de processos. Cada processo tem um valor de prioridade. O valor da prioridade diminui quando o processo é executado por alguns ciclos e aumenta enquanto está à espera. Em cada momento, é seleccionado o processo com maior prioridade para ser executado.

Para além de gerir a prioridade dos processos, deve considerar que o conjunto de processos a gerir aumenta (criação de um processo) e diminui (termina a execução de um processo).

Proponha uma estrutura de dados para manipular a informação sobre os processos. Qual a complexidade de cada operação na sua estrutura de dados?

Das árvores aos amontoados (*heaps*)

- Um vector de elementos que pode ser visto como uma árvore binária

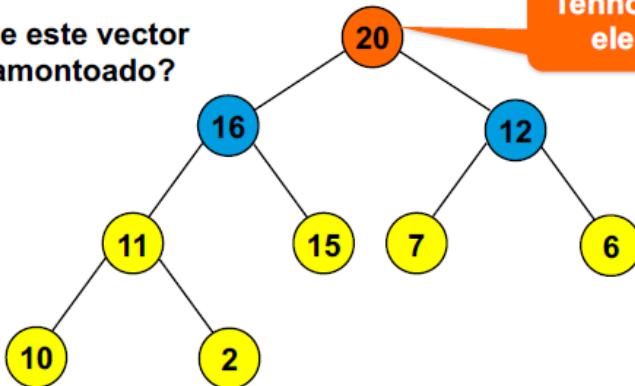


Amontoados: Definição

1. Nenhum nó tem uma chave superior à raiz
2. Heap-condition: *a chave de cada nó é sempre maior ou igual que as chaves de ambos os filhos*

Será que este vector é um amontoado?

Tenho sempre o maior elemento na raiz!



Exercício

Quais dos seguintes vectores corresponde a um amontoado (heap)?

- a. <50, 25, 30, 27, 24, 21, 28>
- b. <50, 30, 25, 27, 24, 28, 21>
- c. <60, 50, 9, 40, 41, 10, 8>
- d. <40, 15, 18, 13, 11, 14, 16>
- e. <60, 30, 80, 10, 35, 70, 40>

1. Nenhum nó tem uma chave superior à raiz
2. Heap-condition: *a chave de cada nó é sempre maior ou igual que as chaves de ambos os filhos*

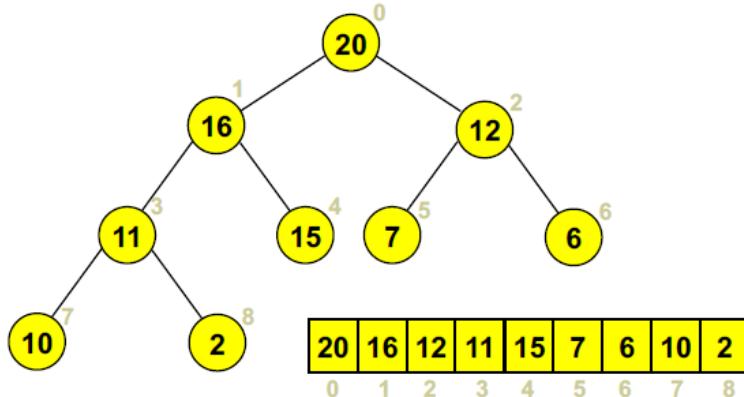
Solução: d

Amontoado: Índices

- Pai do nó i é o nó $\lfloor(i + 1)/2\rfloor - 1$
- Filhos do nó i são os nós $2i + 1$ e $2(i + 1)$

Filho esquerdo

Filho direito



Amontoados: Propriedades

Vector de valores interpretado como uma árvore binária (essencialmente completa)

- $\text{length}[A]$: tamanho do vector
- $\text{heap-size}[A]$: número de elementos no amontoado
- $A[1]$: Raiz da árvore

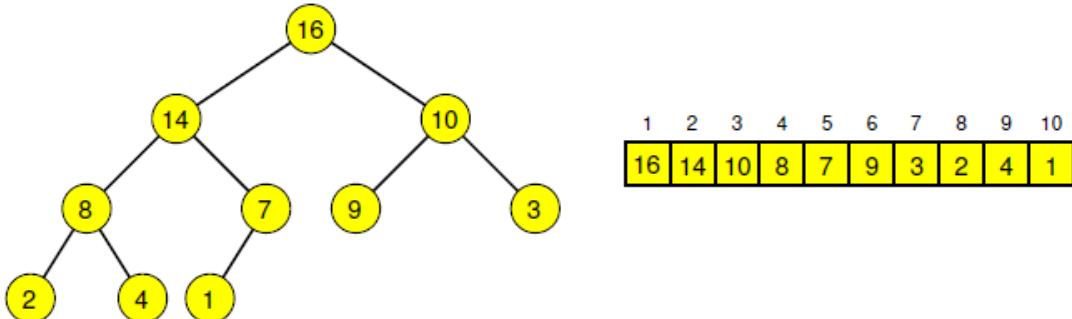
Relações entre nós da árvore

- $\text{Parent}(i) = \lfloor i/2 \rfloor$
- $\text{Left}(i) = 2i$
- $\text{Right}(i) = 2i + 1$

Propriedade de amontoado

- $A[\text{Parent}(i)] \geq A[i]$

Exemplo



Árvores binárias

Árvore Binária Completa

- Cada nó tem 2 (ou 0) filhos
- Qualquer nó folha (i.e. nó sem descendentes) tem uma mesma profundidade d
 - Profundidade: número de nós entre a raiz e um dado nó
- Todos os nós internos tem grau 2
 - Cada nó interno tem exactamente 2 filhos

Árvore Binária Essencialmente Completa

- Todos os nós internos têm grau 2, com 1 possível excepção
 - Existe nó à profundidade $d - 1$ com filho esquerdo, mas sem filho direito
- Nós folha posicionados à profundidade d ou $d - 1$
 - Qualquer nó interno à profundidade $d - 1$, posicionado à esquerda de qualquer nó folha à mesma profundidade

Operação Max-Heapify

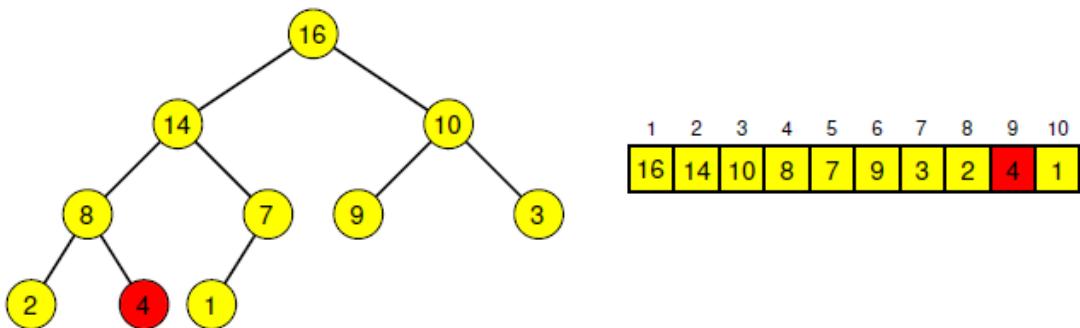
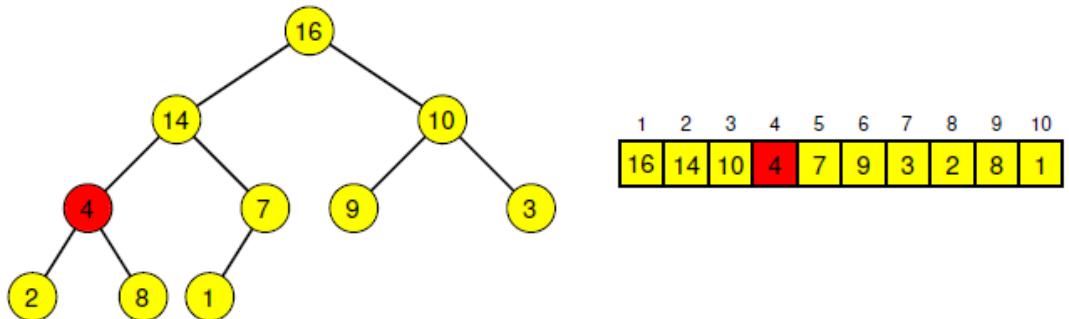
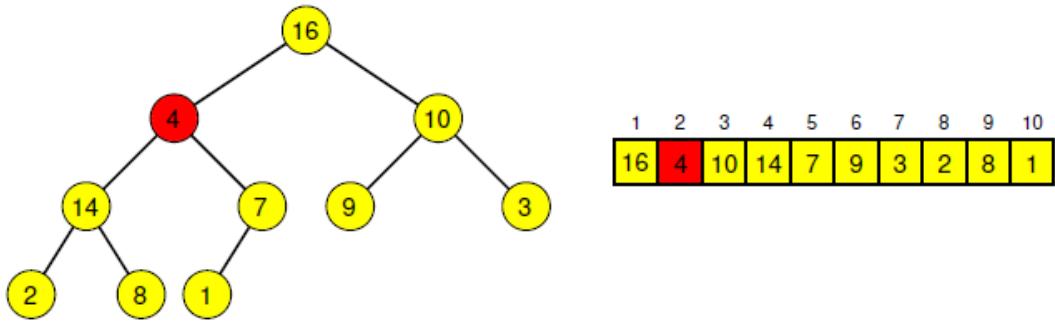
Max-Heapify

- Transforma a árvore com raiz em i num amontoado, assumindo que as árvores com raiz em $\text{Left}(i)$ e $\text{Right}(i)$ são amontoados
- Troca recursivamente valores entre elementos que não verifiquem a propriedade do amontoado
- Complexidade:
 - Altura da Heap: $h = \lfloor \lg n \rfloor$
 - Complexidade de Max-Heapify: $O(h) = O(\lg n)$

Max-Heapify(A, i)

```
1   $l \leftarrow \text{Left}(i)$ 
2   $r \leftarrow \text{Right}(i)$ 
3  if  $l \leq \text{heap-size}[A]$  and  $A[l] > A[i]$ 
4    then  $\text{largest} \leftarrow l$ 
5  else  $\text{largest} \leftarrow i$ 
6  if  $r \leq \text{heap-size}[A]$  and  $A[r] > A[\text{largest}]$ 
7    then  $\text{largest} \leftarrow r$ 
8  if  $\text{largest} \neq i$ 
9    then exchange  $A[i] \leftrightarrow A[\text{largest}]$ 
10   Max-Heapify( $A, \text{largest}$ )
```

Exemplo



Operação Build-Max-Heap

- Construir amontoado a partir de um vector arbitrário
- Chamada selectiva de Max-Heapify

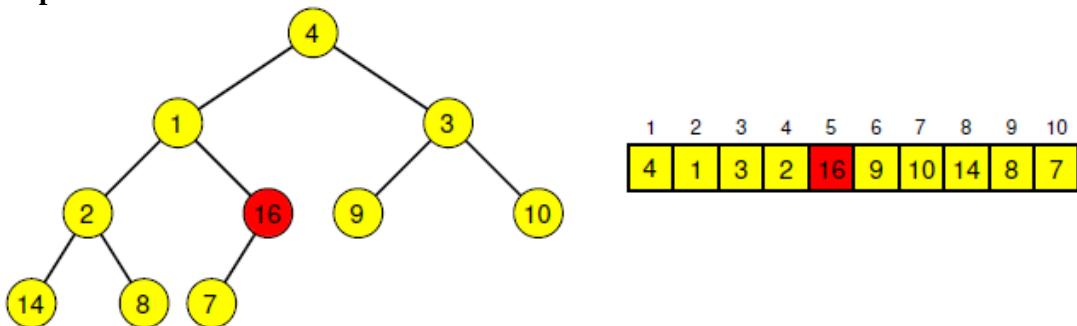
Build-Max-Heap(A)

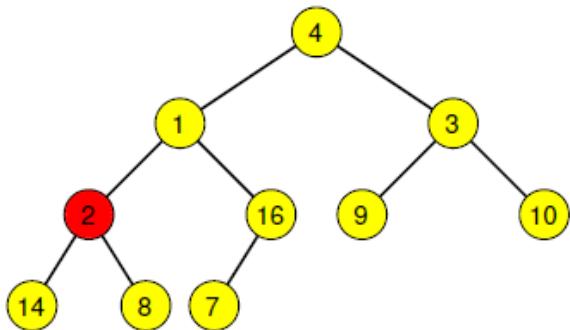
```

1  $heap\text{-}size}[A] \leftarrow length[A]$ 
2 for  $i \leftarrow \lfloor length[A]/2 \rfloor$  downto 1
3   do Max-Heapify( $A, i$ )

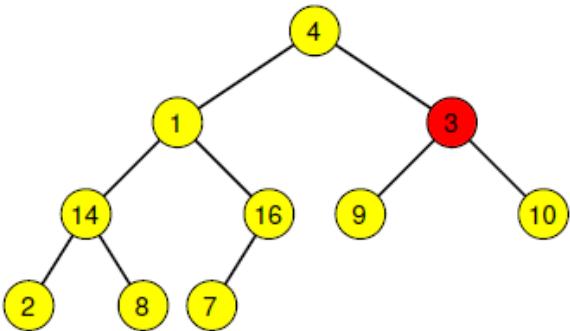
```

Exemplo

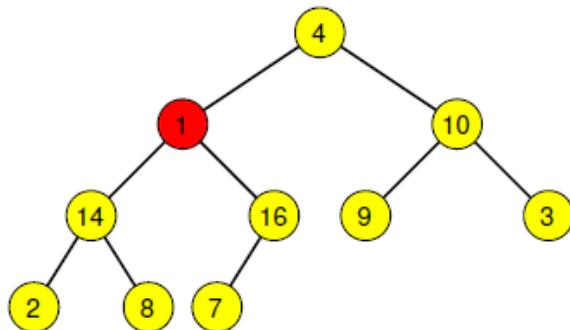




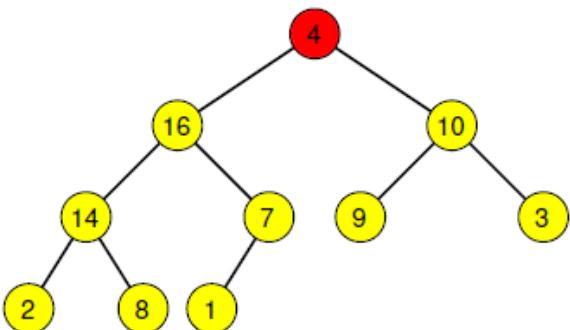
1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7



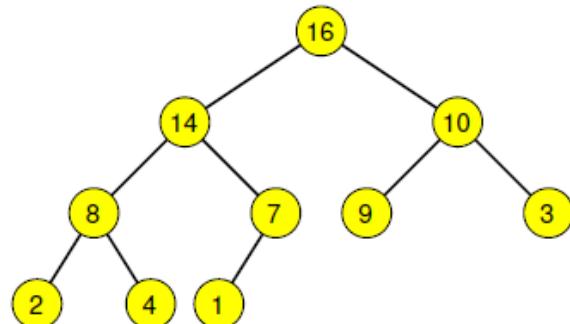
1	2	3	4	5	6	7	8	9	10
4	1	3	14	16	9	10	2	8	7



1	2	3	4	5	6	7	8	9	10
4	1	10	14	16	9	3	2	8	7



1	2	3	4	5	6	7	8	9	10
4	16	10	14	7	9	3	2	8	1



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

- Construir amontoado a partir de um vector arbitrário
- Chamada selectiva de Max-Heapify

Build-Max-Heap(A)

```

1  heap-size[ $A$ ]  $\leftarrow$  length[ $A$ ]
2  for  $i \leftarrow \lfloor \text{length}[A]/2 \rfloor$  downto 1
3    do Max-Heapify( $A, i$ )

```

Complexidade: $O(n \lg n)$

Complexidade ? $O(n \lg n)$, mas é possível provar $O(n)$

Ordenação Heap-sort

- Intuição
 - Extrair consecutivamente o elemento máximo de uma heap
 - Colocar esse elemento na posição (certa) do vector

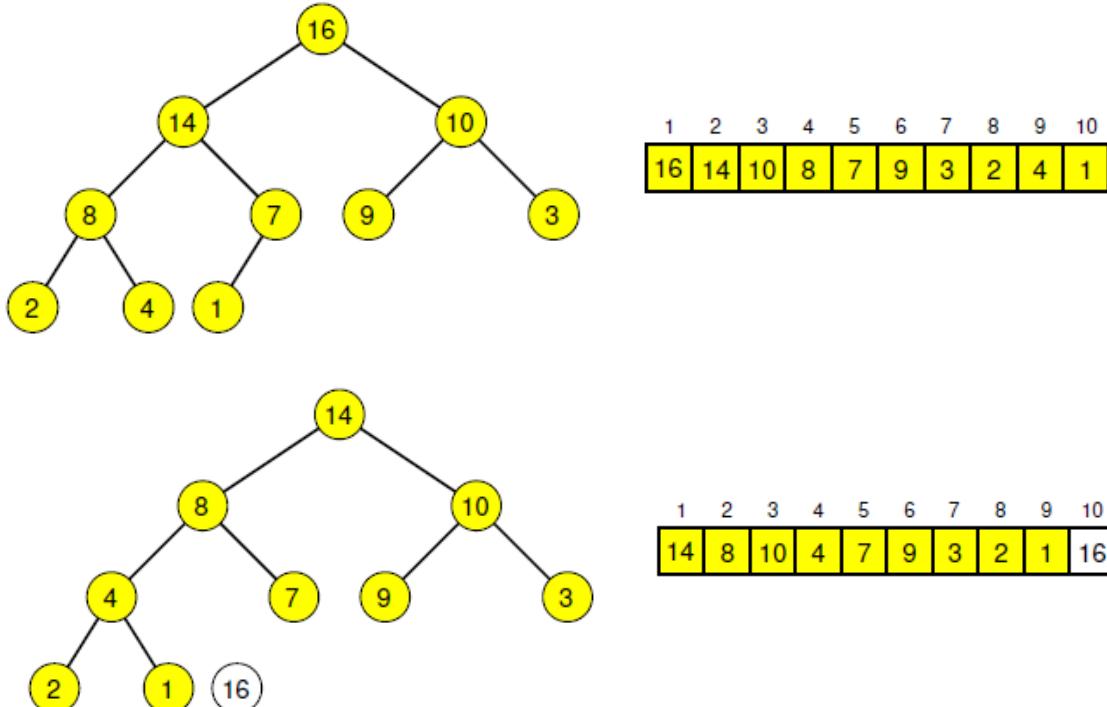
Heap-Sort(A)

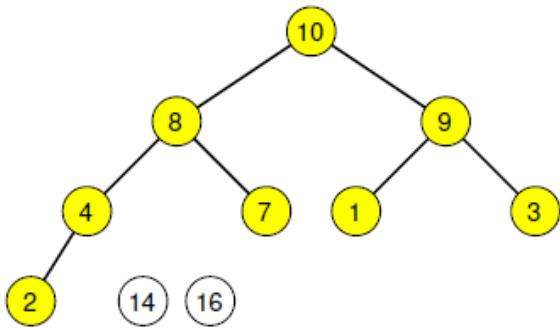
```

1  Build-Max-Heap( $A$ )
2  for  $i \leftarrow \text{length}[A]$  downto 2
3    do exchange  $A[1] \leftrightarrow A[i]$ 
4      heap-size[ $A$ ]  $\leftarrow$  heap-size[ $A$ ] – 1
5      Max-Heapify( $A, 1$ )

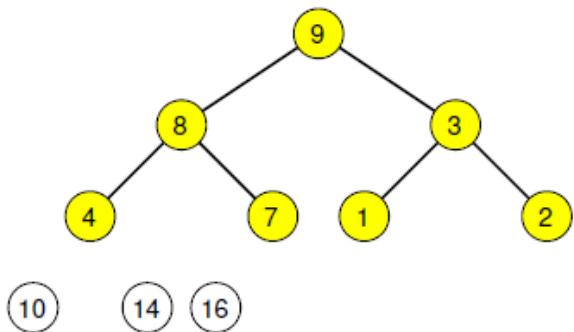
```

Exemplo

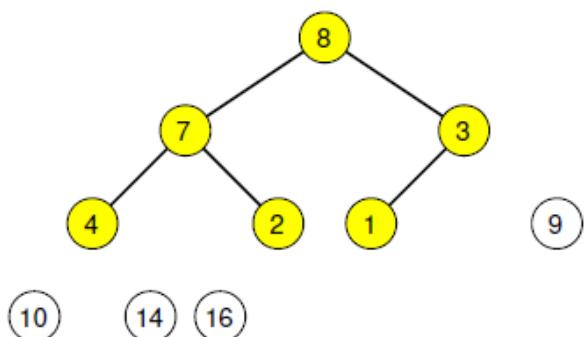




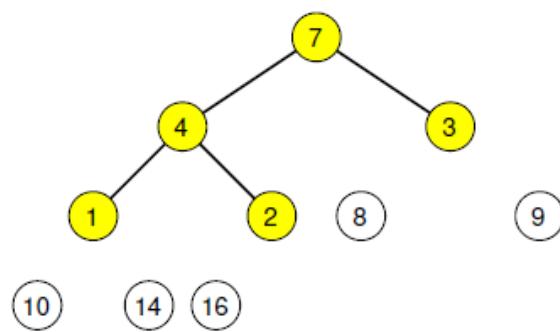
1	2	3	4	5	6	7	8	9	10
10	8	9	4	7	1	3	2	14	16



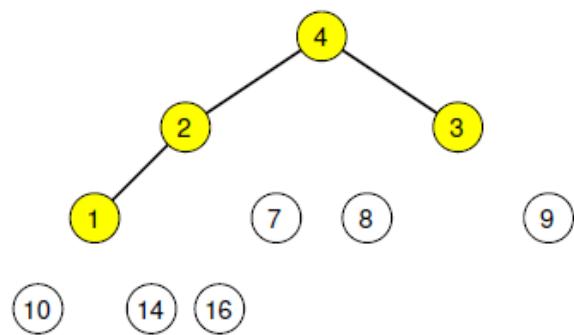
1	2	3	4	5	6	7	8	9	10
9	8	3	4	7	1	2	10	14	16



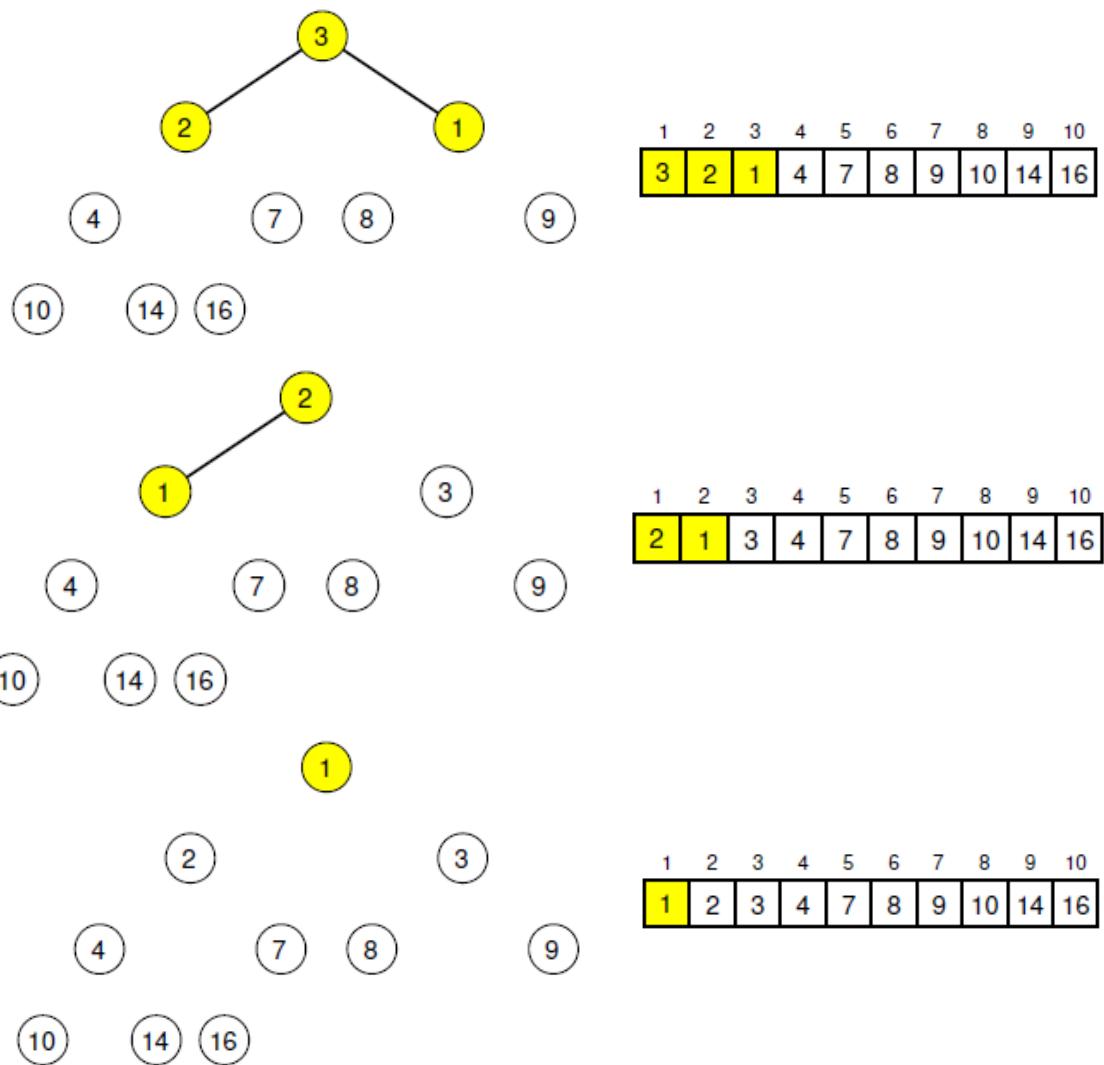
1	2	3	4	5	6	7	8	9	10
8	7	3	4	2	1	9	10	14	16



1	2	3	4	5	6	7	8	9	10
7	4	3	1	2	8	9	10	14	16



1	2	3	4	5	6	7	8	9	10
4	2	3	1	7	8	9	10	14	16



- Intuição

- Extrair consecutivamente o elemento máximo de uma heap
- Colocar esse elemento na posição (certa) do vector

Heap-Sort(A)

```

1 Build-Max-Heap( $A$ )
2 for  $i \leftarrow \text{length}[A]$  downto 2
3   do exchange  $A[1] \leftrightarrow A[i]$ 
4    $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$ 
5   Max-Heapify( $A, 1$ )

```

Complexidade ? $O(n \lg n)$

Heap-Max e Heap-Extract-Max

Heap-Maximum(A)

1 **return** $A[1]$

Complexidade ? $O(1)$

Heap-Extract-Max(A)

```
1  $max \leftarrow A[1]$ 
2  $A[1] \leftarrow A[\text{heap-size}[A]]$ 
3  $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$ 
4 Max-Heapify( $A, 1$ )
5 return  $max$ 
```

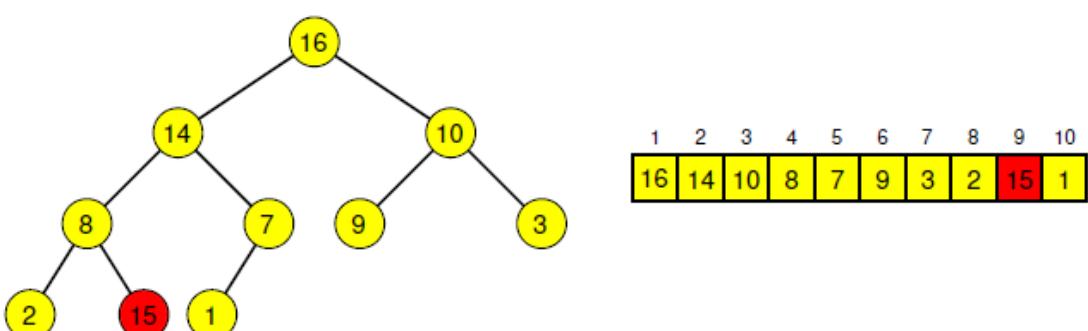
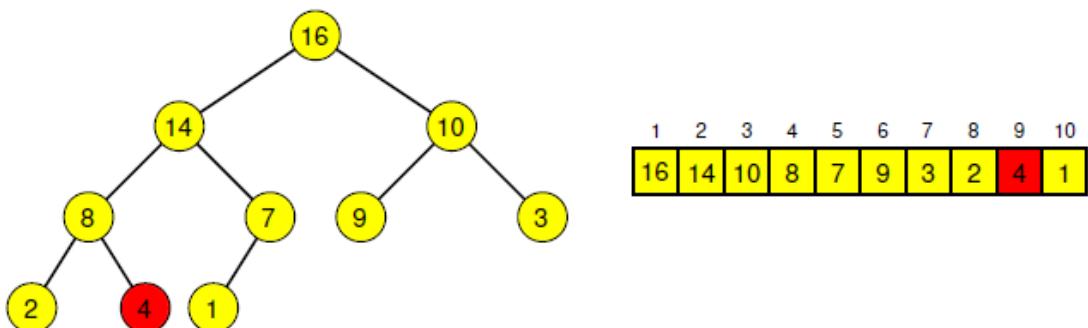
Complexidade ? $O(\lg n)$

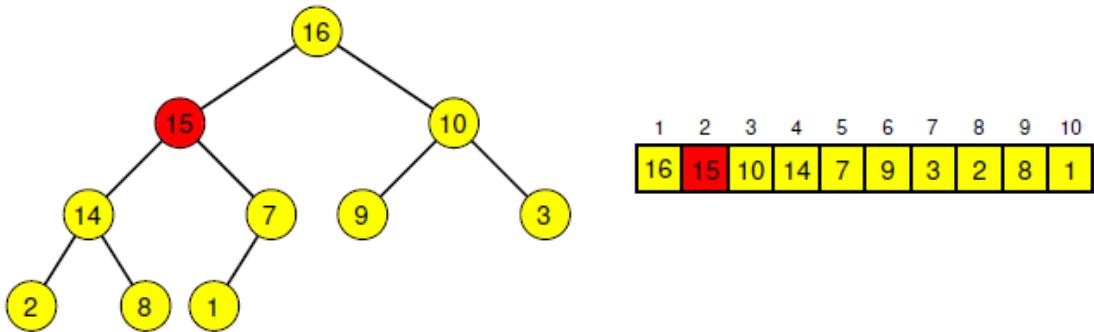
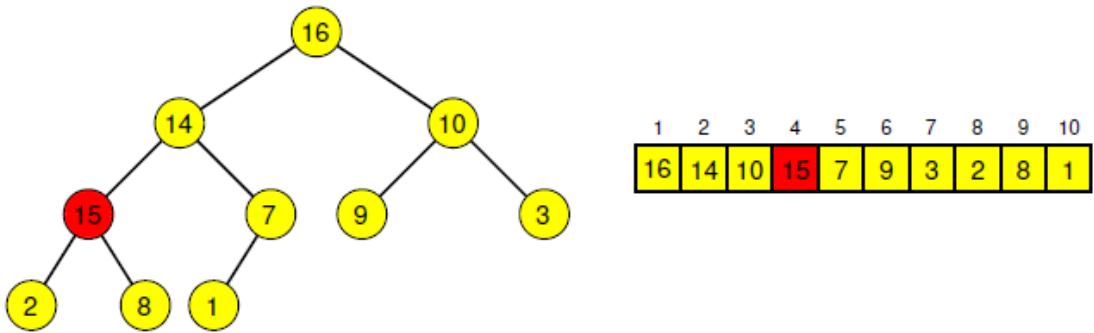
Min e Extract-Min ?

Heap-Increase-Key

Heap-Increase-Key(A, i, key)

```
1  $A[i] \leftarrow key$ 
2 while  $i > 1$  and  $A[\text{Parent}(i)] < A[i]$ 
3     do exchange  $A[i] \leftrightarrow A[\text{Parent}(i)]$ 
4          $i \leftarrow \text{Parent}(i)$ 
```





Heap-Increase-Key(A, i, key)

```

1  $A[i] \leftarrow key$ 
2 while  $i > 1$  and  $A[\text{Parent}(i)] < A[i]$ 
3   do exchange  $A[i] \leftrightarrow A[\text{Parent}(i)]$ 
4    $i \leftarrow \text{Parent}(i)$ 

```

Complexidade ? $O(\lg n)$

Max-heap-insert

Max-Heap-Insert(A, key)

```

1  $\text{heap-size}[A] \leftarrow \text{heap-size}[A] + 1$ 
2  $A[\text{heap-size}[A]] \leftarrow -\infty$ 
3 Heap-Increase-Key( $A, \text{heap-size}[A], key$ )

```

Complexidade ? $O(\lg n)$

Operações em Amontoados: fixDown

Fixdown (i):

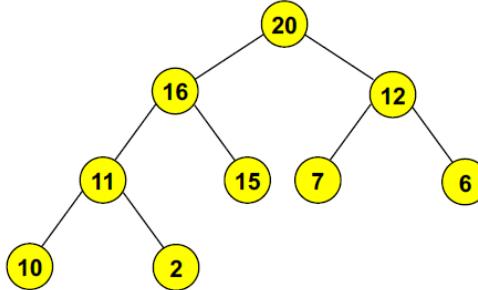
- chamada quando se pretende diminuir a chave de um nó
- confirma se ambos os filhos são menores do que “ele”;
- se não for o caso, troca de posição com o filho maior e volta a chamar-se para a posição onde o filho maior estava.

Ideia do código em Linguagem Coloquial:

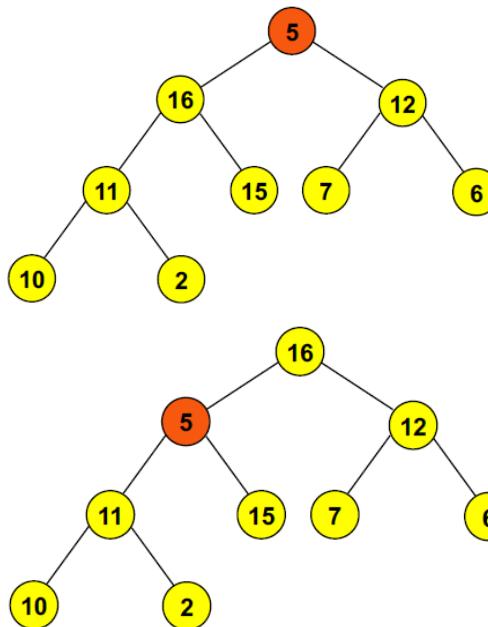
```
Fixdown (indice i)
    verifica se i tem um filho maior que ele
    SE i tem um filho maior:
        troca i com o filho_maior
        Fixdown (pos_do_filho_maior)
```

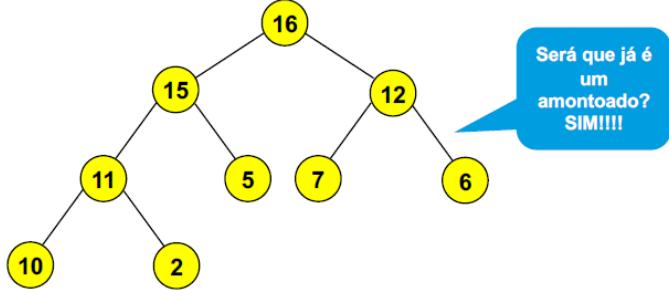
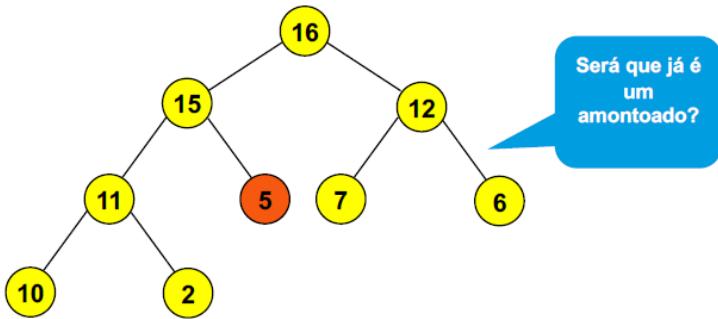
Função
recursiva

- Se tivermos um amontoado...



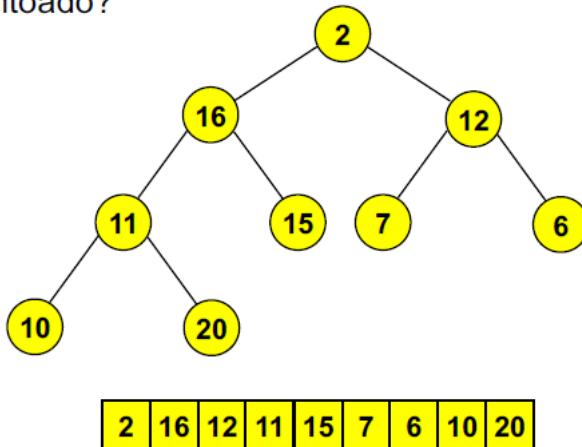
- Se tivermos um amontoado... e alterarmos a raiz.
- Podemos voltar a arrumar o vector num “amontoado”, aplicando o fixDown na raiz



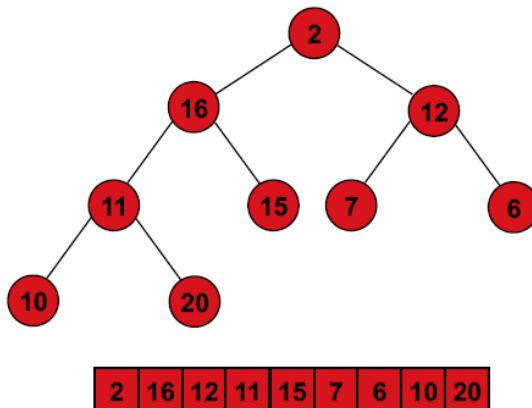


Operações em Amontoados: `buildheap`

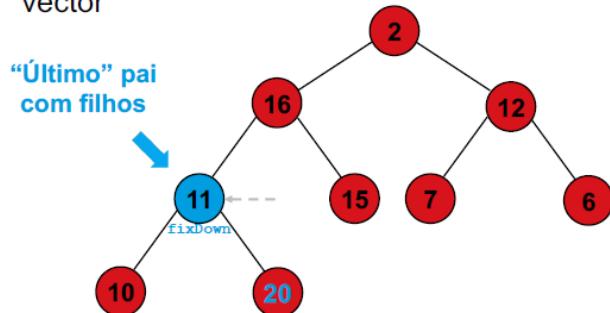
- Mas como transformar um vector arbitrário num amontoado?



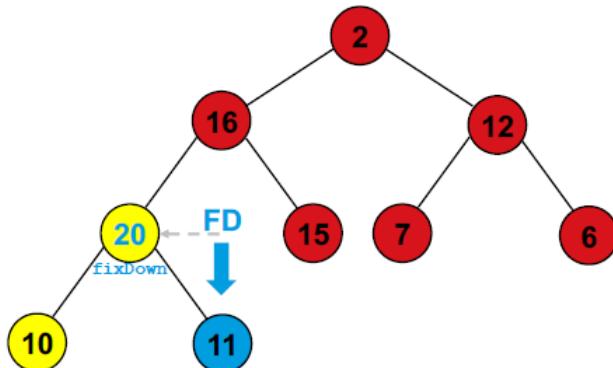
- Começo por transformar o vector numa árvore.



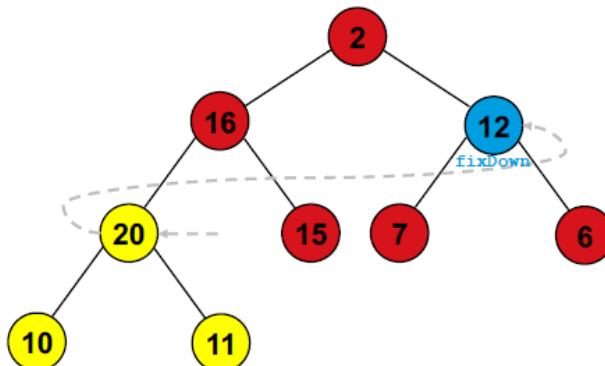
- Chamo **fixDown** do último até ao primeiro elemento do vector



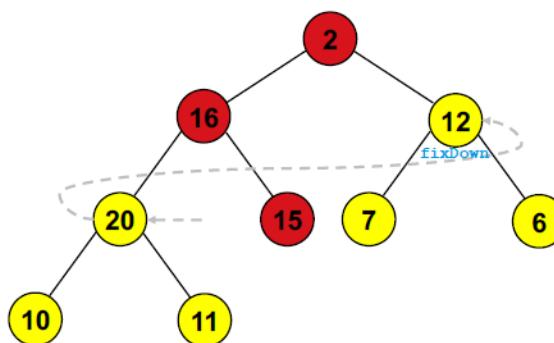
2	16	12	11	15	7	6	10	20
---	----	----	----	----	---	---	----	----



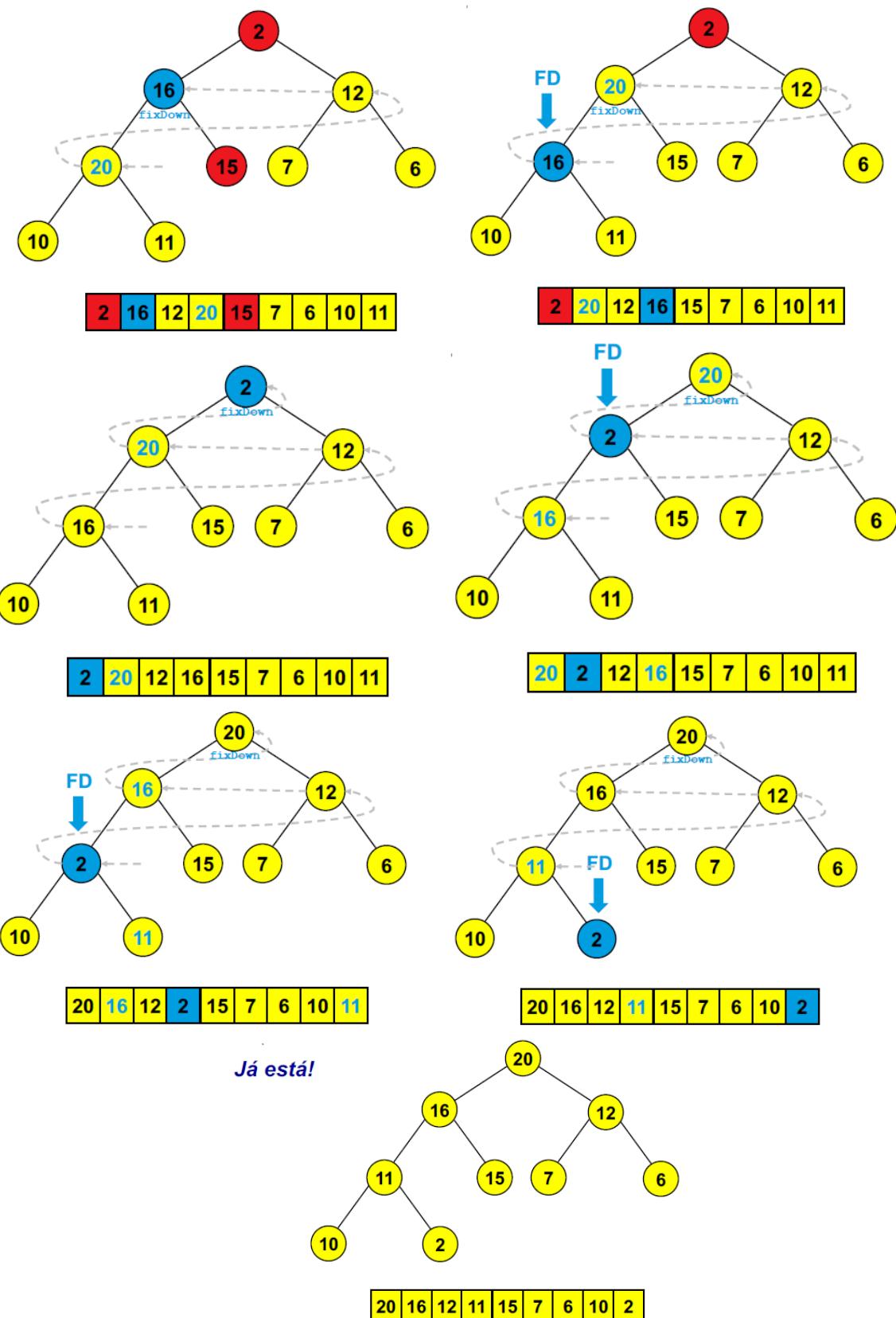
2	16	12	20	15	7	6	10	11
---	----	----	----	----	---	---	----	----



2	16	12	20	15	7	6	10	11
---	----	----	----	----	---	---	----	----



2	16	12	20	15	7	6	10	11
---	----	----	----	----	---	---	----	----



buildheap:

- Chama `fixDown` do último até ao primeiro elemento do vector até todos os elementos cumprirem a heap condition.
- Uma forma mais rápida de o fazer é começar por chamar o `fixDown` ao pai com o índice mais elevado (K=heapsize/2-1), e a todos os índices < K

HeapSort

Como posso partir destas estruturas para ordenar um vector?

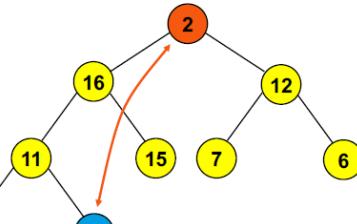
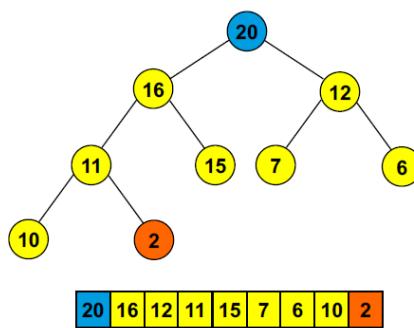


HeapSort - Algoritmo

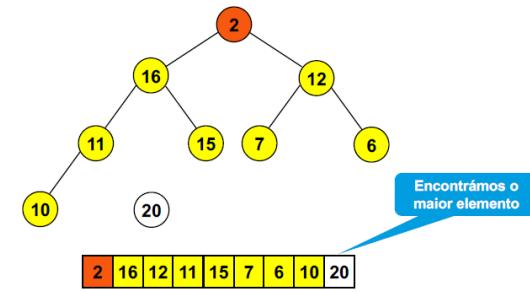
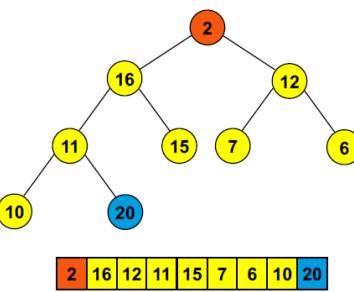
1. transforma o vector num amontoado;
2. troca o **primeiro elemento (raiz)** com o **último**;
3. reduz a dimensão do amontoado em uma unidade;
4. aplica o **fixDown** sobre a nova raiz para reparar o amontoado;

- Trocar o primeiro elemento (raiz) com o último

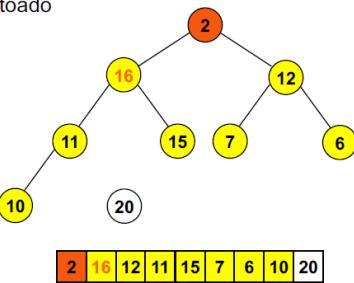
- Trocar o primeiro elemento (raiz) com o último



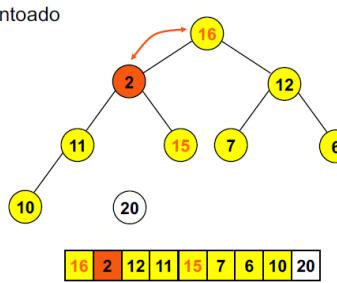
- Reduzir a dimensão do amontoado em uma unidade
- Reduzir a dimensão do amontoado em uma unidade



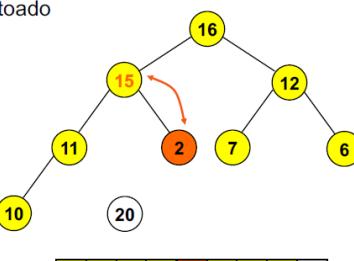
- Executar **fixDown** sobre a nova raiz para reparar o amontoado



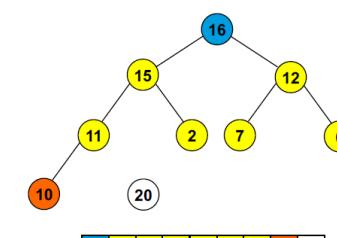
- Executar **fixDown** sobre a nova raiz para reparar o amontoado



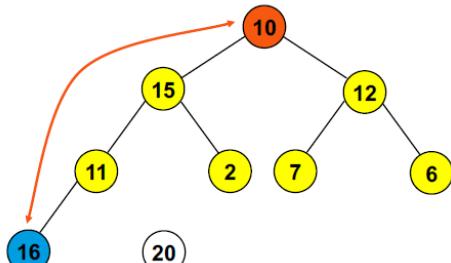
- Executar **fixDown** sobre a nova raiz para reparar o amontoado



- Repetir

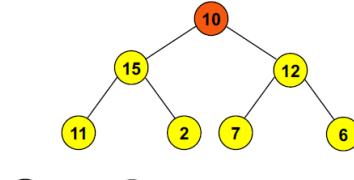


- Repetir



10 | 15 | 12 | 11 | 2 | 7 | 6 | 16 | 20

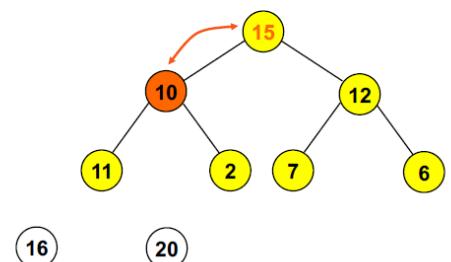
- Repetir



Encontrámos o 2º maior elemento

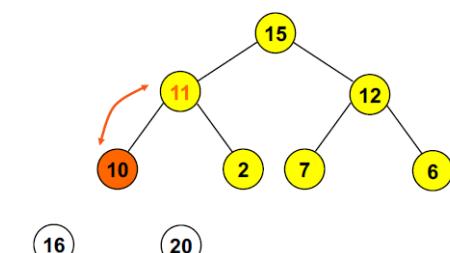
10 | 15 | 12 | 11 | 2 | 7 | 6 | 16 | 20

- Repetir



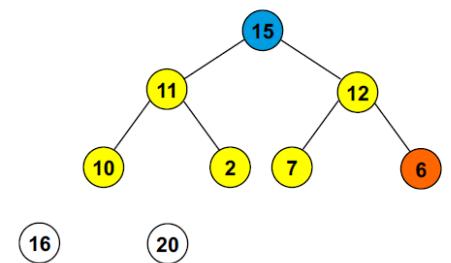
15 | 10 | 12 | 11 | 2 | 7 | 6 | 16 | 20

- Repetir



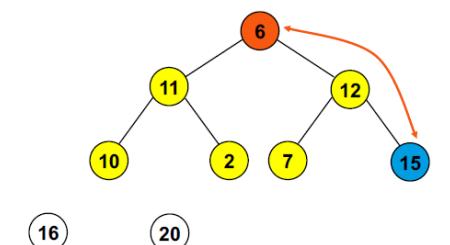
15 | 11 | 12 | 10 | 2 | 7 | 6 | 16 | 20

- Repetir



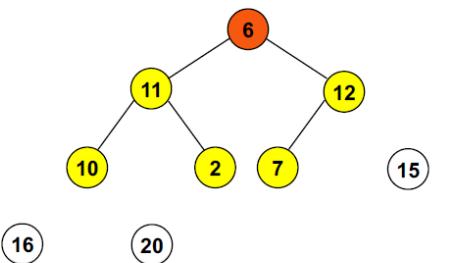
15 | 11 | 12 | 10 | 2 | 7 | 6 | 16 | 20

- Repetir



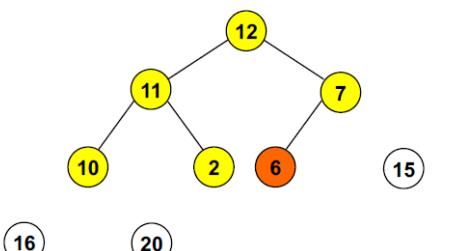
6 | 11 | 12 | 10 | 2 | 7 | 15 | 16 | 20

- Repetir



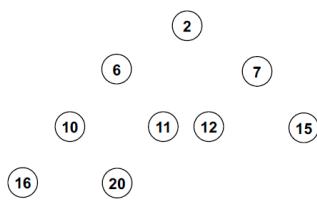
6 | 11 | 12 | 10 | 2 | 7 | 15 | 16 | 20

- Repetir ...



12 | 11 | 7 | 10 | 2 | 6 | 15 | 16 | 20

- No final



2 | 6 | 7 | 10 | 11 | 12 | 15 | 16 | 20

Exercício

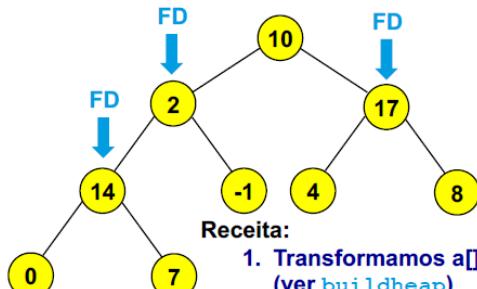
- Considere que se pretende ordenar o vector em baixo, utilizando o algoritmo de ordenação **heapsort**. Qual será a configuração do vector após 2 iterações do heapsort?

a = { 10, 2, 17, 14, -1, 4, 8, 0, 7 }

- **Receita:**

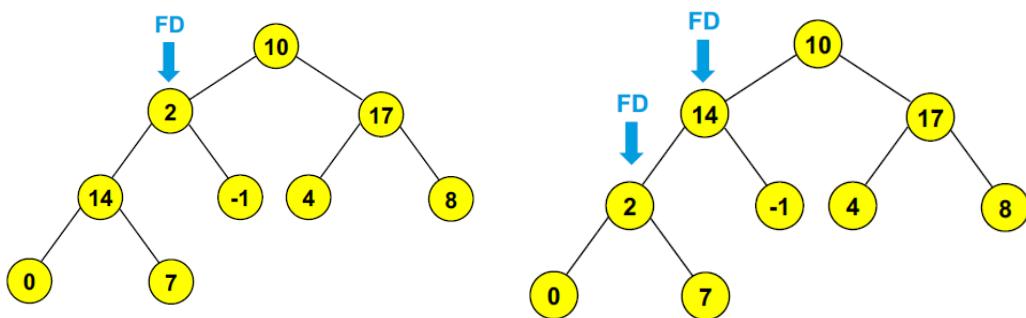
1. Transformamos a[] num amontoado (ver **buildheap**)
2. Trocamos o ultimo elemento com a raiz, e extraímos esse elemento do amontoado.
3. Aplicamos o **fixDown** à raiz
4. Voltamos a 2.

- Começo por transformar o vector num amontoado.

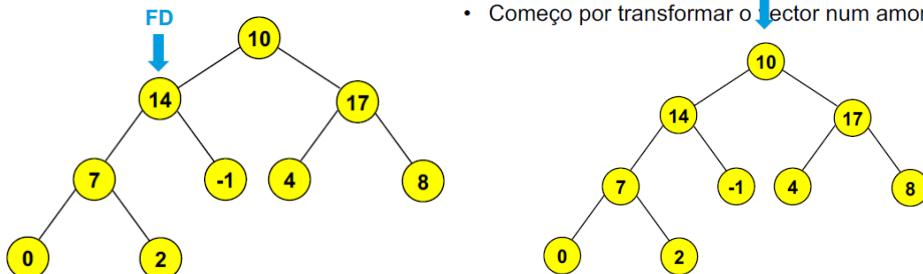


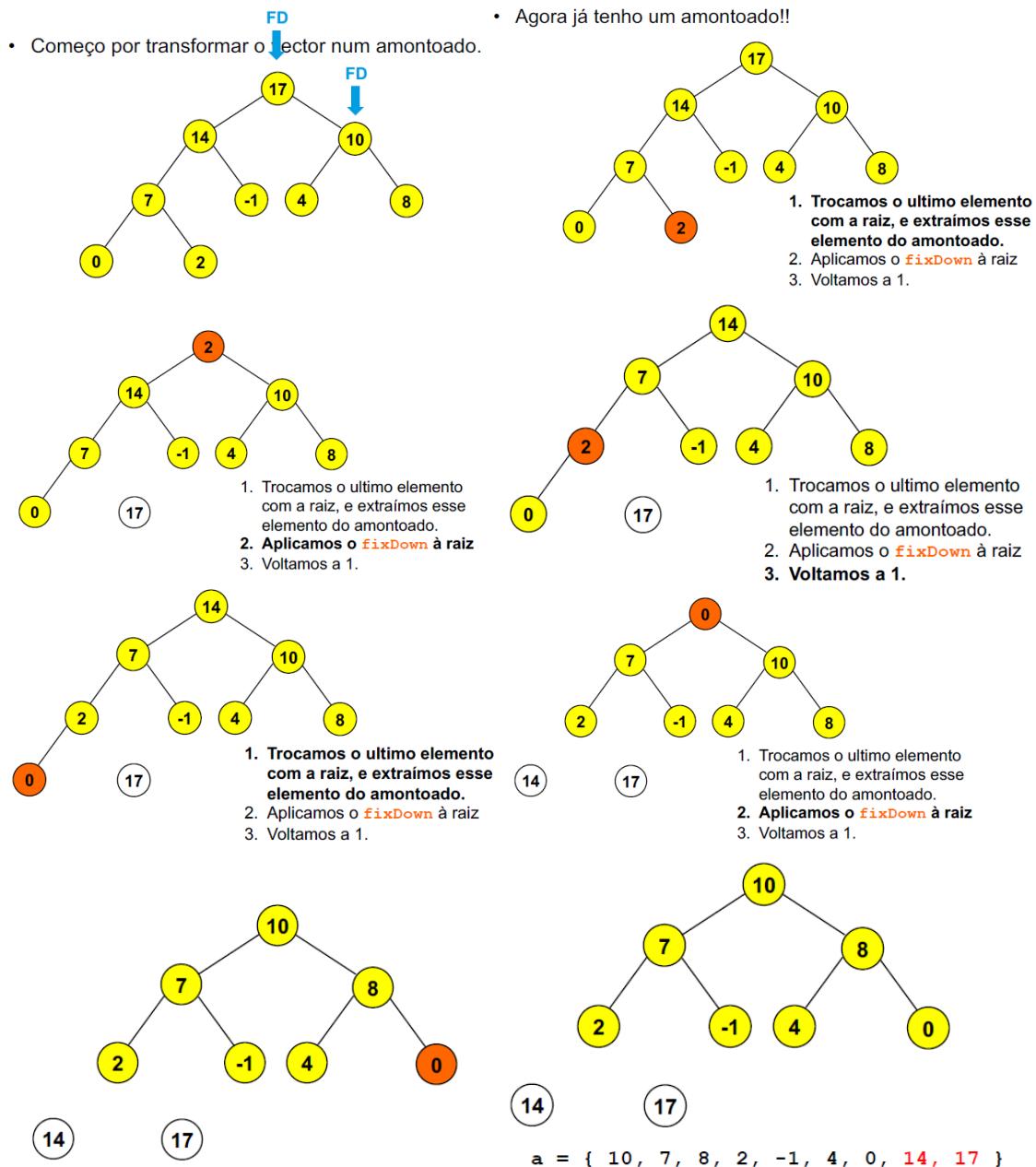
1. Transformamos a[] num amontoado (ver **buildheap**)
2. Trocamos o último elemento com a raiz, e extraímos esse elemento do amontoado.
3. Aplicamos o **fixDown** à raiz
4. Voltamos a 2.

91



- Começo por transformar o vector num amontoado.





Heapsort - Complexidade

- Construção do amontoado (ciclo `for`):
 - $O(N \lg N)$ no pior caso
 - Pode ser provado $O(N)$
- Colocação das chaves (ciclo `while`):
 - $O(N \lg N)$ no pior caso
 - Pode ser provado que para elementos distintos, o melhor caso também é $\Omega(N \lg N)$
- Complexidade no pior caso é $O(N \lg N)$
- Não é estável

Avaliação Experimental

N	Quick	Merge	Heap
12500	2	5	3
25000	7	11	8
50000	13	24	18
100000	27	52	42
200000	58	111	100
400000	122	238	232
800000	261	520	542

Ordenação por Comparação

- Algoritmos de ordenação baseados em comparações são pelo menos $\Omega(N \lg N)$
 - Para N chaves existem $N!$ ordenações possíveis das chaves
 - Algoritmo de ordenação por comparação utiliza comparações de pares de chaves para selecionar uma das $N!$ ordenações
 - Escolher uma folha em árvore com $N!$ folhas
 - Altura da árvore é não inferior $\lg(N!) \approx N \lg N$
- É possível obter algoritmos mais eficientes desde que não sejam apenas baseados em comparações**
- Alternativa:** Utilizar informação quanto às chaves utilizadas
 - Counting Sort
 - Radix Sort

(D) Counting Sort

Ordenação por Comparação

- Algoritmos de ordenação baseados em comparações são pelo menos $O(N \lg N)$
 - Para N chaves existem $N!$ ordenações possíveis das chaves
 - Algoritmo de ordenação por comparação utiliza comparações de pares de chaves para selecionar uma das $N!$ ordenações
 - Escolher uma folha em árvore com $N!$ folhas
 - Altura da árvore é não inferior $\lg(N!) \approx N \lg N$
- É possível obter algoritmos mais eficientes desde que não sejam apenas baseados em comparações**

- Alternativa:** Utilizar informação quanto às chaves utilizadas

Counting Sort - Motivação

- Ordenar $N = r - l + 1$ elementos
- Chaves podem tomar valor inteiro entre 0 e $M - 1$
- Exemplo: $N = 18$, $M = 7$

original

5	2	1	6	5	3	3	4	0	1	2	4	6	0	4	6	3	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

ordenado

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

original

5	2	1	6	5	3	3	4	0	1	2	4	6	0	4	6	3	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

ordenado

0	0																
---	---	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

2x 0's

original

5	2	1	6	5	3	3	4	0	1	2	4	6	0	4	6	3	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

ordenado

0	0	1	1	1													
---	---	---	---	---	--	--	--	--	--	--	--	--	--	--	--	--	--

2x 0's 3x 1's

original

5	2	1	6	5	3	3	4	0	1	2	4	6	0	4	6	3	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

ordenado

0	0	1	1	1	2	2											
---	---	---	---	---	---	---	--	--	--	--	--	--	--	--	--	--	--

2x 0's 3x 1's 2x 2's

original

5	2	1	6	5	3	3	4	0	1	2	4	6	0	4	6	3	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

ordenado

0	0	1	1	1	2	2	3	3	3								
---	---	---	---	---	---	---	---	---	---	--	--	--	--	--	--	--	--

2x 0's 3x 1's 2x 2's 3x 3's

original

5	2	1	6	5	3	3	4	0	1	2	4	6	0	4	6	3	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

ordenado

0	0	1	1	1	2	2	3	3	3	4	4	4					
2x 0's	3x 1's	2x 2's	3x 3's	3x 4's													

original

5	2	1	6	5	3	3	4	0	1	2	4	6	0	4	6	3	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

ordenado

0	0	1	1	1	2	2	3	3	3	4	4	4	4	5	5		
2x 0's	3x 1's	2x 2's	3x 3's	3x 4's	3x 5's												

original

5	2	1	6	5	3	3	4	0	1	2	4	6	0	4	6	3	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

ordenado

0	0	1	1	1	2	2	3	3	3	4	4	4	4	5	5	6	6
2x 0's	3x 1's	2x 2's	3x 3's	3x 4's	2x 5's	3x 6's											

$$N = 18 \quad M = 7$$

int a[]

5	2	1	6	5	3	3	4	0	1	2	4	6	0	4	6	3	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

int cnt[M+1]

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

int b[maxN]

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

Counting Sort: passo 1

$$N = 18 \quad M = 7$$

```
int a[]
```

5	2	1	6	5	3	3	4	0	1	2	4	6	0	4	6	3	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

```
int cnt[M+1]
```

0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---

0 1 2 3 4 5 6 7

```
for (j = 0; j <= M; j++)
    cnt[j] = 0;
```

```
int b[maxN]
```

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

Counting Sort: passo 2

Notar que os 0's são
guardados em `cnt[1]`

$$N = 18 \quad M = 7$$

```
int a[]
```

5	2	1	6	5	3	3	4	0	1	2	4	6	0	4	6	3	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

```
int crt[M+1]
```

0	2	3	2	3	3	3	2	3
---	---	---	---	---	---	---	---	---

0 1 2 3 4 5 6 7

```
for (i = 1; i <= r; i++)
    cnt[a[i]+1]++;
```

```
int b[maxN]
```

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

Começo por registrar
o número de 0's

...depois o número de
1's... etc

Se eu usar este
“histograma” e
transforma-lo na sua
versão “cumulativa”...

$$N = 18 \quad M = 7$$

```
int a[]
```

5	2	1	6	5	3	3	4	0	1	2	4	6	0	4	6	3	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

```
int cnt[M+1]
```

0	2	3	2	3	3	3	2	3
---	---	---	---	---	---	---	---	---

0 1 2 3 4 5 6 7

```
for (i = 1; i <= r; i++)
    cnt[a[i]+1]++;
```

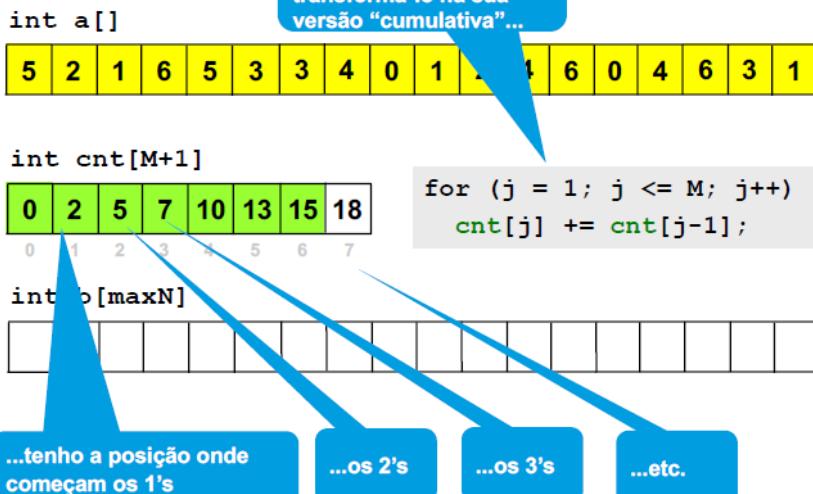
```
int b[maxN]
```

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

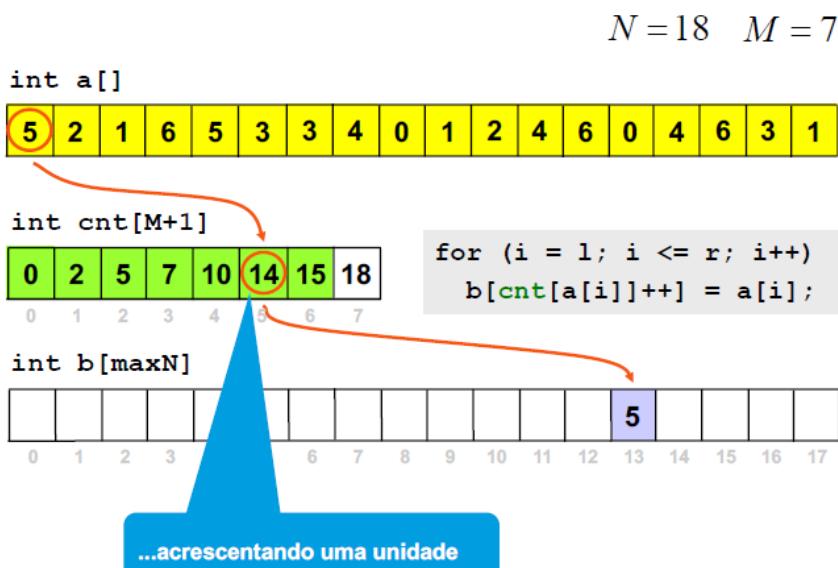
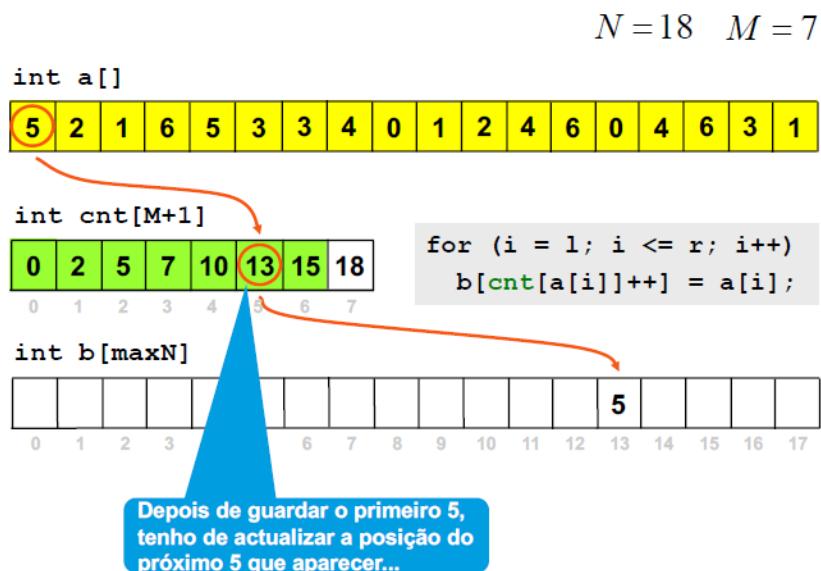
Começo por registrar
o número de 0's

...depois o número de
1's... etc

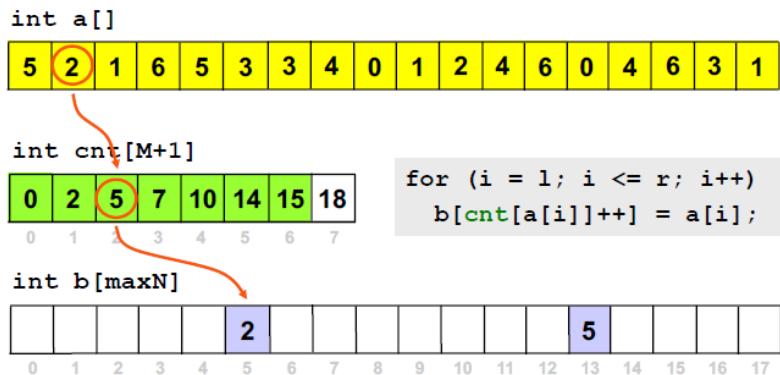
Counting Sort: passo 3



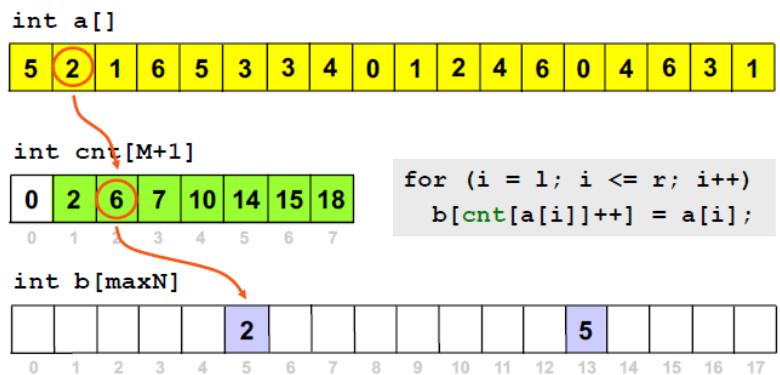
Counting Sort: passo 4



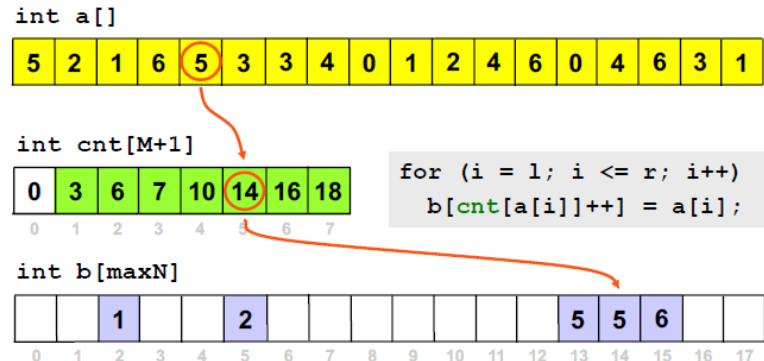
$$N = 18 \quad M = 7$$



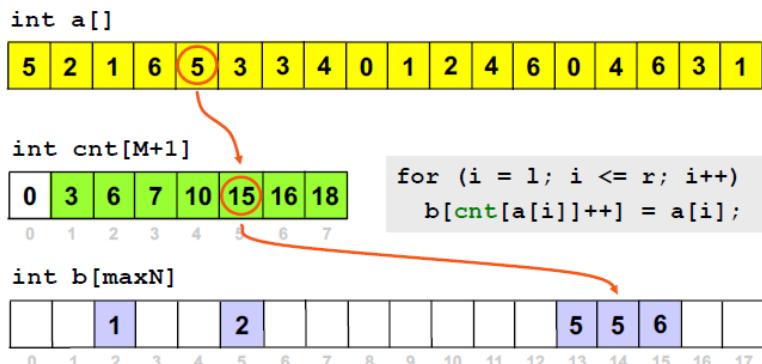
$$N = 18 \quad M = 7$$



$$N = 18 \quad M = 7$$



$$N = 18 \quad M = 7$$



$$N = 18 \quad M = 7$$

```
int a[]
```

5	2	1	6	5	3	3	4	0	1	2	4	6	0	4	6	3	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

```
int cnt[M+1]
```

2	5	7	10	13	15	18	18
---	---	---	----	----	----	----	----

```
for (i = 1; i <= r; i++)
    b[cnt[a[i]]++] = a[i];
```

```
int b[maxN]
```

0	0	1	1	1	2	2	3	3	3	4	4	4	5	5	6	6	6
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

$$0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12 \quad 13 \quad 14 \quad 15 \quad 16 \quad 17$$

$$N = 18 \quad M = 7$$

```
int a[]
```

0	0	1	1	1	1	2	2	3	3	3	4	4	4	5	5	6	6	6
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

```
int cnt[M+1]
```

2	5	7	10	13	15	18	18
---	---	---	----	----	----	----	----

```
for (i = 1; i <= r; i++)
    a[i] = b[i-1];
```

```
int b[maxN]
```

0	0	1	1	1	2	2	3	3	3	4	4	4	5	5	6	6	6
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

$$0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12 \quad 13 \quad 14 \quad 15 \quad 16 \quad 17$$

Counting Sort - Passos

- Utiliza vectores auxiliares `cnt [M + 1]` e `b [maxN]`
- Passo 1: inicializa cada posição de `cnt` a 0
- Passo 2
 - Para cada posição `i` de `a`, faz `cnt[a[i]+1]++`
 - No fim, cada posição `i` de `cnt` tem o número de vezes que a chave `i-1` aparece em `a`
- Passo 3: acumula em cada elemento de `cnt` os elementos anteriores: `cnt[i]` indica a posição ordenada do primeiro elemento com chave `i`
- Passo 4: guarda em `b` os valores de `a` ordenados: `b[cnt[a[i]]++] = a[i]`
- Passo 5: Copia `b` para `a`

Counting Sort - Complexidade

- Complexidade em tempo de execução
 - Dois ciclos relativos à dimensão das chaves M
 - Três ciclos relativos às N chaves
 - Complexidade: $O(N+M)$
- É estável
- Não é *in-place*

(E) Radix sort
(<https://www.youtube.com/watch?v=6du1LrLbDpA>)

Radix Sort - Motivação

- Baseia-se na estrutura dos elementos a ordenar.
- Ordena elementos processando cada dígito/bit/carácter separadamente, usando e.g. o Counting Sort
 - Ex.: organização de strings, tomamos cada letra como um elemento distinto, ou seja, temos $M=26$.
 - Ex.: ordenação de inteiros, tomamos cada dígito como um elemento distinto, ou seja, temos $M=10$.

Primeira versão: **RADIX LSD**
Aplica o Counting Sort sucessivamente dos dígitos menos significativos (*Less Significant Digits*) para os dígitos mais significativos.

LSD Radix Sort

for	tea	tag	ace
tip	ace	tar	ago
ilk	fee	ace	cow
tar	wee	tea	fee
ace	tag	fee	for
fee	ilk	wee	ilk
ago	ago	ago	tag
cow	tip	tip	tar
tag	for	ilk	tea
tea	tar	f or	tip
wee	cow	c pw	wee

LSD Radix Sort (*N* colunas, *M* chars)

				Complexidade?
for	tea	tag	ace	
tip	ace	tar	ago	
ilk	fee	ace	cow	
tar	wee	tea	fee	
ace	tag	fee	for	
fee	ilk	wee	ilk	
ago	ago	ago	tag	
cow	tip	tip	tar	
tag	for	ilk	tea	
tea	tar	for	tip	
wee	cow	cow	wee	

O(N.M)

- Ordena aplicando sucessivamente Counting Sort, dos dígitos menos significativos para os dígitos mais significativos
- Aplicável apenas a chaves de dimensão fixa
- Funciona porque o Counting Sort é estável
 - Preserva as ordenações das iterações anteriores

Exercício

Considere a aplicação do algoritmo **radix sort LSD**, em que cada passo os elementos são ordenados considerando um dígito, ao seguinte vector:

`A = {4327, 5126, 1111, 0721, 1231}`

Qual é o terceiro número da sequência, após o algoritmo ter considerado três dígitos?

4327	1111	1111	1111	
5126	0721	0721	5126	
1111	1231	5126	1231	←
0721	5126	4327	4327	
1231	4327	1231	0721	

4327	1111	1111	1111	0721
5126	0721	0721	5126	1111
1111	1231	5126	1231	1231
0721	5126	4327	4327	4327
1231	4327	1231	0721	5126

Segunda versão: RADIX MSD

Aplica o counting Sort sucessivamente começando pelos dígitos mais significativos (*Most Significant Digits*).

MSD Radix Sort

for	ace	ace	ace
tip	ago	ago	ago
ilk	cow	cow	cow
tar	for	fee	fee
ace	fee	for	for
fee	ilk	ilk	ilk
ago	tip	tar	tag
cow	tar	tag	tar
tag	tag	tea	tea
tea	tea	tip	tip
wee	wee	wee	wee

Radix Sort MSD considera um dígito da chave de cada vez, começando do dígito mais significativo

- Começando no dígito mais significativo (Most Significant Digit), considerar cada n -ésimo dígito e ordenar vector usando apenas esse dígito
- Realização: Para cada dígito, do de maior peso para o de menor peso:
 - Colocar chaves em M caixas (uma para cada valor possível)
 - Ordenar elementos de cada caixa utilizando dígitos subsequentes
 - Se número de elementos não for superior a M , utilizar Insertion Sort
- Resumo: utilizar dígitos, do maior peso para o menor peso e ordenar o vector utilizando o Counting Sort para cada dígito

Binary MSD radix ou Binary Quicksort

Iteração 0		Iteração 1	
32	1 0 0 0 0 0	1	0 0 0 0 0 1
1	0 0 0 0 0 1	9	0 0 1 0 0 1
34	1 0 0 0 1 0	6	0 0 0 1 1 0
9	0 0 1 0 0 1	2	0 0 0 0 1 0
6	0 0 0 1 1 0	20	0 1 0 1 0 0
2	0 0 0 0 1 0	18	0 1 0 0 1 0
20	0 1 0 1 0 0	10	0 0 1 0 1 0
18	0 1 0 0 1 0	32	1 0 0 0 0 0
10	0 0 1 0 1 0	34	1 0 0 0 1 0

Iteração 0									Iteração 2								
32	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1
1	0	0	0	0	0	1	0	0	1	0	0	1	0	0	1	0	1
34	1	0	0	0	1	0	0	0	0	1	1	0	0	0	1	1	0
9	0	0	1	0	0	1	0	0	0	0	1	0	0	1	0	1	0
6	0	0	0	1	1	0	0	0	1	0	1	0	1	0	0	1	0
2	0	0	0	0	1	0	0	1	0	1	0	0	0	1	0	0	0
20	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	0	0
18	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0
10	0	0	1	0	1	0	0	1	0	0	1	0	0	0	1	0	0

Iteração 0									Iteração 3								
32	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0
1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	0
34	1	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0
9	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
6	0	0	0	1	1	0	0	0	1	0	1	0	0	1	0	0	0
2	0	0	0	0	1	0	0	1	0	1	0	0	1	0	0	0	0
20	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	0	0
18	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0
10	0	0	1	0	1	0	0	1	0	0	1	0	0	0	1	0	0

Iteração 0									Iteração 4								
32	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0
1	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	0	0
34	1	0	0	0	1	0	0	0	1	1	0	0	0	1	1	0	0
9	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
6	0	0	0	1	1	0	0	0	1	0	1	0	0	1	0	0	0
2	0	0	0	0	1	0	0	1	0	1	0	0	1	0	0	0	0
20	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	0	0
18	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0
10	0	0	1	0	1	0	0	1	0	0	1	0	0	0	1	0	0

Iteração 0									Iteração 5								
32	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0
1	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	0	0
34	1	0	0	0	1	0	0	0	1	1	1	0	0	1	1	0	0
9	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
6	0	0	0	1	1	0	0	0	1	0	1	0	1	0	1	0	0
2	0	0	0	0	1	0	0	1	0	1	0	0	1	0	0	0	0
20	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	0	0
18	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0
10	0	0	1	0	1	0	0	1	0	0	1	0	0	0	1	0	0

Binary MSD radix ou Binary Quicksort

32	1 0	0 0 0 0
1	0 0	0 0 0 1
34	1 0	0 0 1 0
9	0 0	1 0 0 1
6	0 0	0 1 1 0
2	0 0	0 0 1 0
20	0 1	0 1 0 0
18	0 1	0 0 1 0
10	0 0	1 0 1 0

Não precisamos de considerar apenas 1 bit
(Lab 6)

- Normalmente as instruções são feitas de modo a manipular múltiplos de bits.
- 1 byte = sequência de tamanho fixo de bits (8)
- 1 string = sequência de tamanho variável de bytes
- 1 word = sequência de tamanho fixo de bytes

Avaliação Experimental

- Ordenação de N inteiros (32-bit)

N	Quick	4-bit		8-bit		16-bit
		MSD	LSD	MSD	LSD	LSD
12500	2	7	11	28	4	5
25000	5	14	21	29	8	8
50000	10	49	43	35	18	15
100000	21	77	92	47	39	30
200000	49	133	185	72	81	56
400000	102	278	377	581	169	110
800000	223	919	732	6064	328	219

Exercício

Considere a aplicação do algoritmo **radix sort LSD**, em que cada passo os elementos são ordenados considerando um dígito, ao seguinte vector:

a=
{48372, 62309, 83861, 91874, 18913, 33829,
47812, 95954, 52377, 22394, 56108, 60991}

Qual é o terceiro número da sequência, após o algoritmo ter considerado três dígitos?

56108, 62309, **48372**, 52377, 22394, 47812, 33829,
83861, 91874, 18913, 95954, 60991

Exercício (Verdadeiro ou Falso)

- O algoritmo InsertionSort é $O(N)$.
Falso.
- O algoritmo InsertionSort é $\Theta(N^2)$.
Falso.
- O algoritmo Selection Sort é $\Theta(N^2)$.
Verdadeiro.
- O algoritmo QuickSort é $O(N \log N)$.
Falso.
- O algoritmo Merge Sort é $O(N)$.
Falso
- O algoritmo QuickSort é $\Omega(N \log N)$.
Verdadeiro

- O algoritmo HeapSort é $O(\log N)$

Falso.

Considerando as seguintes afirmações, relativas a algoritmos de ordenação estudados, onde N é a dimensão do conjunto de elementos a ordenar:

1. O algoritmo *selection sort* é $\Theta(N)$.
Falso
2. O algoritmo *insertion sort* é $O(N^2)$.
Verdadeiro
3. O algoritmo *bubble sort* é $\Omega(N)$.
Verdadeiro
4. O algoritmo *quicksort* é $\Omega(N^2)$.
Falso
5. O algoritmo *merge sort* é $O(N \lg N)$.
Verdadeiro

3.10 Algumas aplicações

Problema 1

Dado um vector com n valores, propor um algoritmo eficiente que determina a existência de dois números no vector cuja soma é x .

Solução - Problema 1

- Ordenar vector e resolver problema 2

Complexidade: $O(n \lg n)$

Problema 2

Dado um vector com n valores ordenados, propor um algoritmo eficiente que determina a existência de dois números no vector cuja soma é x .

Solução - Problema 2

- Utilizar dois apontadores colocados inicialmente na primeira (i) e na última (j) posições do vector
- Se soma das posições apontadas é superior a x , decrementar j
- Se soma das posições apontadas é inferior a x , incrementar i
- Se soma das posições apontadas é igual a x , terminar

Complexidade: $O(n)$

Muitos problemas podem ser resolvidos com o auxílio de um algoritmo de ordenação (não necessariamente um dos algoritmos mencionados anteriormente).

- ① Valores distintos. Dado um vetor $v[0..n-1]$ de números inteiros, determinar quantos números distintos há no vetor (ou seja, determinar o cardinal do conjunto de elementos do vetor).
- ② Mediana. Seja $v[0 \dots n-1]$ um vetor de números inteiros, todos diferentes entre si. A mediana do vetor é um elemento do vetor que seja maior que metade dos elementos do vetor e menor que (a outra) metade dos elementos.
- ③ Escalonamento de tarefas. Suponha que n tarefas devem ser processadas em um único processador. Dadas as durações t_1, \dots, t_n das tarefas, em que ordem elas devem ser processadas para minimizar o tempo médio de conclusão de uma tarefa?

Sugestão:

- Implementar outros algoritmos de ordenação, em Python, como por exemplo Mergesort, Shell Sort, Comb sort, Heap sort.
- E se os dados já estão ordenados? Ou ordenados na ordem inversa à pretendida? Verifique o que acontece em termos de tempo de execução para cada algoritmo.

- 1-Aspetos intodutórios
- 2-Funções
- 3-Algoritmos de ordenação e de pesquisa
- 4-Pesquisa linear e pesquisa binária**
- 5- Algoritmos sobre Grafos e Digrafos
- 6-Extensões: noções básicas sobre heurísticas.

4.1 Apresentação do problema

Objectivo

Determinar se um dado elemento figura ou não num vector.

Tipos de pesquisa

- Sequencial
- Binária

4.2 Pesquisa Linear/Sequencial

Estratégia

Percorrer o vector, comparando cada uma das suas componentes com o elemento procurado.

Nota

Especialmente usado quando os elementos do vector não estão ordenados.

4.3 Pesquisa Binária

(<https://www.youtube.com/watch?v=wAy6nDMPYAE>)

(https://www.youtube.com/watch?v=vLS-zRCHo-Y&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=56)

(https://www.youtube.com/watch?v=wAy6nDMPYAE&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=57)

E se o vector estiver ordenado?

Neste caso, não precisamos de percorrer todo o vector em busca do elemento...

Estratégia

Dividir o vector em sucessivas partições, cada vez mais pequenas, restringindo assim, o domínio de pesquisa.

Algoritmo

```
I = 1;  
r = n;  
existe = 0;  
while I ≤ r and existe = 0 do  
    m = ⌊(I + r)/2⌋;  
    if x = am then  
        existe = 1;  
    else  
        if x < am then  
            r = m - 1;  
        else  
            I = m + 1;  
        end  
    end  
end  
if existe = 0 then  
    elemento não foi encontrado;  
else  
    elemento foi encontrado.  
end
```

Desafio 1

- Gerar vectores já ordenados com 100, 1000 e 1000000000 (!) componentes.
- Testar o desempenho dos dois algoritmos.

Desafio 2

- Implementar uma variante da pesquisa binária e linear que conte o número de vezes que um valor figura no vetor.
- Testar o desempenho dos dois algoritmos.

- 1-Aspetos introdutórios
- 2-Funções
- 3-Algoritmos de ordenação e de pesquisa
- 4-Pesquisa linear e pesquisa binária
- 5- Algoritmos sobre Grafos e Digrafos**
- 6-Extensões: noções básicas sobre heurísticas.

5.1 Definições e propriedades fundamentais

5.1.1 Aplicações da Teoria dos Grafos

Leonhard Euler (1707–1783)



Rede Social (Wikipedia): estrutura social composta por pessoas ou organizações, conectadas por um ou vários tipos de relações que compartilham valores e objetivos em comum.



“Rede”

- ① “redes sociais”;
- ② “rede de estradas”;
- ③ “rede de metro”;
- ④ “rede de esgotos”;
- ⑤ “rede de objectos inteligentes”, ou seja, de objectos que comunicam entre si na perspectiva da Internet das Coisas (IoT).¹

¹ IoT, de forma simples, não é mais do que uma extensão da Internet atual, que proporciona aos objetos do dia-a-dia (quaisquer que sejam), desde que possuam capacidade computacional e de comunicação, se conectarem à Internet. A conexão com a rede mundial de computadores permite, primeiro, controlar remotamente os objetos e, segundo, ter acesso aos próprios objetos como fornecedores de serviços.



Como representar, por exemplo, as ligações na rede social *facebook* dos cinco alunos da Unidade Curricular Algoritmos: João, Manuela, Luís, Helena, Pedro, Teresa e Maria?

Como é sabido os matemáticos não podem evitar enumerar... Assim, associe-se a cada aluno, respectivamente, os números 1, 2, 3, 4, 5, 6 e 7. Represente-se cada pessoa pelo seu número dentro de uma circunferência. Para terminar, como representar se dois alunos são amigos "faceboquianos"? Basta traçar uma "linha" que une as duas circunferências! E já está!

Este diagrama, tem um nome: **Grafo**. Estes diagramas poderão ser descritos como um conjunto de pequenos círculos com o seu nome ou número escrito no interior(designados por vértices) ou por um conjunto de pontos. Em qualquer um dos casos, esses vértices, são ligados por segmentos de reta ou linhas curvas (as chamadas *arestas*).

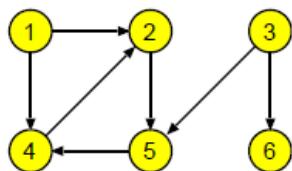
5.1.2 Definições

Grafo

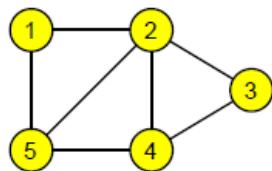
Grafo, $G = (V, E)$, definido por um conjunto V de vértices e um conjunto E de arcos

- Arcos representam ligações entre pares de vértices: $E \subseteq V \times V$
 - Grafo esparso se $|E| << |V \times V|$
 - Caso contrário diz-se que é um grafo denso
- Grafos podem ser dirigidos ou não dirigidos
 - Existência (ou não) da noção de direcção nos arcos

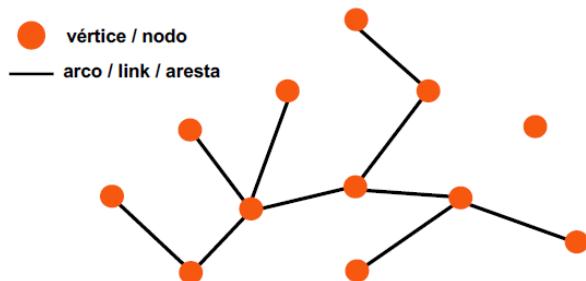
Grafo Dirigido



Grafo Não Dirigido



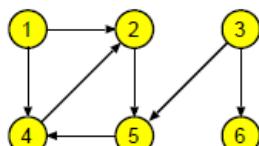
- Constituído por um conjunto V de vértices e E de arcos
 - Arco liga dois vértices
 - Vértice pode estar ligado a qualquer número de outros vértices



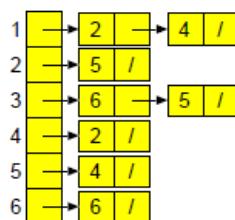
Representação dos arcos

- Matriz de adjacências: arcos representados por matriz
 - Para grafos densos
- Listas de adjacências: arcos representados por listas
 - Para grafos esparsos

Grafo Dirigido



Listas de Adjacências



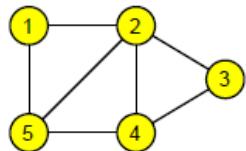
Matriz de Adjacências

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

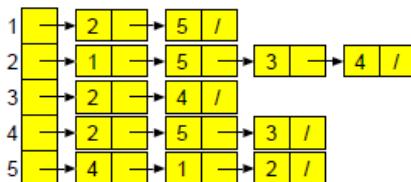
Representação dos arcos

- Matriz de adjacências: arcos representados por matriz
 - Para grafos densos
- Listas de adjacências: arcos representados por listas
 - Para grafos esparsos

Grafo Não Dirigido



Lista de Adjacências



Matriz de Adjacências

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Matriz de Adjacências

- $\Theta(V^2)$ para qualquer grafo

Listas de adjacências

- Tamanho das listas é $|E|$ para grafos dirigidos
- Tamanho das listas é $2|E|$ para grafos não dirigidos
- Tamanho total das listas de adjacências é $O(V + E)$

Grafos pesados

- Existência de uma função de pesos $\omega : E \rightarrow I\mathbb{R}$
- Função de pesos ω associa um peso a cada arco

Grafo

Um *grafo* $G = (V, E)$ é uma estrutura que consiste num conjunto V constituído por um número finito de elementos, designados por *vértices* ou *nós*, e num conjunto E também constituído por um número finito de elementos, designados por *arestas*, de tal forma que cada aresta e está associada a um par de vértices v e w .

Representação de arestas e adjacência de vértices

Escreve-se $e = \{v, w\}$ ou $\{w, v\}$ para indicar que:

- ① e é uma aresta *entre* os nós v e w ;
- ② e é *incidente* em v e w ;
- ③ e liga v e w ;

e diz-se que os vértices v e w são *adjacentes*.

Multigrafo

Um grafo diz-se um *multigrafo* se existe mais do que uma aresta a ligar um par de vértices. Essas arestas dizem-se *arestas múltiplas*.

Tipos de Grafos

Grafos simples e redes

Um grafo diz-se *simples* se não tiver arestas múltiplas nem lacetes. Se, para além disso, a cada aresta estiver associado um número real (correspondente ao custo de passagem por essa aresta, ou ainda à distância entre os dois vértices ligados pela aresta, por exemplo), ao grafo correspondente dá-se o nome de *rede* ou *grafo pesado*.

Lacete

Uma aresta diz-se um *lacete* se ligar um vértice a ele próprio.

Continuando a dar exemplos de redes sociais. Se agora se quiser representar as ligações na rede social *Twitter*.

Do João, Manuela, Luís, Helena, Pedro, Teresa e Maria nem todos os amigos têm paciência para enviar “pequenas mensagens” para certos amigos. Suponha-se que cada amigo é um vértice, tal como foi representado no exemplo da rede social facebook. A relação “o amigo i envia pequenas mensagens para o amigo j ” é representada por uma seta entre o vértices i e j . A esta representação chama-se **Digrafo** ou **Grafo Orientado**.

Redes	Vértices	Ligações	Grafo
Facebook	Friends	Friendship Relations	não orientado
Skype	Contacts	Messages/Conversations	não orientado
Twitter	Users	Follower	orientado
Instagram	Users	Follower	orientado
Tráfego	Cruzamentos	Estradas	orientado

Digrafos

Um *digrafo* (também designado por *grafo dirigido*) é uma estrutura $G = (V, \mathcal{A})$ em que V é o conjunto de vértices e em que os elementos que compõem o conjunto \mathcal{A} são agora pares ordenados de vértices, designados por *arcos*. Escrevemos então $a = (v, w)$ e dizemos que:

- ① a é um arco de v para w ;
- ② o vértice v é *adjacente* ao vértice w ;
- ③ o arco a incide de v para w .

Dois vértices dizem-se *adjacentes* se existir pelo menos um arco de um deles para o outro. Um digrafo diz-se uma *rede dirigida* ou *digrafo pesado* se a cada arco estiver associado um número real.

Grafo associado ao digrafo

Dado um digrafo, podemos construir um grafo com os mesmos vértices do digrafo e em que cada arco é substituído pela aresta que liga os dois vértices envolvidos no arco, obtendo-se assim o denominado *grafo associado ao digrafo*.

Grafo misto

Um grafo $G = (V, E)$ diz-se *misto* se houver pelo menos um elemento de E que é uma aresta e um elemento de E que é um arco.

Grafo bipartido

Um grafo G diz-se *bipartido* se o conjunto dos seus vértices puder ser particionado em dois conjuntos V e W , de tal forma que qualquer aresta ligue um elemento de V com um elemento de W . Neste caso, o grafo é denotado por $G = (V, W, E)$

Digrafo bipartido

Um digrafo G diz-se *bipartido* se o conjunto dos seus vértices puder ser particionado em dois conjuntos V e W , de tal forma que qualquer arco ligue um elemento de V a um elemento de W . Neste caso, o digrafo é denotado por $G = (V, W, \mathcal{A})$

Grafo completo

Um grafo simples com n vértices diz-se *completo* se existir uma aresta a ligar qualquer par de vértices.

Digrafo completo

Um digrafo diz-se *completo* se o grafo que lhe está associado é um grafo completo.

Grafo bipartido completo

Um grafo bipartido $G = (V, W, E)$ diz-se *completo* se existir uma aresta entre qualquer vértice de V e qualquer vértice de W .

Digrafo bipartido completo

Um digrafo bipartido $G = (V, W, \mathcal{A})$ diz-se *completo* se existir um arco entre qualquer vértice de V e qualquer vértice de W .

Subgrafo e clique

Diz-se que o grafo $G' = (V', E')$ é um *subgrafo* de $G = (V, E)$ se o conjunto V' estiver contido no conjunto V e o conjunto E' estiver contido no conjunto E . A um subgrafo completo de um grafo G dá-se o nome de *clique* de G .

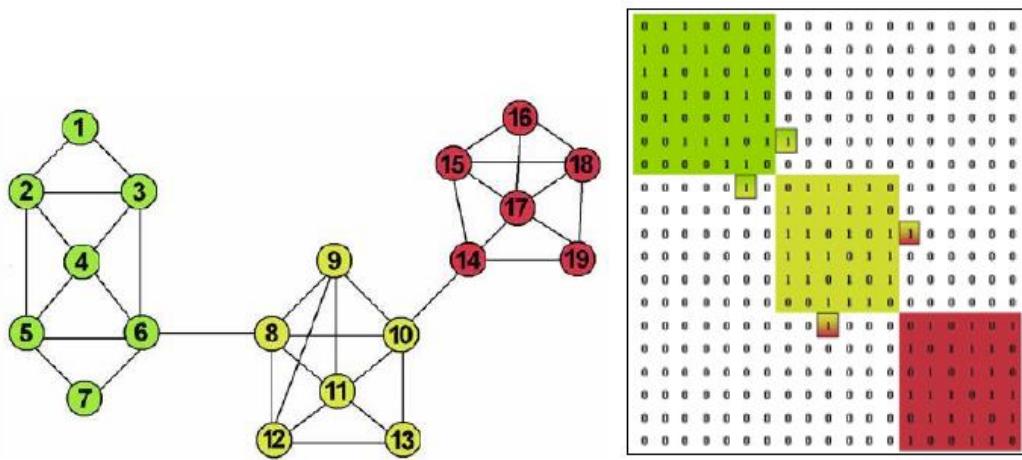
5.2 Matriz de Adjacência e Matriz de Incidência

5.2.1 Matriz de Adjacência de um grafo

Definição

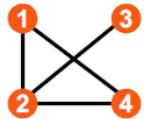
Seja $G = (V, E)$ um grafo sem arestas múltiplas, com $V = \{1, 2, \dots, n\}$. A *matriz de adjacência* de G é a matriz $A = [a_{ij}]$ com n linhas e n colunas, tal que

$$a_{ij} = \begin{cases} 1, & \text{se existir uma aresta que liga os vértices } i \text{ e } j \\ 0, & \text{caso contrário.} \end{cases}$$



Representação: listas

Grafo não orientado



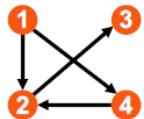
Listas

1	2	4	/
2	1	3	4
3	2	/	
4	1	2	/

Matriz de Adjacências

	1	2	3	4
1	0	1	0	1
2	1	0	1	1
3	0	1	0	0
4	1	1	0	0

Grafo orientado



1	2	4	/
2	3	/	
3	/		
4	2	/	

	1	2	3	4
1	0	1	0	1
2	0	0	1	0
3	0	0	0	0
4	0	1	0	0

Matriz de Adjacências - Vantagens

- Representação mais adequada quando:
 - Há espaço disponível
 - Grafos são densos
 - Algoritmos requerem mais de V^2 operações
- Adição e remoção de arcos é feita de forma eficiente
- Fácil evitar existência de arcos paralelos (repetidos)
- Fácil determinar se dois vértices estão ou não ligados

Matriz de Adjacências - Inconvenientes

- Grafos esparsos de grande dimensão requerem espaço de memória proporcional a V^2
- Neste caso, a simples inicialização do grafo (proporcional a V^2) pode dominar o tempo de execução global do algoritmo
- Para o caso de grafos muito esparsos, mas com um número muito elevado de vértices, pode nem sequer existir memória suficiente para armazenar a matriz

Listas de Adjacências - Vantagens

- Inicialização é proporcional a V
- Utiliza sempre espaço proporcional a $V+E$
 - Adequado para grafos esparsos
 - Algoritmos que assentem na análise de arcos em grafos esparsos.
- Adição de arcos é feita de forma eficiente

Listas de Adjacências - Inconvenientes

- Arcos paralelos e adjacência entre vértices
 - Requer que se pesquise as listas de adjacências, o que pode levar um tempo proporcional a V
- Remoção de arcos
 - Pode levar um tempo proporcional a V (este problema pode ser contornado).
- Não aconselhável para
 - Grafos de grande dimensão que não podem ter arcos paralelos;
 - Grande utilização de remoção de arcos

Representações Alternativas

- Três mecanismos básicos de representação de grafos
 - Vector de arcos (pouco comum)
 - Matriz de adjacências
 - Listas de adjacências
- Produzem diferentes desempenhos ao nível das operações de manipulação
- Escolha deverá depender do problema a resolver

Desempenho das Várias Representações

$E = \text{nº de arestas}$; $V = \text{nº de vértices}$

	Vector de Arcos	Matriz de Adj.	Listas de Adj.
Espaço	$O(E)$	$O(V^2)$	$O(V+E)$
Inicialização	$O(1)$	$O(V^2)$	$O(V)$
Cópia	$O(E)$	$O(V^2)$	$O(E)$
Destrução	$O(1)$	$O(V)$	$O(E)$
Inserir Arco	$O(1)$	$O(1)$	$O(1)$
Encontrar Arco	$O(E)$	$O(1)$	$O(V)$
Remover Arco	$O(E)$	$O(1)$	$O(V)$

Grau de um vértice de um grafo

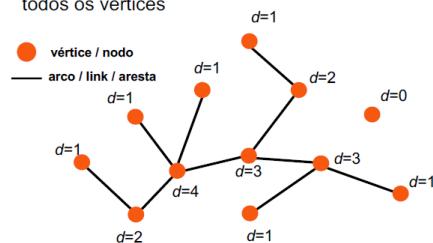
O *grau* de um vértice de um grafo é o número de arestas incidentes nesse vértice. Um vértice diz-se *ímpar* se o seu grau for ímpar, e diz-se *par* se o seu grau for par.

Grau de saída e grau de entrada de um vértice de um digrafo

O *grau de saída* de um vértice de um digrafo é o número de arcos que partem desse vértice, e o *grau de entrada* é o número de arcos que chegam a esse vértice.

Grau de um nodo

- O grau de um vértice (d , *degree*) contabiliza o número de ligações/arcos de um vértice (ou nodo).
- O grau médio de um grafo (z), é a média dos graus de todos os vértices

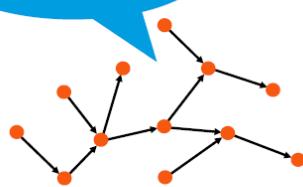


Outras definições

Passamos a ter
um *in-degree* e um
out-degree

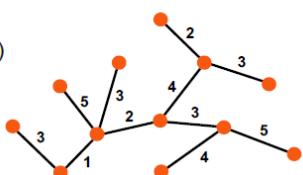
- **Grafo orientado**

- Os arcos podem ter direção



- **Grafo Pesado**

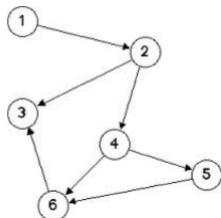
- Os arcos podem ter peso (custo)



Outras definições

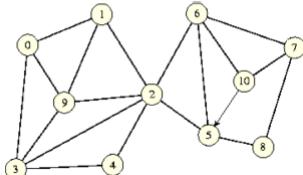
- **Grafos acíclicos orientados**

- para qualquer vértice v , não há nenhum caminho começando e acabando em v .



- **Grafo conexo**

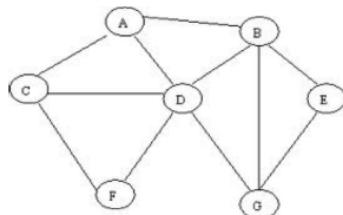
- Para quaisquer vértices v e u , há sempre o caminho a ligar u e v



Outras definições

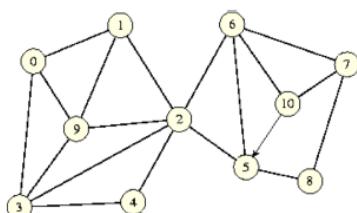
- **Bi-connected graphs**

- para qualquer vértice v , se removermos v , o grafo continua conexo



- **Grafo conexo**

- Para quaisquer vértices v e u , há sempre o caminho a ligar u e v



5.2.2 Matriz de Incidência de um grafo

Definição

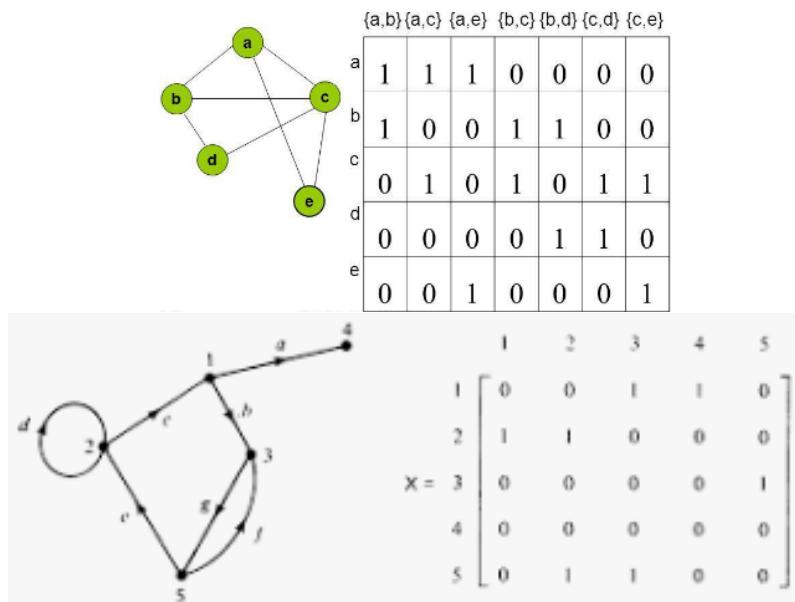
Seja $G = (V, E)$ um grafo, com $V = \{1, 2, \dots, n\}$ e $E = \{e_1, e_2, \dots, e_m\}$. A *matriz de incidência* de G é a matriz $B = [b_{ij}]$ com n linhas e m colunas, em que cada linha corresponde a um vértice e cada coluna corresponde a uma aresta, de tal forma que se e_k é uma aresta de G que liga os vértices i e j , então todos os elementos da coluna k correspondente à aresta e_k são nulos, à exceção dos elementos b_{ik} e b_{jk} , que são ambos iguais a 1 ($b_{ik} = b_{jk} = 1$).

Definição

Seja $G = (V, A)$ um digrafo sem lacetes, com $V = \{1, 2, \dots, n\}$ e $A = \{a_1, a_2, \dots, a_m\}$. A *matriz de incidência* de G é a matriz $B = [b_{ij}]$ com n linhas e m colunas, em que cada linha corresponde a um vértice e cada coluna corresponde a um arco, de tal forma que se a_k é um arco de G que incide do vértice i para o vértice j , então todos os elementos da coluna k correspondente ao arco a_k são nulos, à exceção dos elementos b_{ik} e b_{jk} , sendo $b_{ik} = -1$ e $b_{jk} = 1$.

Lacete

À semelhança dos grafos, um arco de um digrafo diz-se um *lacete* se ligar um vértice a ele próprio.



5.2.3 Alguns resultados teóricos

Propriedade (Primeiro Teorema da Teoria de Grafos)

Se G é um grafo ou um multigrafo sem lacetes com m arestas, então a soma dos graus dos vértices é igual a $2m$.

Propriedade

O número de vértices ímpares num grafo ou num multigrafo sem lacetes é par.

5.3 Ligações em grafos

Caminho

Um *caminho* entre dois vértices v e w de um grafo é uma sequência com um número finito de vértices e de arestas da forma $v_1, e_1, v_2, e_2, v_3, e_3, \dots, v_k, e_k, v_{k+1}$, com $v_1 = v$ e $v_{k+1} = w$, em que e_k é a aresta do caminho que une os vértices v_k e v_{k+1} .

Caminho simples

Um caminho diz-se *simples* se todos os seus vértices são distintos.

Grafo conexo

Um grafo diz-se *conexo* se existir um caminho que liga qualquer par de vértices do grafo.

Caminho fechado, circuito e ciclo

Um caminho que comece num vértice e acabe nesse mesmo vértice diz-se um *caminho fechado*. Um caminho fechado em que todas as arestas sejam distintas diz-se um *círculo*. Um círculo em que todos os vértices sejam distintos diz-se um *ciclo*.

5.4 Conexões em digrafos

Caminho dirigido

Um *caminho dirigido* do vértice v para o vértice w de um digrafo é uma sequência com um número finito de vértices e de arcos da forma $v_1, a_1, v_2, a_2, \dots, v_k, a_k, v_{k+1}$ com $v_1 = v$ e $v_{k+1} = w$, em que a_k é o arco do vértice v_k para o vértice v_{k+1} . Neste caso, dizemos que existe uma *conexão de v para w* .

Vértices fortemente e fracamente conectados

Um par de vértices diz-se *fortemente conectado* se existir uma conexão de cada um deles para o outro. Se existir conexão de apenas um deles para o outro, então teremos um par *fracamente conectado*.

Digrafos fortemente e fracamente conexos

Um digrafo diz-se *fortemente conexo* se cada par de vértices for fortemente conectado, e *fracamente conexo* se cada par de vértices for fracamente conectado.

5.5

Orientações em grafos

Definição

Se G é um grafo, então o digrafo que se obtém substituindo cada aresta de G por um arco é denominado de *orientação de G* .

Grafos fortemente e fracamente orientáveis

Um grafo diz-se *fortemente orientável* se admitir uma orientação fortemente conexa, e *fracamente orientável* se só admitir uma orientação fracamente conexa.

Propriedade (Teorema de Robbins)

Um grafo é fortemente orientável *se e só se* é conexo e não tem pontes.

Ponte

Uma aresta de um grafo conexo diz-se uma *ponte* se a remoção dessa aresta transformar o grafo num grafo desconexo.

Conectividade de um grafo

Algoritmo de pesquisa primária DFS (depth-first search)

Sejam v_1, v_2, \dots, v_n os vértices de um grafo G .

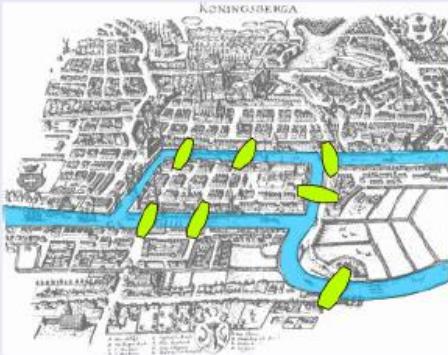
- 1 Começemos por um vértice qualquer arbitrariamente escolhido e atribuamos-lhe a **etiqueta 1**.
- 2 Escolhemos um qualquer vértice adjacente a 1. Como este vértice ainda não foi etiquetado, atribuamos-lhe a etiqueta 2.
- 3 Marquemos a aresta $\{1, 2\}$ como **usada**, por forma a que não possa voltar a ser usada.
- 4 Procedendo desta forma, suponhamos que em dado ponto atribuimos a etiqueta k ao vértice v_i .
- 5 Percorramos todos os vértices não etiquetados adjacentes a v_i , e atribuamos a um deles a etiqueta $k + 1$.
- 6 Marquemos a aresta $\{k, k + 1\}$ como usada.
- 7 Se todos os vértices adjacentes ao vértice k já estiverem etiquetados, regressemos novamente ao vértice $k - 1$ e escolhemos um vértice seu adjacente ainda não etiquetado para a partir dele repetir o processo.
- 8 Se, pelo contrário, existir ainda um vértice adjacente ao vértice k ainda não etiquetado, etiquetemos esse vértice com a etiqueta $k + 1$ e marquemos a aresta $\{k, k + 1\}$ como já tendo sido usada.
- 9 Continuaremos este processo, até que todos os vértices tenham sido etiquetados **ou** até que tenhamos regressado ao vértice 1 com pelo menos um vértice por etiquetar.
- 10 No **primeiro caso**, o grafo é **conexo**; no **segundo caso**, o grafo é **desconexo**.

Desafio

Implementar este algoritmo em Python.

O problema das pontes de Konigsbergh

Representação da cidade de Königsberg, no tempo de Euler

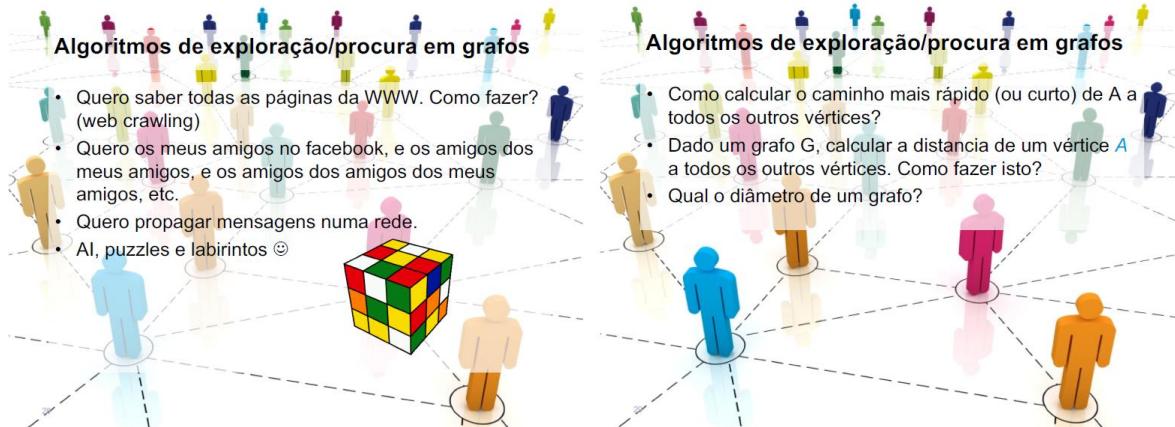


Königsberg, cidade russa actualmente denominada Kalinínegrado, situa-se nas margens do rio Pregel e tem uma margem norte (N), uma margem sul (S), uma ilha a oeste (W) e uma ilha a leste (E).

Será possível, começando em qualquer uma das sete pontes assinaladas na figura, regressar ao ponto de partida, mas passando por cada uma das restantes pontes exactamente uma única vez?

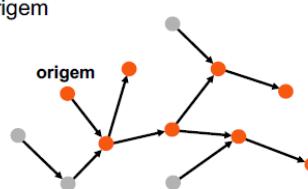
5.2. ALGORITMOS DE EXPLORAÇÃO/PROCURA EM GRAFOS

(<https://www.youtube.com/watch?v=pcKY4hjDrxk>)



Algoritmos de procura em grafos

- Dado um vértice origem/fonte
- Visitar todos os vértices atingíveis a partir da origem
 - Todos os vértices que estão em qualquer caminho do grafo que comece na origem



- A ordem pela qual os vértices são visitados depende do tipo de procura

- **Procurar em grafos é equivalente a percorrer labirintos**
 - Necessário marcar pontos já visitado
 - Ser-se capaz de recuar, num caminho efectuado, até ao ponto de partida
 - Os vários algoritmos de procura em grafos limitam-se a executar uma determinada estratégia de procura em labirintos
 - **Procura em profundidade** primeiro (DFS – “Depth-first-search”)
 - Admite duas implementações: recursiva e com uso de pilha explícita
 - Substituindo a pilha por uma fila FIFO, transforma-se em **Procura em largura** primeiro (BFS – “Breadth-first-search”)

PROCURA EM PROFUNDIDADE PRIMEIRO (DFS = Depth-First Search)

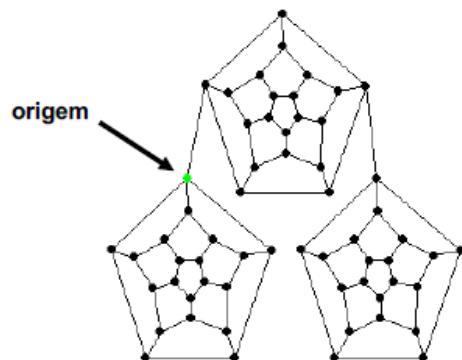
(https://www.youtube.com/watch?v=pcKY4hjDrxk&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=61)

(https://www.youtube.com/watch?v=kyUpc_5705s&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=81)

- Qual a melhor forma de ultrapassar um labirinto?



- Como visitar todos os recantos apenas uma vez?
- Visita primeiro os arcos dos vértices mais recentemente visitados



- Grafo pesquisado dando prioridade aos arcos dos vértices mais recentemente visitados
- Resultado da procura:
 - Floresta DF: $G_\pi = (V, E_\pi)$ $E_\pi = \{(\pi[v], v) : v \in V \wedge \pi[v] \neq NIL\}$
 - Floresta DF composta por várias árvores DF
- Implementação:
 - $\text{color}[u]$: cor do vértice (branco, cinzento, preto)
 - $d[u]$: tempo de início (de visita do vértice)
 - $f[u]$: tempo de fim (de visita do vértice)
 - $\pi[u]$: predecessor



Grafo pesquisado dando prioridade aos arcos dos vértices mais recentemente visitados

Implementação

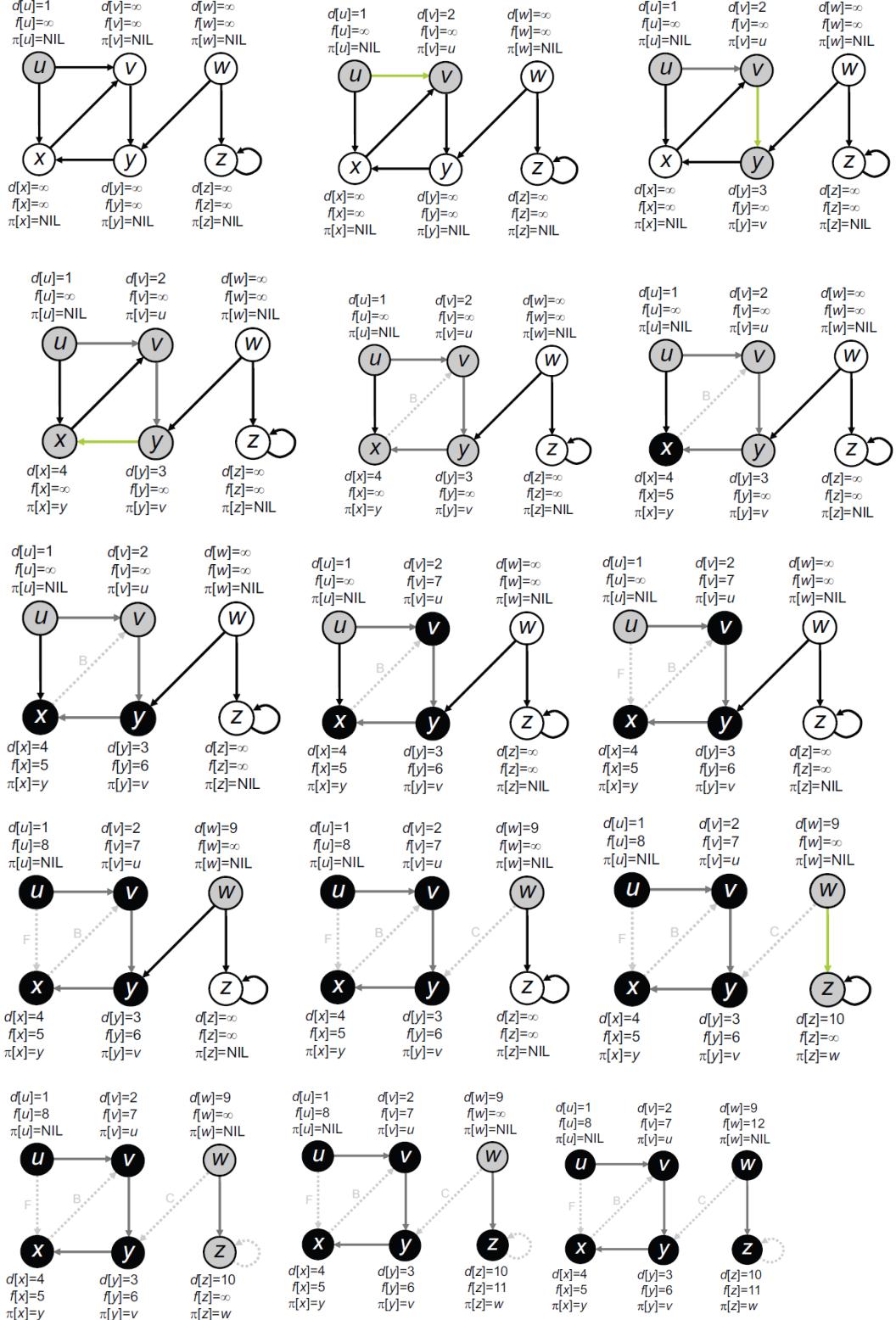
- $\text{color}[v]$: cor do vértice v , (branco, cinzento ou preto)
- $\pi[v]$: predecessor de v na árvore DF
- $d[v]$: tempo de início (de visita do vértice)
- $f[v]$: tempo de fim (de visita do vértice)

Pseudo-código

```
DFS( $G$ )
1 for each vertex  $u \in V[G]$ 
2   do  $\text{color}[u] \leftarrow \text{white}; \pi[u] = NIL$ 
3  $time \leftarrow 1$ 
4 for each vertex  $u \in V[G]$ 
5   do if  $\text{color}[u] = \text{white}$ 
6     then DFS-Visit( $u$ )
```

```
DFS-Visit( $u$ )
1  $\text{color}[u] \leftarrow \text{gray}$ 
2  $d[u] \leftarrow time$ 
3  $time \leftarrow time + 1$ 
4 for each  $v \in Adj[u]$ 
5   do if  $\text{color}[v] = \text{white}$ 
6     then  $\pi[v] \leftarrow u$ 
7     DFS-Visit( $v$ )
8  $\text{color}[u] \leftarrow \text{black}$ 
9  $f[u] \leftarrow time$ 
10  $time \leftarrow time + 1$ 
```

Exemplo (leitura das iterações em linha)



Complexidade

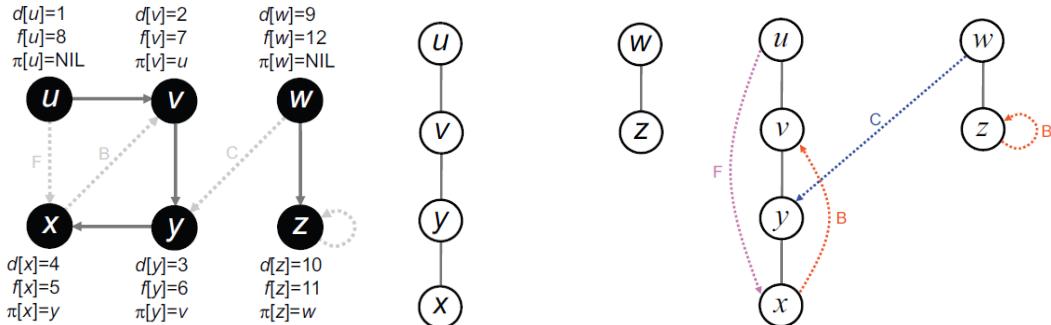
Tempo de execução: $O(V + E)$

- Inicialização: $O(V)$
- Chamadas a DFS-Visit dentro de DFS: $O(V)$
- Arcos analisados em DFS-Visit: $\Theta(E)$
 - Chamadas a DFS-Visit dentro de DFS-Visit: $O(V)$
 - Mas $\sum_{v \in V} |Adj[v]| = \Theta(E)$

Resultados

Floresta Depth-First (DF)

- $G_\pi = (V, E_\pi)$
- $E_\pi = \{(\pi[v], v) : v \in V \wedge \pi[v] \neq NIL\}$
- Floresta DF composta por várias árvores DF

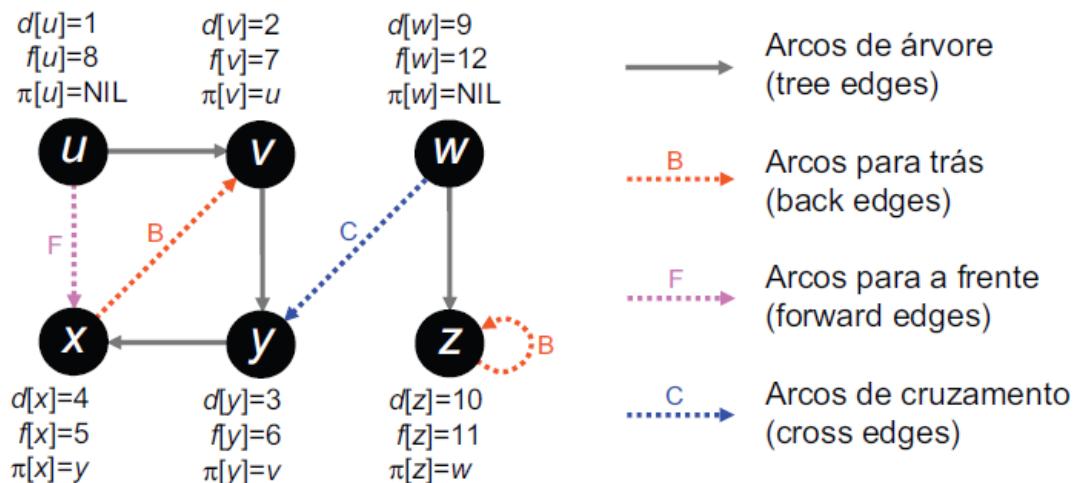


Classificação de arcos (u, v)

- Arcos de árvore: (tree edges)
 - arcos na floresta DF, G_π
 - (u, v) é arco de árvore se v foi visitado devido ao arco (u, v) ser visitado
- Arcos para trás: (back edges)
 - ligam vértice u a vértice v antecessor na mesma árvore DF
- Arcos para a frente: (forward edges)
 - ligam vértice v a vértice descendente na mesma árvore DF
- Arcos de cruzamento: (cross edges)
 - Na mesma árvore DF, se u (ou v) não antecessor de v (ou u)
 - Entre árvores DF diferentes

Classificação de arcos (u, v)

- Arcos de árvore: (tree edges)
 - $d[u] < d[v] < f[v] < f[u]$
 - $\text{color}[v] = \text{white}$ quando (u, v) é analisado
- Arcos para trás: (back edges)
 - $d[v] < d[u] < f[u] < f[v]$
 - $\text{color}[v] = \text{gray}$ quando (u, v) é analisado
- Arcos para a frente: (forward edges)
 - $d[u] < d[v] < f[v] < f[u]$
 - $\text{color}[v] = \text{black}$ quando (u, v) é analisado
- Arcos de cruzamento: (cross edges)
 - $d[v] < f[v] < d[u] < f[u]$
 - $\text{color}[v] = \text{black}$ quando (u, v) é analisado



Resultados

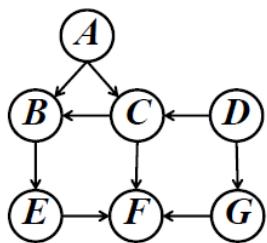
Dado $G = (V, E)$, não dirigido, cada arco é arco de árvore ou arco para trás

- i.e., não existem arcos para a frente e de cruzamento

Numa floresta DF, v é descendente de u se e só se quando u é descoberto existe um caminho de vértices brancos de u para v

- v descendente de $u \Rightarrow$ existe caminho de vértices brancos de u para v
 - Qualquer vértice w descendente de u verifica $[d[w], f[w]] \subset [d[u], f[u]]$, pelo que w é branco quando u é descoberto

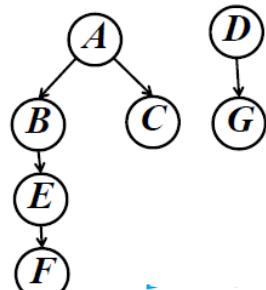
- Exemplo (origem: 'A'):



Lista adjacente (ordem alfab.):

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

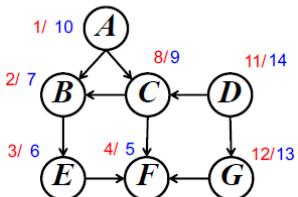
O DFS depende da ordem das listas



Sequência: A B E F C D G

Predecessor: nil A B E A nil D

- Exemplo (origem: 'A'):



Lista adjacente (ordem alfab.):

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

O DFS depende da ordem das listas

Sequência: A B E F C D G

Tempo de descoberta: 1 2 3 4 8 11 12

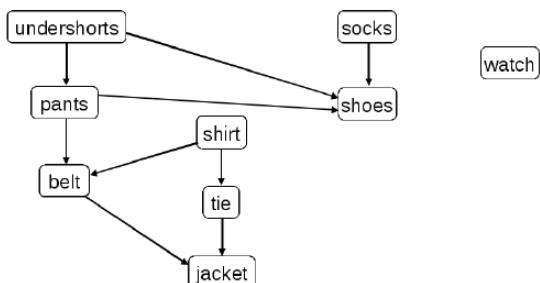
Tempo de fim: 10 7 6 5 9 14 13

• Aplicações

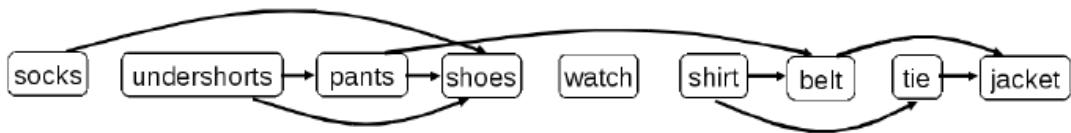
- Visitar todos os elementos de um grafo, sendo que cada vértice é visitado apenas 1 vez!
- Encontrar um caminho entre 2 nodos específicos, u e v, num grafo sem pesos ou com pesos (grafo pesado).
- Saber se um grafo é conexo ou não.
- Saber se um grafo tem ciclos
- Útil para saber a ordenação topológica de um grafo (next class)
- Identificação de componentes fortemente ligadas (next class)

1 Motivação

Grafo que representa um conjunto de dependências ou precedências:



Ordenação Topológica:



2 Definições

Caminhos em Grafos

Dado um grafo $G = (V, E)$, um **caminho** p é uma sequência $\langle v_0, v_1, \dots, v_k \rangle$ tal que para todo o i , $0 \leq i \leq k - 1$, $(v_i, v_{i+1}) \in E$

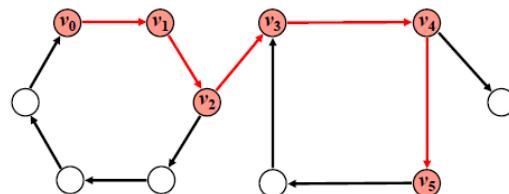
- Se existe um caminho p de u para v , então v diz-se **atingível** a partir de u usando p
- Um **ciclo** num grafo $G = (V, E)$ é um caminho $\langle v_0, v_1, \dots, v_k \rangle$, tal que $v_0 = v_k$
- Um grafo dirigido $G = (V, E)$ diz-se **acíclico** (ou Directed Acyclic Graph (DAG)) se não tem ciclos.

Caminhos, circuitos e ciclos

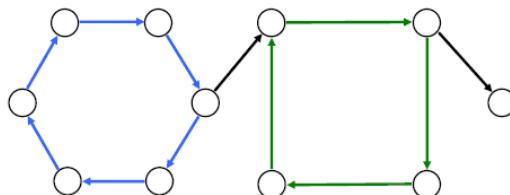
- Dado grafo $G = (V, E)$, um **caminho** p é uma sequência

$\langle v_0, v_1, \dots, v_k \rangle$

tal que para todo o i , $0 \leq i \leq k-1$, $(v_i, v_{i+1}) \in E$



– No grafo em baixo existem 2 ciclos (azul e verde)



- Um grafo dirigido $G = (V, E)$ é **acíclico** se não tem ciclos
 - Directed Acyclic Graph (DAG)
 - Para qualquer vértice v de um DAG, não há nenhum caminho que comece e acabe em v

Caminhos Eulerianos e Circuitos Eulerianos

Definições

Um caminho num grafo diz-se *euleriano* se cada aresta do grafo figurar no caminho exactamente uma vez. Um caminho euleriano fechado diz-se um *círculo euleriano*. Um grafo que contenha um circuito euleriano diz-se um *grafo euleriano*.

Observação

Definições idênticas às anteriores podem ser formuladas para digrafos.

Propriedade

Um grafo conexo G é euleriano se e só se o grau de cada vértice for par.

Problema das Pontes de Königsberg

No Problema das Pontes de Königsberg como nenhum vértice possui grau par, temos que este grafo não é euleriano e portanto, é impossível efetuar tal percurso.

Determinação de um circuito Euleriano

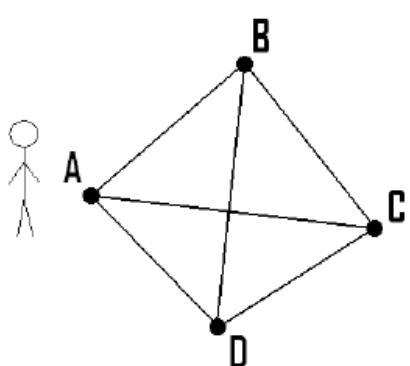
Algoritmo

- 1 Começemos por um vértice v qualquer arbitrariamente escolhido, e construamos um circuito C_1 com início e fim em v .
- 2 Se este circuito contiver todas as arestas do grafo G , então é euleriano e **o algoritmo termina** com um grafo euleriano.
Caso contrário:
 - 3 Apagaremos todas as arestas deste circuito e todos os vértices de grau zero de G , obtendo-se assim um subgrafo conexo H_1 cujos vértices têm também grau par.
 - 4 Escolhamos um vértice u comum ao circuito C_1 e ao subgrafo H_1 .
 - 5 Começando em u , construamos um circuito C_2 que regressa a u atravessando arestas distintas de H_1 .
 - 6 Se $v = u$, então os dois circuitos C_1 e C_2 podem ser associados num só, formando um circuito alargado C_3 .
 - 7 Se $v \neq u$, sejam P e Q dois caminhos simples ligando u e v , formados por arestas de C_1 . Então P , Q e C_2 podem ser reunidos de modo a formar um novo circuito C_3 .
 - 8 Se C_3 contiver todas as arestas de G , então C_3 é um circuito euleriano, e **o algoritmo termina** com um grafo euleriano.
 - 9 Se C_3 ainda não contiver todas as arestas do grafo, repetir todo o processo.

O problema do caixeiro viajante

Especificação do problema

Será possível partir de uma cidade qualquer e regressar a essa mesma cidade, depois de passar por todas as restantes cidades, mas visitando cada uma delas apenas uma única vez?



O Problema do Caixeiro Viajante (PCV) ou Travelling Salesman Problem (TSP) é o nome que usualmente é dado a uma série de problemas reais importantes que podem ser modelados em termos de ciclos Hamiltonianos em grafos completos. Este é um dos mais populares problemas de Optimização Combinatória e de Optimização em Redes. É comum encontrar na categoria de problemas PCV os problemas seguintes:

- rotas de autocarros escolares;
- redes de distribuição postal;
- construção de placas de circuitos integrados;
- programação de máquinas para realizar várias tarefas em sucessão.

Caminhos Hamiltonianos e Circuitos Hamiltonianos

(https://www.youtube.com/watch?v=dQr4wZCiJJ4&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=67)

Definições

Um caminho entre dois vértices de um grafo diz-se *hamiltoniano* se passar por cada vértice do grafo exactamente uma vez. Um caminho fechado que passe por cada vértice exactamente uma vez e cujas arestas sejam todas distintas diz-se um *ciclo hamiltoniano*. Um grafo que contenha um ciclo hamiltoniano diz-se um *grafo hamiltoniano*.

Propriedade (Teorema de Ore)

Um grafo simples com n vértices ($n \geq 3$) é hamiltoniano se a soma dos graus de cada par de vértices não adjacentes for igual ou superior a n .

Propriedade (Teorema de Dirac)

Se o grau de cada vértice de um grafo simples com n vértices for igual ou superior a $\frac{n}{2}$, então o grafo é hamiltoniano.

Propriedade (Corolário do teorema de Dirac)

Um grafo simples com n vértices tem um caminho hamiltoniano se a soma dos graus de qualquer par de vértices não adjacentes for igual ou superior a $n - 1$.

Observação

Definições idênticas às anteriores podem ser formuladas para digrafos.

3 Ordenação Topológica

Ordenação Topológica

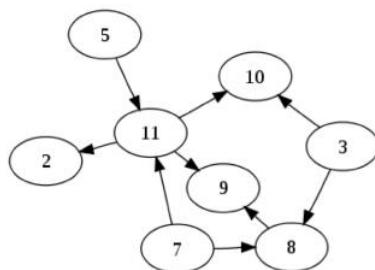
Uma **ordenação topológica** de um DAG $G = (V, E)$ é uma ordenação de todos os vértices tal que se $(u, v) \in E$ então u aparece antes de v na ordenação

Soluções Algorítmicas

- Eliminação de vértices
- Utilizando informação de DFS

Ordenação topológica

- Grafos como representação de um problema de planeamento de tarefas



- Se um grafo tem um arco de X para Y, então a tarefa X deve ser efectuada antes de Y.
- Uma ordenação topológica de um grafo é uma sequência de "tarefas" onde esta condição é verificada.
- Uma **ordenação topológica** de um DAG $G = (V, E)$ é uma ordenação de todos os vértices tal que se $(u, v) \in E$ então u aparece antes de v na ordenação
- Por outras palavras, se um grafo tem um arco dirigido $u \rightarrow v$, então u vem antes de v na ordenação topológica
- Algoritmos
 - Utilizando informação de DFS
 - Eliminação de vértices (não vamos estudar)

Topological-Sort(G)

1. Inicializo as cores, os predecessores e os tempos d e f de todos os nodos, tal como no DFS.
2. Chama o DFS-visit a todos os nodos com $\text{in-degree}=0$, e vou calculando o *finishing-time* (f) para cada nodo.
3. sempre que um vértice é processado (passa a preto) e recebe um *finishing-time*, insiro o vértice no início de uma lista ligada.
4. Retorno a lista ligada

Pseudo-Código Algoritmo Eliminação de Vértices

Topological-Sort-1(G)

```
1  $L \leftarrow \emptyset$                                 ▷ Lista de vértices
2  $Q \leftarrow \emptyset$                                 ▷ Fila de vértices (FIFO)
3 for each  $v \in G$                             ▷ Inicialização  $O(V)$ 
4     do if  $v$  sem arcos de entrada  $(w, v)$ 
5         then Enqueue( $Q, v$ )
6 while  $Q \neq \emptyset$                                 ▷ Ciclo Principal  $O(V + E)$ 
7     do  $u = \text{Head}(Q)$ 
8         Eliminar todos os arcos  $(u, v)$ 
9         if  $v$  sem arcos de entrada  $(w, v)$ 
10            then Enqueue( $Q, v$ )
11        Dequeue( $Q$ )
12        Colocar  $u$  no fim da lista  $L$ 
13 return  $L$ 
```

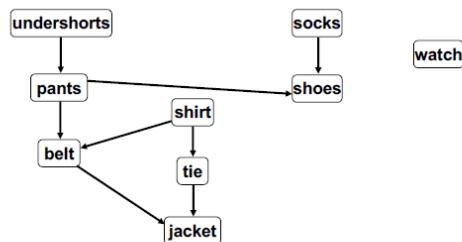
Pseudo-Código Algoritmo Utilizando informação de DFS

Topological-Sort-2(G)

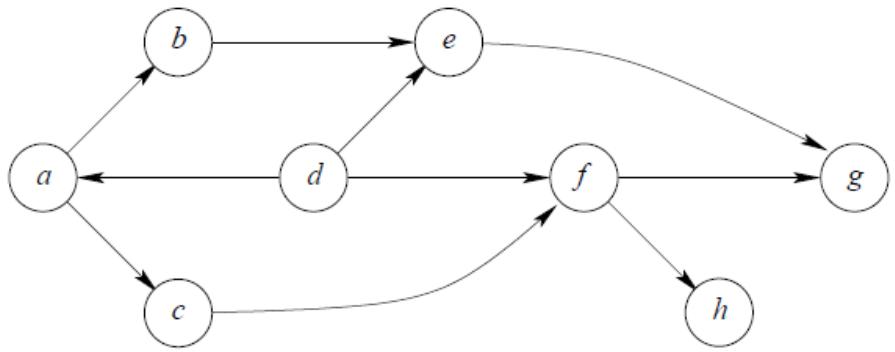
- 1 Executar $\text{DFS}(G)$ para cálculo do tempo de fim $f[v]$ para cada vértice v
- 2 Quando um vértice é terminado, inserir no princípio de lista ligada
- 3 **return** lista ligada de vértices

Complexidade: $O(V + E)$

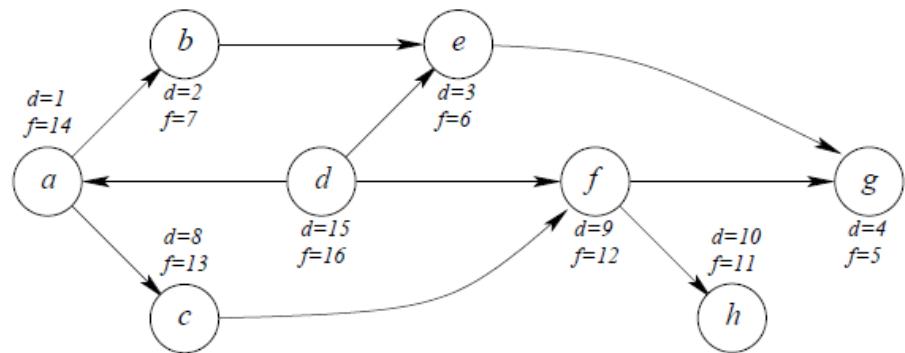
Ordenação Topológica: Exemplo



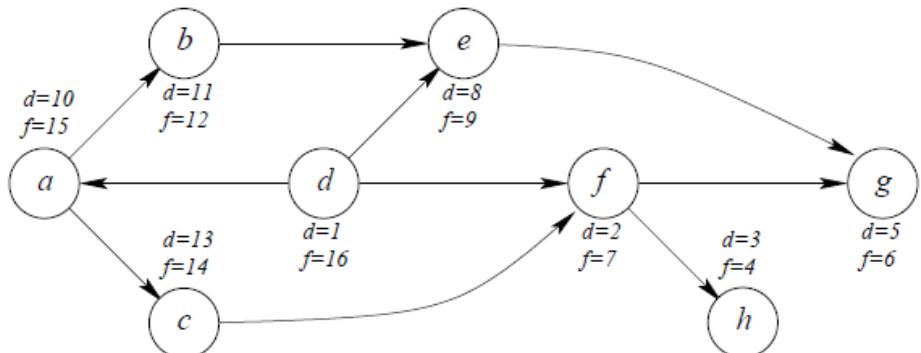
Podem existir várias ordenações válidas!



Exemplo Ordenação?

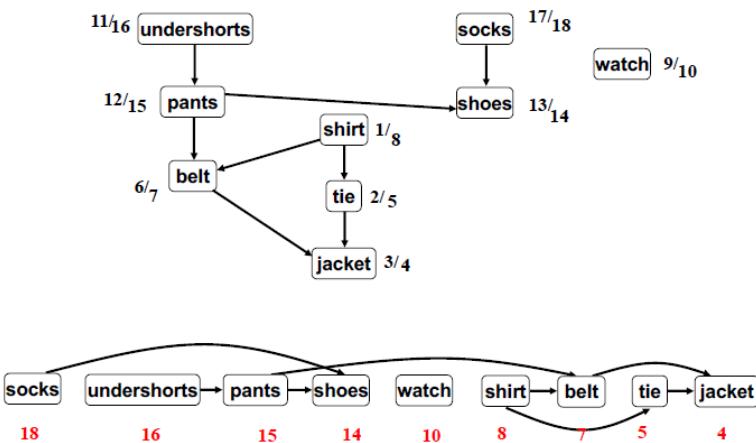


Ordenação: d, a, c, f, h, b, e, g



Ordenação: d, a, c, b, e, f, g, h

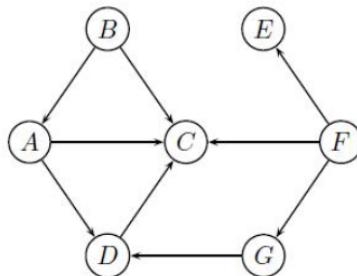
Ordenação Topológica: Exemplo & algoritmo



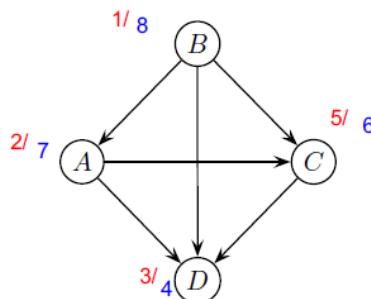
Exercício: ordenação topológica

- Qual das sequências de vértices corresponde a uma ordenação topológica válida do grafo abaixo ?

- <A,B,C,D,E,F,G>
- <B,A,D,C,G,F,E>
- <B,A,D,F,E,G,C>
- <B,F,A,E,D,C,G>
- <C,G,E,B,A,D,F>
- <F,E,G,D,C,B,A>
- <F,G,E,B,A,D,C>



Considere o seguinte grafo dirigido:



Indique uma ordenação topológica dos vértices do grafo.

Solução: B A C D

4 Componentes Fortemente Ligados

Componentes Fortemente Ligados (SCC)

Strongly Connected Components

• Um grafo (ou sub-grafo) diz-se **fortemente ligado** se para cada par de vértices u e v existe um caminho de u para v e de v para u

• Os componentes fortemente ligados de um grafo são os sub-grafos de tamanho máximo que são fortemente ligados

• Todos os vértices pertencem a uma componente fortemente ligada, mesmo que esta só contenha o próprio elemento

- Definição:

- Dado grafo dirigido $G = (V, E)$ um **componente fortemente ligado** (SCC) é um conjunto máximo de vértices $U \subseteq V$, tal que para $u, v \in U$, u é atingível a partir de v , e v é atingível a partir de u
 - Obs: um vértice simples é um SCC

- Outras definições:

- **Grafo transposto** de $G = (V, E)$
 - $G^T = (V, E^T)$ tal que: $E^T = \{(u, v) : (v, u) \in E\}$

- OBS: G e G^T têm os mesmos SCCs

Componente Fortemente Ligado

Dado um grafo dirigido $G = (V, E)$ um **componente fortemente ligado** (ou Strongly Connected Component (SCC)) é um conjunto máximo de vértices $U \subseteq V$, tal que para quaisquer $u, v \in U$, u é atingível a partir de v , e v é atingível a partir de u

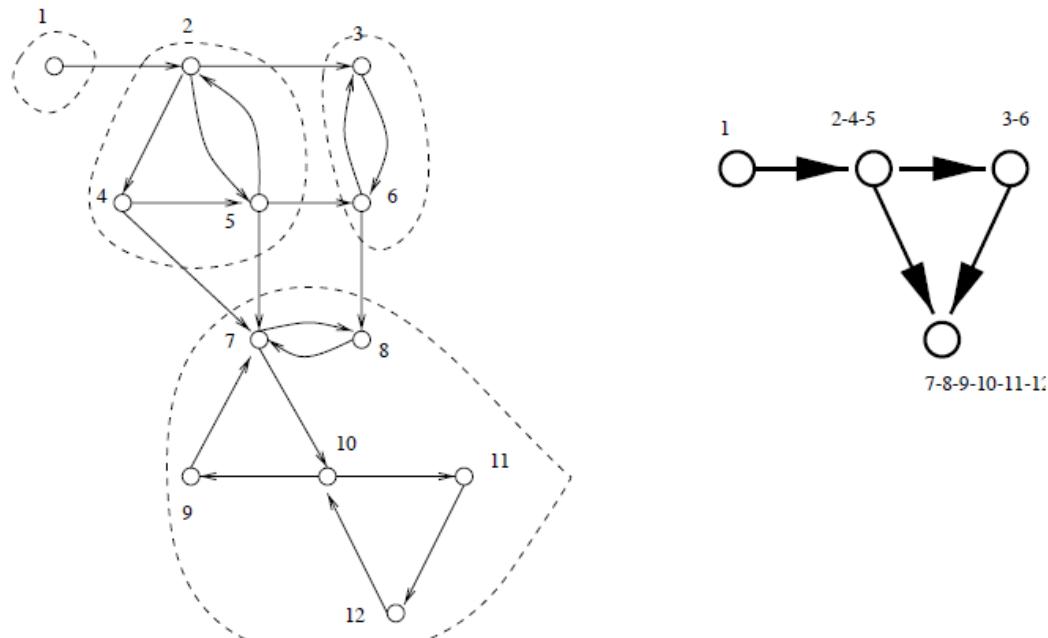
Nota: um vértice simples pode definir um SCC.

Grafo Transposto

Dado um grafo dirigido $G = (V, E)$, o grafo transposto de G é definido da seguinte forma:

$$G^T = (V, E^T) \text{ tal que } E^T = \{(u, v) : (v, u) \in E\}$$

Componentes Fortemente Ligados (SCC)



Componentes Fortemente Ligados (Implementação)

Strongly-Connected-Components(G):

1. Calcula $\text{DFS}(G)$, e o *finishing-time* (f) de cada nó.
2. Calcula G^T
3. Inicializa as cores, os predecessores e os tempos d e f de todos os nós de G^T , tal como no DFS.
4. Chama o DFS-visit para os nós de G^T por ordem inversa do *finishing-time* (f) obtido em 1.
5. Retorna a floresta-DF obtida em 4. Cada uma das árvores obtida é uma Componente Fortemente Ligada.

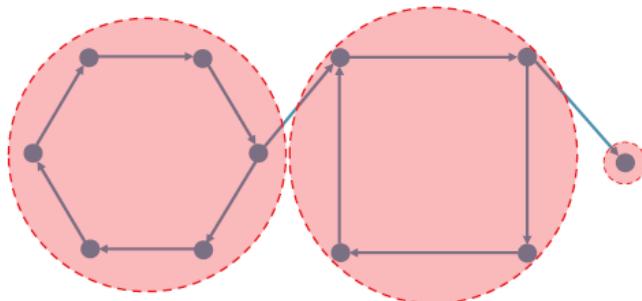
Pseudo-Código

$\text{SCCs}(G)$

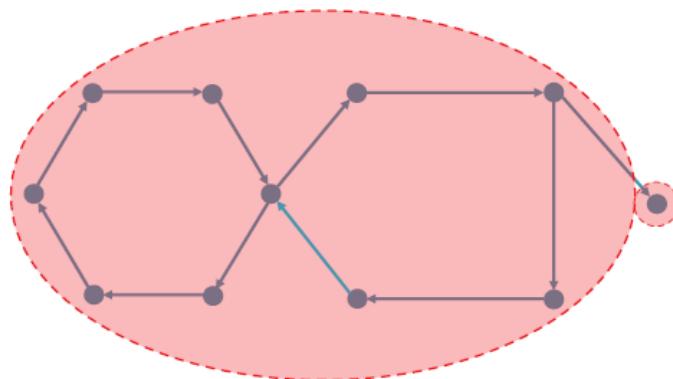
- 1 Executar $\text{DFS}(G)$ para cálculo do tempo de fim $f[v]$ para cada vértice v
- 2 Representar G^T
- 3 Executar $\text{DFS}(G^T)$ \triangleright (no ciclo principal de DFS considere os vértices por
- 4 \triangleright ordem decrescente de tempo de fim de $\text{DFS}(G)$)
- 5 O conjunto de vértices de cada árvore DF corresponde a um SCC

Complexidade: $O(V + E)$

- Ex. 1: Quantos SCCs ?



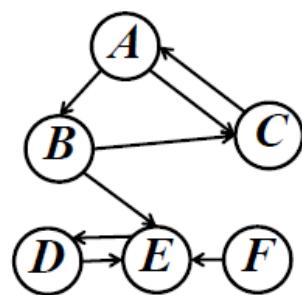
- Resposta: 3
- Ex. 2: Quantos SCCs ?



- Resposta: 2

Considere o grafo representado pela seguinte matriz de adjacências:

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



Quantos componentes fortemente ligados tem o grafo?

Solução: 3

ALGORITMO DE TARJAN

Intuição

- Baseado no algoritmo DFS
- Raiz de um SCC: primeiro vértice do SCC a ser descoberto
- Utilização de arcos para trás e de cruzamento na mesma árvore DF para identificação de ciclos
- $d[v]$: Número de vértices visitados quando v é descoberto
- $low[v]$: O menor valor de $d[]$ atingível por um arco para trás ou de cruzamento na sub-árvore de v
- Se $d[v] = low[v]$, então v é raiz de um SCC

Pseudo-código

Algoritmo de Tarjan

```
SCC_Tarjan(G)
1 visited  $\leftarrow 0$ 
2  $L \leftarrow \emptyset$ 
3 for each vertex  $u \in V[G]$ 
4   do  $d[u] \leftarrow \infty$ 
5 for each vertex  $u \in V[G]$ 
6   do if  $d[u] = \infty$ 
7     then Tarjan_Visit( $u$ )
```

Algoritmo de Tarjan

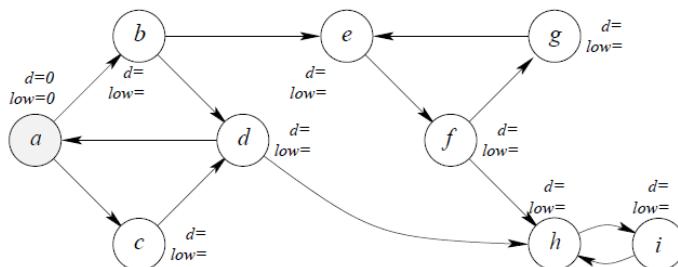
```
Tarjan_Visit( $u$ )
1  $d[u] \leftarrow low[u] \leftarrow visited$ 
2 visited  $\leftarrow visited + 1$ 
3 Push( $L, u$ )
4 for each  $v \in Adj[u]$ 
5   do if ( $d[v] = \infty \text{ || } v \in L$ )
6      $\triangleright$  Ignora vértices de SCCs já identificados
7     then if  $d[v] = \infty$ 
8       then Tarjan_Visit( $v$ )
9      $low[u] \leftarrow \min(low[u], low[v])$ 
10    if  $d[u] = low[u]$   $\triangleright$  Raiz do SCC
11      then repeat
12         $v \leftarrow Pop(L)$ 
13       $\triangleright$  Vértices retirados definem SCC
14    until  $u = v$ 
```

Complexidade

Tempo de execução: $O(V + E)$

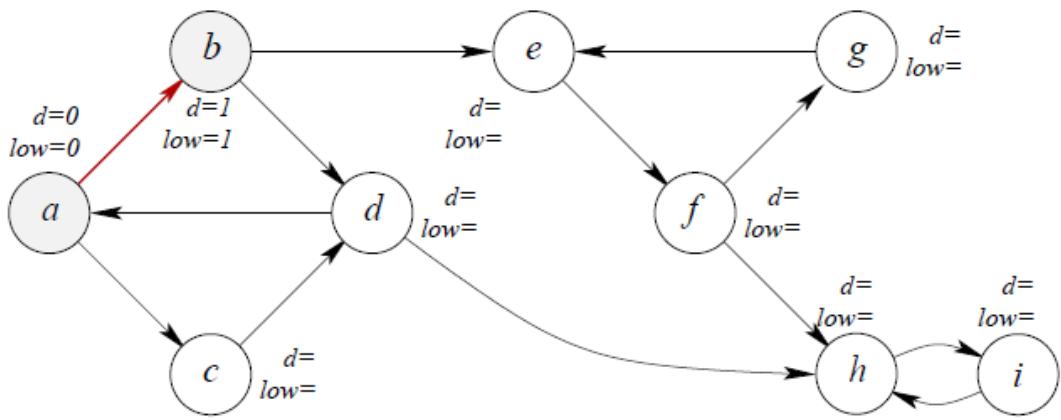
- Inicialização: $O(V)$
- Chamadas a Tarjan_Visit: $O(V)$
- Listas de adjacência de cada vértice analisadas apenas 1 vez: $\Theta(E)$

Exemplo



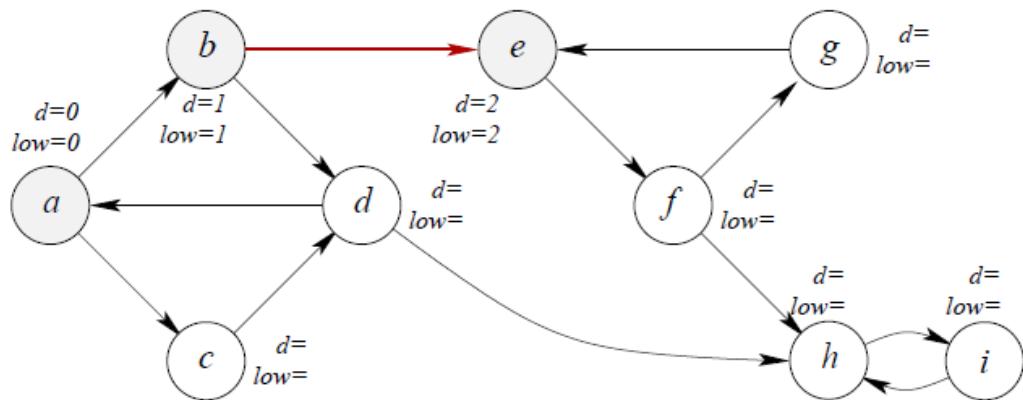
$L: a$

SCCs:



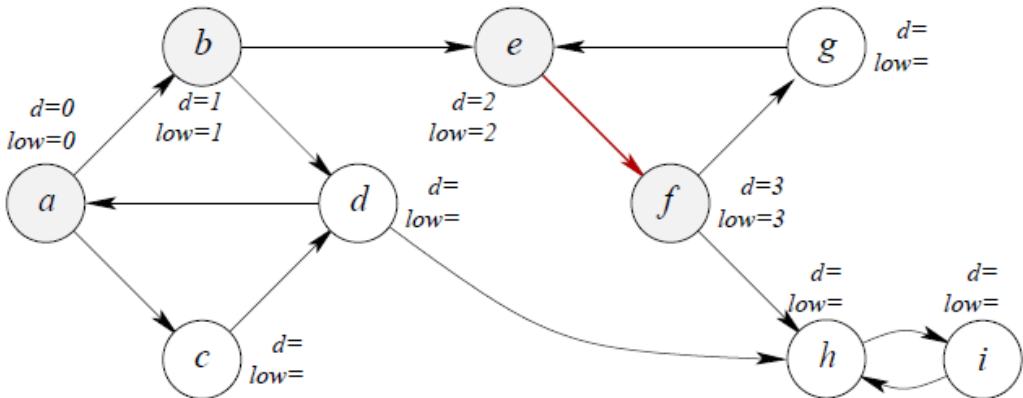
L: a, b

SCCs:



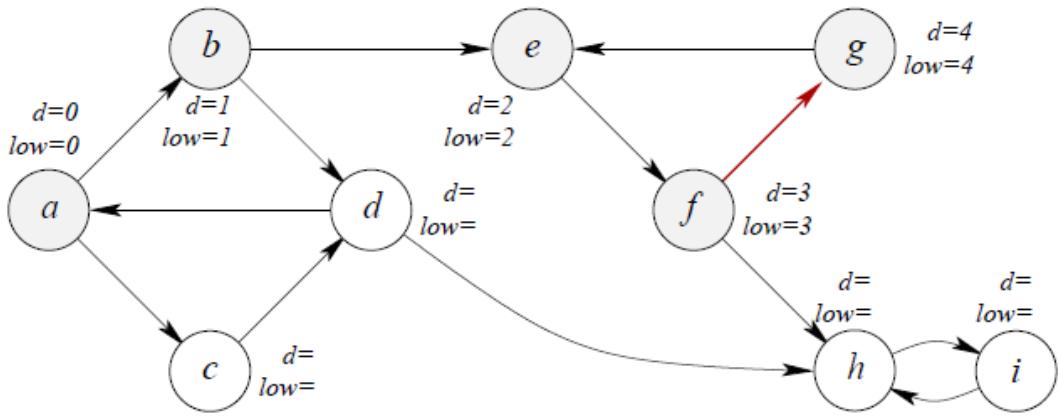
L: a, b, e

SCCs:



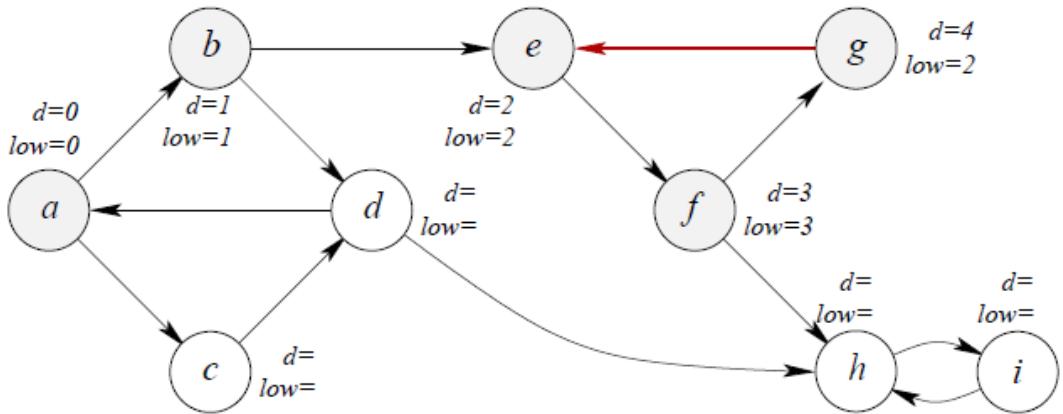
L: a, b, e, f

SCCs:



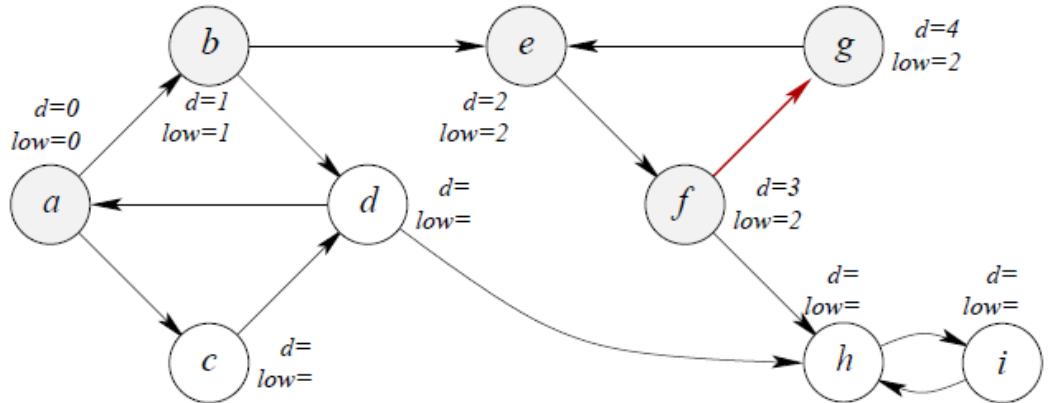
$L: a, b, e, f, g$

$SCCs:$



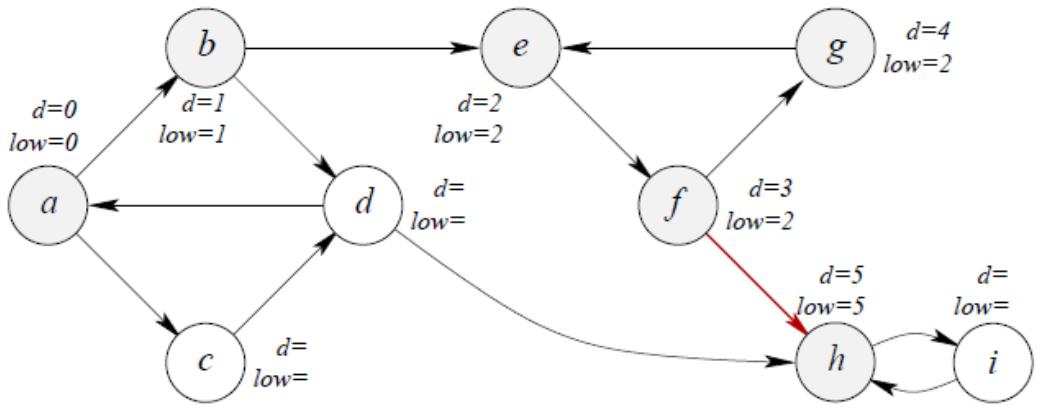
$L: a, b, e, f, g$

$SCCs:$



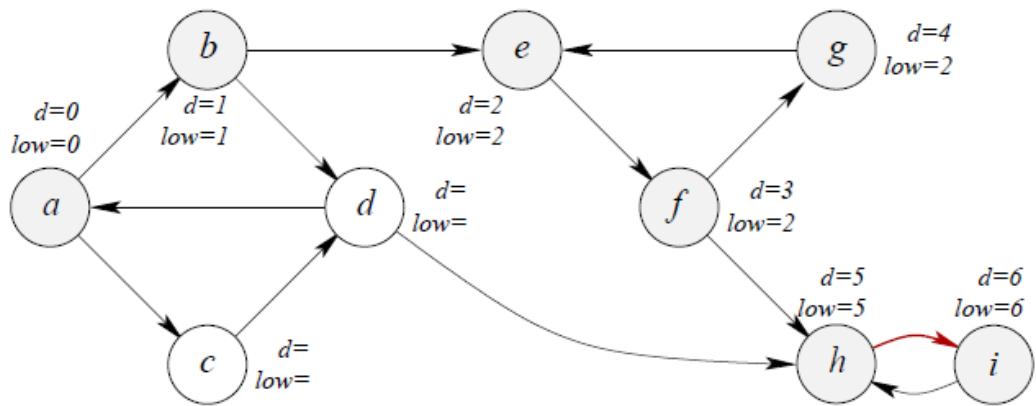
$L: a, b, e, f, g$

$SCCs:$



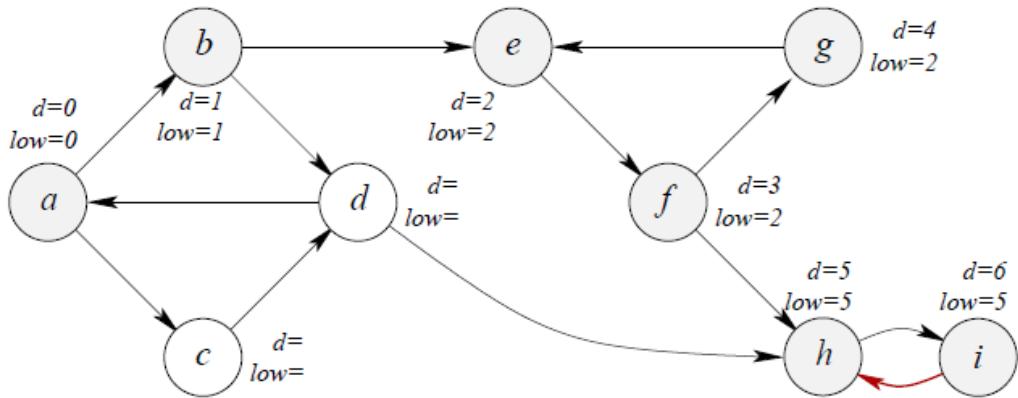
L: a, b, e, f, g, h

SCCs:



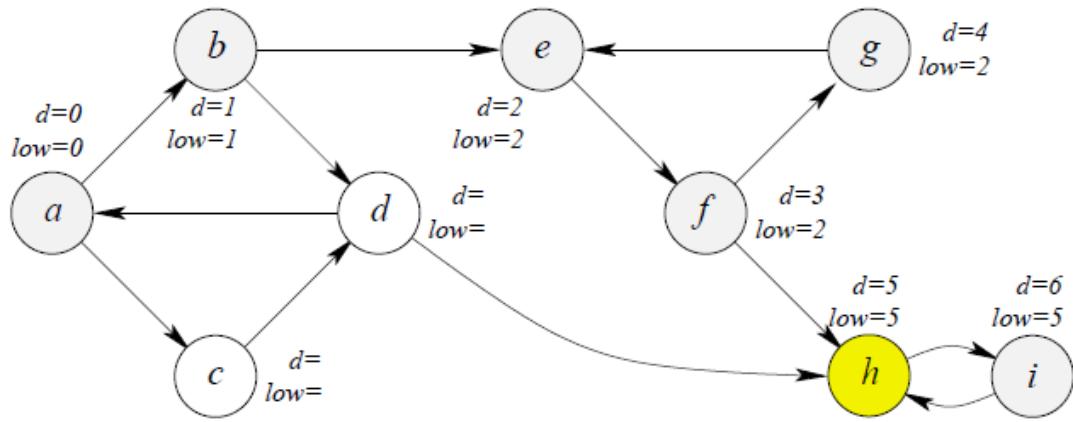
L: a, b, e, f, g, h, i

SCCs:



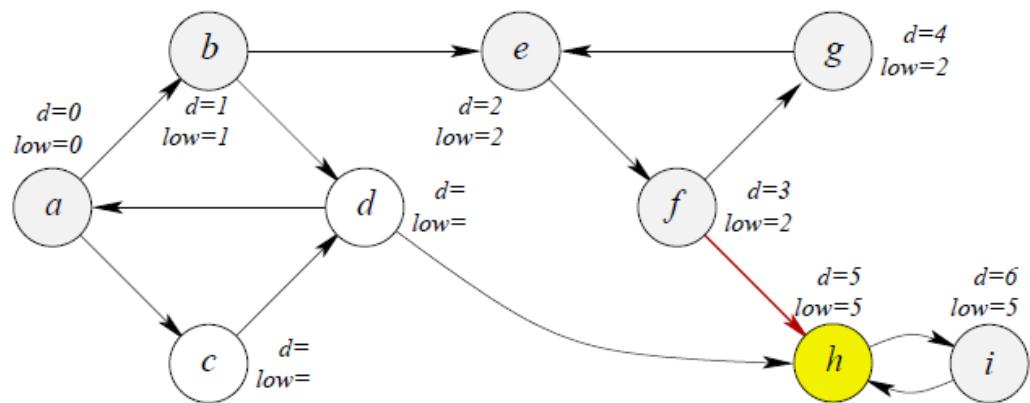
L: a, b, e, f, g, h, i

SCCs:



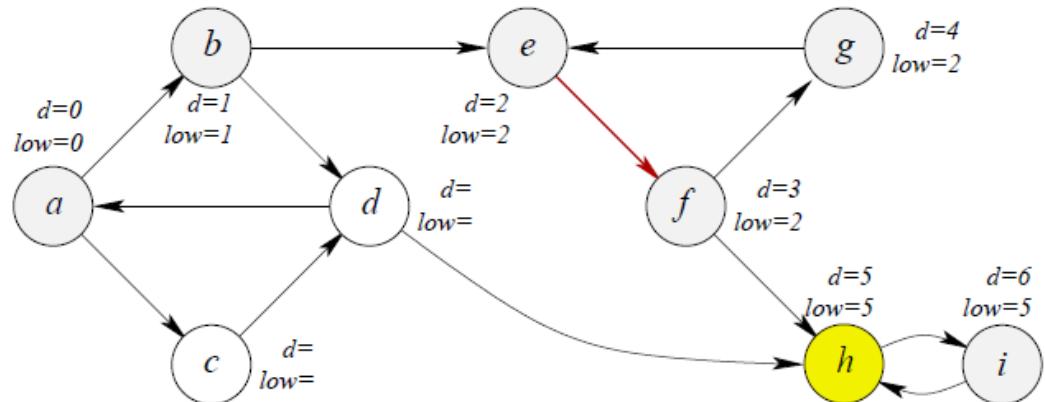
$L: a, b, e, f, g$

$SCCs: \{h, i\}$



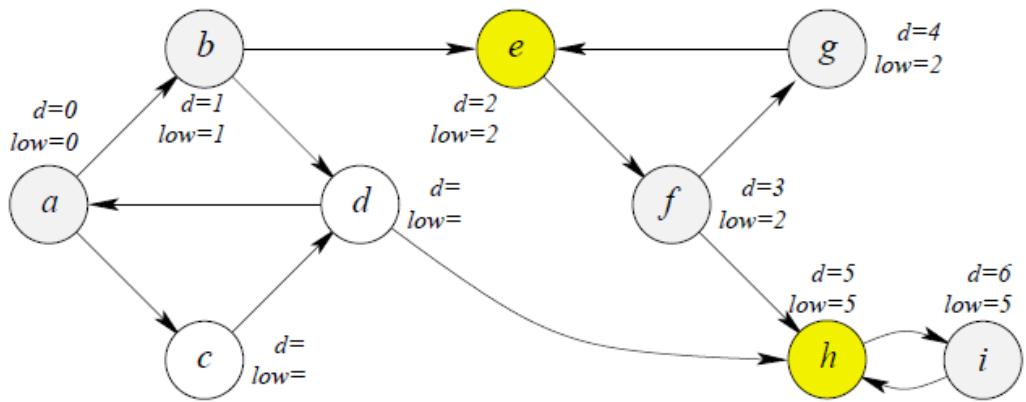
$L: a, b, e, f, g$

$SCCs: \{h, i\}$



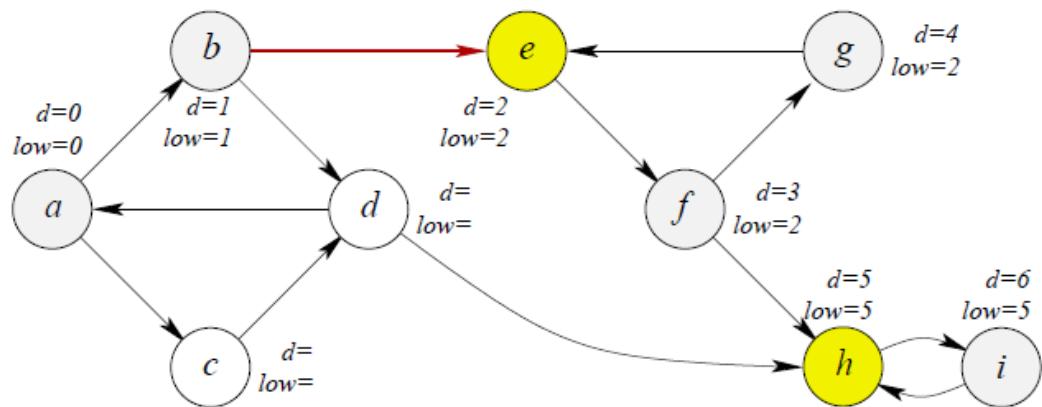
$L: a, b, e, f, g$

$SCCs: \{h, i\}$



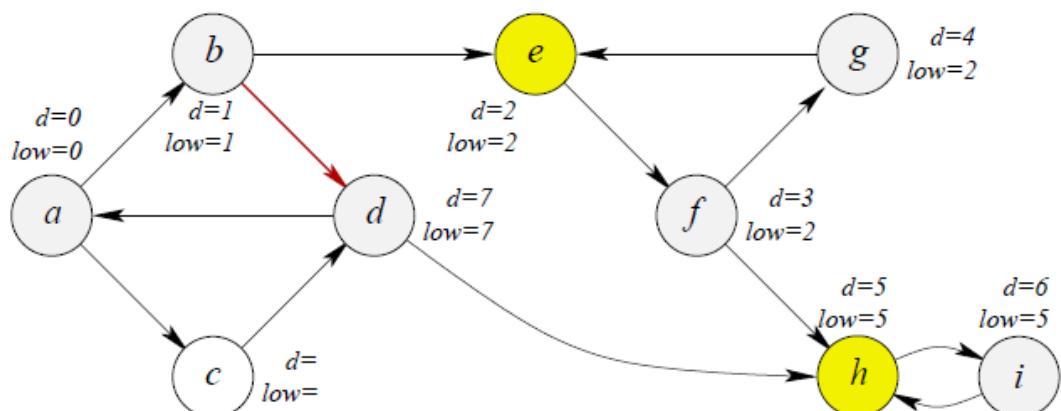
$L: a, b$

$SCCs: \{h, i\} \ \{e, f, g\}$



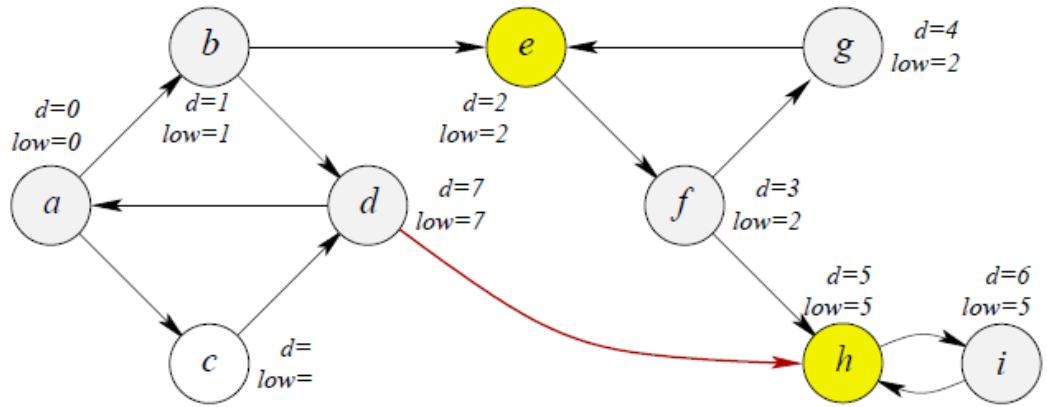
$L: a, b$

$SCCs: \{h, i\} \ \{e, f, g\}$



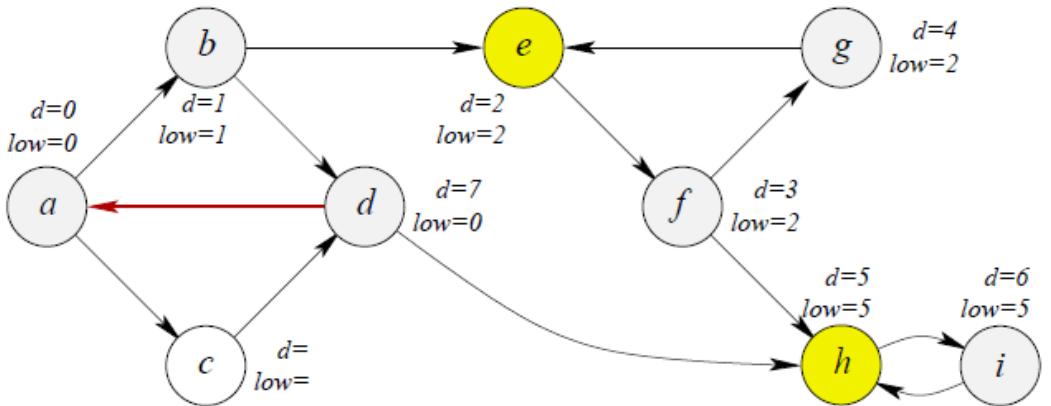
$L: a, b, d$

$SCCs: \{h, i\} \ \{e, f, g\}$



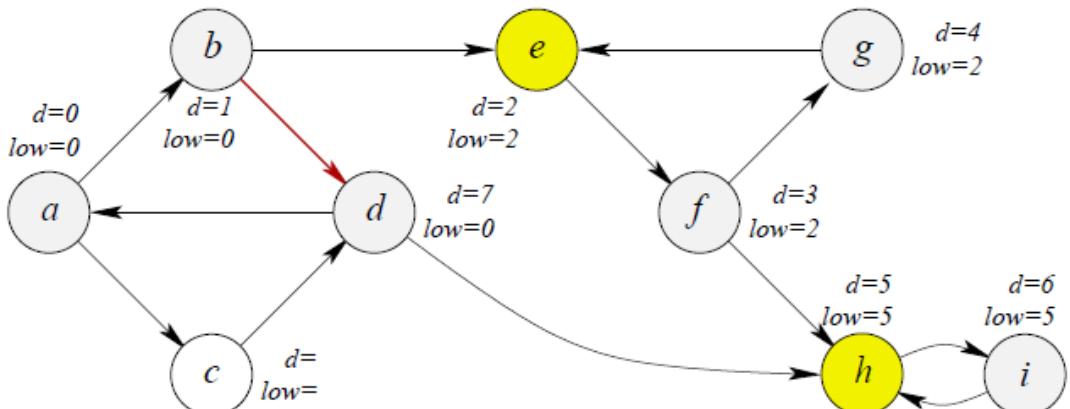
$L: a, b, d$

$SCCs: \{h, i\} \quad \{e, f, g\}$



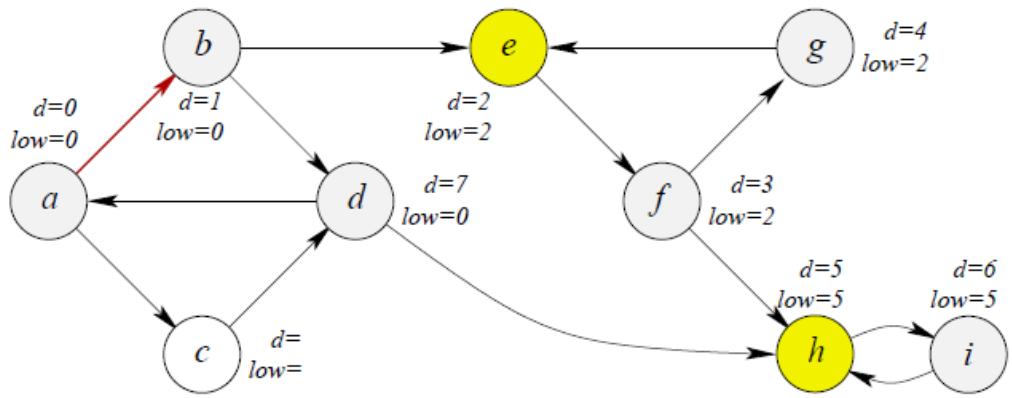
$L: a, b, d$

$SCCs: \{h, i\} \quad \{e, f, g\}$



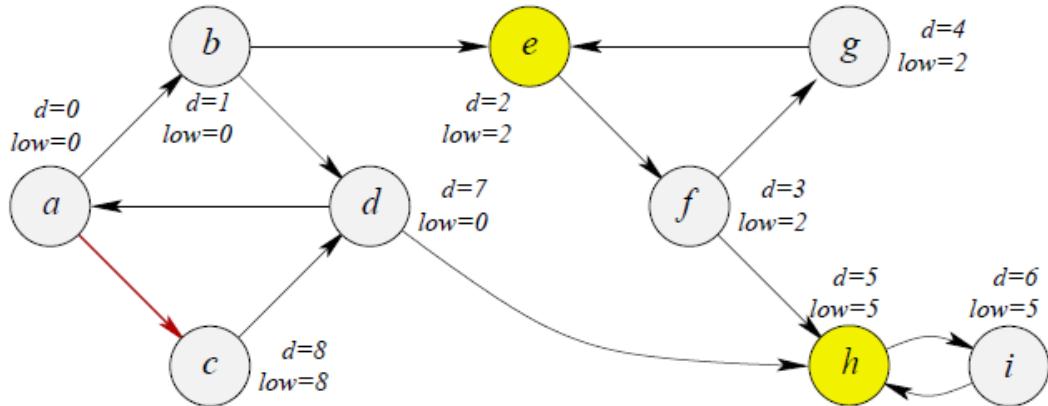
$L: a, b, d$

$SCCs: \{h, i\} \quad \{e, f, g\}$



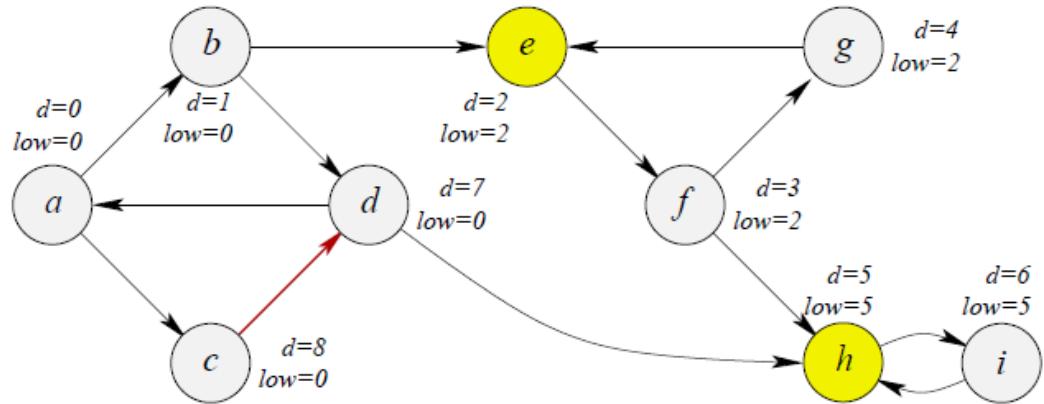
$L: a, b, d$

$SCCs: \{h, i\} \{e, f, g\}$



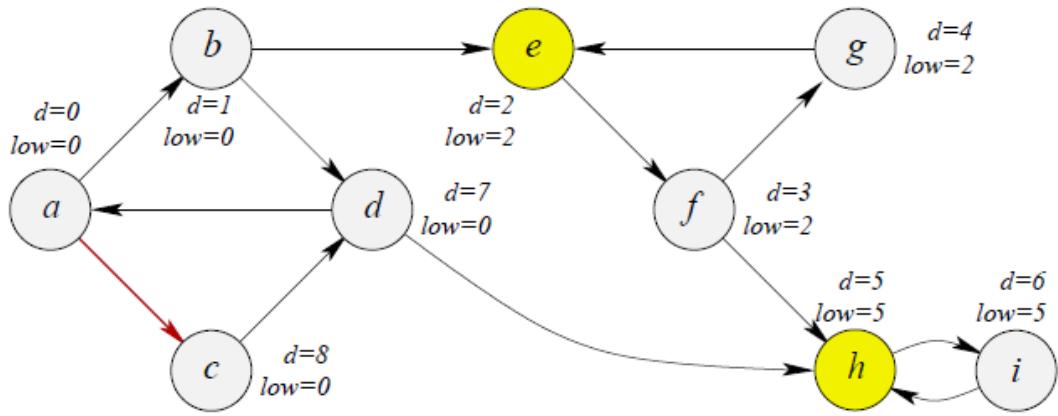
$L: a, b, d, c$

$SCCs: \{h, i\} \{e, f, g\}$



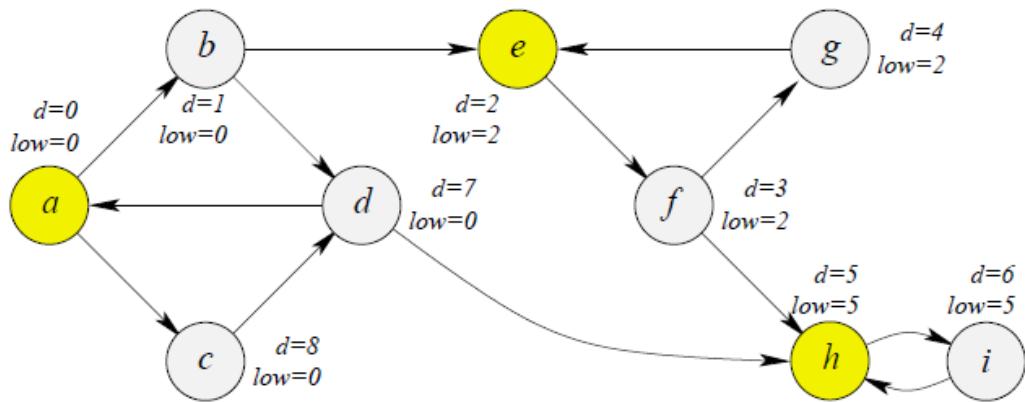
$L: a, b, d, c$

$SCCs: \{h, i\} \{e, f, g\}$



$L: a, b, d, c$

$SCCs: \{h, i\} \quad \{e, f, g\}$



$L:$

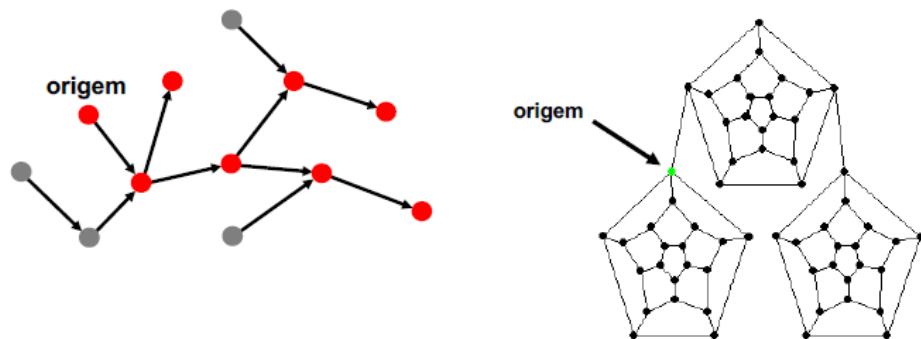
$SCCs: \{h, i\} \quad \{e, f, g\} \quad \{a, b, c, d\}$

5 Procura em Largura Primeiro

Procura em Largura Primeiro (BFS)

Breadth-First Search

- Visita os vértices por ordem da sua distância à origem
- Vértices mais próximos são visitados em primeiro lugar
 - Vértices não atingíveis a partir da origem, não são visitados



- Dados $G = (V, E)$ e vértice fonte s , BFS explora sistematicamente vértices de G para descobrir todos os vértices atingíveis a partir de s
 - Cálculo da distância: menor número de arcos de s para cada vértice atingível
 - Identificação de árvore BF: caminho mais curto de s para cada vértice atingível v
- Fronteira entre nós descobertos e não descobertos é expandida uniformemente
 - Vértices à distância k descobertos antes de qualquer vértice à distância $k+1$
- Aplicações
 - Explorar todos os elementos de um grafo acessíveis a partir de uma origem **src**, sendo que cada nodo é visitado apenas 1 vez!
 - Encontrar todos os nodos de uma componente conexa.
 - Encontrar todos os caminhos mais curtos entre 2 nodos, u e v , de um grafo sem pesos.
 - Entre outras...

Dados $G = (V, E)$ e vértice fonte s , o algoritmo BFS explora sistematicamente os vértices de G para descobrir todos os vértices atingíveis a partir de s

- Fronteira entre nós descobertos e não descobertos é expandida uniformemente
 - Nós à distância k descobertos antes de qualquer nó à distância $k + 1$
- Cálculo da distância: menor número de arcos de s para cada vértice atingível
 - $\delta(u, v)$: distância do caminho mais curto de u para v
- Identificação de árvore Breadth-First (BF): caminho mais curto de s para cada vértice atingível v

Implementação

- $\text{color}[v]$: cor do vértice v , (branco, cinzento ou preto)
 - branco: não visitado
 - cinzento: já visitado, mas algum dos adjacentes não visitado ou procura em algum dos adjacentes não terminada
 - preto: já visitado e procura nos adjacentes já terminada
- $\pi[v]$: predecessor de v na árvore BF
- $d[v]$: tempo de descoberta do vértice v

Outras definições

- Caminho mais curto definido como o caminho de s para v composto pelo menor número de arcos
- $\delta(s, v)$: denota o comprimento do caminho mais curto de s a v

Pseudo-código

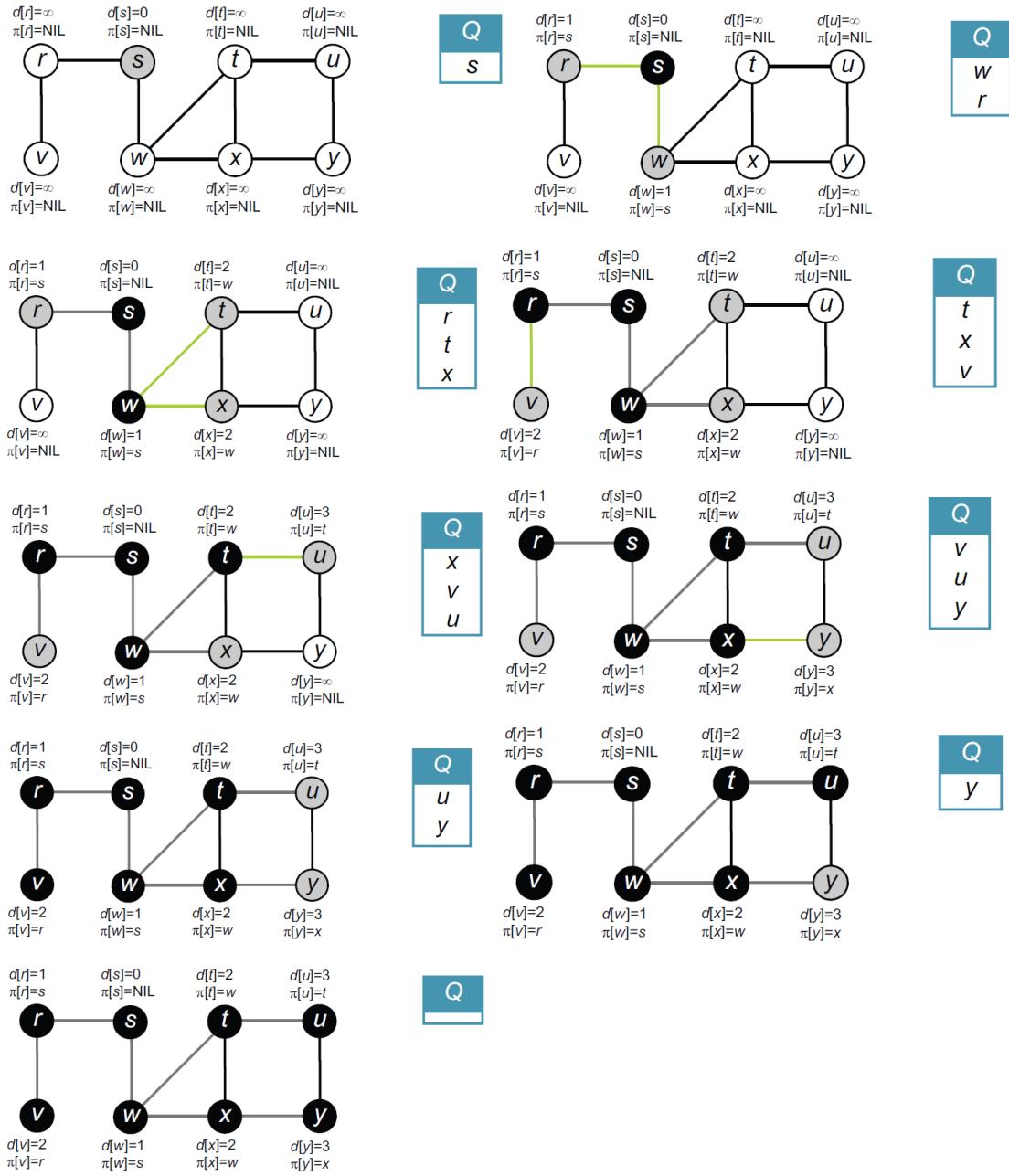
BFS(G, s)

```
1  for each vertex  $u \in V[G] - \{s\}$            ▷ Inicialização
2      do  $\text{color}[u] \leftarrow \text{white}$ ;  $d[u] \leftarrow \infty$ ;  $\pi[u] \leftarrow \text{NIL}$ 
3   $\text{color}[s] \leftarrow \text{gray}$ ;  $d[s] \leftarrow 0$ ;  $\pi[s] \leftarrow \text{NIL}$ 
4   $Q \leftarrow \{s\}$ 
5  while  $Q \neq \emptyset$                          ▷ Ciclo Principal
6      do  $u \leftarrow \text{Dequeue}(Q)$ 
7          for each  $v \in \text{Adj}[u]$ 
8              do if  $\text{color}[v] = \text{white}$ 
9                  then  $\text{color}[v] \leftarrow \text{gray}$ ;  $d[v] \leftarrow d[u] + 1$ ;  $\pi[v] \leftarrow u$ 
10                 Enqueue( $Q, v$ )
11       $\text{color}[u] \leftarrow \text{black}$ 
```

• Implementação

- $\text{color}[v]$: cor do vértice v , branco, cinzento e preto
 - • branco: não visitado (não processado)
 - • cinzento: já visitado mas algum dos adjacentes não visitado ou procura em algum dos adjacentes não terminada (a processar)
 - • preto: já visitado e procura nos adjacentes já terminada (processado)
- $\pi[v]$: predecessor de v na árvore BF
- $d[v]$: distância de v à origem

Exemplo



Complexidade

Tempo de execução: $O(V + E)$

- Inicialização: $O(V)$
- Para cada vértice
 - Colocado na fila apenas 1 vez: $O(V)$
 - Lista de adjacências visitada 1 vez: $O(E)$

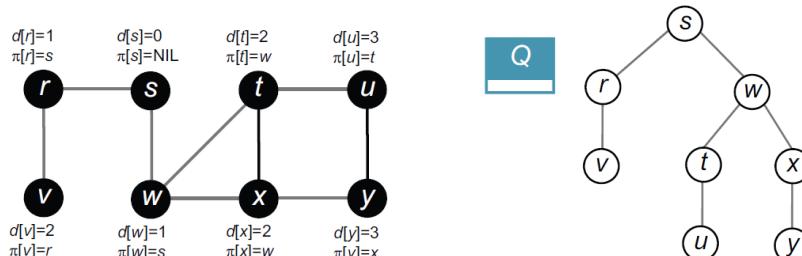
Resultados

- Para qualquer arco $(u, v) : \delta(s, v) \leq \delta(s, u) + 1$
- Se u atingível, então v atingível
 - caminho mais curto para v não pode ser superior a caminho mais curto para u mais arco (u, v)
 - Se u não atingível, então resultado é válido (independentemente de v ser atingível)
- No final da BFS: $d[u] = \delta(s, u)$, para todo o vértice $u \in V$

Árvore Breadth-First

- Árvore BF é sub-grafo de G
- Vértices atingíveis a partir de s
- Arcos que definem a relação predecessor da BFS
- $G_\pi = (V_\pi, E_\pi)$
- $V_\pi = \{v \in V : \pi[v] \neq NIL\} \cup \{s\}$
- $E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\}$

- Árvore BF: pai na árvore é o predecessor π

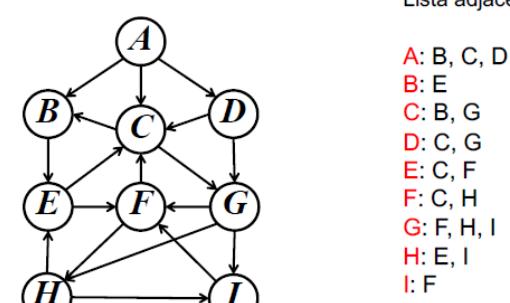


Procura em Largura Primeiro (BFS)

- Exemplo 1:

Lista adjacente:

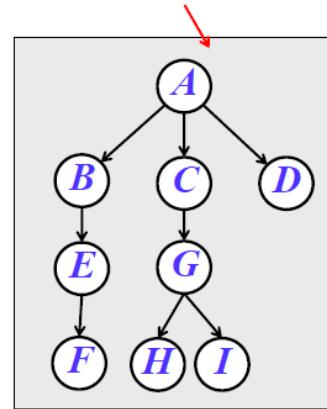
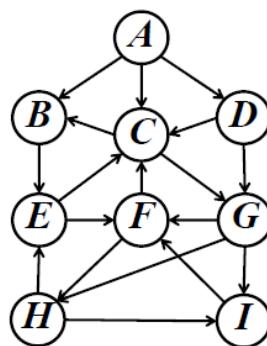
O BFS depende
da ordem das listas



Sequência: A B C D E G F H I
Distância: 0 1 1 1 2 2 3 3 3

Procura em Largura Primeiro (**BFS trees**)

- Exemplo 1:



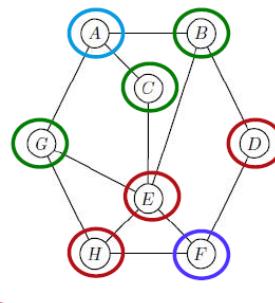
Sequência: A B C D E G F H I

Distância: 0 1 1 1 2 2 3 3 3

Predecessores: NIL A A A A B C E G G

(ou pais)

- Qual a sequência de vértices visitados numa travessia em largura primeiro (BFS) sobre o grafo abaixo, com origem no vértice A ?
Considere que os vértices são visitados por ordem alfabética e que os vértices adjacentes de um vértice também são visitados por ordem alfabética.



A B C G D E H F

6 Problemas

Problema 1

Grafo Bipartido

Indique um algoritmo eficiente para determinar se um grafo $G = (V, E)$ é bipartido.

- Grafo G é bipartido se V pode ser dividido em L e R , tal que todos os arcos de G incidentes em 1 vértice de L e 1 vértice de R

Problema 2

Grafo Semi-Ligado

Indique um algoritmo eficiente para determinar se um grafo $G = (V, E)$ é semi-ligado.

- Um grafo dirigido $G = (V, E)$ diz-se semi-ligado se para qualquer par de vértices (u, v) , u é atingível a partir de v ou v é atingível a partir de u

Problema 3

Pontos de Articulação

Indique um algoritmo eficiente para determinar se um grafo $G = (V, E)$ não dirigido e ligado tem pontos de articulação.

- Um grafo não dirigido diz-se ligado se para qualquer par de vértices $u, v \in V$, existe pelo menos um caminho entre u e v .
- Um vértice $u \in V$ diz-se um ponto de articulação de um grafo se a remoção do vértice u implicar que o grafo deixa de ser ligado.

Problema 4 - Coloração de vértices

Grafo colorável

Um grafo diz-se *colorável* se for possível atribuir cores aos seus vértices, de tal forma que dois vértices adjacentes não tenham a mesma cor. Se tal distribuição de cores for possível recorrendo apenas a k cores, o grafo diz-se k -colorável. O menor valor de k que torna o grafo G k -colorável designa-se por *número cromático* de G .

Caso particular: grafos 2-coloráveis, também designados por grafos bi-coloráveis

- Qualquer ciclo com p vértices é bi-colorável se e só se p é par.
- G é bi-colorável se e só se não contém ciclos com um número ímpar de vértices.

Algoritmo de coloração de vértices num grafo bi-colorável

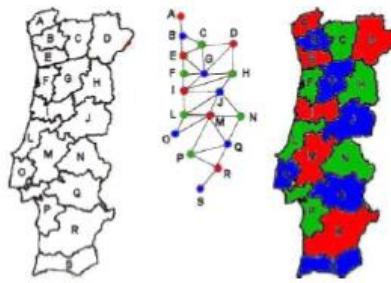
(https://www.youtube.com/watch?v=052VkJlaQ4&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=66)

Algoritmo de coloração BFS (breadth-first search) de nível-a-nível

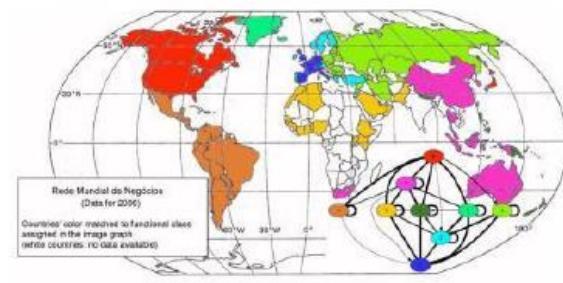
- 1 Atribuir o nível 0 a um vértice v qualquer arbitrariamente escolhido;
- 2 Atribuir o nível 1 a todos os vértices adjacentes a v ;
- 3 Para cada nível i :
 - 1 Sejam $v_{i1}, v_{i2}, \dots, v_{ir}$ todos os vértices no nível i ;
 - 2 Consideremos todos os vértices adjacentes a v_{i1} que não figuram nos níveis 0,1, até i e coloquemos estes vértices no nível $i + 1$;
 - 3 consideremos todos os vértices adjacentes a v_{i2} que não figurem no nível $i + 1$, e coloquemo-los nesse nível;
 - 4 ...

O processo termina quando a todos os vértices do grafo tiver sido atribuído um nível. O processo de atribuição de cores consiste em atribuir uma cor aos vértices dos níveis pares e ao vértice do nível 0, e outra cor (diferente da primeira) aos vértices dos níveis ímpares.

Mapa colorido de Portugal e seu respectivo grafo



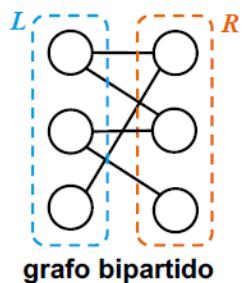
Relações na rede mundial de negócios e o respectivo grafo



Desafio!

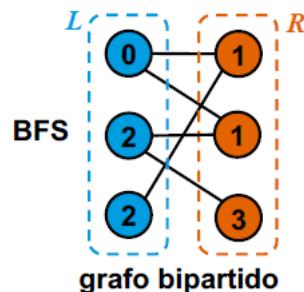
- Algoritmo eficiente para determinar se grafo $G = (V, E)$ é bipartido ?

– Grafo G é bipartido se V pode ser dividido em L e R , tal que todos os arcos de G são incidentes em 1 vértice de L e 1 vértice de R



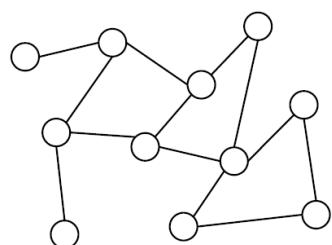
– Solução: realizar uma BFS (largura primeiro) em cada componente ligado do grafo e verificar se os vértices adjacentes já visitados têm paridade do tempo de descoberta diferente da do vértice actual

- se sim, para todos os vértices, então o grafo é bipartido
- se não, para algum vértice, então o grafo não é bipartido.



Problema

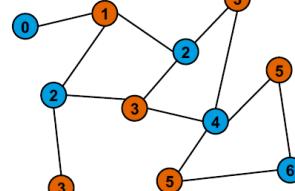
- O grafo abaixo é bipartido?



BFS ?

- O grafo abaixo é bipartido?

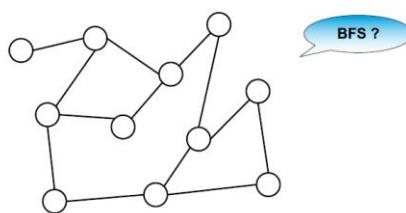
Sim!



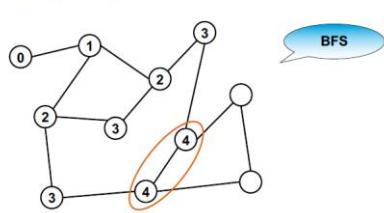
BFS

Problema

- O grafo abaixo é bipartido?



- O grafo abaixo é bipartido? Não



5.3. CAMINHOS MAIS CURTOS

Problema do caminho de custo mínimo numa rede dirigida

Suponhamos que a cada arco de um digrafo é atribuído um custo ou peso numérico (que também pode ser considerado como uma distância). O *problema do caminho de custo mínimo* consiste em determinar, caso exista, o caminho de menor soma de pesos entre dois vértices dados.

1 Definições

- Caminhos Mais Curtos

Definições

- Dado um grafo $G = (V, E)$, dirigido, com uma função de pesos $w : E \rightarrow \mathbb{R}$, define-se o **peso de um caminho** p , onde $p = < v_0, v_1, \dots, v_k >$, como a soma dos pesos dos arcos que compõem p :

$$w(p) = \sum_{i=1}^k w(v_{i-1}, v_i)$$

- O **peso do caminho mais curto** de u para v é definido por:

$$\delta(u, v) = \begin{cases} \min \{w(p) : u \rightarrow_p v\} & \text{se existe caminho de } u \text{ para } v \\ \infty & \text{caso contrário} \end{cases}$$

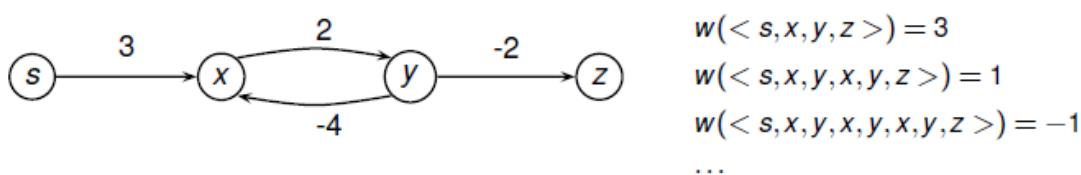
- Um **caminho mais curto** de u para v é qualquer caminho p tal que $w(p) = \delta(u, v)$

Problemas de Caminhos Mais Curtos

- **Caminhos Mais Curtos com Fonte Única (SSSPs)**
 - Identificar o caminho mais curto de um vértice fonte $s \in V$ para qualquer outro vértice $v \in V$
- **Caminhos Mais Curtos com Destino Único**
 - Identificar o caminho mais curto de qualquer vértice $v \in V$ para um vértice destino $t \in V$
- **Caminho Mais Curto entre Par Único**
 - Identificar caminho mais curto entre dois vértices u e v
- **Caminhos Mais Curtos entre Todos os Pares (APSPs)**
 - Identificar um caminho mais curto entre cada par de vértices de V

Ciclos Negativos

- Arcos podem ter pesos com valor negativo
- É possível a existência de ciclos com peso total negativo
 - Se ciclo negativo não atingível a partir da fonte s , então $\delta(s, v)$ bem definido
 - Se ciclo negativo atingível a partir da fonte s , então os pesos dos caminhos mais curtos não são bem definidos
 - Neste caso, é sempre possível encontrar um caminho mais curto de s para qualquer vértice incluído no ciclo e define-se $\delta(s, v) = -\infty$



Identificação de Ciclos Negativos

- Dijkstra: requer pesos não negativos
- Bellman-Ford: identifica ciclos negativos e reporta a sua existência

2 Caminhos Mais Curtos com Fonte Única

- Representação de Caminhos Mais Curtos
- Propriedades dos Caminhos Mais Curtos
- Operação de Relaxação

Representação de Caminhos Mais Curtos

- Para cada vértice $v \in V$ associar predecessor $\pi[v]$
- Após identificação dos caminhos mais curtos, $\pi[v]$ indica qual o vértice anterior a v num caminho mais curto de s para v
- Sub-grafo de predecessores $G_\pi = (V_\pi, E_\pi)$:

$$V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\}$$

$$E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\}$$

Representação de Caminhos Mais Curtos

- Uma **árvore de caminhos mais curtos** é um sub-grafo dirigido $G' = (V', E')$, $V' \subseteq V$ e $E' \subseteq E$, tal que:
 - V' é o conjunto de vértices atingíveis a partir de s em G
 - G' forma uma árvore com raiz s
 - Para todo o $v \in V'$, o único caminho de s para v em G' é um caminho mais curto de s para v em G

Observações:

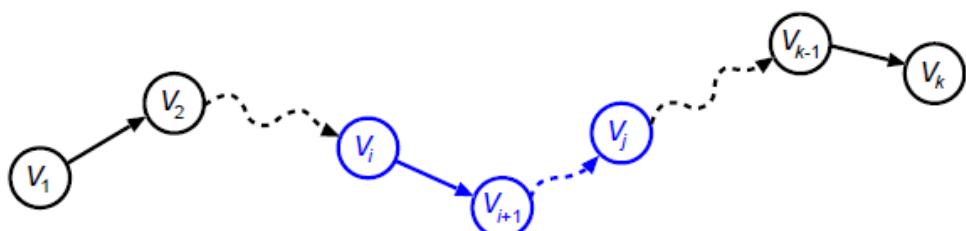
- Após identificação dos caminhos mais curtos de G a partir de fonte s , G' é dado por $G_\pi = (V_\pi, E_\pi)$
- Dados os mesmos grafo G e vértice fonte s , G' não é necessariamente único

Propriedades dos caminhos mais curtos

Sub-estrutura óptima

Sub-caminhos de caminhos mais curtos são caminhos mais curtos

- Seja $p = < v_1, v_2, \dots, v_k >$ um caminho mais curto entre v_1 e v_k , e seja $p_{ij} = < v_i, v_{i+1}, \dots, v_j >$ um sub-caminho de p entre v_i e v_j .
- Então p_{ij} é um caminho mais curto entre v_i e v_j
 - Porquê? Se existisse caminho mais curto entre v_i e v_j então seria possível construir caminho entre v_1 e v_k mais curto do que p ; Contradição, dado que p é um caminho mais curto entre v_1 e v_k .



Sub-estrutura óptima

Sub-caminhos de caminhos mais curtos são caminhos mais curtos

- Seja $p = \langle v_1, v_2, \dots, v_k \rangle$ um caminho mais curto entre v_1 e v_k , e seja $p_{ij} = \langle v_i, v_{i+1}, \dots, v_j \rangle$ um sub-caminho de p entre v_i e v_j .
- Então p_{ij} é um caminho mais curto entre v_i e v_j
 - Porquê? Se existisse caminho mais curto entre v_i e v_j então seria possível construir caminho entre v_1 e v_k mais curto do que p ;
Contradição, dado que p é um caminho mais curto entre v_1 e v_k .

Peso de um Caminho Mais Curto

Seja $p = \langle s, \dots, v \rangle$ um caminho mais curto entre s e v , que pode ser decomposto em $p_{su} = \langle s, \dots, u \rangle$ e (u, v) . Então

$$\delta(s, v) = \delta(s, u) + w(u, v)$$

- p_{su} é caminho mais curto entre s e u
- $\delta(s, v) = w(p) = w(p_{su}) + w(u, v) = \delta(s, u) + w(u, v)$

Relação caminho mais curto com arcos do grafo

Para todos os arcos $(u, v) \in E$ verifica-se $\delta(s, v) \leq \delta(s, u) + w(u, v)$

- Caminho mais curto de s para v não pode ter mais peso do que qualquer outro caminho de s para v
- Assim, peso do caminho mais curto de s para v não superior ao peso do caminho mais curto de s para u seguido do arco (u, v) (i.e. exemplo de um dos caminhos de s para v)

Operação de Relaxação

- Operação básica dos algoritmos para cálculo dos caminhos mais curtos com fonte única.
- $d[v]$: denota a estimativa do caminho mais curto de s para v ; limite superior no valor do peso do caminho mais curto;
- Algoritmos aplicam sequência de relaxações dos arcos de G após inicialização para actualizar a estimativa do caminho mais curto

Initialize-Single-Source(G, s)

```
1 for each  $v \in V[G]$ 
2     do  $d[v] \leftarrow \infty$ 
3          $\pi[v] \leftarrow NIL$ 
4      $d[s] \leftarrow 0$ 
```

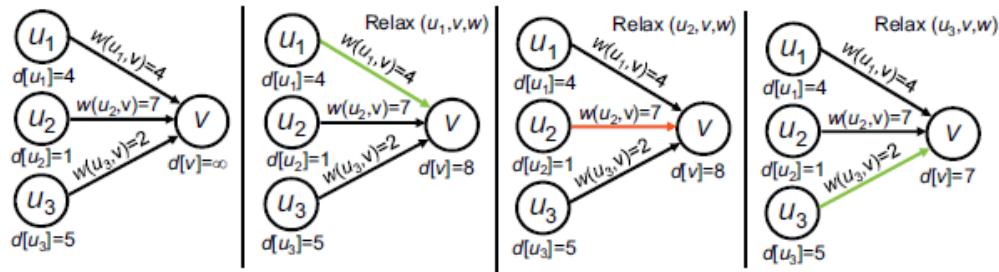
Relax(u, v, w)

```
1 if  $d[v] > d[u] + w(u, v)$ 
2 then  $d[v] \leftarrow d[u] + w(u, v)$ 
3      $\pi[v] \leftarrow u$ 
```

Propriedades da Relaxação

Após relaxar arco (u, v) , temos que $d[v] \leq d[u] + w(u, v)$

- Se $d[v] > d[u] + w(u, v)$ antes da relaxação, então $d[v] = d[u] + w(u, v)$ após relaxação
- Se $d[v] \leq d[u] + w(u, v)$ antes da relaxação, então $d[v] \leq d[u] + w(u, v)$ após relaxação
- Em qualquer caso, $d[v] \leq d[u] + w(u, v)$ após relaxação



Propriedades da Relaxação

$d[v] \geq \delta(s, v)$ para qualquer $v \in V[G]$ e para qualquer sequência de passos de relaxação nos arcos de G . Se $d[v]$ atinge o valor $\delta(s, v)$, então o valor não é mais alterado

- $d[v] \geq \delta(s, v)$ é válido após inicialização
- Prova por contradição para os restantes casos:
 - Seja v o primeiro vértice para o qual a relaxação do arco (u, v) causa $d[v] < \delta(s, v)$
 - Após relaxar arco: $d[u] + w(u, v) = d[v] < \delta(s, v) \leq \delta(s, u) + w(u, v)$
 - Pelo que, $d[u] < \delta(s, u)$ antes da relaxação de (u, v) ; Contradição, dado que v seria o primeiro vértice para o qual $d[v] < \delta(s, v)$
- Após ter $d[v] = \delta(s, v)$, o valor de $d[v]$ não pode decrescer; e pela relaxação também não pode crescer!

Propriedades da Relaxação

Seja $p = < s, \dots, u, v >$ um caminho mais curto em G , e seja $\text{Relax}(u, v, w)$ executada no arco (u, v) . Se $d[u] = \delta(s, u)$ antes da chamada a $\text{Relax}(u, v, w)$, então $d[v] = \delta(s, v)$ após a chamada a $\text{Relax}(u, v, w)$

- Se $d[u] = \delta(s, u)$ então valor de $d[u]$ não é mais alterado
- Após relaxar arco (u, v) :

$$d[v] \leq d[u] + w(u, v) = \delta(s, u) + w(u, v) = \delta(s, v)$$
- Mas, $d[v] \geq \delta(s, v)$, pelo que $d[v] = \delta(s, v)$, e não se altera !

Resumo das propriedades dos caminhos mais curtos

Sub-estrutura óptima

Sub-caminhos de caminhos mais curtos são caminhos mais curtos

- Seja $p = \langle v_1, v_2, \dots, v_k \rangle$ um caminho mais curto entre v_1 e v_k , e seja $p_{ij} = \langle v_i, v_{i+1}, \dots, v_j \rangle$ um sub-caminho de p entre v_i e v_j .
- Então p_{ij} é um caminho mais curto entre v_i e v_j

Sub-estrutura óptima

Seja $p = \langle s, \dots, v \rangle$ um caminho mais curto entre s e v , que pode ser decomposto em $p_{su} = \langle s, \dots, u \rangle$ e (u, v) . Então $\delta(s, v) = \delta(s, u) + w(u, v)$

Relação caminho mais curto com arcos do grafo

Para todos os arcos $(u, v) \in E$ verifica-se $\delta(s, v) \leq \delta(s, u) + w(u, v)$

Resumo das propriedades das operações de relaxação

Propriedades da Relaxação

Após relaxar arco (u, v) , temos que $d[v] \leq d[u] + w(u, v)$

Propriedades da Relaxação (2)

$d[v] \geq \delta(s, v)$ para qualquer $v \in V[G]$ e para qualquer sequência de passos de relaxação nos arcos de G . Se $d[v]$ atinge o valor $\delta(s, v)$, então o valor não é mais alterado

Propriedades da Relaxação (3)

Seja $p = \langle s, \dots, u, v \rangle$ um caminho mais curto em G , e seja $\text{Relax}(u, v, w)$ executada no arco (u, v) . Se $d[u] = \delta(s, u)$ antes da chamada a $\text{Relax}(u, v, w)$, então $d[v] = \delta(s, v)$ após a chamada a $\text{Relax}(u, v, w)$

3 Algoritmo Dijkstra

(<https://www.youtube.com/watch?v=XB4MIlexjvY0>)

(https://www.youtube.com/watch?v=XB4MIlexjvY0&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=45)

Restrição à aplicação do algoritmo

Os custos dos arcos da rede dirigida têm que ser todos não negativos.

Matriz de custos

Na rede dirigida $G = (V, E)$ suponhamos que $V = \{1, 2, \dots, n\}$ e consideremos a matriz de custos ou pesos $C = [c_{ij}]$, com

$$c_{ij} = \begin{cases} \text{peso do arco } (i, j) & , \text{ se existir arco de } i \text{ para } j \\ +\infty & , \text{ caso contrário} \end{cases}$$

Consideraremos também, sem perda de generalidade, os elementos diagonais da matriz de pesos todos iguais a 0.

Objectivo

O algoritmo de Dijkstra tem como objectivo a determinação do caminho mais curto e a distância mais curta de um vértice dado para todos os restantes vértices. Vamos supor, sem perda, de generalidade que se trata do vértice 1.

Organização do Algoritmo

- Todos os arcos com **pesos não negativos**
- Algoritmo Greedy
- Algoritmo mantém conjunto de vértices S com pesos dos caminhos mais curtos já calculados
- A cada passo algoritmo selecciona vértice u em $V - S$ com menor estimativa do peso do caminho mais curto
 - vértice u é inserido em S
 - arcos que saem de u são relaxados

Dijkstra(G, w, s)

```
1 Initialize-Single-Source( $G, s$ )
2  $S \leftarrow \emptyset$ 
3  $Q \leftarrow V[G]$                                 ▷ Fila de Prioridade
4 while  $Q \neq \emptyset$ 
5   do  $u \leftarrow \text{Extract-Min}(Q)$ 
6      $S \leftarrow S \cup \{u\}$ 
7     for each  $v \in \text{Adj}[u]$ 
8       do Relax( $u, v, w$ )                  ▷ Actualização de  $Q$ 
```

Inicialização

- Atribuir a cada vértice i uma etiqueta, que pode ser permanente ou provisória.
- A etiqueta permanente $L(i)$ do vértice i é a distância mais curta de 1 até i , ao passo que a etiqueta provisória $L'(i)$ de i é um majorante da distância referida.
- Em cada passo do processo, P vai ser o conjunto dos vértices com etiquetas permanentes e T o conjunto dos restantes vértices, ou seja, dos vértices que têm etiquetas provisórias.
- O processo inicia-se com $P = \{1\}$, $L(1) = 0$ e $L'(i) = c_{1i}$, para cada $i = 2, \dots, n$.

Em cada iteração

Passo 1 (Atribuição de uma etiqueta permanente): Determinar um vértice $k \in T$ tal que $L'(k)$ é mínimo. Parar se não existir esse vértice k , uma vez que isso quererá dizer que não existe caminho entre 1 e k . Caso contrário, acrescentar k ao conjunto P e fazer $L(k) = L'(k)$. Parar se $P = V$.

Passo 2 (Actualização das etiquetas provisórias): Se $j \in T$, actualizar o valor de $L'(j)$ para

$$L'(j) = \min\{L'(j), L'(k) + c_{kj}\}.$$

Regressar ao Passo 1.

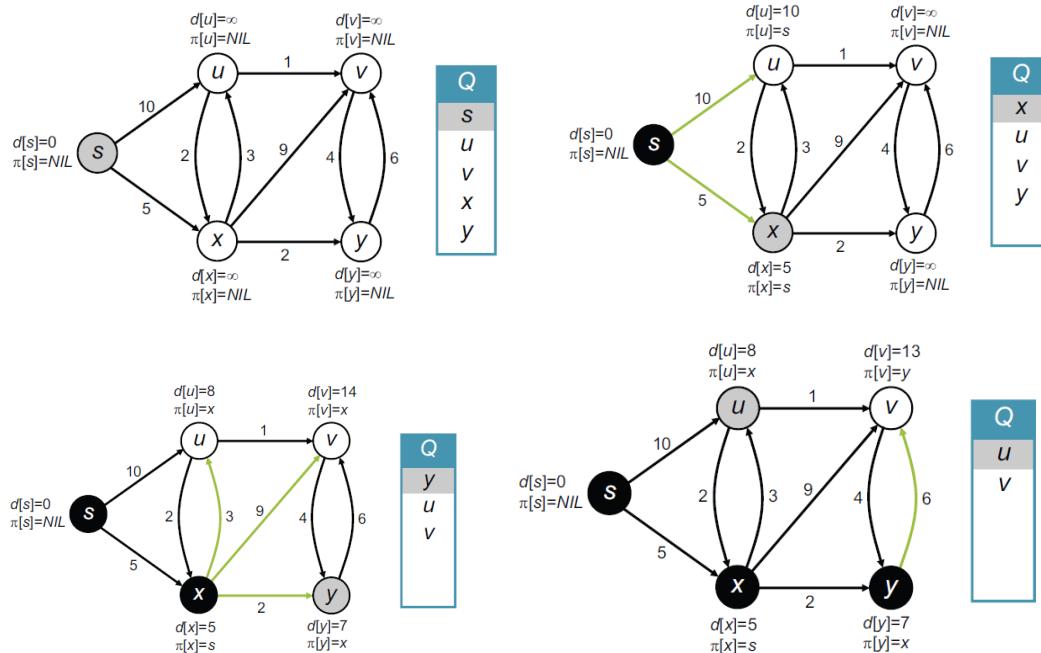
Teorema

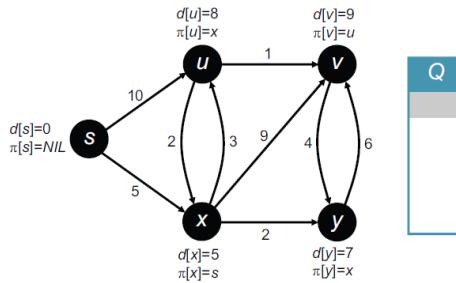
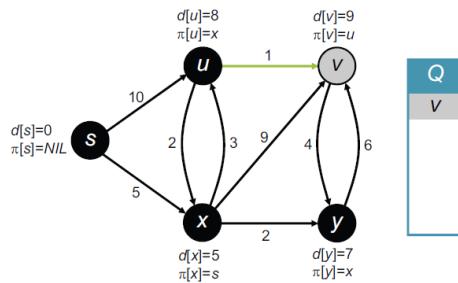
O algoritmo de Dijkstra determina a distância mais curta entre o vértice 1 e cada um dos restantes vértices i da rede dirigida.

Determinação do caminho mais curto

Uma vez conhecida a distância mais curta, o caminho mais curto entre o vértice 1 e o vértice i obtém-se examinando os vértices j tais que $L(j) < L(i)$ e existe um arco de j para i .

Exemplo 1 (<https://www.youtube.com/watch?v=XB4MIexjvY0>)





Complexidade

- Fila de prioridade baseada em amontoados (heap)
- Quando um vértice é extraído da fila Q , implica actualização de Q
 - Cada vértice é extraído apenas 1 vez $O(V)$
 - Actualização de Q : $O(\lg V)$
 - Então, $O(V \lg V)$
- Para cada arco (i.e. $O(E)$) operação de relaxação é aplicada apenas 1 vez. Cada operação de relaxação pode implicar uma actualização de Q em $O(\lg V)$
- Complexidade algoritmo Dijkstra: $O((V + E) \lg V)$

Correcção do Algoritmo

Provar invariante do algoritmo: $d[u] = \delta(s, u)$ quando u é adicionado ao conjunto S , e que a igualdade é posteriormente mantida

- Prova por contradição. Assume-se que existe um primeiro vértice u tal que $d[u] \neq \delta(s, u)$ quando u é adicionado a S
- Necessariamente temos que $u \neq s$ porque $d[s] = \delta(s, s) = 0$
- $S \neq \emptyset$ porque $s \in S$ quando u é adicionado a S
- Tem que existir caminho mais curto de s para u , dado que caso contrário teríamos $d[u] = \delta(s, u) = \infty$

Correcção do Algoritmo (2)

Pressuposto: u é o primeiro vértice tal que $d[u] \neq \delta(s, u)$ quando u é adicionado a S

- Seja $p = < s, \dots, x, y, \dots, u >$ o caminho mais curto de s para u
- Tem que existir pelo menos um vértice do caminho p que ainda não esteja em S , caso contrário, $d[u] = \delta(s, u)$ devido à relaxação dos arcos que compõem o caminho p

Correcção do Algoritmo (2)

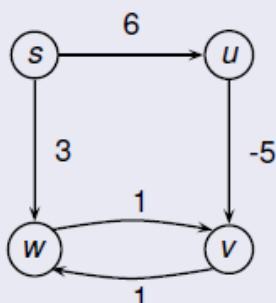
Pressuposto: u é o primeiro vértice tal que $d[u] \neq \delta(s, u)$ quando u é adicionado a S

- Seja $p = \langle s, \dots, x, y, \dots, u \rangle$ o caminho mais curto de s para u
- Tem que existir pelo menos um vértice do caminho p que ainda não esteja em S , caso contrário, $d[u] = \delta(s, u)$ devido à relaxação dos arcos que compõem o caminho p
- Seja (x, y) um arco de p tal que $x \in S$ e $y \notin S$
 - Temos que $d[x] = \delta(s, x)$ porque $x \in S$ e u é o primeiro vértice em que isso não ocorre
 - Temos também que $d[y] = \delta(s, y)$ porque o arco (x, y) foi relaxado quando x foi adicionado a S
 - Como y é predecessor de u no caminho mais curto até u , então $\delta(s, y) \leq \delta(s, u)$, porque os pesos dos arcos são não-negativos
 - Mas se u é adicionado a S antes de y , temos que $d[u] \leq d[y]$. Logo, $d[u] \leq \delta(s, y) \leq \delta(s, u)$. O que contradiz o pressuposto de $d[u] \neq \delta(s, u)$.

Pesos Negativos no Grafo

Os pesos do grafo têm que ser **não-negativos** para garantir a correcção do algoritmo

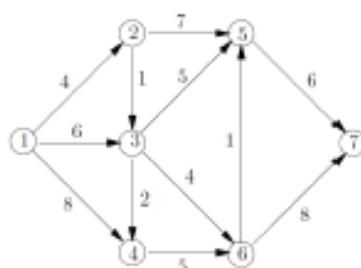
Exemplo



Execução do Algoritmo Dijkstra:

- Analisar w com $d[w] = 3$
- Analisar v com $d[v] = 4$
- Analisar u com $d[u] = 6$
- No final temos que $d[w] = 3 \neq \delta(s, w) = 2$

Exemplo 2 (resolução alternativa)



Interação 1

Passo 1

$$P = \{1\}, T = \{2, 3, 4, 5, 6, 7\},$$

$$L(1) = 0, \begin{cases} L'(2) = 4 \\ L'(3) = 6 \\ L'(4) = 8 \\ L'(5) = - \\ L'(6) = - \\ L'(7) = - \end{cases}$$

Ao vértice 2 é atribuída uma etiqueta permanente.

Passo 2

$$P = \{1, 2\}, T = \{3, 4, 5, 6, 7\},$$

$$\begin{cases} L(1) = 0 \\ L(2) = 4 \end{cases}, \begin{cases} L'(3) = \min\{6, L'(2) + c_{23}\} = 5 \\ L'(4) = \min\{8, L'(2) + -\} = 8 \\ L'(5) = \min\{-, L'(2) + c_{25}\} = 11 \\ L'(6) = \min\{-, L'(2) + -\} \\ L'(7) = \min\{-, L'(2) + -\} \end{cases}$$

Interação 2

Passo 1

$$P = \{1, 2\}, T = \{3, 4, 5, 6, 7\},$$

$$\begin{cases} L(1) = 0 \\ L(2) = 4 \end{cases}, \begin{cases} L'(3) = 5 \\ L'(4) = 8 \\ L'(5) = 11 \\ L'(6) = - \\ L'(7) = - \end{cases}$$

Ao vértice 3 é atribuída uma etiqueta permanente.

Passo 2

$$P = \{1, 2, 3\}, T = \{4, 5, 6, 7\},$$

$$\begin{cases} L(1) = 0 \\ L(2) = 4 \\ L(3) = 5 \end{cases}, \begin{cases} L'(4) = \min\{8, L'(3) + c_{34}\} = 7 \\ L'(5) = \min\{11, L'(3) + c_{35}\} = 10 \\ L'(6) = \min\{-, L'(3) + c_{36}\} = 9 \\ L'(7) = \min\{-, L'(3) + -\} \end{cases}$$

Interação 3

Passo 1

$$P = \{1, 2, 3\}, T = \{4, 5, 6, 7\},$$

$$\begin{cases} L(1) = 0 \\ L(2) = 4 \\ L(3) = 5 \end{cases}, \begin{cases} L'(4) = 7 \\ L'(5) = 10 \\ L'(6) = 9 \\ L'(7) = - \end{cases}$$

Ao vértice 4 é atribuída uma etiqueta permanente.

Passo 2

$$P = \{1, 2, 3, 4\}, T = \{5, 6, 7\},$$

$$\begin{cases} L(1) = 0 \\ L(2) = 4 \\ L(3) = 5 \\ L(4) = 7 \end{cases}, \begin{cases} L'(5) = \min\{10, L'(4) + -\} = 10 \\ L'(6) = \min\{9, L'(4) + c_{46}\} = 9 \\ L'(7) = \min\{-, L'(4) + -\} \end{cases}$$

Interação 4

Passo 1

$$P = \{1, 2, 3, 4\}, T = \{5, 6, 7\},$$

$$\begin{cases} L(1) = 0 \\ L(2) = 4 \\ L(3) = 5 \\ L(4) = 7 \end{cases}, \quad \begin{cases} L'(5) = 10 \\ L'(6) = 9 \\ L'(7) = - \end{cases}$$

Ao vértice 6 é atribuída uma etiqueta permanente.

Passo 2

$$P = \{1, 2, 3, 4, 6\}, T = \{5, 7\},$$

$$\begin{cases} L(1) = 0 \\ L(2) = 4 \\ L(3) = 5 \\ L(4) = 7 \\ L(6) = 9 \end{cases}, \quad \begin{cases} L'(5) = \min\{10, L'(6) + c_{65}\} = 10 \\ L'(7) = \min\{-, L'(6) + c_{67}\} = 17 \end{cases}$$

Interação 5

Passo 1

$$P = \{1, 2, 3, 4, 6\}, T = \{5, 7\},$$

$$\begin{cases} L(1) = 0 \\ L(2) = 4 \\ L(3) = 5 \\ L(4) = 7 \\ L(6) = 9 \end{cases}, \quad \begin{cases} L'(5) = 10 \\ L'(7) = 17 \end{cases}$$

Ao vértice 5 é atribuída uma etiqueta permanente.

Passo 2

$$P = \{1, 2, 3, 4, 5, 6\}, T = \{7\},$$

$$\begin{cases} L(1) = 0 \\ L(2) = 4 \\ L(3) = 5 \\ L(4) = 7 \\ L(5) = 10 \\ L(6) = 9 \end{cases}, \quad L'(7) = \min\{17, L'(5) + c_{57}\} = 16$$

Interação 6

Passo 1

$$P = \{1, 2, 3, 4, 5, 6\}, T = \{7\},$$

$$\begin{cases} L(1) = 0 \\ L(2) = 4 \\ L(3) = 5 \\ L(4) = 7 \\ L(5) = 10 \\ L(6) = 9 \end{cases}, \quad L'(7) = 16$$

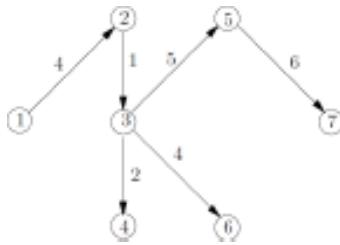
Ao vértice 7 é atribuída uma etiqueta permanente.

Passo 2

$$P = \{1, 2, 3, 4, 5, 6, 7\}, T = \emptyset,$$

$$\begin{cases} L(1) = 0 \\ L(2) = 4 \\ L(3) = 5 \\ L(4) = 7 \\ L(5) = 10 \\ L(6) = 9 \\ L(7) = 16 \end{cases}$$

Construção de uma árvore de cainhos mais curtos



Processo

Para cada vértice i distinto do vértice 1, determina-se um vértice j tal que existe um arco de j para i , $L(j) < L(i)$ e $L(j) + c_{ji} = L(i)$, e inclui-se o arco (j, i) nessa árvore.

4 Algoritmo Bellman-Ford

(<https://www.youtube.com/watch?v=FtN3BYH2Zes>)

(https://www.youtube.com/watch?v=FtN3BYH2Zes&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=53)

(https://www.youtube.com/watch?v=2raV0H9KqY8&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=80)

Organização do Algoritmo

- Permite pesos negativos e identifica existência de ciclos negativos
- Baseado em sequência de passos de relaxação
- Apenas requer manutenção da estimativa associada a cada vértice

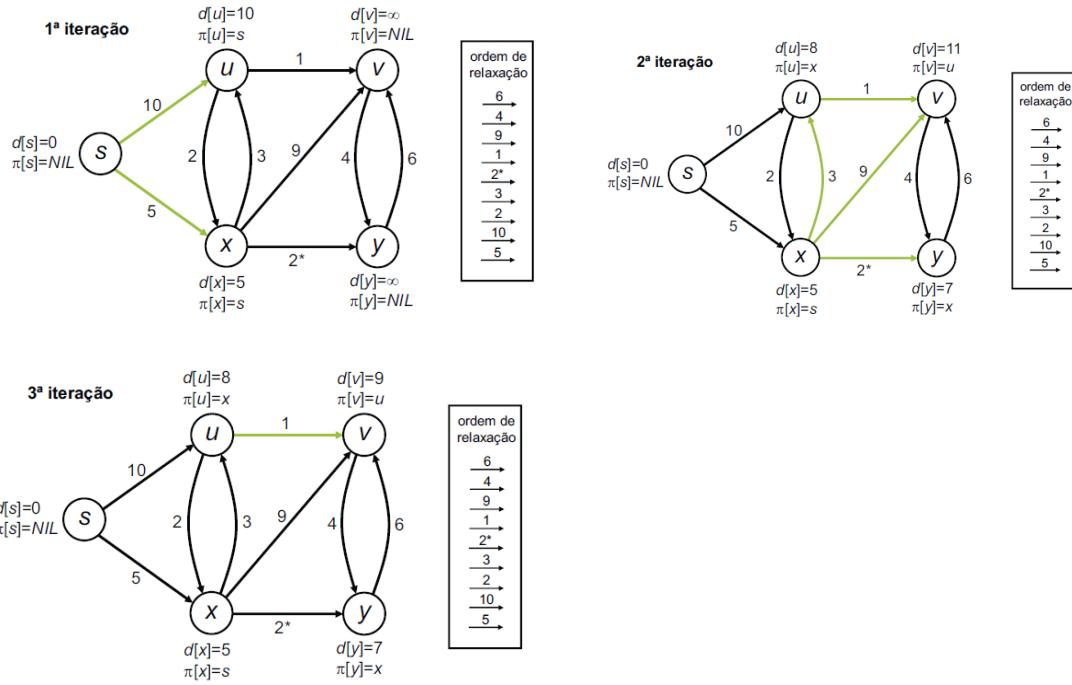
Bellman-Ford(G, w, s)

```

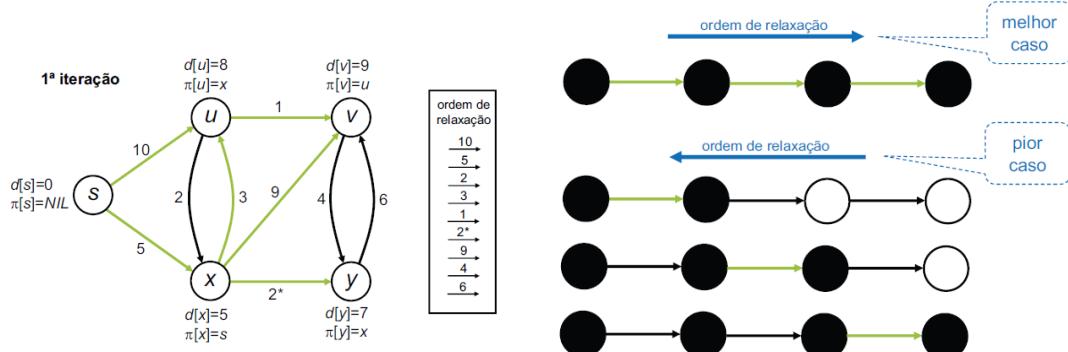
1 Initialize-Single-Source( $G, s$ )
2 for  $i \leftarrow 1$  to  $|V[G]| - 1$ 
3   do for each  $(u, v) \in E[G]$ 
4     do Relax( $u, v, w$ )
5   for each  $(u, v) \in E[G]$ 
6     do if  $d[v] > d[u] + w(u, v)$ 
7       then return FALSE           ▷ Ciclo negativo
8   return TRUE                   ▷ Sem ciclos negativos

```

Exemplo



Exemplo



Complexidade

- Inicialização: $\Theta(V)$
- A complexidade do ciclo `for` é $O(VE)$ devido aos dois ciclos, em V e em E.
 - Em cada iteração todos os arcos são relaxados
- Complexidade algoritmo Bellman-Ford: $O(VE)$

Correcção

Se $G = (V, E)$ não contém ciclos negativos, então após a aplicação do algoritmo de Bellman-Ford, $d[v] = \delta(s, v)$ para todos os vértices atingíveis a partir de s

- Seja v atingível a partir de s , e seja $p = < v_0, v_1, \dots, v_k >$ um caminho mais curto entre s e v , com $v_0 = s$ e $v_k = v$
- p é simples, pelo que $k \leq |V| - 1$
- Prova: provar por indução que $d[v_i] = \delta(s, v_i)$ para $i = 0, 1, \dots, k$, após iteração i sobre os arcos de G , e que valor não é alterado posteriormente
 - Base: $d[v_0] = \delta(s, v_0) = 0$ após inicialização (e não se altera)
 - Passo indutivo: assumir $d[v_{i-1}] = \delta(s, v_{i-1})$ após iteração $(i-1)$
 - Arco (v_{i-1}, v_i) relaxado na iteração i , pelo que $d[v_i] = \delta(s, v_i)$ após iteração i (e não se altera)

Correcção (2)

Se $G = (V, E)$ não contém ciclos negativos, o algoritmo de Bellman-Ford retorna TRUE. Se o grafo contém algum ciclo negativo atingível a partir de s , o algoritmo retorna FALSE

- Se não existem ciclos negativos, resultado anterior assegura que para qualquer arco $(u, v) \in E$, $d[v] \leq d[u] + w(u, v)$, pelo que teste do algoritmo falha para todo o (u, v) e o valor retornado é TRUE
- Caso contrário, na presença de pelo menos um ciclo negativo atingível a partir de s , $c = < v_0, v_1, \dots, v_k >$, onde $v_0 = v_k$, temos que $\sum_{i=1}^k w(v_{i-1}, v_i) < 0$

Correcção (3)

Se $G = (V, E)$ não contém ciclos negativos, o algoritmo de Bellman-Ford retorna TRUE. Se o grafo contém algum ciclo negativo atingível a partir de s , o algoritmo retorna FALSE

- Prova por contradição. Admitir que algoritmo retorna TRUE na presença de ciclo negativo. Para que devolva TRUE é necessário que $d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i)$, para $i = 1, \dots, k$
- Somando as desigualdades ao longo do ciclo temos que:

$$\sum_{i=1}^k d[v_i] \leq \sum_{i=1}^k d[v_{i-1}] + \sum_{i=1}^k w(v_{i-1}, v_i)$$

- Note-se que $\sum_{i=1}^k d[v_i] = \sum_{i=1}^k d[v_{i-1}]$ por ser um ciclo
- Temos então que $\sum_{i=1}^k w(v_{i-1}, v_i) \geq 0$, o que contradiz a existência de um ciclo negativo. Logo, o algoritmo retorna FALSE

5 Caminhos mais curtos em DAGs

DAG-Shortest-Path(G, w, s)

- 1 Ordenação topológica dos vértices de G
- 2 Initialize-Single-Source(G, s)
- 3 **for each** $u \in V[G]$ por ordem topológica
- 4 **do for each** $v \in Adj[u]$
- 5 **do Relax(u, v, w)**

Complexidade: $O(V + E)$

Correcção

Dado $G = (V, E)$, dirigido, acíclico, como resultado do algoritmo, temos que $d[v] = \delta(s, v)$ para todo o $v \in V$

- Seja v atingível a partir de s , e seja $p = < v_0, v_1, \dots, v_k >$ um caminho mais curto entre s e v , com $v_0 = s$ e $v_k = v$
- Ordenação topológica implica que analisados por ordem $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$
- Prova por indução: $d[v_i] = \delta(s, v_i)$ sempre que cada vértice v_i é terminado
 - Base: Estimativa de s não alterada após inicialização;
 $d[s] = d[v_0] = \delta(s, v_0) = 0$
 - Indução: $d[v_{i-1}] = \delta(s, v_{i-1})$ após terminar análise de v_{i-1}
 - Relaxação do arco (v_{i-1}, v_i) causa $d[v_i] = \delta(s, v_i)$, pelo que
 $d[v_i] = \delta(s, v_i)$ após terminar análise de v_i

Resumo

Algoritmo Dijkstra

- Só permite pesos não negativos
- Complexidade: $O((V + E) \lg V)$

Algoritmo Bellman-Ford

- Permite pesos negativos e identifica ciclos negativos
- Complexidade: $O(VE)$

Caminhos mais curtos em DAGs

- Grafos aciclicos. Podemos fazer ordenação topológica dos vértices
- Complexidade: $O(V + E)$

Duas variantes

- Determinação do caminho de custo mínimo (ou de distância mínima), dito *caminho mais curto*, entre dois vértices v e w dados → Algoritmo de Dijkstra;
- Determinação do caminho mais curto de cada vértice para todos os restantes vértices → Algoritmo de Floyd–Marshall.

1 Motivação

Problema

Considere que está a gerir um serviço para manipular um conjunto de dispositivos físicos de backup. Existe um conjunto de correias transportadoras com intercepções e braços robot para manipular os dispositivos.

O serviço recebe pedidos tais como "transferir a tape X do armário 25 para o armário 73".

O serviço deve estar automatizado por forma a conseguir satisfazer todos os possíveis pedidos de transferência de dispositivos.

2 Definições

Caminhos Mais Curtos Entre Todos os Pares

Encontrar caminhos mais curtos entre todos os pares de vértices

- Se **pesos não negativos**, utilizar algoritmo de Dijkstra, assumindo cada vértice como fonte: $O(V(V+E)\lg V)$ (que é $O(V^3\lg V)$ se o grafo é denso)
- Se existem **pesos negativos**, utilizar algoritmo de Bellman-Ford, assumindo cada vértice como fonte: $O(V^2E)$ (que é $O(V^4)$ se o grafo é denso)
- Objectivo: Encontrar algoritmos mais eficientes

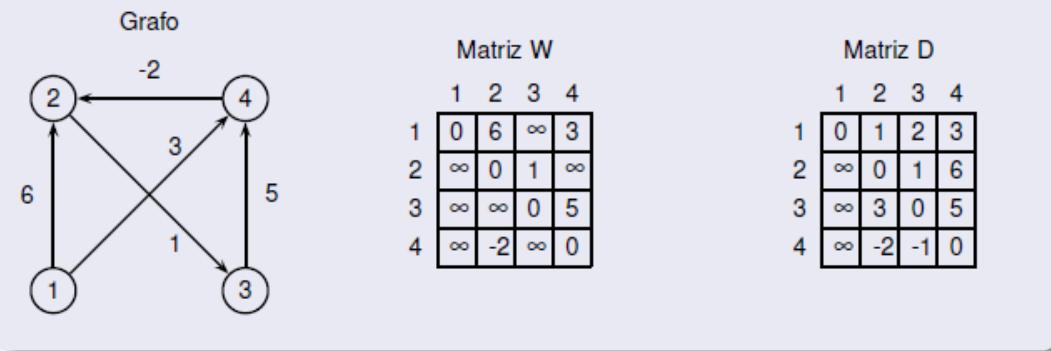
Caminhos Mais Curtos Entre Todos os Pares

- Representação do grafo: utilização de **matriz de adjacências**
- Pesos dos arcos: matriz $W(n \times n)$

$$w_{ij} = \begin{cases} 0 & \text{se } i=j \\ \text{peso do arco } (i,j) & \text{se } i \neq j, (i,j) \in E \\ \infty & \text{se } i \neq j, (i,j) \notin E \end{cases}$$

- Representação dos caminhos mais curtos: matriz $D(n \times n)$
 - d_{ij} é o peso do caminho mais curto entre os vértices i e j
 - $d_{ij} = \delta(v_i, v_j)$

Exemplo Representação



Representação dos Caminhos Mais Curtos

- Representação dos predecessores: matriz Π ($n \times n$)
- $\pi_{ij} = \text{NIL}$ se $i = j$ ou não existe caminho de i para j
- Caso contrário: π_{ij} denota o predecessor de j num caminho mais curto de i para j
- Sub-grafo de predecessores $G_{\pi,i} = (V_{\pi,i}, E_{\pi,i})$:

$$V_{\pi,i} = \{j \in V : \pi_{ij} \neq \text{NIL}\} \cup \{i\}$$

$$E_{\pi,i} = \{(\pi_{ij}, j) \in E : j \in V_{\pi,i} - \{i\}\}$$

- Sub-grafo de predecessores $G_{\pi,i}$ é induzido pela linha i da matriz Π

3 Solução Recursiva

Solução Recursiva

- Propriedade de sub-estrutura óptima dos caminhos mais curtos:
Sub-caminhos de caminhos mais curtos são também caminhos mais curtos
- $d_{ij}^{(m)}$: denota o peso mínimo dos caminhos do vértice i para o vértice j não contendo mais do que m arcos
- Com $m = 0$, existe caminho de i para j se e só se $i = j$

$$d_{ij}^{(0)} = \begin{cases} 0 & \text{se } i = j \\ \infty & \text{se } i \neq j \end{cases}$$

- Para $m \geq 1$:

$$d_{ij}^{(m)} = \min\{d_{ij}^{(m-1)}, \min_{1 \leq k \leq n} \{d_{ik}^{(m-1)} + w_{kj}\}\}$$

$$d_{ij}^{(m)} = \min_{1 \leq k \leq n} \{d_{ik}^{(m-1)} + w_{kj}\} \text{ porque } w_{jj} = 0$$

Pseudo-Código

- Calcular sequência de matrizes $D^{(1)}, \dots, D^{(n-1)}$, onde $D^{(n-1)}$ contém os pesos dos caminhos mais curtos
- Note-se que $D^{(1)} = W$

Extend-Shortest-Paths(D, W)

```
1   $n = \text{rows}[W]$ 
2   $D'$ : matriz ( $n \times n$ )
3  for  $i = 1$  to  $n$ 
4      do for  $j = 1$  to  $n$ 
5          do  $d'_{ij} = \infty$ 
6              for  $k = 1$  to  $n$ 
7                  do  $d'_{ij} = \min(d'_{ij}, d_{ik} + w_{kj})$ 
8  return  $D'$ 
```

- Complexidade: $\Theta(n^3)$ para cada matriz; Total: $\Theta(n^4)$

Observações

- Genericamente: calcular $D^{(i)}$ em função de $D^{(i-1)}$ (e de W)
- Complexidade para cálculo de $D^{(n)}$: $\Theta(n^4)$
- É possível melhorar complexidade reduzindo número de matrizes calculadas: $O(n^3 \lg n)$
 - A cada iteração, calcular $D^{(2i)}$ em função de $D^{(i)}$ e de $D^{(i)}$

4 Algoritmo Floyd-Warshall

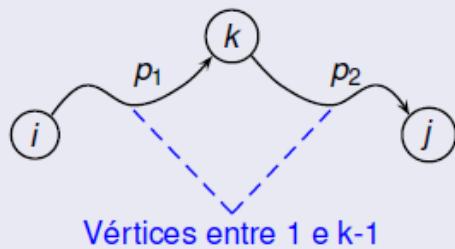
(https://www.youtube.com/watch?v=oNI0rf2P9gE&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=49)

- Fecho Transitivo

Conceitos

- Caracterização de um caminho mais curto $p = \langle v_1, v_2, \dots, v_k \rangle$
 - Vértices intermédios de caminho p são $\{v_2, \dots, v_{k-1}\}$
- Considerar todos os caminhos entre i e j com vértices intermédios retirados do conjunto $\{1, \dots, k\}$ e seja p um caminho mais curto (Nota: p é simples)
- Se k não é vértice intermédio de p , então todos os vértices intermédios de p estão em $\{1, \dots, k-1\}$
- Se k é vértice intermédio de p , então existem caminhos p_1 e p_2 , respectivamente de i para k e de k para j com vértices intermédios em $\{1, \dots, k\}$
 - k não é vértice intermédio de p_1 e de p_2
 - p_1 e p_2 com vértices intermédios em $\{1, \dots, k-1\}$

Formulação



$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{se } k = 0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{se } k \geq 1 \end{cases}$$

Características

- Indicado para redes dirigidas em que os arcos têm custos quaisquer;
- Informa da existência de eventuais ciclos com custo total negativo.

Objectivo

O algoritmo de Floyd–Marshall tem como objectivo a determinação do caminho mais curto e a distância mais curta de cada vértice para qualquer outro vértice da rede dirigida.

Pseudo-Código

Floyd-Warshall(D, W)

```
1   $n = \text{rows}[W]$ 
2   $D^{(0)} = W$ 
3  for  $k = 1$  to  $n$ 
4      do for  $i = 1$  to  $n$ 
5          do for  $j = 1$  to  $n$ 
6              do  $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$ 
7  return  $D^{(n)}$ 
```

Complexidade: $\Theta(n^3)$

Observação

Podemos evitar uma matriz por cada passo do algoritmo. A linha e a coluna k não são alteradas na iteração k :

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

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

Nota: $d_{kk}^{(k-1)} = 0$

Pseudo-Código

Floyd-Warshall(D, W)

```
1   $n = \text{rows}[W]$ 
2   $D = W$ 
3  for  $k = 1$  to  $n$ 
4      do for  $i = 1$  to  $n$ 
5          do for  $j = 1$  to  $n$ 
6              do  $d_{ij} = \min(d_{ij}, d_{ik} + d_{kj})$ 
7  return  $D$ 
```

Fecho Transitivo de um Grafo Dirigido

- Dado um grafo $G = (V, E)$ dirigido, o fecho transitivo é definido por $G^* = (V, E^*)$ tal que,

$$E^* = \{(i, j) : \text{existe caminho de } i \text{ para } j \text{ em } G\}$$

- Algoritmo:

- Atribuir a cada arco peso 1 e utilizar algoritmo de Floyd-Warshall
- Se $d_{ij} \neq \infty$, então $(i, j) \in E^*$
- Complexidade: $O(n^3)$

Algoritmo

Input: Matriz de pesos A e matriz $P = [p_{ij}]$ tal que $p_{ij} = j$

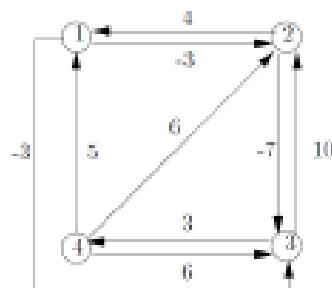
Output: Matrizes A e P modificadas

```

for j = 1, ..., n do
    for i = 1, ..., n do
        for k = 1, ..., n do
            if aik > aij + ajk then
                aik = aij + ajk
                pik = pij
            end
        end
    end
end

```

Exemplo



Inicialização das matrizes A e P

$$A^{(0)} = \begin{bmatrix} 0 & 4 & -3 & - \\ -3 & 0 & -7 & - \\ - & 10 & 0 & 3 \\ 5 & 6 & 6 & 0 \end{bmatrix}, \quad P^{(0)} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

Aplicação (<https://www.youtube.com/watch?v=oNI0rf2P9gE>)

$$A^{(1)} = \begin{bmatrix} 0 & 4 & -3 & - \\ -3 & 0 & -7 & - \\ - & 10 & 0 & 3 \\ 5 & 6 & 2 & 0 \end{bmatrix}, \quad P^{(1)} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 1 & 4 \end{bmatrix}$$

$$A^{(2)} = \begin{bmatrix} 0 & 4 & -3 & - \\ -3 & 0 & -7 & - \\ 7 & 10 & 0 & 3 \\ 3 & 6 & -1 & 0 \end{bmatrix}, \quad P^{(2)} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 2 & 2 & 3 & 4 \\ 2 & 2 & 2 & 4 \end{bmatrix}$$

$$A^{(3)} = \begin{bmatrix} 0 & 4 & -3 & - \\ -3 & 0 & -7 & -4 \\ 7 & 10 & 0 & 3 \\ 3 & 6 & -1 & 0 \end{bmatrix}, \quad P^{(3)} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 3 \\ 2 & 2 & 3 & 4 \\ 2 & 2 & 2 & 4 \end{bmatrix}$$

$$A' = A^{(4)} = \begin{bmatrix} 0 & 4 & -3 & 0 \\ -3 & 0 & -7 & -4 \\ 6 & 9 & 0 & 3 \\ 3 & 6 & -1 & 0 \end{bmatrix}, \quad P' = P^{(4)} = \begin{bmatrix} 1 & 2 & 3 & 3 \\ 1 & 2 & 3 & 3 \\ 4 & 4 & 3 & 4 \\ 2 & 2 & 2 & 4 \end{bmatrix}$$

Se a rede dirigida tiver n vértices, o algoritmo requer n iterações para calcular o caminho mais curto e a distância mais curta entre qualquer par de vértices, terminando com as matrizes $A' = A^{(n)}$ e $P' = P^{(n)}$.

É possível mostrar-se que o elemento na posição (i, j) da matriz A' é igual ao comprimento do caminho mais curto do vértice i para o vértice j , e que, se o elemento (i, j) da matriz P' é igual a k , então (i, k) é o primeiro arco do caminho mais curto referido, podendo-se usar esta informação para construir o caminho mais curto do vértice i para o vértice j .

Resultado

O comprimento do caminho mais curto entre os vértices 3 e 1 é o elemento $(3, 1)$ de A' , que é igual a 6. Por outro lado, o elemento na posição $(3, 1)$ em P' é igual a 4. Logo $(3, 4)$ é o primeiro arco desse caminho, e como o elemento na posição $(4, 1)$ em P' é 2, então o arco seguinte é $(4, 2)$. A seguir constatamos que o elemento na posição $(2, 1)$ é 1, pelo que o arco seguinte, e último do caminho, é o arco $(2, 1)$.

Outro exemplo

Digrafo com 4 vértices cuja matriz de pesos é dada por

$$A = \begin{bmatrix} 0 & - & - & 1 \\ 2 & 0 & 1 & - \\ - & - & 0 & - \\ - & -4 & 3 & 0 \end{bmatrix}$$

2^a iteração do algoritmo

$$A^{(2)} = \begin{bmatrix} 0 & - & - & 1 \\ 2 & 0 & 1 & 3 \\ - & - & 0 & - \\ -3 & -4 & 3 & -1 \end{bmatrix}, \quad P^{(2)} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 1 \\ 1 & 2 & 3 & 4 \\ 2 & 2 & 3 & 2 \end{bmatrix}$$

Conclusão

O elemento diagonal de $A^{(2)}$ na posição (4, 4) é negativo. Isto quer dizer que a rede tem um ciclo de custo total negativo, concluindo-se que o problema não tem solução óptima limitada, ou seja, que se consegue criar um caminho com um custo tão negativo quanto se quiser, bastando para isso percorrer esse ciclo um número arbitrário de vezes. Por outro lado, o elemento na posição (4, 4) da matriz $P^{(2)}$ é 2, obtendo-se o arco (4, 2); o elemento na posição (2, 4) é 1, obtendo-se o arco (2, 1), e o elemento na posição (1, 4) é 4, obtendo-se o arco (1, 4). Identificámos assim o ciclo $4 \rightarrow 2 \rightarrow 1 \rightarrow 4$ de custo total igual a -1 .

5 Algoritmo Johnson

Conceitos

- Utiliza algoritmos de Dijkstra e de Bellman-Ford
- Baseado em **re-pesagem dos arcos**
- Se arcos com pesos não negativos, utilizar Dijkstra para cada vértice
- Caso contrário, calcular novo conjunto de pesos não negativos w' , tal que
 - Um caminho mais curto de u para v com função w é também caminho mais curto com função w'
 - Para cada arco (u, v) o peso $w'(u, v)$ é não negativo

Re-pesagem dos arcos

- Dado $G = (V, E)$, com função de pesos w e de re-pesagem $h : V \rightarrow R$, seja $w'(u, v) = w(u, v) + h(u) - h(v)$
- Seja $p = \langle v_0, v_1, \dots, v_k \rangle$. Então $w(p) = \delta(v_0, v_k)$ se e só se $w'(p) = \delta'(v_0, v_k) = \delta(v_0, v_k) + h(v_0) - h(v_k)$
- Existe ciclo negativo com w se e só se existe ciclo negativo com w'
- Verificar que $w'(p) = w(p) + h(v_0) - h(v_k)$

$$\begin{aligned}w'(p) &= \sum_{i=1}^k w'(v_{i-1}, v_i) \\&= \sum_{i=1}^k (w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)) \\&= \sum_{i=1}^k w(v_{i-1}, v_i) + h(v_0) - h(v_k) \\&= w(p) + h(v_0) - h(v_k)\end{aligned}$$

Propriedade de re-pesagem

Caminhos mais curtos mantém-se após a repesagem. Se p é caminho mais curto com função de peso w , então também é caminho mais curto com função de peso w'

- Verificar que $w(p) = \delta(v_0, v_k) \rightarrow w'(p) = \delta'(v_0, v_k)$
- Hipótese: existe outro caminho mais curto p_z de v_0 para v_k após a re-pesagem em que $w'(p_z) < w'(p)$

$$w(p_z) + h(v_0) - h(v_k) = w'(p_z) < w'(p) = w(p) + h(v_0) - h(v_k)$$

- Para que isso seja verdade, temos que $w(p_z) < w(p)$, o que contradiz o facto de p ser caminho mais curto com função de peso w

Observação: Para quaisquer caminhos p_1, p_2 entre v_0 e v_k , verifica-se que $w(p_1) < w(p_2) \leftrightarrow w'(p_1) < w'(p_2)$

Propriedade de re-pesagem

Prova de que $w'(p) = \delta'(v_0, v_k) \rightarrow w(p) = \delta(v_0, v_k)$ é muito semelhante à anterior em que se admite existir um caminho mais curto p_z com função de peso w

Ciclos Negativos

Existe ciclo negativo com w se e só se existe com w'

- Considere-se que o caminho $p_c = < v_0, \dots, v_k >$ define um ciclo negativo. Então, $v_0 = v_k$ e $w(p_c) < 0$.
- $w'(p_c) = w(p_c) + h(v_0) - h(v_k) = w(p_c)$, dado que $v_0 = v_k$

Resumo

Caminhos mais curtos e ciclos negativos não se alteram com a mudança da função de peso $w'(u, v) = w(u, v) + h(u) - h(v)$

Organização

- Dado $G = (V, E)$, criar $G' = (V', E')$ definido do seguinte modo:
 - $V' = V \cup \{s\}$
 - $E' = E \cup \{(s, v) : v \in V\}$
 - $\forall v \in V : w(s, v) = 0$
- Ciclos negativos são detectados pela execução do algoritmo de Bellman-Ford aplicado a G'
- Se não existirem ciclos negativos:
 - Definir $h(v) = \delta(s, v)$
 - Pela propriedade dos caminhos mais curtos, para cada arco (u, v) , temos que $h(v) \leq h(u) + w(u, v)$
 - Logo, $w'(u, v) = w(u, v) + h(u) - h(v) \geq 0$
- Executar Dijkstra para todo o vértice $u \in V$ com função de peso w'
 - Cálculo $\delta'(u, v)$ para todo o $u \in V$
 - Mas também $\delta(u, v) = \delta'(u, v) - h(u) + h(v)$

Pseudo-Código

Johnson(G)

```
1 Representar  $G'$ 
2 if Bellman-Ford( $G', w, s$ ) = FALSE
3   then print "Indicar ciclo negativo"
4   else atribuir  $h(v) = \delta(s, v)$ , calculado com Bellman-Ford
5       calcular  $w'(u, v) = w(u, v) + h(u) - h(v)$  para cada arco  $(u, v)$ 
6       for each  $u \in V[G]$ 
7           do executar Dijkstra( $G, w', u$ ); calcular  $\delta'(u, v)$ 
8            $d_{uv} = \delta'(u, v) + h(v) - h(u)$ 
9 return D
```

Complexidade

- Bellman-Ford: $O(VE)$
- Executar Dijkstra para cada vértice: $O(V(V + E) \lg V)$ (assumindo amontoado binário)
- Total: $O(V(V + E) \lg V)$
- Útil para grafos esparsos

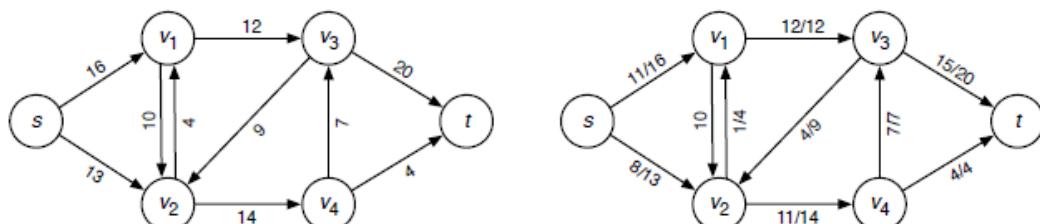
5.4. FLUXOS MÁXIMOS

1 Motivação

Problema: fornecer água a Lisboa

Pretende-se determinar qual o volume de água máximo (por segundo), que é possível fazer chegar a Lisboa a partir da Barragem de Castelo de Bode

- Existe uma rede de condutas de água que permitem o envio da água de Castelo de Bode para Lisboa
- Cada conduta apresenta uma capacidade limite, de metros cúbicos por segundo
- Objectivo: Encontrar um algoritmo eficiente para resolver este problema



Fluxos Máximos em Grafos

Dado um grafo dirigido $G = (V, E)$:

- Com um vértice fonte s e um vértice destino t
- Em que cada arco (u, v) é caracterizado por uma capacidade não negativa $c(u, v)$
- A capacidade de cada arco (u, v) indica o valor limite de "fluxo" que é possível enviar de u para v através do arco (u, v)
- Pretende-se calcular o valor máximo de "fluxo" que é possível enviar do vértice fonte s para o vértice destino t , respeitando as restrições de capacidade dos arcos

Aplicações

Envio de materiais em rede de transportes

- Água, petróleo ou gás
- Contentores
- Electricidade
- Bytes
- ...

Fluxos Máximos em Grafos

Para redes de fluxo com múltiplas fontes e/ou destinos

- Definir super-fonte que liga a todas as fontes
- Definir super-destino ao qual ligam todos os destinos
- Capacidades infinitas entre super-fonte e fontes, e entre destinos e super-destino

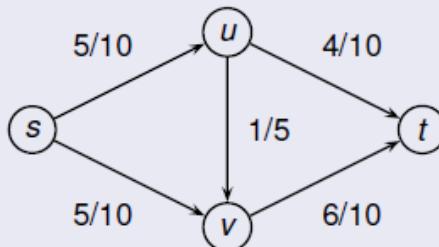
2 Definições

Fluxos Máximos em Grafos

- Uma **rede de fluxo** $G = (V, E)$ é um grafo dirigido em que cada arco (u, v) tem capacidade $c(u, v) \geq 0$
 - Se $(u, v) \notin E$, então $c(u, v) = 0$
- Dois vértices especiais: **fonte s** e **destino t**
- Todos os vértices de G pertencem a um caminho de s para t
- Grafo é ligado, $|E| \geq |V| - 1$
- Um fluxo $G = (V, E)$ é uma função $f : V \times V \rightarrow R$ tal que:
 - **Restrição de Capacidade:** $f(u, v) \leq c(u, v), \forall u, v \in V$
 - **Simetria:** $f(u, v) = -f(v, u), \forall u, v \in V$
 - **Conservação de Fluxo** $\sum_{v \in V} f(u, v) = 0, \forall u \in V - \{s, t\}$

Fluxos Máximos em Grafos

- Valor de um fluxo: $|f| = \sum_{v \in V} f(s, v)$
- **Problema do Fluxo Máximo:** Dada rede de fluxo G com fonte s e destino t , calcular o fluxo de valor máximo de s para t



Valor do Fluxo: 10

Fluxo Máximo: 20

Propriedades

- Dados conjuntos de vértices X e Y : $\sum_{x \in X} \sum_{y \in Y} f(x, y)$
- Considere rede de fluxo $G = (V, E)$, uma função de fluxo f em G e $X, Y, Z \subseteq V$:
 - $f(X, X) = 0$
 - $f(X, Y) = -f(Y, X)$
 - Se $X \cap Y = \emptyset$:
 - $f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$
 - $f(Z, X \cup Y) = f(Z, X) + f(Z, Y)$

3 Método Ford-Fulkerson

- Rede Residual
- Caminhos de Aumento
- Cortes em Redes de Fluxo

Método Ford-Fulkerson

- Redes residuais
- Caminhos de aumento
- Cortes em redes de fluxo
- Teorema do Fluxo-Máximo Corte-Mínimo
- Algoritmo Genérico de Ford-Fulkerson

Ford-Fulkerson-Method(G, s, t)

```

1 Inicializar fluxo  $f$  a 0
2 while existe caminho de aumento  $p$ 
3     do aumentar fluxo  $f$  utilizando  $p$ 
4 return  $f$ 

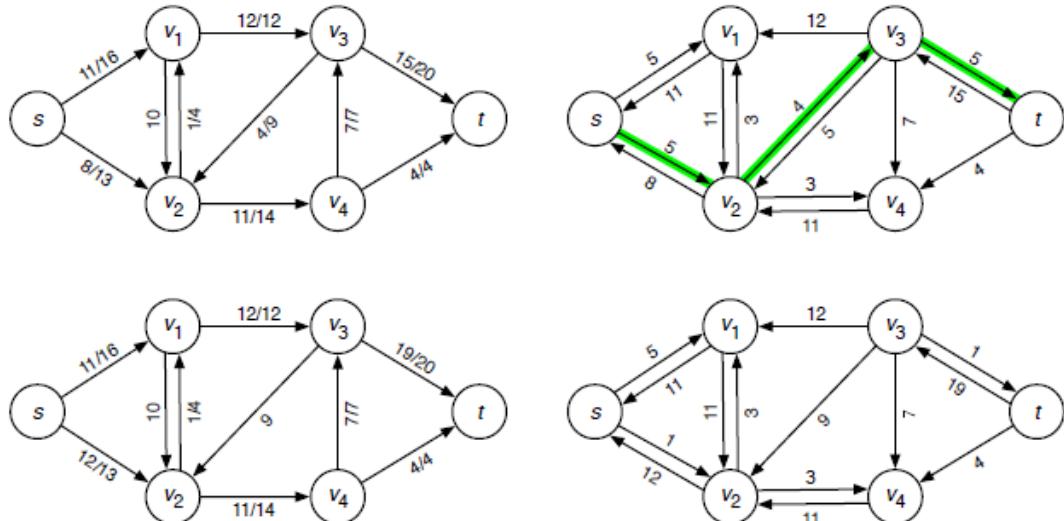
```

Rede Residual

Dado $G = (V, E)$, um fluxo f , e vértices $u, v \in V$

- $c_f(u, v)$ denota a **capacidade residual** de (u, v) : fluxo líquido adicional que é possível enviar de u para v (fluxo que sobra)
- $c_f(u, v) = c(u, v) - f(u, v)$
- $G_f = (V, E_f)$ denota a **rede residual** de G : onde $E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$
- Cada arco (residual) de G_f permite apenas fluxo líquido positivo

Exemplo



Fluxo de Soma

Seja $G = (V, E)$ uma rede de fluxo, f um fluxo, G_f a rede residual de G e f' um fluxo em G_f

- Fluxo de soma $f + f'$ definido para cada par $u, v \in V$:
$$(f + f')(u, v) = f(u, v) + f'(u, v)$$
- Fluxo de soma é um fluxo com valor $|f + f'| = |f| + |f'|$
- Propriedades de um fluxo são verificadas: restrição de capacidade, simetria e conservação de fluxo
- Observação: f' é definido em G_f e é um fluxo
- Cálculo do valor de fluxo

$$\begin{aligned}|f + f'| &= \sum_{v \in V} (f + f')(s, v) \\&= \sum_{v \in V} (f(s, v) + f'(s, v)) \\&= \sum_{v \in V} f(s, v) + \sum_{v \in V} f'(s, v) \\&= |f| + |f'|\end{aligned}$$

Caminho de Aumento

Seja $G = (V, E)$ uma rede de fluxo, f um fluxo e G_f a rede residual de G :

- **caminho de aumento** p :
 - caminho simples de s para t na rede residual G_f
- **capacidade residual** de p :
 - $c_f(p) = \min\{c_f(u, v) : (u, v) \text{ em } p\}$
- $c_f(p)$ permite definir fluxo f_p em G_f , $|f_p| = c_f(p) > 0$
- $f' = f + f_p$ é um fluxo em G , com valor $|f'| = |f| + |f_p| > |f|$

Corte em Rede de Fluxo

Um **corte** (S, T) de $G = (V, E)$ é uma partição de V em S e $T = V - S$, tal que $s \in S$ e $t \in T$

- fluxo líquido do corte (S, T) : $f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v)$
- capacidade do corte (S, T) : $c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v)$

Fluxo através de um Corte

Se $G = (V, E)$ com fluxo f , então o fluxo líquido através de um corte (S, T) é $f(S, T) = |f|$

- $V = T \cup S$, pelo que $f(S, V) = f(S, T \cup S) = f(S, T) + f(S, S)$
- Logo, $f(S, T) = f(S, V)$, dado que $f(S, S) = 0$
- $f(S, V) = f(s, V) + f(S - \{s\}, V) = f(s, V) = |f|$
- Obs: para $u \in S - \{s\}$ temos $f(u, V) = 0$ (conservação de fluxo nos vértices intermédios)

Corte em Rede de Fluxo

Qualquer valor de fluxo é **limitado superiormente** pela **capacidade** de qualquer corte de G

- Seja (S, T) qualquer corte de G , e f um fluxo:

$$|f| = f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) \leq \sum_{u \in S} \sum_{v \in T} c(u, v) = c(S, T)$$

Teorema Fluxo Máximo - Corte Mínimo

Seja $G = (V, E)$, com fonte s e destino t , e um fluxo f . Então as proposições seguintes são equivalentes:

- ① f é um fluxo máximo em G
- ② A rede residual G_f não contém caminhos de aumento
- ③ $|f| = c(S, T)$ para um corte (S, T) de G

1 \rightarrow 2

- Admitir que f é fluxo máximo em G mas que G_f tem caminho de aumento
- Então é possível definir um novo fluxo $f + f_p$ com valor $|f| + |f_p| > |f|$; uma contradição

Teorema Fluxo Máximo - Corte Mínimo

Seja $G = (V, E)$, com fonte s e destino t , e um fluxo f . Então as proposições seguintes são equivalentes:

- ① f é um fluxo máximo em G
- ② A rede residual G_f não contém caminhos de aumento
- ③ $|f| = c(S, T)$ para um corte (S, T) de G

$2 \rightarrow 3$

- $S = \{v \in V : \text{existe caminho de } s \text{ para } v \text{ em } G_f\}$; $T = V - S$; $s \in S$ e $t \in T$
- Com $u \in S$ e $v \in T$, temos $f(u, v) = c(u, v)$, pelo que $|f| = f(S, T) = c(S, T)$

Teorema Fluxo Máximo - Corte Mínimo

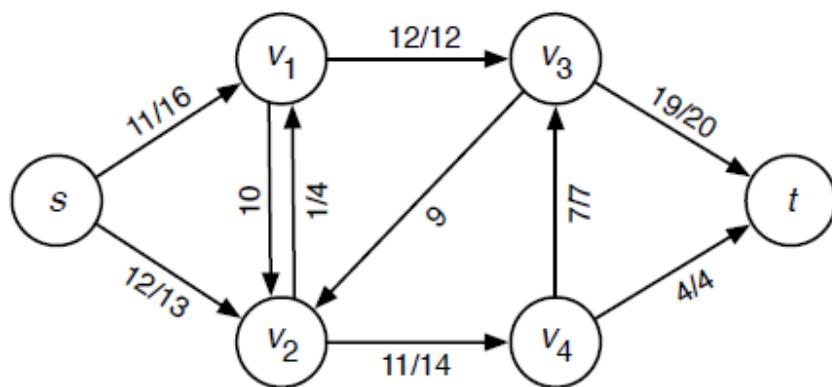
Seja $G = (V, E)$, com fonte s e destino t , e um fluxo f . Então as proposições seguintes são equivalentes:

- ① f é um fluxo máximo em G
- ② A rede residual G_f não contém caminhos de aumento
- ③ $|f| = c(S, T)$ para um corte (S, T) de G

$3 \rightarrow 1$

- Dado que $|f| \leq c(S, T)$, para qualquer corte (S, T) de G
- Como $|f| = c(S, T)$ (definido acima), então f é fluxo máximo

Exemplo

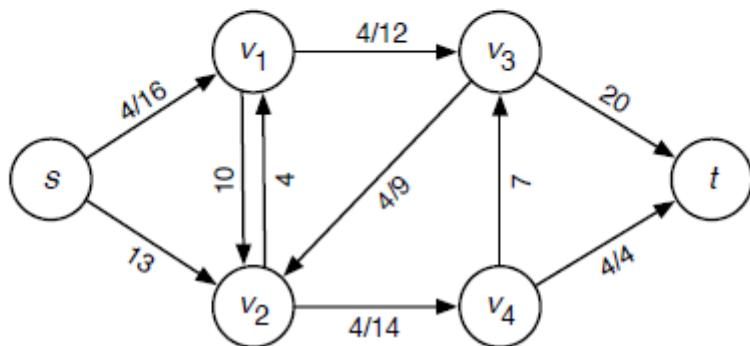
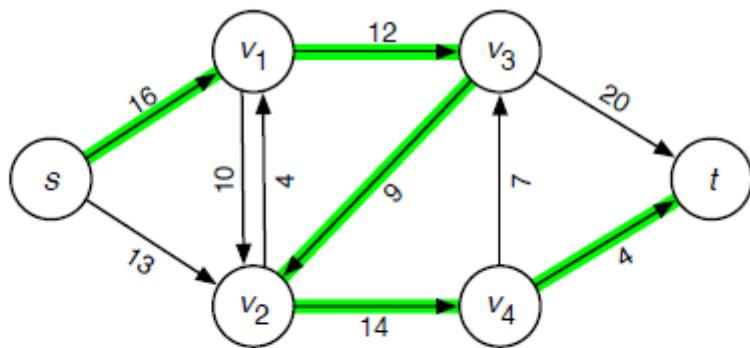


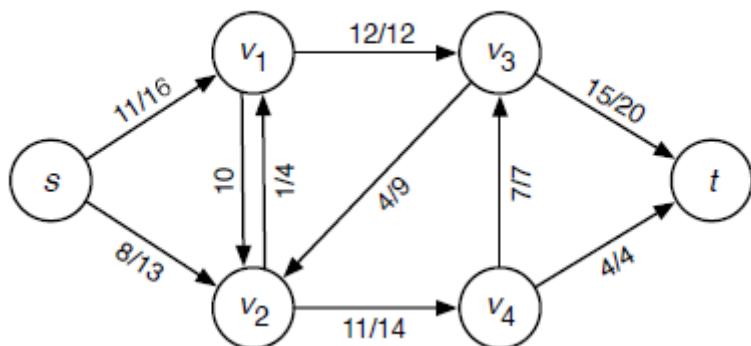
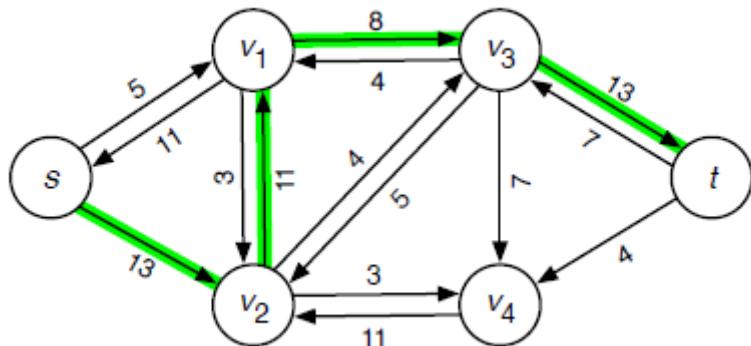
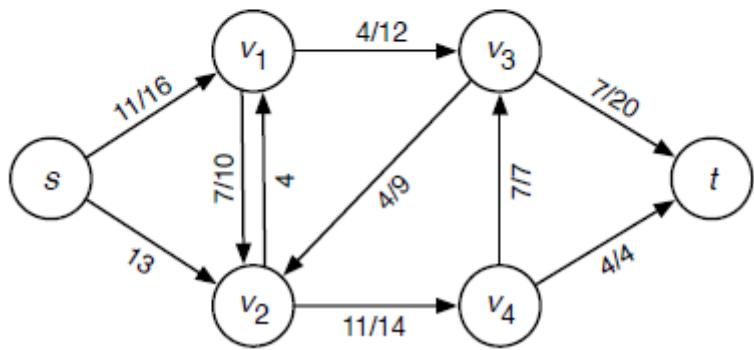
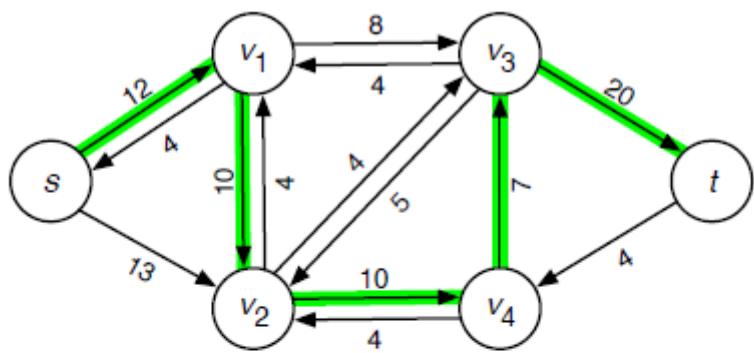
Corte mínimo? $(\{s, v_1, v_2, v_4\}, \{v_3, t\})$

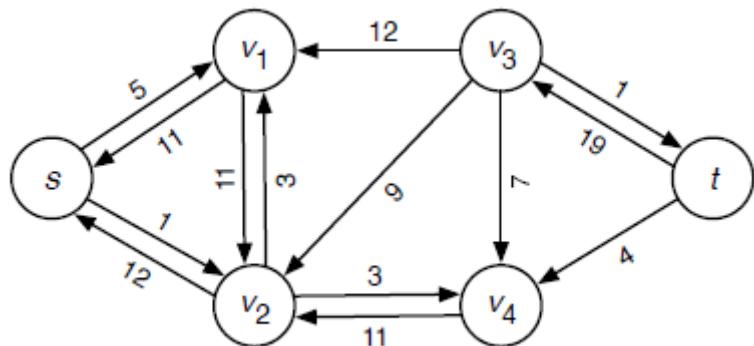
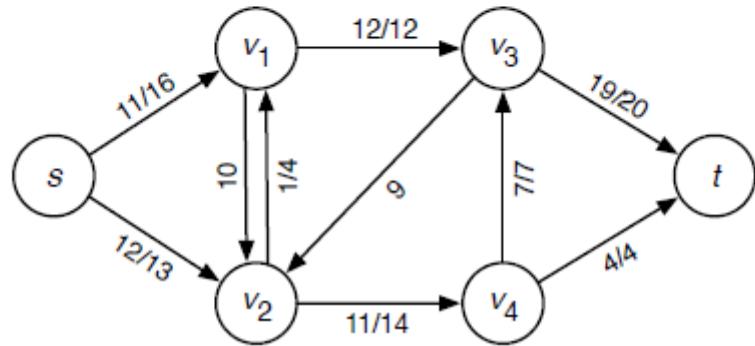
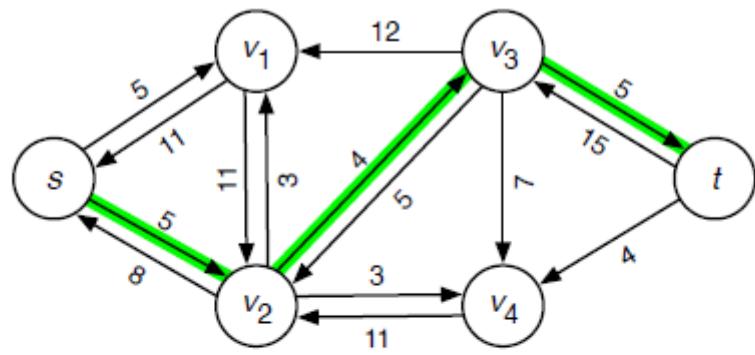
Ford-Fulkerson(G, s, t)

```
1  for each edge  $(u, v) \in E[G]$ 
2      do  $f[u, v] \leftarrow 0$ 
3           $f[v, u] \leftarrow 0$ 
4  while there exists a path  $p$  from  $s$  to  $t$  in  $G_f$ 
5      do  $c_f(p) \leftarrow \min\{c_f(u, v) : (u, v) \in p\}$ 
6          for each edge  $(u, v) \in p$ 
7              do  $f[u, v] \leftarrow f[u, v] + c_f(p)$ 
8                   $f[v, u] \leftarrow -f[u, v]$ 
```

Exemplo

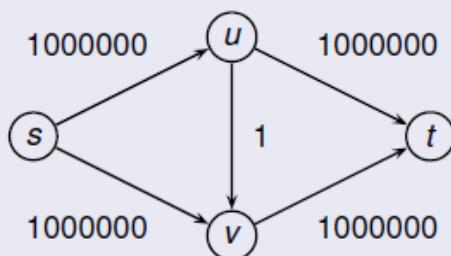




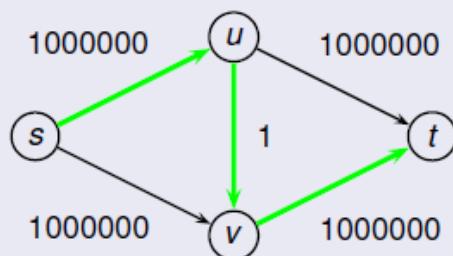


Análise Algoritmo Básico

- Número de aumentos de fluxo pode ser elevado
- Ex: Fluxo máximo = 2000000
- No pior caso: número de caminhos de aumento é 2000000



Rede de Fluxo



Caminho de aumento p com $c_f(p) = 1$

Análise Algoritmo Básico

- Número de caminhos de aumento limitado por valor máximo do fluxo $|f^*|$
- Complexidade: $O(E |f^*|)$
- Por exemplo: DFS para encontrar caminho de aumento

4 Algoritmo Edmonds-Karp

Algoritmo Edmonds-Karp

- Implementação do método de Ford-Fulkerson
- Escolher caminho de aumento mais curto no número de arcos
- Utilizar BFS em G_f para identificar caminho mais curto
- Complexidade: $O(V E^2)$

Análise Edmonds-Karp

Definições

- $\delta_f(s, v)$: distância mais curta de s para v na rede residual G_f
- $\delta_{f'}(s, v)$: distância mais curta de s para v na rede residual $G_{f'}$
- Sequência de eventos considerada:

$$f \rightarrow G_f \rightarrow BFS \rightarrow p \rightarrow f' \rightarrow G_{f'} \rightarrow BFS \rightarrow p'$$

Resultados:

- $\delta_f(s, v)$ cresce monotonamente com cada aumento de fluxo
- Número de aumentos de fluxo é $O(V E)$
- Tempo de execução é $O(V E^2)$
- $O(E)$ devido a BFS e aumento de fluxo a cada passo

Análise Edmonds-Karp

$\delta_f(s, v)$ cresce de forma monótona com cada aumento de fluxo

- Prova por contradição: considere-se o primeiro $v \in V$ tal que, após aumento de fluxo (de f para f'), a distância do caminho mais curto diminui, $\delta_{f'}(s, v) < \delta_f(s, v)$
- Seja $p = \langle s, \dots, u, v \rangle$ o caminho mais curto de s para v em $G_{f'}$
 - $\delta_{f'}(s, u) = \delta_f(s, v) - 1$
 - $\delta_{f'}(s, u) \geq \delta_f(s, u)$ (porque v é o primeiro que falha)
- $(u, v) \in E_f$: $\delta_f(s, v) \leq \delta_f(s, u) + 1 \leq \delta_{f'}(s, u) + 1 = \delta_{f'}(s, v)$
- $(u, v) \notin E_f$ e $(u, v) \in E_{f'}$: aumento de fluxo de v para u
 - Aumento sempre pelo caminho mais curto, então o caminho mais curto entre s e u em G_f tem como último arco (v, u) :
 - Então, $\delta_f(s, v) = \delta_f(s, u) - 1 \leq \delta_{f'}(s, u) - 1 = \delta_{f'}(s, v) - 2$ Contradição !

Análise Edmonds-Karp

Número de aumentos de fluxo é $O(V E)$

- arco (u, v) na rede residual G_f é **crítico** se capacidade residual de p é igual à capacidade residual do arco
 - arco crítico desaparece após aumento de fluxo
- Quantas vezes pode arco (u, v) ser arco crítico?
 - Como caminhos de aumento são caminhos mais curtos,
 $\delta_f(s, v) = \delta_f(s, u) + 1$
 - (u, v) só volta à rede residual após arco (v, u) aparecer em caminho de aumento (com fluxo f')

Como, $\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1$

Dado que, $\delta_f(s, v) \leq \delta_{f'}(s, v)$ (resultado anterior)

Obtém-se, $\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1 \geq \delta_f(s, v) + 1 = \delta_f(s, u) + 2$

Análise Edmonds-Karp

Número de aumentos de fluxo é $O(V E)$

- Distância de s a u aumenta pelo menos de duas unidades entre cada par de vezes que (u, v) é crítico
 - No limite, distância de s a u é não superior a $|V| - 2$
 - Pelo que arco (u, v) pode ser crítico $O(V)$ vezes
 - Existem $O(E)$ pares de vértices
 - Na execução do algoritmo de Edmonds-Karp o número total de vezes que arcos podem ser críticos é $O(V E)$
 - Como cada caminho de aumento tem um arco crítico, então existem $O(V E)$ caminhos de aumento

Análise Edmonds-Karp

Número de aumentos de fluxo é $O(V E)$

- Distância de s a u aumenta pelo menos de duas unidades entre cada par de vezes que (u, v) é crítico
 - No limite, distância de s a u é não superior a $|V| - 2$
 - Pelo que arco (u, v) pode ser crítico $O(V)$ vezes
 - Existem $O(E)$ pares de vértices
 - Na execução do algoritmo de Edmonds-Karp o número total de vezes que arcos podem ser críticos é $O(V E)$
 - Como cada caminho de aumento tem um arco crítico, então existem $O(V E)$ caminhos de aumento
- Complexidade de Edmonds-Karp é $O(V E^2)$
 - Complexidade de BFS é $O(V + E) = O(E)$
 - Aumento de fluxo em $O(E)$

5 Emparelhamento Bipartido Máximo

Emparelhamento Bipartido Máximo

Considere um grafo $G = (V, E)$ não dirigido

- **Emparelhamento:**

- $M \subseteq E$, tal que para qualquer vértice $v \in V$ não mais do que um arco em M é incidente em v

- **Emparelhamento Máximo:**

- Emparelhamento cardinalidade máxima (na dimensão de M)

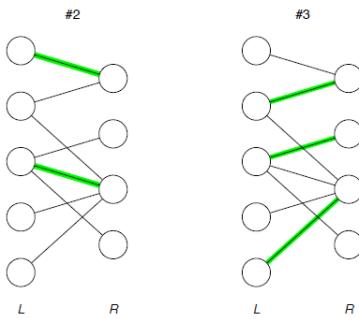
- **Grafo Bipartido:**

- Grafo pode ser dividido em $V = L \cup R$, em que L e R são disjuntos e em que todos os arcos de E estão entre L e R

- **Emparelhamento Bipartido Máximo:**

- Emparelhamento máximo em que G é bipartido

Exemplo



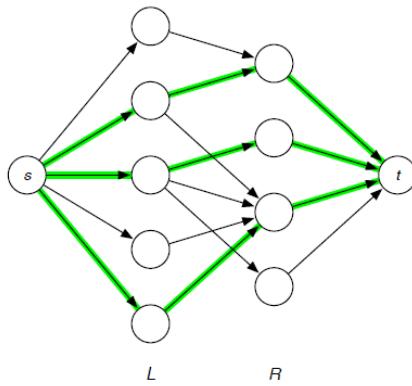
Utilização Redes de Fluxo

Construir grafo $G' = (V', E')$:

$$\begin{aligned}V' &= V \cup \{s, t\} \\E' &= \{(s, u) : u \in L\} \cup \\&\quad \{(u, v) : u \in L, v \in R, \text{ e } (u, v) \in E\} \cup \\&\quad \{(v, t) : v \in R\}\end{aligned}$$

- Atribuir capacidade unitária a cada arco de E'
- Emparelhamento bipartido máximo em G equivale a encontrar fluxo máximo em G'

Exemplo



Análise da Correcção

1. Dados G e G' , se M é um emparelhamento em G , existe um fluxo f de valor inteiro em G' , com $|f| = |M|$

- Seja M um emparelhamento, e $(u, v) \in M$
- Definir f utilizando arcos de M , $f(s, u) = f(u, v) = f(v, t) = 1$
- Para restantes arcos $(u, v) \in E'$, $f(u, v) = 0$
- Os caminhos $s \rightarrow u \rightarrow v \rightarrow t$ para todo o $(u, v) \in M$ são disjuntos em termos dos vértices, com excepção de s e t
- Como existem $|M|$ caminhos, cada um com uma contribuição de uma unidade de fluxo para o fluxo total f , $|f| = |M|$

Análise da Correcção

2. Dados G e G' , se $|f|$ é um fluxo de valor inteiro em G' , existe um emparelhamento M em G , com $|f| = |M|$

- Definir $M = \{(u, v) : u \in L, v \in R \text{ e } f(u, v) > 0\}$
 - Para cada $u \in L$, existe no máximo um $v \in R$ tal que $f(u, v) = 1$
 - Apenas um arco incidente com capacidade 1
 - Capacidades são inteiras
 - De forma simétrica para $v \in R$
- Logo M é um emparelhamento
- $|M| = f(L, R) = f(s, L) = f(s, V') = |f|$

Análise da Correcção

- Se todas as capacidades têm valor inteiro, então para fluxo máximo f , $|f|$ é inteiro
 - Indução no número de iterações do algoritmo genérico de Ford-Fulkerson
- Emparelhamento bipartido máximo $|M|$ em G corresponde a $|f|$, em que f é o fluxo máximo de G'
 - Se $|M|$ é emparelhamento máximo em G , e $|f|$ não é máximo em G' , então existe f' que é máximo
 - f' é inteiro, $|f'| > |f|$
 - e f' corresponde a emparelhamento $|M'|$, com $|M'| > |M|$; Contradição!

Análise da Complexidade

- A aplicação do algoritmo genérico de Ford-Fulkerson tem complexidade $O(E|f^*|)$
- Emparelhamento bipartido máximo é não superior a $\min(|L|, |R|) = O(V)$ e tem valor inteiro (i.e., no caso do emparelhamento máximo, $|f^*| = O(V)$)
- Complexidade de identificação do emparelhamento bipartido máximo é $O(VE)$

- 1-Aspetos introdutórios
- 2-Funções
- 3-Algoritmos de ordenação e de pesquisa
- 4-Pesquisa linear e pesquisa binária
- 5- Algoritmos sobre Grafos e Dígrafos
- 6-Extensões: noções básicas sobre heurísticas.

6.1 DEFINIÇÃO E PROPRIEDADES FUNDAMENTAIS

6.1.1 Árvore

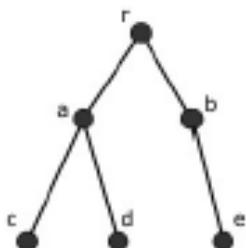
Definição

Dá-se o nome de árvore a todo e qualquer grafo conexo sem ciclos.

Propriedades

- Um grafo é uma árvore se e só se entre cada par de vértices existe um e um só caminho simples.
- Um grafo conexo é uma árvore se e só se cada sua aresta é uma ponte.
- Um grafo com n vértices é uma árvore se e só se tiver $n - 1$ arestas.

Ilustração



- Se r é raiz da árvore, então:
 - ▶ o nó r é **pai** dos nós a e b ;
 - ▶ os nós a e b são **filhos** de r ;
 - ▶ os nós c e d são **filhos** de a .

Terminologia

- nível;
- profundidade de um nó ou vértice (comprimento do caminho da raiz até ao nó);
- altura de uma árvore (maior das profundidades dos seus vértices);
- folha,

A) Árvores geradoras

Definição

O subgrafo T do grafo G diz-se uma *árvore geradora* de G se T é uma árvore e tem o mesmo número de vértices de G .

Propriedade

Um grafo é conexo se e só se contém uma árvore geradora.

Nota importante

O algoritmo DFS pode ser usado para obter uma árvore geradora de um grafo conexo.

B) Árvores dirigidas

Árvore dirigida

Um digrafo diz-se uma *árvore dirigida* se o seu grafo correspondente é uma árvore.

Árvore com raiz

Uma árvore dirigida diz-se uma *árvore com raiz* se :

- existe um vértice (denominado *raiz*) cujo grau de entrada é zero;
- todos os restantes vértices têm grau de entrada igual a um.

Definições

- Um vértice de uma árvore com raiz diz-se um vértice *terminal* (ou uma *folha*) se o seu grau de saída é igual a zero.
- Qualquer vértice que não seja uma folha nem uma raiz diz-se um vértice *intermédio* ou *interno*.
- Dá-se o nome de *nível* de um vértice de uma árvore dirigida com raiz ao número de arcos desde a raiz até esse vértice. Por definição, a raiz tem nível zero. Se os níveis de uma árvore estiverem dispostos como $0, 1, \dots, k$ então diz-se que k é a profundidade da árvore.
- O vértice v diz-se *descendente* do vértice u de uma árvore dirigida se existe um arco de u para v .

C) Árvores binárias

Árvore binária

Uma árvore dirigida diz-se ser *binária* se o grau de saída de cada vértice é no máximo igual a dois, e diz-se ser *binária regular* se o grau de saída de cada vértice intermédio é exactamente igual a dois. Uma árvore binária regular diz-se *completa* se as suas folhas estão todas no mesmo nível.

6.2 ÁRVORES GERADORAS DE PESO OU CUSTO MÍNIMO NUMA REDE

Rede

Se associarmos um número real, denominado *peso* ou *custo*, a cada aresta de um grafo, obtemos uma *rede*. O peso ou custo de uma rede é a soma dos custos das arestas que a constituem. Em particular, o peso ou custo de uma árvore geradora T de uma rede G é a soma dos pesos das arestas de T .

Problema da árvore geradora de peso ou custo mínimo, também conhecida por árvore geradora minimal

Determinar uma árvore geradora T da rede G cujo peso não excede o peso de qualquer outra árvore geradora nessa rede.

6.3 ÁRVORES ABANGENTES

1 Definições

- Estrutura de dados para conjuntos disjuntos
- Operações

Estrutura de Dados para Conjuntos Disjuntos

Permite manter uma colecção de **conjuntos dinâmicos disjuntos**

- Cada conjunto caracterizado por **representante**, um elemento do conjunto
- Representante não alterado devido a consultas à estrutura de dados

Operações sobre Conjuntos Disjuntos

Cada elemento da estrutura é representado por objecto x

- **Make-Set(x)**
 - Cria novo conjunto que apenas inclui elemento apontado por x
 - x aponta para único elemento do conjunto, o representante do conjunto
- **Union(x,y)**
 - Realiza a união dos conjuntos que contém x e y, respectivamente S_x e S_y
 - Novo conjunto criado: $S_x \cup S_y$
 - S_x e S_y eliminados (conjuntos disjuntos)
 - Novo representante será o representante de S_x ou S_y
- **Find-Set(x)**
 - Retorna apontador para representante do conjunto que contém x

2 Aplicações

Problema

Considere que está na equipa de projecto de uma rede de distribuição entre um conjunto de cidades. Foram efectuados estudos que calcularam o custo $c(u, v)$ associado a cada ligação possível da nova rede. Pretende-se saber qual o menor custo total de uma rede que interligue todas as cidades.

Solução

- Representar a rede como um grafo não-dirigido e pesado
- Função de pesos é definida pelo custo entre as possíveis ligações
- Rede de menor custo será a Árvore Abrangente de Menor Custo (MST) do grafo

Algoritmo Kruskal (Cap. 23)

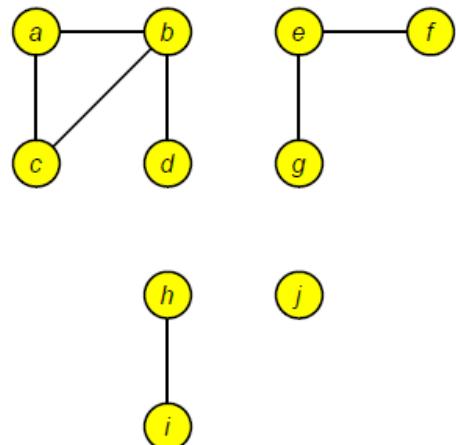
- **Make-Set:** Cria cada conjunto (disjunto) que representa conjunto de vértices
- **Find-Set:** Identifica conjunto a que pertence um dado vértice
- **Union:** Coloca conjuntos de vértices num mesmo conjunto

Exemplo Aplicação

Identificar os componentes ligados de um grafo não dirigido $G = (V, E)$

Connected-Components(G)

```
1 for each  $v \in V[G]$ 
2   do Make-Set( $v$ )
3 for each  $(u, v) \in E[G]$ 
4   do if Find-Set( $u$ )  $\neq$  Find-Set( $v$ )
5     then Union( $u, v$ )
```



Same-Component(u, v)

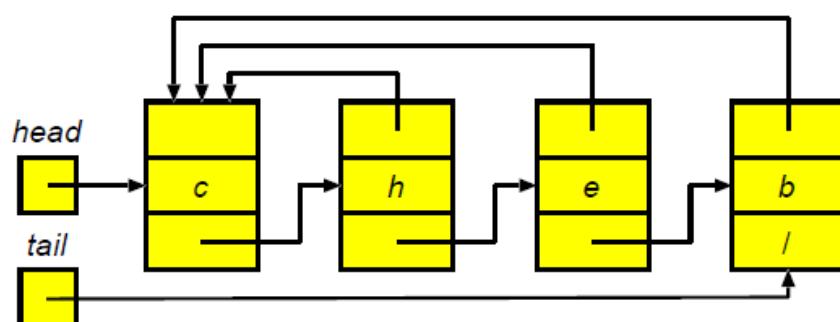
```
1 if Find-Set( $u$ )  $\neq$  Find-Set( $v$ )
2 then return FALSE
3 else return TRUE
```

3 Estruturas Baseadas em Listas

Utilização de Lista Ligada

Organização

- Elementos de cada conjunto em lista (simplesmente) ligada
- Primeiro elemento é o representante do conjunto
- Todos os elementos incluem apontador para o representante do conjunto



Tempos de Execução

- Union(x, y):
 - Colocar elementos de x no fim da lista de y
 - Actualizar ponteiros de elementos de x para representante
- Operações sobre n elementos x_1, x_2, \dots, x_n
 - n operações Make-Set(x_i)
 - $\Theta(n)$
 - $n - 1$ operações Union(x_{i-1}, x_i), para $i = 2, \dots, n$
 - Cada operação Union(x_{i-1}, x_i) actualiza $i - 1$ elementos
 - Custo das $n - 1$ operações: $\sum_{i=1}^{n-1} i = \Theta(n^2)$
 - Número total de operações é $m = 2n - 1$ logo, em média, cada operação requer tempo $\Theta(n)$

Heurística União Pesada

- A cada conjunto associar o número de elementos
- Para cada operação Union, juntar lista com menor número de elementos à lista com maior número de elementos
 - Necessário actualizar menor número de ponteiros para representante
- Custo total de m operações é melhorado

Heurística União Pesada

- A cada conjunto associar o número de elementos
- Para cada operação Union, juntar lista com menor número de elementos à lista com maior número de elementos
 - Necessário actualizar menor número de ponteiros para representante
- Custo total de m operações é melhorado
- Sequência de m operações (que incluem n operações Union) é:
 $O(m + n \lg n)$
- Prova:
 - Para cada objecto num conjunto com n elementos, calcular limite superior do número de vezes que ponteiro para representante é actualizado

Tempos de Execução (com Heurística)

- Sempre que o ponteiro para o representante x é actualizado, x encontra-se no conjunto com menor número de elementos
 - Da 1^a vez, conjunto resultante com pelo menos 2 elementos
 - Da 2^a vez, conjunto resultante com pelo menos 4 elementos
 - ...
 - Após representante de x ter sido actualizado $\lceil \lg k \rceil$ vezes, conjunto resultante tem pelo menos k elementos
- Maior conjunto tem n elementos
 - Cada ponteiro actualizado não mais do que $\lceil \lg n \rceil$ vezes
- Tempo total para actualizar n elementos é $O(n \lg n)$
- Make-Set e Find-Set têm tempos de execução $O(1)$ e há $O(m)$ destas operações
- Tempo total para m operações (com n Union) é $O(m + n \lg n)$

4 Estruturas Baseadas em Árvores

Organização

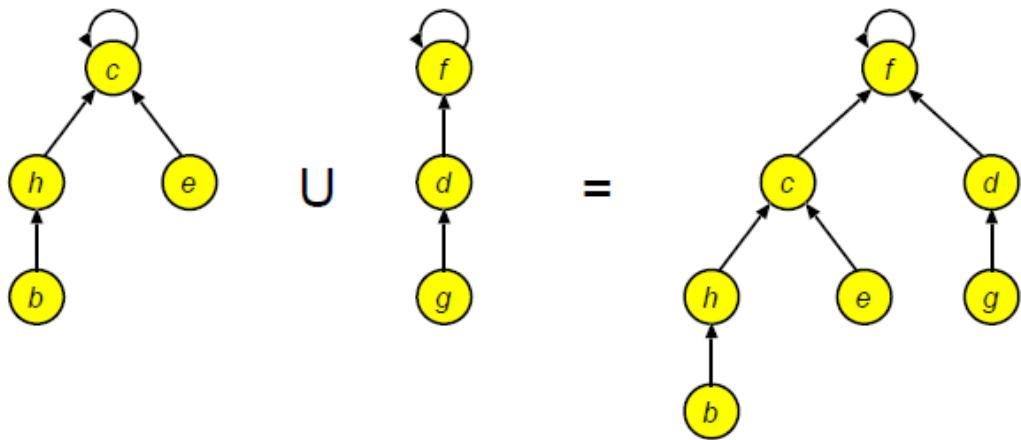
- Cada conjunto representado por uma árvore
- Cada elemento aponta apenas para antecessor na árvore
- Representante da árvore é a raiz
- Antecessor da raiz é a própria raiz

Operações

- Find-Set: Percorrer antecessores até raiz ser encontrada
- Union: Raiz de uma árvore aponta para raiz da outra árvore

Complexidade

- Sequência de $O(m)$ operações é $O(m n)$
- Pior caso ocorre quando as árvores que são apenas listas dos n elementos



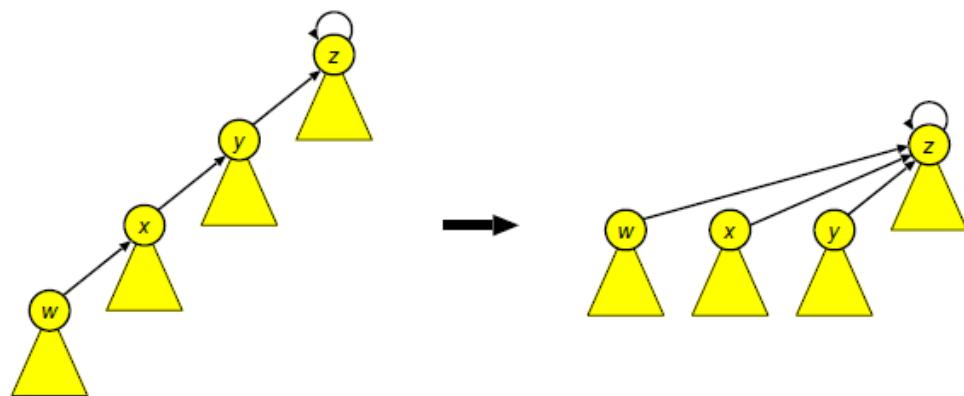
Heurística - União por Categoria

Numa união de dois conjuntos, colocar árvore com menos elementos a apontar para árvore com mais elementos

- Utilizar estimativa do número de elementos em cada sub-árvore
- categoria (*rank*): aproxima logaritmo do tamanho da sub-árvore e é um limite superior na altura da sub-árvore
- Numa união, raiz da árvore com menor *rank* aponta para raiz da árvore com maior *rank*

Heurística - Compressão de Caminhos

Em cada operação Find-Set coloca cada nó visitado a apontar directamente para a raiz da árvore (representante do conjunto)



Make-Set(x)

- 1 $p[x] \leftarrow x$
- 2 $rank[x] \leftarrow 0$

Find-Set(x)

- 1 **if** $x \neq p[x]$
- 2 **then** $p[x] \leftarrow \text{Find-Set}(p[x])$
- 3 **return** $p[x]$

Union(x, y)

- 1 **Link(Find-Set(x), Find-Set(y))**

Link(x, y)

- 1 **if** $rank[x] > rank[y]$
- 2 **then** $p[y] \leftarrow x$
- 3 **else** $p[x] \leftarrow y$
- 4 **if** $rank[x] = rank[y]$
- 5 **then** $rank[y] \leftarrow rank[y] + 1$

Complexidade

Execução de m operações sobre n elementos: $O(m \alpha(n))$

Onde $\alpha(n) = \min\{k : A_k(1) \geq n\}$

$$A_k(j) = \begin{cases} j+1 & \text{se } k=0 \\ A_{k-1}^{(j+1)}(j) & \text{se } k \geq 1 \end{cases}$$

$$\begin{cases} A_{k-1}^{(0)}(j) & = j \\ A_{k-1}^{(i)}(j) & = A_{k-1}(A_{k-1}^{(i-1)}(j)) \end{cases}$$

$$A_1(j) = 2 * j + 1$$

$$A_2(j) = 2^{j+1}(j+1) - 1$$

$$\alpha(n) = \min\{k : A_k(1) \geq n\} \quad A_k(j) = \begin{cases} j+1 & \text{se } k=0 \\ A_{k-1}^{(j+1)}(j) & \text{se } k \geq 1 \end{cases}$$

$$A_1(1) = A_0^{(2)}(1) = A_0(A_0(1)) = A_0(2) = 3$$

$$A_1(3) = A_0^{(4)}(1) = A_0(A_0(A_0(A_0(3)))) = 7$$

$$A_2(1) = A_1^{(2)}(1) = A_1(A_1(1)) = A_1(3) = 7$$

$$A_3(1) = A_2^{(2)}(1) = A_2(A_2(1)) = A_2(7) = 2^8 * 8 - 1 = 2047$$

$$A_4(1) = A_3^{(2)}(1) = A_3(A_3(1)) = A_3(2047) = A_2^{(2048)}(2047) >> A_2(2047) >> 10^{80}$$

$$\alpha(n) = \begin{cases} 0 & \text{se } 0 \leq n \leq 2 \\ 1 & \text{se } n = 3 \\ 2 & \text{se } 4 \leq n \leq 7 \\ 3 & \text{se } 8 \leq n \leq 2047 \\ 4 & \text{se } 2048 \leq n \leq A_4(1) \end{cases}$$

1 Motivação

Problema: Estabelecer uma rede com o menor custo

Suponha que pretende instalar uma nova rede de fornecimento de para um serviço (p.e. TV por cabo, gás natural, ou outro) numa urbanização.

Para estabelecer a rede é necessário fazer obras na via pública para instalar a infraestrutura (por exemplo, colocação de cabos de fibra óptica ou novas condutas de gás).

O objectivo é fornecer o serviço a todas as casas da urbanização através de uma rede. No entanto, cada possível ligação na urbanização tem um custo e pretende-se minimizar o custo total da instalação.

Problema: Estabelecer uma rede com o menor custo

Suponha que pretende instalar uma nova rede de fornecimento de para um serviço (p.e. TV por cabo, gás natural, ou outro) numa urbanização.

Para estabelecer a rede é necessário fazer obras na via pública para instalar a infraestrutura (por exemplo, colocação de cabos de fibra óptica ou novas condutas de gás).

O objectivo é fornecer o serviço a todas as casas da urbanização através de uma rede. No entanto, cada possível ligação na urbanização tem um custo e pretende-se minimizar o custo total da instalação.

Solução :

- Cada casa da urbanização é modelada como um vértice num grafo
- Cada possível ligação entre casas corresponde a um arco pesado cujo peso indica o custo da ligação
- A solução do problema corresponde à árvore abrangente de menor custo (*Minimum Spanning Tree (MST)*) do grafo

2 Definições

- Árvores Abrangentes de Menor Custo

Árvores Abrangentes

- Um grafo não dirigido $G = (V, E)$, diz-se **ligado** se para qualquer par de vértices existe um caminho que liga os dois vértices
- Dado grafo não dirigido $G = (V, E)$, ligado, uma **árvore abrangente** é sub-conjunto acíclico $T \subseteq E$, que liga todos os vértices
- O tamanho da árvore é $|T| = |V| - 1$

Árvores Abrangentes de Menor Custo

Dado grafo $G = (V, E)$, ligado, não dirigido, com uma função de pesos $w : E \rightarrow R$, identificar uma árvore abrangente T , tal que a soma dos pesos dos arcos de T é minimizada

$$\min w(T) = \sum_{(u,v) \in T} w(u,v)$$

3 Algoritmo (greedy) genérico

Abordagem Greedy

- Manter conjunto A que é um subconjunto de uma MST T
- A cada passo do algoritmo identificar arco (u, v) que pode ser adicionado a A sem violar a invariante
- $A \cup \{(u, v)\}$ é sub-conjunto de uma MST T
 - (u, v) é declarado um arco seguro para A

Algoritmo

MST-Genérico(G, w)

- 1 $A = \emptyset$
- 2 while A não forma árvore abrangente identificar arco seguro (u, v) para A
- 3 do $A = A \cup \{(u, v)\}$
- 4 return A

Critérios de Optimalidade

- Um **corte** $(S, V - S)$ de um grafo não dirigido $G = (V, E)$ é uma partição de V
- Um arco $(u, v) \in E$ **cruza** o corte $(S, V - S)$ se um dos extremos está em S e o outro está em $V - S$
- Um corte **respeita** um conjunto de arcos A se nenhum arco de A cruza o corte
- Um arco diz-se um **arco leve** que cruza um corte se o seu peso é o menor de todos os arcos que cruzam o corte

Critérios de Optimalidade

Seja $G = (V, E)$ um grafo não dirigido, ligado, com função de pesos w . Seja A um sub-conjunto de E incluído numa MST T , seja $(S, V - S)$ qualquer corte de G que **respeita** A , e seja (u, v) um **arco leve** que **cruza** $(S, V - S)$. Então (u, v) é um **arco seguro** para A .

Prova

- MST T , com $A \subseteq T$, e arco leve $(u, v) \notin T$
- Objectivo: Construir outra MST T' que inclui $A \cup \{(u, v)\}$
- (u, v) é um arco seguro para A

Critérios de Optimalidade

- O arco (u, v) forma ciclo com arcos do caminho p , definido em T , que liga u a v
- Dado u e v estarem nos lados opostos do corte $(S, V - S)$, então existe pelo menos um arco (x, y) do caminho p em T que cruza o corte

Arco (x, y)

- $(x, y) \notin A$, porque $(S, V - S)$ respeita A
- Remoção de (x, y) divide T em dois componentes
- Inclusão de (u, v) permite formar $T' = T - \{(x, y)\} \cup \{(u, v)\}$
- Dado que (u, v) é um arco leve que cruza o corte $(S, V - S)$, e porque (x, y) também cruza o corte: $w(u, v) \leq w(x, y)$

Critérios de Optimalidade

Conclusão

- $w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$ porque $w(u, v) \leq w(x, y)$
- Mas T é MST, pelo que $w(T) \leq w(T')$, por definição de MST
- Logo, $w(T') = w(T)$, e T' também é MST

(u, v) é seguro para A :

- Verifica-se $A \subseteq T'$, dado que por construção $A \subseteq T$, e $(x, y) \notin A$
- Assim, verifica-se também $A \cup (u, v) \subseteq T'$
- T' é MST, pelo que (u, v) é seguro para A

4 Algoritmo de Prim

(<https://www.youtube.com/watch?v=4ZIRH0eK-qQ>)

Algoritmo de Prim

- MST construída a partir de um vértice raíz r
- Algoritmo mantém sempre uma árvore A
- Árvore A é extendida a partir do vértice r
- A cada passo é escolhido um arco leve, seguro para A
- Utilização de fila de prioridade Q

Notação

- $\text{key}[v]$: menor peso de qualquer arco que ligue v a um vértice na árvore
- $\pi[v]$: antecessor de v na árvore

Pseudo-Código

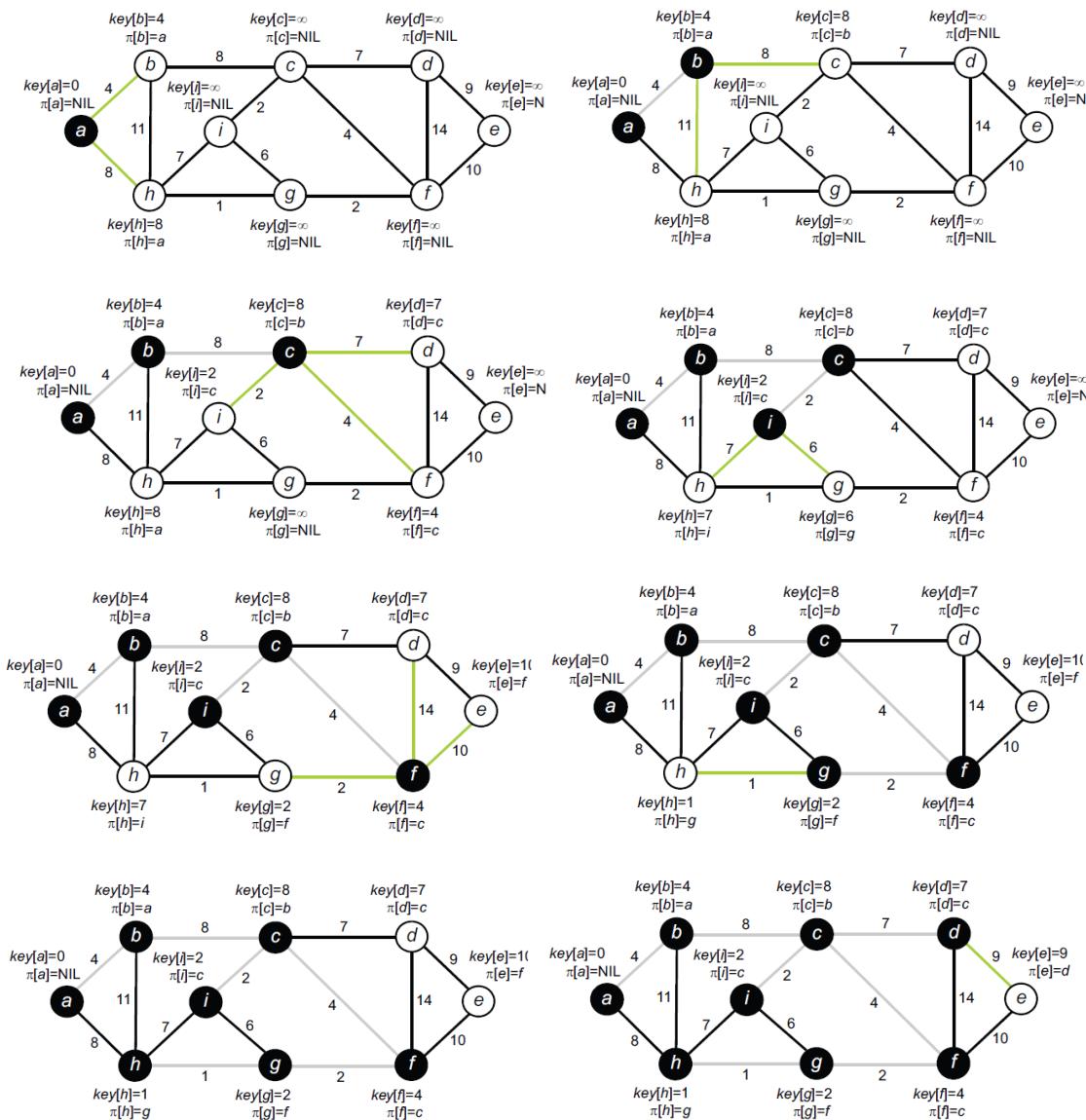
MST-Prim(G, w, r)

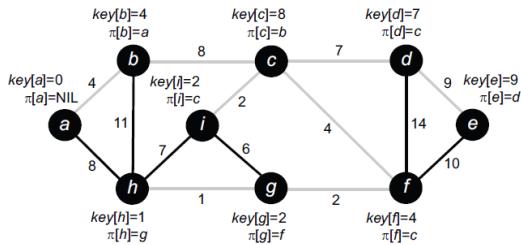
```

1   for each  $u \in V[G]$                                ▷ Inicialização
2       do  $\text{key}[u] = \infty$ 
3            $\pi[u] = \text{NIL}$ 
4    $\text{key}[r] = 0$ 
5    $Q = V[G]$                                         ▷ Fila de Prioridade
6   while ( $Q \neq \emptyset$ )
7       do  $u = \text{Extract-Min}(Q)$ 
8           for each  $v \in \text{Adj}[u]$ 
9               do if ( $v \in Q$  and  $w(u, v) < \text{key}[v]$ )
10                  then  $\pi[v] = u$ 
11                   $\text{key}[v] = w(u, v)$  ▷ Actualização de  $Q$ 

```

Exemplo





Complexidade

- Fila de prioridade baseada em amontoados (heap)
- Quando um vértice é extraído da fila Q , implica actualização de Q
 - Cada vértice é extraído apenas 1 vez $O(V)$
 - Actualização de Q : $O(\lg V)$
 - Então, $O(V \lg V)$
- Para cada arco (i.e. $O(E)$) existe no pior-caso uma actualização de Q em $O(\lg V)$
- Complexidade algoritmo Prim: $O(V \lg V + E \lg V)$
- Logo, é possível assegurar $O(E \lg V)$ porque grafo é ligado

EM ALTERNATIVA...

Em cada iteração do algoritmo temos uma partição do conjunto de vértices $V = \{1, 2, \dots, n\}$ em dois subconjuntos P e Q , em que P é o conjunto dos vértices já processados e Q o conjunto dos restantes vértices.

Apresentação

- 1 Considerar $P = \{1\}, Q = V \setminus P$, e c_{ij} igual ao peso da aresta entre os vértices i e j , se existir.
- 2 Para cada vértice $i \in Q$, calcular a sua *etiqueta* $t(i)$, definida por

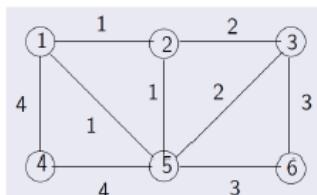
$$t(i) = \begin{cases} c_{ij} & \text{se } i \text{ e } j \text{ são adjacentes} \\ +\infty & \text{caso contrário} \end{cases}$$

- 3 Considerar um vértice $v \in Q$ de etiqueta mínima.
- 4 Localizar um vértice $u \in P$ tal que o peso da aresta entre u e v seja $t(v)$ e aceitar a aresta $\{u, v\}$ como aresta da árvore geradora de peso mínimo T .
- 5 $P = P \cup \{v\}, Q = Q \setminus \{v\}$.
- 6 Recalcular a função t : Se $w \in Q$, então $t(w) = \min\{t(w), d(v, w)\}$, com

$$d(v, w) = \begin{cases} c_{vw} & \text{se } v \text{ e } w \text{ são adjacentes} \\ +\infty & \text{caso contrário} \end{cases}$$

- 7 Se $Q = \emptyset$, o Algoritmo termina. Caso contrário, regressar ao Passo 3.

Exemplo



$P = \{1\}$, $Q = \{2, 3, 4, 5, 6\}$ e

$$\begin{cases} t(2) = 1 \\ t(3) = +\infty \\ t(4) = 4 \\ t(5) = 1 \\ t(6) = +\infty \end{cases}$$

O vértice 2, por exemplo, é escolhido para entrar em P (uma vez que $t(5) = t(2)$, também poderíamos ter escolhido o vértice 5 para entrar em P) e a aresta $\{1, 2\}$ é introduzida em T , $P = \{1, 2\}$ e $Q = \{3, 4, 5, 6\}$.

$P = \{1, 2\}$, $Q = \{3, 4, 5, 6\}$ e

$$\begin{cases} t(3) = \min\{t(3), d(2, 3)\} = 2 \\ t(4) = \min\{t(4), d(2, 4)\} = 4 \\ t(5) = \min\{t(5), d(2, 5)\} = 1 \\ t(6) = \min\{t(6), d(2, 6)\} = +\infty \end{cases}$$

O vértice 5 é escolhido para entrar em P e a aresta $\{1, 5\}$, por exemplo, é escolhida para entrar em T (uma vez que $d(2, 5) = d(1, 5)$, também poderíamos ter escolhido $\{2, 5\}$ para entrar em T).

$P = \{1, 2, 5\}$, $Q = \{3, 4, 6\}$ e

$$\begin{cases} t(3) = \min\{t(3), d(5, 3)\} = 2 \\ t(4) = \min\{t(4), d(5, 4)\} = 4 \\ t(6) = \min\{t(6), d(5, 6)\} = 3 \end{cases}$$

O vértice 3 é escolhido para entrar em P e a aresta $\{2, 3\}$, por exemplo, é escolhida para entrar em T (uma vez que $d(5, 3) = d(2, 3)$, também poderíamos ter escolhido $\{5, 3\}$ para entrar em T).

$P = \{1, 2, 3, 5\}$, $Q = \{4, 6\}$ e

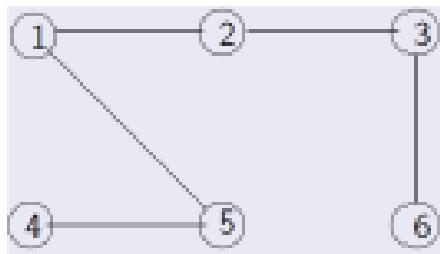
$$\begin{cases} t(4) = \min\{t(4), d(3, 4)\} = 4 \\ t(6) = \min\{t(6), d(3, 6)\} = 3 \end{cases}$$

O vértice 6 é escolhido para entrar em P e a aresta $\{3, 6\}$, por exemplo, é escolhida para entrar em T (uma vez que $d(5, 6) = d(3, 6)$, também poderíamos ter escolhido $\{5, 6\}$ para entrar em T).

$P = \{1, 2, 3, 5, 6\}$, $Q = \{4\}$ e $t(4) = \min\{t(4), d(6, 4)\} = 4$.

O vértice 4 é escolhido para entrar em P e a aresta $\{5, 4\}$, por exemplo, é escolhida para entrar em T (uma vez que $d(1, 4) = d(5, 4)$, também poderíamos ter escolhido $\{1, 4\}$ para entrar em T), $P = \{1, 2, 3, 4, 5, 6\}$ e $Q = \emptyset$.

Como $Q = \emptyset$, o algoritmo termina com uma árvore geradora de peso mínimo formada pelas arestas $\{1, 2\}$, $\{1, 5\}$, $\{2, 3\}$, $\{3, 6\}$ e $\{4, 5\}$.



5 Algoritmo de Boruvka

Algoritmo de Boruvka

- Começar com N conjuntos de vértices (N árvores)
 - Noção de arco marcado
- A cada passo do algoritmo,
 - Para cada árvore T
 - seleccionar arco de menor peso incidente em T
 - marcar arco seleccionado (passa a estar incluído em T)
 - arco pode ser duplamente seleccionado
 - Juntar na mesma árvore as árvores ligadas por arcos marcados
- Algoritmo termina quando existir apenas uma árvore

Complexidade

- Número de passos do algoritmo: $O(\lg V)$
- Número de arcos analisados em cada passo: $O(E)$
- Manutenção das árvores em cada passo: $O(E)$
- Logo, é possível assegurar $O(E \lg V)$

Correcção do Algoritmo

- Algoritmo correcto apenas se **pesos com valores distintos**
- Problema: Escolha de dois arcos de peso igual que ligam duas árvores!
- Com pesos iguais é necessário estabelecer relação de ordem (artificial) entre todos os arcos (e.g. ordem lexicográfica)

6 Algoritmo de Kruskal

(https://www.youtube.com/watch?v=4ZIRH0eK-qQ&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=44)

Passo I

Ordenar as arestas da rede por ordem não decrescente de pesos.

Passo II

Aceitar as $n - 1$ primeiras arestas da lista que não provoquem ciclos.

Algoritmo de Kruskal

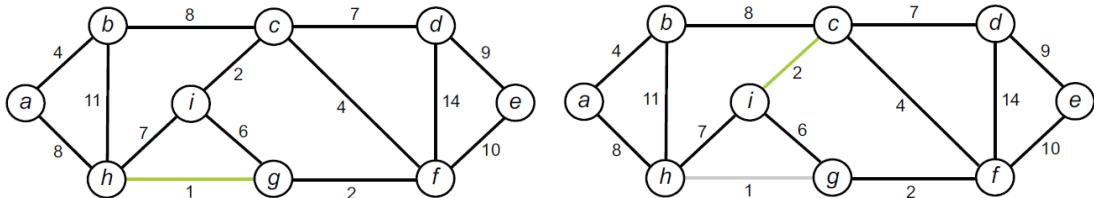
- Algoritmo mantém floresta (de árvores) A
- Utilização de uma estrutura de dados para representar conjuntos disjuntos
- Cada conjunto representa uma sub-árvore de uma MST
- Em cada passo é escolhido um arco leve, seguro para A

Pseudo-Código

MST-Kruskal(G, w)

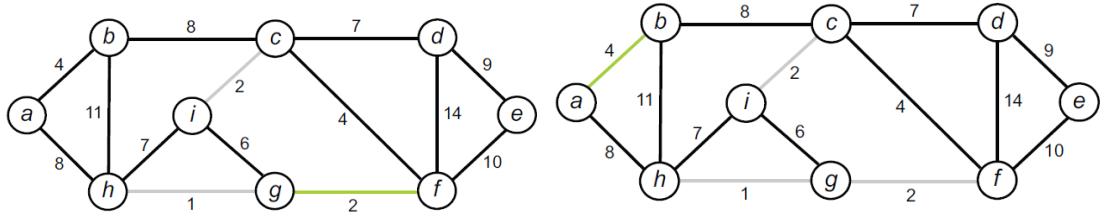
```
1  $A = \emptyset$ 
2 for each  $v \in V[G]$  ▷ Cria conjunto para cada  $v$ 
3   do Make-Set( $v$ )
4 Ordenar arcos de  $E[G]$  por ordem de peso não decrescente
5 for each  $(u, v) \in E[G]$ , por ordem não decrescente de  $w(u, v)$ 
6   do if ( $\text{Find-Set}(u) \neq \text{Find-Set}(v)$ ) ▷  $(u, v)$  é arco leve, seguro para  $A$ 
7     then  $A = A \cup \{(u, v)\}$ 
8     Union  $(u, v)$ 
9 return  $A$ 
```

Exemplo



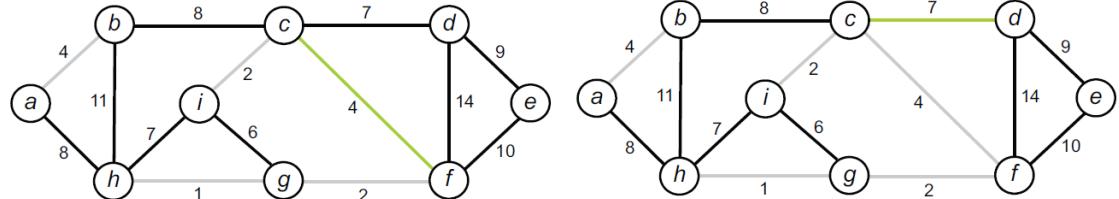
$$A = \{(h,g)\}$$

$$A = \{(h,g), (i,c)\}$$



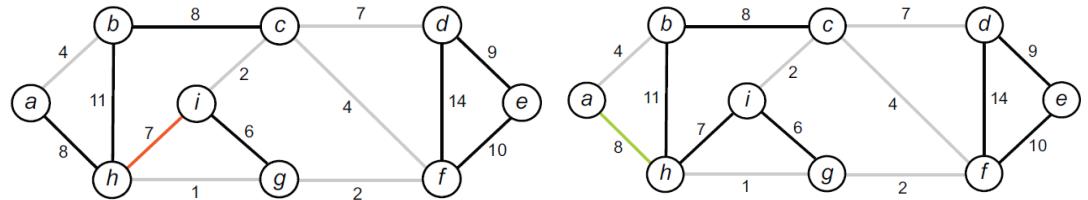
$$A = \{ (h,g), (i,c), (g,f) \}$$

$$A = \{ (h,g), (i,c), (g,f), (a,b) \}$$



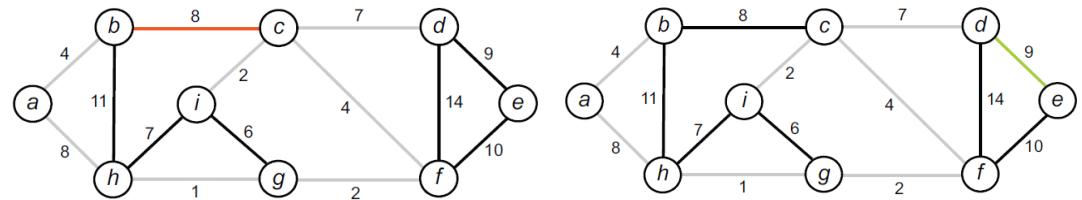
$$A = \{ (h,g), (i,c), (g,f), (a,b), (c,f) \}$$

$$A = \{ (h,g), (i,c), (g,f), (a,b), (c,f), (c,d) \}$$



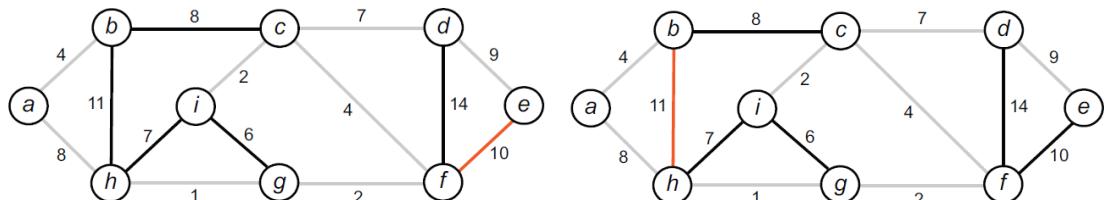
$$A = \{ (h,g), (i,c), (g,f), (a,b), (c,f), (c,d) \}$$

$$A = \{ (h,g), (i,c), (g,f), (a,b), (c,f), (c,d), (a,h) \}$$



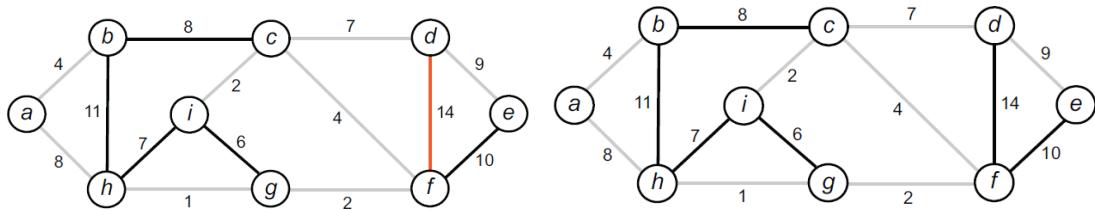
$$A = \{ (h,g), (i,c), (g,f), (a,b), (c,f), (c,d), (a,h) \}$$

$$A = \{ (h,g), (i,c), (g,f), (a,b), (c,f), (c,d), (a,h), (d,e) \}$$



$$A = \{ (h,g), (i,c), (g,f), (a,b), (c,f), (c,d), (a,h), (d,e) \}$$

$$A = \{ (h,g), (i,c), (g,f), (a,b), (c,f), (c,d), (a,h), (d,e) \}$$



$$A = \{ (h,g), (i,c), (g,f), (a,b), (c,f), (a,h), (d,e) \}$$

$$A = \{ (h,g), (i,c), (g,f), (a,b), (c,f), (c,d), (a,h), (d,e) \}$$

Complexidade

- Depende da implementação das operações sobre conjuntos disjuntos
- Inicialização: $O(E \lg E)$ devido à ordenação dos arcos
- Operações sobre os conjuntos disjuntos
 - $O(V)$ operações de Make-Set
 - $O(E)$ operações de Find-Set e Union
 - Com estruturas de dados adequadas (árvores com compressão de caminhos e união por categorias) para conjuntos disjuntos é possível estabelecer que $O((V + E) \alpha(E, V))$
 - Como $|E| \geq V - 1$ porque o grafo é ligado, então temos $O(E \alpha(E, V))$
- Logo, é possível assegurar $O(E \lg E)$
- Dado que $E < V^2$, obtém-se também $O(E \lg V)$

6.4. TÉCNICAS DE SÍNTSE DE ALGORITMOS

1 Motivação

- Selecção de Actividades

Técnicas para Síntese de Algoritmos

- Dividir para conquistar
- Programação dinâmica
- Algoritmos greedy
 - Estratégia: a cada passo da execução do algoritmo escolher opção que **localmente** se afigura como a melhor para encontrar solução óptima
 - Estratégia permite obter solução óptima?
 - Exemplo: Prim, Dijkstra

Selecção de Actividades

- Seja $S = \{1, 2, \dots, n\}$ um conjunto de actividades que pretendem utilizar um dado recurso
- Apenas uma actividade pode utilizar o recurso de cada vez
- Actividade i :
 - tempo de início: s_i
 - tempo de fim: f_i
 - execução da actividade durante $[s_i, f_i]$
- Actividades i e j compatíveis apenas se $[s_i, f_i] \cap [s_j, f_j] = \emptyset$
- Objectivo: encontrar conjunto máximo de actividades mutuamente compatíveis

Selecção de Actividades

- Admitir que $f_1 \leq f_2 \leq \dots \leq f_n$
- Escolha greedy:
 - Escolher actividade com o menor tempo de fim
- Porquê?
 - Maximizar espaço para restantes actividades serem realizadas

Seleccionar_Actividades_Greedy(s, f)

```
1   $n \leftarrow \text{length}[s]$ 
2   $A \leftarrow \{1\}$ 
3   $j \leftarrow 1$ 
4  for  $i \leftarrow 2$  to  $n$ 
5    do if  $s_i \geq f_j$ 
6      then  $A \leftarrow A \cup \{i\}$ 
7       $j \leftarrow i$ 
8  return  $A$ 
```

Selecção de Actividades

- Algoritmo encontra soluções de tamanho máximo para o problema de selecção de actividades

Selecção de Actividades

- Algoritmo encontra soluções de tamanho máximo para o problema de selecção de actividades
- Existe uma solução óptima que começa com escolha greedy, i.e. actividade 1
 - Seja A uma solução óptima que começa em k
 - Seja B uma solução óptima que começa em 1: $B = A - \{k\} \cup \{1\}$
 - $f_1 \leq f_k$
 - Actividades em B são mutuamente disjuntas e $|A| = |B|$
 - Logo, B é também solução óptima !

Selecção de Actividades

- Algoritmo encontra soluções de tamanho máximo para o problema de selecção de actividades
- Após escolha greedy, problema reduz-se a encontrar solução para actividades compatíveis com actividade 1
 - Seja A solução óptima, e que começa em 1
 - $A' = A - \{1\}$ é solução óptima para $S' = \{i \in S : s_i \geq f_1\}$
 - Caso contrário, existiria uma solução $|B'| > |A'|$ para S' que permitiria obter solução B para S com mais actividades do que A ; uma contradição !
- Aplicar indução no número de escolhas greedy
- Algoritmo calcula solução óptima !

2 Características Algoritmos Greedy

Características Algoritmos Greedy

- Propriedade da escolha greedy
 - Óptimo (global) para o problema pode ser encontrado realizando escolhas locais ótimas
(em programação dinâmica, esta escolha está dependente de resultados de sub-problemas)
- Sub-estrutura óptima
 - Solução óptima do problema engloba soluções óptimas para sub-problemas

3 Exemplos

- Problema da Mochila Fraccionário
- Minimizar Tempo no Sistema
- Códigos de Huffman

Definição do Problema

- Dados n objectos ($1, \dots, n$) e uma mochila
- Cada objecto tem um valor v_i e um peso w_i
- Peso transportado pela mochila não pode exceder W
- É possível transportar fração x_i do objecto: $0 \leq x_i \leq 1$
- Objectivo: maximizar o valor transportado pela mochila e respeitar a restrição de peso

Observações

- Soma do peso dos n objectos deve exceder peso limite W . Caso contrário a solução é trivial.
- Solução óptima tem que encher mochila completamente, $\sum x_i w_i = W$
- Caso contrário poderíamos transportar mais frações, com mais valor !
- Complexidade Solução: $O(n)$ ou $O(n \lg n)$

Encher_Mochila_Greedy(v, w, W)

```
1 weight ← 0
2 while (weight < W)
3     do escolher objecto  $i$  com  $v_i/w_i$  máximo
4         if ( $w_i + weight \leq W$ )
5             then  $x_i \leftarrow 1$ ;  $weight \leftarrow weight + w_i$ 
6             else  $x_i \leftarrow (W - weight)/w_i$ ;  $weight \leftarrow W$ 
```

Optimalidade da Solução Greedy

Se objectos forem escolhidos por ordem decrescente de v_i/w_i , então algoritmo encontra solução óptima

- Admitir: $v_1/w_1 \geq \dots \geq v_n/w_n$
- Solução calculada por algoritmo greedy: $X = (x_1, \dots, x_n)$
 - Se $x_i = 1$ para todo o i , solução é necessariamente óptima
 - Caso contrário, seja j o menor índice para o qual $x_j < 1$
 - $x_i = 1, i < j$
 - $x_i = 0, i > j$
 - Relação de pesos: $\sum_{i=1}^n x_i w_i = W$
 - Valor da solução: $\sum_{i=1}^n x_i v_i = V(X)$

Optimalidade da Solução Greedy

- Qualquer solução possível: $Y = (y_1, \dots, y_n)$
 - Peso: $\sum_{i=1}^n y_i w_i \leq W$
 - Valor: $V(Y) = \sum_{i=1}^n y_i v_i$
- Relação X vs. Y :
 - Peso: $\sum_{i=1}^n (x_i - y_i) w_i \geq 0$
 - Valor: $V(X) - V(Y) = \sum_{i=1}^n (x_i - y_i) v_i = \sum_{i=1}^n (x_i - y_i) w_i (v_i / w_i)$
- Seja j o menor índice tal que $x_j < 1$. Casos possíveis:
 - $i < j \Rightarrow x_i = 1 \wedge x_i - y_i \geq 0 \wedge v_i / w_i \geq v_j / w_j$
 - $i > j \Rightarrow x_i = 0 \wedge x_i - y_i \leq 0 \wedge v_i / w_i \leq v_j / w_j$
 - $i = j \Rightarrow v_i / w_i = v_j / w_j$
- Verifica-se sempre que: $(x_i - y_i)(v_i / w_i) \geq (x_j - y_j)(v_j / w_j)$

Optimalidade da Solução Greedy

- Considerando que $(x_i - y_i)(v_i / w_i) \geq (x_j - y_j)(v_j / w_j)$
- Verifica-se que:
$$V(X) - V(Y) = \sum_{i=1}^n (x_i - y_i) w_i (v_i / w_i) \geq (v_j / w_j) \sum_{i=1}^n (x_i - y_i) w_i \geq 0$$
- Logo, $V(X)$ é a melhor solução possível entre todas as soluções possíveis
- Algoritmo calcula solução óptima !

- Minimizar Tempo no Sistema

Definição do Problema

- Dado um servidor com n clientes, com tempo de serviço conhecido (i.e. cliente i leva tempo t_i), objectivo é minimizar tempo total despendido no sistema (pelo total dos n clientes)
- Minimizar: $\sum_{i=1}^n$ (tempo total despendido no sistema pelo cliente i)

Definição do Problema

- Dado um servidor com n clientes, com tempo de serviço conhecido (i.e. cliente i leva tempo t_i), objectivo é minimizar tempo total despendido no sistema (pelo total dos n clientes)
- Minimizar: $\sum_{i=1}^n$ (tempo total dispendido no sistema pelo cliente i)

Solução Greedy

Servir clientes por ordem crescente do tempo de serviço.

Optimalidade da Solução Greedy

- $P = p_1 p_2 \dots p_n$, permutação dos inteiros de 1 a n
- Com clientes servidos pela ordem P , tempo de serviço do cliente a ser servido na posição i é $s_i = t_{p_i}$ (por exemplo, $s_1 = t_{p_1} = t_5$)
- Tempo total passado no sistema por todos os clientes é:
$$T(P) = \sum_{k=1}^n (n - k + 1)s_k$$
 - s_1 aparece n vezes
 - s_2 aparece $n - 1$ vezes
 - ...
 - s_n aparece 1 vez
- Assumir clientes não ordenados por ordem crescente de tempo de serviço em P
 - Então existem a e b , com $a < b$, e $s_a > s_b$

Optimalidade da Solução Greedy

- Trocar ordem dos clientes a e b , de modo a obter ordem P'
- Corresponde a P com inteiros p_a e p_b trocados

$$T(P') = (n - a + 1)s_b + (n - b + 1)s_a + \sum_{k=1, k \neq a, b}^n (n - k + 1)s_k$$

- Obtendo-se,

$$\begin{aligned} T(P) - T(P') &= (n - a + 1)(s_a - s_b) + (n - b + 1)(s_b - s_a) \\ &= (b - a)(s_a - s_b) > 0 \end{aligned}$$

- P' é uma ordem melhor (com menor tempo de serviço)
- Algoritmo calcula solução óptima !

- Códigos de Huffman

(https://www.youtube.com/watch?v=co4_ahEDCho)

(https://www.youtube.com/watch?v=co4_ahEDCho&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=43)

Aplicação de árvores binárias a problemas de codificação

Tipos de códigos binários

- Comprimento fixo (exemplo: ASCII);
- Comprimento variável.

Propriedade do prefixo

- A palavra w é *prefixo* da palavra v se existir uma palavra p tal que $v = wp$.
- Um código diz-se um *código de prefixo*, se nenhum símbolo de um carácter for prefixo do símbolo de outro carácter.

Algoritmo de Huffman para obter o “melhor” código

Exemplo

Construção de um código de Huffman para um alfabeto de seis letras e com distribuição de frequências dada por $S^1 = \{3, 5, 6, 8, 10, 14\}$.

Somando as duas primeiras frequências e reordenando...

- ① $S^1 = \{3, 5, 6, 8, 10, 14\}$;
- ② $S^2 = \{6, 8, 8, 10, 14\}$;
- ③ $S^3 = \{8, 10, 14, 14\}$;
- ④ $S^4 = \{14, 14, 18\}$;
- ⑤ $S^5 = \{18, 28\}$.

Código óptimo

Código $\{000, 001, 110, 111, 01, 10\}$ com comprimentos $3, 3, 3, 3, 2, 2$. Este código tem comprimento total $3 \times 3 + 3 \times 5 + 3 \times 6 + 3 \times 8 + 2 \times 10 + 2 \times 14 = 114$.

Algoritmo de Huffman: Descompressão e aplicação

Descompressão:

Generalizando, o processo de descompressão é simplesmente uma questão de se traduzir a palavra-código de volta para os caracteres. Isso é geralmente realizado percorrendo a árvore do código de nó em nó, conforme a orientação dada por cada bit lido da palavra-código, até encontrar-se uma folha. Ou seja, o processo resume-se a voltar, por meio do caminho na árvore, para a folha associada ao símbolo original.

Aplicação

A codificação de Huffman é amplamente utilizada em aplicações de compressão, que vão de GZIP, PKZIP, BZIP2 a formatos de imagens como JPEG e PNG. Isso ocorre pela codificação ser simples, rápida e, principalmente, não ter patente. Consequentemente, não se pagam royalties para utilização comercial de aplicações que empregam este método.

Aplicação: Compressão de Dados

- Exemplo: Ficheiro com 100000 caracteres

	a	b	c	d	e	f
Frequência ($\times 1000$)	45	13	12	16	9	5
Código Fixo	000	001	010	011	100	101

- Tamanho do ficheiro comprimido: $3 \times 100000 = 300000$ bits
- Código de largura variável pode ser melhor do que de largura fixa
 - Aos caracteres mais frequentes associar códigos de menor dimensão

Aplicação: Compressão de Dados

- Código de comprimento variável:

	a	b	c	d	e	f
Frequência ($\times 1000$)	45	13	12	16	9	5
Código Variável	0	101	100	111	1101	1100

- Número de bits necessário:
 - $(45*1 + 13*3 + 12*3 + 16*3 + 9*4 + 5*4)*1000 = 224000$ bits
- Códigos livres de prefixo:
 - Nenhum código é prefixo de outro código
 - $001011101 \rightarrow 0.0.101.1101$
 - Código óptimo representado por árvore binária (completa)

Códigos de Huffman

- Dada uma árvore T associada a um código livre de prefixo
 - $f(c)$: frequência (ocorrências) do carácter c no ficheiro
 - $d_T(c)$: profundidade da folha c na árvore
 - $B(T)$: número de bits necessários para representar ficheiro

$$B(T) = \sum_{c \in C} f(c)d_T(c)$$

- Código de Huffman: construir árvore T que corresponde ao código livre de prefixo óptimo
 - Começar com $|C|$ folhas (para cada um dos caracteres do ficheiro) e realizar $|C| - 1$ operações de junção para obter árvore final

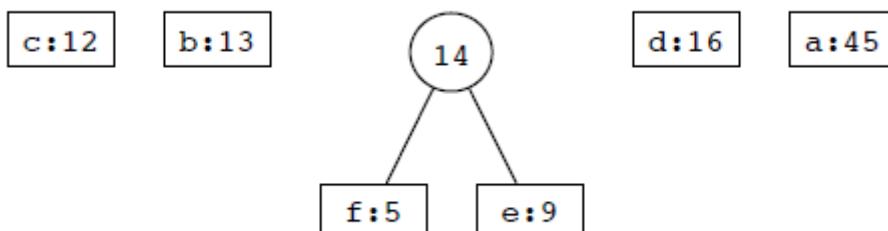
Huffman(C)

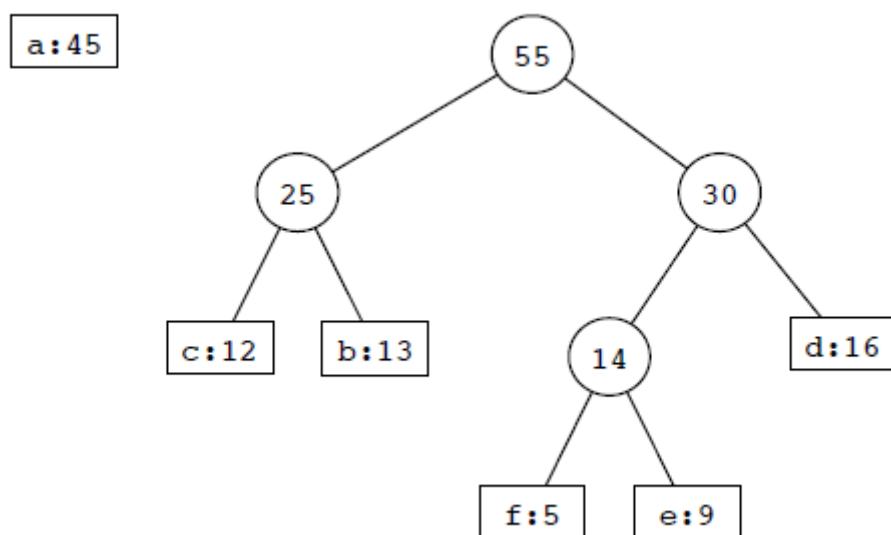
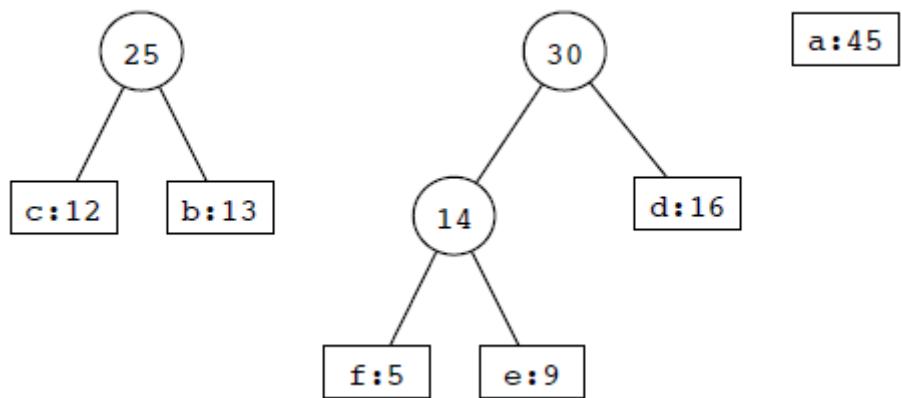
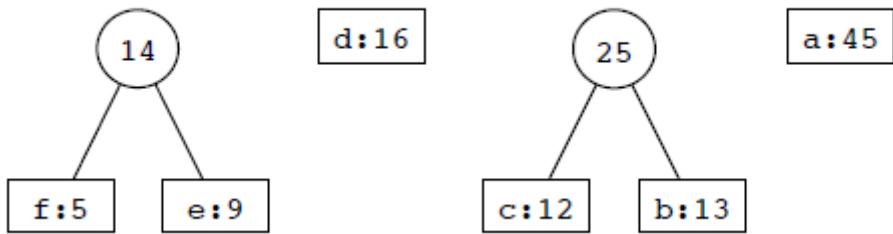
```
1   $n \leftarrow |C|$ 
2   $Q \leftarrow C$             $\triangleright$  Constrói fila de prioridade
3  for  $i \leftarrow 1$  to  $n - 1$ 
4      do  $z \leftarrow \text{AllocateNode}()$ 
5           $x \leftarrow \text{left}[z] \leftarrow \text{ExtractMin}(Q)$ 
6           $y \leftarrow \text{right}[z] \leftarrow \text{ExtractMin}(Q)$ 
7           $f[z] \leftarrow f[x] + f[y]$ 
8           $\text{Insert}(Q, z)$ 
9  return ExtractMin( $Q$ )
```

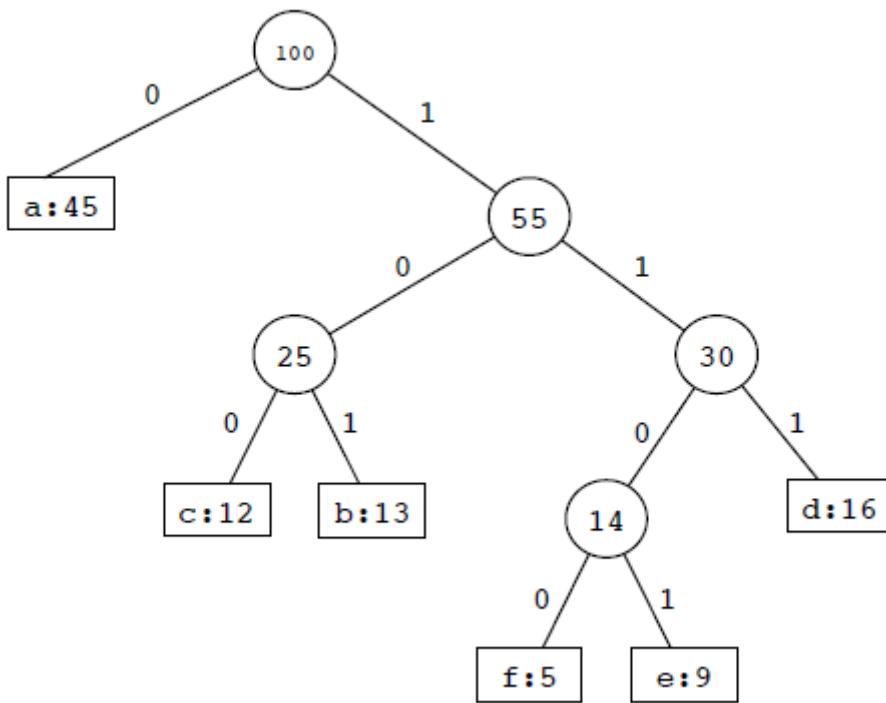
Tempo de Execução: $O(n \lg n)$

Exemplo

$f:5$ $e:9$ $c:12$ $b:13$ $d:16$ $a:45$







Códigos de Huffman

- Código de Huffman para alfabeto C :
 - Cada caractere $c \in C$ com frequência $f[c]$
 - x, y caracteres em C com as menores frequências
 - Código óptimo para C representado por árvore binária T

Optimalidade da Solução Greedy

Facto 1: (propriedade da escolha greedy)

- Existe código livre de prefixo para C tal que os códigos para x e y (com as menores frequências) têm o mesmo comprimento e diferem apenas no último bit
 - T árvore que representa código óptimo
 - b e c caracteres são nós folha de maior profundidade em T
 - Admitir, $f[b] \leq f[c]$, e $f[x] \leq f[y]$
 - Notar também que, $f[x] \leq f[b]$, e $f[y] \leq f[c]$
 - T' : trocar posições de b e x em T
 - T'' : trocar posições de c e y em T'
 - Neste caso, $B(T) \geq B(T') \wedge B(T') \geq B(T'')$
 - Mas, T é óptimo, pelo que $B(T'), B(T'') \geq B(T)$
 - Logo, T'' também é uma árvore óptima !

Optimalidade da Solução Greedy

Facto 2: (sub-estrutura óptima)

- Sendo z um nó interno de T , e x e y nós folha
- Considerar um caracter z com $f[z] = f[x] + f[y]$
- Então $T' = T - \{x, y\}$ é óptima para $C' = C - \{x, y\} \cup \{z\}$
 - $B(T) = B(T') + f[x] + f[y]$
 - Se T' é não óptimo, então existe T'' tal que $B[T''] < B[T']$
 - Mas z é nó folha também em T'' (devido a facto 1)
 - Adicionando x e y como filhos de z em T''
 - Código livre de prefixo para C com custo:
 - $B[T''] + f[x] + f[y] < B[T]$
 - mas T é óptimo ($B[T''] + f[x] + f[y] \geq B[T]$); pelo que T' também é óptimo

Optimalidade da Solução Greedy

- O algoritmo Huffman produz um código livre de prefixo óptimo
- Ver factos anteriores
 - Propriedade da escolha greedy
 - Sub-estrutura óptima