1. Programação Dinâmica

Carla Ferreira
(carla.ferreira@fct.unl.pt)

Licenciatura em Engenharia Informática
DI-FCT-UNL

Planeamento

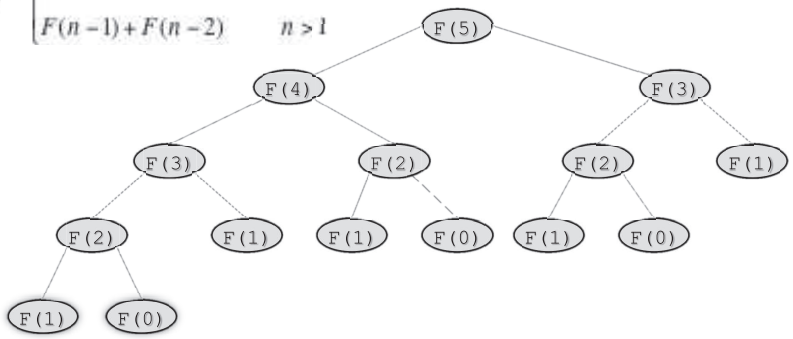
Apresentação da disciplina
Programação Dinâmica
Introdução aos Grafos
Percurso em largura e profundidade
Ordenação Topológica
Árvore Mínima de Cobertura
TAD Partição
Complexidade Amortizada
Algoritmo de Prim
TAD Fila com Prioridade Adaptável
Algoritmo de Dijskstra
Filas Binomiais
Filas de Fibonacci
Algoritmo de Bellman-Ford
Fluxo Máximo: Algoritmo de Ford-Fulkerson
Algoritmo de Edmonds-Karp
Introdução à Teoria da Complexidade
Exemplos de Problemas NP
Redutibilidade entre Problemas de Decisão

Números de Fibonacci

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

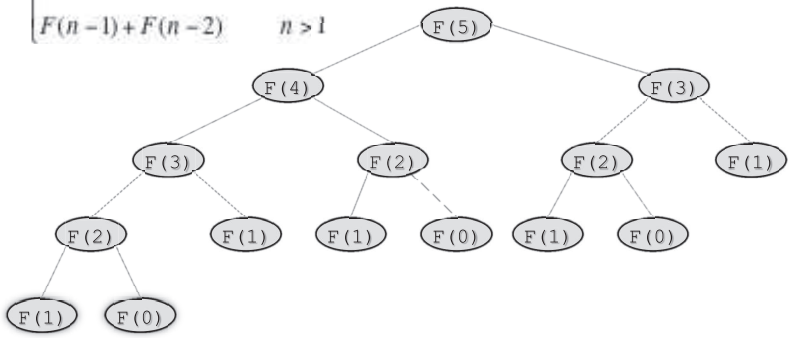
Números de Fibonacci

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



Números de Fibonacci

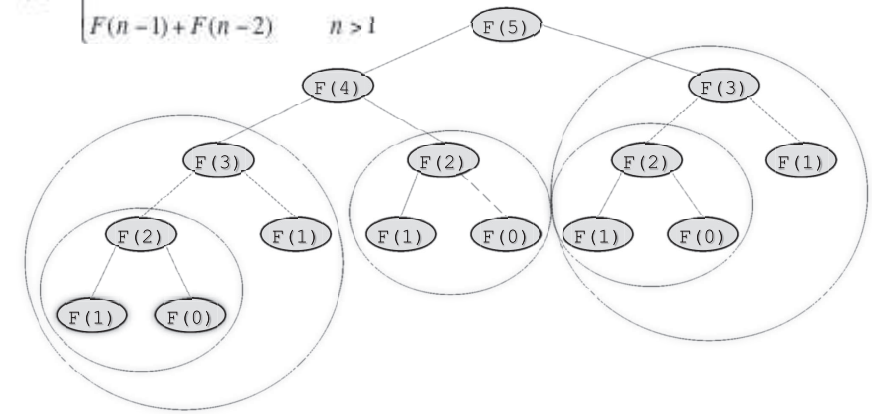
$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F(n-1) + F(n-2) & n > 1 \end{cases}$$



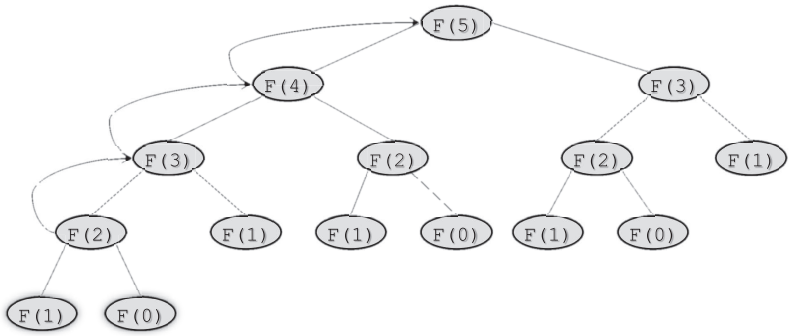
- Complexidade da implementação recursiva: $O(2^n)$

Números de Fibonacci

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



Números de Fibonacci



Números de Fibonacci

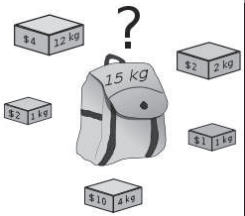
```
int Fibonaci(int n) {
    int result = 0;
    if (n > 0) {
        int[] table = new int[n+1];
        table[0] = 0;
        table[1] = 1;
        for (int i = 2; i ≤ n; i++)
                table[i] = table[i-1] + table[i-2]
        result = table[n];
    }
    return result;
}
```

- Complexidade da solução iterativa: $O(n)$

Números de Fibonacci

- Embora seja um problema artificial, ilustra algumas das características da programação dinâmica:
  - Subproblemas coincidentes
  - Solução construtiva que permite eliminar cálculos repetidos

Problema da Mochila (Knapsack)



- Dados N objectos {1, ..., N} e uma mochila
- Cada objecto tem um valor v_i e um peso w_i
- Peso total na mochila não pode exceder P
- Objectivo:
  - Maximizar o valor transportado pela mochila sem exceder o limite de peso
- Formalização:
  - Determinar o subconjunto A de {1, ...,N } que satisfaz a condição:

$$\max \sum_{i \in A} v_i \quad tal \quad que \quad \sum_{i \in A} w_i \leq P$$

Problema da Mochila

- Comparar diferentes abordagens:
  - Solução Exaustiva (*Brute Force*)
  - *Algoritmo Greedy*
  - *Programação Dinâmica*

Problema da Mochila

- Solução Exaustiva
  - Avaliar todas as possibilidades e seleccionar o melhor
  - Exemplo
    - $v_1 = 70$, $w_1 = 5$
    - $v_2 = 90$, $w_2 = 5$
    - $v_3 = 130$, $w_3 = 7$
    - $P = 10$

Problema da Mochila

- Solução Exaustiva
  - Subconjuntos:
    1. $\{\ \}$ Lucro = 0, Peso = 0
    2. $\{ v_1 \}$ Lucro = 70, Peso = 5
    3. $\{ v_2 \}$ Lucro = 90, Peso = 5
    4. $\{ v_3 \}$ Lucro = 130, Peso = 7
    5. $\{ v_1, v_2 \}$ Lucro = 160, Peso = 10
    6. $\{ v_1, v_3 \}$ Lucro = 200, Peso = 12
    7. $\{ v_2, v_3 \}$ Lucro = 220, Peso = 12
    8. $\{ v_1, v_2, v_3 \}$ Lucro = 290, Peso = 17
- *Complexidade é exponencial!*
  - *Um conjunto de dimensão n tem 2^n subconjuntos*

Problema da Mochila

- *Algoritmo Greedy*
  - Critério de selecção: a cada passo selecciona o objecto com maior valor
  - Exemplo
    - $v_1 = 70$, $w_1 = 5$
    - $v_2 = 90$, $w_2 = 5$
    - $v_3 = 130$, $w_3 = 7$
    - $P = 10$
- Nem sempre encontra a solução óptima!

Problema da Mochila

- *Algoritmo Greedy*
  - Critério de selecção: a cada passo selecciona o objecto com maior valor
  - Exemplo
    - $v_1 = 70$, $w_1 = 5$
    - $v_2 = 90$, $w_2 = 5$
    - $v_3 = 130$, $w_3 = 7$
    - $P = 10$
- Nem sempre encontra a solução óptima!
  - Neste problema o primeiro objecto seleccionado impede que se encontre a solução óptima

Metodologia

- Etapas para a construção de um algoritmo baseado em programação dinâmica:
  1. Caracterizar a estrutura da solução óptima
  2. Definir recursivamente o valor de uma solução óptima, em função de soluções óptimas de subproblemas
  3. Calcular o valor da solução óptima segundo uma estratégia *bottom-up*
  4. Construir solução a partir da informação obtida

Metodologia

- Etapas para a construção de um algoritmo baseado em programação dinâmica:
  1. **Caracterizar a estrutura da solução óptima**
  2. Definir recursivamente o valor de uma solução óptima, em função de soluções óptimas de subproblemas
  3. Calcular o valor da solução óptima segundo uma estratégia *bottom-up*
  4. Construir solução a partir da informação obtida

Estrutura da Solução Óptima

- Assumir o subproblema no qual conjunto de objectos está restrito a {1,...,i} com i ≤ N e o peso máximo da mochila é k ≤ P
- L(i,k) denota o valor máximo que é possível transportar para este subproblema

- A solução óptima para o problema inicial está em L(N,P)

Metodologia

- Etapas para a construção de um algoritmo baseado em programação dinâmica:
  1. Caracterizar a estrutura da solução óptima
  2. **Definir recursivamente o valor de uma solução óptima, em função de soluções óptimas de subproblemas**
  3. Calcular o valor da solução óptima segundo uma estratégia *bottom-up*
  4. Construir solução a partir da informação obtida

Definir Solução Óptima

- Construção da função recursiva L(i,k)
  - Não existem objectos a transportar na mochila ou o peso máximo é zero
    - Nesse caso L(i,k) = 0

Definir Solução Óptima

- Construção da função recursiva L(i,k)
  - Não existem objectos a transportar na mochila ou o peso máximo é zero
    - Nesse caso L(i,k) = 0
  - O peso do objecto i excede k
    - O objecto i não não pode ser colocado na mochila. Nesse caso L(i,k) = L(i-1,k)

Definir Solução Óptima

- Construção da função recursiva L(i,k)
  - Não existem objectos a transportar na mochila ou o peso máximo é zero
    - Nesse caso L(i,k) = 0
  - O peso do objecto i excede k
    - O objecto i não não pode ser colocado na mochila. Nesse caso L(i,k) = L(i-1,k)
  - Caso contrário, é preciso comparar o lucro entre colocar ou não o objecto i na mochila
    - max(L(i-1,k) , $L(i-1,k-w_i)+v_i$)

Definir Solução Óptima

- Construção da função recursiva L(i,k)

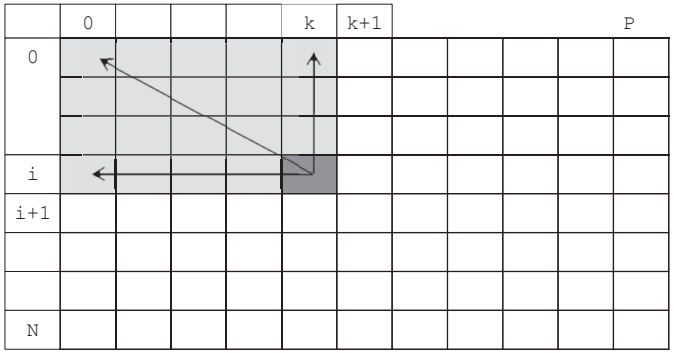
$$L(i,k) = \begin{cases} 0 & se \quad i = 0 \vee k = 0 \\ L(i-1,k) & se \quad i > 0 \wedge k > 0 \wedge w_i > k \\ \max(L(i-1,k), L(i-1,k-w_i)+v_i) & caso \quad contrário \end{cases}$$

Metodologia

- Etapas para a construção de um algoritmo baseado em programação dinâmica:
  1. Caracterizar a estrutura da solução óptima
  2. Definir recursivamente o valor de uma solução óptima, em função de soluções óptimas de subproblemas
  3. **Calcular o valor da solução óptima segundo uma estratégia *bottom-up***
  4. Construir solução a partir da informação obtida

Cálculo da Solução Óptima

$$L(i,k) = \begin{cases} 0 & se \quad i = 0 \vee k = 0 \\ L(i-1,k) & se \quad i > 0 \wedge k > 0 \wedge w_i > k \\ \max(L(i-1,k), L(i-1,k-w_i) + v_i) & caso \quad contrário \end{cases}$$



Cálculo da Solução Óptima

$$L(i,k) = \begin{cases} 0 & se \quad i = 0 \vee k = 0 \\ L(i-1,k) & se \quad i > 0 \wedge k > 0 \wedge w_i > k \\ \max(L(i-1,k), L(i-1,k-w_i) + v_i) & caso \quad contrário \end{cases}$$

- Exemplo
  - $v_1 = 70$, $w_1 = 5$; $v_2 = 90$, $w_2 = 5$; $v_3 = 130$, $w_3 = 7$; $P = 10$

i\k	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	→	→	→	→	→	→	→	→	→	→
2	0	→	→	→	→	→	→	→	→	→	→
3	0	→	→	→	→	→	→	→	→	→	→

↓

Cálculo da Solução Óptima

$$L(i,k) = \begin{cases} 0 & se \quad i = 0 \vee k = 0 \\ L(i-1,k) & se \quad i > 0 \wedge k > 0 \wedge w_i > k \\ \max(L(i-1,k), L(i-1,k-w_i) + v_i) & caso \quad contrário \end{cases}$$

- Exemplo
  - $v_1 = 70$, $w_1 = 5$; $v_2 = 90$, $w_2 = 5$; $v_3 = 130$, $w_3 = 7$; $P = 10$

i\k	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	70	70	70	70	70	70
2	0										
3	0										

Cálculo da Solução Óptima

$$L(i,k) = \begin{cases} 0 & se \quad i = 0 \vee k = 0 \\ L(i-1,k) & se \quad i > 0 \wedge k > 0 \wedge w_i > k \\ \max(L(i-1,k), L(i-1,k-w_i) + v_i) & caso \quad contrário \end{cases}$$

- Exemplo
  - $v_1 = 70$, $w_1 = 5$; $v_2 = 90$, $w_2 = 5$; $v_3 = 130$, $w_3 = 7$; $P = 10$

i\k	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	70	70	70	70	70	70
2	0	0	0	0	0	90	90	90	90	90	160
3	0										

Cálculo da Solução Óptima

$$L(i,k) = \begin{cases} 0 & se \quad i = 0 \vee k = 0 \\ L(i-1,k) & se \quad i > 0 \wedge k > 0 \wedge w_i > k \\ \max(L(i-1,k), L(i-1,k-w_i)+v_i) & caso \quad contrário \end{cases}$$

- Exemplo
  - $v_1 = 70$, $w_1 = 5$; $v_2 = 90$, $w_2 = 5$; $v_3 = 130$, $w_3 = 7$; $P = 10$

i\k	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	70	70	70	70	70	70
2	0	0	0	0	0	90	90	90	90	90	160
3	0	0	0	0	0	90	90	130	130	130	**160**

Programação da Função L

```
int functionL(int[] v, int[] w, int N, int P) {
    int[][] tableL = new int[N+1][P+1];

    for (int k=0; k ≤ P; k++) //Linha 0: Base da recursivade
        tableL[0][k] = 0;

    for (int i=0; i ≤ N; i++) //Coluna 0: Base da recursivade
        tableL[i][0] = 0;

    for (int i=1; i ≤ N; i++) //Caso geral
        for (int k=1; k ≤ P; k++)
            if (w[i] > k)
                tableL[i][k] = tableL[i-1][k];
            else {
                int value = tableL[i-1][k-w[i]] + v[i];
                if (value > tableL[i-1][k])
                    tableL[i][k] = value;
                else
                    tableL[i][k] = tableL[i-1][k];
            }
    return tableL[N][P];
}
```


Complexidade

```
for (int k=0; k ≤ P; k++)
    …
```
$O(P)$

```
for (int i=0; i ≤ N; i++)
    ...
```
$O(N)$

```
for (int i=1; i ≤ N; i++)
    for (int k = 1; k ≤ P; k++)
    ...
```
$O(N \times P)$

Total $O(N \times P)$

Metodologia

- Etapas para a construção de um algoritmo baseado em programação dinâmica:
  1. Caracterizar a estrutura da solução óptima
  2. Definir recursivamente o valor de uma solução óptima, em função de soluções óptimas de subproblemas
  3. Calcular o valor da solução óptima segundo uma estratégia *bottom-up*
  4. **Construir solução a partir da informação obtida**

Construção da Solução

Exemplo

- $v_1 = 70$, $w_1 = 5$; $v_2 = 90$, $w_2 = 5$; $v_3 = 130$, $w_3 = 7$; $P = 10$

i\k	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	70	70	70	70	70	70
2	0	0	0	0	0	90	90	90	90	90	160
3	0	0	0	0	0	90	90	130	130	130	160

Construção da Solução

```
void solution(int[][] tableL, int[] w, int N, int P)

    int k = P;
    for (int i=N; i ≥ 1; i--)
        if (tableL[i][k] ≠ tableL[i-1][k]) {
            print(i);
            k = k - w[i];
        }
}
```


Características da Programação Dinâmica

- Subestrutura óptima:
  - Solução óptima engloba soluções óptimas para subproblemas
- Subproblemas coincidentes:
  - Solução recursiva resolve repetidamente os mesmos subproblemas
    - Sobreposição de problemas
    - Número limitado de subproblemas
    - Solução construtiva para eliminar calculo repetido de subproblemas
- Não é possível aplicar um algoritmo Greedy
  - Porque a escolha óptima local não conduz a uma escolha óptima global

Efectuar Troco

- Dado um conjunto de moedas com valores

  $1 < d_1 < d_2 < \ldots < d_n$

  qual é o menor número de moedas necessárias para efectuar um troco de valor N

- Exemplo:
  - Troco de 3 euros e 45 cêntimos

Relembrar a Metodologia

- Etapas para a construção de um algoritmo baseado em programação dinâmica:
  1. Caracterizar a estrutura da solução óptima
  2. Definir recursivamente o valor de uma solução óptima, em função de soluções óptimas de subproblemas
  3. Calcular o valor da solução óptima segundo uma estratégia *bottom-up*
  4. Construir solução a partir da informação obtida

Efectuar Troco

- Dado um conjunto de moedas com valores

  $1 < d_1 < d_2 < \ldots < d_n$

  qual é o menor número de moedas necessárias para efectuar um troco de valor N

- Aplicar a metodologia referida no slide anterior...

1. Programação Dinâmica

Carla Ferreira
(carla.ferreira@fct.unl.pt)

Licenciatura em Engenharia Informática
DI-FCT-UNL

Maior Sub-sequência Comum

- Dadas duas sequências:
  - $X = x_1 x_2 \ldots x_m$
  - $Y = y_1 y_2 \ldots y_n$

  Determinar a sub-sequência comum de maior comprimento (LCS)

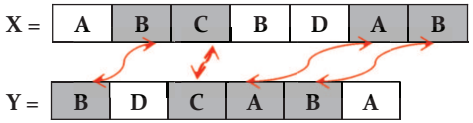
- Exemplo:

X =	A	B	C	B	D	A	B

Y =	B	D	C	A	B	A

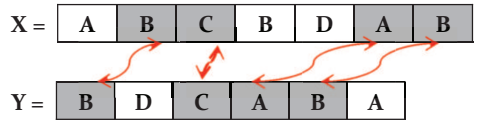
Maior Sub-sequência Comum

- LCS: **BCAB**

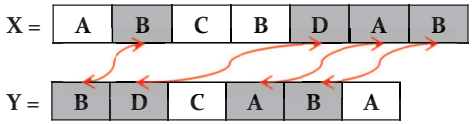


Maior Sub-sequência Comum

- LCS: **BCAB**



- LCS: **BDAB**



Maior Sub-sequência Comum

- Solução Exaustiva
  - Considerar a inclusão de cada caracter de X e Y
  - Total de sub-sequências em X: 2^m
  - Total de sub-sequências em Y: 2^n
  - Total de casos a analisar: 2^{m+n}

Estrutura da Solução Óptima

- X_i denota o prefixo $x_1 \ldots x_i$ de $x_1 \ldots x_m$
  - X_0 denota a sequência vazia
- Y_j denota o prefixo $y_1 \ldots y_j$ de $y_1 \ldots y_n$
  - Y_0 denota a sequência vazia

Estrutura da Solução Óptima

- X_i denota o prefixo $x_1 \dots x_i$ de $x_1 \dots x_m$
  - X_0 denota a sequência vazia
- Y_j denota o prefixo $y_1 \dots y_j$ de $y_1 \dots y_n$
  - Y_0 denota a sequência vazia

- Começar por determinar o comprimento da LCS entre X_m e Y_n
- Assumir o subproblema considerando apenas os prefixos X_i e Y_j
- C(i,j) denota o comprimento da LCS entre as sequências X_i e Y_j

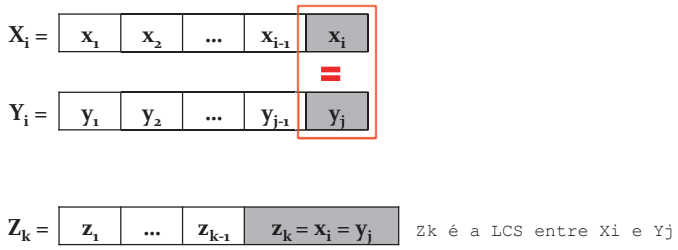
- A solução óptima para o problema inicial está em C(m,n)

Definir Solução Óptima

- Construção da função recursiva C(i,j)
  - Se uma das sequências é a sequência vazia
    - Nesse caso o comprimento da LCS é zero, i.e, C(i,j) = 0

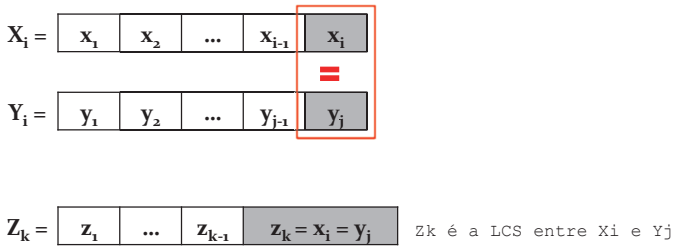
Definir Solução Óptima

- Construção da função recursiva C(i,j)
  - …
  - Se X_i e Y_j terminam no mesmo caracter x_i=y_j



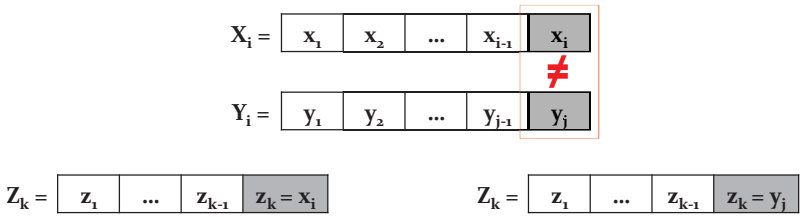
Definir Solução Óptima

- Construção da função recursiva C(i,j)
  - …
  - Se X_i e Y_j terminam no mesmo caracter x_i=y_j
    - Nesse caso o comprimento da LCS é C(i-1,j-1) + 1



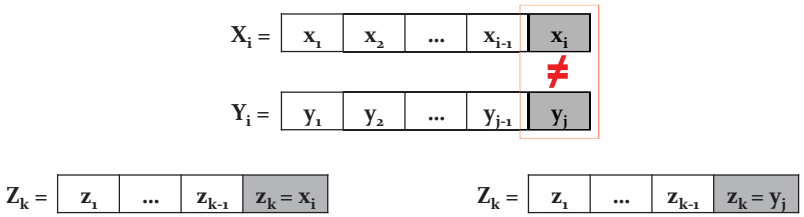
Definir Solução Óptima

- Construção da função recursiva C(i,j)
  - …
  - Se X_i e Y_j terminam em caracteres diferentes $x_i \neq y_j$



Definir Solução Óptima

- Construção da função recursiva C(i,j)
  - …
  - Se X_i e Y_j terminam em caracteres diferentes $x_i \neq y_j$
    - Nesse caso o comprimento da LCS é max(C(i,j-1), C(i-1,j))



Definir Solução Óptima

- Construção da função recursiva C(i,j)

$$C(i,j) = \begin{cases} 0 & se \quad i = 0 \vee j = 0 \\ C(i-1,j-1)+1 & se \quad i > 0 \wedge j > 0 \wedge x_i = y_j \\ \max(C(i-1,j),C(i,j-1)) & se \quad i > 0 \wedge j > 0 \wedge x_i \neq y_j \end{cases}$$

Cálculo da Solução Óptima

$$C(i,j) = \begin{cases} 0 & se \quad i = 0 \vee j = 0 \\ C(i-1,j-1)+1 & se \quad i > 0 \wedge j > 0 \wedge x_i = y_j \\ \max(C(i-1,j),C(i,j-1)) & se \quad i > 0 \wedge j > 0 \wedge x_i \neq y_j \end{cases}$$

		0	1	2	3	4	5	6	7
			A	B	C	B	D	A	B
0		0	0	0	0	0	0	0	0
1	B	0	0	1	1	1	1	1	1
2	D	0	0	1	1	1	2	2	2
3	C	0	0	1	2	2	2	2	2
4	A	0	1	1	2	2	2	3	3
5	B	0	1	2	2	3	3	3	4
6	A	0	1	2	2	3	3	4	4

Programação da Função C

```
int functionC(char[] X, char[] Y, int M, int N) {
    int[][] tableC = new int[M+1][N+1];

    for (int j=0; j ≤ M; j++) //Linha 0: Base da recursivade
        tableC[0][k] = 0;

    for (int i=0; i ≤ N; i++) //Coluna 0: Base da recursivade
        tableC[i][0] = 0;

    for (int i=1; i ≤ M; i++) //Caso geral
        for (int j=1; j ≤ N; j++)
            if (X[i] == Y[j])
                tableC[i][j] = tableC[i-1][j-1] + 1;
            else if (tableC[i-1][j] > tableC[i][j-1])
                tableC[i][j] = tableC[i-1][j];
            else
                tableC[i][j] = tableC[i][j-1];

    return tableC[M][N];
}
```


Complexidade

```
for (int j=0; j ≤ N; j++)
    …
```
$O(N)$

```
for (int i=0; i ≤ M; i++)
    ...
```
$O(M)$

```
for (int i=1; i ≤ M; i++)
    for (int j = 1; j ≤ N; j++)
    ...
```
$O(M \times N)$

Total $O(M \times N)$

Construção da Solução (1/2)

```
void LCS(char[] X, char[] Y, int M, int N, int[][] tableL, char[][] move) {

    ...

    for (int i=1; i ≤ M; i++) //Caso geral
        for (int j=1; j ≤ N; j++)
            if (X[i] == Y[j]) {
                tableC[i][j] = tableC[i-1][j-1] + 1;
                move[i][j] = 'D';
            }
            else if (tableC[i-1][j] ≥ tableC[i][j-1]) {
                    tableC[i][j] = tableC[i-1][j];
                    move[i][j] = 'U';
            }
            else {
                    tableC[i][j] = tableC[i][j-1];
                    move[i][j] = 'L';
            }
}
```


Construção da Solução

		0	1	2	3	4	5	6	7
			A	B	C	B	D	A	B
0		0	0	0	0	0	0	0	0
1	B	0	0 ↑	1 ↖	1 ←	1 ↖	1 ←	1 ←	1 ↖
2	D	0	0 ↑	1 ↑	1 ↑	1 ↑	2 ↖	2 ←	2 ←
3	C	0	0 ↑	1 ↑	2 ↖	2 ←	2 ↑	2 ↑	2 ↑
4	A	0	1 ↖	1 ↑	2 ↑	2 ↑	2 ↑	3 ↖	3 ←
5	B	0	1 ↑	2 ↖	2 ↑	3 ↖	3 ←	3 ↑	4 ↖
6	A	0	1 ↖	2 ↑	2 ↑	3 ↑	3 ↑	4 ↖	4 ↑

Construção da Solução

		0	1	2	3	4	5	6	7
			A	B	C	**B**	**D**	**A**	**B**
0		0	0	0	0	0	0	0	0
1	**B**	0	0 ↑	1 ↖	1 ←	1 ↖	1 ←	1 ←	1 ↖
2	**D**	0	0 ↑	1 ↑	1 ↑	1 ↑	2 ↖	2 ←	2 ←
3	C	0	0 ↑	1 ↑	2 ↖	2 ←	2 ↑	2 ↑	2 ↑
4	**A**	0	1 ↖	1 ↑	2 ↑	2 ↑	2 ↑	3 ↖	3 ←
5	**B**	0	1 ↑	2 ↖	2 ↑	3 ↖	3 ←	3 ↑	4 ↖
6	A	0	1 ↖	2 ↑	2 ↑	3 ↑	3 ↑	4 ↖	4 ↑

Construção da Solução (2/2)

```
void printLCS(char[][] move, char[] X, int I, int J) {
    if ( I>0 && J>0)
        switch (move[I][J]) {
            case 'D' : printLCS(move,X,I-1,J-1);
                       print(X[I]);
                       break;
            case 'U' : printLCS(move,X,I-1,J);;
                       break;
            case 'L' : printLCS(move,X,I,J-1);
                       break;
        }
}
```

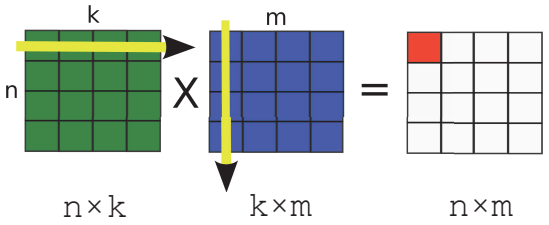

1. Programação Dinâmica

Carla Ferreira
(carla.ferreira@fct.unl.pt)

Licenciatura em Engenharia Informática
DI-FCT-UNL

Problema da Produto de Matrizes

- Produto de matrizes



  - É uma operação associativa
    $(A_1 \times A_2) \times A_3 = A_1 \times (A_2 \times A_3)$
  - Não é uma operação comutativa
    $A_1 \times A_2 \neq A_2 \times A_1$

Produto de Matrizes

```
int[][] prodMatrix(int[] A1, int[] A2, int N, int K, int M) {
    //A1 tem dimensão [N+1][K+1]
    //A2 tem dimensão [K+1][M+1]

    int[][] C = new int[N+1][M+1];

    for (int i=1; i ≤ N; i++)
        for (int j=1; j ≤ M; j++)
            C[i][j] = 0;
            for (int r=1; r ≤ K; r++)
                C[i][j] = C[i][j] + A1[i][r]×A2[r][j]

    return C;
}
```

- São necessárias $N \times K \times M$ multiplicações escalares

Produto de Matrizes

- Exemplo
  - $A_1(10\times100)$, $A_2(100\times10)$, $A_3(10\times50)$, $A_4(50\times30)$
  - Existem 5 colocações possíveis de parêntesis
    - $A_1\times(A_2\times(A_3\times A_4))$ -> 75M multiplicações escalares
    - $A_1\times((A_2\times A_3)\times A_4)$ -> 230M multiplicações escalares
    - $(A_1\times A_2)\times(A_3\times A_4)$ -> 28M multiplicações escalares
    - $(A_1\times(A_2\times A_3))\times A_4$ -> 115M multiplicações escalares
    - $((A_1\times A_2)\times A_3)\times A_4$ -> 30M multiplicações escalares

Produto de Matrizes

- Qual é a ordem pela qual se devem multiplicar as matrizes de forma a minimizar o número de multiplicações?

Problema do Produto de Matrizes

- Dada uma cadeia de produtos de matrizes $A_1\times A_2\times\ldots\times A_n$ determinar a colocação de parêntesis que minimize o número total de multiplicações escalares

Problema do Produto de Matrizes

- Dada uma cadeia de produtos de matrizes $A_1 \times A_2 \times \ldots \times A_n$ determinar a colocação de parêntesis que minimize o número total de multiplicações escalares

$$\underset{p_0}{A_1} \times \underset{p_1}{A_2} \times \underset{p_2}{A_3} \times \underset{p_3}{\ldots} \times \underset{p_{n-1}}{A_n} \underset{p_n}{}$$

  - Matriz A_i tem dimensão $p_{i-1}p_i$ (para $1 \leq i \leq n$)

Estrutura da Solução Óptima

- $A_{i..j}$ denota o produto das matrizes $A_i \times A_{i+1} \times .. \times A_j$
- Assumir que a solução óptima para o produto de matrizes $A_{1..n}$ divide o produto entre A_k e A_{k+1} para algum inteiro k tal que $1 \leq k < n$
  - A colocação parêntesis para $A_{1..k}$ também é óptima
  - Caso contrário existia uma melhor colocação de parêntesis para $A_{1..k}$ e como consequência para $A_{1..n}$

Estrutura da Solução Óptima

- $M(i,j)$ denota o número mínimo de multiplicações escalares para o cálculo de $A_{i..j}$ (com $i \leq j$)
  - Solução óptima para o problema original está em $M(1,n)$

Definir Solução Óptima

- Construção da função recursiva $M(i,j)$
  - Se $i = j$, então $M(i,j)=0$

Definir Solução Óptima

- Construção da função recursiva `M(i,j)`
  - Se `i = j`, então `M(i,j)=0`
  - Se `i < j`, então escolher o valor `k` (com `i≤k<j`) que minimize a última colocação de parêntesis

Definir Solução Óptima

- Construção da função recursiva `M(i,j)`
  - Se `i = j`, então `M(i,j)=0`
  - Se `i < j`, então escolher o valor `k` (com `i≤k<j`) que minimize a última colocação de parêntesis
    - Se `k = i`, então $A_i\ A_{i+1..j}$
    - Se `k = i+1`, então $A_{i..i+1}\ A_{i+2..j}$
    - Se `k = i+2`, então $A_{i..i+2}\ A_{i+3..j}$
    - `...`
    - Se `k = j-1`, então $A_{i..j-1}\ A_j$

Definir Solução Óptima

- Construção da função recursiva `M(i,j)`
  - Se `i = j`, então `M(i,j)=0`
  - Se `i < j`, então escolher o valor `k` (com `i≤k<j`) que minimize a última colocação e parêntesis
    - $A_{i..k}\ A_{k+1..j}$
    - $M(i,j) = M(i,k) + M(k+1,j) + p_{i-1}p_kp_j$

Definir Solução Óptima

- Construção da função recursiva `M(i,j)`

$$M(i,j) = \begin{cases} 0 & se \quad i = j \\ \min\limits_{i \le k < j}(M(i,k) + M(k+1,j) + p_{i-1}p_kp_j) & se \quad i < j \end{cases}$$

Definir Solução Óptima

$$M(i,j) = \begin{cases} 0 & se \quad i = j \\ \min_{i \le k < j}(M(i,k) + M(k+1,j) + p_{i-1}p_k p_j) & se \quad i < j \end{cases}$$

$A_1 \times A_2 \times A_3 \times A_4$

10 100 10 50 30

i\j	1	2	3	4
1	0			
2		0		
3			0	
4				0

Definir Solução Óptima

$$M(i,j) = \begin{cases} 0 & se \quad i = j \\ \min_{i \le k < j}(M(i,k) + M(k+1,j) + p_{i-1}p_k p_j) & se \quad i < j \end{cases}$$

$A_1 \times A_2 \times A_3 \times A_4$

10 100 10 50 30

i\j	1	2	3	4
1	0	10M		
2		0	50M	
3			0	15M
4				0

M(1,2) = min(M(1,1) + M(2,2) + 10×100×10)
= 10000
M(2,3) = min(M(2,2) + M(3,3) + 100×10×50)
= 50000
M(3,4) = min(M(3,3) + M(4,4) + 10×50×30)
= 15000

Definir Solução Óptima

$$M(i,j) = \begin{cases} 0 & se \quad i = j \\ \min_{i \le k < j}(M(i,k) + M(k+1,j) + p_{i-1}p_k p_j) & se \quad i < j \end{cases}$$

$A_1 \times A_2 \times A_3 \times A_4$

10 100 10 50 30

i\j	1	2	3	4
1	0	10M	15M	
2		0	50M	45M
3			0	15M
4				0

M(1,3) = min(M(1,1) + M(2,3) + 10×100×50,
M(1,2) + M(3,3) + 10×10×50)
= min(100000, 15000) = 15000
M(2,4) = min(M(2,2) + M(3,4) + 100×10×30,
M(2,3) + M(4,4) + 100×50×30)
= min(45000, 200000) = 45000

Definir Solução Óptima

$$M(i,j) = \begin{cases} 0 & se \quad i = j \\ \min_{i \le k < j}(M(i,k) + M(k+1,j) + p_{i-1}p_k p_j) & se \quad i < j \end{cases}$$

$A_1 \times A_2 \times A_3 \times A_4$

10 100 10 50 30

i\j	1	2	3	4
1	0	10M	15M	28M
2		0	50M	45M
3			0	15M
4				0

M(1,4) = min(M(1,1) + M(2,4) + 10×100×30,
M(1,2) + M(3,4) + 10×10×30,
M(1,3) + M(4,4) + 10×50×30)
= min(75000, 28000, 30000)
= 28000

Programação da Função M

```
int functionM(int[] dim) {
    int N = dim.length-1;
    int[][] tableM = new int[N+1][N+1];

    for (int i=1; i ≤ N; i++) //Diagonal: cadeia com 1 matriz
        tableM[i][i] = 0;

    for (int h=1; h < N; h++) //Caso geral: cadeia com h+1 matrizes
        for (int i=1; i ≤ N-h ; i++) {
            int j = i + h;
            tableM[i][j] = tableM[i+1][j] + dim[i-1]*dim[i]*dim[j];
            for (int k=i+1; k < j; j++)  {
                int valor = tableM[i][k] + tableM[k+1][j] +
                            dim[i-1]*dim[k]*dim[j];
                if (valor < tableC[i][j])
                    tableM[i][j] = valor;
            }
        }
    return tableM[1][N];
}
```


Complexidade

```
for (int i=1; i ≤ N; i++)                    O(N)
    …
for (int h=1; h < N; h++)                    O(N)
    ...
    for (int i=1; i ≤ N-h; i++)              O(N)      ← O(N^3)
        int j = i+h;
        ...
        for (int k = i+1; k < j; k++)        O(N)
            ...
```

Total $O(N^3)$

Complexidade (ciclo do caso geral)

$$\begin{aligned}
\sum_{h=1}^{N-1}\sum_{i=1}^{N-h}\sum_{k=i}^{j-1} i &= \sum_{h=1}^{N-1}\sum_{i=1}^{N-h}((j-1)-i+1) \\
&= \sum_{h=1}^{N-1}\sum_{i=1}^{N-h} h \\
&= \sum_{h=1}^{N-1}(N-h)h \\
&= \sum_{h=1}^{N-1} Nh - h^2 \\
&= N\sum_{h=1}^{N-1} h + \sum_{h=1}^{N-1} h^2 \\
&= N\frac{(N-1)N}{2} + \frac{(N-1)(2N-1)}{6} \\
&= (\tfrac{1}{2}N^3 + \cdots) - (\tfrac{1}{3}N^3 + \cdots) = \tfrac{1}{6}N^3 + \cdots = O(N^3)
\end{aligned}$$

Construção da Solução

i\j	1	2	3	4
1	0	10M **1**	15M **2**	28M **2**
2		0	50M **2**	45M **2**
3			0	15M **3**
4				0

```
M(1,2) = min(M(1,1) + M(2,2) + 10×100×10)
       = 10000
M(2,3) = min(M(2,2) + M(3,3) + 100×10×50)
       = 50000
M(3,4) = min(M(3,3) + M(4,4) + 10×50×30)
       = 15000
M(1,3) = min( M(1,1) + M(2,3) + 10×100×50,
              M(1,2) + M(3,3) + 10×10×50)
       = min(100000, 15000) = 15000
M(2,4) = min( M(2,2) + M(3,4) + 100×10×30,
              M(2,3) + M(4,4) + 100×50×30)
       = min( 45000, 200000) = 45000
M(1,4) = min( M(1,1) + M(2,4) + 10×100×30,
              M(1,2) + M(3,4) + 10×10×30,
              M(1,3) + M(4,4) + 10×50×30)
       = min( 75000, 28000, 30000)
       = 28000
```

Construção da Solução

```
void matrizMult(int[] dim, int[][] tableM, int[][] pos) {
    int N = dim.length-1;

    ...

    for (int h=1; h < N; h++) //Caso geral: cadeia com h+1 matrizes
        for (int i=1; i ≤ N-h ; i++) {
            int j = i + h;
            tableM[i][j] = tableM[i+1][j] + dim[i-1]*dim[i]*dim[j];
            pos[i][j] = i
            for (int k=i+1; k < j; j++)  {
                int valor = tableM[i][k] + tableM[k+1][j] +
                            dim[i-1]*dim[k]*dim[j];
                if (valor < tableC[i][j]) {
                    tableM[i][j] = valor;
                    pos[i][j] = k
                }
            }
        }
}
```


Construção da Solução

i\j	1	2	3	4
1	0	10M **1**	15M **2**	28M **2**
2		0	50M **2**	45M **2**
3			0	15M **3**
4				0

1 pos(1,4) = 2
$(A_1 \times A_2) \times (A_3 \times A_4)$

2 pos(1,pos(1,4)) = pos(1,2) = 1
- $A_1 \times A_2$

3 pos(pos(1,4)+1,4) = pos(3,4) = 3
- $A_3 \times A_4$

Construção da Solução

i\j	1	2	3	4
1	0	10M **1**	15M **2**	28M **2**
2		0	50M **2**	45M **2**
3			0	15M **3**
4				0

1 pos(1,4) = 2
$(A_1 \times A_2) \times (A_3 \times A_4)$

2 pos(1,pos(1,4)) = pos(1,2) = 1
- $A_1 \times A_2$

2.1 pos(1,pos(1,2)) = pos(1,1)

2.2 pos(pos(1,2)+1,2) = pos(2,2)

3 pos(pos(1,4)+1,4) = pos(3,4) = 3
- $A_3 \times A_4$

Construção da Solução

i\j	1	2	3	4
1	0	10M **1**	15M **2**	28M **2**
2		0	50M **2**	45M **2**
3			0	15M **3**
4				0

1 pos(1,4) = 2
$(A_1 \times A_2) \times (A_3 \times A_4)$

2 pos(1,pos(1,4)) = pos(1,2) = 1
- $A_1 \times A_2$

2.1 pos(1,pos(1,2)) = pos(1,1)

2.2 pos(pos(1,2)+1,2) = pos(2,2)

3 pos(pos(1,4)+1,4) = pos(3,4) = 3
- $A_3 \times A_4$

3.1 pos(3,pos(3,4)) = pos(3,3)

3.2 pos(pos(3,4)+1,4) = pos(4,4)

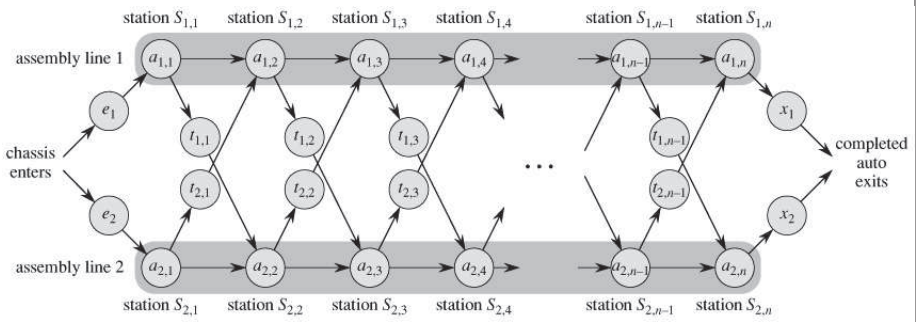
Construção da Solução

```
void printMatrix(int[][] pos, int i, int j) {
    if (i==j)
        print("A" + i);
    else {
        print("(");
        printMatrix(pos,i,pos[i][j]);
        printMatrix(pos,pos[i][j]+1,j);
        print(")");
    }
}
```

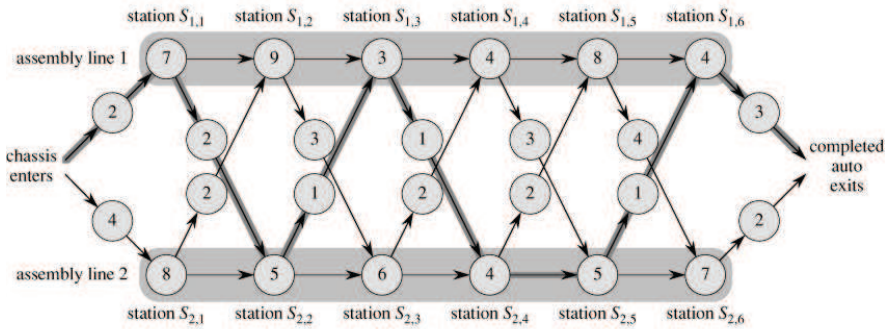

Escalonamento de uma Linha de Montagem

- Existem 2 linhas de montagem, cada uma com n estações de trabalho
  - Estações da linha de montagem 1: $S_{1,1}$, $S_{1,2}$, $S_{1,3}$, . . . , $S_{1,n}$
  - Estações da linha de montagem 2: $S_{2,1}$, $S_{2,2}$, $S_{2,3}$, . . . , $S_{2,n}$
  - Estações $S_{1,j}$ e $S_{2,j}$ realizam a mesma tarefa, mas têm diferentes tempos de produção $a_{1,j}$ e $a_{2,j}$
  - Cada linha de montagem tem um tempo de entrada e_1 e e_2, e um tempo de saída x_1 e x_2
  - Existe um tempo de transferência $t_{i,j}$ da estação $S_{i,j}$ para a estação seguinte na linha de montagem oposta
- Determinar a selecção das estações de trabalho das 2 linhas de montagem que minimizam o tempo de produção

Exemplo: Linha de Montagem



Exemplo: Linha de Montagem



Relembrar a Metodologia

- Etapas para a construção de um algoritmo baseado em programação dinâmica:
  1. Caracterizar a estrutura da solução óptima
  2. Definir recursivamente o valor de uma solução óptima, em função de soluções óptimas de subproblemas
  3. Calcular o valor da solução óptima segundo uma estratégia *bottom-up*
  4. Construir solução a partir da informação obtida

Exemplo: Linha de Montagem

- Determinar a selecção das estações de trabalho das 2 linhas de montagem que minimizam o tempo de produção
- Aplicar a metodologia referida no slide anterior...

2. Conceitos Básicos de Grafos

Planeamento

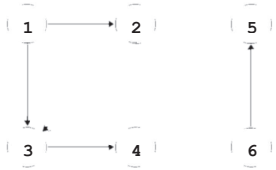
Apresentação da disciplina
Programação Dinâmica
Introdução aos Grafos
Percurso em largura e profundidade
Ordenação Topológica
Árvore Mínima de Cobertura
TAD Partição
Complexidade Amortizada
Algoritmo de Prim
TAD Fila com Prioridade Adaptável
Algoritmo de Dijskstra
Filas Binomiais
Filas de Fibonacci
Algoritmo de Bellman-Ford
Fluxo Máximo: Algoritmo de Ford-Fulkerson
Algoritmo de Edmonds-Karp
Introdução à Teoria da Complexidade
Exemplos de Problemas NP
Redutibilidade entre Problemas de Decisão

O que é um Grafo?

- Informalmente um grafo é um conjunto de nós ligados por um conjunto de linhas ou setas

Grafo Simples

- G = (V, A)
  - V - conjunto de vértices ou nós
  - A - colecção de arcos ou arestas
  - $A \subseteq V \times V$
  - Sem lacetes

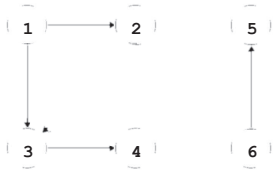


$V = \{1,2,3,4,5,6\}$

$A = \{(1,2),(1,3),(2,3),(3,4),(6,5)\}$

Grafo Orientado

- Os arcos têm sentido

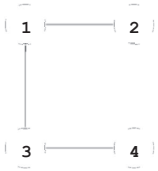


$V = \{1,2,3,4,5,6\}$

$A = \{(1,2),(1,3),(2,3),(3,4),(6,5)\}$

Grafo Não Orientado

- Os arcos não têm sentido, ou seja, $(v,w) = (w,v)$



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

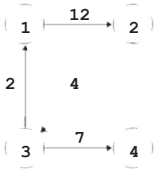
$A = \{(2,1),(3,1),(2,3),(3,4)\}$

Grafo Pesado

- Cada arco têm um peso (ou custo)

Caminho

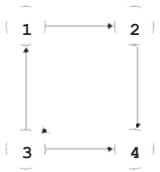
- Sequência não vazia de vértices v_1, v_2, ..., v_n, tal que, para qualquer i = 1,2, ...,n-1 temos que $(v_i, v_{i+1}) \in A$



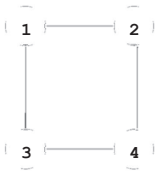
- Caminho: 1,2,3,1,2
- Comprimento: 4
- Comprimento Pesado: 30
- Caminho Simples: 1,2,3,1 ou 1,2,3,4

Ciclo

- Grafo orientado: um caminho onde o primeiro e o último vértices são iguais
- Grafo não orientado: um caminho onde o primeiro e o último vértices são iguais, e que não passa repetidamente pelo mesmo arco



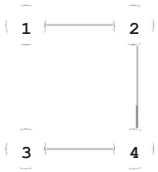
Ciclo: 3,1,2,3



Ciclo: 1,2,4,3,1

Grafo Conexo

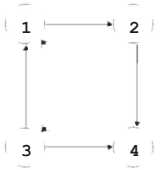
- Grafo não orientado tal que:
  - $\forall v, w \in V$ existe um caminho de v para w

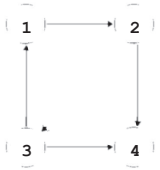


Grafo Fortemente/Fracamente Conexo

- Grafo Fortemente Conexo: grafo orientado tal que:
  - $\forall v,w \in V$ existe um caminho de v para w
- Grafo Fracamente Conexo: grafo orientado tal que, ignorando o sentido dos arcos:
  - $\forall v,w \in V$ existe um caminho de v para w

Fortemente conexo

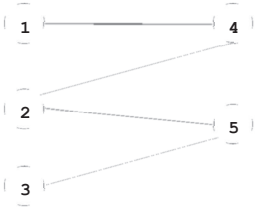




Fracamente conexo

Grafo Bipartido

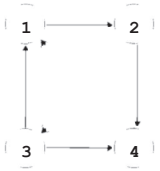
- Grafo não orientado G = (V,E) tal que V pode ser particionado em 2 conjuntos V_1 e V_2 tais que para qualquer arco (v,w) de E, v e w não pertencem ao mesmo subconjunto

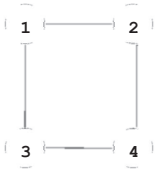


Grau de um Vértice

- Grafo não orientado: número de arcos incidentes
- Grafo orientado
  - Grau de entrada (*in degree*): número de arcos de entrada
  - Grau de saída (*out degree*): número de arcos de saída

Grau de entrada de V1 é 2
Grau de saída de V3 é 2





Número de arcos incidentes de V2 é 3

TAD Vértice

```
public interface Vertex<V> {
   // Returns the vertex label.
   V label( );
}
```

TAD Arco

```
public interface Edge<V,E> {
  // Returns the edge label.
  E label( );

  // Returns an array of length 2 with the edge end-vertices.
  Vertex<V>[] endVertices( );

  // Returns the edge end-vertex that is distinct from the specified vertex.
  Vertex<V> oppositeVertex( Vertex<V> endVertex );

}
```


TAD Grafo

```
public interface AnyGraph<V,E> {
  // Returns the number of vertices.
  int numVertices( );

  // Returns the number of edges.
  int numEdges( );

  // Returns an iterator of the vertices.
  Iterator<Vertex<V>> vertices( );

  // Returns an iterator of the edges.
  Iterator<Edge<V,E>> edges( );

  // Inserts and returns a new (isolated) vertex
  // associated with the specified label.
  Vertex<V> insertVertex( V label );
```


TAD Grafo

```
public interface AnyGraph<V,E> {
  ...
  // Removes the specified vertex and all its incident edges.
  void removeVertex( Vertex<V> vertex );

  // Inserts and returns the edge (vertex1, vertex2) and
  // associates it with the specified label.
  Edge<V,E> insertEdge( Vertex<V> vertex1, Vertex<V> vertex2, E label );

  // Removes the specified edge.
  void removeEdge( Edge<V,E> edge );

  // Returns true iff there is an edge of the form (vertex1, vertex2).
  boolean edgeExists( Vertex<V> vertex1, Vertex<V> vertex2 );
}
```

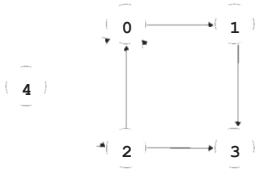

TAD Grafo Não Orientado

```
public interface UndGraph<V,E> extends AnyGraph<V,E> {
  // Returns the degree of the specified vertex.
  int degree( Vertex<V> vertex );

  // Returns an iterator of the vertices adjacent to the specified vertex.
  Iterator<Vertex<V>> adjacentVertices( Vertex<V> vertex );

  // Returns an iterator of the edges incident upon the specified vertex.
  Iterator<Edge<V,E>> incidentEdges( Vertex<V> vertex );
}
```

TAD Grafo Orientado

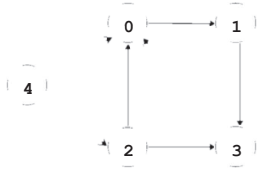
```
public interface Digraph<V,E> extends AnyGraph<V,E> {
   // Returns the in-degree of the specified vertex.
   int inDegree( Vertex<V> vertex );

   // Returns the out-degree of the specified vertex.
   int outDegree( Vertex<V> vertex );

   // Returns an iterator of the vertices adjacent to the specified
   // vertex along incoming edges to it.
   Iterator<Vertex<V>> inAdjacentVertices( Vertex<V> vertex );

   // Returns an iterator of the vertices adjacent to the specified
   // vertex along outgoing edges from it.
   Iterator<Vertex<V>> outAdjacentVertices( Vertex<V> vertex );

   //Returns an iterator of the incoming edges to the specified vertex.
   Iterator<Edge<V,E>> inIncidentEdges( Vertex<V> vertex );

   //Returns an iterator of the outgoing edges from the specified vertex.
   Iterator<Edge<V,E>> outIncidentEdges( Vertex<V> vertex );
```

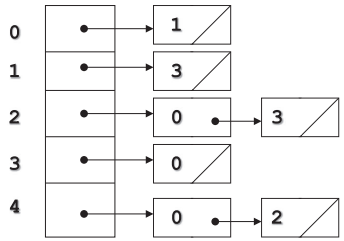

Matriz de Adjacências (Sucessores)



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

Lista de Adjacências (Sucessores)





Complexidade

Complexidade Temporal	Matriz de Adjacências (de Sucessores)	Lista de Adjacências (de Sucessores)
Pesquisar Arco (v,w)	**O(1)**	**O(#Suc(V))**
Pesquisar Sucessores v	**O(#V)**	**O(#Suc(V))**
Pesquisar Antecessores v	**O(#V)**	**O(#A)**
Inserir Arco (v,w)	**O(1)**	**O(#Suc(V))**
Remover Arco (v,w)	**O(1)**	**O(#Suc(V))**

Complexidade

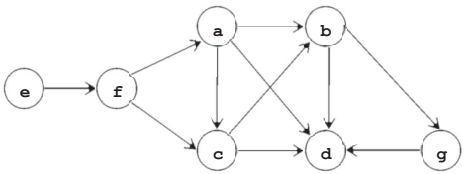
Complexidade Espacial	Matriz de Adjacências	Lista de Adjacências
	$O((\#V)^2)$	$O(\#V + \#E)$

3. Percursos em Largura e Profundidade

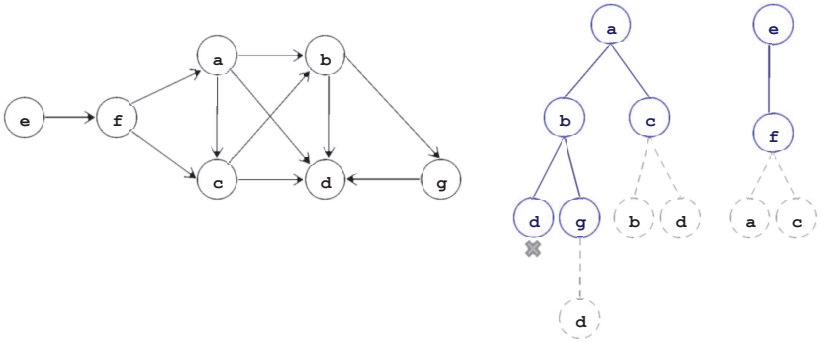
Planeamento

Apresentação da disciplina
Programação Dinâmica
Introdução aos Grafos
Percurso em largura e profundidade
Ordenação Topológica
Árvore Mínima de Cobertura
TAD Partição
Complexidade Amortizada
Algoritmo de Prim
TAD Fila com Prioridade Adaptável
Algoritmo de Dijskstra
Filas Binomiais
Filas de Fibonacci
Algoritmo de Bellman-Ford
Fluxo Máximo: Algoritmo de Ford-Fulkerson
Algoritmo de Edmonds-Karp
Introdução à Teoria da Complexidade
Exemplos de Problemas NP
Redutibilidade entre Problemas de Decisão

Percurso em Profundidade



Percurso em Profundidade



Ordem: **a b d g c e f**

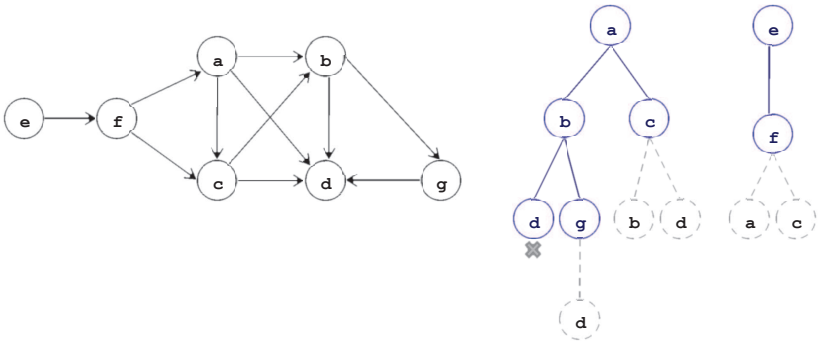
Percurso em Profundidade

```
void dfsTraversal( Graph graph ) {
  boolean[] explored = new boolean[ graph.numVertices() ];

  for every Vertex v in graph.vertices()
      explored[v] = false;

  for every Vertex v in graph.vertices()
      if ( !explored[v] )
          dfsExplore(graph, explored, v);
}
```


Árvore em Profundidade (recursivo)

```
void dfsExplore( Graph graph, boolean[] explored, Vertex root ) {
  TREAT(root);
  explored[root] = true;
  for every Vertex v in graph.outAdjacentVertices(root)
      if ( !explored[v] )
          dfsExplore(graph, explored, v);
}
```


Percurso em Profundidade (recursivo)

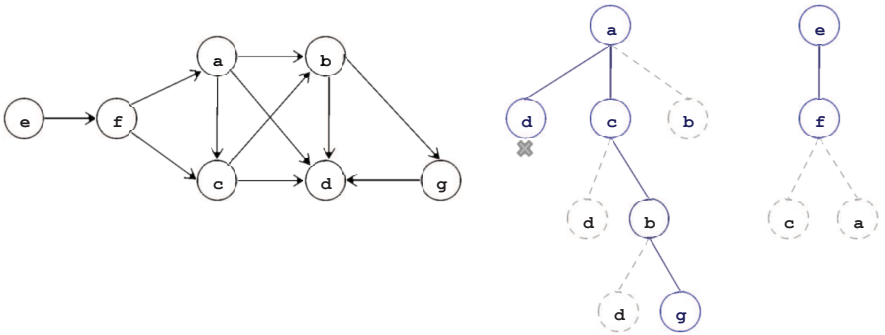


Ordem: **a b d g c e f**

Árvore em Profundidade (iterativo)

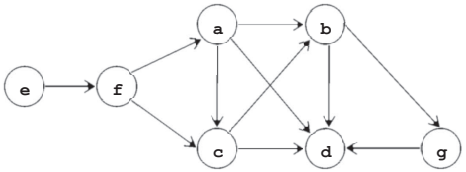
```
void dfsExplore( Graph graph, boolean[] explored, Vertex root ) {
   Stack<Vertex> foundUnexplored = new StackInList<Vertex>();
   foundUnexplored.push(root);
   do {
       Vertex vertex = foundUnexplored.pop();
       if ( !explored[vertex] ) {
           TREAT(vertex); explored[vertex] = true;
           for every Vertex w in graph.outAdjacentVertices(vertex)
                if ( !explored[w] )
                    foundUnexplored.push(w);
       }
   } while ( !foundUnexplored.isEmpty() );
}
```


Percurso em Profundidade (iterativo)

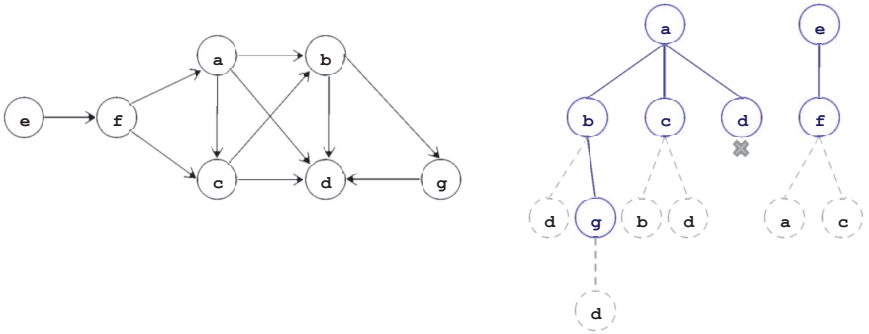


Ordem: **a d c b g e f**

Percurso em Largura



Percurso em Largura



Ordem: **a b c d g e f**

Percurso em Largura

```
void bfsTraversal( Graph graph ) {
  boolean[] found = new boolean[ graph.numVertices() ];

  for every Vertex v in graph.vertices()
      found[v] = false;

  for every Vertex v in graph.vertices()
      if ( !found[v] )
          bfsExplore(graph, found, v);
}
```


Árvore em Largura (iterativo)

```
void bfsExplore( Graph graph, boolean[] found, Vertex root ) {
  Queue<Vertex> waiting= new QueueInArray<Vertex>(graph.numVertices()-1);
  waiting.enqueue(root);
  found[root] = true;
  do {
      Vertex vertex = waiting.dequeue(); TREAT(vertex);
      for every Vertex w in graph.outAdjacentVertices(vertex)
      if ( !found[w] ) {
          waiting.enqueue(w);
          found[w] = true;
      }
  } while ( !waiting.isEmpty() );
}
```


Complexidade

Implementação	Percurso em Profundidade Recursivo ou Percurso em Profundidade Iterativo ou Percurso em Largura Iterativo
Matriz de Adjacências	**O((#V)^2)**
Lista de Adjacências	**O(#V + #A)**

4. Ordenação Topológica

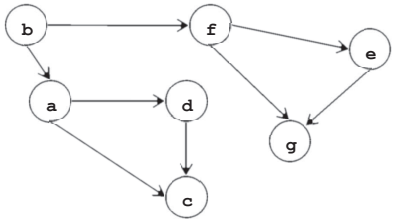
Planeamento

Apresentação da disciplina
Programação Dinâmica
Introdução aos Grafos
Percurso em largura e profundidade
Ordenação Topológica
Árvore Mínima de Cobertura
TAD Partição
Complexidade Amortizada
Algoritmo de Prim
TAD Fila com Prioridade Adaptável
Algoritmo de Dijskstra
Filas Binomiais
Filas de Fibonacci
Algoritmo de Bellman-Ford
Fluxo Máximo: Algoritmo de Ford-Fulkerson
Algoritmo de Edmonds-Karp
Introdução à Teoria da Complexidade
Exemplos de Problemas NP
Redutibilidade entre Problemas de Decisão

Ordenação Topológica

- Dado um grafo G = (V,A), orientado e acíclico, uma ordenação topológica é uma permutação de V tal que

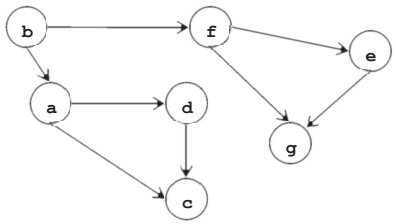
$$\forall x,y \in V \quad (x,y) \in A \Rightarrow x \quad precede \quad y$$



Ordenação Topológica

- Dado um grafo G = (V,A), orientado e acíclico, uma ordenação topológica é uma permutação de V tal que

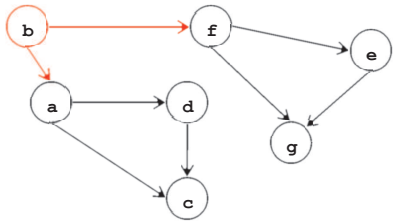
$$\forall x,y \in V \quad (x,y) \in A \Rightarrow x \quad precede \quad y$$



Ordenações:

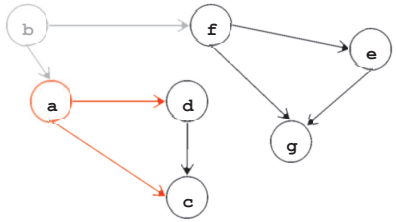
- `b a d c f e g`
- `b f e g a d c`
- `...`

Exemplo



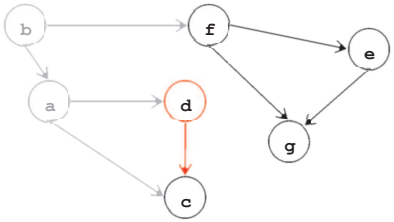
Ordenação: `b`

Exemplo



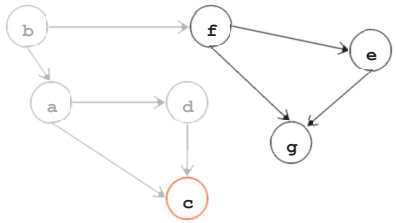
Ordenação: `b a`

Exemplo



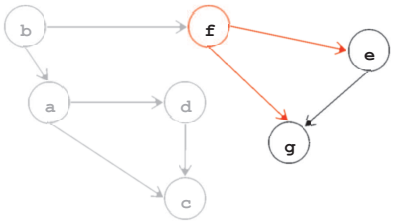
Ordenação: `b a d`

Exemplo



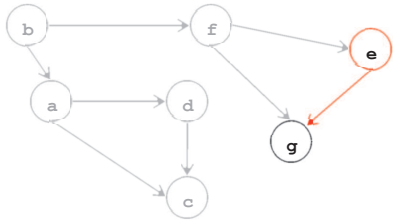
Ordenação: `b a d c`

Exemplo



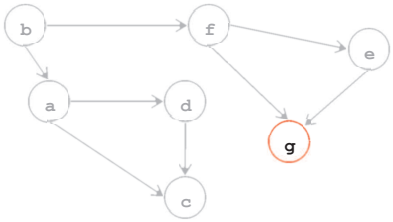
Ordenação: `b a d c f`

Exemplo



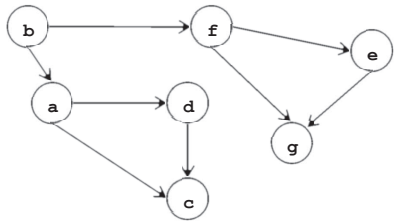
Ordenação: `b a d c f e`

Exemplo



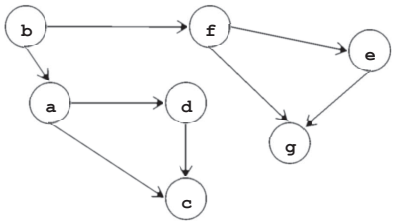
Ordenação: `b a d c f e g`

Antecessores



	Número de antecessores							
a	1							
b	**0**							
c	2							
d	1							
e	1							
f	1							
g	2							
Ordenação	**b**							

Antecessores



	Número de antecessores							
a	1	**0**	–	–	–	–	–	–
b	**0**	–	–	–	–	–	–	–
c	2	2	1	**0**	–	–	–	–
d	1	1	**0**	–	–	–	–	–
e	1	1	1	1	1	**0**	–	–
f	1	0	0	0	**0**	–	–	–
g	2	2	2	2	2	1	**0**	–
Ordenação	**b**	**a**	**d**	**c**	**f**	**e**	**g**	

Ordenação Topológica

```
void topologicalSort( Digraph graph ) {
   Queue<Vertex> ready =
      new QueueInArray<Vertex>( graph.numVertices() )
   int[] inCounter = new int[ graph.numVertices() ];

   for every Vertex v in graph.vertices() {
      inCounter[v] = graph.inDegree(v);
      if ( inCounter[v] == 0 )
          ready.enqueue(v);
   }
```


Ordenação Topológica

```
   while ( !ready.isEmpty() ) {
      Vertex vertex = ready.dequeue();
      TREAT(vertex);
      for every Vertex w in graph.outAdjacentVertices(vertex) {
          inCounter[w]--;
          if ( inCounter[w] == 0 )
            ready.enqueue(w);
      }
   }
}
```


Teste à Aciclicidade

```
boolean isAcyclic( Digraph graph ) {
   Queue<Vertex> ready =
      new QueueInArray<Vertex>( graph.numVertices() );
   int[] inCounter = new int[ graph.numVertices() ];

   for every Vertex v in graph.vertices() {
      inCounter[v] = graph.inDegree(v);
      if ( inCounter[v] == 0 )
          ready.enqueue(v);
   }
   int numSortedVertices = 0;
```


Teste à Aciclicidade

```
   while ( !ready.isEmpty() ) {
      Vertex vertex = ready.dequeue();
      TREAT(vertex);
      numSortedVertices++;
      for every Vertex w in graph.outAdjacentVertices(vertex) {
          inCounter[w]--;
          if ( inCounter[w] == 0 )
            ready.enqueue(w);
      }
   }
   return numSortedVertices == graph.numVertices();
}
```

Complexidade

Implementação	Ordenação Topológica (grafo orientado e acíclico) ou Teste à Aciclicidade (grafo orientado)
Matriz de Adjacências	$O((\#V)^2)$
Lista de Adjacências	$O(\#V + \#A)$

5. Árvore Mínima de Cobertura

Planeamento

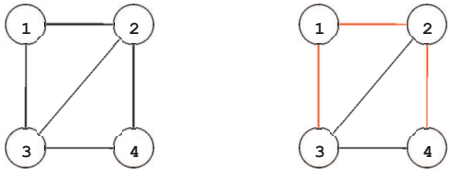
Apresentação da disciplina
Programação Dinâmica
Introdução aos Grafos
Percurso em largura e profundidade
Ordenação Topológica
Árvore Mínima de Cobertura
TAD Partição
Complexidade Amortizada
Algoritmo de Prim
TAD Fila com Prioridade Adaptável
Algoritmo de Dijskstra
Filas Binomiais
Filas de Fibonacci
Algoritmo de Bellman-Ford
Fluxo Máximo: Algoritmo de Ford-Fulkerson
Algoritmo de Edmonds-Karp
Introdução à Teoria da Complexidade
Exemplos de Problemas NP
Redutibilidade entre Problemas de Decisão

Árvore de Cobertura

- Um grafo G=(V,A) não orientado, diz-se **conexo** se
  - $\forall v,w \in V$ existe um caminho de v para w
- Um grafo G=(V,A) não orientado, diz-se um **árvore** se é acíclico e conexo
- Dado grafo não dirigido e conexo G = (V, A), uma **árvore abrangente** é sub-conjunto acíclico T $\subseteq$ A, que liga todos os vértices

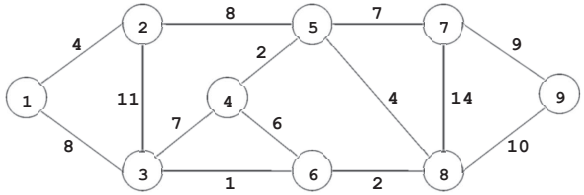
Árvore de Cobertura

- Um grafo G=(V,A) não orientado, diz-se **conexo** se
  - $\forall v,w \in V$ existe um caminho de v para w
- Um grafo G=(V,A) não orientado, diz-se um **árvore** se é acíclico e conexo
- Dado grafo não dirigido e conexo G = (V, E), uma **árvore abrangente** é sub-conjunto acíclico T ⊆ E, que liga todos os vértices



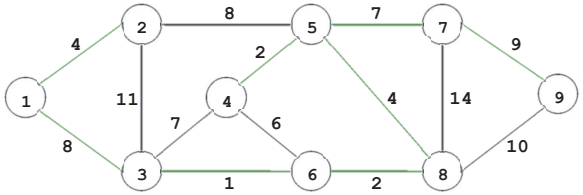
Árvore Mínima de Cobertura

- Árvore de cobertura de custo mínimo (nenhuma árvore de cobertura tem custo menor)



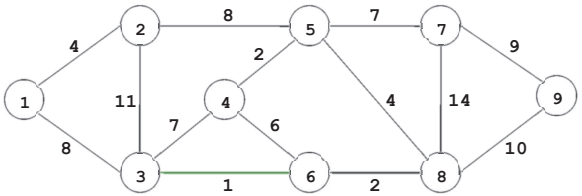
Árvore Mínima de Cobertura

- Árvore de cobertura de custo mínimo (nenhuma árvore de cobertura tem custo menor).



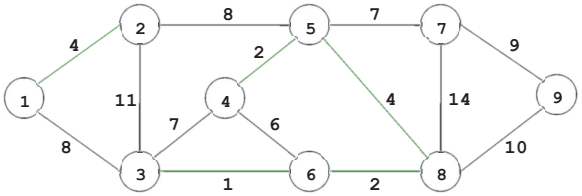
Algoritmo de Kruskal

```
1  (3,6)     7  (5,7)
2  (4,5)     8  (1,3)
2  (6,8)     8  (2,5)
4  (1,2)     9  (7,9)
4  (5,8)     10 (8,9)
6  (4,6)     11 (2,3)
7  (3,4)     14 (7,8)
```



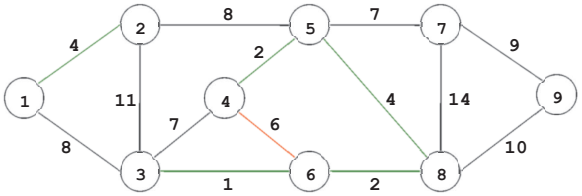
Algoritmo de Kruskal

1	(3,6)	7	(5,7)
2	(4,5)	8	(1,3)
2	(6,8)	8	(2,5)
4	(1,2)	9	(7,9)
4	(5,8)	10	(8,9)
6	(4,6)	11	(2,3)
7	(3,4)	14	(7,8)

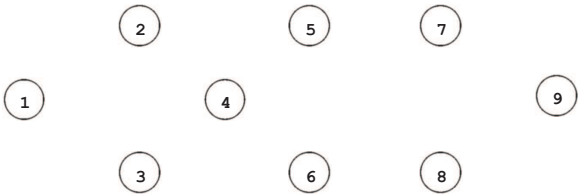


Algoritmo de Kruskal

1	(3,6)	7	(5,7)
2	(4,5)	8	(1,3)
2	(6,8)	8	(2,5)
4	(1,2)	9	(7,9)
4	(5,8)	10	(8,9)
6	(4,6)	11	(2,3)
7	(3,4)	14	(7,8)



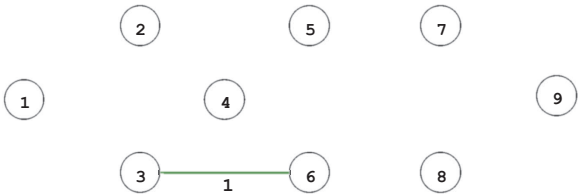
Algoritmo de Kruskal



{1} {2} {3} {4} {5} {6} {7} {8} {9}

Algoritmo de Kruskal

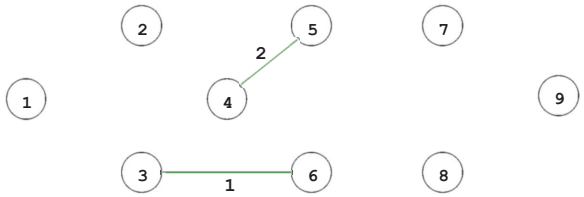
1 (3,6)



{1} {2} {3,6} {4} {5} {7} {8} {9}

Algoritmo de Kruskal

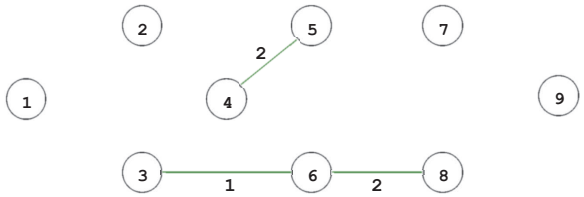
```
1  (3,6)
2  (4,5)
```




```
{1} {2} {3,6} {4,5} {7} {8} {9}
```


Algoritmo de Kruskal

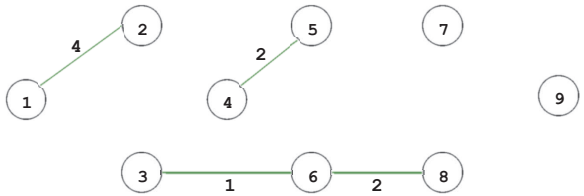
```
1  (3,6)
2  (4,5)
2  (6,8)
```




```
{1} {2} {3,6,8} {4,5} {7} {9}
```


Algoritmo de Kruskal

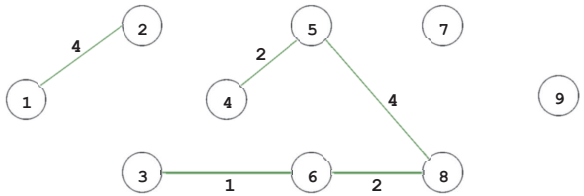
```
1  (3,6)
2  (4,5)
2  (6,8)
4  (1,2)
```




```
{1,2} {3,6,8} {4,5} {7} {9}
```


Algoritmo de Kruskal

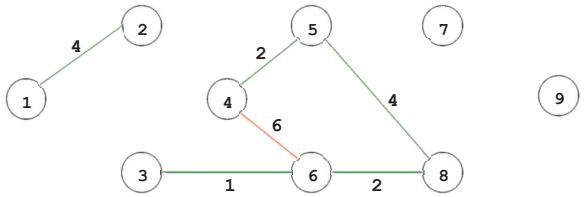
```
1  (3,6)
2  (4,5)
2  (6,8)
4  (1,2)
4  (5,8)
```




```
{1,2} {3,6,8,4,5} {7} {9}
```

Algoritmo de Kruskal

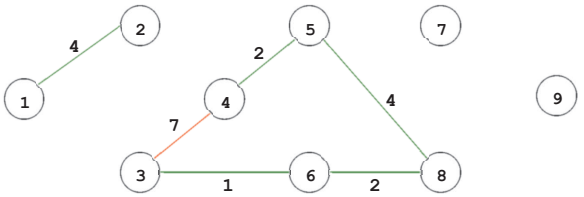
```
1  (3,6)
2  (4,5)
2  (6,8)
4  (1,2)
4  (5,8)
6  (4,6)
```




```
{1,2} {3,6,8,4,5} {7} {9}
```


Algoritmo de Kruskal

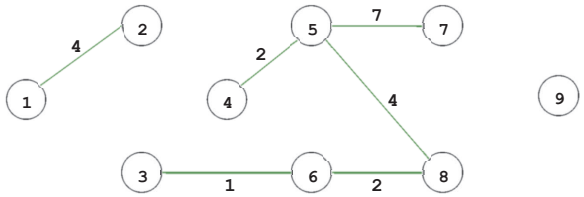
```
1  (3,6)
2  (4,5)
2  (6,8)
4  (1,2)
4  (5,8)
6  (4,6)
7  (3,4)
```




```
{1,2} {3,6,8,4,5} {7} {9}
```


Algoritmo de Kruskal

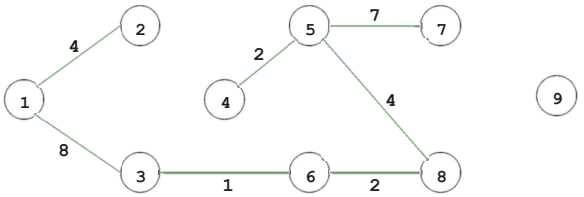
```
1  (3,6)    7  (5,7)
2  (4,5)
2  (6,8)
4  (1,2)
4  (5,8)
6  (4,6)
7  (3,4)
```




```
{1,2} {3,6,8,4,5,7} {9}
```


Algoritmo de Kruskal

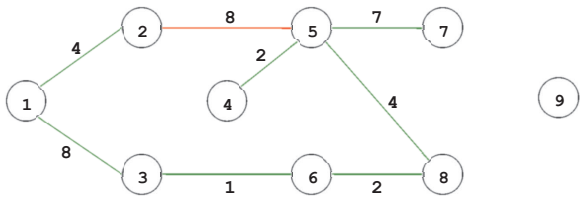
```
1  (3,6)    7  (5,7)
2  (4,5)    8  (1,3)
2  (6,8)
4  (1,2)
4  (5,8)
6  (4,6)
7  (3,4)
```




```
{1,2,3,6,8,4,5,7} {9}
```

Algoritmo de Kruskal

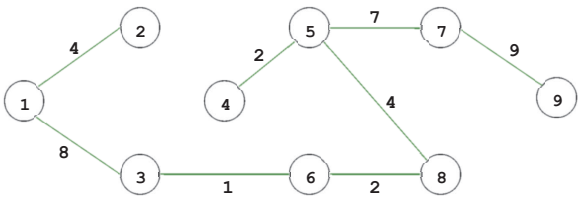
```
1  (3,6)     7  (5,7)
2  (4,5)     8  (1,3)
2  (6,8)     8  (2,5)
4  (1,2)
4  (5,8)
6  (4,6)
7  (3,4)
```




```
{1,2,3,6,8,4,5,7} {9}
```


Algoritmo de Kruskal

```
1  (3,6)     7  (5,7)
2  (4,5)     8  (1,3)
2  (6,8)     8  (2,5)
4  (1,2)     9  (7,9)
4  (5,8)
6  (4,6)
7  (3,4)
```




```
{1,2,3,6,8,4,5,7,9}
```


TAD Partição (n elementos)

- 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
- Os elementos dos conjuntos são `0,1,2, … ,n-1`
- Domínio = `{0, 1, ..., n-1}`

Interface UnionFind

```
public interface UnionFind {

   // Creates the partition {{0}, {1}, . . . , {domainSize – 1}}
   // UnionFind( int domainSize );

   // Returns the representative of the set that contains
   // the specified element.
   int find( int element ) throws InvalidElementException;

   // Removes the two distinct sets S1 and S2 whose representatives
   // are the specified elements, and inserts the set S1 U S2.
   // The representative of the new set S1 U S2 can be any of
   // its members.
   void union( int representative1, int representative2 ) throws
    InvalidElementException, NotRepresentativeException,
    EqualSetsException;
}
```

Fila de Prioridade de Arcos

```
MinPriorityQueue<E,Edge<?,E>> buildEdgesPQ( UndiGraph<?,E> graph ) {
   int size = graph.numEdges();
   Entry<E,Edge<?,E>>[] auxArray =
        (Entry<E,Edge<?,E>>[]) new Entry[size];

   int pos = 0;
   for every Edge<?,E> e in graph.edges()
    auxArray[pos++] = new EntryClass<E,Edge<?,E>>(e.label(), e);

   MinPriorityQueue<E,Edge<?,E>> priQueue =
    new MinHeap<E,Edge<?,E>>(size, auxArray, size);

   return priQueue;
}
```


Árvore Mínima de Cobertura (Kruskal)

```
Iterator<Edge<?,E>> mstKruskal( UndiGraph<?,E> graph ) {
   MinPriorityQueue<E,Edge<?,E>> allEdges = buildEdgesPQ(graph);

   UnionFind vertPartition =
    new UnionFindInArray( graph.numVertices() );

   List<Edge<?,E>> mst = new DoublyLinkedList<Edge<?,E>>();

   int mstFinalSize = graph.numVertices() - 1;
```


Árvore Mínima de Cobertura (Kruskal)

```
   while ( mst.size() < mstFinalSize ) {
        Edge<?,E> edge = allEdges.removeMin().getValue();
        Vertex<?>[] endPoints = edge.endVertices();

        int rep1 = vertPartition.find( endPoints[0] );
        int rep2 = vertPartition.find( endPoints[1] );
        if ( rep1 != rep2 ) {
            mst.addLast(edge);
            vertPartition.union(rep1, rep2);
        }
   }
   return mst.iterator();
}
```


Árvore Mínima de Cobertura (Kruskal)

```
Iterator<Edge<?,E>> mstKruskal( UndiGraph<?,E> graph ) {
   MinPriorityQueue<E,Edge<?,E>> allEdges = buildEdgesPQ(graph);

   UnionFind vertPartition =
    new UnionFindInArray( graph.numVertices() );

   List<Edge<?,E>> mst = new DoublyLinkedList<Edge<?,E>>();

   int mstFinalSize = graph.numVertices() - 1;
   while ( mst.size() < mstFinalSize ) {
        Edge<?,E> edge = allEdges.removeMin().getValue();
        Vertex<?>[] endPoints = edge.endVertices();

        int rep1 = vertPartition.find( endPoints[0] );
        int rep2 = vertPartition.find( endPoints[1] );
        if ( rep1 != rep2 ) {
            mst.addLast(edge);
            vertPartition.union(rep1, rep2);
        }
   }
   return mst.iterator();
}
```

Complexidade (a completar...)

- Criar fila de prioridade — O(#A)
- Criar partição — ?
- Criar lista — O(1)
- Ciclo (executado entre #V – 1 e #A vezes)
  - Instruções executadas entre #V – 1 e #A vezes
    - Remover mínimo — O(log#A)
    - Seleccionar 2 representantes — ?
  - Instruções executadas #V-1 vezes
    - Adicionar vertíce na lista — O(1)
    - Unir representantes — ?

União sem Estratégia Representante sem Efeitos Laterais

- Criar fila de prioridade — O(#A)
- Criar partição — O(#V)
- Criar lista — O(1)
- Ciclo (executado entre #V – 1 e #A vezes)
  - Instruções executadas entre #V – 1 e #A vezes
    - Remover mínimo — O(log#A)
    - Seleccionar 2 representantes — O(#V)
  - Instruções executadas #V-1 vezes
    - Adicionar vertíce na lista — O(1)
    - Unir representantes — O(1)
- Total — O(#A#V)

União sem Estratégia Representante sem Efeitos Laterais

$$O(\#A + \#V + \#A(\log\#A + \#V)) =$$

$$O(\#A \log\#A + \#A\#V) =$$

$$O(\#A \log\#V + \#A\#V) =$$

$$O(\#A\#V)$$

União por Dimensão ou por Altura Representante sem Efeitos Laterais

- Criar fila de prioridade — O(#A)
- Criar partição — O(#V)
- Criar lista — O(1)
- Ciclo (executado entre #V – 1 e #A vezes)
  - Instruções executadas entre #V – 1 e #A vezes
    - Remover mínimo — O(log#A)
    - Seleccionar 2 representantes — O(log#V)
  - Instruções executadas #V-1 vezes
    - Adicionar vertíce na lista — O(1)
    - Unir representantes — O(1)
- Total — O(#Alog#V)

União por Dimensão ou por Altura Representante sem Efeitos Laterais

$$O(\#A + \#V + \#A(\log\#A + \log\#V)) =$$

$$O(\#A \log\#A + \#A \log\#V) =$$

$$O(\#A \log\#V + \#A \log\#V) =$$

$$O(\#A \log\#V) =$$

União por Dimensão ou por Altura Representante com Compressão do Caminho

- Criar fila de prioridade O(#A)
- Criar partição O(#V)
- Criar lista O(1)
- Ciclo (executado entre #V – 1 e #A vezes)
  - Instruções executadas entre #V – 1 e #A vezes
    - Remover mínimo O(log#A)
    - Seleccionar 2 representantes O(log#V)
  - Instruções executadas #V-1 vezes
    - Adicionar vertíce na lista O(1)
    - Unir representantes O(1)
- Total O(#Alog#V)

União por Dimensão ou por Altura Representante com Compressão do Caminho

$$O(\#A + \#V + \#A(\log\#A + 2R)) =$$

$$O(\#A \log\#A + \#A (2R)) =$$

$$O(\#A \log\#V + \#A \alpha(\#V)) =$$
$$(\alpha(\#V) \leq 4)$$

$$O(\#A \log\#V + \#A) =$$

$$O(\#A \log\#V)$$

6. TAD Partição (Conjuntos Disjuntos)

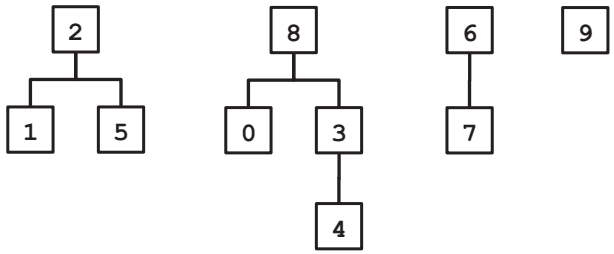
Planeamento

Apresentação da disciplina
Programação Dinâmica
Introdução aos Grafos
Percurso em largura e profundidade
Ordenação Topológica
Árvore Mínima de Cobertura
TAD Partição
Complexidade Amortizada
Algoritmo de Prim
TAD Fila com Prioridade Adaptável
Algoritmo de Dijskstra
Filas Binomiais
Filas de Fibonacci
Algoritmo de Bellman-Ford
Fluxo Máximo: Algoritmo de Ford-Fulkerson
Algoritmo de Edmonds-Karp
Introdução à Teoria da Complexidade
Exemplos de Problemas NP
Redutibilidade entre Problemas de Decisão

Interface UnionFind

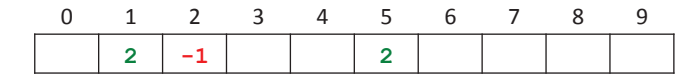
```
public interface UnionFind {

   // Creates the partition {{0}, {1}, . . . , {domainSize – 1}}
   // UnionFind( int domainSize );

   // Returns the representative of the set that contains
   // the specified element.
   int find( int element ) throws InvalidElementException;

   // Removes the two distinct sets S1 and S2 whose representatives
   // are the specified elements, and inserts the set S1 U S2.
   // The representative of the new set S1 U S2 can be any of
   // its members.
   void union( int representative1, int representative2 ) throws
    InvalidElementException, NotRepresentativeException,
    EqualSetsException;
}
```

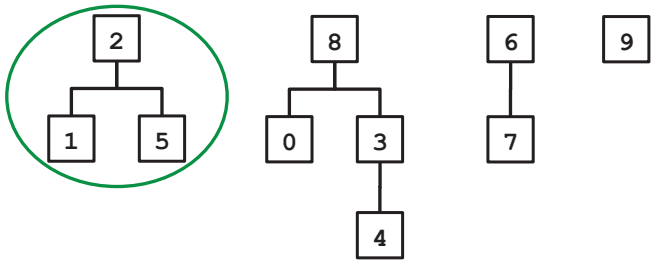

Implementação em Floresta



{1,2,5} {0,3,4,8} {6,7} {9}

Implementação em Floresta

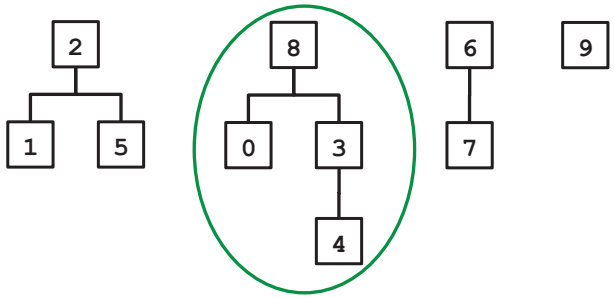




{1,2,5} {0,3,4,8} {6,7} {9}

Implementação em Floresta

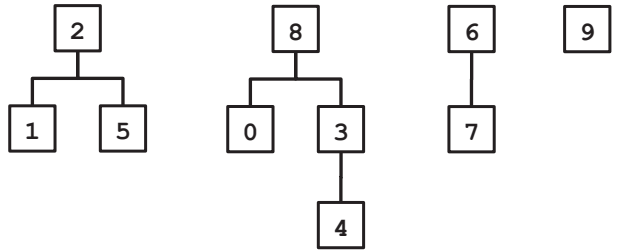
0	1	2	3	4	5	6	7	8	9
8			8	3				-1	



{1,2,5} {0,3,4,8} {6,7} {9}

Implementação em Floresta

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

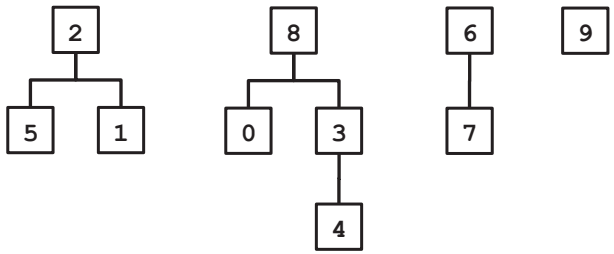


{1,2,5} {0,3,4,8} {6,7} {9}

Complexidade (Pior Caso)

- Criar partição (dimensão n) O(n)
- Representante O(n)
- União O(1)

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



Complexidade: Algoritmo Kruskal

Criar fila de prioridade	O(#A)
Criar partição	O(#V)
Criar lista	O(1)
Ciclo (executado entre #V – 1 e #A vezes)	
Instruções executadas entre #V – 1 e #A vezes	
Remover mínimo	O(log #A)
Seleccionar 2 representantes	O(#V)
Instruções executadas #V-1 vezes	
Adicionar vertíce na lista	O(1)
Unir representantes	O(1)
Total (união sem estratégia)	O(#A#V)

Complexidade: Algoritmo Kruskal

$$O(\#A + \#V + \#A(\log \#A + \#V) + (\#V-1)) =$$

$$O(\ \#A + \#V + \#A \log \#A + \#A\#V + \#V) =$$

$(\#A < \#A\#V \text{ e } \#V < \#A\#V)$

$$O(\ \#A \log \#A\ +\ \#A\#V\ ) =$$

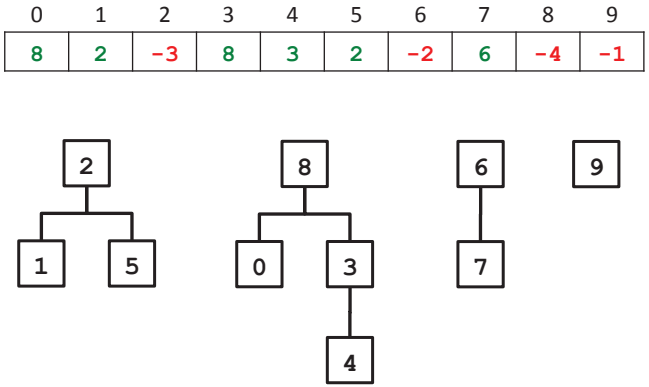
$(\#A < \#V^2)$

$$O(\ \#A \log \#V\ +\ \#A\#V\ ) =$$

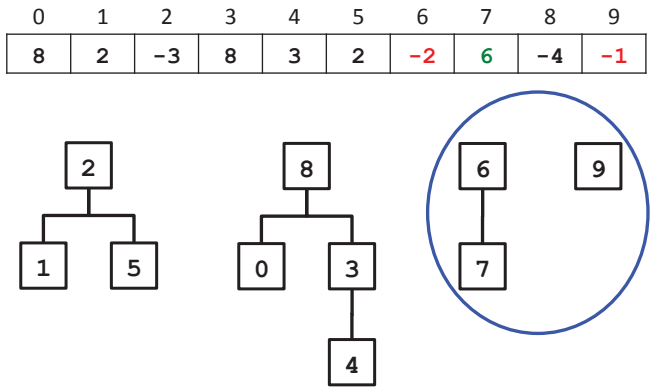
$(\#A < \#V\#V)$

$$O(\ \#A\#V\ )$$

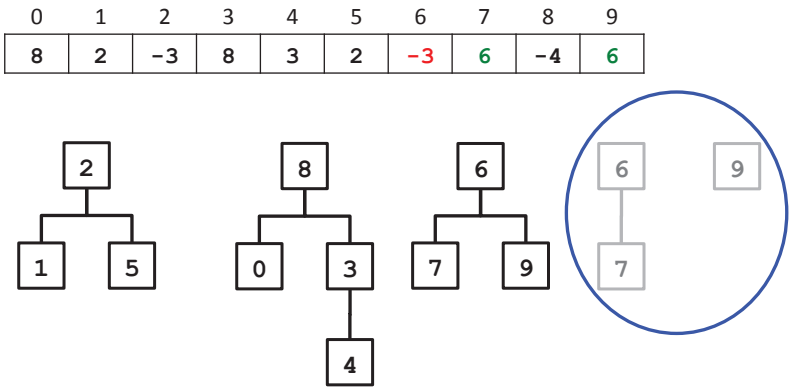
União por Dimensão



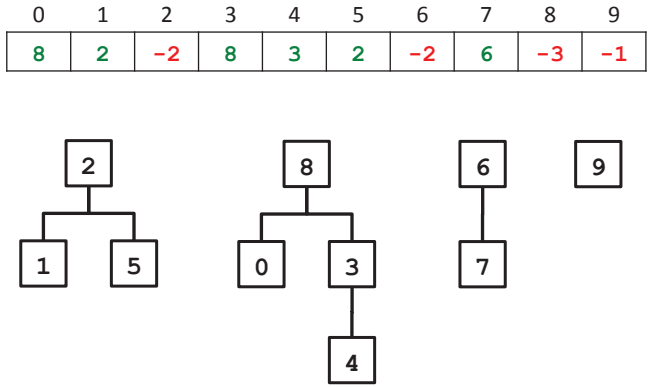
União por Dimensão



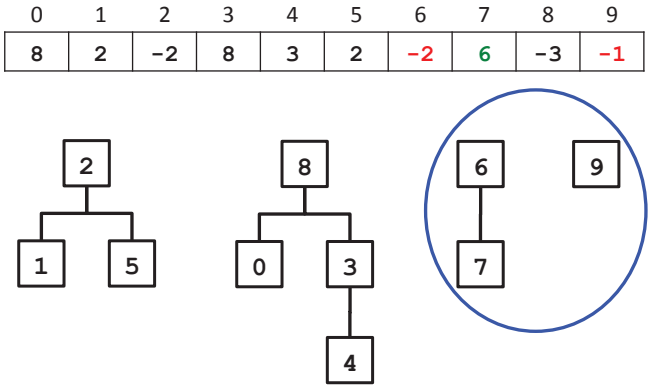
União por Dimensão



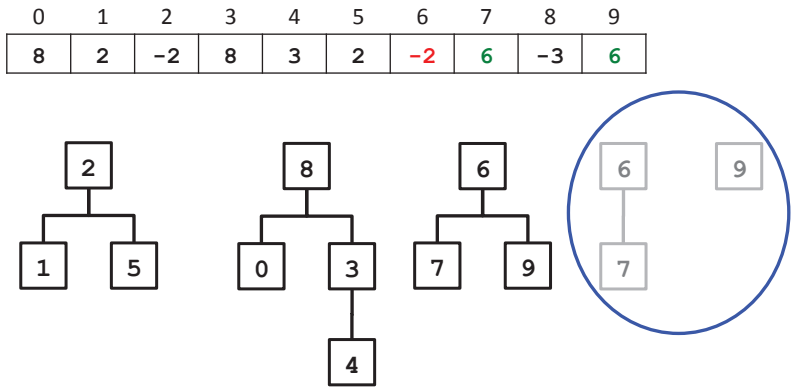
União por Altura



União por Altura

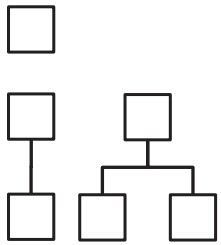


União por Altura



Complexidade
Representante com União por Dimensão

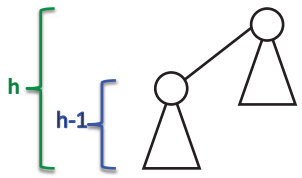
Número **mínimo** de nós de uma árvore com altura h



```
nodes(1) = 1

nodes(2) = 2
```

Complexidade
Representante com União por Dimensão

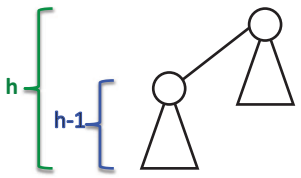
Número **mínimo** de nós de uma árvore com altura h



$$
\begin{aligned}
nodes(1) &= 1 \\
nodes(2) &= 2 \\
nodes(h) &= 2 \times nodes(h-1)
\end{aligned}
$$

Complexidade
Representante com União por Dimensão

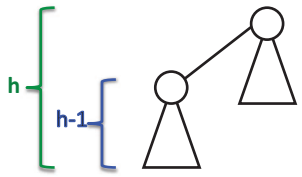
Número **mínimo** de nós de uma árvore com altura h



$$
\begin{aligned}
nodes(1) &= 1 && = 2^0 \\
nodes(2) &= 2 && = 2^1 \\
nodes(h) &= 2 \times nodes(h-1) && = 2 \times 2^{(h-1)-1} \\
& && = 2^{h-1}
\end{aligned}
$$

Complexidade
Representante com União por Altura

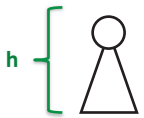
Número **mínimo** de nós de uma árvore com altura h



$$
\begin{aligned}
nodes(1) &= 1 && = 2^0 \\
nodes(2) &= 2 && = 2^1 \\
nodes(h) &= 2 \times nodes(h-1) && = 2 \times 2^{(h-1)-1} \\
& && = 2^{h-1}
\end{aligned}
$$

Complexidade
Representante com União por Dimensão ou Altura

Dado um conjunto de dimensão n, existe um h tal que:



$$2^{(h-1)} \leq n < 2^h$$

$$2^{(h-1)} \leq n$$

$$h-1 \leq \log(n)$$

$$\mathbf{h \leq 1+\log(n)}$$

Complexidade: Algoritmo Kruskal

Criar fila de prioridade	O(#A)
Criar partição	O(#V)
Criar lista	O(1)
Ciclo (executado entre #V – 1 e #A vezes)	
Instruções executadas entre #V – 1 e #A vezes	
Remover mínimo	O(log #A)
Seleccionar 2 representantes	O(log #V)
Instruções executadas #V-1 vezes	
Adicionar vertíce na lista	O(1)
Unir representantes	O(1)
Total (união por altura ou dimensão)	O(#A log #V)

Complexidade: Algoritmo Kruskal

União por altura ou dimensão

$$O(\#A + \#V + \#A(\log \#A + \log \#V) + (\#V\text{-}1)) =$$

$$O(\ \#A + \#V + \#A \log \#A + \#A \log \#V + \#V) =$$

$(\#A < \#A \log \#A \text{ e } \#V < \#A \log \#A)$

$$O(\ \#A \log \#A\ +\ \#A \log \#V\ ) =$$

$(\#A < \#V^2)$

$$O(\ \#A \log \#V\ +\ \#A \log \#V\ ) =$$

$$O(\ \#A \log \#V\ )$$

Interface UnionFind

```
public interface UnionFind {

   // Creates the partition {{0}, {1}, . . . , {domainSize – 1}}
   // UnionFind( int domainSize );

   // Returns the representative of the set that contains
   // the specified element.
   int find( int element ) throws InvalidElementException;

   // Removes the two distinct sets S1 and S2 whose representatives
   // are the specified elements, and inserts the set S1 U S2.
   // The representative of the new set S1 U S2 can be any of
   // its members.
   void union( int representative1, int representative2 ) throws
    InvalidElementException, NotRepresentativeException,
    EqualSetsException;
}
```


Classe Partição em Vector

```
public class UnionFindInArray implements UnionFind {
   // The partition is a forest implemented in an array
   protected int[] partition;

   // Definition of the range of valid elements
   protected String validRangeMsg;
```

Criar Partição

```
// Creates the partition {{0}, {1}, . . . , {domainSize – 1}}
public UnionFindInArray( int domainSize ) {
     partition = new int[domainSize];
     for ( int i = 0; i < domainSize; i++ )
         partition[i] = -1;

     int lastElement = domainSize – 1;
     validRangeMsg =
         “Range of valid elements: 0, 1, ..., ” + lastElement;
}
```


Validação

```
protected boolean isInTheDomain( int number ) {
     return ( number >= 0 ) && ( number < partition.length );
}

// Pre-condition: 0 <= element < partition.length
protected boolean isRepresentative( int element ) {
     return partition[element] < 0;
}
```


Representante (Recursivo)

```
public int find( int element ) throws InvalidElementException {
     if ( !this.isInTheDomain(element) )
         throw new InvalidElementException(validRangeMsg);
     return this.findRec(element);
}

// Pre-condition: 0 <= element < partition.length
protected int findRec( int element ) {
     if ( partition[element] < 0 )
         return element;
     else
         return this.findRec( partition[element] );
}
```


Representante (Iterativo)

```
public int find( int element ) throws InvalidElementException {
     if ( !this.isInTheDomain(element) )
         throw new InvalidElementException(validRangeMsg);

     int node = element;
     while( partition[node] >= 0 )
         node = partition[node];
     return node;
}
```

União por Dimensão

```
public void union( int rep1, int rep2 ) throws
    InvalidElementException, NotRepresentativeException,
    EqualSetsException {

    if ( !this.isInTheDomain(rep1) || ! this.isInTheDomain(rep2) )
        throw new InvalidElementException(validRangeMsg);
    if ( !this.isRepresentative(rep1) )
        throw new NotRepresentativeException("First argument");
    if ( !this.isRepresentative(rep2) )
        throw new NotRepresentativeException("Second argument");
    if ( rep1 == rep2 )
        throw new EqualSetsException("The two arguments are equal");
```


União por Dimensão

```
        if ( partition[rep1] <= partition[rep2] ) {
            // Size(S1) >= Size(S2)
            partition[rep1] += partition[rep2];
            partition[rep2] = rep1;
        }
        else {
            // Size(S1) < Size(S2)
            partition[rep2] += partition[rep1];
            partition[rep1] = rep2;
        }
    }//Closes union method
```

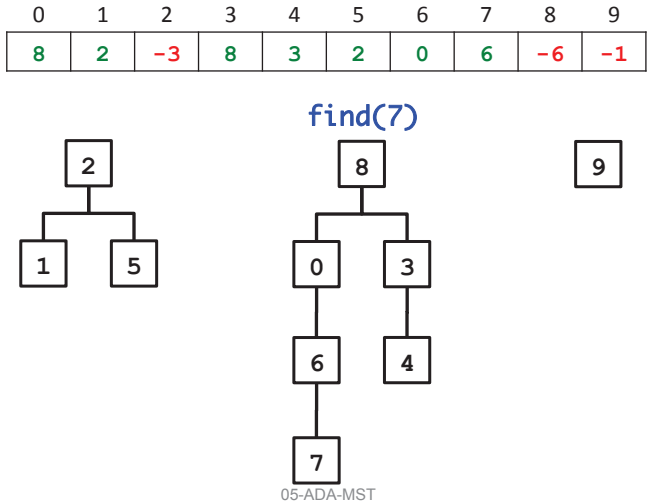

União por Altura

```
public void union( int rep1, int rep2 ) throws
    InvalidElementException, NotRepresentativeException,
    EqualSetsException {

    if ( !this.isInTheDomain(rep1) || ! this.isInTheDomain(rep2) )
        throw new InvalidElementException(validRangeMsg);
    if ( !this.isRepresentative(rep1) )
        throw new NotRepresentativeException("First argument");
    if ( !this.isRepresentative(rep2) )
        throw new NotRepresentativeException("Second argument");
    if ( rep1 == rep2 )
        throw new EqualSetsException("The two arguments are equal");
```

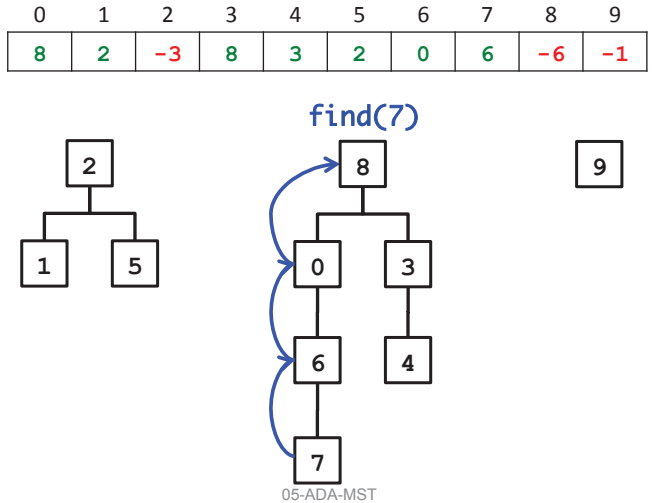

União por Altura

```
        if ( partition[rep1] <= partition[rep2] ) {
            // Height(S1) >= Height(S2)
            if ( partition[rep1] == partition[rep2] )
                partition[rep1]--;
            partition[rep2] = rep1;
        }
        else
            // Height(S1) < Height(S2)
            partition[rep1] = rep2;
    } //Closes union method
} //Closes class UnionFindInArray
```

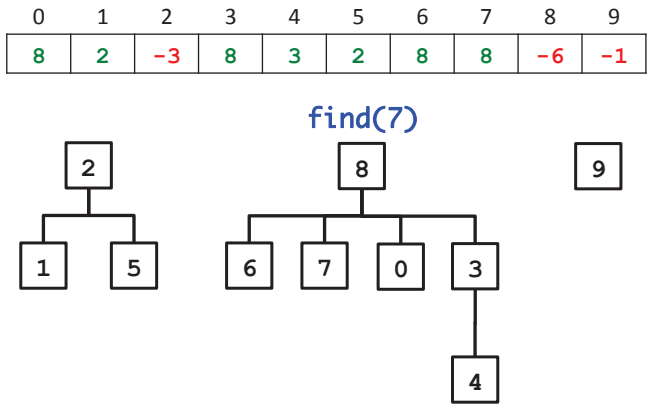
Compressão do Caminho



Compressão do Caminho



Compressão do Caminho



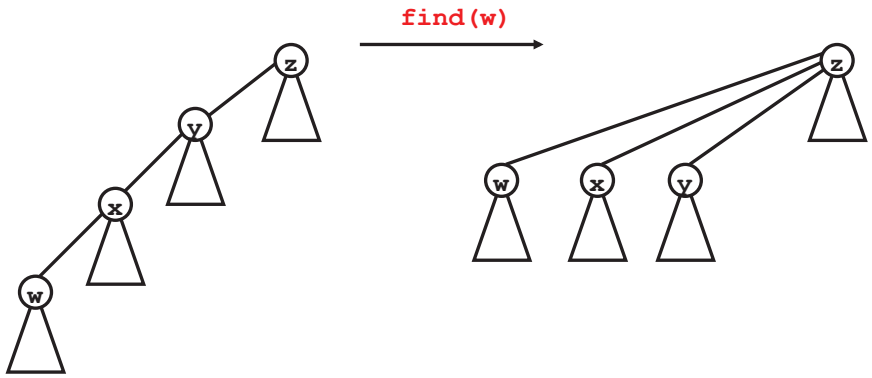
Representante com Compressão do Caminho

```
public int find( int element ) throws InvalidElementException {
     if ( !this.isInTheDomain(element) )
         throw new InvalidElementException(validRangeMsg);
     return this.findPathCompr(element);
}

// Pre-condition: 0 <= element < partition.length
protected int findPathCompr ( int element ) {
     if ( partition[element] < 0 )
         return element;
     else {
         partition[element] =this.findPathCompr(partition[element]);
         return partition[element];
}
```

Complexidade: Algoritmo Kruskal

Criar fila de prioridade	O(#A)
Criar partição	O(#V)
Criar lista	O(1)
Ciclo (executado entre #V – 1 e #A vezes)	
Instruções executadas entre #V – 1 e #A vezes	
Remover mínimo	O(log #A)
Seleccionar 2 representantes **(Representante com Compressão do Caminho)**	O(log #V)
Instruções executadas #V-1 vezes	
Adicionar vertíce na lista	O(1)
Unir representantes	O(1)
Total (união por nível ou dimensão)	O(#A log #V)

Como Determinar a Complexidade?



Complexidade (Pior Caso)

Representante com Compressão do Caminho

- Execução de m operações sobre n elementos [Tarjan 75]:

$O(m\ \alpha(n))$

$$\alpha(n) = \{ k \mid A_k(1) \geq n \}$$

$$A_k(j) = \begin{cases} j+1 & se \quad k = 0 \\ A_{k-1}^{(j+1)}(j) & se \quad k \geq 1 \end{cases}$$

[Tarjan 75] Robert Endre Tarjan. Efficiency of a Good But Not Linear Set Union Algorithm. J. ACM 22, 2 (April 1975), 215-225.

Complexidade (Pior Caso)

Representante com Compressão do Caminho

- Execução de m operações sobre n elementos:

$O(m\ \alpha(n))$

$$\alpha(n) = \{ k \mid A_k(1) \geq n \}$$

$$A_k(j) = \begin{cases} j+1 & se \quad k = 0 \\ A_{k-1}^{(j+1)}(j) & se \quad k \geq 1 \end{cases}$$

$$A_1(j) = 2j+1 \qquad A_2(j) = 2^{j+1}(j+1) - 1$$

Complexidade (Pior Caso)

Representante com Compressão do Caminho

$$\alpha(n) = \{ k \mid A_k(1) \geq n \} \qquad A_k(j) = \begin{cases} j+1 & se \quad k = 0 \\ A_{k-1}^{(j+1)}(j) & se \quad k \geq 1 \end{cases}$$

$$A_1(j) = 2j + 1 \qquad A_2(j) = 2^{j+1}(j+1) - 1$$

$A_1(1) = A_0^{(2)}(1) = A_0(A_0(1)) = A_0(2) = 3$

$A_1(3) = A_0^{(2)}(3) = A_0(A_0(3)) = A_0(4) = 5$

$A_2(1) = A_1^{(3)}(1) = A_1(A_1(A_1(1))) = A_1(5) = 7$

$A_3(1) = A_2^{(2)}(1) = A_2(A_2(1)) = A_2(7) = 2^8 \times 8 - 1 = 2047$

$A_4(1) = A_3^{(2)}(1) = A_3(A_3(1)) = A_3(2047) = A_2^{(2048)}(2047) >> A_2(2047) >> 10^{80}$

Complexidade (Pior Caso)

Representante com Compressão do Caminho

$$\alpha(n) = \begin{cases} 0 & 0 \leq n \leq 2 \\ 1 & n = 3 \\ 2 & 4 \leq n \leq 7 \\ 3 & 8 \leq n \leq 2047 \\ 4 & 2048 \leq n \leq A_4(1) \end{cases}$$

Execução de m operações sobre n elementos: O(m α(n))

Complexidade (Pior Caso)

- Conjunto de dimensão **n**:
  - **U** operações de união
  - **R** operações de representante

	Complexidade	Complexidade Execução de U+R operações
União sem estratégia	**O(1)**	**O(U + R n)**
Representante sem efeitos laterais	**O(n)**	
União por dimensão ou por altura	**O(1)**	**O(U + R log n)**
Representante sem efeitos laterais	**O(log n)**	
União por dimensão ou por nível	**O(1)**	**O((U+R)α(n))**
Representante com Compressão do Caminho	**O(log n)**	

7. Complexidade Amortizada

Planeamento

Apresentação da disciplina
Programação Dinâmica
Introdução aos Grafos
Percurso em largura e profundidade
Ordenação Topológica
Árvore Mínima de Cobertura
TAD Partição
Complexidade Amortizada
Algoritmo de Prim
TAD Fila com Prioridade Adaptável
Algoritmo de Dijskstra
Filas Binomiais
Filas de Fibonacci
Algoritmo de Bellman-Ford
Fluxo Máximo: Algoritmo de Ford-Fulkerson
Algoritmo de Edmonds-Karp
Introdução à Teoria da Complexidade
Exemplos de Problemas NP
Redutibilidade entre Problemas de Decisão

Análise da Complexidade

- Complexidade no pior caso
  - Analisar o tempo de execução em função dos "piores" dados de entrada
- Complexidade no caso esperado
  - Analisa o tempo de execução médio em função das distribuições dos dados de entrada
- Complexidade amortizada
  - Determina a complexidade do pior caso de uma sequência de operações

Complexidade Amortizada

- Analisar no pior caso uma sequência de operações numa ED
  - Não envolve probabilidades
- Mostra que embora algumas operações possam ter um custo individual elevado, o custo total da sequência de operações pode ser baixo

Complexidade Amortizada

- Exemplo: TAD Partição
  - Sequência de U operações de união e R operações de representante numa partição com n elementos*
    - $O((U+R)\alpha(n))$
  - Embora uma das operações tenha um custo individual elevado, o custo médio de cada operação é $\alpha(n)$

* Considerando a operação de união por altura ou nível e a operação de representante com compressão de caminho

Pilha com MultiDesempilha

```
void push( E element );

E pop( );

void multiPop( int k ) {

   while ( !this.isEmpty() && k > 0 ) {
        E element = this.pop();
        k--;
   }
}
```


Pilha com MultiDesempilha

```
void push( E element );    // Pior caso: O(1)

E pop( );                  // Pior caso: O(1)

void multiPop( int k ) {   // Pior caso:
                           // s é o número de elementos na pilha
   while ( !this.isEmpty() && k > 0 ) {
        E element = this.pop();
        k--;
   }
}
```


Pilha com MultiDesempilha

```
void push( E element );    // Pior caso: O(1)

E pop( );                  // Pior caso: O(1)

void multiPop( int k ) {   // Pior caso: O(min(k,s))
                           // s é o número de elementos na pilha
   while ( !this.isEmpty() && k > 0 ) {
        E element = this.pop();
        k--;
   }
}
```


Pilha com MultiDesempilha

```
void push( E element );    // Pior caso: O(1)

E pop( );                  // Pior caso: O(1)

void multiPop( int k ) {   // Pior caso: O(min(k,s))
                           // s é o número de elementos na pilha
   while ( !this.isEmpty() && k > 0 ) {
        E element = this.pop();
        k--;
   }
}
```

- Custo, no pior caso, de uma sequência de n operações (`push`, `pop` e `multiPop`), numa pilha inicialmente vazia:

Pilha com MultiDesempilha

```
void push( E element );    // Pior caso: O(1)

E pop( );                  // Pior caso: O(1)

void multiPop( int k ) {   // Pior caso: O(min(k,s))
                           // s é o número de elementos na pilha
   while ( !this.isEmpty() && k > 0 ) {
        E element = this.pop();
        k--;
   }
}
```

- Custo, no pior caso, de uma sequência de n operações (`push`, `pop` e `multiPop`), numa pilha inicialmente vazia: $O(n^2)$

Métodos

- Agregação
  - O tempo total necessário para a execução das n operações é calculado e dividido por n
- Contabilidade
  - A cada operação é atribuído um custo amortizado que pode ser inferior, igual ou superior ao custo real
    - Ao executar operações de custo amortizado superior ao custo real é associado crédito a um objecto da ED
- Potencial
  - A cada operação é atribuído um custo amortizado que pode ser inferior, igual ou superior ao custo real
    - Em vez de associar crédito aos objectos, associa-se um valor à ED, chamado potencial.

Método Agregação

- Se o tempo de execução de n operações é T(n)
- Custo amortizado de uma operação é T(n)/n

- É um método pouco versátil, com resultados menos precisos
  - Todas as operações têm a mesma complexidade amortizada

Agregação - Pilha

- Só é possível fazer `pop` de um objecto uma única vez por cada `push` desse objecto na pilha
- O número de operações `pop` não pode exceder número de operações de `push`
- No máximo ocorreram n operações de `push`

Agregação - Pilha

- Só é possível fazer `pop` de um objecto uma única vez por cada `push` desse objecto na pilha
- O número de operações `pop` não pode exceder número de operações de `push`
- No máximo ocorreram n operações de `push`

- Uma sequência de n operações `push`, `pop`, e `multiPop` tem um tempo de execução O(n)

- O custo amortizado é O(n)/n = O(1)
- Todas as operações têm um custo amortizado O(1)

Método da Contabilidade

- É atribuído um custo amortizado ĉ a cada operação, que pode ser superior, igual ou inferior ao custo real c
- Se o custo amortizado ĉ for superior ao custo real c, o crédito (ĉ – c) é associado a um objecto na ED
- Posteriormente, o crédito é usado para "pagar" operações cujo custo amortizado é menor que o custo real

Método da Contabilidade

- É atribuído um custo amortizado ĉ a cada operação, que pode ser superior, igual ou inferior ao custo real c
- Se o custo amortizado ĉ for superior ao custo real c, o crédito (ĉ – c) é associado a um objecto na ED
- Posteriormente, o crédito é usado para "pagar" operações cujo custo amortizado é menor que o custo real
- O custo total amortizado nunca é inferior ao custo total real. Ou seja, o crédito acumulado nunca é negativo:

para todo n $$\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \geq 0$$

Contabilidade - Pilha

Operação	Custo Real c_i	Custo Amortizado $\hat{c}_i$	Complexidade Amortizada
`push`	1	2	O(1)
`pop`	1	0	O(1)
`multiPop k`	`min(k,s)`	0	O(1)

(s é o número de elementos da pilha)

- Cada objecto empilhado tem 1 crédito
  - Por cada `push` é feito um pagamento adiantado da saída da pilha (por `pop` ou `multiPop`)

Contabilidade - Pilha

Operação	Estado da Pilha	Crédito Acumulado
	<>	0
push e1	<e1>	1
push e2	<e1,e2>	2
multiPop 3	<>	0
push e3	<e3>	1
push e4	<e3,e4>	2
pop	<e3>	1
multiPop 1	<>	0

Contabilidade - Pilha

Operação	Custo Real c_i	Custo Amortizado $\hat{c}_i$	Complexidade Amortizada
push	1	2	O(1)
pop	1	0	O(1)
multiPop k	min(k,s)	0	O(1)

(s é o número de elementos da pilha)

- O crédito total acumulado nunca é negativo: é o número de elementos na pilha.
- Uma sequência de n operações `push`, `pop`, e `multiPop` tem tempo de execução O(n)
- Todas as operações têm um custo amortizado O(1)

Método do Potencial

- É atribuído um custo amortizado ĉ a cada operação, que pode ser superior, igual ou inferior ao custo real c
- Em vez de atribuir crédito aos objectos, associa-se um valor à ED, chamado potencial.
- Define-se uma função potencial ϕ, que atribui a cada ED um número real $\phi(D)$.

Método do Potencial

- Considerar para cada i = 1, ..., n:
  - D_0 representa a ED inicial
  - D_i representa a ED após a operação i ($D_0, D_1, ..., D_{i-1}, D_i$)
  - c_i representa o custo real da operação i
- Custo amortizado da operação i:

$$\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1})$$

- Custo amortizado total:

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \phi(D_i) - \phi(D_{i-1}))$$

$$= (\sum_{i=1}^{n} c_i) + \phi(D_n) - \phi(D_0)$$

Método do Potencial

- O custo total amortizado nunca é inferior ao custo total real. Como o custo amortizado total:

$$(\sum_{i=1}^{n} c_i) + \phi(D_n) - \phi(D_0)$$

- basta garantir que, para todo i = 1, ..., n:

$$\phi(D_i) \geq \phi(D_0)$$

- A função potencial ϕ é definada de forma que:
  1. $\phi(D_0) = 0$
  2. $\phi(D_i) \geq 0$ para qualquer i

Potencial - Pilha

- Definir a função potencial como sendo o número de elementos s_p na pilha P:

$$\phi(P) = s_p$$

P1. $\phi(P_0) = 0$ onde P_0 é a pilha vazia

P2. $\phi(P) \geq 0$

- Custo amortizado da operação i:

$$\hat{c}_i = c_i + \phi(P_i) - \phi(P_{i-1})$$

Potencial - Pilha

- Seja P uma pilha qualquer e s_P o número de elementos em P

$$\phi(P) = s_p$$

Operação	Custo Real c_i	Dif. de Potencial $\Delta_i = \phi(P_i) - \phi(P_{i-1})$	Custo Amortizado $\hat{c}_i = c_i + \Delta_i$	Complexidade Amortizada
`push`	1			
`pop`	1			
`multiPop k`	`min(`s_p`,k)`			

Potencial - Pilha

- Seja P uma pilha qualquer e s_P o número de elementos em P

$$\phi(P) = s_p$$

Operação	Custo Real c_i	Dif. de Potencial $\Delta_i = \phi(P_i) - \phi(P_{i-1})$	Custo Amortizado $\hat{c}_i = c_i + \Delta_i$	Complexidade Amortizada
`push`	1	1	2	O(1)
`pop`	1	-1	0	O(1)
`multiPop k`	`min(`s_p`,k)`	`-min(`s_p`,k)`	0	O(1)

Potencial - Pilha

- Seja P uma pilha qualquer e s_P o número de elementos em P

$$\phi(P) = s_p$$

Operação	Custo Real c_i	Dif. de Potencial $\Delta_i = \phi(P_i) - \phi(P_{i-1})$	Custo Amortizado $\hat{c}_i = c_i + \Delta_i$	Complexidade Amortizada
`push`	1	1	2	O(1)
`pop`	1	-1	0	O(1)
`multiPop k`	`min(`s_p`,k)`	`-min(`s_p`,k)`	0	O(1)

- Uma sequência de n operações `push`, `pop`, e `multiPop` tem tempo de execução O(n)
- Todas as operações têm um custo amortizado O(1)

Contador Binário

$$x = \sum_{i=0}^{c-1} counter[i] \cdot 2^i$$

Ctr	counter[4]	counter[3]	counter[2]	counter[1]	counter[0]
0	0	0	0	0	0
1	0	0	0	0	1
2	0	0	0	1	0
3	0	0	0	1	1
4	0	0	1	0	0
5	0	0	1	0	1
6	0	0	1	1	0
7	0	0	1	1	1
8	0	1	0	0	0

Contador Binário

```
void increment( int[] counter ) {     // Pior caso:
   int pos = 0;                       // c é a capacidade do vector
   while ( pos < counter.length && counter[pos] == 1 ) {
        counter[pos] = 0;             // reset bit: Atribuir ZERO
        pos++;
   }
   if ( pos < counter.length )
        counter[pos] = 1;             // set bit: Atribuir UM
}
```


Contador Binário

```
void increment( int[] counter ) {     // Pior caso: O(c)
   int pos = 0;                       // c é a capacidade do vector
   while ( pos < counter.length && counter[pos] == 1 ) {
        counter[pos] = 0;             // reset bit: Atribuir ZERO
        pos++;
   }
   if ( pos < counter.length )
        counter[pos] = 1;             // set bit: Atribuir UM
}
```

- Custo, no pior caso, de uma sequência de n operações (`increment`), num contador inicialmente a zero:

Contador Binário

```
void increment( int[] counter ) {     // Pior caso: O(c)
   int pos = 0;                       // c é a capacidade do vector
   while ( pos < counter.length && counter[pos] == 1 ) {
        counter[pos] = 0;             // reset bit: Atribuir ZERO
        pos++;
   }
   if ( pos < counter.length )
        counter[pos] = 1;             // set bit: Atribuir UM
}
```

- Custo, no pior caso, de uma sequência de n operações (`increment`), num contador inicialmente a zero: $O(nc)$

Contabilidade - Contador

Operação	Custo Real c_i	Custo Amortizado $\hat{c}_i$	Complexidade Amortizada
`Atribuir ZERO`	1	0	O(1)
`Atribuir UM`	1	2	O(1)
`increment`	k	2	O(1)

(k é o número de atribuições efectuadas)

- Cada atribuição de UM tem 1 crédito
  - Por cada atribuição de UM é feito um pagamento adiantado da sua colocação a zero
- Cada operação `increment` tem custo amortizado 2, porque atribuí um único UM

Contabilidade - Contador

Operação	Estado do Contador	Crédito Acumulado
	00000	0
increment	00001	1
increment	00010	1
increment	00011	2
increment	00100	1
increment	00101	2
increment	00110	2
increment	00111	3
increment	01000	1
increment	01001	2

Contabilidade - Contador

Operação	Custo Real c_i	Custo Amortizado $\hat{c}_i$	Complexidade Amortizada
`Atribuir ZERO`	1	0	O(1)
`Atribuir UM`	1	2	O(1)
`increment`	k	2	O(1)

(k é o número de atribuições efectuadas)

- O crédito total acumulado nunca é negativo: é o número de UNS no vector.
- Uma sequência de n operações `increment` tem tempo de execução O(n)
- A operação `increment` tem um custo amortizado O(1)

Contador Binário

```
void increment( int[] counter ) {     // Pior caso: O(c)
   int pos = 0;                       // c é a capacidade do vector
   while ( pos < counter.length && counter[pos] == 1 ) {
        counter[pos] = 0;
        pos++;
   }
   if ( pos < counter.length )
        counter[pos] = 1;
}
```

- Custo, no pior caso, de uma sequência de n operações (`increment`), num contador inicialmente a zero: O(nc)

Potencial - Contador

- Definir a função potencial como sendo o número de UNS 1_C no contador C:

$$\phi(C) = 1_C$$

P1. $\phi(C_0) = 0$ onde C_0 é um contador só com zeros

P2. $\phi(C) \geq 0$

Potencial - Contador

- Seja C um contador qualquer e 1_C o número de UNS em C

$$\phi(C) = 1_C$$

ϕ(C)		counter[4]	counter[3]	counter[2]	counter[1]	counter[0]
$\phi(C_0) = 1_{C_0} = 0$	C_0	0	0	0	0	0
$\phi(C_1) = 1_{C_1} = 1$	C_1	0	0	0	0	1
$\phi(C_2) = 1_{C_2} = 1$	C_2	0	0	0	1	0
$\phi(C_3) = 1_{C_3} = 2$	C_3	0	0	0	1	1
$\phi(C_4) = 1_{C_4} = 1$	C_4	0	0	1	0	0
$\phi(C_5) = 1_{C_5} = 2$	C_5	0	0	1	0	1
$\phi(C_6) = 1_{C_6} = 2$	C_6	0	0	1	1	0
$\phi(C_7) = 1_{C_7} = 3$	C_7	0	0	1	1	1
$\phi(C_8) = 1_{C_8} = 1$	C_8	0	1	0	0	0
...	...	...	...	...	...	...

Potencial - Contador

- Seja C um contador qualquer e 1_C o número de UNS em C

$$\phi(C) = 1_C$$

Operação	Custo Real c_i	Dif. de Potencial $\Delta_i = \phi(C_i) - \phi(C_{i-1})$	Custo Amortizado $\hat{c}_i = c_i + \Delta_i$	Complexidade Amortizada
`increment`	k+1			

(k é o número de UNS que passam a ZERO)

	counter[4]	counter[3]	counter[2]	counter[1]	counter[0]	k	Custo Real
...	...	...	...	...	...	...	...
C_7	0	0	1	1	1	3	k+1 = 4
C_8	0	1	0	0	0	0	k+1 = 1
C_9	0	1	0	0	1	0	k+1 = 1
...	...	...	...	...	...	...	...

Potencial - Contador

- Seja C um contador qualquer e 1_C o número de UNS em C

$$\phi(C) = 1_C$$

Operação	Custo Real c_i	Dif. de Potencial $\Delta_i = \phi(C_i) - \phi(C_{i-1})$	Custo Amortizado $\hat{c}_i = c_i + \Delta_i$	Complexidade Amortizada
`increment`	`k+1`	`-k+1`	2	O(1)

(k é o número de UNS que passam a ZERO)

- Uma sequência de n operações `increment` tem tempo de execução O(n)
- A operação `increment` tem um custo amortizado O(1)

Tabela Dinâmica

```
// int currentSize
// E[] table (preenchida de 0 a currentSize – 1)

void insert( E element ) { // Pior caso:
                           // s é o número de elementos na tabela
   if ( table == null )
        table = new E[1];
   else if ( currentSize == table.length ) {
        E[] newTable = new E[ 2 * currentSize ];
        System.arraycopy(table, 0, newTable, 0, currentSize);
        table = newTable;
   }
   table[ currentSize++ ] = element;
}
```


Tabela Dinâmica

```
// int currentSize
// E[] table (preenchida de 0 a currentSize – 1)

void insert( E element ) { // Pior caso: O(s)
                           // s é o número de elementos na tabela
   if ( table == null )
        table = new E[1];
   else if ( currentSize == table.length ) {
        E[] newTable = new E[ 2 * currentSize ];
        System.arraycopy(table, 0, newTable, 0, currentSize);
        table = newTable;
   }
   table[ currentSize++ ] = element;
}
```

- Custo, no pior caso, de uma sequência de n operações (`insert`), numa tabela inicialmente vazia:

Tabela Dinâmica

```
// int currentSize
// E[] table (preenchida de 0 a currentSize – 1)

void insert( E element ) { // Pior caso: O(s)
                           // s é o número de elementos na tabela
   if ( table == null )
        table = new E[1];
   else if ( currentSize == table.length ) {
        E[] newTable = new E[ 2 * currentSize ];
        System.arraycopy(table, 0, newTable, 0, currentSize);
        table = newTable;
   }
   table[ currentSize++ ] = element;
}
```

- Custo, no pior caso, de uma sequência de n operações (`insert`), numa tabela inicialmente vazia: $O(n^2)$

Tabela Dinâmica: Exemplo

1. insert
2. insert

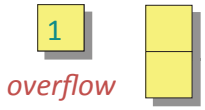


Tabela Dinâmica: Exemplo

1. insert
2. insert

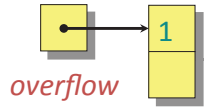


Tabela Dinâmica: Exemplo

1. insert
2. insert

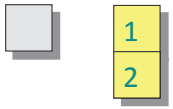


Tabela Dinâmica: Exemplo

1. insert
2. insert
3. insert

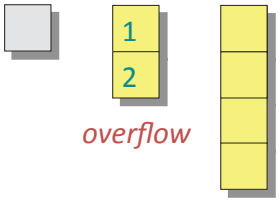


Tabela Dinâmica: Exemplo

1. insert
2. insert
3. insert

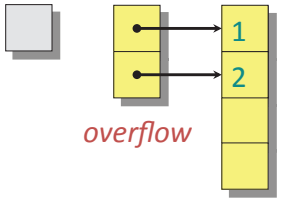


Tabela Dinâmica: Exemplo

1. insert
2. insert
3. insert

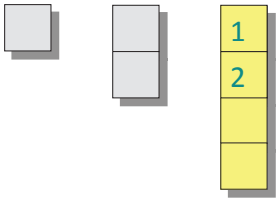


Tabela Dinâmica: Exemplo

1. insert
2. insert
3. insert
4. insert

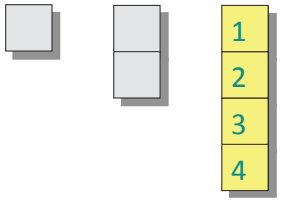


Tabela Dinâmica: Exemplo

1. insert
2. insert
3. insert
4. insert
5. insert

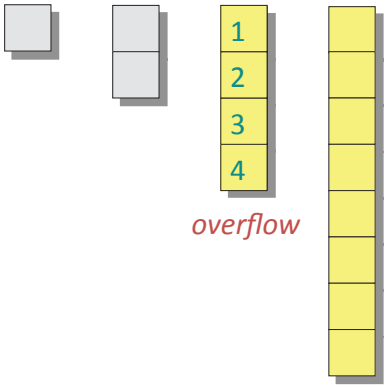


Tabela Dinâmica: Exemplo

1. insert
2. insert
3. insert
4. insert
5. insert

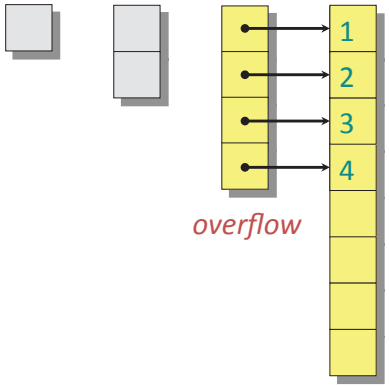


Tabela Dinâmica: Exemplo

1. insert
2. insert
3. insert
4. insert
5. insert

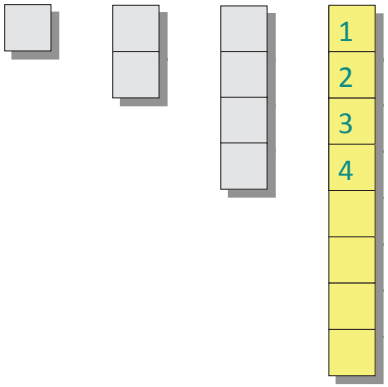
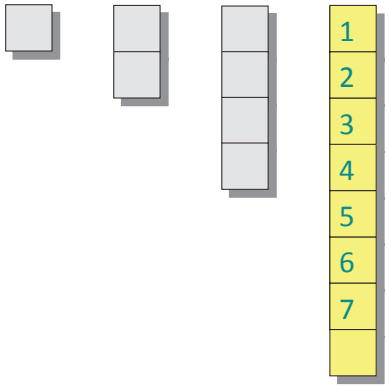


Tabela Dinâmica: Exemplo

1. insert
2. insert
3. insert
4. insert
5. insert
6. insert
7. insert



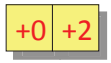
Contabilidade - Contador

Operação	Custo Real c_i	Custo Amortizado $\hat{c}_i$	Complexidade Amortizada
`Atribuir elemento novo`	1	3	O(1)
`insert`	k	3	O(1)

(k é o número de atribuições efectuadas)

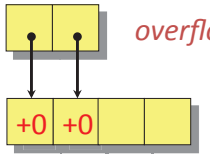
- Cada elemento na tabela tem 2 créditos
  - Os 2 créditos são para "pagar" a sua transferência para outra tabela e como pré-pagamento da transferência de outro elemento que já foi transferido alguma vez
- Cada operação de `insert` tem custo amortizado 3, porque atribui apenas um elemento novo.

Contabilidade - Contador



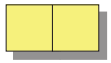
overflow

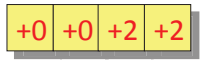
Contabilidade - Contador



overflow

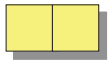
Contabilidade - Contador

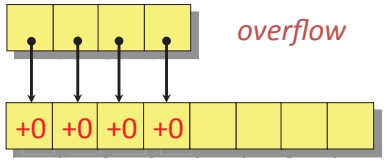




overflow

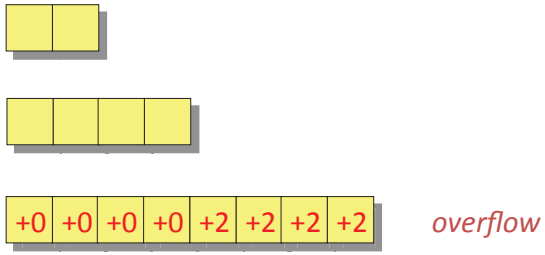
Contabilidade - Contador



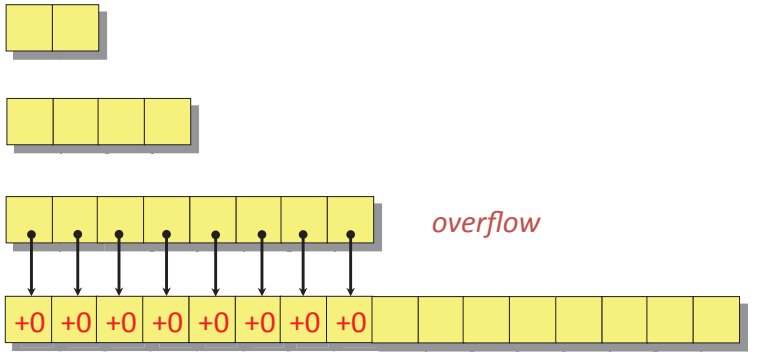


overflow

Contabilidade - Contador



Contabilidade - Contador



Contabilidade - Contador

Operação	Custo Real c_i	Custo Amortizado $\hat{c}_i$	Complexidade Amortizada
`Atribuir elemento novo`	1	3	O(1)
`insert`	k	3	O(1)

(k é o número de atribuições efectuadas)

- Uma sequência de n operações `insert` tem tempo de execução O(n)
- A operação `insert` tem um custo amortizado O(1)

Potencial - Tabela

- Seja T uma tabela qualquer, s_T o número de elementos em T, e c_T a capacidade de T.

$$\phi(T) = 2s_T - c_T$$

P1. $\phi(T_0) = 0$ onde T_0 é a tabela vazia com capacidade zero

P2. $\phi(T) \geq 0$ nos outros casos (factor de ocupação ≥ 50%)

$$\begin{aligned} c_T \geq s_T &> \tfrac{1}{2}c_T \\ 2s_T &> c_T \\ \phi(T) &> 0 \end{aligned}$$

Potencial - Tabela

- Seja T uma tabela qualquer, s_T o número de elementos em T, e c_T a capacidade de T.

$$\phi(T) = 2s_T - c_T$$

Operação `insert`	Custo Real c_i	Dif. de Potencial $\Delta_i = \phi(T_i) - \phi(T_{i-1})$	Custo Amortizado $\hat{c}_i = c_i + \Delta_i$	Complexidade Amortizada
`não expande`	1			
`expande`	s_{T_i}			

Potencial - Tabela

- Seja T uma tabela qualquer, s_T o número de elementos em T, e c_T a capacidade de T.

$$\phi(T) = 2s_T - c_T$$

Operação `insert`	Custo Real c_i	Dif. de Potencial $\Delta_i = \phi(T_i) - \phi(T_{i-1})$	Custo Amortizado $\hat{c}_i = c_i + \Delta_i$	Complexidade Amortizada
`não expande`	1	2	3	O(1)
`expande`	s_{T_i}	$3-s_{T_i}$	3	O(1)

Potencial - Tabela

- Seja T uma tabela qualquer, s_T o número de elementos em T, e c_T a capacidade de T.

$$\phi(T) = 2s_T - c_T$$

Operação `insert`	Custo Real c_i	Dif. de Potencial $\Delta_i = \phi(T_i) - \phi(T_{i-1})$	Custo Amortizado $\hat{c}_i = c_i + \Delta_i$	Complexidade Amortizada
`não expande`	1	2	3	O(1)
`expande`	s_{T_i}	$3-s_{T_i}$	3	O(1)

- Uma sequência de n operações `insert` tem tempo de execução O(n)
- A operação `insert` tem um custo amortizado O(1)

Exempo: ClearableTable

```
public class ClearableTable<E> {
   // Memory of the table: an array
   protected E[] table;
   // Number of elements in the table
   protected int currentSize;

   public ClearableTable( int capacity ) {
        table = (E[]) new Object[capacity];
        currentSize = 0;
   }
   public void addLast( E element ) {
    if ( currentSize < table.length )
            table[currentSize++] = element;
   }
   public void makeEmpty( ) {
    for ( int i = 0; i < currentSize; i++ )
            table[i] = null;
        currentSize = 0;
   }
}
```

Exempo: QueueInStack

```
public class QueueInStack{
   private Stack<Integer> A;
   private Stack<Integer> B;

   void enqueue( E element ) {
        A.push( E );
    }

   //pre: !( A.isEmpty() && B.isEmpty() )
   E dequeue( ) {
        if( B.isEmpty() ) {
            while ( !A.Empty() ) {
                E element = A.pop( );
                B.push( E );
            }
        return B.pop();
   }
```


8. Árvore Mínima de Cobertura Algoritmo de Prim

Planeamento

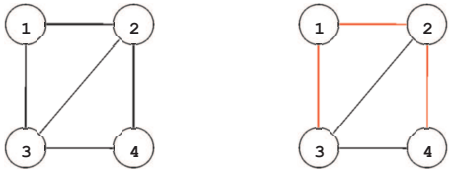
Apresentação da disciplina
Programação Dinâmica
Introdução aos Grafos
Percurso em largura e profundidade
Ordenação Topológica
Árvore Mínima de Cobertura (Algoritmo de Kruskal)
TAD Partição
Complexidade Amortizada
Árvore Mínima de Cobertura (Algoritmo de Prim)
TAD Fila com Prioridade Adaptável
Algoritmo de Dijskstra
Filas Binomiais
Filas de Fibonacci
Algoritmo de Bellman-Ford
Fluxo Máximo: Algoritmo de Ford-Fulkerson
Algoritmo de Edmonds-Karp
Introdução à Teoria da Complexidade
Exemplos de Problemas NP
Redutibilidade entre Problemas de Decisão

Árvore de Cobertura

- Um grafo G=(V,A) não orientado, diz-se **conexo** se
  - $\forall v,w \in V$ existe um caminho de v para w
- Um grafo G=(V,A) não orientado, diz-se um **árvore** se é acíclico e conexo
- Dado grafo não dirigido e conexo G = (V, A), uma **árvore abrangente** é sub-conjunto acíclico T $\subseteq$ A, que liga todos os vértices

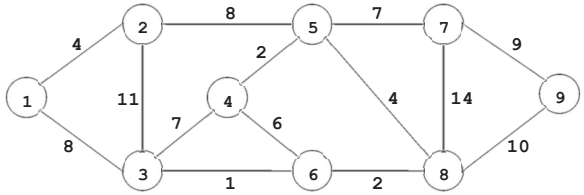
Árvore de Cobertura

- Um grafo G=(V,A) não orientado, diz-se **conexo** se
  - $\forall v,w \in V$ existe um caminho de v para w
- Um grafo G=(V,A) não orientado, diz-se um **árvore** se é acíclico e conexo
- Dado grafo não dirigido e conexo G = (V, A), uma **árvore abrangente** é sub-conjunto acíclico T ⊆ A, que liga todos os vértices



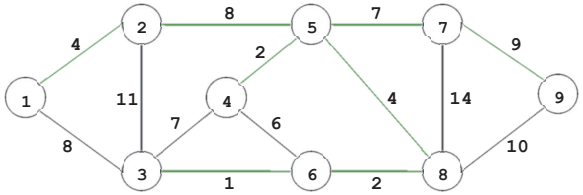
Árvore Mínima de Cobertura

- Árvore de cobertura de custo mínimo (nenhuma árvore de cobertura tem custo menor)

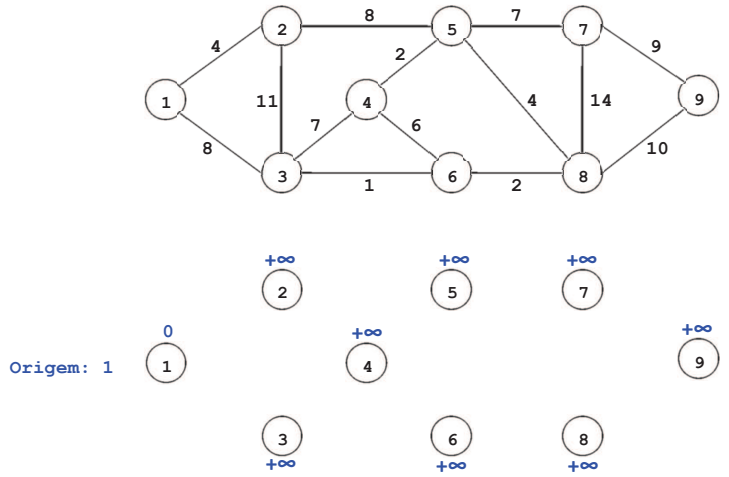


Árvore Mínima de Cobertura

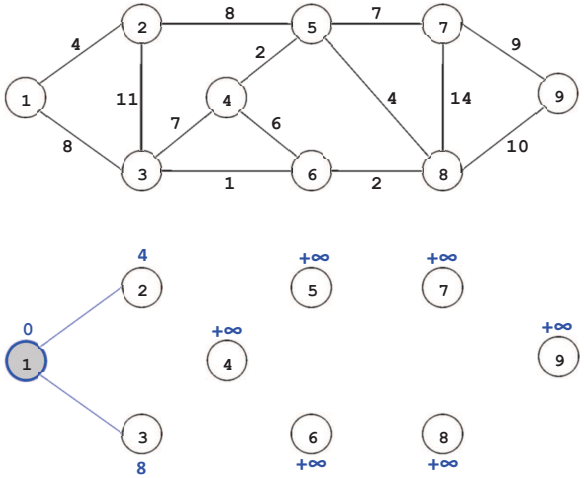
- Árvore de cobertura de custo mínimo (nenhuma árvore de cobertura tem custo menor).



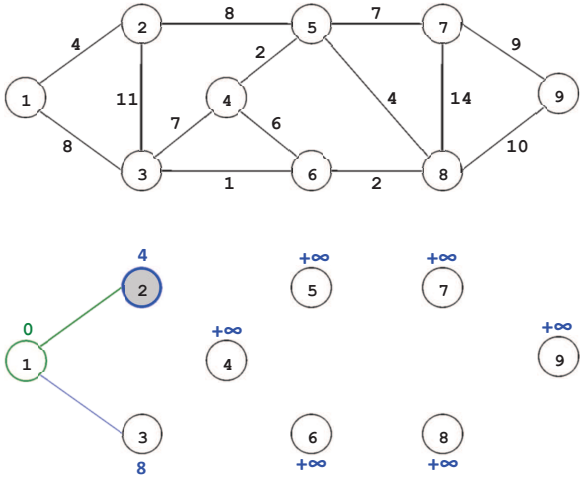
Algoritmo de Prim - Inicialização



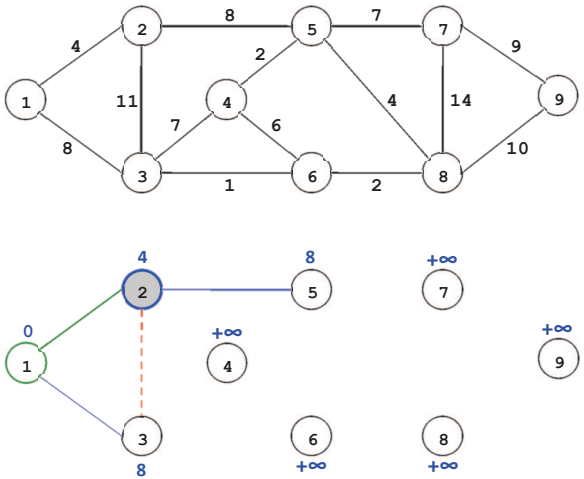
Algoritmo de Prim



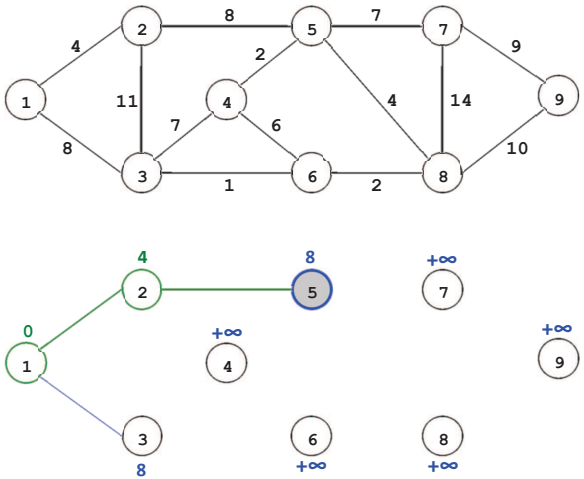
Algoritmo de Prim



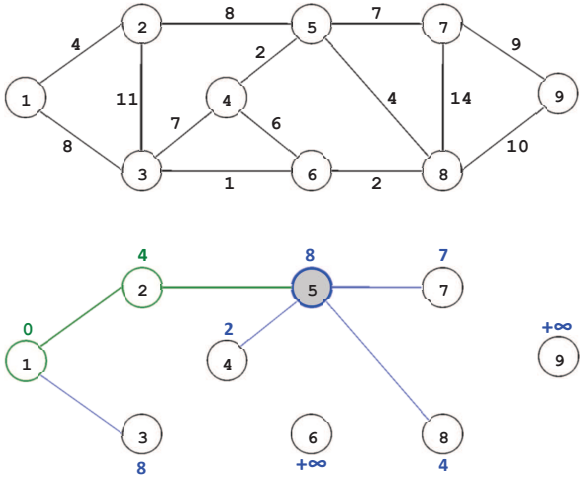
Algoritmo de Prim



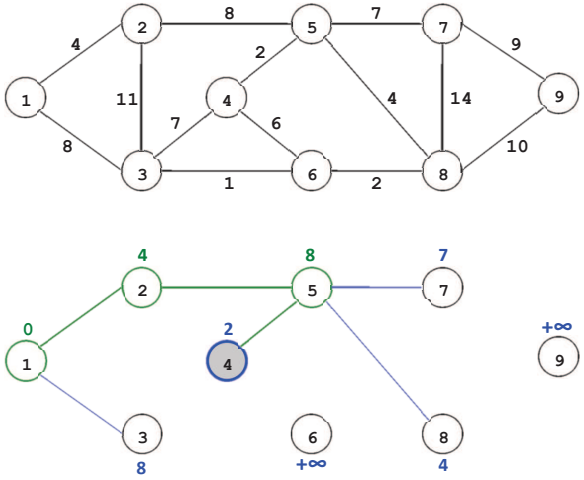
Algoritmo de Prim



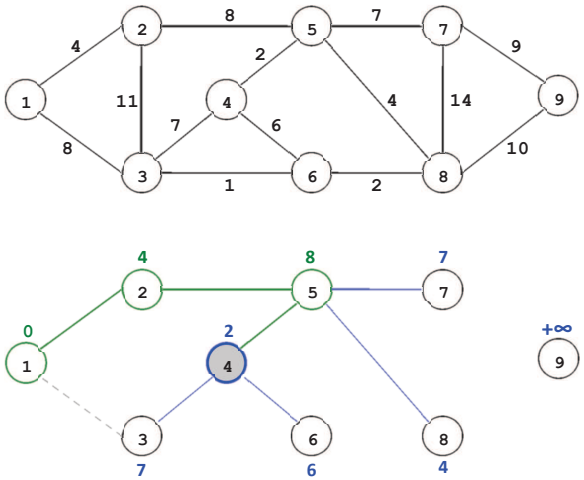
Algoritmo de Prim



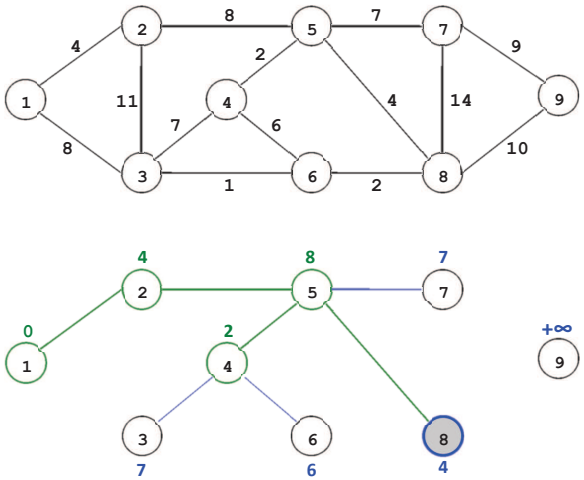
Algoritmo de Prim



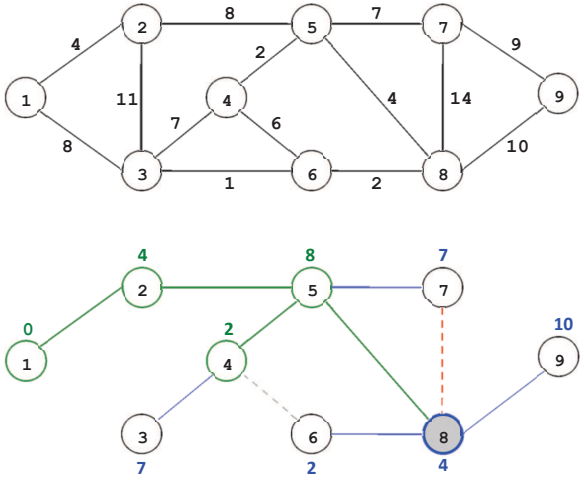
Algoritmo de Prim



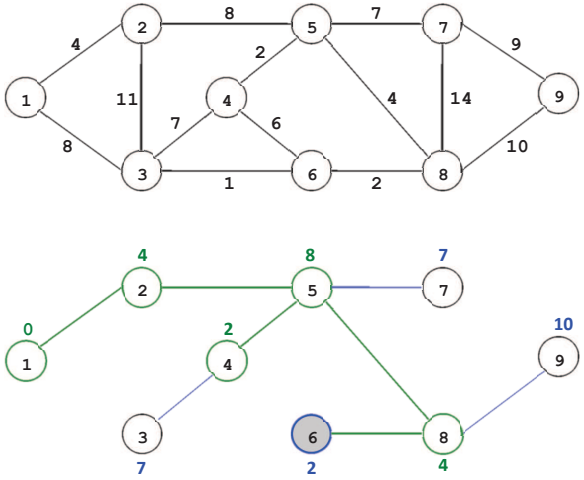
Algoritmo de Prim



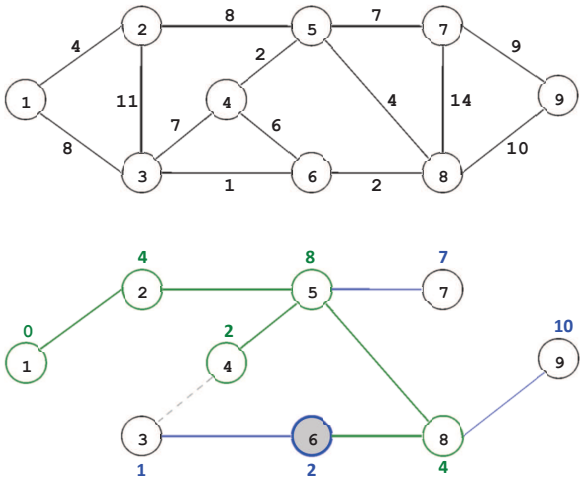
Algoritmo de Prim



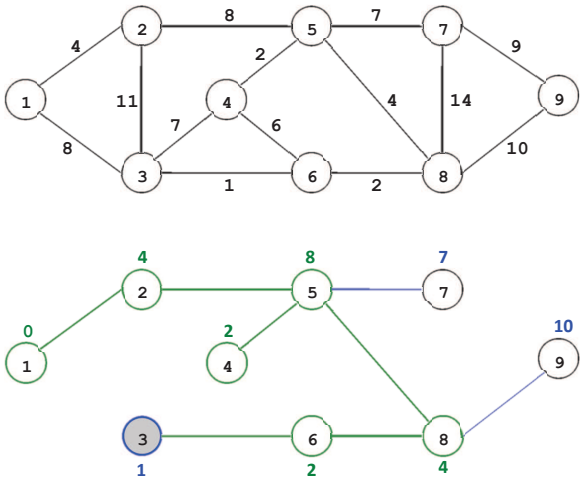
Algoritmo de Prim



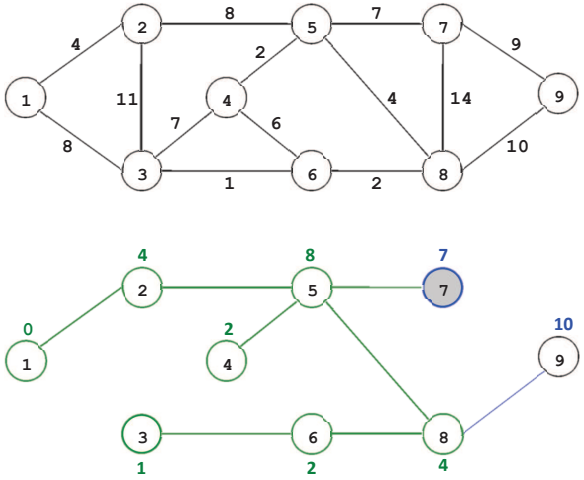
Algoritmo de Prim



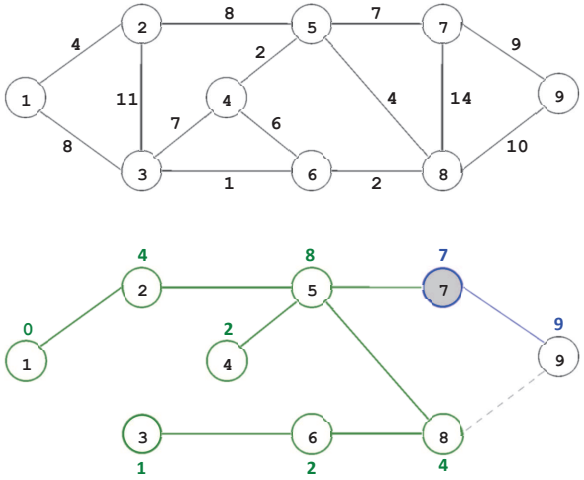
Algoritmo de Prim



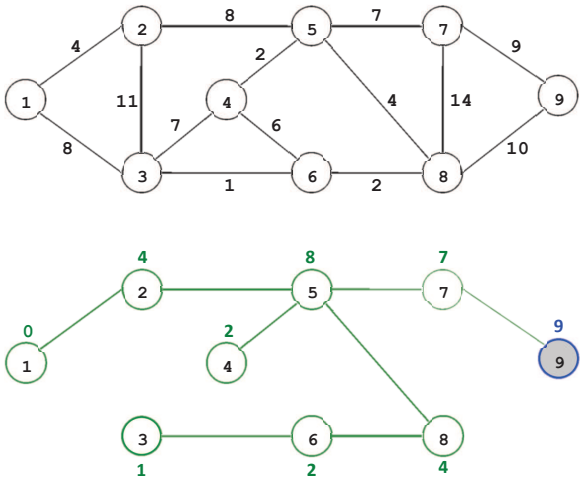
Algoritmo de Prim



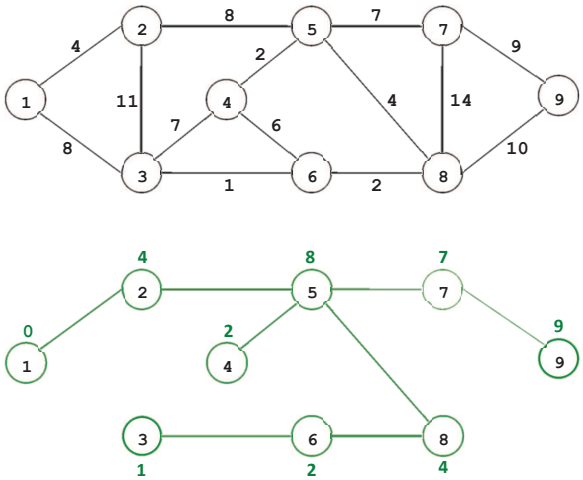
Algoritmo de Prim



Algoritmo de Prim



Algoritmo de Prim



Informação Necessária

```
// Indica o conjunto de vértices não seleccionados
// com caminho a partir do vértice o
AdaptMinPriorityQueue ligados

//Indica se v já foi seleccionado
boolean seleccionado[v]

// Peso do arco via[v] se há caminho de o para v, ou
// +∞ quando não caminho de o para v
ℜ₀⁺ ∪ {+∞}  custo[v]

// Arco de peso mínimo que liga v à árvore mínina, ou
// indefinido caso v não esteja ligado
Edge via[v]
```


Estratégia

- Situação inicial:
  - Global
    - `ligados = {o}`
  - Informação para o vértice o:
    - `seleccionado[o] = false`
    - `custo[o] = 0`
    - `via[o]` não está definida

Estratégia

- Situação inicial:
  - Global
    - `ligados = {o}`
  - Informação para o vértice o:
    - `seleccionado[o] = false`
    - `custo[o] = 0`
    - `via[o]` não está definida
  - Informação para os restantes vértices:
    - `seleccionado[v] = false`
    - `custo[v] = +∞`
    - `via[v]` não está definida

Estratégia

- Situação inicial:
  - Global
    - `ligados = {o}`
  - Informação para o vértice o:
    - `seleccionado[o] = false`
    - `custo[o] = 0`
    - `via[o]` não está definida
  - Informação para os restantes vértices:
    - `seleccionado[v] = false`
    - `custo[v] = +∞`
    - `via[v]` não está definida
- Cada iteração:
  - Escolher o vértice, nunca seleccionado, de custo mínimo

Árvore Mínima de Cobertura (Prim)

```
Iterator<Edge<?,E>> mstPrim( UndiGraph<?,E> graph ) {
   boolean[] selected = new boolean[ graph.numVertices() ];
   E[] cost = new E[ graph.numVertices() ];
   Edge<?,E>[] via = new Edge<?,E>[ graph.numVertices() ];
   AdaptMinPriorityQueue<E,Vertex> connected =
           new AdaptMinHeap<E,Vertex>( graph.numVertices() );
   List<Edge<?,E>> mst = new DoublyLinkedList<Edge<?,E>>();
```


Árvore Mínima de Cobertura (Prim)

```
   for every Vertex v in graph.vertices() {
        selected[v] = false;
        cost[v] = +∞;
   }
   Vertex origin = graph.aVertex();
   cost[origin] = 0;
   connected.insert(cost[origin], origin);
```


Árvore Mínima de Cobertura (Prim)

```
   do {
        Vertex vertex = connected.removeMin().getValue();
        selected[vertex] = true;
        if ( vertex != origin )
            mst.addLast( via[vertex] );
        exploreVertex(graph, vertex, selected, cost, via, connected);
   } while ( !connected.isEmpty() );
   return mst.iterator();
}
```


Árvore Mínima de Cobertura (Prim)

```
void exploreVertex( UndiGraph<?,E> graph, Vertex source,
    boolean[] selected, E[] cost, Edge<?,E>[] via,
    AdaptMinPriorityQueue<E,Vertex> connected ) {

        for every Edge<?,E> e in graph.incidentEdges(source) {
            Vertex vertex = e.oppositeVertex(source);
            if ( !selected[vertex] && e.label() < cost[vertex] ) {
                boolean vertexIsInQueue = cost[vertex] < +∞;
                cost[vertex] = e.label();
                via[vertex] = e;
                if ( vertexIsInQueue )
                    connected.decreaseKey(vertex, cost[vertex]);
                else
                    connected.insert(cost[vertex], vertex);
            }
        }
}
```

Complexidade (a completar...)

Criar fila	?
Criar lista ligada	O(1)
Inicializar 2 vectores	O(#V)
Inserir origem na fila	?
#V remover mínimo da fila	?
#V-1 inserir à cauda na lista	O(#V)
#V percorrer sucessores directos	$O((\#V)^2)$ ou O(#A)
#A inserir na fila ou decrementar chave	?

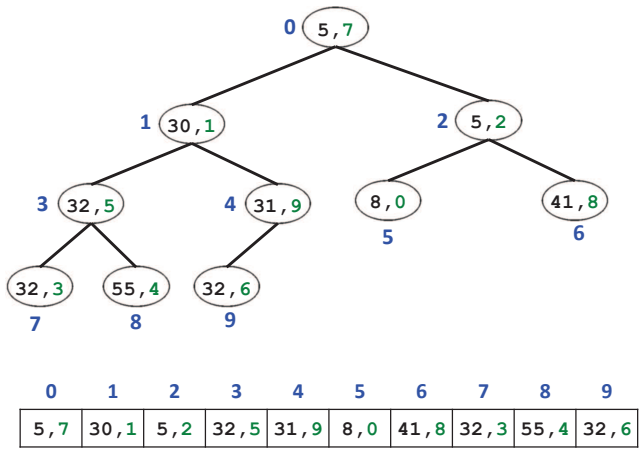
TAD Fila com Prioridade por Mínimos (K,V)

```
public interface MinPriorityQueue<K extends Comparable<K>, V> {
  // Returns true iff the priority queue contains no entries.
  boolean isEmpty( );

  // Returns the number of entries in the priority queue.
  int size( );

  // Returns an entry with the smallest key in the priority
  // queue.
  Entry<K,V> minEntry( ) throws EmptyPriorityQueueException;

  // Inserts the entry (key, value) in the priority queue.
  void insert( K key, V value );

  // Removes an entry with the smallest key from the priority
  // queue and returns that entry.
  Entry<K,V> removeMin( ) throws EmptyPriorityQueueException;
}
```

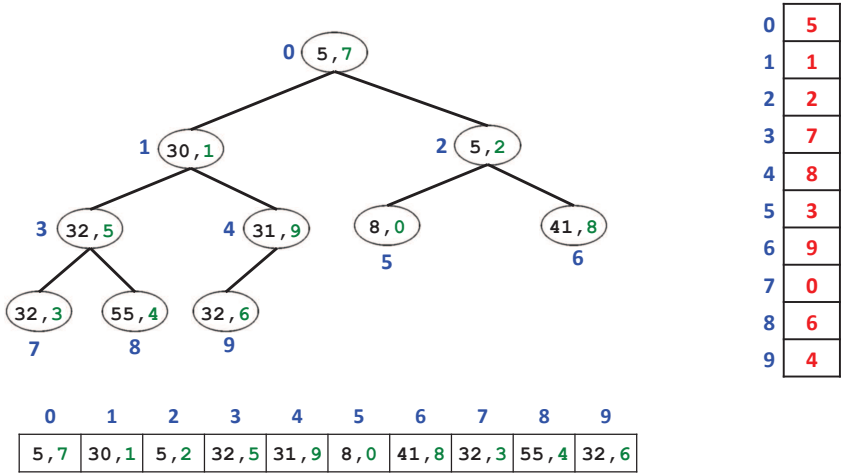

TAD Fila com Prioridade Adaptável Organizada por Mínimos (K,V)

```
public interface AdaptMinPriorityQueue<K extends Comparable<K>,
  V extends Comparable<V>> extends MinPriorityQueue<K,V> {

  // If the priority queue contains an entry with the specified
  // value, returns the associated key and replaces it by the
  // specified key. Otherwise, returns null.
  K decreaseKey( V value, K newKey ) throws InvalidKeyException;

  // If the priority queue contains an entry with the specified
  // value, removes and returns that entry.
  // Otherwise, returns null.
  Entry<K,V> remove( V value );
}
```


Implementação em Heap



Implementação em Heap e Vector



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

0	1	2	3	4	5	6	7	8	9
5,7	30,1	5,2	32,5	31,9	8,0	41,8	32,3	55,4	32,6

Descrição Informal das Operações

- `void insert( K key, V value )`
  - Dicionário: Pesquisa-se o valor, para descobrir se já existe
  - Heap: Coloca-se a nova entrada no fim do heap e executa-se o borbulhar ascendente a partir dessa posição (a última ocupada)
- `Entry<K,V> minEntry( ) throws EmptyPriorityQueueException`
  - Heap: Retorna-se a primeira entrada do heap (posição zero). Coloca-se a última entrada no início do heap e executa-se o borbulhar descendente a partir da posição zero

Descrição Informal das Operações

- `K decreaseKey( V value, K newKey ) throws InvalidKeyException`
  - Dicionário: Pesquisa-se o valor, para descobrir a posição da entrada no heap
  - Heap: Executa-se o borbulhar ascendente a partir dessa posição

Descrição Informal das Operações

- `Entry<K,V> remove( V value )`
  - Dicionário: Pesquisa-se o valor, para descobrir a posição POS da entrada no heap
  - Heap: Coloca-se a última entrada em POS e compara-se essa chave com a chave a ser removida:
    - Se a chave agora em POS é menor, executa-se o borbulhar ascendente a partir de POS
    - Se a chave agora em POS é maior, executa-se o borbulhar descendente a partir de POS
- Sempre que uma entrada é removida do heap ou muda de posição no heap, é necessário alterar o dicionário.

Complexidades da Fila com Prioridade Adaptável em Heap e Vector (n entradas)

	Melhor Caso	Pior Caso	Caso Esperado
`isEmpty`	O(1)	O(1)	O(1)
`size`	O(1)	O(1)	O(1)
`minEntry`	O(1)	O(1)	O(1)
`insert`	O(1)	O(log n)	O(log n)
`removeMin`	O(1)	O(log n)	O(log n)
`decreaseKey`	O(1)	O(log n)	O(log n)
`remove`	O(1)	O(log n)	O(log n)

Complexidades da Fila com Prioridade Adaptável em Heap e Tabela de Dispersão (n entradas)

	Melhor Caso	Pior Caso	Caso Esperado
`isEmpty`	O(1)	O(1)	O(1)
`size`	O(1)	O(1)	O(1)
`minEntry`	O(1)	O(1)	O(1)
`insert`	O(1)	O(n log n)	O(log n)
`removeMin`	O(1)	O(n log n)	O(log n)
`decreaseKey`	O(1)	O(n log n)	O(log n)
`remove`	O(1)	O(n log n)	O(log n)

Complexidade do Algoritmo de Prim Fila Implementada em Heap e Vector

Criar fila	O(#V)
Criar lista ligada	O(1)
Inicializar 2 vectores	O(#V)
Inserir origem na fila	O(1)
#V remover mínimo da fila	O(#V log #V)
#V-1 inserir à cauda na lista	O(#V)
#V percorrer sucessores directos	$O((\#V)^2)$ ou O(#A)
#A inserir na fila ou decrementar chave	O(#A log #V)
TOTAL (matriz de adjacências)	
TOTAL (listas de adjacências)	

Complexidade do Algoritmo de Prim Fila Implementada em Heap e Vector

Criar fila	O(#V)
Criar lista ligada	O(1)
Inicializar 2 vectores	O(#V)
Inserir origem na fila	O(1)
#V remover mínimo da fila	O(#V log #V)
#V-1 inserir à cauda na lista	O(#V)
#V percorrer sucessores directos	$O((\#V)^2)$ ou O(#A)
#A inserir na fila ou decrementar chave	O(#A log #V)
TOTAL (matriz de adjacências):	$O((\#V)^2 + \#A \log \#V)$
TOTAL (listas de adjacências):	O(#A log #V)

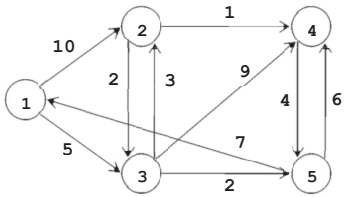
9. Caminho Mais Curto de um vértice a todos os outros Algoritmo de Dijkstra

Planeamento

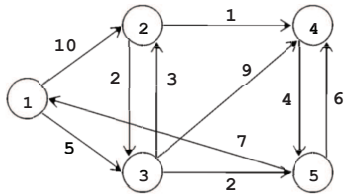
Apresentação da disciplina
Programação Dinâmica
Introdução aos Grafos
Percurso em largura e profundidade
Ordenação Topológica
Árvore Mínima de Cobertura (Algoritmo de Kruskal)
TAD Partição
Complexidade Amortizada
Árvore Mínima de Cobertura (Algoritmo de Prim)
TAD Fila com Prioridade Adaptável
Algoritmo de Dijskstra
Filas Binomiais
Filas de Fibonacci
Algoritmo de Bellman-Ford
Fluxo Máximo: Algoritmo de Ford-Fulkerson
Algoritmo de Edmonds-Karp
Introdução à Teoria da Complexidade
Exemplos de Problemas NP
Redutibilidade entre Problemas de Decisão

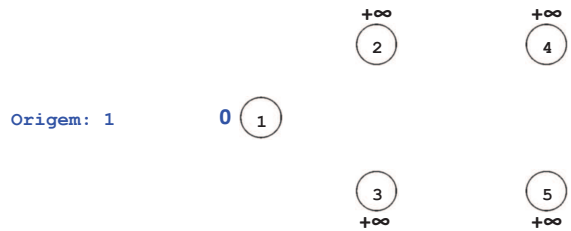
Algoritmo de Dijkstra

- Dado um grafo orientado e pesado e um vértice o, determina um caminho mais curto de o para qualquer vértice v
  - Este algoritmo só pode ser aplicado se os pesos dos arcos não forem negativos

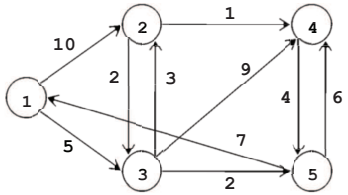


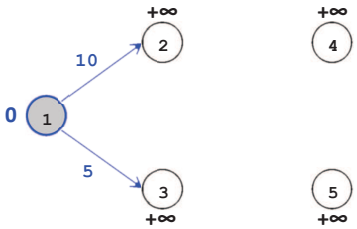
Algoritmo de Dijkstra - Inicialização



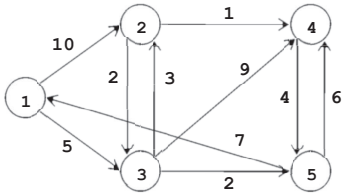


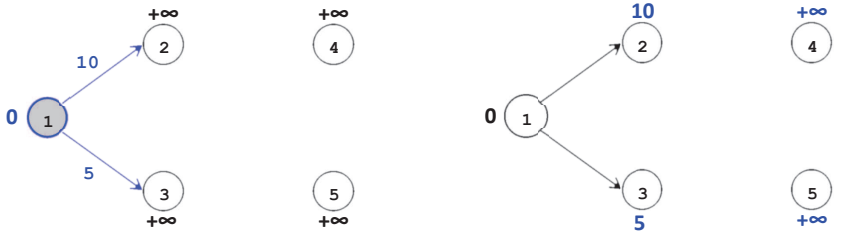
Algoritmo de Dijkstra



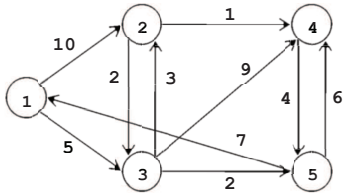


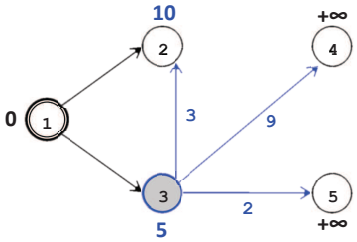
Algoritmo de Dijkstra



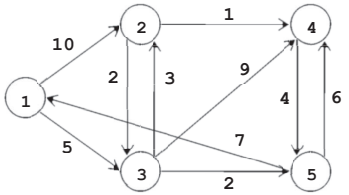


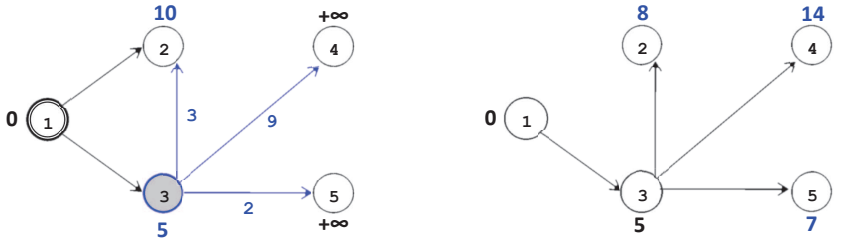
Algoritmo de Dijkstra



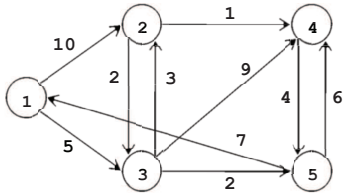


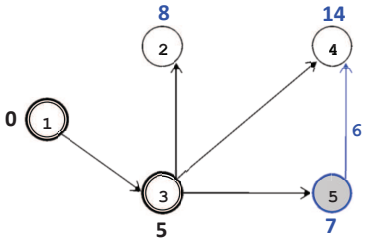
Algoritmo de Dijkstra



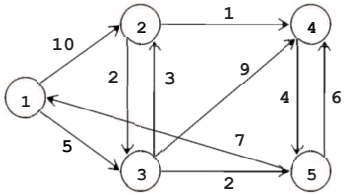


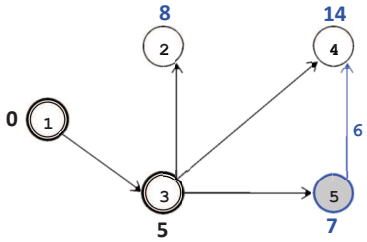
Algoritmo de Dijkstra

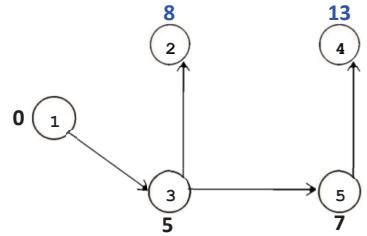




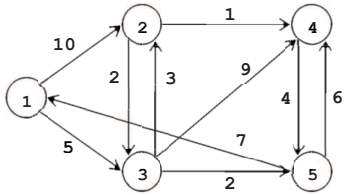
Algoritmo de Dijkstra

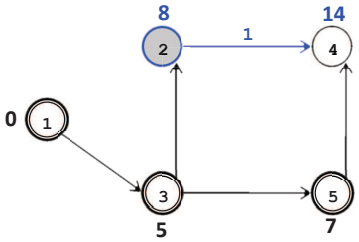




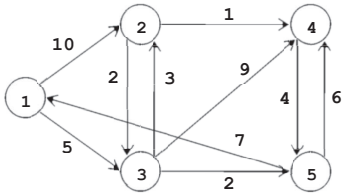


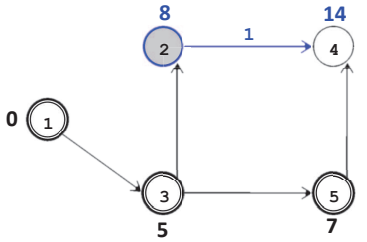
Algoritmo de Dijkstra

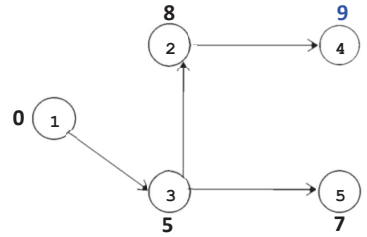




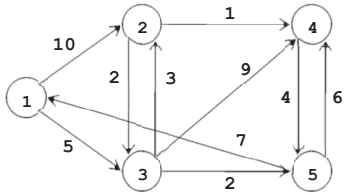
Algoritmo de Dijkstra

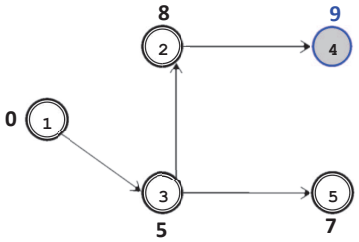




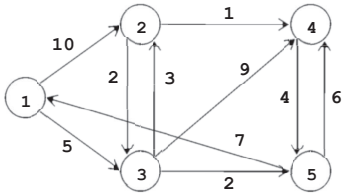


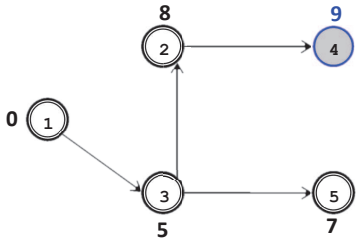
Algoritmo de Dijkstra

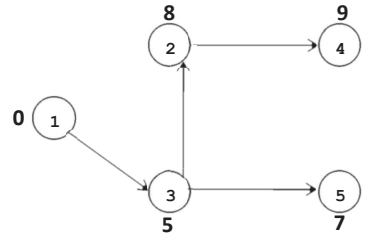




Algoritmo de Dijkstra







Informação Necessária

```
// Indica o conjunto de vértices não seleccionados
// para os quais há um caminho a partir do vértice o
AdaptMinPriorityQueue ligados

// Indica se v já foi seleccionado, i.e, já existe um caminho
// mais curto de o para v
boolean seleccionado[v]

// Comprimento do caminho mais curto (até ao momento) o para v,
// ou, +∞ quando ainda não há caminho de o para v
ℜ₀⁺ ∪ {+∞} comprimento[v]

// Se estiver definido, indica que um caminho mais curto de o
// para v tem a forma o, . . ., via[v], v.
Vertex via[v]
```


Estratégia

- Situação inicial:
  - Global
    - `ligados = {o}`
  - Informação para o vértice o:
    - `seleccionado[o] = false`
    - `comprimento[o] = 0`
    - `via[o] = o`

Estratégia

- Situação inicial:
  - Global
    - `ligados = {o}`
  - Informação para o vértice o:
    - `seleccionado[o] = false`
    - `comprimento[o] = 0`
    - `via[o] = o`
  - Informação para os restantes vértices:
    - `seleccionado[v] = false`
    - `comprimento[v] = +∞`
    - `via[v]` não está definida

Estratégia

- Situação inicial:
  - Global
    - `ligados = {o}`
  - Informação para o vértice o:
    - `seleccionado[o] = false`
    - `comprimento[o] = 0`
    - `via[o] = o`
  - Informação para os restantes vértices:
    - `seleccionado[v] = false`
    - `comprimento[v] = +∞`
    - `via[v]` não está definida
- Cada iteração:
  - Selecciona-se um vértice nunca seleccionado, v, tal que `comprimento[v]` é mínimo.

Caminho Mais Curto (Disjkstra)

```
Pair<E[], Vertex[]>  dijkstra( DiGraph<?,E> graph, Vertex origin ) {
   boolean[] selected = new boolean[ graph.numVertices() ];
   E[] length= new E[ graph.numVertices() ];
   Vertex[] via = new Vertex[ graph.numVertices() ];
   AdaptMinPriorityQueue<E,Vertex> connected =
            new AdaptMinHeap<E,Vertex>( graph.numVertices() );
```


Caminho Mais Curto (Disjkstra)

```
   for every Vertex v in graph.vertices() {
        selected[v] = false;
        length[v] = +∞;
   }
   length[origin] = 0;
   via[origin] = origin;
   connected.insert(length[origin], origin);
```

Caminho Mais Curto (Disjkstra)

```
    do {
        Vertex vertex = connected.removeMin().getValue();
        selected[vertex] = true;
        exploreVertex(graph, vertex, selected, cost, via, connected);
    } while ( !connected.isEmpty() );
    return PairClass<E[], Vertex> (length, via);
}
```


Caminho Mais Curto (Disjkstra)

```
void exploreVertex( DiGraph<?,E> graph, Vertex source,
    boolean[] selected, E[] length, Edge<?,E>[] via,
    AdaptMinPriorityQueue<E,Vertex> connected ) {

        for every Edge<?,E> e in graph.outIncidentEdges(source) {
            Vertex vertex = e.endVertices()[1];
            if ( !selected[vertex) {
                E newLength = length[source] + e.label();
                if (newLength < length[vertex] ) {
                // Actualizar o menor caminho de origin a vertex
                }
            }
        }
}
```


Caminho Mais Curto (Disjkstra)

```
// Corpo do método exploreVertex
for every Edge<?,E> e in graph.outIncidentEdges(source) {
    Vertex vertex = e.endVertices()[1];
    if ( !selected[vertex) {
        E newLength = length[source] + e.label();
        if (newLength < length[vertex] ) {
            boolean vertexIsInQueue = length[vertex] < +∞;
            length[vertex] = newLength;
            via[vertex] = source;
            if ( vertexIsInQueue )
                connected.decreaseKey(vertex, length[vertex]);
            else
                connected.insert(length[vertex], vertex);
        }
    }
}
```


Complexidade do Algoritmo de Dijkstra Fila Implementada em Heap e Vector

Criar fila	O(#V)
Inicializar 2 vectores	O(#V)
Inserir origem na fila	O(1)
#V remover mínimo da fila	O(#V log #V)
#V percorrer sucessores directos	$O((\#V)^2)$ ou O(#A)
#A inserir na fila ou decrementar chave	O(#A log #V)
TOTAL (matriz de adjacências)	
TOTAL (listas de adjacências)	

Complexidade do Algoritmo de Dijkstra Fila Implementada em Heap e Vector

Criar fila	O(#V)
Inicializar 2 vectores	O(#V)
Inserir origem na fila	O(1)
#V remover mínimo da fila	O(#V log #V)
#V percorrer sucessores directos	$O((\#V)^2)$ ou O(#A)
#A inserir na fila ou decrementar chave	O(#A log #V)
TOTAL (matriz de adjacências)	$O((\#V)^2 + \#A \log \#V)$
TOTAL (listas de adjacências)	O((#V + #A) log #V)

Escrita do Caminho

```
void writePath( Vertex via[], Vertex origin, Vertex destination ) {
   if (destination != origin) {
        writePath(via, origin, via[destination])
        WRITE(“, “);
   }
   WRITE(destination);
}
```


10. Filas Binomiais

Planeamento

Apresentação da disciplina
Programação Dinâmica
Introdução aos Grafos
Percurso em largura e profundidade
Ordenação Topológica
Árvore Mínima de Cobertura (Algoritmo de Kruskal)
TAD Partição
Complexidade Amortizada
Árvore Mínima de Cobertura (Algoritmo de Prim)
TAD Fila com Prioridade Adaptável
Algoritmo de Dijskstra
Filas Binomiais
Filas de Fibonacci
Algoritmo de Bellman-Ford
Fluxo Máximo: Algoritmo de Ford-Fulkerson
Algoritmo de Edmonds-Karp
Introdução à Teoria da Complexidade
Exemplos de Problemas NP
Redutibilidade entre Problemas de Decisão

TAD Fila com Prioridade por Mínimos (K,V)

```
public interface MinPriorityQueue<K extends Comparable<K>, V> {
   // Returns true iff the priority queue contains no entries
   boolean isEmpty( );

   // Returns the number of entries in the priority queue
   int size( );

   // Returns an entry with the smallest key in the priority queue
   Entry<K,V> minEntry( ) throws EmptyPriorityQueueException;

   // Inserts the entry (key, value) in the priority queue
   void insert( K key, V value );

   // Removes an entry with the smallest key from the priority
   // queue and returns that entry.
   Entry<K,V> removeMin( ) throws EmptyPriorityQueueException;
}
```


TAD Fila com Prioridade Adaptável Organizada por Mínimos (K,V)

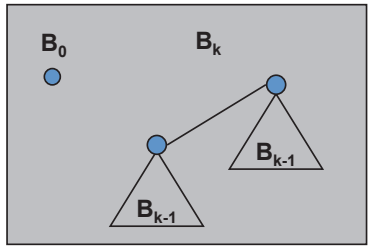
```
public interface AdaptMinPriorityQueue<K extends Comparable<K>,
   V extends Comparable<V>> extends MinPriorityQueue<K,V> {

   // If the priority queue contains an entry with the specified
   // value, returns the associated key and replaces it by the
   // specified key. Otherwise, returns null.
   K decreaseKey( V value, K newKey ) throws InvalidKeyException;

   // If the priority queue contains an entry with the specified
   // value, removes and returns that entry.
   // Otherwise, returns null.
   Entry<K,V> remove( V value );
}
```

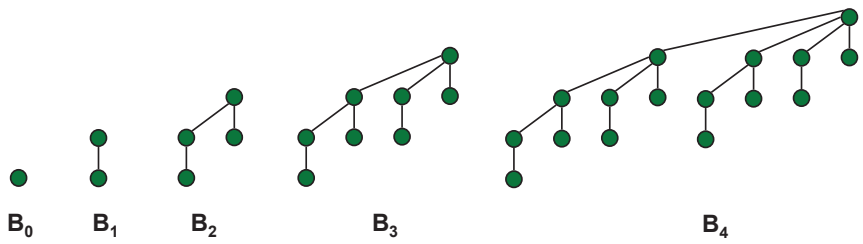

TAD Fila com Prioridade Fundível Organizada por Mínimos (K,V)

```
public interface MergeMinPriorityQueue<K extends Comparable<K>,
   V extends Comparable<V>> extends AdaptMinPriorityQueue<K,V> {

   // Removes all of the entries from the specified queue and
   // inserts them in the mergeable priority queue.
   // The two queues must be different and
   // their values must be all distinct.
   void merge( MergeMinPriorityQueue<K,V> priorityQueue )
        throws EqualPriorityQueuesException;
}
```


Árvore Binomial

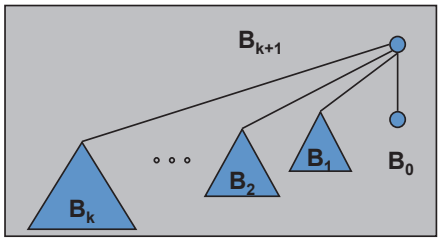
- Definição recursiva

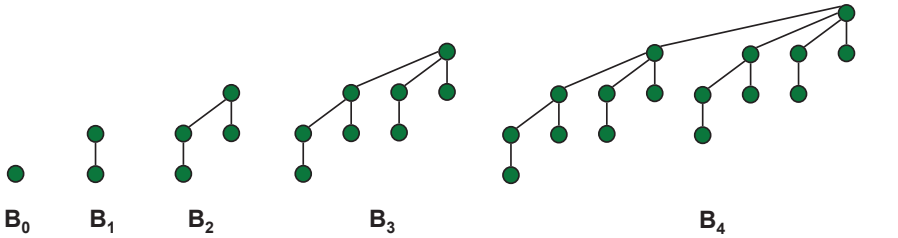




Árvore Binomial

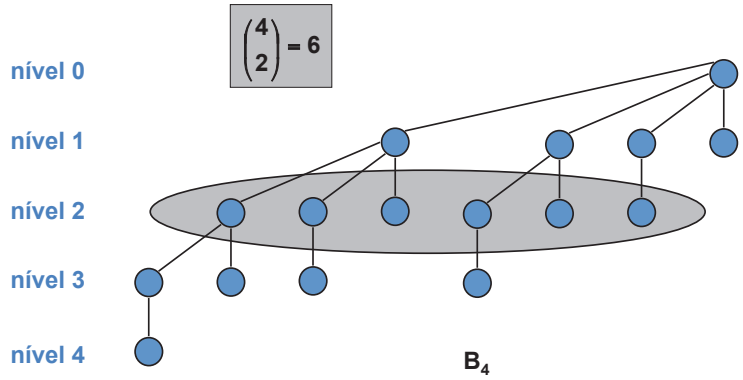
- Propriedades da árvore binomial B_k de ordem k
  - Número de nós = 2^k
  - Altura = k
  - Ordem da raiz = k





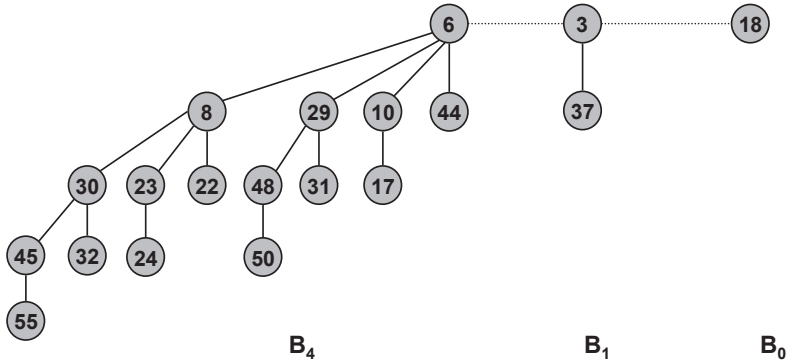
Árvore Binomial

- Propriedades da árvore binomial B_k de ordem k
  - B_k tem $\binom{k}{i}$ nós no nível i.



Fila Binomial

- Floresta de árvores binomiais que satisfazem as seguintes propriedades:
  - Cada árvore é uma fila ordenada por mínimos
  - 0 ou 1 árvores binomiais de ordem k

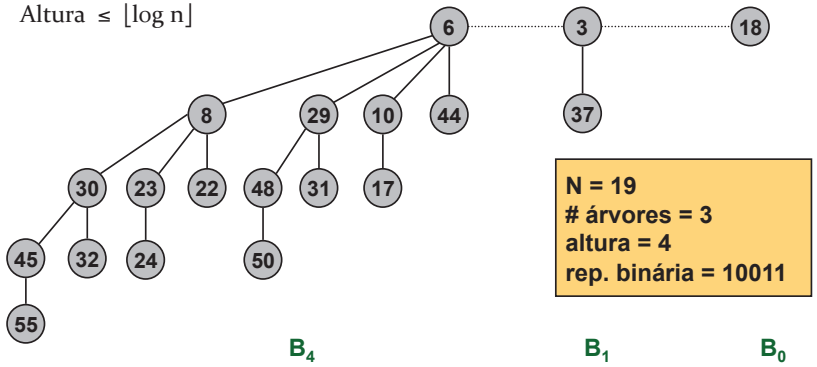


Fila Binomial: Propriedades

- Propriedades de uma fila binomial com **n** nós
  - Chave mínima na raíz de B_o, B_1, . . . , B_k
  - Contém árvore binomial B_i sse $b_i = 1$ onde $b_n \ldots b_2 b_1 b_o$ é uma representação binária de n
  - No máximo $\lfloor \log n \rfloor + 1$ árvores binomiais
  - Altura $\leq \lfloor \log n \rfloor$

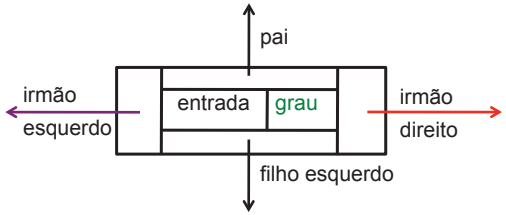
Fila Binomial: Propriedades

- Propriedades de uma fila binomial com **n** nós
  - Chave mínima na raíz de $B_0, B_1, \ldots, B_k$
  - Contém árvore binomial B_i sse $b_i = 1$ onde $b_n \ldots b_2 b_1 b_0$ é uma representação binária de n
  - No máximo $\lfloor \log n \rfloor + 1$ árvores binomiais
  - Altura $\leq \lfloor \log n \rfloor$

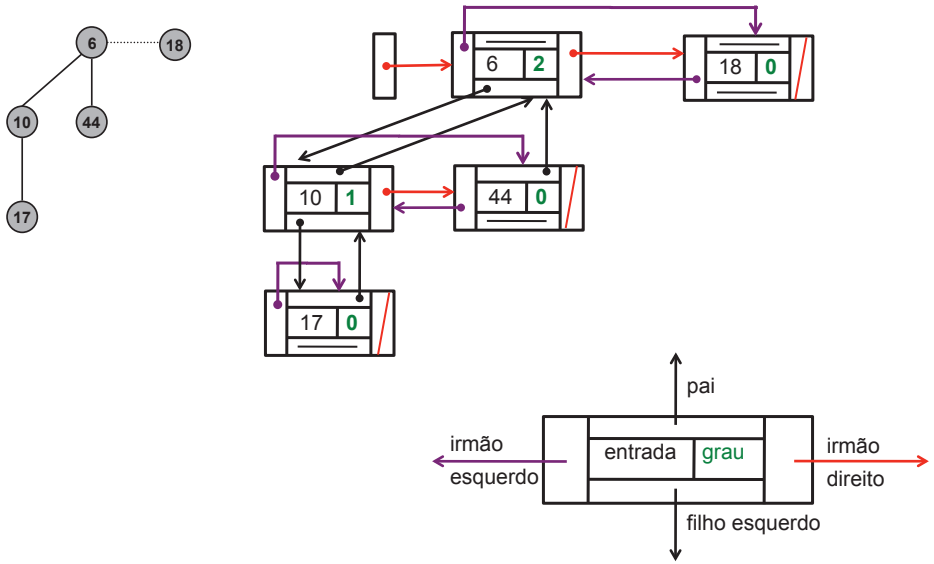


Fila Binomial: Implementação

- Cada nó tem 4 apontadores (parent, left son, left, right)
- Raizes das árvores ligada por uma lista ligada
  - Ordem das árvores decresce da esquerda para a direita



Fila Binomial: Implementação



Descrição das Operações (n entradas)

- `BinQueue( )`
  - Inicializa-se a `head` a `null`
  - Complexidade — $O(1)$ em todos os casos
- `boolean isEmpty( )`
  - Testa se a `head` é `null`
  - Complexidade — $O(1)$ em todos os casos

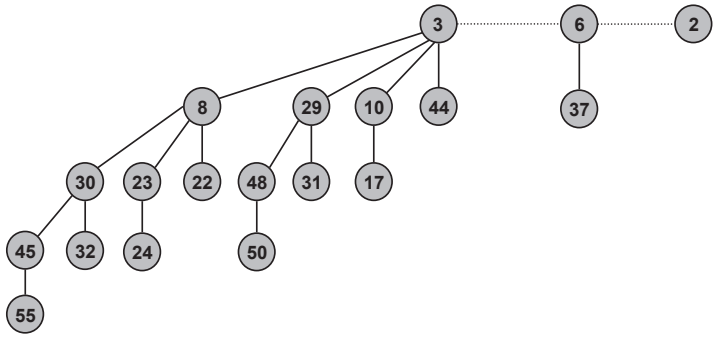
Descrição das Operações (n entradas)

- `Entry<K,V> minEntry( )`
  - Percorrem-se as raizes das árvores (principais) e retorna-se uma dessas entradas com chave mínima.

  Complexidade — O(log n)

Fila Binomial: `minEntry`

- Tempo de execução no pior caso — O(log n)
  - No máximo $\lfloor \log n \rfloor + 1$ árvores binomiais

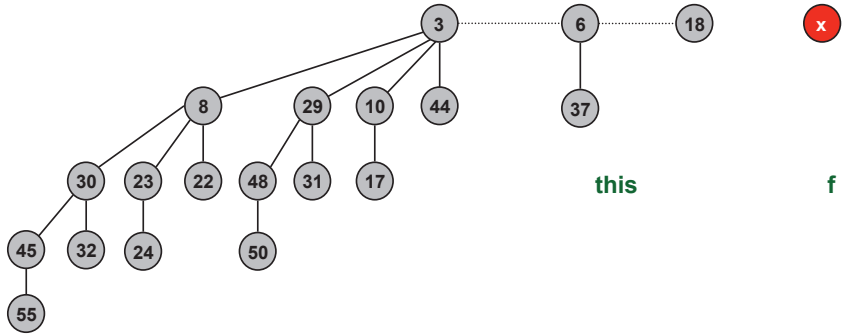


Descrição das Operações (n entradas)

- `void insert( )`
  - Cria-se uma fila binomial f com a entrada (key, value)
  - Funde-se a fila com f: `this.merge(f)`

  Complexidade: igual à complexidade da fusão

Fila Binomial: `insert`

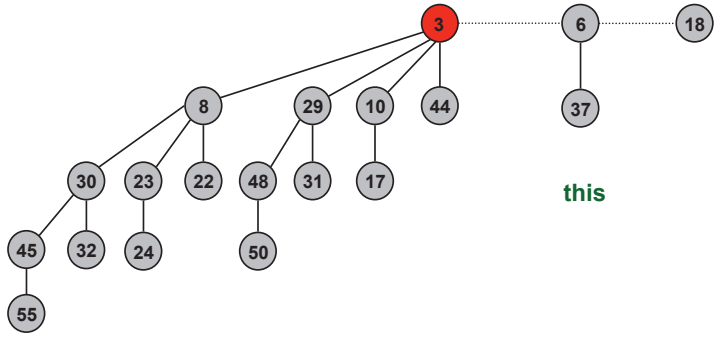


Descrição das Operações (n entradas)

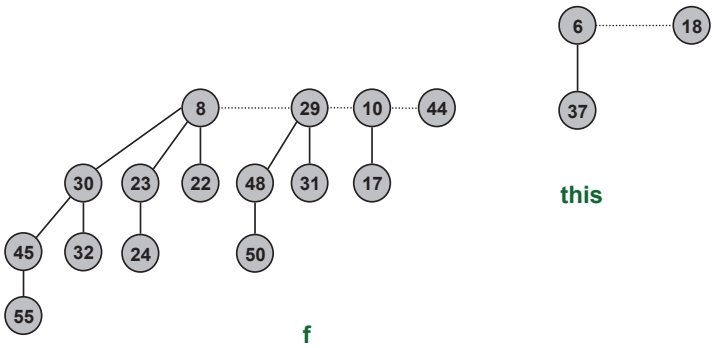
- `Entry<K,V> removeMin( )`
  - Descobre-se uma entrada `e` com chave mínima (e a árvore B_k cuja raiz é `e`) — O(log n)
  - Retira-se a árvore B_k da fila — O(1)
  - Cria-se uma fila binomial f com as sub-árvores da raiz de B_k — O(log n)
  - Funde-se a fila com `f`: `this.merge(f)`
  - Retorna-se `e`

  Complexidade — log n + complexidade da fusão

Árvore Binomial: `removeMin`



Árvore Binomial: `removeMin`



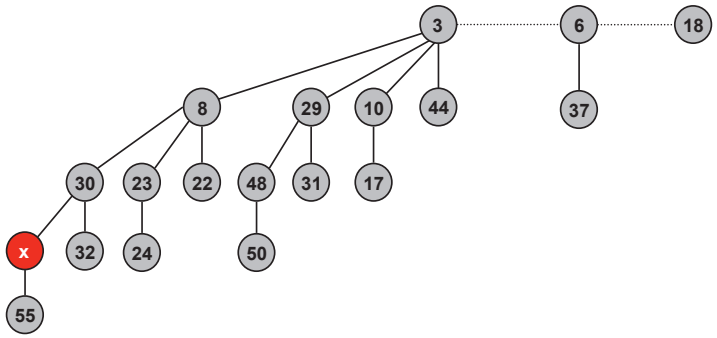
Descrição das Operações (n entradas)

- `K decreaseKey(BinNode<K,V> node, K newKey)`
  - Executa-se o borbulhar ascendente a partir do nó (como no heap)

  Complexidade — O(log n) no pior caso

Árvore Binomial: decreaseKey

- Tempo de execução — O(log n)
  - Proporcional ao nível do nó x ≤ ⌊log n⌋



Descrição das Operações (n entradas)

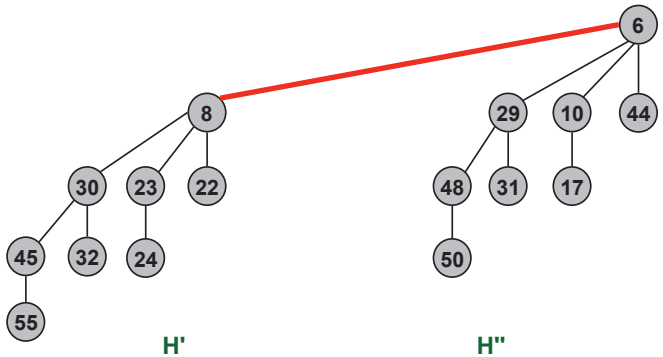
- O método remove não acrescenta funcionalidade

```
Entry<K,V> remove( BinNode<K,V> node ) {
     Entry<K,V> entry = node.getEntry();
     this.decreaseKey(node, -∞);
     this.removeMin();
     return entry;
}
```

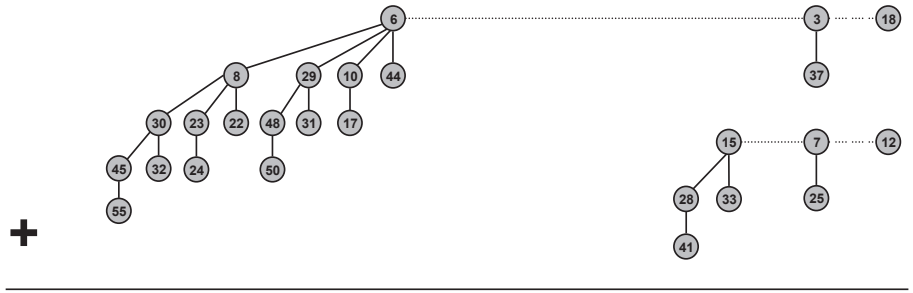
Complexidade — log n + complexidade da fusão

Fila Binomial : merge

- Criar fila H que é a fusão de H' e H''
  - Se H' e H'' são ambas árvores binomiais de ordem k
    - Ligar as raizes de H' e H''
    - Escolher a menor chave para raíz de H



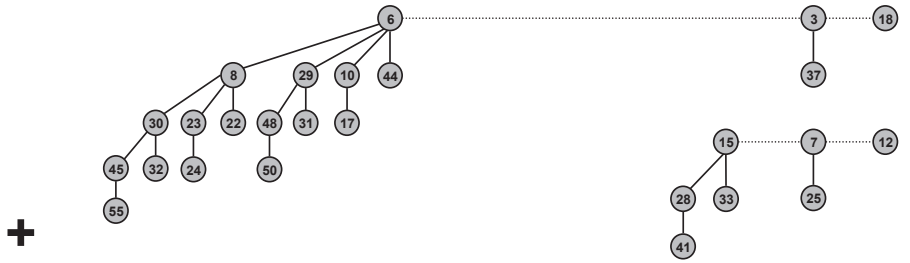
Fila Binomial : merge



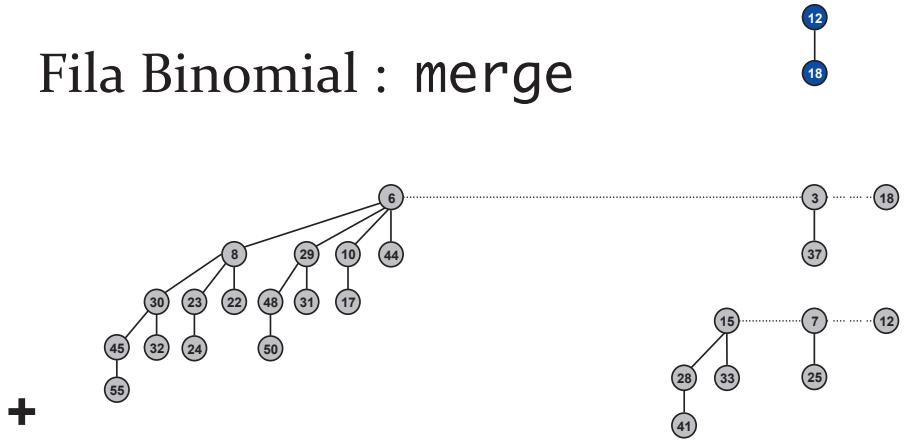
19 + 7 = 26

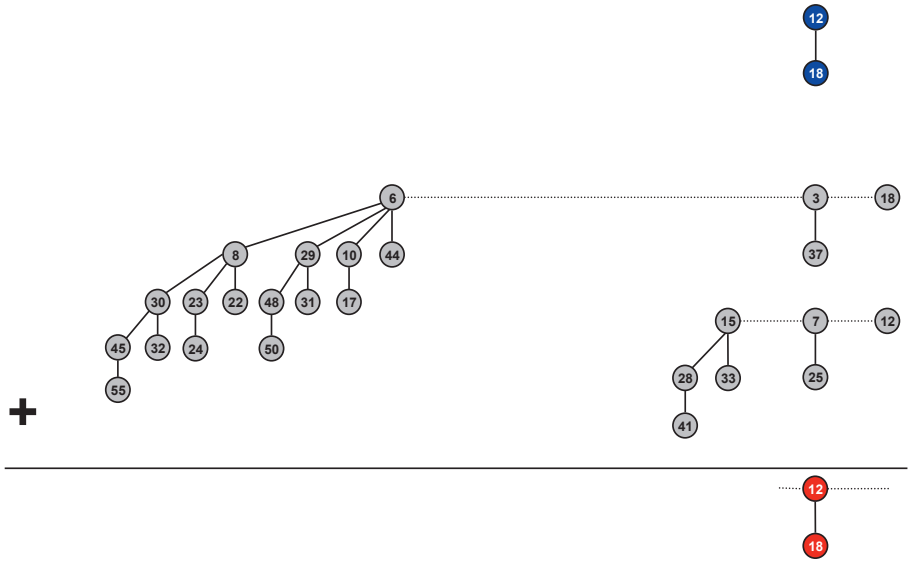
		1	1	1	
	1	0	0	1	1
+	0	0	1	1	1
	1	1	0	1	0

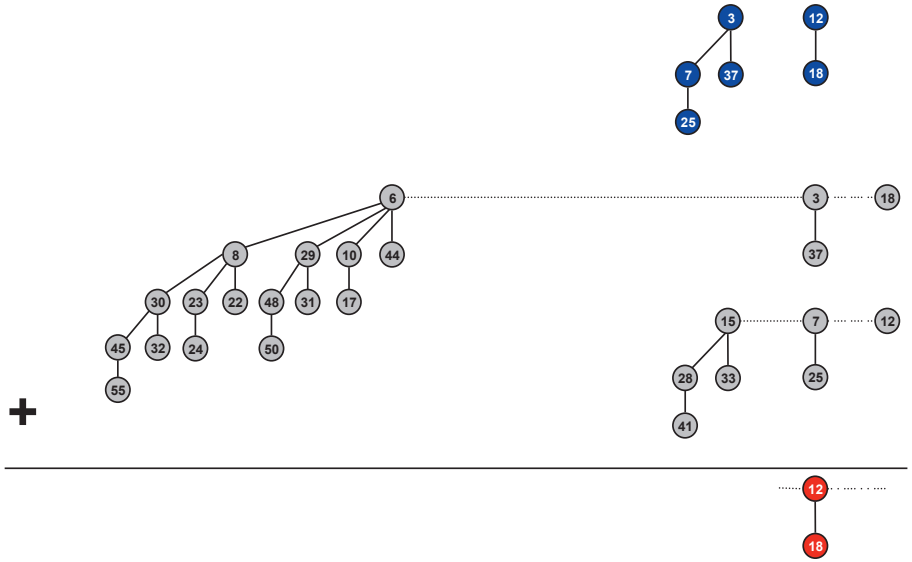
Fila Binomial : merge

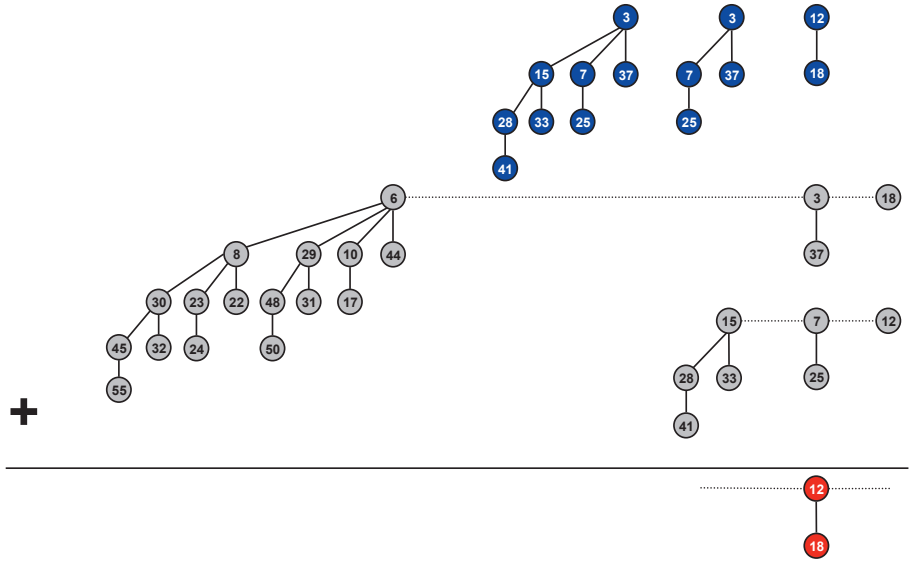


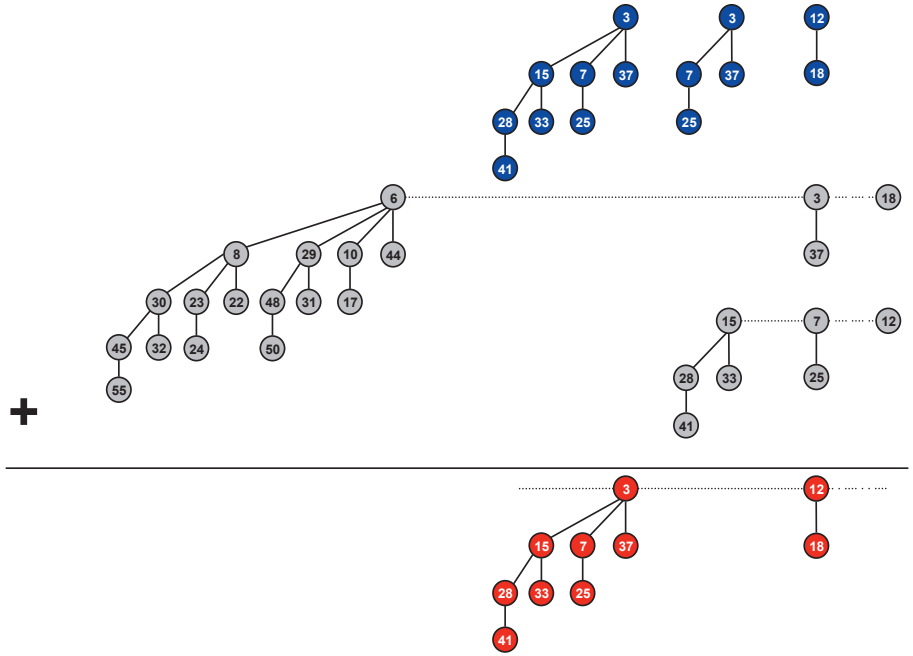
Fila Binomial : merge

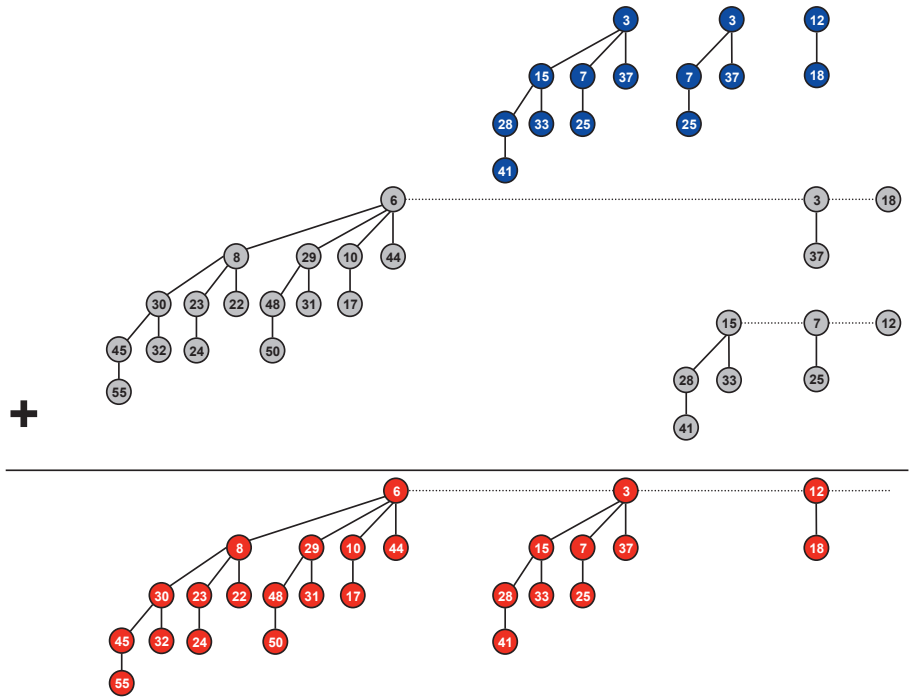












Complexidade da Fusão

- O número de passos do algoritmo de fusão é inferior ou igual ao número total de árvores binomiais iniciais
- Sejam
  - n_1 e n_2 os números de entradas das filas binomiais iniciais
  - $n = n_1 + n_2$ o número de entradas da fila binomial resultante

$$\begin{aligned} \texttt{merge} &\leq \lfloor \log n_1 \rfloor + 1 + \lfloor \log n_2 \rfloor + 1 \\ &\leq 2\lfloor \log n \rfloor + 2 \\ &= O(\log n) \end{aligned}$$

Complexidades da Fila com Prioridade Fundível no Pior Caso (n entradas)

	Heap	Fila Binomial
`isEmpty`	O(1)	O(1)
`size`	O(1)	O(1)
`minEntry`	O(1)	O(log n)
`insert`	O(log n)	O(log n)
`removeMin`	O(log n)	O(log n)
`decreaseKey`	O(log n)	O(log n)
`remove`	O(log n)	O(log n)
`merge` (m entradas)	O(n + m)	O(log (n+m))

Complexidade dos Algoritmos de Prim e Dijkstra com Heap ou Fila Binomial (grafo em listas de adjacências)

Criar fila	O(#V)
Inicializar 2 vectores	O(#V)
Inserir origem na fila	O(1)
(#V ou ≤ #V) remover mínimo da fila	O(#V log #V)
(#V ou ≤ #V) percorrer sucessores directos	O(#A)
(#A ou ≤ #A) inserir na fila ou decrementar chave	O(#A log #V)

TOTAL Prim	O(#A log #V)
TOTAL Dijkstra	O((#V + #A) log #V)

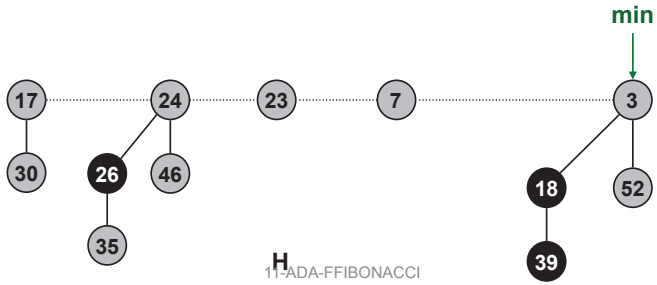
11. Filas de Fibonacci

Planeamento

Apresentação da disciplina
Programação Dinâmica
Introdução aos Grafos
Percurso em largura e profundidade
Ordenação Topológica
Árvore Mínima de Cobertura (Algoritmo de Kruskal)
TAD Partição
Complexidade Amortizada
Árvore Mínima de Cobertura (Algoritmo de Prim)
TAD Fila com Prioridade Adaptável
Algoritmo de Dijskstra
Filas Binomiais
Filas de Fibonacci
Algoritmo de Bellman-Ford
Fluxo Máximo: Algoritmo de Ford-Fulkerson
Algoritmo de Edmonds-Karp
Introdução à Teoria da Complexidade
Exemplos de Problemas NP
Redutibilidade entre Problemas de Decisão

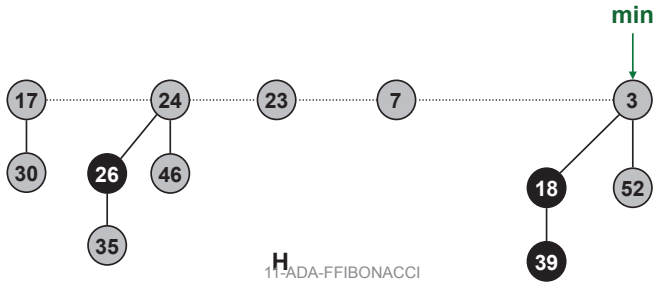
Fila de Fibonacci: Estrutura

- Conjunto de filas ordenadas por mínimos
  - Alguns nós estão marcados
    - Um nó está marcado se perdeu um filho desde que é filho do pai actual
    - Uma raiz nunca está marcada



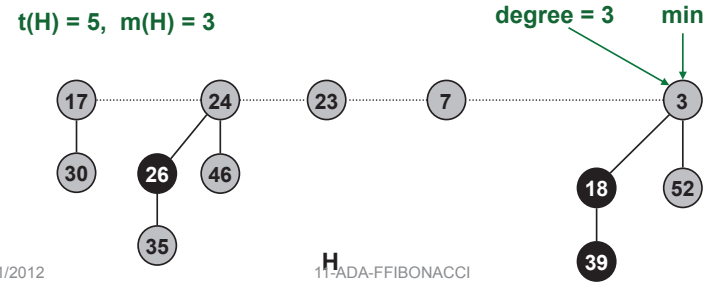
Fila de Fibonacci: Implementação

- A fila é implementada em lista circular duplamente ligada com cabeça
- A cabeça da lista (que implementa a fila) aponta para uma árvore cuja raiz tem a chave mínima
- Os filhos de um nó estão implementados em lista circular duplamente ligada, com cabeça
- A cabeça da lista dos filhos aponta para um filho qualquer



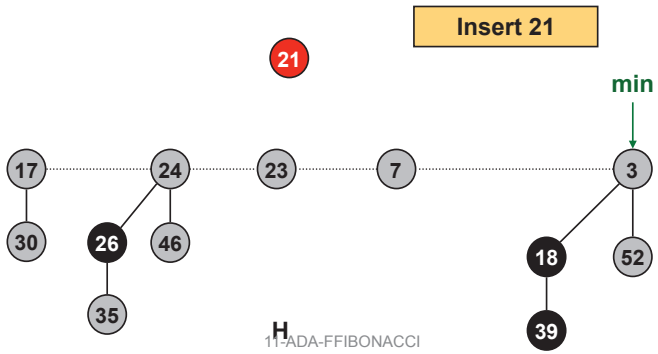
Fila de Fibonacci: Estruturas

- degree = ordem de um nó
- mark = marca de um nó (cinzento ou preto)
- t(H) = # árvores
- m(H) = # nós marcados



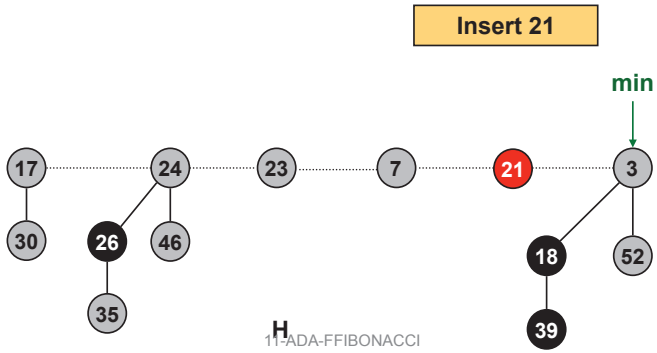
Fila de Fibonacci: Inserção

- Criar árvore singular
- Adicionar árvore à esquerda do mínimo
- Actualizar o apontador para o mínimo



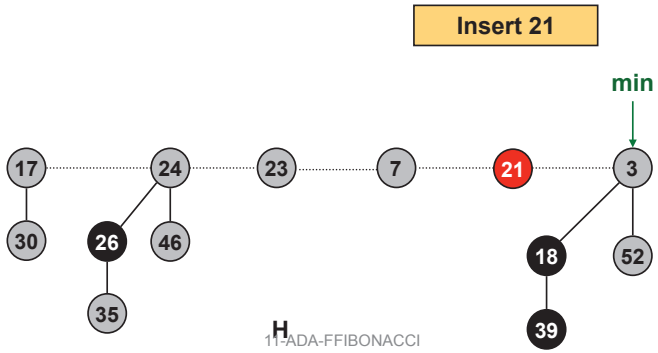
Fila de Fibonacci: Inserção

- Criar árvore singular
- Adicionar árvore à esquerda do mínimo
- Actualizar o apontador para o mínimo



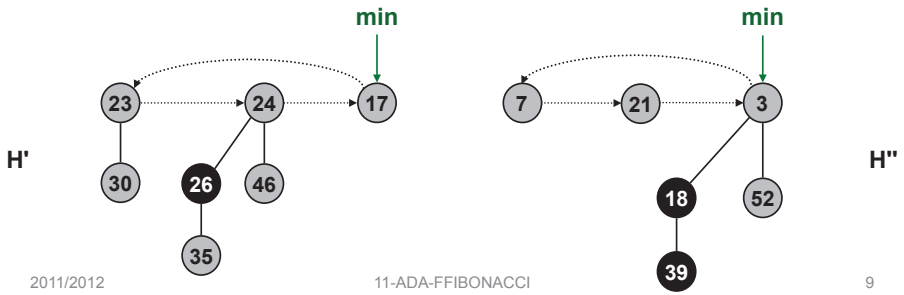
Fila de Fibonacci: Inserção

- Criar árvore singular
- Adicionar árvore à esquerda do mínimo
- Actualizar o apontador para o mínimo
- Complexidade — O(1)



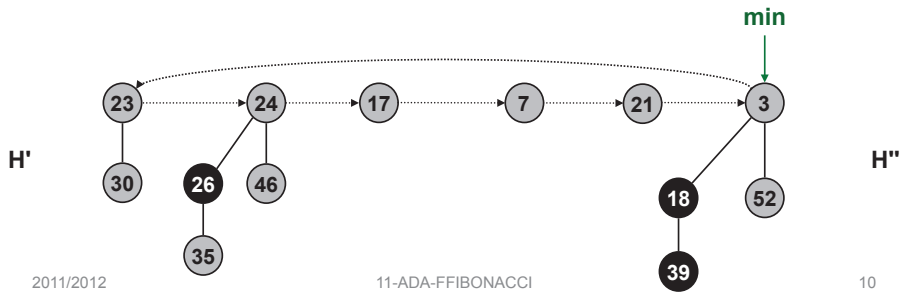
Fila de Fibonacci: Fusão

- Concatenar 2 filas de fibonacci
- As listas das raízes são filas duplamente ligadas circulares



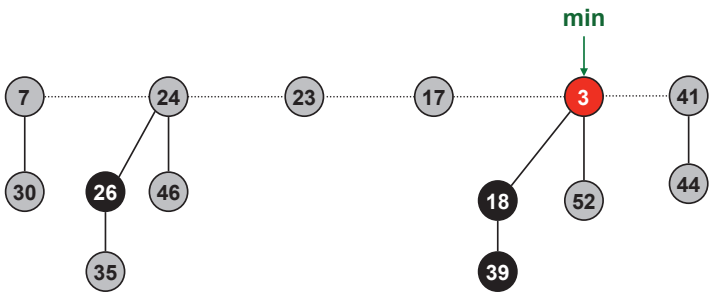
Fila de Fibonacci: Fusão

- Concatenar 2 filas de fibonacci
- As listas das raízes são filas duplamente ligadas circulares



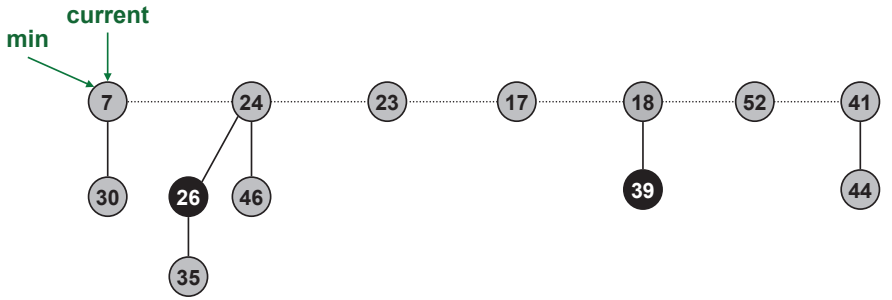
Fila de Fibonacci: Remover Mínimo

- Remover mínimo e concatenar os seus filhos na lista das raízes
- Consolidar árvores de forma que não existam duas raízes com a mesma ordem



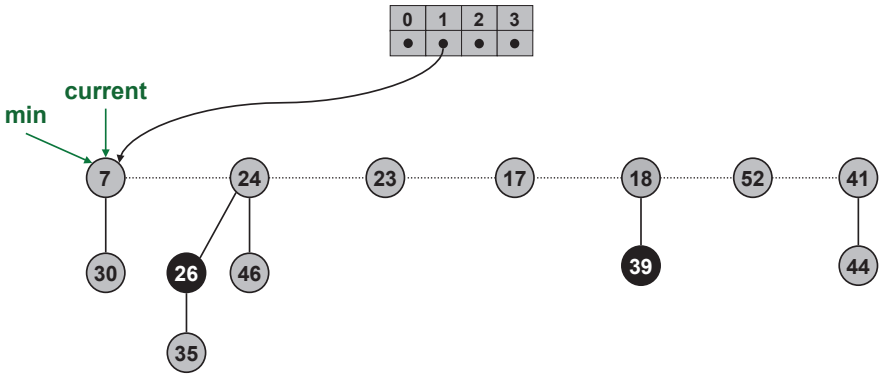
Fila de Fibonacci: Remover Mínimo

- Remover mínimo e concatenar os seus filhos na lista das raízes
- Consolidar árvores de forma que não existam duas raízes com a mesma ordem



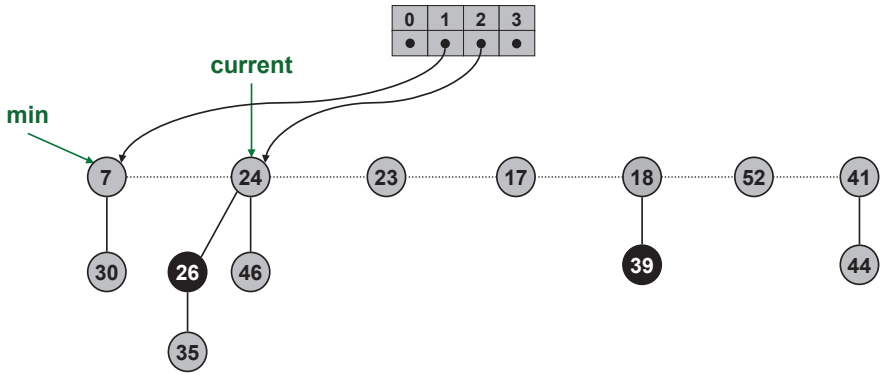
Fila de Fibonacci: Remover Mínimo

- Remover mínimo e concatenar os seus filhos na lista das raízes
- Consolidar árvores de forma que não existam duas raízes com a mesma ordem



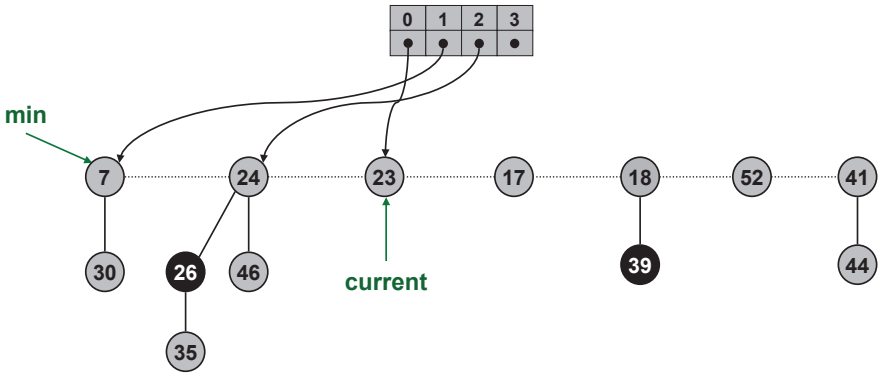
Fila de Fibonacci: Remover Mínimo

- Remover mínimo e concatenar os seus filhos na lista das raízes
- Consolidar árvores de forma que não existam duas raízes com a mesma ordem



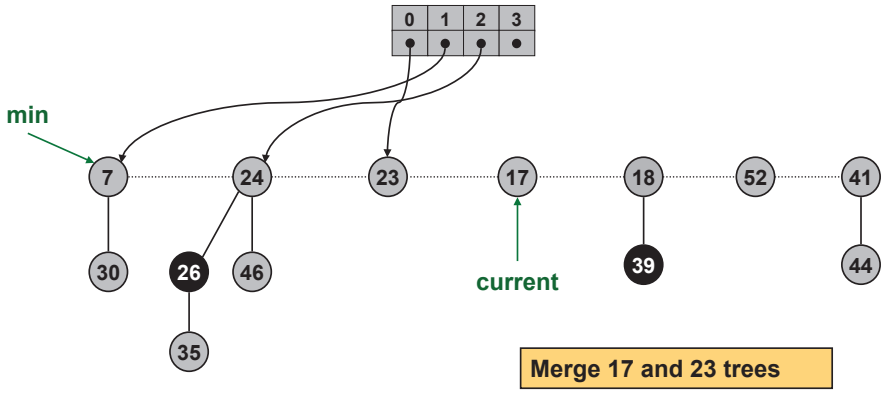
Fila de Fibonacci: Remover Mínimo

- Remover mínimo e concatenar os seus filhos na lista das raízes
- Consolidar árvores de forma que não existam duas raízes com a mesma ordem



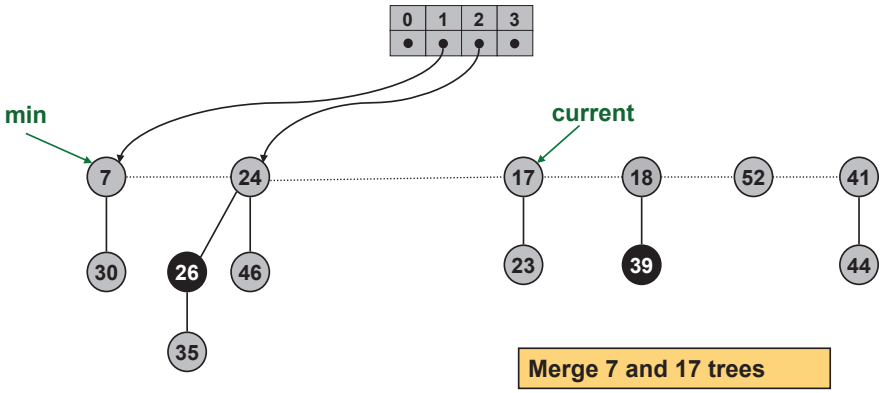
Fila de Fibonacci: Remover Mínimo

- Remover mínimo e concatenar os seus filhos na lista das raízes
- Consolidar árvores de forma que não existam duas raízes com a mesma ordem



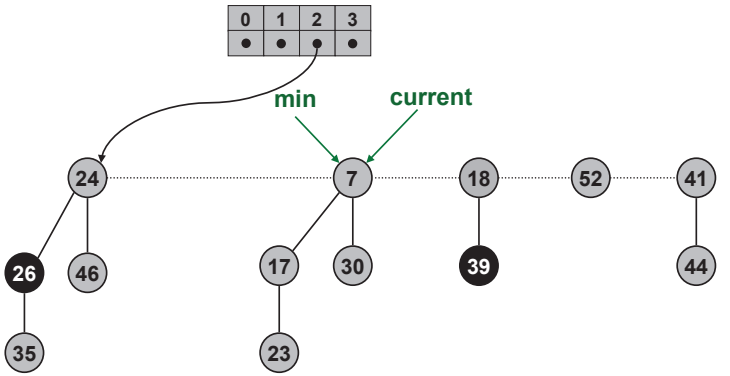
Fila de Fibonacci: Remover Mínimo

- Remover mínimo e concatenar os seus filhos na lista das raízes
- Consolidar árvores de forma que não existam duas raízes com a mesma ordem



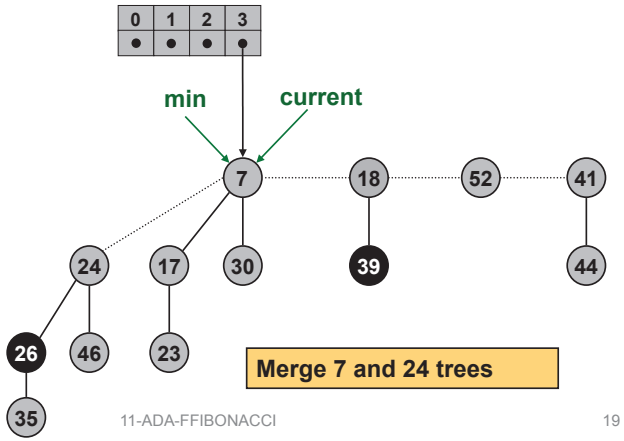
Fila de Fibonacci: Remover Mínimo

- Remover mínimo e concatenar os seus filhos na lista das raízes
- Consolidar árvores de forma que não existam duas raízes com a mesma ordem



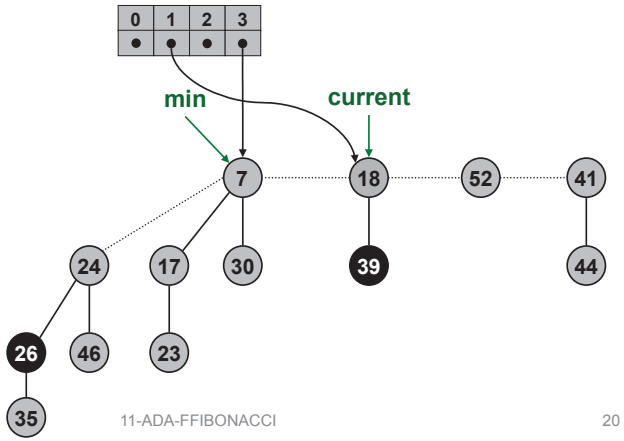
Fila de Fibonacci: Remover Mínimo

- Remover mínimo e concatenar os seus filhos na lista das raízes
- Consolidar árvores de forma que não existam duas raízes com a mesma ordem



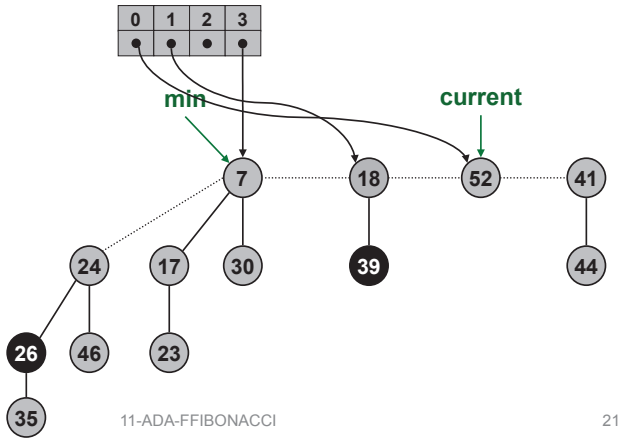
Fila de Fibonacci: Remover Mínimo

- Remover mínimo e concatenar os seus filhos na lista das raízes
- Consolidar árvores de forma que não existam duas raízes com a mesma ordem



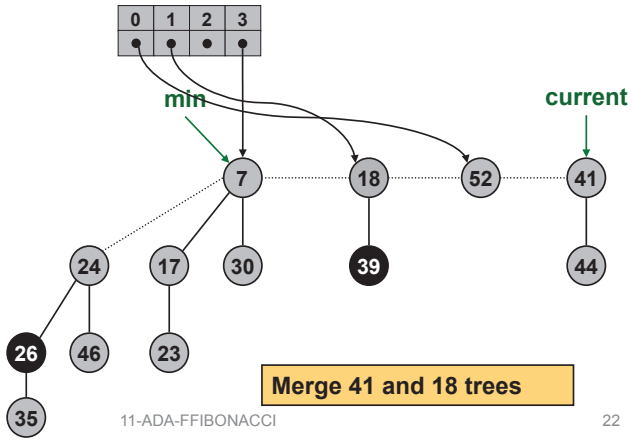
Fila de Fibonacci: Remover Mínimo

- Remover mínimo e concatenar os seus filhos na lista das raízes
- Consolidar árvores de forma que não existam duas raízes com a mesma ordem



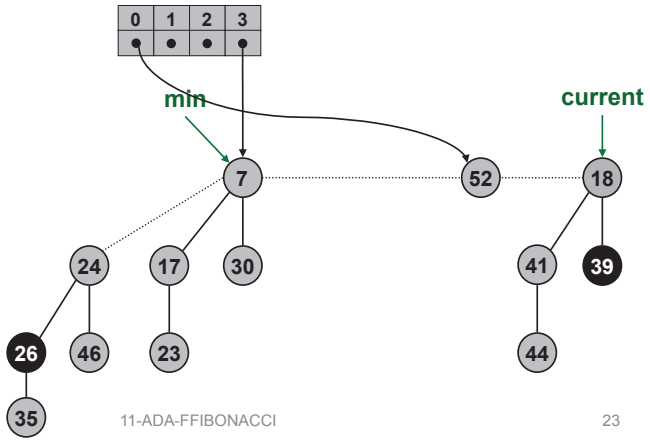
Fila de Fibonacci: Remover Mínimo

- Remover mínimo e concatenar os seus filhos na lista das raízes
- Consolidar árvores de forma que não existam duas raízes com a mesma ordem



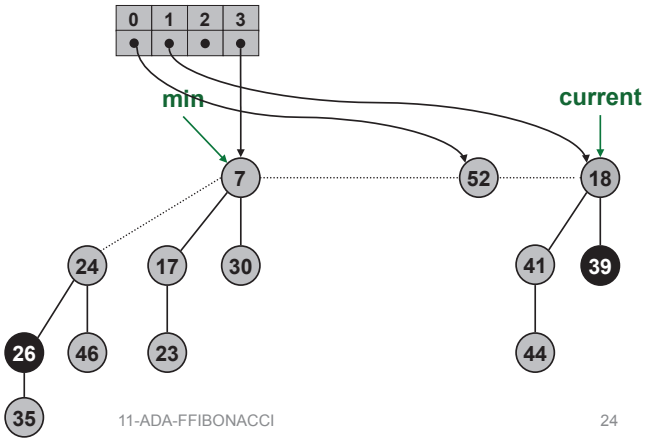
Fila de Fibonacci: Remover Mínimo

- Remover mínimo e concatenar os seus filhos na lista das raízes
- Consolidar árvores de forma que não existam duas raízes com a mesma ordem



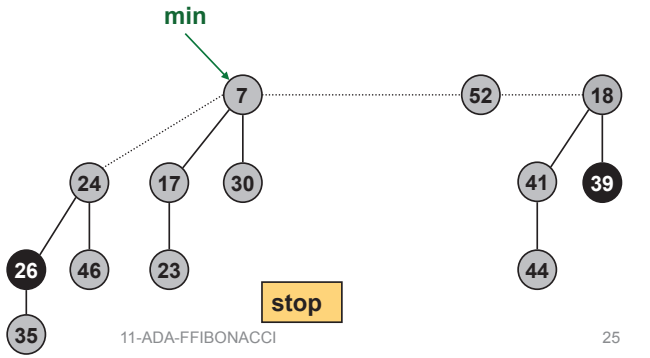
Fila de Fibonacci: Remover Mínimo

- Remover mínimo e concatenar os seus filhos na lista das raízes
- Consolidar árvores de forma que não existam duas raízes com a mesma ordem



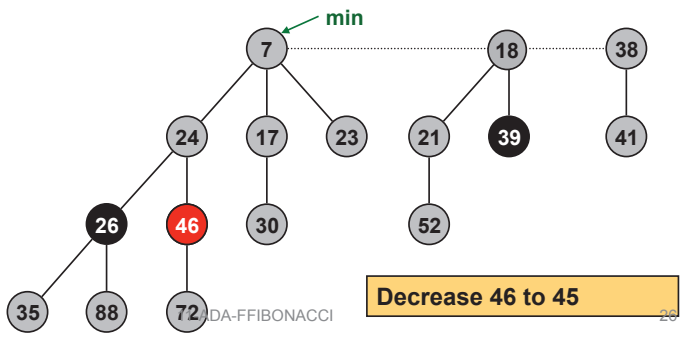
Fila de Fibonacci: Remover Mínimo

- Remover mínimo e concatenar os seus filhos na lista das raízes
- Consolidar árvores de forma que não existam duas raízes com a mesma ordem



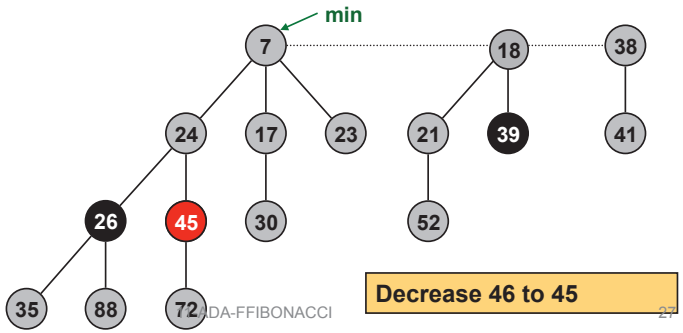
Fila de Fibonacci: Decrescer Chave

- Decrescer chave do elemento x para k
  - Caso 0: proprierdade min-heap não violada
    - Decrescer chave de x para k
    - Alterar o apontador mínimo se necessário



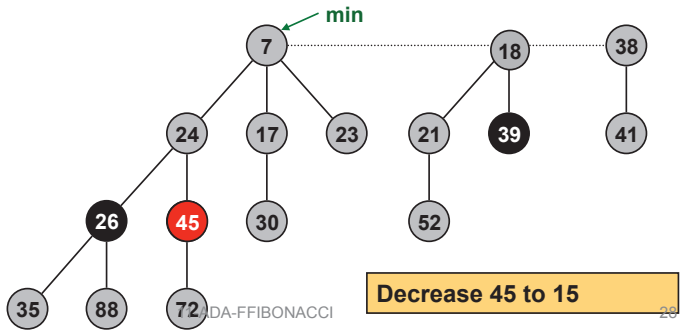
Fila de Fibonacci: Decrescer Chave

- Decrescer chave do elemento x para k
  - Caso 0: proprierdade min-heap não violada
    - Decrescer chave de x para k
    - Alterar o apontador mínimo se necessário



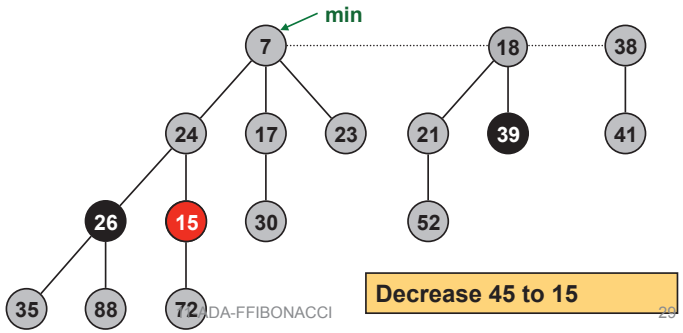
Fila de Fibonacci: Decrescer Chave

- Decrescer chave do elemento x para k
  - Caso 1: pai de x não tem marca
    - Decrescer chave de x para k
    - Remover ligação de x ao seu pai
    - Marcar pai
    - Adicionar árvore com raíz em x à lista de raízes e actualizar mínimo



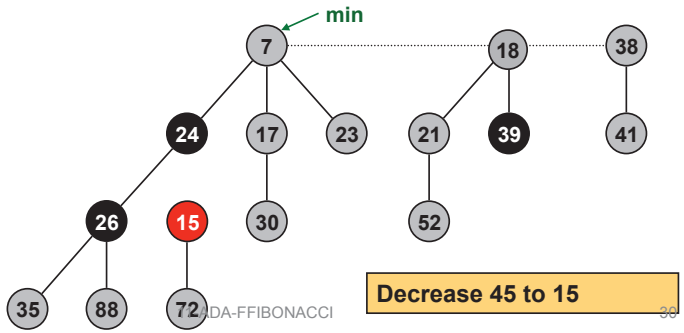
Fila de Fibonacci: Decrescer Chave

- Decrescer chave do elemento x para k
  - Caso 1: pai de x não tem marca
    - Decrescer chave de x para k
    - Remover ligação de x ao seu pai
    - Marcar pai
    - Adicionar árvore com raíz em x à lista de raízes e actualizar mínimo



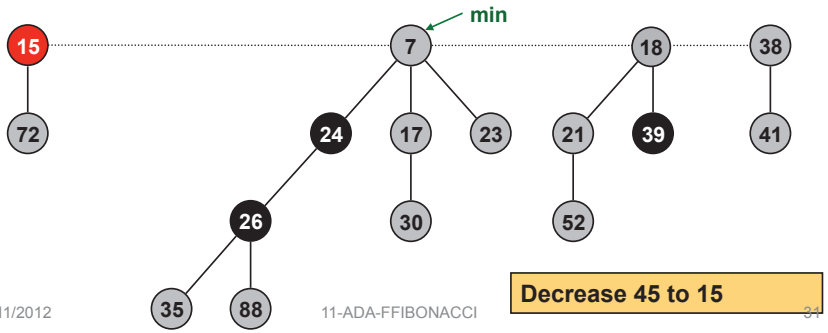
Fila de Fibonacci: Decrescer Chave

- Decrescer chave do elemento x para k
  - Caso 1: pai de x não tem marca
    - Decrescer chave de x para k
    - Remover ligação de x ao seu pai
    - Marcar pai
    - Adicionar árvore com raíz em x à lista de raízes e actualizar mínimo



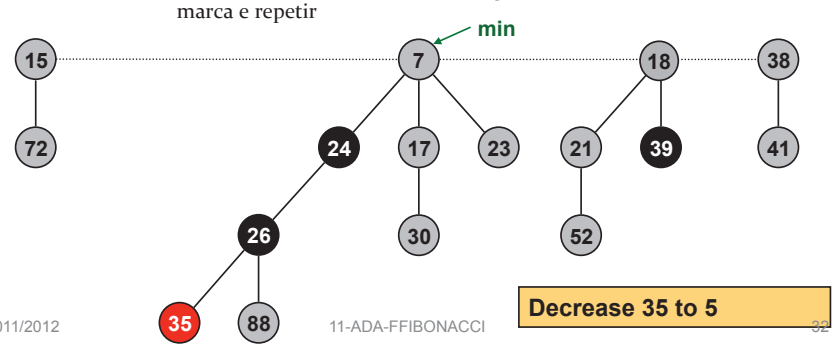
Fila de Fibonacci: Decrescer Chave

- Decrescer chave do elemento x para k
  - Caso 1: pai de x não tem marca
    - Decrescer chave de x para k
    - Remover ligação de x ao seu pai
    - Marcar pai
    - Adicionar árvore com raíz em x à lista de raízes e actualizar mínimo



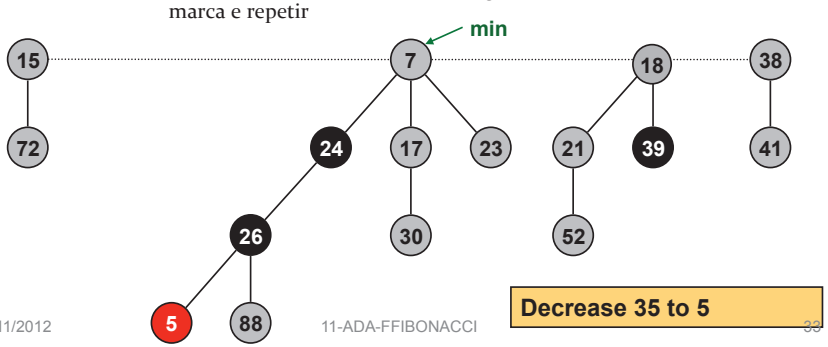
Fila de Fibonacci: Decrescer Chave

- Decrescer chave do elemento x para k
  - Caso 2: pai de x tem marca
    - Decrescer chave de x para k
    - Remover ligação de x ao seu pai p[x] e adicionar x à lista de raízes
    - Remover ligação entre p[x] e p[p[x]] e adicionar p[x] à lista de raízes
      - If p[p[x]] não está marcado, marcar p[p[x]]
      - If p[p[x]] está marcado, remover ligação entre p[p[x]] e o seu pai, retirar marca e repetir



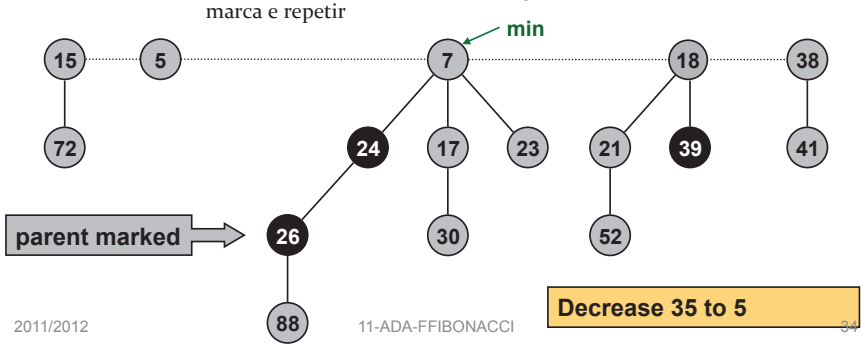
Fila de Fibonacci: Decrescer Chave

- Decrescer chave do elemento x para k
  - Caso 2: pai de x tem marca
    - Decrescer chave de x para k
    - Remover ligação de x ao seu pai p[x] e adicionar x à lista de raízes
    - Remover ligação entre p[x] e p[p[x]] e adicionar p[x] à lista de raízes
      - If p[p[x]] não está marcado, marcar p[p[x]]
      - If p[p[x]] está marcado, remover ligação entre p[p[x]] e o seu pai, retirar marca e repetir



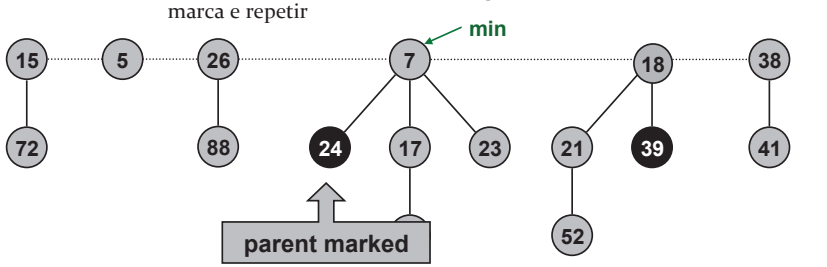
Fila de Fibonacci: Decrescer Chave

- Decrescer chave do elemento x para k
  - Caso 2: pai de x tem marca
    - Decrescer chave de x para k
    - Remover ligação de x ao seu pai p[x] e adicionar x à lista de raízes
    - Remover ligação entre p[x] e p[p[x]] e adicionar p[x] à lista de raízes
      - If p[p[x]] não está marcado, marcar p[p[x]]
      - If p[p[x]] está marcado, remover ligação entre p[p[x]] e o seu pai, retirar marca e repetir



Fila de Fibonacci: Decrescer Chave

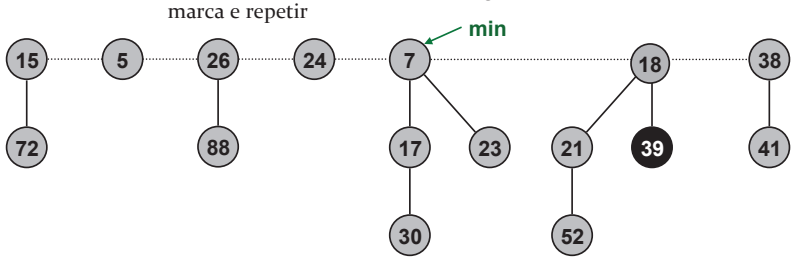
- Decrescer chave do elemento x para k
  - Caso 2: pai de x tem marca
    - Decrescer chave de x para k
    - Remover ligação de x ao seu pai p[x] e adicionar x à lista de raízes
    - Remover ligação entre p[x] e p[p[x]] e adicionar p[x] à lista de raízes
      - If p[p[x]] não está marcado, marcar p[p[x]]
      - If p[p[x]] está marcado, remover ligação entre p[p[x]] e o seu pai, retirar marca e repetir



Decrease 35 to 5

Fila de Fibonacci: Decrescer Chave

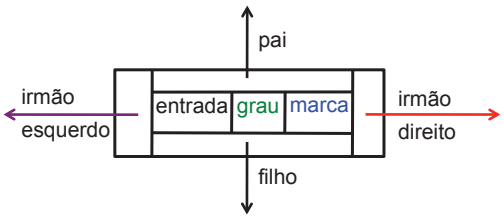
- Decrescer chave do elemento x para k
  - Caso 2: pai de x tem marca
    - Decrescer chave de x para k
    - Remover ligação de x ao seu pai p[x] e adicionar x à lista de raízes
    - Remover ligação entre p[x] e p[p[x]] e adicionar p[x] à lista de raízes
      - If p[p[x]] não está marcado, marcar p[p[x]]
      - If p[p[x]] está marcado, remover ligação entre p[p[x]] e o seu pai, retirar marca e repetir



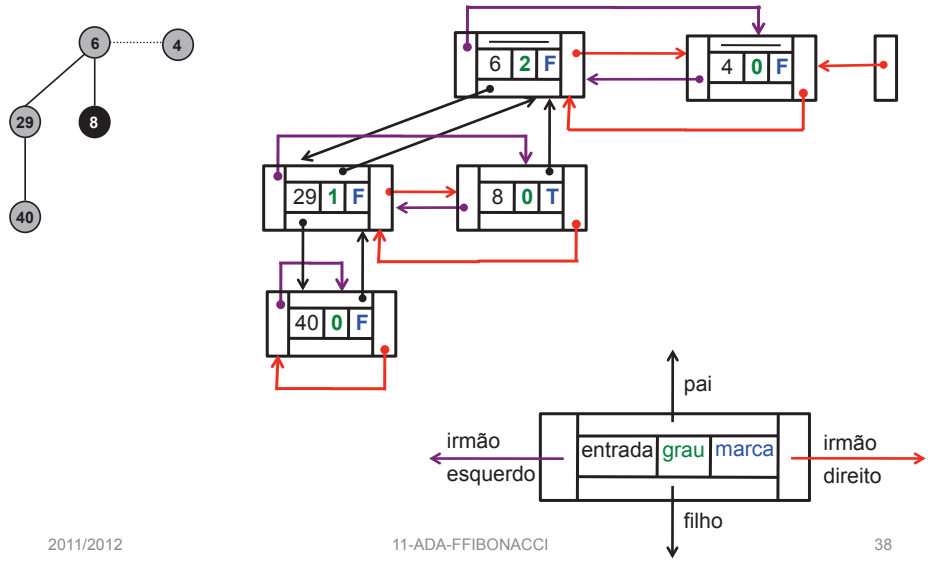
Decrease 35 to 5

Fila de Fibonacci: Implementação

- A fila é implementada em lista circular duplamente ligada com cabeça
- A cabeça da lista (que implementa a fila) aponta para uma árvore cuja raiz tem a chave mínima
- Os filhos de um nó estão implementados em lista circular duplamente ligada, com cabeça
- A cabeça da lista dos filhos aponta para um filho qualquer



Fila de Fibonacci: Implementação



Classe Nó de Árvore de Fibonacci

```
// The node is marked if it has lost a child since
// it has the current parent. Roots are unmarked
private boolean mark;

public FibNode( K key, V value ) {
     entry = new EntryClass<K,V>(key, value);
     degree = 0;
     child = null;
     leftSibling = this;
     rightSibling = this;
     parent = null;
     mark = false;
}

// Todos os SETs e GETs
...
```


Classe Nó de Árvore de Fibonacci

```
class FibNode<K,V> {
   // Entry stored in the node
   private EntryClass<K,V> entry;

   // The degree of the node
   private int degree;

   // (Pointer to) a child
   private FibNode<K,V> child;

   // (Pointer to) the left sibling
   private FibNode<K,V> leftSibling;

   // (Pointer to) the right sibling
   private FibNode<K,V> rightSibling;

   // (Pointer to) the parent
   private FibNode<K,V> parent;
```


Classe Nó de Árvore de Fibonacci

```
public void incrementDegree( ) {
     degree++;
}

public void decrementDegree( ) {
     degree--;
}

public void makeSingleton( ) {
     leftSibling = this;
     rightSibling = this;
}
```


Classe Nó de Árvore de Fibonacci

```
   public boolean isMarked( ) {
        return mark;
   }

   public void mark( ) {
        mark = true;
   }

   public void unmark( ) {
        mark = false;
   }
} // End of class FibNode
```

Criação de uma Fila Vazia

- `FibQueue( )`
  - Inicializa-se `min` a `null`
  - Inicializa-se `currentSize` a `0`
  - Complexidade — $O(1)$ em todos os casos

Classe Interna Fila de Fibonacci

```
class FibQueue<K extends Comparable<K>, V> {

   // (Pointer to) a tree with the smallest key
   protected FibNode<K,V> min;

   // Number of entries in the priority queue
   protected int currentSize;

   public FibQueue( ) {
        min = null;
        currentSize = 0;
   }
   . . .
}
```


Classe Pública Fila de Fibonacci

```
public class FibonacciQueue<K extends Comparable<K>,
               V extends Comparable<V>>
               implements MergeMinPriorityQueue<K,V> {

   // The Fibonacci queue
   protected FibQueue<K,V> queue;

   // The dictionary
   protected Dictionary<V, FibNode<K,V>> dict;

   public FibonacciQueue( ) {
        queue = new FibQueue<K,V>();
        dict = new SepChainHashTable<V, FibNode<K,V>>();
   }
   . . .
}
```


Classe Pública Fila de Fibonacci

```
   // Returns true iff the priority queue contains no entries
   public boolean isEmpty( ) {
        return dict.isEmpty();
   }

   // Returns the number of entries in the priority queue
   public int size( ) {
        return dict.size();
   }

   // Returns an entry with the smallest key in the priority queue
   public Entry<K,V> minEntry( ) throws EmptyPriorityQueueException {
        if ( this.isEmpty() )
            throw new EmptyPriorityQueueException();

        return queue.minEntry();
   }
```

Classe Pública Fila de Fibonacci

```
// Inserts the entry (key, value) in the priority queue
public void insert( K key, V value ) throws InvalidValueException {
     if ( dict.find(value) != null )
         throw new InvalidValueException(
             "The queue already has an entry with the value.");

     FibNode<K,V> node = queue.insert(key, value);
     dict.insert(value, node);
}
```


Classe Pública Fila de Fibonacci

```
// Removes an entry with the smallest key from the priority queue
// and returns that entry
public Entry<K,V> removeMin( ) throws EmptyPriorityQueueException {
     if ( this.isEmpty() )
         throw new EmptyPriorityQueueException();

     Entry<K,V> entry = queue.removeMin();
     dict.remove( entry.getValue() );
     return entry;
}
```


Classe Pública Fila de Fibonacci

```
// If the priority queue contains an entry with the specified value,
// returns the associated key and replaces it by the specified key
// (which is less than the old one). Otherwise, returns null
public K decreaseKey( V value, K newKey ) throws InvalidKeyException{
     FibNode<K,V> node = dict.find(value);
     if ( node == null )
         return null;

     K oldKey = node.getKey();
     if ( oldKey.compareTo(newKey) <= 0 )
         throw new InvalidKeyException(
             "The specified key is not less than the existent one.");

     queue.decreaseKey(node, newKey);
     return oldKey;
}
```


Classe Pública Fila de Fibonacci

```
// If the priority queue contains an entry with the specified value,
// removes and returns that entry. Otherwise, returns null
public Entry<K,V> remove( V value ) {
     FibNode<K,V> node = dict.remove(value);
     if ( node == null )
         return null;
     else {
         queue.remove(node);
         return node.getEntry();
     }
}
```

Classe Pública Fila de Fibonacci

```
// Removes all of the entries from the specified queue and
// inserts them in the mergeable priority queue.
// The two queues must be different and
// their values must be all distinct
public void merge( MergeMinPriorityQueue<K,V> priorityQueue )
        throws EqualPriorityQueuesException {

    if ( this == priorityQueue )
        throw new EqualPriorityQueuesException(
            "The two priority queues are equal");

    if ( priorityQueue instanceof FibonacciQueue )
        . . .
    else
        . . .
}
```


Classe Pública Fila de Fibonacci

```
if ( priorityQueue instanceof FibonacciQueue ) {
    FibonacciQueue<K,V> auxQueue =
            (FibonacciQueue<K,V>) priorityQueue;
    this.mergeDict(auxQueue);
    queue.merge(auxQueue.queue);
}
else
    while ( !priorityQueue.isEmpty() ) {
        Entry<K,V> entry = priorityQueue.removeMin();
        this.insert(entry.getKey(), entry.getValue());
    }
```


Classe Pública Fila de Fibonacci

```
// Removes all entries from the dictionary of the specified queue.
// Values that exist in this Fibonacci queue are removed from the
// specified FibQueue.
// Those that do not exist are inserted in this dictionary
protected void mergeDict( FibonacciQueue<K,V> priQueue ) {
    Iterator<Entry<V, FibNode<K,V>>> iter =
            priQueue.dict.iterator();

    while ( iter.hasNext() ) {
        Entry<V, FibNode<K,V>> entry = iter.next();
        if ( dict.find( entry.getKey() ) == null )
            dict.insert(entry.getKey(), entry.getValue());
        else
            priQueue.queue.remove( entry.getValue() );
    }

    priQueue.dict = new SepChainHashTable<V, FibNode<K,V>>();
}
```


Classe Interna Fila de Fibonacci

```
class FibQueue<K extends Comparable<K>, V> {

   // (Pointer to) a tree with the smallest key
   protected FibNode<K,V> min;

   // Number of entries in the priority queue
   protected int currentSize;

   public FibQueue( ) {
       min = null;
       currentSize = 0;
   }
   . . .
}
```

Métodos Públicos da Classe Interna

- `boolean isEmpty( );`
- `Entry<K,V> minEntry( );`
- `FibNode<K,V> insert( K key, V value );`
- `Entry<K,V> removeMin( );`
- `void decreaseKey( FibNode<K,V> node, K newKey );`
- `void remove( FibNode<K,V> node );`
- `void merge( FibQueue<K,V> queue );`

Descrição das Operações (n entradas)

- `boolean isEmpty( )`
  - Testar se min é `null`
  - Complexidade — O(1) em todos os casos
- `minEntry( )`
  - Retornar a entrada guardada em `min`
  - Complexidade — O(1) em todos os casos

Classe Interna Fila de Fibonacci

```
// Returns true iff the queue contains no entries
public boolean isEmpty( ) {
     return this.min == null;
}

// Returns an entry with the smallest key in the queue.
// Pre-condition: the queue is not empty
public Entry<K,V> minEntry( ) {
     return this.min.getEntry();
}
```


Descrição das Operações (n entradas)

- `FibNode<K,V> insert( K key, V value )`
  - Criar uma árvore `t` com a entrada (key, value) — O(1)
  - Inserir `t` na fila (por exemplo, à esquerda de min) — O(1)
  - Actualizar `min`, se key for menor que a chave em `min`— O(1)
  - Retornar `t` — O(1)

  Complexidade — O(1) em todos os casos.

Classe Interna Fila de Fibonacci

```
// Inserts the entry (key, value) in the queue and
// returns the node which contains that entry
public FibNode<K,V> insert( K key, V value )  {
     FibNode<K,V> newTree = new FibNode<K,V>(key, value);

     if (this.isEmpty() )
         this.min = newTree;
     else {
         this.insertTree(min, newTree);
         // Update min
         if ( key.compareTo( this.min.getKey() ) < 0 )
             this.min = newTree;
     }
     this.currentSize++;
     return newTree;
}
```


Descrição das Operações (n entradas)

- `void merge( FibQueue<K,V> queue)`
  - Concatenar a fila com queue (por exemplo, colocando queue à direita de `min`) — O(1)
  - Actualizar `min`, se a chave mínima de queue for menor que a chave em `min`— O(1)

  Complexidade — O(1) em todos os casos

Classe Interna Fila de Fibonacci

```
// Merges this queue with the specified one
// Pre-condition: the two queues are different
public void merge( FibQueue<K,V> queue ) {
     if ( !queue.isEmpty() ) {
         if ( this.isEmpty() )
             this.min = queue.min;
         else {
             // Insert queue to the right of min
             // Update min
             ...
         }
         queue.min = null;
         this.currentSize += queue.currentSize;
         queue.currentSize = 0;
     }
}
```


Classe Interna Fila de Fibonacci

```
     else {
         // Insert queue to the right of min
         FibNode<K,V> firstQ1 = this.min.getRightSibling();
         FibNode<K,V> firstQ2 = queue.min.getRightSibling();
         this.min.setRightSibling(firstQ2);
         firstQ2.setLeftSibling(this.min);
         firstQ1.setLeftSibling(queue.min);
         queue.min.setRightSibling(firstQ1);
         // Update min
         if ( queue.min.getKey().compareTo( this.min.getKey() ) < 0 )
             this.min = queue.min;
     }
```

Descrição das Operações (n entradas)

- `Entry<K,V> removeMin( )`
  - Guardar a entrada `e` guardada em `min` — O(1)
  - Para cada filho de `min` , colocar a marca a false, colocar pai a null e inserir na fila (d é grau de `min`) — O(d)
  - Retira-se a árvore `min` da fila, ficando `min` a apontar para uma árvore qualquer (ou a null, se a fila ficar vazia) — O(1)
  - Consolidar a fila — `this.consolidate()`
  - Retornar `e`

  Complexidade — d + consolidação

Classe Interna Fila de Fibonacci

```
// Removes an entry with the smallest key from the queue
// and returns that entry
// Pre-condition: the queue is not empty
public Entry<K,V> removeMin( ) {
     Entry<K,V> entry = this.min.getEntry();
     this.insertChildren( this.min.getChild() );
     this.removeMinTree();
     if (this.min != null )
         this.consolidate();
     this.currentSize--;
     return entry;
}
```


Classe Interna Fila de Fibonacci

```
protected void insertChildren( FibNode<K,V> firstChild ) {
     if ( firstChild != null ) {
         FibNode<K,V> tree = firstChild;
         do {
             FibNode<K,V> nextTree = tree.getRightSibling();
             tree.setParent(null);
             tree.unmark();
             this.insertTree(this.min, tree);
             tree = nextTree;
         } while ( tree != firstChild );
     }
}
```


Classe Interna Fila de Fibonacci

```
protected void consolidate( ) {
     FibNode<K,V>[] trees = this.buildTrees();
     this.rebuildQueue(trees);
}
```

Classe Interna Fila de Fibonacci

```
// Returns an array with the new queue trees (indexed by degree)
// Pre-condition: the queue is not empty
protected FibNode<K,V>[] buildTrees( ) {
     int capacity = this.maxDegree() + 1;
     FibNode<K,V>[] trees = (FibNode<K,V>[]) new FibNode[capacity];
     FibNode<K,V> currTree = this.min;
     do {
         ...
     } while ( currTree != this.min );
     return trees;
}
```


Classe Interna Fila de Fibonacci

```
     FibNode<K,V> currTree = this.min;
     do {
         int currDegree = currTree.getDegree();
         FibNode<K,V> nextTree = currTree.getRightSibling();
         while ( trees[currDegree] != null ) {
             currTree = this.linkTrees(currTree, trees[currDegree]);
             trees[currDegree] = null;
             currDegree++;
         }
         trees[currDegree] = currTree;
         currTree = nextTree;
     } while ( currTree != this.min );
```


Complexidade da Consolidação

- Sejam **t** o número de árvores na fila e **d** o grau da árvore mínima
- Ciclo exterior (do-while) de buildTrees: cada passo é constante e o número de passos é t+d–1 — O(t+d)
- Ciclo interior (while) de buildTrees: cada passo é constante e o número de passos não excede t+d–2 — O(t+d)
- Complexidade de rebuildQueue: cada passo é constante e o número de passos não excede t+d–1 — O(t+d)

Complexidade Total — O(t+d)

Complexidade da Remoção do Mínimo

- Sejam **t** o número de árvores na fila e **d** o grau da árvore mínima
- Complexidade de insertChildren — O(d)
- Complexidade de removeMinTree — O(1)
- Complexidade de consolidate — O(t+d)

Complexidade Total — O(t+d)

Como t pode ser n e d < n, a complexidade é O(n)

Descrição das Operações (n entradas)

- `void decreaseKey(FibNode<K,V> node, K newKey)`
  1. Se (node é raiz ou newKey ≥ chave do pai), passar a (3)
  2. No caso contrário, cortar node: remover node da lista dos filhos do pai; colocar o pai de node a null; colocar a marca de node a false; e inserer node na fila
     - Se o pai é raiz, passar a (3)
     - Se o pai não é raiz e não está marcado, coloca-se a marca do pai a true. Passar a (3)
     - Se o pai está marcado, (o pai não é raiz) cortar pai
  3. Actualizar min, se newKey for menor que a chave em min

  Complexidade — altura da árvore, no pior caso

Classe Interna Fila de Fibonacci

```
// Replaces the key in the specified node by the specified key
// Pre-condition: the specified key is less than the one in the node
public void decreaseKey( FibNode<K,V> node, K newKey ) {
     node.setKey(newKey);
     FibNode<K,V> parent = node.getParent();
     if ( parent != null && parent.getKey().compareTo(newKey) > 0 ) {
         this.cut(node, parent);
         this.cascadingCut(parent);
     }
     // Update min
     if ( newKey.compareTo( this.min.getKey() ) < 0 )
         this.min = node;
 }
```


Classe Interna Fila de Fibonacci

```
// Cuts the link between the specified node and its parent
// and makes the specified node a queue root.
// The parent's degree is decreased but its mark is not changed
protected void cut( FibNode<K,V> node, FibNode<K,V> parent ) {
     this.removeChild(parent, node);
     node.setParent(null);
     node.unmark();
     this.insertTree(this.min, node);
}
```


Classe Interna Fila de Fibonacci

```
// Recursively cuts all marked ancestors of the specified node
// (starting with the specified node) until an unmarked node is
// found. Marks that unmarked node, unless it is the root
protected void cascadingCut( FibNode<K,V> node )  {
     FibNode<K,V> parent = node.getParent();
     if ( parent != null )
         if ( node.isMarked() ) {
             this.cut(node, parent);
             this.cascadingCut(parent);
         }
         else
             node.mark();
}
```

Classe Interna Fila de Fibonacci

```
// Removes the entry in the specified node
Entry<K,V> remove( FibNode<K,V> node ) {
     Entry<K,V> entry = node.getEntry();
     this.decreaseKey(node, -∞);
     this.removeMin();
     return entry;
}
```


Filas de Fibonacci

- Propriedade Fundamental
  - Seja x um nó qualquer com grau d (para d > 0). Se y_1, y_2, . . . , y_d forem os filhos de x, pela ordem com que foram ligados a x, então:
    - grau(y_1) ≥ 0
    - grau(y_i) ≥ i – 2, para i = 2, 3, . . . , d.

Grau Máximo de uma Árvore de uma Fila de Fibonacci (n entradas)

Seja d o grau de árvore qualquer da fila, então:

$$n \geq N(d) \geq \phi^d$$

onde ϕ

$$\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$$

Portanto

$$d \leq \lfloor \log_\phi n \rfloor$$

Grau Máximo de uma Árvore de uma Fila de Fibonacci (n entradas)

Seja d o grau de árvore qualquer da fila, então:

$$n \geq N(d) \geq \phi^d$$

onde ϕ

$$\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$$

Portanto

$$d \leq \lfloor \log_\phi n \rfloor$$

Nota: A designação destas filas vem do facto que:

N(d) ≥ Fibonacci(d+2) ≥ ϕ^d

Complexidades da Fila com Prioridade Fundível no Pior Caso (n entradas)

	Heap	Fila Binomial	Fila Fibonacci
`isEmpty`	O(1)	O(1)	O(1)
`size`	O(1)	O(1)	O(1)
`minEntry`	O(1)	O(log n)	O(1)
`insert`	O(log n)	O(log n)	O(1)
`removeMin`	O(log n)	O(log n)	O(n)
`decreaseKey`	O(log n)	O(log n)	O(n)
`remove`	O(log n)	O(log n)	O(n)
`merge` (m entradas)	O(n + m)	O(log (n+m))	O(1)

Complexidade Amortizada

Seja Q uma fila de Fibonacci qualquer, t_Q o número de árvores em Q e m_Q o número de nós marcados em Q

$$\phi(Q) = t_Q + 2m_Q$$

P1. $\phi(Q_0) = 0$ onde Q_0 é a fila de fibonacci vazia

P2. $\phi(Q) \geq 0$

- Custo amortizado da operação i:

$$\hat{c}_i = c_i + \phi(Q_i) - \phi(Q_{i-1})$$

Complexidade Amortizada

Seja Q uma fila de Fibonacci qualquer, t_Q o número de árvores em Q e m_Q o número de nós marcados em Q

$$\phi(Q) = t_Q + 2m_Q$$

Operação	Custo Real c_i	Dif. de Potencial $\Delta_i = \phi(Q_i) - \phi(Q_{i-1})$	Complexidade Amortizada
`isEmpty`	1	0	O(1)
`minEntry`	1	0	O(1)
`insert`	1	1	O(1)
`merge`	1	0	O(1)

Complexidade Amortizada

Seja Q uma fila de Fibonacci qualquer, t_Q o número de árvores em Q e m_Q o número de nós marcados em Q

$$\phi(Q) = t_Q + 2m_Q$$

Operação	Custo Real c_i	Dif. de Potencial $\Delta_i = \phi(Q_i) - \phi(Q_{i-1})$	Complexidade Amortizada
`removeMin`	`t+d`		

`d é o grau da árvore mínima`

- $\phi(Q_{i-1})$ - Antes da consolidação
  - $\phi(Q_{i-1}) = t + 2m$
- $\phi(Q_i)$ - Depois da consolidação (já não existem várias árvores de um dado grau)
  - $\phi(Q_i) = ?$

```
protected void consolidate( ) {
    FibNode<K,V>[] trees = this.buildTrees();
    this.rebuildQueue(trees);
}

protected FibNode<K,V>[] buildTrees( ) {
    int capacity = this.maxDegree() + 1;
    FibNode<K,V>[] trees = (FibNode<K,V>[]) new FibNode[capacity];
    FibNode<K,V> currTree = this.min;
    do {
        ...
    } while ( currTree != this.min );
    return trees;
}
```


Complexidade Amortizada

Seja Q uma fila de Fibonacci qualquer, t_Q o número de árvores em Q e m_Q o número de nós marcados em Q

$$\phi(Q) = t_Q + 2m_Q$$

Operação	Custo Real c_i	Dif. de Potencial $\Delta_i = \phi(Q_i) - \phi(Q_{i-1})$	Complexidade Amortizada
removeMin	t+d		

d é o grau da árvore mínima

- $\phi(Q_{i-1})$ - Antes da consolidação
  - $\phi(Q_{i-1}) = t + 2m$
- $\phi(Q_i)$ - Depois da consolidação (já não existem várias árvores de um dado grau)
  - $\phi(Q_i) = (d+1) + 2m$

Complexidade Amortizada

Seja Q uma fila de Fibonacci qualquer, t_Q o número de árvores em Q e m_Q o número de nós marcados em Q

$$\phi(Q) = t_Q + 2m_Q$$

Operação	Custo Real c_i	Dif. de Potencial $\Delta_i = \phi(Q_i) - \phi(Q_{i-1})$	Complexidade Amortizada
removeMin	t+d	$\leq \log_\phi(n)+1-t$	$O(\log_\phi(n))$

d é o grau da árvore mínima

- $\phi(Q_{i-1})$ - Antes da consolidação
  - $\phi(Q_{i-1}) = t + 2m$
- $\phi(Q_i)$ - Depois da consolidação (já não existem várias árvores de um dado grau)
  - $\phi(Q_i) = (d+1) + 2m$

Complexidade Amortizada

Seja Q uma fila de Fibonacci qualquer, t_Q o número de árvores em Q e m_Q o número de nós marcados em Q

$$\phi(Q) = t_Q + 2m_Q$$

Operação	Custo Real c_i	Dif. de Potencial $\Delta_i = \phi(Q_i) - \phi(Q_{i-1})$	Complexidade Amortizada
decreaseKey	1+k		

k é o número de cortes efectuados

- $\phi(Q_{i-1})$
  - $\phi(Q_{i-1}) = t + 2m$
- $\phi(Q_i)$
  - $t' = t + k$
  - $m' = m - (k-1) + 1$ **(k-1 nós desmarcados, a última chamada pode marcar um nó)**

Complexidade Amortizada

Seja Q uma fila de Fibonacci qualquer, t_Q o número de árvores em Q e m_Q o número de nós marcados em Q

$$\phi(Q) = t_Q + 2m_Q$$

Operação	Custo Real c_i	Dif. de Potencial $\Delta_i = \phi(Q_i) - \phi(Q_{i-1})$	Complexidade Amortizada
decreaseKey	1+k	4-k	O(1)

k é o número de cortes efectuados

- $\phi(Q_{i-1})$
  - $\phi(Q_{i-1}) = t + 2m$
- $\phi(Q_i)$
  - $t' = t + k$
  - $m' = m - (k-1) + 1$ **(k-1 nós desmacados, a última chamada pode marcar um nó)**

Complexidades da Fila com Prioridade Fundível no Pior Caso e Amortizada (n entradas)

	Heap	Fila Binomial	Fila Fibonacci
isEmpty	O(1)	O(1)	O(1)
size	O(1)	O(1)	O(1)
minEntry	O(1)	O(log n)	O(1)
insert	O(log n)	O(log n)	O(1)
removeMin	O(log n)	O(log n)	O(log n)
decreaseKey	O(log n)	O(log n)	O(1)
remove	O(log n)	O(log n)	O(log n)
merge (m entradas)	O(n + m)	O(log (n+m))	O(1)

Complexidade dos Algoritmos de Prim e Dijkstra com Heap ou Fila Binomial (grafo em listas de adjacências)

Criar fila	O(#V)
Inicializar 2 vectores	O(#V)
Inserir origem na fila	O(1)
(#V ou ≤ #V) remover mínimo da fila	O(#V log #V)
(#V ou ≤ #V) percorrer sucessores directos	O(#A)
(#A ou ≤ #A) inserir na fila ou decrementar chave	O(#A ~~log #V~~)

TOTAL Prim ou Dijkstra O(#A + #V log #V)

Classe Interna – métodos a completar

- `protected int maxDegree()`
- `protected void insertTree( FibNode<K,V> head, FibNode<K,V> newTree );`
- `protected void removeMinTree( );`
- `protected void rebuildQueue( FibNode<K,V>[] trees );`
- `protected FibNode<K,V> linkTrees( FibNode<K,V> tree1, FibNode<K,V> tree2 );`
- `protected void removeChild( FibNode<K,V> parent, FibNode<K,V> child );`

Classe Interna Fila de Fibonacci

```
class FibQueue<K extends Comparable<K>, V> {

   static final double GOLDEN_RATIO = 1.618;

   // (Pointer to) a tree with the smallest key
   protected FibNode<K,V> min;

   // Number of entries in the priority queue
   protected int currentSize;

   . . .

   // Returns the max degree of a tree in the queue.
   protected int maxDegree( ) {
        return (int) ( Math.log(currentSize)/Math.log(GOLDEN_RATIO) );
   }
}
```


Classe Interna Fila de Fibonacci

```
// Inserts the specified tree in the list,
// to the left of the specified head
// Pre-condition: the queue is not empty
protected void insertTree( FibNode<K,V> head, FibNode<K,V> newTree ){
     // TO DO
}

// Removes the head of the queue and updates min.
// If the queue becomes empty, min is set to null;
// otherwise, min points to one of the remaining trees
// Pre-condition: the queue is not empty
protected void removeMinTree( ) {
     // TO DO
}
```


Classe Interna Fila de Fibonacci

```
// Rebuilds the queue with the trees in the specified array
// Pre-condition: the queue is not empty
protected void rebuildQueue( FibNode<K,V>[] trees ) {
     // TO DO
}

// Links two trees of degree k and
// returns the resulting tree of degree k + 1
protected FibNode<K,V> linkTrees( FibNode<K,V> tree1,
                                  FibNode<K,V> tree2 ) {
     // TO DO
}
```


Classe Interna Fila de Fibonacci

```
// Removes the specified child from the list of children
// of the specified parent
protected void removeChild( FibNode<K,V> parent,
                            FibNode<K,V> child ) {
     // TO DO
}
```

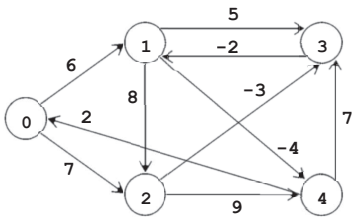
12. Caminho Mais Curto de um vértice a todos os outros Algoritmo de Bellman-Ford

Planeamento

Apresentação da disciplina
Programação Dinâmica
Introdução aos Grafos
Percurso em largura e profundidade
Ordenação Topológica
Árvore Mínima de Cobertura (Algoritmo de Kruskal)
TAD Partição
Complexidade Amortizada
Árvore Mínima de Cobertura (Algoritmo de Prim)
TAD Fila com Prioridade Adaptável
Algoritmo de Dijskstra
Filas Binomiais
Filas de Fibonacci
Algoritmo de Bellman-Ford
Fluxo Máximo: Algoritmo de Ford-Fulkerson
Algoritmo de Edmonds-Karp
Introdução à Teoria da Complexidade
Exemplos de Problemas NP
Redutibilidade entre Problemas de Decisão

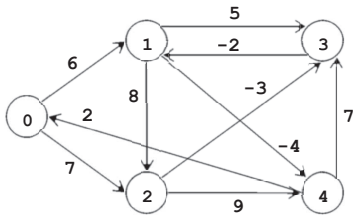
Algoritmo de Bellman-Ford

- Dado um grafo orientado e pesado e um vértice 0, determina um caminho mais curto de 0 para qualquer vértice v
  - Este algoritmo pode ser aplicado a grafos com pesos dos arcos negativos



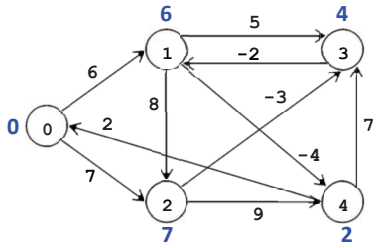
Algoritmo de Dijkstra

- Não pode ser aplicado a grafos com arcos negativos



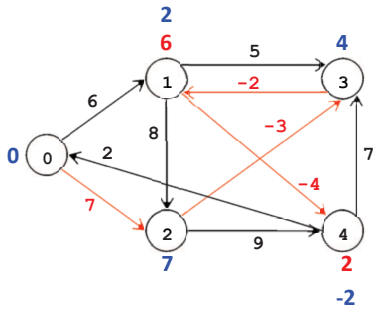
Algoritmo de Dijkstra

- Não pode ser aplicado a grafos com arcos negativos



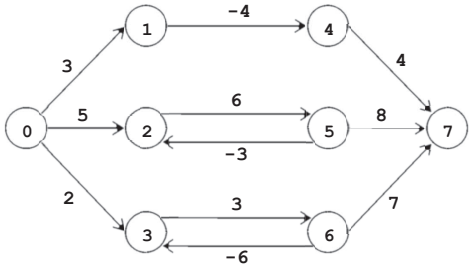
Algoritmo de Dijkstra

- Não pode ser aplicado a grafos com arcos negativos



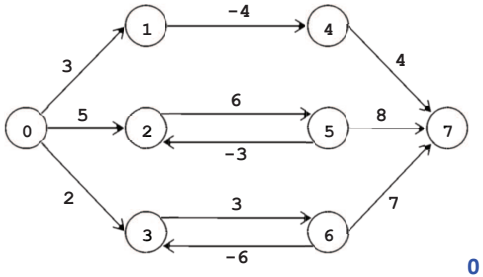
Algoritmo de Bellman-Ford

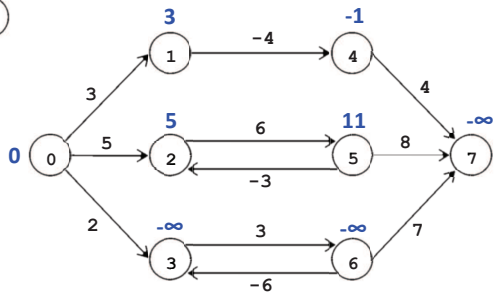
- O que acontece quando existem ciclos negativos?



Algoritmo de Bellman-Ford

- O que acontece quando existem ciclos negativos?





Caminho Simples

- Se existe algum caminho de o para v e o grafo não tem ciclos de peso negativo acessíveis a partir de o, então:
  - existe um caminho mais curto de o para v
  - esse caminho tem, no máximo, #V vértices e #V – 1 arcos

Propriedade: Caminho mais curto

- Se

  $o, w_1, w_2, \ldots, w_n, v$ (com $n \geq o$),

  é um caminho mais curto de o para v, então

  $o, w_1, w_2, \ldots, w_n$

  é um caminho mais curto de o para w_n

Problema

- Para todos os vértices v, determinar o comprimento dos caminhos mais curtos de o para v que têm, no máximo, #V – 1 arcos

Problema

- Para todos os vértices v, determinar o comprimento dos caminhos mais curtos de o para v que têm, no máximo, #V – 1 arcos

Programação Dinâmica

Programação Dinâmica: Metodologia

- Etapas para a construção de um algoritmo baseado em programação dinâmica:
  1. Caracterizar a estrutura da solução óptima
  2. Definir recursivamente o valor de uma solução óptima, em função de soluções óptimas de subproblemas
  3. Calcular o valor da solução óptima segundo uma estratégia *bottom-up*
  4. Construir solução a partir da informação obtida

Definir Solução Óptima

- Construção da função recursiva L(v,i)
  - Caminho com zero arcos (i = 0)
    - Se v = 0, então L(v,0) = 0
    - Se v ≠ 0, então L(v,0) = +∞

Definir Solução Óptima

- Construção da função recursiva L(v,i)
  - Caminho com zero arcos (i = 0)
    - Se v = 0, então L(v,0) = 0
    - Se v ≠ 0, então L(v,0) = +∞
  - Caminho com um ou mais arcos (i>0)
    - O caminho tem, no máximo, i-1 arcos, nesse caso
      L(v,i) = L(v,i-1)
    - O caminho tem, no máximo, i arcos, sendo que o último arco é (w,v), nesse caso
      L(w,i-1) + peso(w,v)

Definir Solução Óptima

- Construção da função recursiva L(v,i)

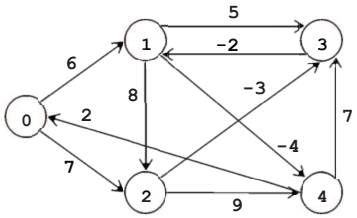
$$L(v,i) = \begin{cases} 0 & se \quad i = 0 \wedge v = 0 \\ +\infty & se \quad i = 0 \wedge v \neq 0 \\ \min(L(v,i-1), \min_{\{w \mid (w,v) \in A\}} (L(w,i-1) + peso(w,v))) & se \quad i > 0 \end{cases}$$

Cálculo da Solução Óptima

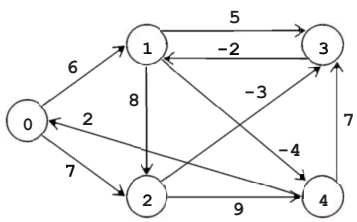
$$L(v,i) = \begin{cases} 0 & se \quad i = 0 \wedge v = 0 \\ +\infty & se \quad i = 0 \wedge v \neq 0 \\ \min(L(v,i-1), \min_{\{w|(w,v)\in A\}}(L(w,i-1) + peso(w,v))) & se \quad i > 0 \end{cases}$$

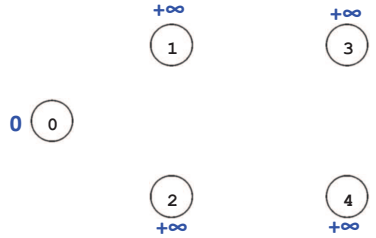
comprimento

vértices	0	1	2	3	4
0	0	0	0	0	0
1	+∞	6	6	2	2
2	+∞	7	7	7	7
3	+∞	+∞	4	4	4
4	+∞	+∞	2	2	-2

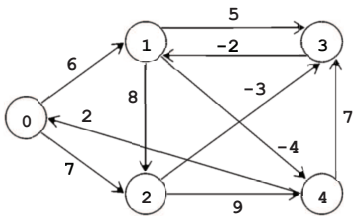


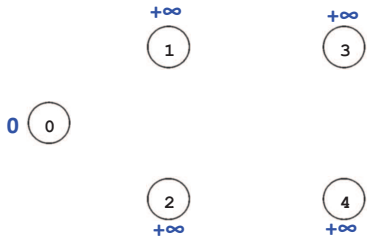
Simulação do Algoritmo (i = 1)

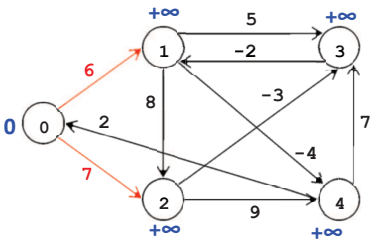


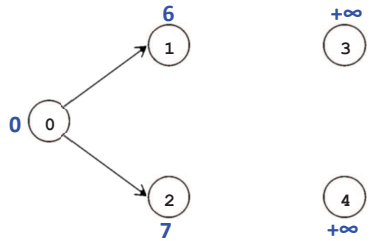


Simulação do Algoritmo (i = 1)

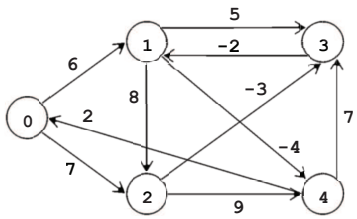


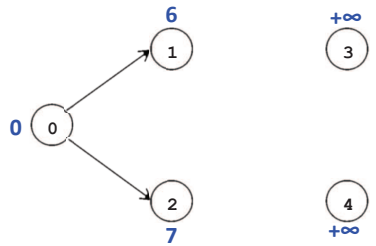




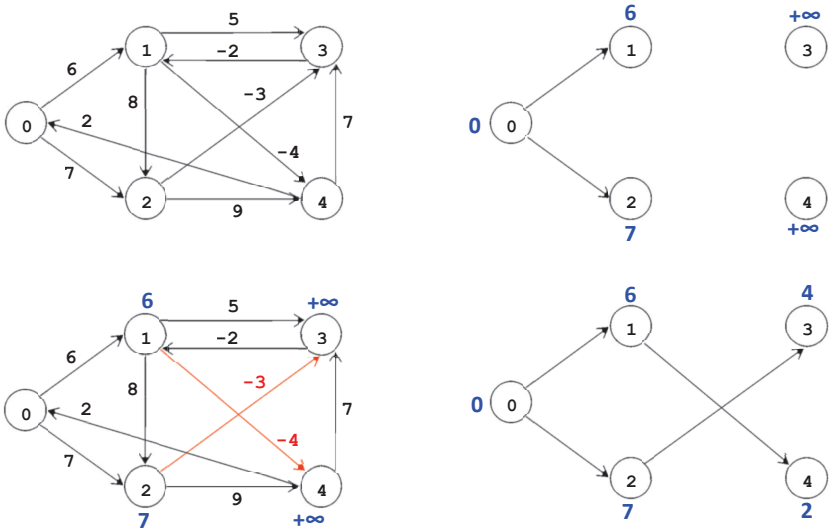


Simulação do Algoritmo (i = 2)

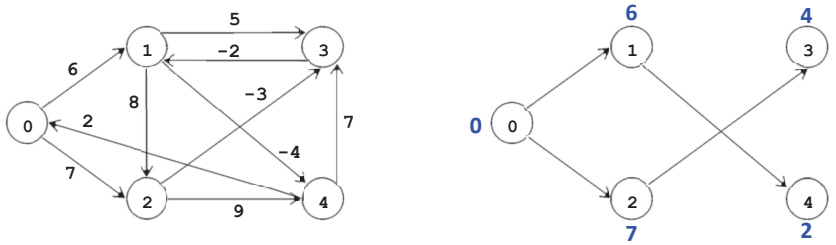




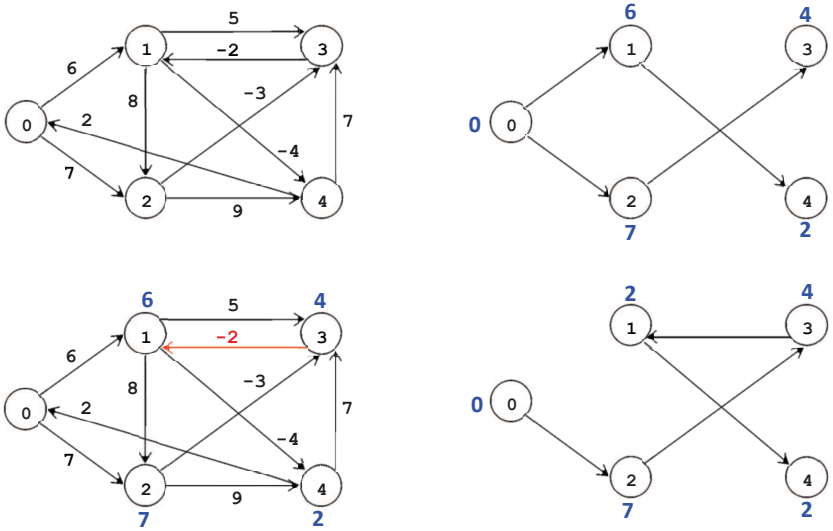
Simulação do Algoritmo (i = 2)



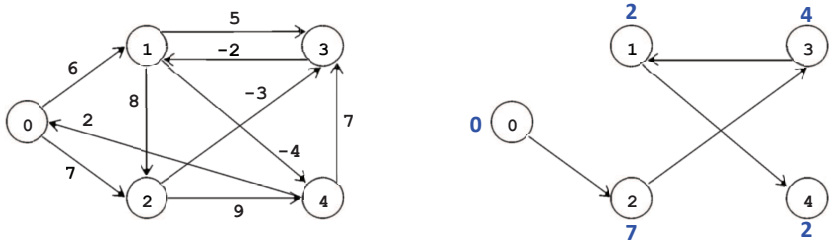
Simulação do Algoritmo (i = 3)



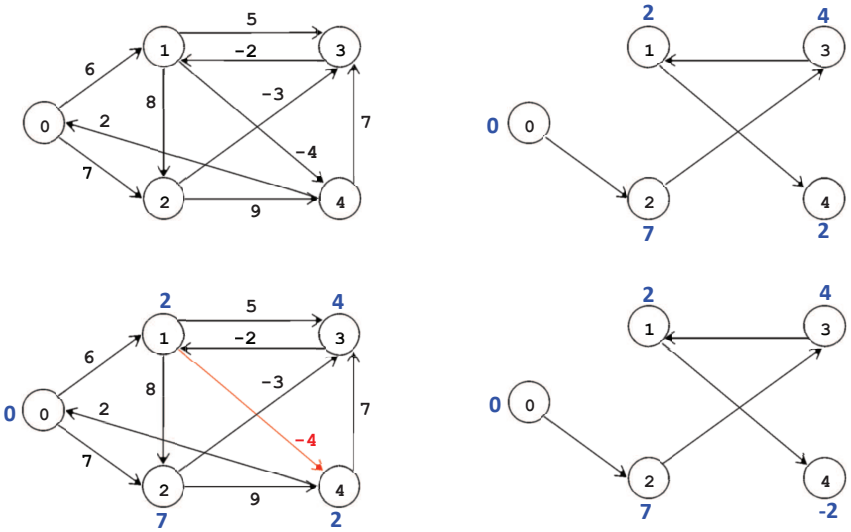
Simulação do Algoritmo (i = 3)



Simulação do Algoritmo (i = 4)



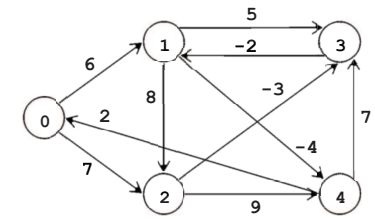
Simulação do Algoritmo (i = 4)



Algoritmo de Bellman-Ford

- Em vez de guardar os valores da função L num vector de dimensão #V×#V utiliza-se um vector de #V posições

Simulação dos Algoritmos



Vector Bidimensional

	0	1	2	3	4
0	0	0	0	0	0
1	+∞	6	6	2	2
2	+∞	7	7	7	7
3	+∞	+∞	4	4	4
4	+∞	+∞	2	2	-2

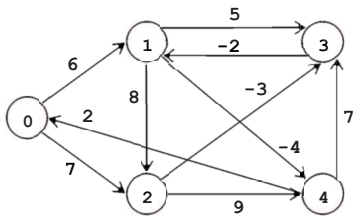
Ordem Unidimensional

```
(0,1)   (2,3)
(0,2)   (2,4)
(1,2)   (3,1)
(1,3)   (4,0)
(1,4)   (4,3)
```

Vector Unidimensional

	0	1	2	3	4
0	0				
1	+∞				
2	+∞				
3	+∞				
4	+∞				

Simulação dos Algoritmos



Vector Bidimensional

	0	1	2	3	4
0	0	0	0	0	0
1	+∞	6	6	2	2
2	+∞	7	7	7	7
3	+∞	+∞	4	4	4
4	+∞	+∞	2	2	-2

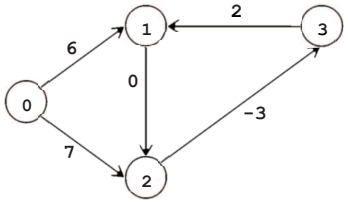
Ordem Unidimensional

```
(0,1)   (2,3)
(0,2)   (2,4)
(1,2)   (3,1)
(1,3)   (4,0)
(1,4)   (4,3)
```

Vector Unidimensional

	0	1	2	3	4
0	0	0	0	0	0
1	+∞	4	2	2	2
2	+∞	7	7	7	7
3	+∞	4	4	4	4
4	+∞	4	0	-2	-2

Simulação dos Algoritmos



(0,1) (2,3)
(0,3) (3,1)
(1,2)

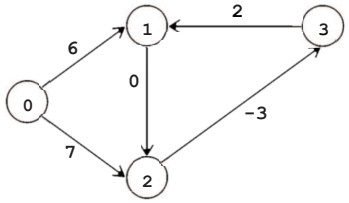
Vector Bidimensional

	0	1	2	3	4	5
0	0	0	0	0	**0**	**0**
1	+∞	6	6	6	**5**	**5**
2	+∞	7	6	6	**6**	**5**
3	+∞	+∞	4	3	**3**	**3**

Vector Unidimensional

	0	1	2	3	4	5
0	0	0	0	0	**0**	**0**
1	+∞	6	5	4	**3**	**2**
2	+∞	6	5	4	**1**	**0**
3	+∞	4	3	2	**3**	**2**

Simulação dos Algoritmos



(0,1) (2,3)
(0,3) (3,1)
(1,2)

Vector Bidimensional

	0	1	2	3	4	5
0	0	0	0	0	**0**	**0**
1	+∞	6	6	6	**5**	**5**
2	+∞	7	6	6	**6**	**5**
3	+∞	+∞	4	3	**3**	**3**

Vector Unidimensional

	0	1	2	3	4	5
0	0	0	0	0	**0**	**0**
1	+∞	6	5	4	**3**	**2**
2	+∞	6	5	4	**1**	**0**
3	+∞	4	3	2	**3**	**2**

Caminho Mais Curto (Bellman-Ford)

```
Pair<E[], Vertex[]> bellmanFord( Digraph<?,E> graph, Vertex origin )
    throws NegativeWeightCycleException {

    E[] length = new E[ graph.numVertices() ];
    Vertex[] via = new Vertex[ graph.numVertices() ];
    for every Vertex v in graph.vertices()
        length[v] = +∞;
    length[origin] = 0;
    via[origin] = origin;
```


Caminho Mais Curto (Bellman-Ford)

```
    boolean changes = false;
    for ( int i = 1; i < graph.numVertices(); i++ )  {
        changes = updateLengths(graph, length, via);
        if ( !changes )
            break;
    }
    // Negative-weight cycles detection
    if ( changes && updateLengths(graph, length, via) )
        throw new NegativeWeightCycleException();
    else
        return new PairClass<E[], Vertex[]>(length, via);
}
```

Caminho Mais Curto (Bellman-Ford)

```
boolean updateLengths( Digraph<?,E> graph,E[] length, Vertex[] via ){
    boolean changes = false;
    for every Edge<?,E> e in graph.edges() {
        Vertex[] endPoints = e.endVertices();
        if ( length[endPoints[0]] < +∞ ) {
            E newLength = length[endPoints[0]] + e.label();
            if ( newLength < length[endPoints[1]] ) {
                length[endPoints[1]] = newLength;
                via[endPoints[1]] = endPoints[0];
                changes = true;
            }
        }
    }
   return changes;
}
```


Complexidade

Implementação	Algoritmo Bellman-Ford
Matriz de Adjacências	$O((\#V)^3)$
Lista de Adjacências	$O(\#V \#A)$

Problema

- Calcular o caminho mais curto entre cada par de vértices

Problema

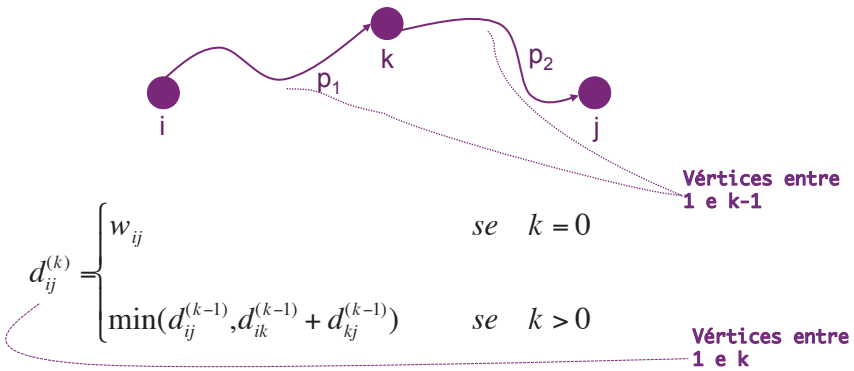
- Calcular o caminho mais curto entre cada par de vértices

$$w_{ij} = \begin{cases} 0 & \text{se } i = j \\ \text{peso do arco (i,j)} & \text{se } i \neq j \text{ e } (i,j) \in A \\ \infty & \text{se } i \neq j \text{ e } (i,j) \notin A \end{cases}$$

$d_{ij}^{(k)}$ é o peso do caminho mais curto entre i e j com vértices intermédios retirados do conjunto de vértices {1,…,k}

Problema

- Calcular o caminho mais curto entre cada par de vértices



$$d_{ij}^{(k)} = \begin{cases} w_{ij} & se \quad k = 0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & se \quad k > 0 \end{cases}$$

Algoritmo Floyd-Warshall

```
int[][] functionD( int[][] W, int N) {
   int[][][] tableD = new int[N+1][N+1][N+1];

   for (int i=0; i ≤ N; i++) //Linha 0: Base da recursivade
        for (int j=0; j ≤ N; j++)
            tableD[0][i][j] = W[i][j];


   for (int k=1; k ≤ N; k++) //Caso geral
        for (int i=1; i ≤ N; i++)
            for (int j=1; j ≤ N; j++) {
                int value = tableD[k-1][i][k] + tableD[k-1][k][j]
                if (tableD[k-1][i][j] < value)
                    tableD[k][i][j] = value;
                else
                    tableD[k][i][j] = tableD[k-1][i][j];
            }

   return tableD[N];
}
```


Algoritmo Floyd-Warshall

```
int[][] functionD( int[][] W, int N) {
   int[][] tableD = new int[N+1][N+1];

   for (int i=0; i ≤ N; i++) //Linha 0: Base da recursivade
        for (int j=0; j ≤ N; j++)
            tableD[i][j] = W[i][j];


   for (int k=1; k ≤ N; k++) //Caso geral
        for (int i=1; i ≤ N; i++)
            for (int j=1; j ≤ N; j++) {
                int value = tableD[i][k] + tableD[k][j]
                if (tableD [i][j] < value)
                        tableD[i][j] = value;


   return tableD;
}
```

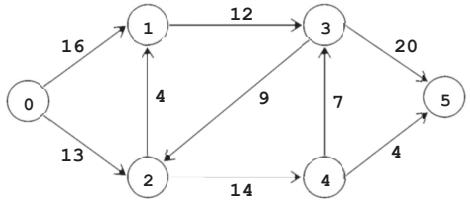

13. Fluxo Máximo entre Dois Vértices

Planeamento

Apresentação da disciplina
Programação Dinâmica
Introdução aos Grafos
Percurso em largura e profundidade
Ordenação Topológica
Árvore Mínima de Cobertura (Algoritmo de Kruskal)
TAD Partição
Complexidade Amortizada
Árvore Mínima de Cobertura (Algoritmo de Prim)
TAD Fila com Prioridade Adaptável
Algoritmo de Dijskstra
Filas Binomiais
Filas de Fibonacci
Algoritmo de Bellman-Ford
Fluxo Máximo: Método de Ford-Fulkerson
Algoritmo de Edmonds-Karp
Introdução à Teoria da Complexidade
Exemplos de Problemas NP
Redutibilidade entre Problemas de Decisão

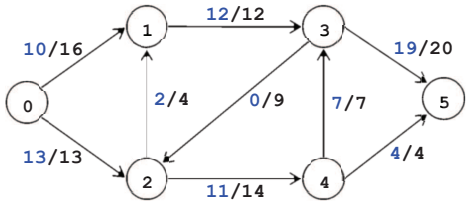
Fluxo Máximo entre Dois Vértices

- Dado um grafo orientado e pesado e dois vértice f (fonte) e d (dreno), determinar o fluxo máximo de f para d
  - Assume-se que qualquer vértices pertencem a um caminho de f para d (`#A ≥ #V–1`)



Fluxo Máximo entre Dois Vértices

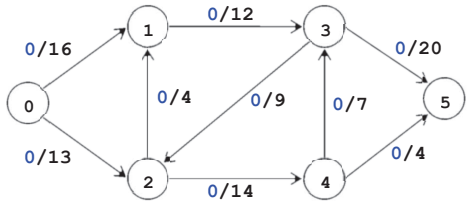
- Dado um grafo orientado e pesado e dois vértice f (fonte) e d (dreno), determinar o fluxo máximo de f para d
  - Assume-se que qualquer vértices pertencem a um caminho de f para d (`#A ≥ #V–1`)



Fluxo Máximo do Vértice 0 para o Vértice 5: 23

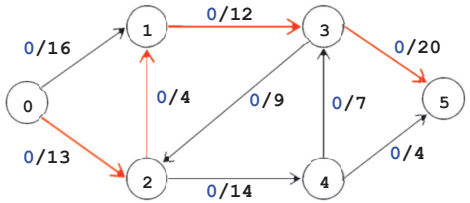
Fluxo Máximo entre Dois Vértices (Tentativa)

- Descobrir um caminho da fonte ao dreno
- Determinar o incremento máximo de fluxo permitido
- Actualizar fluxo



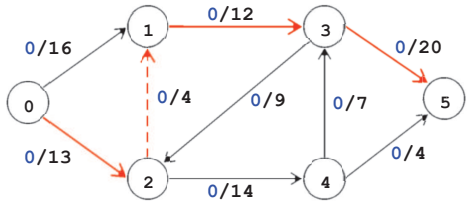
Fluxo Máximo entre Dois Vértices (Tentativa)

- Descobrir um caminho da fonte ao dreno ✔
  - `0, 2, 1, 3, 5`
- Determinar o incremento máximo de fluxo permitido
- Actualizar fluxo



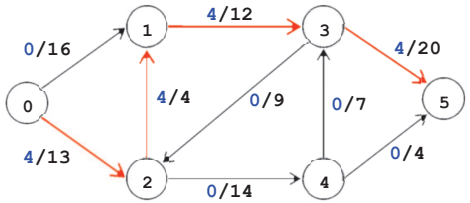
Fluxo Máximo entre Dois Vértices (Tentativa)

- Descobrir um caminho da fonte ao dreno ✔
  - `0, 2, 1, 3, 5`
- Determinar o incremento máximo de fluxo permitido ✔
  - `4`
- Actualizar fluxo



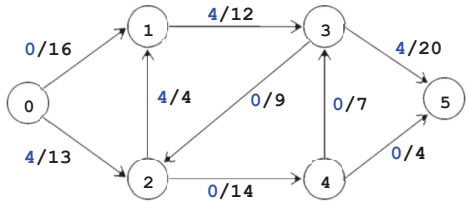
Fluxo Máximo entre Dois Vértices (Tentativa)

- Descobrir um caminho da fonte ao dreno ✔
  - `0, 2, 1, 3, 5`
- Determinar o incremento máximo de fluxo permitido ✔
  - `4`
- Actualizar fluxo ✔



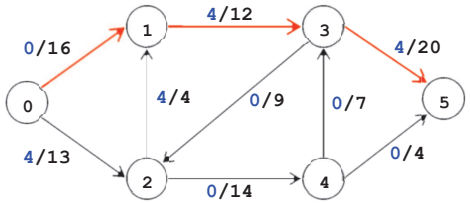
Fluxo Máximo entre Dois Vértices (Tentativa)

- Descobrir um caminho da fonte ao dreno
- Determinar o incremento máximo de fluxo permitido
- Actualizar fluxo



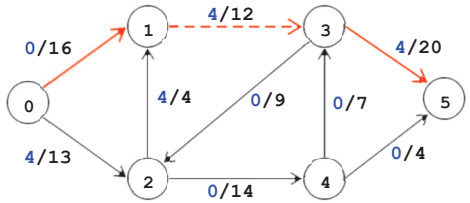
Fluxo Máximo entre Dois Vértices (Tentativa)

- Descobrir um caminho da fonte ao dreno ✔
  - `0, 1, 3, 5`
- Determinar o incremento máximo de fluxo permitido
- Actualizar fluxo



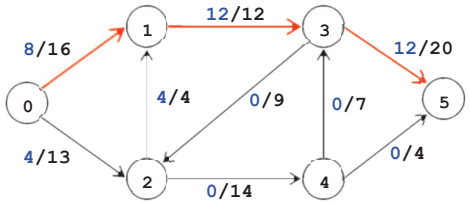
Fluxo Máximo entre Dois Vértices (Tentativa)

- Descobrir um caminho da fonte ao dreno ✔
  - `0, 1, 3, 5`
- Determinar o incremento máximo de fluxo permitido ✔
  - `8`
- Actualizar fluxo



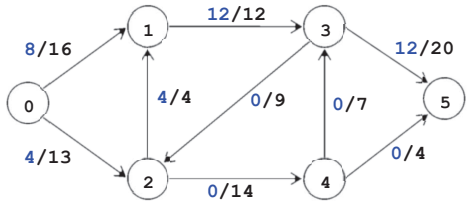
Fluxo Máximo entre Dois Vértices (Tentativa)

- Descobrir um caminho da fonte ao dreno ✔
  - `0, 1, 3, 5`
- Determinar o incremento máximo de fluxo permitido ✔
  - `8`
- Actualizar fluxo ✔



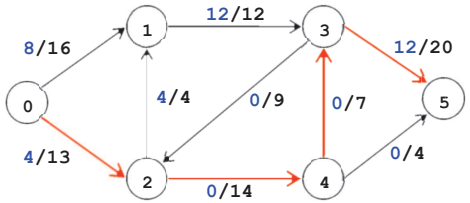
Fluxo Máximo entre Dois Vértices (Tentativa)

- Descobrir um caminho da fonte ao dreno
- Determinar o incremento máximo de fluxo permitido
- Actualizar fluxo



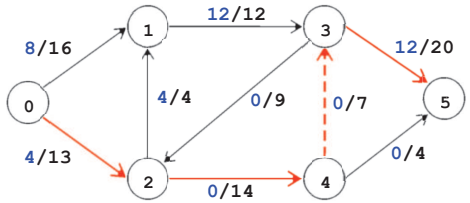
Fluxo Máximo entre Dois Vértices (Tentativa)

- Descobrir um caminho da fonte ao dreno ✔
  - `0, 2, 4, 3, 5`
- Determinar o incremento máximo de fluxo permitido
- Actualizar fluxo



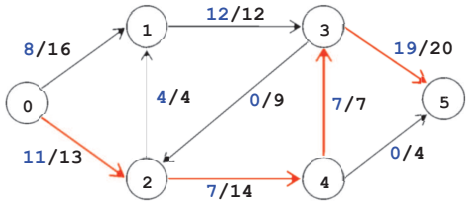
Fluxo Máximo entre Dois Vértices (Tentativa)

- Descobrir um caminho da fonte ao dreno ✔
  - `0, 2, 4, 3, 5`
- Determinar o incremento máximo de fluxo permitido ✔
  - `7`
- Actualizar fluxo



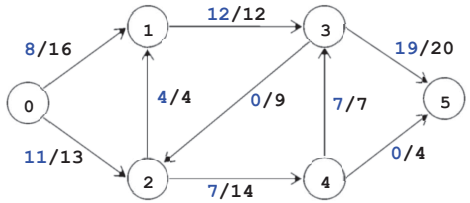
Fluxo Máximo entre Dois Vértices (Tentativa)

- Descobrir um caminho da fonte ao dreno ✔
  - `0, 2, 4, 3, 5`
- Determinar o incremento máximo de fluxo permitido ✔
  - `7`
- Actualizar fluxo ✔



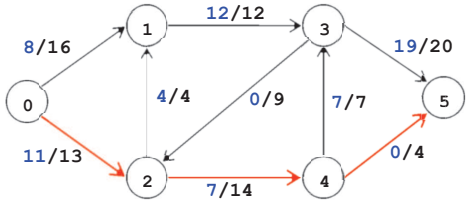
Fluxo Máximo entre Dois Vértices (Tentativa)

- Descobrir um caminho da fonte ao dreno
- Determinar o incremento máximo de fluxo permitido
- Actualizar fluxo



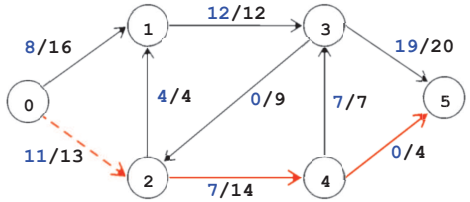
Fluxo Máximo entre Dois Vértices (Tentativa)

- Descobrir um caminho da fonte ao dreno ✔
  - `0, 2, 4, 5`
- Determinar o incremento máximo de fluxo permitido
- Actualizar fluxo



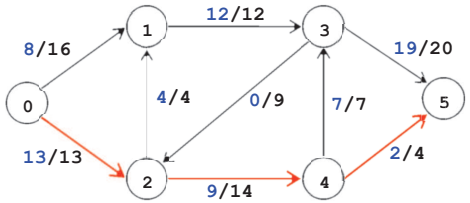
Fluxo Máximo entre Dois Vértices (Tentativa)

- Descobrir um caminho da fonte ao dreno ✔
  - `0, 2, 4, 5`
- Determinar o incremento máximo de fluxo permitido ✔
  - `2`
- Actualizar fluxo



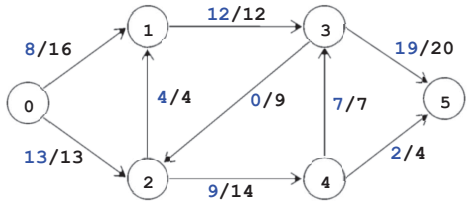
Fluxo Máximo entre Dois Vértices (Tentativa)

- Descobrir um caminho da fonte ao dreno ✔
  - `0, 2, 4, 5`
- Determinar o incremento máximo de fluxo permitido ✔
  - `2`
- Actualizar fluxo ✔



Fluxo Máximo entre Dois Vértices (Tentativa)

- Não há caminho da fonte ao dreno para o qual seja possível aumentar o fluxo
- Contudo, o valor de fluxo máximo é superior a 21

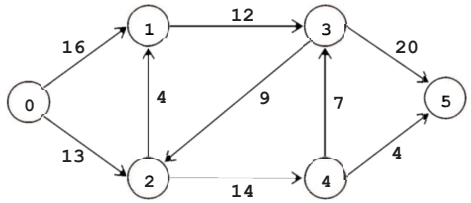


Rede de Fluxos

- Seja G=(V,A) um grafo orientado e pesado, cujos arcos tem pesos não negativos. A rede de fluxos de G é um grafo R=(V,A'), orientado e pesado, tal que:
  - A ⊆ A' (todos os arcos de G estão na rede R)
  - Se (u,v) ∈ A e (v,u) ∉ A, então (v,u) ∈ A' e tem peso 0

Rede de Fluxos

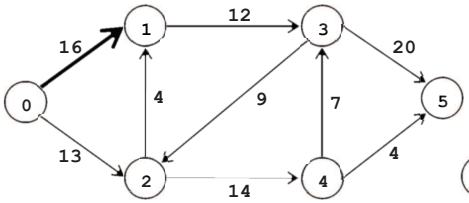
- Seja G=(V,A) um grafo orientado e pesado, cujos arcos tem pesos não negativos. A rede de fluxos de G é um grafo R=(V,A'), orientado e pesado, tal que:
  - A ⊆ A' (todos os arcos de G estão na rede R)
  - Se (u,v) ∈ A e (v,u) ∉ A, então (v,u) ∈ A' e tem peso 0



G=(V,A)

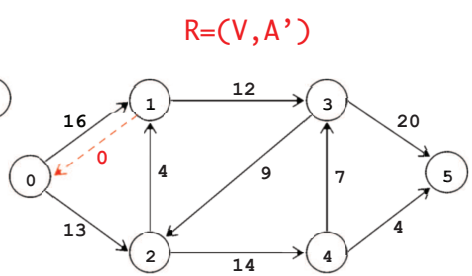
Rede de Fluxos

- Seja G=(V,A) um grafo orientado e pesado, cujos arcos tem pesos não negativos. A rede de fluxos de G é um grafo R=(V,A'), orientado e pesado, tal que:
  - A ⊆ A' (todos os arcos de G estão na rede R)
  - Se (u,v) ∈ A e (v,u) ∉ A, então (v,u) ∈ A' e tem peso 0



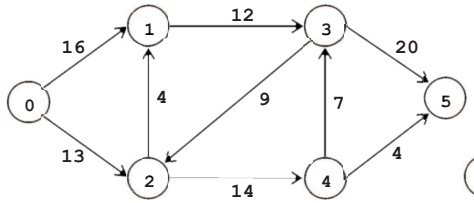
G=(V,A)

R=(V,A')



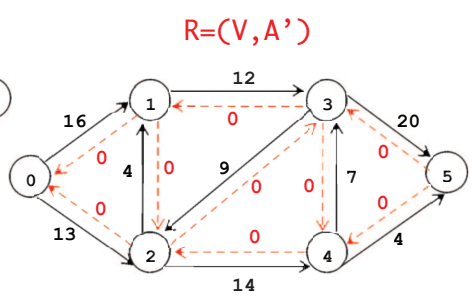
Rede de Fluxos

- Seja G=(V,A) um grafo orientado e pesado, cujos arcos tem pesos não negativos. A rede de fluxos de G é um grafo R=(V,A'), orientado e pesado, tal que:
  - A ⊆ A' (todos os arcos de G estão na rede R)
  - Se (u,v) ∈ A e (v,u) ∉ A, então (v,u) ∈ A' e tem peso 0



G=(V,A)

R=(V,A')



Fluxo

- Sejam:
  - Seja R=(V,A') uma rede de fluxos
  - f ∈ V o vértice fonte
  - d ∈ V o vértice dreno
  - c(u,v) a capacidade do arco (u,v) ∈ A'
- Um **fluxo** G = (V, E) é uma função $\phi : A' \rightarrow \Re$ tal que:
  - $\phi(u,v) \leq c(u,v)$ para u,v ∈ V (restrição de capacidade)
  - $\phi(u,v) = -\phi(v,u)$ para u,v ∈ V (simetria)
  - Para u ∈ V\{f,d}: $\sum_{\{v|(u,v)\in A'\}} \phi(u,v) = 0$ (conservação de fluxo)

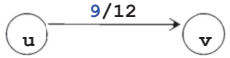
Rede Residual

- Sejam:
  - Seja R = (V,A') uma rede de fluxos
  - c(u,v) a capacidade do arco (u,v) ∈ A'
  - ϕ um fluxo na rede de fluxos R
- A **rede residual** de R induzida por ϕ é um grafo orientado R_ϕ=(V,A''), tal que:

  A'' = { (u,v) ∈ A' | c(u,v) − ϕ(u,v) > 0 }

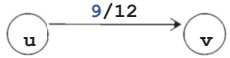
Rede Residual

- Grafo original G = (V,A)
  - c(u,v) = 12
  - ϕ(u,v) = 9

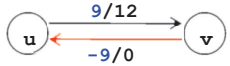


Rede Residual

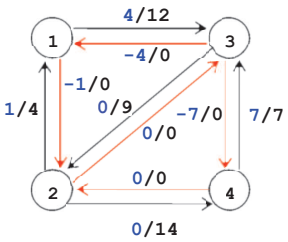
- Grafo original G = (V,A)
  - c(u,v) = 12
  - ϕ(u,v) = 9



- Rede Residual R_ϕ = (V,A'')
  - c(v,u) = 0
  - ϕ(v,u) = -9

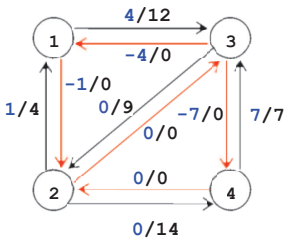


Rede Residual

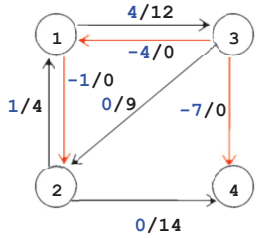


A''= { (u,v) ∈ A' | c(u,v) − ϕ(u,v) > 0 }

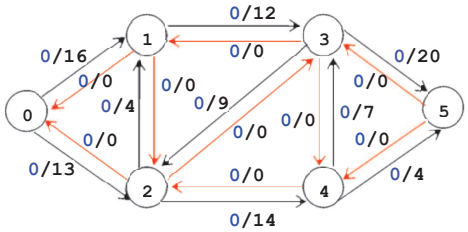
Rede Residual



A''= { (u,v) ∈ A' | c(u,v) − ϕ(u,v) > 0 }

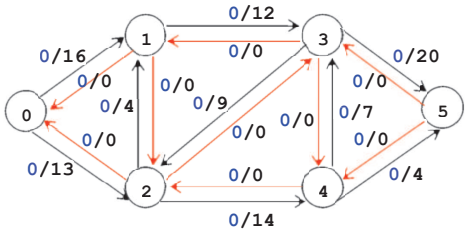


Método de Ford-Fulkerson

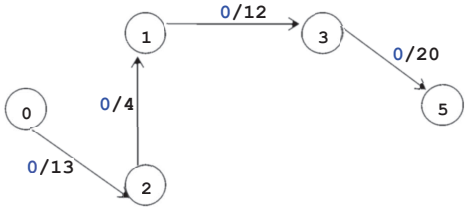


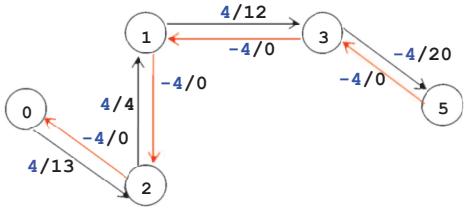
- Descobrir um caminho na rede residual
- Determinar o incremento máximo
- Actualizar fluxo

Método de Ford-Fulkerson

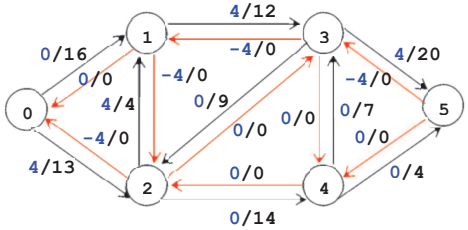


- Descobrir um caminho na rede residual
  - 0, 2, 1, 3, 5
- Determinar o incremento máximo
  - 4
- Actualizar fluxo

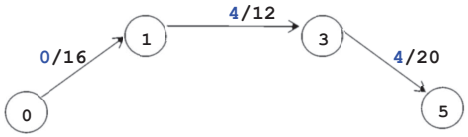


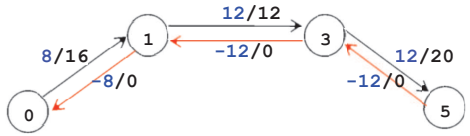


Método de Ford-Fulkerson

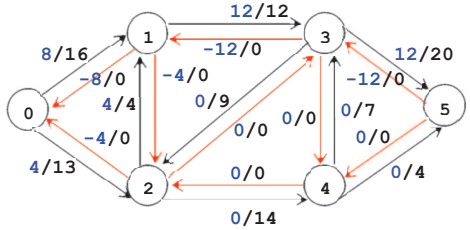


- Descobrir um caminho na rede residual
  - 0, 1, 3, 5
- Determinar o incremento máximo
  - 8
- Actualizar fluxo

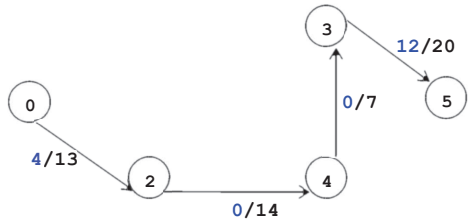


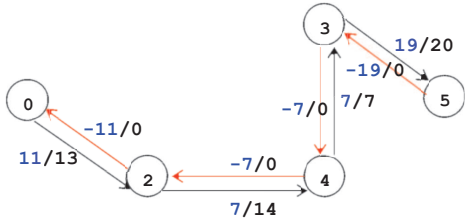


Método de Ford-Fulkerson

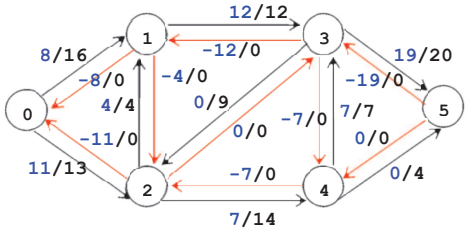


- Descobrir um caminho na rede residual
  - 0, 2, 4, 3, 5
- Determinar o incremento máximo
  - 7
- Actualizar fluxo

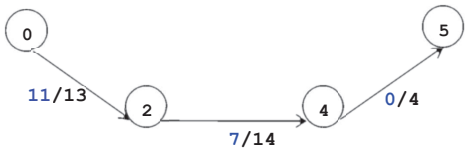


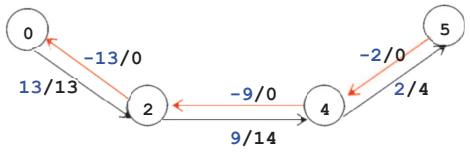


Método de Ford-Fulkerson

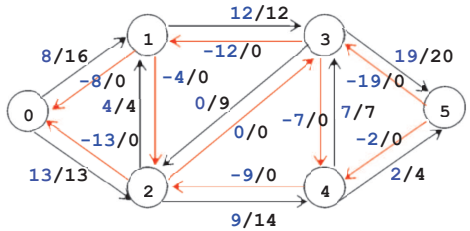


- Descobrir um caminho na rede residual
  - 0, 2, 4, 5
- Determinar o incremento máximo
  - 2
- Actualizar fluxo



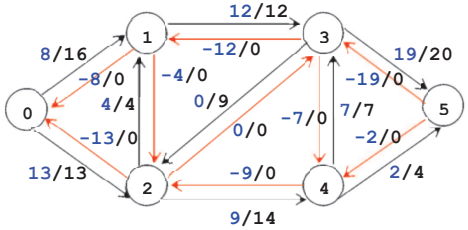


Método de Ford-Fulkerson

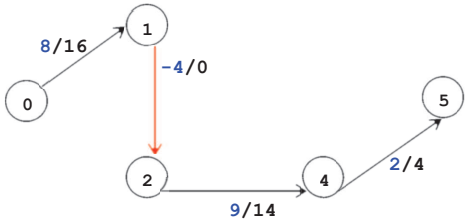


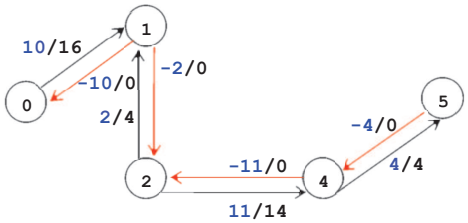
- Descobrir um caminho na rede residual
  - 0, 1, 2, 4, 5
- Determinar o incremento máximo
  - 2
- Actualizar fluxo

Método de Ford-Fulkerson

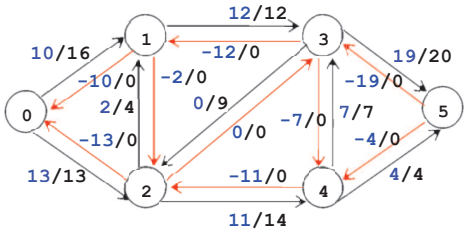


- Descobrir um caminho na rede residual
  - 0, 1, 2, 4, 5
- Determinar o incremento máximo
  - 2
- Actualizar fluxo

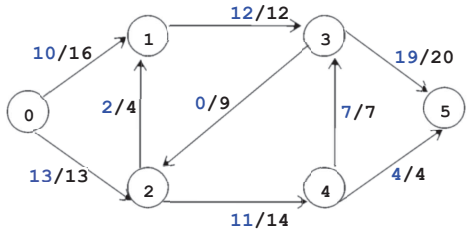




Método de Ford-Fulkerson



Método de Ford-Fulkerson



Método Ford-Fulkerson

- Não especifica como descobrir os caminhos da fonte para o dreno na rede residual
- Se os pesos forem números racionais, o método pode não convergir
  - Converter os valores racionais para valores inteiros
- Se os pesos forem inteiros, o número máximo de iterações não excede o valor dos fluxos máximos

Método Ford-Fulkerson

- Não especifica como descobrir os caminhos da fonte para o dreno na rede residual
- Se os pesos forem números racionais, o método pode não convergir
  - Converter os valores racionais para valores inteiros
- Se os pesos forem inteiros, o número máximo de iterações não excede o valor dos fluxos máximos
  - Em cada iteração o valor do fluxo é incrementado pelo menos uma unidade

Algoritmo de Edmonds-Karp

- Aplica o método Ford-Fulkerson
- Usa uma pesquisa em largura na rede residual para descobrir os caminhos da fonte para o dreno
  - Considera que cada arco tem custo (distância) um
    - Os caminhos têm comprimento mínimo

Construir a Rede de Fluxos

```
Digraph<?,E> buildNetwork( Digraph<?,E> graph ) {
   // The network has every vertex and every edge in the graph
   Digraph<?,E> network = graph.clone();
   // For every edge (v1, v2) in the graph (V,A), if (v2, v1) ∉ A,
   // insert edge (v2, v1) with capacity zero in the network
   for every Edge<?,E> e in graph.edges() {
        Vertex[] endPoints = e.endVertices();
        if ( !graph.edgeExists(endPoints[1], endPoints[0]) )
            network.insertEdge(endPoints[1], endPoints[0], 0);
   }
   return network;
}
```


Algoritmo de Edmonds-Karp

```
Pair<E, E[][]> edmondsKarp( Digraph<?,E> graph, Vertex source,
                            Vertex sink ) {
   Digraph<?,E> network = buildNetwork(graph);
   int numVert = network.numVertices();
   E[][] flow = new E[numVert][numVert];
   for every Edge<?,E> e in network.edges() {
       Vertex[] endPoints = e.endVertices();
       flow[ endPoints[0] ][ endPoints[1] ] = 0;
   }
   Vertex[] via = new Vertex[numVert];
   E flowValue = 0;
   E increment;
```

Algoritmo de Edmonds-Karp

```
while ( (increment = findPath(network,flow,source,sink,via)) != 0 ) {
      flowValue += increment;
      // Update flow
      Vertex vertex = sink;
      while ( vertex != source ) {
            Vertex origin = via[vertex];
            flow[origin][vertex] += increment;
            flow[vertex][origin] -= increment;
            vertex = origin;
      }
}
return new PairClass<E, E[][]>(flowValue, flow);
}
```


Algoritmo de Edmonds-Karp

```
E findPath( Digraph<?,E> network, E[][] flow, Vertex source,
            Vertex sink, Vertex[] via ) {
      int numVert = network.numVertices();
      Queue<Vertex> waiting = new QueueInArray<Vertex>(numVert - 1);
      boolean[] found = new boolean[numVert];
      for every Vertex v in network.vertices()
            found[v] = false;
      E[] pathIncr = new E[numVert];
      waiting.enqueue(source);
      found[source] = true;
      via[source] = source;
      pathIncr[source] = +∞;
```


Algoritmo de Edmonds-Karp

```
do {
      Vertex origin = waiting.dequeue();
      for every Edge<?,E> e in network.outIncidentEdges(origin) {
            Vertex destin = e.endVertices()[1];
            E residue = e.label() - flow[origin][destin];
            if ( !found[destin] && residue > 0 ) {
                  ...
            }
      }
} while ( !waiting.isEmpty() );
return 0;
}
```


Algoritmo de Edmonds-Karp

```
for every Edge<?,E> e in network.outIncidentEdges(origin) {
      Vertex destin = e.endVertices()[1];
      E residue = e.label() - flow[origin][destin];
      if ( !found[destin] && residue > 0 ) {
         via[destin] = origin;
         pathIncr[destin] = Math.min(pathIncr[origin], residue);
         if ( destin == sink )
               return pathIncr[destin];
         else {
               waiting.enqueue(destin);
               found[destin] = true;
         }
      }
}
```

Complexidade do Algoritmo de Edmonds-Karp
Grafo em Lista de Adjacências

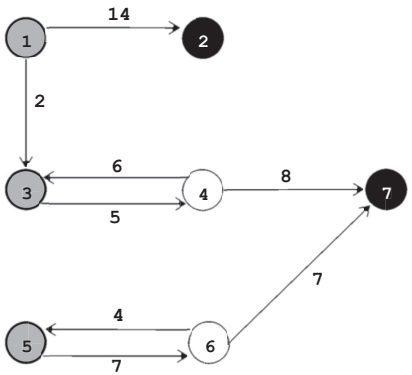
Construir rede de fluxos	O(#V × #A)
Inicializar fluxo	O(#A)
k × Descobrir caminho	O(k × #A)
k × Actualizar o fluxo	O(k × #V)
TOTAL	O((#V + k) × #A)

Complexidade do Algoritmo de Edmonds-Karp
Grafo em Lista de Adjacências

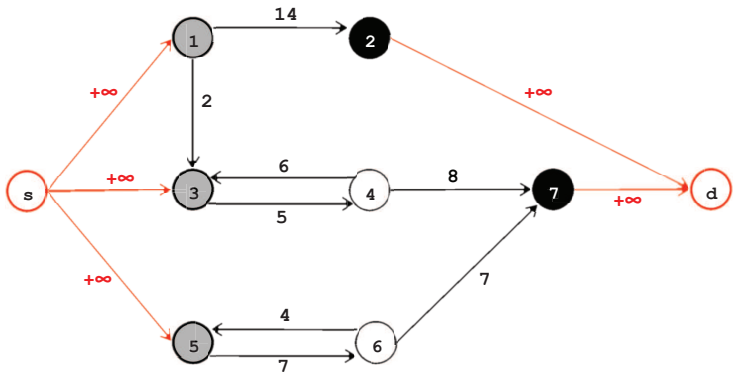
Construir rede de fluxos	O(#V × #A)
Inicializar fluxo	O(#A)
k × Descobrir caminho	O(k × #A)
k × Actualizar o fluxo	O(k × #V)
TOTAL	O((#V + k) × #A)

O número de iterações (k) do algoritmo não excede #V × #A
A complexidade do algoritmo não excede $O(\#V \times (\#A)^2)$

Multiplas Fontes ou Drenos



Multiplas Fontes ou Drenos



14. Introdução à Teoria da Complexidade

Planeamento

Apresentação da disciplina
Programação Dinâmica
Introdução aos Grafos
Percurso em largura e profundidade
Ordenação Topológica
Árvore Mínima de Cobertura (Algoritmo de Kruskal)
TAD Partição
Complexidade Amortizada
Árvore Mínima de Cobertura (Algoritmo de Prim)
TAD Fila com Prioridade Adaptável
Algoritmo de Dijskstra
Filas Binomiais
Filas de Fibonacci
Algoritmo de Bellman-Ford
Fluxo Máximo: Método de Ford-Fulkerson
Algoritmo de Edmonds-Karp
Introdução à Teoria da Complexidade
Exemplos de Problemas NP
Redutibilidade entre Problemas de Decisão

Complexidade: Ordens de Grandeza

Complexidade Polinomial

1	(constante)
log(n)	(logarítmica)
n	(linear)
n×log(n)	
n^2	(quadrática)
n^3	(cúbica)
n^4	
...	

Complexidade Exponencial

2^n

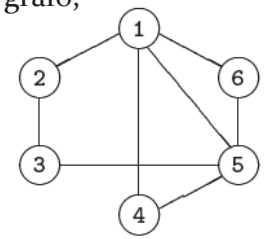
3^n

n!

...

Circuito de Euler

- Dado um grafo não orientado e conexo, um Circuito de Euler é um circuito que passa uma só vez por cada **arco** do grafo,



1, 6, 5, 4, 1, 2, 3, 5, 1

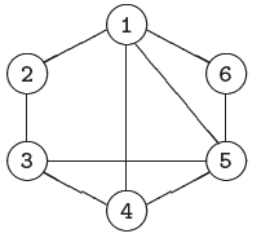
- Dado um grafo G não orientado e conexo, é possível determinar que G tem um Circuito de Euler? Em caso afirmativo, é possível construir esse circuito?

Circuito de Euler

- Existe um algoritmo polinomial para determinar se um grafo tem um circuito de Euler:
  - Um grafo não orientado e conexo tem um circuito de Euler sse todos os seus vértices têm um número par de sucessores
- Existe um algoritmo polinomial para encontrar um um circuito de Euler num grafo não orientado e conexo cujos vértices têm um número par de sucessores

Circuito de Hamilton

- Dado um grafo não orientado e conexo, um circuito de Hamilton é um circuito simples que passa uma só vez por cada **vértice**.



`1, 2, 3, 4, 5, 6, 1`

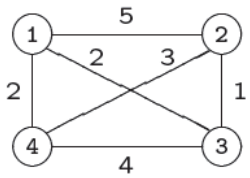
- Dado um grafo G não orientado e conexo, é possível determinar que G tem um circuito de Hamilton? Em caso afirmativo, é possível construir esse circuito?

Facto

- Embora os problemas do circuito de Euler e do circuito de Hamilton tenham enunciados similares, as soluções computacionais têm ordens de complexidade muito diferentes
  - Existem algoritmos polinomiais para resolver os problemas do Circuito de Euler
  - Todos os algoritmos que se conhecem para resolver os problemas do Circuito de Hamilton são exponenciais

Problema do Caixeiro Viajante

- Dado um grafo não orientado, pesado e completo, existe um Circuito de Hamilton cujo comprimento pesado não excede k?



Para k = 8, temos o circuito:

`1, 3, 2, 4`

Complexidade Exponencial

- Algoritmo O((#V)!) para resolver o problema do Caixeiro Viajante

Quantidade	Número
PC instruções / segundo	10^9
Super-computador instruções / segundo	10^{12}
Segundos por ano	10^9
Idade do universo (estimado)	10^{13}
Electrões no universo (estimado)	10^{79}

Complexidade Exponencial

- Algoritmo O((#V)!) para resolver o problema do Caixeiro Viajante

Quantidade	Número
PC instruções / segundo	10^9
Super-computador instruções / segundo	10^{12}
Segundos por ano	10^9
Idade do universo (estimado)	10^{13}
Electrões no universo (estimado)	10^{79}

- Para um grafo com 1000 vértices, não é possível determinar a solução

$1000! \gg 10^{1000} \gg 10^{79} \times 10^{13} \times 10^9 \times 10^{12}$

Propriedades dos Problemas

- Como determinar se um problema tem uma solução prática?
  - Usar resultados da Teoria da Complexidade

Existe Algoritmo Polinomial	Sem Solução Polinomial Conhecida
Caminho mais curto	Caminho mais longo
Circuito de Euler	Circuito de Hamilton
2-SAT	3-SAT
...	...

Classes de Problemas

Classe de Problemas **P**

Conjunto de todos os problemas resolúveis em tempo polinomial

Classe de Problemas **EXP**

Conjunto de todos os problemas resolúveis em tempo exponencial

Problemas Tratáveis/Intratáveis

Problemas **Tratáveis**

Conjunto de todos os problemas resolúveis em tempo polinomial

Problemas **Intratáveis**

Conjunto de todos os problemas **não** resolúveis em tempo polinomial

Inclui a classe de problemas com algoritmos exponenciais ou com complexidade pior que exponencial

Decidibilidade

Algoritmo

Um algoritmo para resolver um problema é um procedimento que calcula a solução de qualquer instância num número finito de passos

Decidibilidade

Um problema de decisão diz-se

- Decidível, se existir um algoritmo para o resolver
- Indecidível, caso contrário

Classes de Problemas

Classe de Problemas **P**

Conjunto de todos os problemas resolúveis em tempo polinomial

Classe de Problemas **EXP**

Conjunto de todos os problemas resolúveis em tempo exponencial

Classe de Problemas **UNDECIDABLE**

Conjunto de todos os problemas indecidíveis

Problema da Terminação

TERM: Dados um programa P e uma entrada E, P termina com E?

Problema da Terminação é indecidível [Turing 1936]

é impossível definir um algoritmo que, dados um programa arbitrário e uma entrada arbitrária, decida se o programa termina com essa entrada

Indecidibilidade do Problema da Terminação

- Se existisse o algoritmo `termina`, seria possível construir o programa `A`

```
// Indica se p termina com a entrada e
// e é uma sequência de bits qualquer
boolean termina( Programa p, Entrada e );

boolean A( Programa p ) {
   if termina(p, p) then
        while true do { }
   else return true;
}
```


Indecidibilidade do Problema da Terminação

- Se existisse o algoritmo `termina`, seria possível construir o programa `A`

```
// Indica se p termina com a entrada e
// e é uma sequência de bits qualquer
boolean termina( Programa p, Entrada e );

boolean A( Programa p ) {
   if termina(p, p) then
        while true do { }
   else return true;
}
```

- **O programa `A` termina com a entrada `A`?**
  - Se `A` termina com a entrada `A`, `termina(A, A)` é verdade, logo `A` não termina com a entrada `A`

Indecidibilidade do Problema da Terminação

- Se existisse o algoritmo `termina`, seria possível construir o programa `A`

```
// Indica se p termina com a entrada e
// e é uma sequência de bits qualquer
boolean termina( Programa p, Entrada e );

boolean A( Programa p ) {
   if termina(p, p) then
        while true do { }
   else return true;
}
```

- **O programa `A` termina com a entrada `A`?**
  - Se `A` termina com a entrada `A`, `termina(A, A)` é verdade, logo `A` não termina com a entrada `A`
  - Se `A` não termina com a entrada `A`, `termina(A, A)` é falso, logo `A` termina com a entrada `A`

Cálculo de Predicados de Primeira Ordem

PRED: Dada uma fórmula f do Cálculo de Predicados de Primeira Ordem, f é válida?

Instâncias:

$$(\forall x(Px \Rightarrow Qx)) \Rightarrow ((\exists x Px) \Rightarrow (\exists x Qx))$$

$$\exists x(Px \vee Qx) \Rightarrow \exists x(Px \wedge Qx)$$

Este problema é indecidível [Church 1936, Turing 1936/7]

Décimo Problema de Hilbert
(da lista de 23 problemas publicados por Hilbert em 1900)

HILBERT: Dada uma equação polinomial de coeficientes inteiros, $P(x_1,x_2, \ldots,x_n) = 0$, esta equação tem uma solução inteira?

Instâncias:

$$x^2 - 3x + 2 = 0$$
$$7x^5y^2 - 4x^2y^3 + 14y^2 + 8y - 3 = 0$$

Este problema é indecidível [Matijacevic 1970]

Classes de Problemas

Classe de Problemas **P**

Conjunto de todos os problemas resolúveis em tempo polinomial

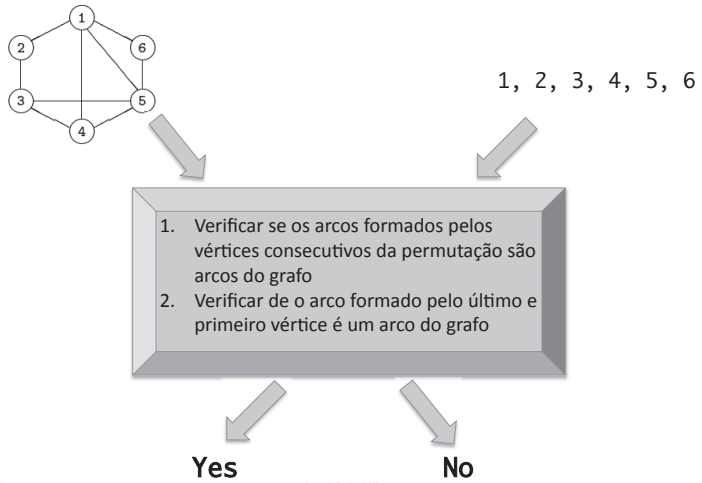
Classe de Problemas **NP**

Conjunto de todos os problemas verificáveis em tempo polinomial

Classe de Problemas **EXP**

Conjunto de todos os problemas resolúveis em tempo exponencial

Algoritmo de Verificação Problema do Circuito de Hamilton



Porquê Estudar Problemas NP?

- Ao reconhecer um problema como **NP**, existem 3 soluções:
  - Implementar um algoritmo com complexidade exponencial
  - Restringir e limitar o problema de forma a encontrar uma solução polinomial
  - Usar algoritmos de aproximação

Porquê Estudar Problemas **NP**?

- Ao reconhecer um problema como **NP**, existem 3 soluções:
  - Implementar um algoritmo com complexidade exponencial
  - Restringir e limitar o problema de forma a encontrar uma solução polinomial
  - Usar algoritmos de aproximação
- **P** vs **NP**
  - Millennium Prize Problems (www.claymath.org/millennium/)

Classes de Problemas

Classe de Problemas **P**

Conjunto de todos os problemas resolúveis em tempo polinomial

Classe de Problemas **NP**

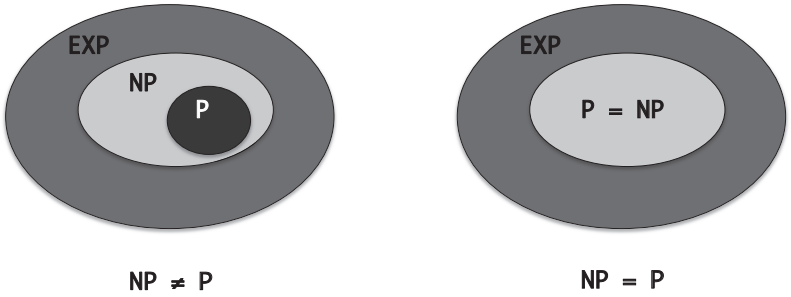
Conjunto de todos os problemas verificáveis em tempo polinomial

Classe de Problemas **EXP**

Conjunto de todos os problemas resolúveis em tempo exponencial

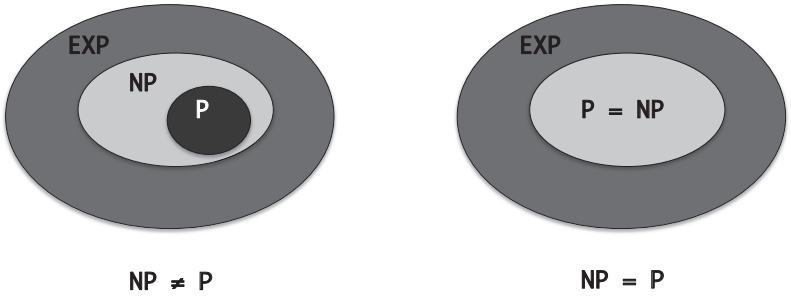
Hierarquia de Classes

- P = NP?



Hierarquia de Classes

- P = NP?



O consenso é que P ≠ NP

Problemas de Decisão

Um problema de decisão é um problema cujas instâncias têm a solução "SIM" ou "NÃO".

Problemas **NP**

NP é classe de problemas de decisão *X* para os quais existe:

- Um algoritmo polinomial ***A*** (que verifica a prova do sim)
- Uma função ***P*** (que fornece a prova do sim)
- Uma constante ***k***

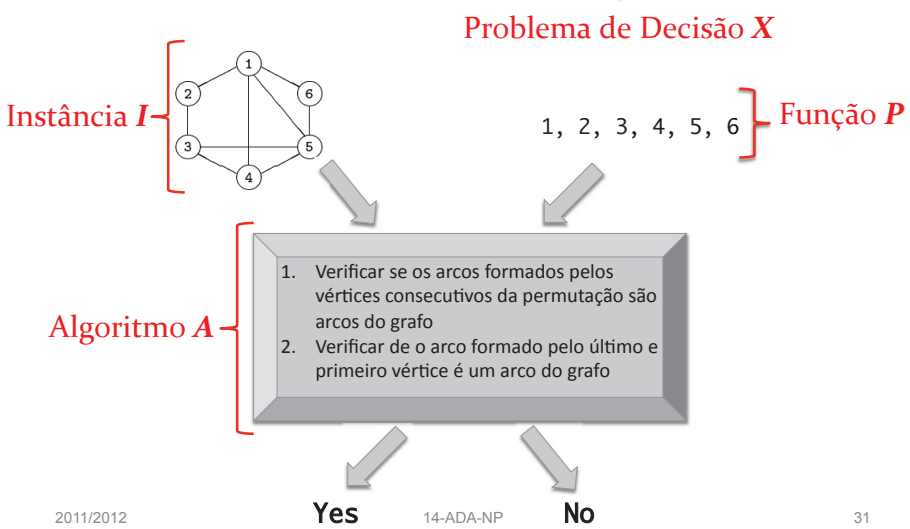
tais que, para qualquer instância *I* de *X*:

- Se a solução de *I* for "SIM" e a dimensão de *I* for *n*,

  então a dimensão de ***P(I)*** é $O(n^k)$ e ***A***(*I*,***P***(*I*)) retorna "SIM"
- Se a solução de *I* for "NÃO",

  então não existe nenhum valor para ***P(I)*** que faça ***A***(*I*,***P***(*I*)) retorna "SIM"

Problema do Circuito de Hamilton

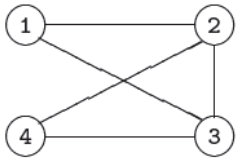


Circuito de Hamilton

Seja G um grafo não orientado e conexo, um Circuito de Hamilton é um circuito simples que passa por todos os vértices de G

HAMILTON: Dado um grafo G, não orientado e conexo, G tem um Circuito de Hamiton?

Instância:



Prova do Sim: 1 2 4 3 (permutação de vértices que caracteriza o circuito)

HAMILTON é um problema NP

1. Existe um algoritmo polinomial que, dados:
   - um grafo G = (V, A), não orientado e conexo
   - uma permutação $\rho = v_1 v_2 \ldots v_n$ dos vértices de G

   verifica se $\rho = v_1 v_2 \ldots v_n v_1$ é um Circuito de Hamilton em G

   Descrição do algoritmo

   O algoritmo percorre a permutação testando se os arcos (v_i, v_{i+1}) pertencem a A, para todo i = 1, ..., n. Depois, testa se o arco (v_n, v_1) também pertence a A. O algoritmo retorna **true** sse todos os testes tiverem sucesso.

HAMILTON é um problema NP

Análise da complexidade do algoritmo

A complexidade temporal do algoritmo é polinomial no número de vértices (#V) e no número de arcos (#A). Se o conjunto de arcos estiver num vector, o número de passos é $O(n\ \#A) = O(\#V\#A)$.

2. A dimensão da permutação ρ é polinomial na dimensão de G, porque ρ tem #V vértices

Satisfazibilidade

SAT: Dada uma fórmula proposicional na forma conjuntiva

$$f = \bigwedge_{1 \le i \le k} C_i$$

f é satisfazível?

Instâncias:

$(x \vee y \vee \neg z) \wedge (\neg x \vee \neg y) \wedge z$

$(x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)$

$(x \vee y) \wedge (\neg x) \wedge (\neg y)$

Satisfazibilidade

SAT: Dada uma fórmula proposicional na forma conjuntiva

$$f = \bigwedge_{1 \le i \le k} C_i$$

f é satisfazível?

Instâncias:

$(x \vee y \vee \neg z) \wedge (\neg x \vee \neg y) \wedge z$

$(x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)$

$(x \vee y) \wedge (\neg x) \wedge (\neg y)$

Prova do Sim: Um atribuição de valores às variáveis que satisfaz a formula

SAT é um problema NP

1. Existe um algoritmo polinomial que, dados:
   - uma fórmula proposicional $f = \wedge_{1 \le i \le k} C_i$ na forma normal conjuntiva
   - uma atribuição $\theta = \{(v_1, b_1), (v_2, b_2), \ldots (v_n, b_n)\}$ de valores de verdade às variáveis de f

   verifica se θ satisfaz f

   Descrição do algoritmo

   O algoritmo avalia a fórmula, calculando o valor lógico de cada uns dos termos C_i, para i=1,...,k. Para avaliar $C_i = \vee_{1 \le j \le m_i} D_{ij}$, percorre os literais D_{ij}, para j=1,...,m_i. Se o literal for uma variável v_u, o seu valor lógico é b_u. Se o literal tiver a forma $\neg v_u$, seu valor lógico é o complementar de b_u. Os valores b_u são obtidos por pesquisa em θ.

SAT é um problema NP

Descrição do algoritmo (cont.)

A avaliação do termo C_i é verdade se o valor lógico de algum dos literais D_{ij} for verdade. O algoritmo retorna true sse a avaliação de todos os termos C_i for verdade.

Análise da complexidade do algoritmo

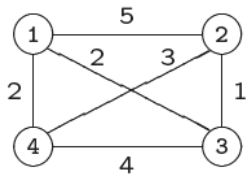
A complexidade temporal do algoritmo é polinomial no número de símbolos de *f*. Se a atribuição estiver guardado num vector e *f* tiver n variáveis distintas, o_v ocorrências dessas variáveis e o_v ocorrências de operadores lógicos, o número de passos do algoritmo é $O(no_v+o_p)$.

2. A dimensão da atribuição θ é polinomial na dimensão de *f*, porque θ tem 2n elementos, *f* tem o_v+o_p elementos e $n \le o_v$

Caixeiro Viajante

CAIXEIRO: Dado um grafo G, não orientado, pesado e completo, com arcos de custo positivo, e um inteiro $k \ge 1$, G tem um circuito de Hamiton cujo comprimento pesado não excede k?

Instância:

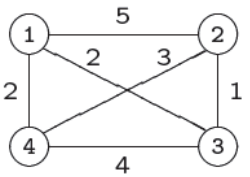


k = 8

Caixeiro Viajante

CAIXEIRO: Dado um grafo G, não orientado, pesado e completo, com arcos de custo positivo, e um inteiro $k \ge 1$, G tem um circuito de Hamiton cujo comprimento pesado não excede k?

Instância:



k = 8

Prova do Sim: 1 3 2 4 (permutação de vértices que caracteriza o circuito)

CAIXEIRO é um problema NP

1. Existe um algoritmo polinomial que, dados:
   - um grafo G = (V,A), não orientado, pesado e completo, com arcos de custo positivo
   - um inteiro $k \geq 1$
   - uma permutação $\rho = v_1 v_2 ... v_n$ dos vértices de G

   verifica se $\rho = v_1 v_2 ... v_n v_1$ é um Circuito de Hamilton cujo comprimento pesado não excede k

   Descrição do algoritmo

   O algoritmo percorre a permutação ρ, calculando a soma dos pesos dos arcos (v_i, v_{i+1}), para todo i = 1, ..., n. Depois, ao resultado soma o peso do arco (v_n, v_1). O algoritmo retorna **true** sse o resultado final for inferior ou igual a k.

CAIXEIRO é um problema NP

Análise da complexidade do algoritmo

A complexidade temporal do algoritmo é polinomial no número de vértices (#V). Como o grafo é não orientado e completo,

$$\#A = \frac{\#V(\#V-1)}{2}$$

Se o conjunto dos arcos estiver guardado num vector , o número de passos do algoritmo é:

$$O(n\frac{\#V(\#V-1)}{2}) = O(\#V\frac{\#V(\#V-1)}{2}) = O(\#V^3)$$

2. A dimensão da permutação ρ é polinomial na dimensão de (G,k), porque ρ tem #V vértices

Conjunto de Ataque

Sejam D um conjunto finito, *C* um conjunto de subconjuntos de D e $k \geq 1$. Um conjunto de ataque de *C* de dimensão k é um conjunto $A \subseteq D$ tal que: #A = k e $(\forall X \in C) X \cap A \neq \emptyset$

ATAQUE: Dados um conjunto finito D, um conjunto *C* de subconjuntos de D e um inteiro $k \geq 1$, *C* tem um conjunto de ataque de dimensão k?

Instância:

({1, 2, 3, 4, 5, 6, 7, 8}, {{1, 2, 3}, {4, 5, 6}, {2, 3, 5, 7}}, 2)

Conjunto de Ataque

Sejam D um conjunto finito, *C* um conjunto de subconjuntos de D e $k \geq 1$. Um conjunto de ataque de *C* de dimensão k é um conjunto $A \subseteq D$ tal que: #A = k e $(\forall X \in C) X \cap A \neq \emptyset$

ATAQUE: Dados um conjunto finito D, um conjunto *C* de subconjuntos de D e um inteiro $k \geq 1$, *C* tem um conjunto de ataque de dimensão k?

Instância:

({1, 2, 3, 4, 5, 6, 7, 8}, {{1, 2, 3}, {4, 5, 6}, {2, 3, 5, 7}}, 2)

Prova do Sim: {1, 5}

ATAQUE é um problema NP

1. Existe um algoritmo polinomial que, dados:
   - um conjunto finito D
   - um conjunto *C* de subconjuntos de D
   - um inteiro $k \geq 1$
   - um subconjunto A de D

   verifica se A é um conjunto de ataque de *C* de dimensão k

   Descrição do algoritmo

   O algoritmo começa por contar o número de elementos de A, retornando false se esse número diferente de k. Caso contrário, o algoritmo percorre o conjunto *C* e, para cada elemento X de *C*, testa se a intersecção de X com A não é vazia. O algoritmo retorna true sse todos os testes tiverem sucesso.

ATAQUE é um problema NP

Descrição do algoritmo (cont.)

Para determinar se a intersecção de dois conjuntos não é vazia, é necessário percorrer um dos conjuntos, verificando se algum dos seus elementos pertence ao outro conjunto.

Análise da complexidade do algoritmo

A complexidade temporal do algoritmo é polinomial no número de elementos de D (#D), de *C* (#C) e de A (#A). A contagem do número de elementos de A é O(#A). Cada teste de intersecção é O(#A#D), porque os elementos de *C* são subconjuntos de D. No máximo são realizados #C testes de intersecção. Logo, o número de passos do algoritmo é

$$O(\#A + \#C\ \#A\ \#D) = O(\#C\ \#A\ \#D)$$

ATAQUE é um problema NP

2. A dimensão conjunto A é polinomial na dimensão de (D,*C*,k), porque A é um subconjunto de D.

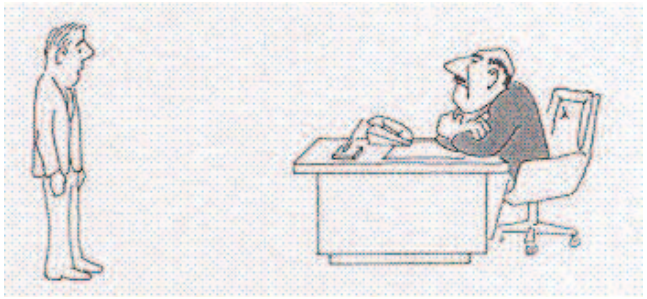
Provar Problemas NP-Completos

- Seja X um problema de decisão tal que $Y \leq_P X$, em que $Y \in NPC$. Se $X \in NP$, então $X \in NPC$
  - $Y \in NPC$
    - $Z \leq_P Y$ para qualquer $Z \in NP$
  - Como $\leq_P$ é transitiva e que $Y \leq_P X$, obtemos
    - $Z \leq_P X$ para qualquer $Z \in NP$
  - Deste modo
    - $X \in NP$
    - $Z \leq_P X$ para qualquer $Z \in NP$
  - Podemos concluir que $X \in NPC$

Provar Problemas NP-Completos

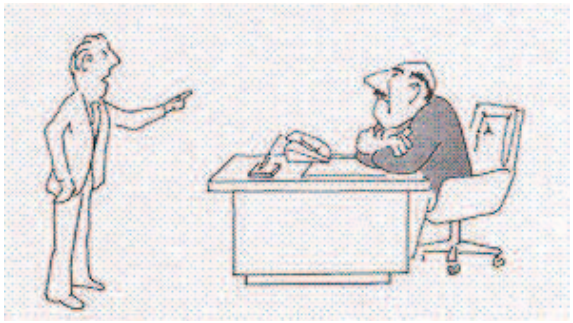
- Abordagem para provar que $X \in NPC$
  - Provar que $X \in NP$
  - Escolher $Y \in NPC$
  - Definir um redução polinomial ϕ de Y para X, ou seja, $Y \leq_p X$

Importância de Redutibilidade



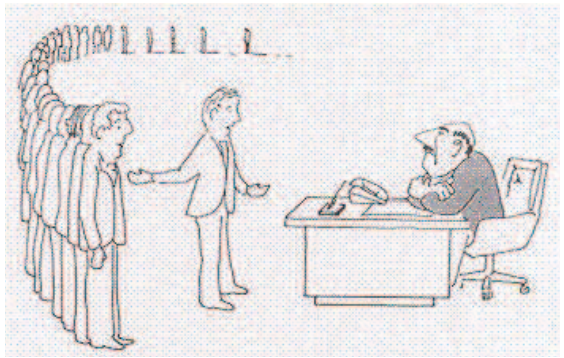
"I can't find an efficient algorithm, I guess I'm just too dumb."

Importância de Redutibilidade



"I can't find an efficient algorithm, because no such algorithm is possible."

Importância de Redutibilidade



"I can't find an efficient algorithm, but neither can all these famous people."

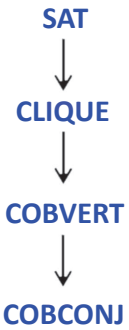
15. Redutibilidade de Problemas NP-Completos

Planeamento

Apresentação da disciplina
Programação Dinâmica
Introdução aos Grafos
Percurso em largura e profundidade
Ordenação Topológica
Árvore Mínima de Cobertura (Algoritmo de Kruskal)
TAD Partição
Complexidade Amortizada
Árvore Mínima de Cobertura (Algoritmo de Prim)
TAD Fila com Prioridade Adaptável
Algoritmo de Dijskstra
Filas Binomiais
Filas de Fibonacci
Algoritmo de Bellman-Ford
Fluxo Máximo: Método de Ford-Fulkerson
Algoritmo de Edmonds-Karp
Introdução à Teoria da Complexidade
Exemplos de Problemas NP
Redutibilidade entre Problemas de Decisão

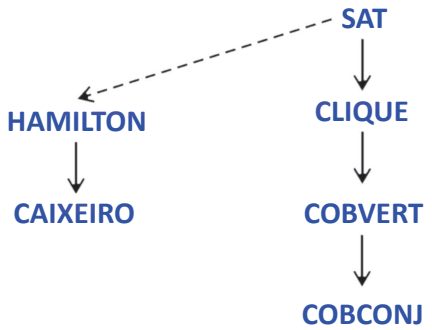
Reduções

Steve Cook



Reduções

Steve Cook



HAMILTON → CAIXEIRO

HAMILTON: Dado um grafo G, não orientado e conexo, G tem um circuito de Hamilton, ou seja, um circuito simples que passa por todos os vértices de G?

CAIXEIRO: Dado um grafo G, não orientado, pesado e completo, com arcos de custo positivo, e um inteiro $k \geq 1$, G tem um circuito de Hamilton cujo comprimento pesado não excede k?

$$\begin{array}{lll} \mathit{HAMILTON} & \rightarrow & \mathit{CAIXEIRO} \\ (V,A) & \mapsto & \end{array}$$

HAMILTON → CAIXEIRO

HAMILTON: Dado um grafo G, não orientado e conexo, G tem um circuito de Hamilton, ou seja, um circuito simples que passa por todos os vértices de G?

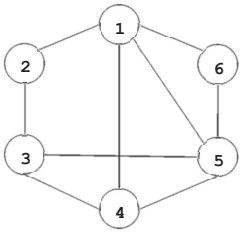
CAIXEIRO: Dado um grafo G, não orientado, pesado e completo, com arcos de custo positivo, e um inteiro $k \geq 1$, G tem um circuito de Hamilton cujo comprimento pesado não excede k?

$$\begin{array}{lll} \mathit{HAMILTON} & \rightarrow & \mathit{CAIXEIRO} \\ (V,A) & \mapsto & \end{array}$$

HAMILTON → CAIXEIRO

HAMILTON: Dado um grafo G, não orientado e conexo, G tem um circuito de Hamilton, ou seja, um circuito simples que passa por todos os vértices de G?

CAIXEIRO: Dado um grafo G, não orientado, pesado e completo, com arcos de custo positivo, e um inteiro $k \geq 1$, G tem um circuito de Hamilton cujo comprimento pesado não excede k?



HAMILTON → CAIXEIRO

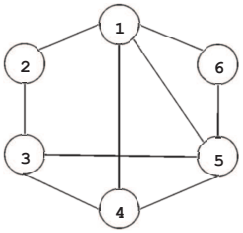
HAMILTON: Dado um grafo G, não orientado e conexo, G tem um circuito de Hamilton, ou seja, um circuito simples que passa por todos os vértices de G?

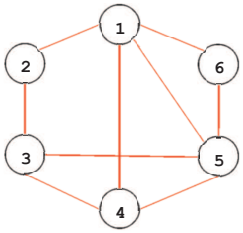
CAIXEIRO: Dado um grafo G, não orientado, pesado e completo, com arcos de custo positivo, e um inteiro $k \geq 1$, G tem um circuito de Hamilton cujo comprimento pesado não excede k?

HAMILTON → CAIXEIRO

HAMILTON: Dado um grafo G, não orientado e conexo, G tem um circuito de Hamilton, ou seja, um circuito simples que passa por todos os vértices de G?

CAIXEIRO: Dado um grafo G, não orientado, pesado e completo, com arcos de custo positivo, e um inteiro k ≥ 1, G tem um circuito de Hamilton cujo comprimento pesado não excede k?



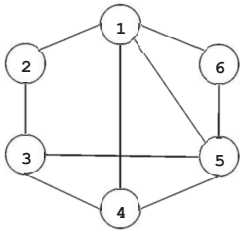


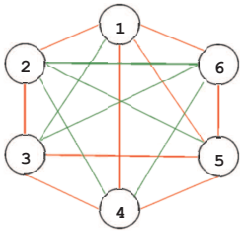
k = ?

HAMILTON → CAIXEIRO

HAMILTON: Dado um grafo G, não orientado e conexo, G tem um circuito de Hamilton, ou seja, um circuito simples que passa por todos os vértices de G?

CAIXEIRO: Dado um grafo G, não orientado, pesado e completo, com arcos de custo positivo, e um inteiro k ≥ 1, G tem um circuito de Hamilton cujo comprimento pesado não excede k?



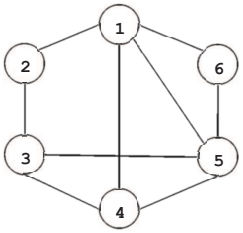


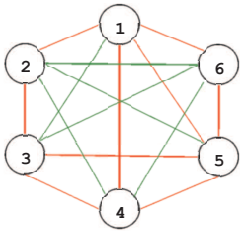
k = ?

HAMILTON → CAIXEIRO

HAMILTON: Dado um grafo G, não orientado e conexo, G tem um circuito de Hamilton, ou seja, um circuito simples que passa por todos os vértices de G?

CAIXEIRO: Dado um grafo G, não orientado, pesado e completo, com arcos de custo positivo, e um inteiro k ≥ 1, G tem um circuito de Hamilton cujo comprimento pesado não excede k?





2

1

k = ?

HAMILTON → CAIXEIRO

HAMILTON: Dado um grafo G, não orientado e conexo, G tem um circuito de Hamilton, ou seja, um circuito simples que passa por todos os vértices de G?

CAIXEIRO: Dado um grafo G, não orientado, pesado e completo, com arcos de custo positivo, e um inteiro k ≥ 1, G tem um circuito de Hamilton cujo comprimento pesado não excede k?

$$HAMILTON \rightarrow CAIXEIRO$$

$$(V,A) \mapsto ((V,A'),\#V)$$

$$A' = \{(a,b,1) \mid (a,b) \in A\} \cup \{(a,b,2) \mid (a,b) \notin A\}$$

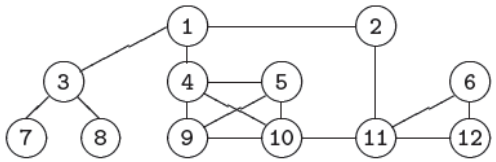
Clique

Seja G = (V,A) um grafo não orientado e n ≥ 1. Uma clique de G de dimensão n é um conjunto V′ ⊆ V tal que:

$$\#V' = n \text{ e } (\forall a,b \in V')\, a \neq b \Rightarrow (a,b) \in A$$

CLIQUE: Dados um grafo G, não orientado e um inteiro n ≥ 1, G tem uma clique de dimensão n?

Instância:



n = 4

SAT → CLIQUE

SAT: Dada uma fórmula proposicional na forma conjuntiva

$$f = \bigwedge_{1 \le i \le k} C_i$$

f é satisfazível?

CLIQUE: Dados um grafo G = (V,A) não orientado e um inteiro n ≥ 1, existe um conjunto V′ ⊆ V tal que:

$$\#V' = n \text{ e } (\forall a,b \in V')\, a \neq b \Rightarrow (a,b) \in A$$

$$SAT \rightarrow CLIQUE$$

$$\bigwedge_{1 \le i \le k} C_i \mapsto$$

SAT → CLIQUE

SAT: Dada uma fórmula proposicional na forma conjuntiva

$$f = \bigwedge_{1 \le i \le k} C_i$$

f é satisfazível?

CLIQUE: Dados um grafo G = (V,A) não orientado e um inteiro n ≥ 1, existe um conjunto V′ ⊆ V tal que:

$$\#V' = n \text{ e } (\forall a,b \in V')\, a \neq b \Rightarrow (a,b) \in A$$

$$SAT \rightarrow CLIQUE$$

$$\bigwedge_{1 \le i \le k} C_i \mapsto ((V,A),k)$$

com: $V=\{<\alpha,i> \mid \alpha \text{ é um literal de } C_i\}$

$A=\{(<\alpha,i>,<\beta,j>) \mid i \neq j \text{ e } \alpha \neq \neg\beta\}$

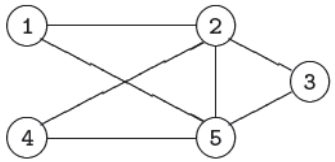
Cobertura de Vértices

Seja G = (V,A) um grafo não orientado e k ≥ 1. Uma cobertura de vértices de G de dimensão k é um conjunto V′ ⊆ V tal que:

$$\#V' = k \text{ e} (\forall (a,b) \in A) a \in V' \vee b \in V'$$

COBVERT: Dado um grafo G, não orientado e um inteiro k ≥ 1, G tem uma cobertura de vértices de dimensão k?

Instância:



k = 2

Prova do Sim: V′ = {2, 5}

CLIQUE → COBVERT

CLIQUE: Dados um grafo G = (V,A) não orientado e um inteiro n ≥ 1, existe um conjunto V′ ⊆ V tal que:

#V' = n e $(\forall a,b \in V')a \neq b \Rightarrow (a,b) \in A$

COBVERT: Dados um grafo G = (V,A) não orientado e um inteiro k ≥ 1, existe um conjunto V′ ⊆ V tal que:

#V' = k e $(\forall (a,b) \in A)a \in V' \vee b \in V'$

CLIQUE	→	*COBVERT*
$((V,A),n)$	↦	

CLIQUE → COBVERT

CLIQUE: Dados um grafo G = (V,A) não orientado e um inteiro n ≥ 1, existe um conjunto V′ ⊆ V tal que:

#V' = n e $(\forall a,b \in V')a \neq b \Rightarrow (a,b) \in A$

COBVERT: Dados um grafo G = (V,A) não orientado e um inteiro k ≥ 1, existe um conjunto V′ ⊆ V tal que:

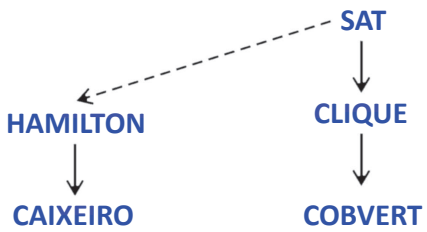
#V' = k e $(\forall (a,b) \in A)a \in V' \vee b \in V'$

CLIQUE	→	*COBVERT*
$((V,A),n)$	↦	$((V,A'),\#V - n)$

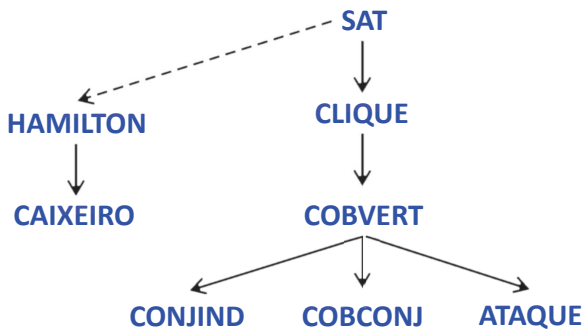
com:

A'={ (a,b) ∈ V×V | a ≠ b e (a,b) ∉ A }

Reduções



Reduções



Cobertura de Conjuntos

Sejam $X = < S_i \mid i \in I >$ uma sequência de conjuntos e $d \geq 1$. Uma cobertura de conjuntos de X de dimensão d é uma sequência $X' = < S_j \mid j \in J >$ tal que: $J \subseteq I$, $\#J = d$ e $\bigcup_{i \in I} S_i = \bigcup_{j \in J} S_j$

COBCONJ: Dados uma sequência X de conjuntos e um inteiro $d \geq 1$, X tem uma cobertura de conjuntos de dimensão k?

Instâncias:

(< {1, 2, 3, 5} , {2, 3, 4}, {1, 4, 5}, {2, 4, 6} >, 2)

(< {1, 2}, {2, 3}, {3, 4}, {4, 5} >, 3)

COBVERT → COBCONJ

COBVERT: Dados um grafo G = (V,A) não orientado e um inteiro $k \geq 1$, existe um conjunto $V' \subseteq V$ tal que:

$\#V' = k$ e $(\forall (a,b) \in A) a \in V' \vee b \in V'$

COBCONJ: Dados uma sequência de conjuntos $X = < S_i \mid i \in I >$ e um inteiro $d \geq 1$, existe uma sequência $X' = < S_j \mid j \in J >$ tal que:

$J \subseteq I$, $\#J = d$ e $\bigcup_{i \in I} S_i = \bigcup_{j \in J} S_j$

$COBVERT \rightarrow COBCONJ$

$((V,A),k) \mapsto$

COBVERT → COBCONJ

COBVERT: Dados um grafo G = (V,A) não orientado e um inteiro $k \geq 1$, existe um conjunto $V' \subseteq V$ tal que:

$\#V' = k$ e $(\forall (a,b) \in A) a \in V' \vee b \in V'$

COBCONJ: Dados uma sequência de conjuntos $X = < S_i \mid i \in I >$ e um inteiro $d \geq 1$, existe uma sequência $X' = < S_j \mid j \in J >$ tal que:

$J \subseteq I$, $\#J = d$ e $\bigcup_{i \in I} S_i = \bigcup_{j \in J} S_j$

$COBVERT \rightarrow COBCONJ$

$((V,A),k) \mapsto (< S_v \mid v \in V >, k)$

com:

$S_v = \{ (a,b) \in A \mid v = a \text{ ou } v = b \}$

Conjunto de Ataque

Sejam D um conjunto finito, *C* um conjunto de subconjuntos de D e $k \geq 1$. Um conjunto de ataque de *C* de dimensão k é um conjunto $A \subseteq D$ tal que: $\#A = k$ e $(\forall X \in C) X \cap A \neq \emptyset$

ATAQUE: Dados um conjunto finito D, um conjunto *C* de subconjuntos de D e um inteiro $k \geq 1$, *C* tem um conjunto de ataque de dimensão k?

Instância:

({1, 2, 3, 4, 5, 6, 7, 8}, {{1, 2, 3}, {4, 5, 6}, {2, 3, 5, 7}}, 2)

Conjunto de Ataque

Sejam D um conjunto finito, C um conjunto de subconjuntos de D e $k \geq 1$. Um conjunto de ataque de C de dimensão k é um conjunto $A \subseteq D$ tal que: $\#A = k$ e $(\forall X \in C) X \cap A \neq \emptyset$

ATAQUE: Dados um conjunto finito D, um conjunto C de subconjuntos de D e um inteiro $k \geq 1$, C tem um conjunto de ataque de dimensão k?

Instância:

({1, 2, 3, 4, 5, 6, 7, 8}, {{1, 2, 3}, {4, 5, 6}, {2, 3, 5, 7}}, 2)

Prova do Sim: {1, 5}

COBVERT → ATAQUE

COBVERT: Dados um grafo G = (V,A) não orientado e um inteiro $k \geq 1$, existe um conjunto $V' \subseteq V$ tal que:

$\#V' = k$ e $(\forall (a,b) \in A) a \in V' \vee b \in V'$

ATAQUE: Dados um conjunto finito D, um conjunto C de subconjuntos de D e um inteiro $n \geq 1$, existe um um conjunto $A \subseteq D$ tal que:

$\#A = n$ e $(\forall X \in C) X \cap A \neq \emptyset$

$$\begin{array}{ccc} COBVERT & \rightarrow & ATAQUE \\ ((V,A),k) & \mapsto & \end{array}$$

COBVERT → ATAQUE

COBVERT: Dados um grafo G = (V,A) não orientado e um inteiro $k \geq 1$, existe um conjunto $V' \subseteq V$ tal que:

$\#V' = k$ e $(\forall (a,b) \in A) a \in V' \vee b \in V'$

ATAQUE: Dados um conjunto finito D, um conjunto C de subconjuntos de D e um inteiro $n \geq 1$, existe um conjunto $A \subseteq D$ tal que:

$\#A = n$ e $(\forall X \in C) X \cap A \neq \emptyset$

$$\begin{array}{ccc} COBVERT & \rightarrow & ATAQUE \\ ((V,A),k) & \mapsto & (V,A',k) \end{array}$$

com:

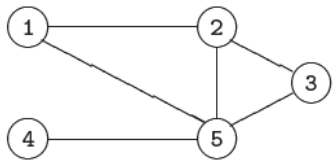
$A' = \{ \{a, b\} \mid (a,b) \in A \}$

Conjunto Independente

Sejam G = (V,A) um grafo não orientado e um inteiro $m \geq 1$. Um conjunto independente de G de dimensão m é um conjunto $I \subseteq V$ tal que: $\#I = m$ e $(\forall x,y \in I)(x,y) \notin A$

CONJIND: Dados um grafo G não orientado e um inteiro $m \geq 1$, G tem um conjunto independente de dimensão m?

Instância:



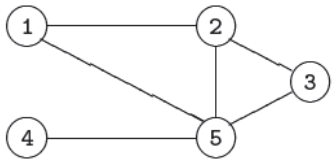
m = ?

Conjunto Independente

Sejam G = (V,A) um grafo não orientado e um inteiro m ≥ 1. Um conjunto independente de G de dimensão m é um conjunto I ⊆ V tal que: #I = m e $(\forall x,y \in I)(x,y) \notin A$

CONJIND: Dados um grafo G não orientado e um inteiro m ≥ 1, *G* tem um conjunto independente de dimensão m?

Instância:



m = 3

COBVERT → CONJIND

COBVERT: Dados um grafo G = (V,A) não orientado e um inteiro k ≥ 1, existe um conjunto V′ ⊆ V tal que:

#V' = k e $(\forall (a,b) \in A) a \in V' \vee b \in V'$

CONJIND: Dados um grafo G = (V, A) não orientado e um inteiro m ≥ 1, existe um conjunto I ⊆ V tal que:

#I = m e $(\forall x,y \in I)(x,y) \notin A$

$$\begin{array}{lcl} COBVERT & \rightarrow & CONJIND \\ ((V,A),k) & \mapsto & \end{array}$$

COBVERT → CONJIND

COBVERT: Dados um grafo G = (V,A) não orientado e um inteiro k ≥ 1, existe um conjunto V′ ⊆ V tal que:

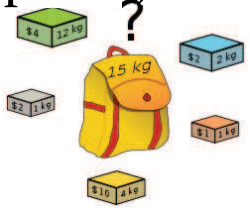
#V' = k e $(\forall (a,b) \in A) a \in V' \vee b \in V'$

CONJIND: Dados um grafo G = (V, A) não orientado e um inteiro m ≥ 1, existe um conjunto I ⊆ V tal que:

#I = m e $(\forall x,y \in I)(x,y) \notin A$

$$\begin{array}{lcl} COBVERT & \rightarrow & CONJIND \\ ((V,A),k) & \mapsto & ((V,A),\#V-k) \end{array}$$

Problema da Mochila (Knapsack)



- Dados N objectos {1, ..., N} e uma mochila
- Cada objecto tem um valor v_i e um peso w_i
- Peso total na mochila não pode exceder P
- Objectivo:
  - Maximizar o valor transportado pela mochila sem exceder o limite de peso
- Formalização:
  - Determinar o subconjunto A de {1, ...,N } que satisfaz a condição:

$$\max \sum_{i \in A} v_i \quad tal \quad que \quad \sum_{i \in A} w_i \leq P$$

Problema da Mochila (Knapsack)

- Tempo de execução O(N × P)
  - Preenchimento da tabela N × P
- Pseudo-Polinomial

Problema da Mochila (Knapsack)

- Tempo de execução O(N × P)
  - Preenchimento da tabela N × P
- Pseudo-Polinomial
  - Exponencial na dimensão da representação binária de P
  - **$O(N \times 2^X)$**
    - X = número de bits da representação binária de P

Problema da Mochila (Knapsack)

- Tempo de execução O(N × P)
  - Preenchimento da tabela N × P
- Pseudo-Polinomial
  - Exponencial na dimensão da representação binária de P
  - **$O(N \times 2^X)$**
    - X = número de bits da representação binária de P
- **KNAPSACK** é um problema NP-Completo

Algoritmos de Aproximação

Algoritmos de Aproximação

- Algoritmos, com complexidade polinomial, que calculam soluções aproximadas para problemas NP-completos

Algoritmos de Aproximação

- Algoritmos para problemas NP-completos
  - Algoritmos de tempo polinomial
  - Com garantias quanto ao máximo afastamento do valor calculado face à solução óptima

Definições

- Limite da razão p(n)
  - Algoritmo de aproximação para um problema de optimização
  - Para qualquer instância de tamanho n
    - Custo da solução calculada, C
    - Custo da solução óptima, C*
    - $\max(C/C^*, C^*/C) \leq p(n)$
      - maximização: $0 \leq C \leq C^*$
      - minimização: $0 \leq C^* \leq C$
      - $p(n) \geq 1$

Cobertura de Vértices

COBVERT: Dados um grafo G = (V,A) não orientado e um inteiro $k \geq 1$, existe um conjunto $V' \subseteq V$ tal que:

$$\#V' = k \quad e \quad (\forall (a,b) \in A) a \in V' \vee b \in V'$$

COBVERT$_{APROX}$: Calcular uma cobertura de vértices com o número mínimo de vértices

Cobertura de Vértices

- Algoritmo de aproximação:

```
COBVERT-Aprox(G)
    C = ∅
    A' = A[G]
    while A' ≠ ∅
        Seja (u, v) arco de A'
        C = C ∪ { u, v }
        Remover de A' qualquer arco incidente em u ou v
    return C
```

- Exemplo

Cobertura de Vértices

- Teorema:
  - $COBVERT_{APROX}$ é um algoritmo de aproximação com limite da razão 2 para o problema COBVERT
    - $\max(C/C^*, C^*/C) = C/C^* \leq 2$
- Prova:
  - E: conjunto de arcos seleccionados pelo algoritmo
  - Conjunto C é cobertura de vértices
    - Ciclo iterado até todos os arcos de E estarem cobertos
  - Por inspecção, nenhum par de arcos em E tem vértices comuns
  - Também por inspecção, $|C| = 2|E|$

Cobertura de Vértices

- Cobertura dos arcos em E requer pelo menos um vértice incidente em cada um dos arcos de E
  - Qualquer cobertura de vértices tem que cobrir arcos de E
    - Válido também para C*
    - $|E| \leq |C^*|$
  - Conclusão:
    - $|C| = 2|E| \leq 2|C^*|$

Problema da Mochila (Knapsack)

- Dados N objectos {1, ..., N} e uma mochila
- Cada objecto tem um valor v_i e um peso w_i
- Peso total na mochila não pode exceder P
- Objectivo:
  - Maximizar o valor transportado pela mochila sem exceder o limite de peso
- Formalização:
  - Determinar o subconjunto A de {1, ...,N } que satisfaz a condição:

$$\max \sum_{i \in A} v_i \quad tal \quad que \quad \sum_{i \in A} w_i \leq P$$

Problema da Mochila

- Algoritmo Greedy:

```
KNAP-Greedy(w,v,P)
    Ordenar w e v tal que v[i]/w[i] ≥ v[j]/w[j] para 1 ≤ i ≤ j ≤ n
    peso = 0; valor = 0
    for i = 1 to n
        if peso + w[i] ≤ P
            valor += v[i]
            peso += w[i]
    return valor
```

- Exemplo

Problema da Mochila

- Algoritmo de Aproximação:

```
KNAP-Aprox(w,v,P)
    maior = max {v[i] : 1 ≤ i ≤ n}  // Admite-se w[i] ≤ P, 1 ≤ i ≤ n
    return max {maior, KNAP-Greedy(w,v,P)}
```

- Exemplo

Problema da Mochila

- Teorema:
  - O algoritmo KNAP-Aprox é um algoritmo de aproximação com limite da razão 2 para o problema KNAPSACK
- Prova:
  - C*: solução óptima; C: solução retornada pelo algoritmo
  - Escolher menor l tal que, $\sum_{i=1}^{l} w_i \geq P$
  - KNAP-Greedy(w,v,P) verifica, $\text{KNAP-Greedy}(w,v,P) \geq \sum_{i=1}^{l-1} v_i$
  - E, maior ≥ v_l (por definição de maior)

Problema da Mochila

- Notando que max(x,y) ≥ (x+y)/2, obtemos:

$$\begin{aligned} C &= \max(\text{maior}, \text{KNAP-Greedy}(w,v,P)) \\ &\geq \left(\text{maior} + \text{KNAP-Greedy}(w,v,P)\right)/2 \\ &\geq \left(v_l + \sum_{i=1}^{l-1} v_i\right)/2 = \sum_{i=1}^{l} v_i/2 = C'/2 \\ &\geq C^*/2 \end{aligned}$$

- Definição de P ' e C':

$$P' = \sum_{i=1}^{l} w_i \qquad C' = \sum_{i=1}^{l} v_i$$

- Notar que P ' ≥ P e $v_1/w_1 \geq v_2/w_2 \geq \ldots \geq v_l/w_l$
  - Pelo que C' ≥ C*
    - C' é a solução óptima para P'

Caixeiro Viajante

CAIXEIRO: Dado um grafo G, não orientado, pesado e completo, com arcos de custo positivo, e um inteiro $k \geq 1$, G tem um circuito de Hamilton cujo comprimento pesado não excede k?

Adaptação:

- Encontrar o circuito de Hamilton de menor peso em G
- O custo dos arcos satisfaz a desigualdade triangular:
  - $w(t,v) \leq w(t,u) + w(u,v)$ para quaisquer arcos (t,v), (t,u) e (u,v) de G

Caixeiro Viajante

- Algoritmo de Aproximação:

```
CAIXEIRO-Approx(G,w)
    Seleccionar qualquer r ∈ V para vértice raiz
    Construir MST T a partir de r
    L = lista de vértices resultantes de visita de T em pré-ordem
    Definir circuito H que visita vértices de G pela ordem L
    return H
```

Visita de árvore que lista a raiz antes dos vértices de cada sub-árvore

- Exemplo

Caixeiro Viajante

- Teorema:
  - O algoritmo CAIXEIRO-Aprox é um algoritmo de aproximação com limite da razão 2 para instâncias do problema do CAIXEIRO que verificam a desigualdade triangular

Algoritmos de Aproximação

Cobertura de Conjuntos

Sejam $X = < S_i \mid i \in I >$ uma sequência de conjuntos e $d \geq 1$. Uma cobertura de conjuntos de X de dimensão d é uma sequência $X' = < S_j \mid j \in J >$ tal que: $J \subseteq I$, $\#J = d$ e $\bigcup_{i \in I} S_i = \bigcup_{j \in J} S_j$

COBCONJ: Dados uma sequência X de conjuntos e um inteiro $d \geq 1$, X tem uma cobertura de conjuntos de dimensão k?

Instâncias:

(< {1, 2, 3, 5} , {2, 3, 4}, {1, 4, 5}, {2, 4, 6} >, 2)

(< {1, 2}, {2, 3}, {3, 4}, {4, 5} >, 3)

Cobertura de Conjuntos

- Algoritmo de Aproximação:

```
COBCONJ-Aprox(X,F)
    U = X
    C = ∅
    while U ≠ ∅
        Seleccionar S ∈ F que maximiza |S ∩ U|
        U = U - S
        C = C ∪ { S }
    return C
```

- Exemplo

- Teorema
  - O algoritmo COBCONJ-Aprox tem limite da razão $H(\max\{|S|: S \in F\})$, com $H(d) = 1 + 1/2 + \ldots + 1/d$, $H(0) = 0$